Assignment 1

For due date see learn.bcit.ca

All work should be done individually.

Deliverables:

- The final modified `engine.py` which contains the implementation of your `MinimaxAI`, `IterativeDeepeningAI`, and `MCTSAI` bots. If you feel fitting the code required for all three bots does not feel right to be in one single python file, you can split it into multiple files and submit the splits along with your modified `engine.py`. Just make sure I can run each one through `engine.py` (probably using command line arguments!).
- A report (in `.pdf` format) that describes how you did the assignment and answers the questions mentioned in Assignment Description section. Don't forget to mention the problems you had in implementation of your code (if any). Also explain any missing components of your implementation as well as your thoughts on the problem(s).
- The zipped `mcts.model` file. Make sure the zip file size is under 2 MB.

Intelligent Chess Environment:

In this assignment we implement multiple AI experts in Chess. The implementation environment is already created. `engine.py` contains the `AIEngine` class which is used to create a `Game` object from `Chessnut` (https://github.com/cgearhart/Chessnut) library which does not provide any AI components or GUI but rather the dynamics of the chess game. `Chessnut` can import chess games in Forsyth-Edwards Notation (FEN), provide the valid moves in the current game state, and apply selected moves. `AIEngine` also instantiates `GenericAI` which is the implementation of a very naïve AI algorithm which makes a random move every time it wants to play.

`engine.py` has a text based `main` which is readily available. You can run the text based engine using `python engine.py` and start playing with the pre-implemented naïve bot. Later, when you create more intelligent bots you can simply replace `GenericAI` with your own implementation (e.g. `MinimaxAI`) and still run the script with the same procedure. You need to run `pip install Chessnut` in your python environment to install `Chessnut` library.

Not only you can play the text based engine, but also you can use the other script `web_ui.py` to run a `flask` web application and play with `AIEngine` bots in http://localhost:3636/ in your web browser. To run this script you need to run `pip install flask` in your python environment to install `flask` library. `web_ui.py` loads up `templates/index.html` besides the script. The html page itself loads up the chess piece images from `static/img/chesspieces/wikipedia/<piece_image_name>.png`.

The chess UI in `web_ui.py` uses chessboard-1.0.0.js (which also provides the chess piece images) and chess.js to provide the board and chess piece move functionality. You don't need to modify, change, or upload `web_ui.py` and any of its required files. **You don't even have to use it**. It's just there if you prefer to test your implementation using a graphical interface rather than a text based UI. If its not your preference, don't even run `web_ui.py`.

Assignment Description:

I hope you have played a bit with the naïve code and have won the bot a few times before getting started to read this part. Now that you have experience playing with the bot, you know that the bot does not make really good moves. Its not a wonder as you know the bot takes random move decisions. Your job is to use what you have learned so far specifically about *Adversarial Search* to implement a number of better bots.

- Get started by getting familiar with `GenericAI` class and the way its being used in `AIEngine`.
- As you may have noticed there is a `make_move` function in `GenericAI` which states where the random decision is being made and that's your starting point!
- Spot `MinimaxAI` class which is the empty skeleton of a class inheriting from `GenericAI`. You can see that `make_move` is already overriden and its ready for you to get started in re-writing the function in a way that it uses Minimax decision making.
- For each of the following bots, **in your report**:
  - Provide the pseudo-code of what you implemented.
  - Explain any problems that this implementation could have (theoretically as well as in your experience).
  - Run each bot against this one for a few times and report its win percentage (calculated using the content of `AIEngine.leaderboard`). Comment on the factor(s) responsible for its win/loss in the games and explain how do you compare this bot with the other bots you created?

(25 points) Implement `MinimaxAI.minimax` and use it in `MinimaxAI.make_move` so that your `MinimaxAI` bot uses Minimax decision making process with alpha-beta pruning (you can use cut-off as well if you like).

(15 points) Create another class and call it `IterativeDeepeningAI`. Modify you previous implementation to perform iterative deepening search and use it in `IterativeDeepeningAI.make_move`.

(60 points) Create another class and call it `MCTSAI` and modify its `make_move` function to use Monte Carlo Tree Search for decision making. Use `play_with_self` function (or `play_with_self_non_verbose`) in `AIEngine` that lets two bots play with each other.

Leave two instances of `MCTSAI` bots to play with each other for a while (start with a couple of hours) and collect their decision information in file called `mcts.model` rewrite this file with more collected win/lose/draw scores every full run. You will load `mcts.model` once you deploy `MCTSAI` against a human player. Your `MCTSAI` bot does not have to be perfect but should be able to do better than `GenericAI`. Explain every step of the process for training and testing. Also explain how I can test your trained `MCTSAI` bot (if I need to do anything other than what I do for testing other bots).

More Explanations: This part contains more detailed explanations about how you will implement the `MCTSAI` bot. So if you think you know what you are doing, just skip this part. Your main function will be a for loop which calls `play_with_self_non_verbose` many times and in each call, loads `mcts.model` file, makes two instances of `MCTSAI` bots play with each other and then saves the modified MCST in `mcts.model`. This design lets you kill the loop anytime you want to test the performance of the model you have created so far! **I will not assign any grades to submissions** that only have 4 functions `selection`, `expansion`, `simulation`, and `backpropagation` which are **not filled or are filled with lines of code that do not do anything towards creating/improving the MCT**.
In fact you don't need such 4 functions in the current implementation. The function `make_move` which `MCTSAI` inherits from `GenericAI` class will be the right place to implement whatever required for selection and expansion. The function `get_moves` will provide you with all possible moves and in 80% of the times you can select from what the tree knows (unless the tree node has never been expanded) and in the remaining 20% of the times (plus the cases that the tree node is not expanded at all), you can expand by randomly selecting one of the possible moves! `simulation` itself is implicitly implemented in the `play_with_self_non_verbose` function, and to implement `backpropagation` you can override and fill in `record_winner` function in `MCTSAI` class. I don't comment on how you create the MCTree structure and leave that to you as I'm pretty sure you have proper knowledge from COMP 8042 for doing it. Just make sure the tree nodes have back pointers to their parents (this will come in handy in `backpropagation`).
Important note: the bots intelligently decide on going for lengthy games ending in draw by 50 moves to avoid losing games. Make sure you try dealing with this problem.
Important note 2: I have created a starter csv file containing Roy Lopez opening book moves with different variations and provided it in `static/roy_lopez_fens.csv`. You can use this file to populate an initial version of the tree to prevent it from drifting into useless moves. If you know chess and prefer other openings you can extend this file which will get the MCTS algorithm in a stronger starting point.