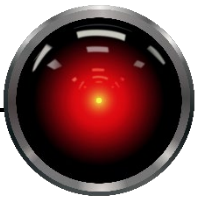

Assignment 3 – A hop, skip, and a jump…

Part I: Questions (20 points each)

1. A common strategy in tic-tac-toe is the so-called 3 corners strategy where a player places their symbol in 3 corners leaving two paths to win, e.g.:

x	o	x
o		
		x

How could we use a transposition table to reduce the search space in a case like this?

2. Chance nodes Γ and Ψ have outcomes: A, B, C, and D for Γ and E and F for Ψ. Given the following table of probabilities and utilities, would a player maximize their utility by selecting Γ or Ψ?

P(outcome)	utility
P(A)=.35	10
P(B)=.50	15
P(C)=.05	29
P(D)=.10	14
P(E)=.25	18
P(F)=.75	13

Show your work.

3. The CheckerBoard class in the programming assignment below cannot detect stalemates that occur when the board is in the same configuration three different times. How could one modify the class to detect this efficiently? (No implementation is required, just well laid out plan that could be implemented by any skilled computer scientist.)

Part II: Programming Assignment – 120 points

You are to write an agent that plays checkers and the main game loop for conducting a game between agents. The following are provided for you on Canvas in a zip archive. The following is a subset of the files in directory lib/

- checkerboard.py – Contains the checkerboard class that we have discussed in class.
- abstractstrategy.py – An abstract Strategy class that should be extended (see below) to implement your utility function. It contains the following methods, most of which raise a NotImplemented exception:

  - constructor. Takes arguments player, game, and maxplies, and an optional boolean variable verbose. Player specifies the maxplayer and is ‘r’ or ‘b’, game is a CheckerBoard class (or instance, but you really only need to access the class methods), and maxplies is the tree depth at which the cutoff function will prune the search. As you have seen in previous assignments, variable verbose is simply a Boolean variable to turn on and off your debug statements.
  - evaluate. Takes a CheckerBoard and turn and determines the strength related to the maxplayer given in the constructor. For example, a strong red board should return a high score if the constructor was invoked with ‘r’, and a low score with ‘b’. The optional argument turn may be used to enhance your evaluation function for a specific player’s turn, but may be ignored if you do not want to create a turn specific evaluation function.
  - verbose. Sets debug mode to True or False.
  - play. Takes a checkerboard and determines the best move with respect to alpha-beta search for the maxplayer.
- human.py – A concrete strategy class derived from abstractstrategy.Strategy that lets humans play. Uses the charIO module that is also provided. It is designed to let you play against an agent. A list of moves is presented and you press a character corresponding to the move. Note that some debuggers will not emulate terminal I/O without a carriage return and you may need to press enter.
- WCDF_Revised_Rules.doc – Rules of checkers. The checkerboard class implements most rules including draws. It does not implement the draw on entering the same state 3 times (rule 1.32.1).
- tonto - A compiled strategy called *Tonto* is available. Source code is not provided, but compiled versions for Python 3.7 and 3.8 are in the __pycache__ directory. See the checkers.py skeleton for details on how to import it. You can use it to play against your strategy. It isn’t too smart…

There are also several other support classes in the code archive. Note that a discussion group has been created that will permit you to share compiled strategies with your classmates if you so choose. Note that compiled strategies only work with the version of Python that compiled them. You can always tell by looking at the file. For example, tonto for Python 3.8 is in lib/__pycache__/tonto.cpython-38.pyc. See the checkers skeleton for an example of how to load a compiled class for which you do not have source.

You are to implement two classes in **ai.py**: Strategy and AlphaBeta. Strategy is a concrete implementation of abstractstrategy.Strategy. There are no requirements on how good your evaluation function must be, but 1) you are expected to make a reasonable effort (it is fun, I promise) and 2) your evaluation function should return reasonable results. For example, if we set up a board where black has a very strong advantage, reporting a score that favors red would not be appropriate. Class AlphaBetaSearch implements an alpha-beta pruning minimax search. Your minimax search must implement a ply-based cutoff test.

As always, you must stick to the provided application programming interface or gradescope tests will fail. As with previous assignments, we will be mixing our solutions with yours so that we can test individual poritions of your functionality (e.g. our AlphaBeta with your Strategy, etc.)

The red and black variables should be instantiated to concrete strategy classes (not instances of classes). That is, to play as human red player and a computer black player with 5 turn lookahead (2*5 plies), you would call Game(red=human.Strategy, black=ai.Strategy, maxplies=10). The init argument to Game allows one to pass in a specific checkerboard which is interesting for examining the strength of one's board utility function without playing a full game.

The game should be implemented in **checkers.py** (the entry point of the program) with the following signature:

```
        def Game(red=Strategy.Stategy, black=Human.Strategy, init=None,
                    maxplies=8, verbose=False)
```

Within Game, you can create instances of your strategy, e.g.:

```
        redplayer = red('r', checkerboard.CheckerBoard, maxplies)
```

which creates an instance of whatever strategy you passed in as red

and then take turns calling play on the different strategies with the evolving game board.

Hints: This is a rather complicated program and it is easy to make mistakes.

Design for testability and make sure that small components of your algorithm are working. There are a number of predefined board configurations in boardlibrary. It is strongly suggested that you work through some of these by hand. Suggestions:

- Try your board utility function on known boards. Several are provided for you in the dictionary boardlibrary.boards. See the unit tests in lib/unittest.py for an example of accessing several board configurations. You are free to add to these if you wish, but they were designed for testing the Checkerboard class, not your Strategy and AlphaBeta search.
- Add debugging statements and make sure that that your alpha-beta cutoffs work correctly on small examples.

To turn in:
Submit your code modules to gradescope along with a game.txt file logging your strategy playing against either Tonto, you, or a loved one that you managed to convince to play your game because it was just too cool.