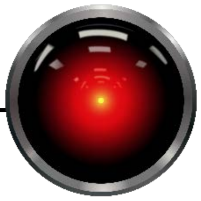

Assignment 3

Part I: Questions (20 points each)

1. Backgammon rolls are between 2 and 12. Find the probability for each pair of numbers. That is without distinguishing between (5,6) and (6,5).
2. Explain in your own words the role of chance nodes in stochastic minimax search and how they relate to classic minimax search.
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 a program that plays checkers. The following are provided for you on Blackboard in a zip archive:

- checkerboard.py – Contains the checkerboard class that we have discussed in class.
- abstractstrategy.py – An abstract Strategy class that should be extended to implement your utility function. It contains the following methods:
  - constructor. Takes arguments player, game, and maxplies. player is ‘r’ or ‘b’, game is a CheckerBoard class (not instance, used to access class methods), and maxplies is the tree depth to which the cutoff function should be applied.
  - utility. Takes a CheckerBoard and determines the strength related to player. For example a strong red board should return a high score if the constructor was invoked with ‘r’, and a low score with ‘b’. Note that this is not implemented in the abstract class.
  - play. Takes a checkerboard and determines the best move with respect to alpha-beta search for the player associated with the class instance. This must also implemented in the derived class.
- human.py – A concrete strategy class derived from AbstractStrategy that lets humans play. Uses the charIO module that is also provided.

There are also several other support classes in the zip file including rules for playing checkers. Note that a discussion group has been created that will permit you to share compiled strategies with your classmates.

You are to implement class Strategy in the file ai.py. Strategy is a concrete class derived from AbstractStrategy and should follow its interface. You will need to design an alpha-beta search and evaluation function. The evaluation function is part of Strategy, but the alpha-beta search is a separate class that must be contained within ai.py. Your alpha-beta search should be a class with the following signature:

```
class AlphaBetaSearch:
    """AlphaBetaSearch
    Conduct alpha beta searches from a given state.

    Example usage:
    # Given an instance of a class derived from AbstractStrategy, set up class
    # to determine next move, maximizing utility with respect to red player
    # and minimiizing with respect to black player.  Search 3 plies.
    search = AlphaBetaSearch(strategy, 'r', 'b', 3)

    # To find the move, run the alphabeta method
    best_move = search.alphabeta(some_checker_board)
    """

    def __init__(self, strategy, maxplayer, minplayer, maxplies=3,
                 verbose=False):
        """"AlphaBetaSearch - Initialize a class capable of alphabeta search
        strategy - implementation of AbstractStrategy class
        maxplayer - name of player that will maximize the utility function
        minplayer - name of player that will minimize the utility function
        maxplies- Maximum ply depth to search
        verbose - Output debugging information
        """

    def alphabeta(self, state):
        """alphbeta(state) - Run an alphabeta search from the current
       state.  Returns best action.
      """

    # define other helper methods as needed
```

You will also need to implement a game playing function in **checkers.py** with the following signature:

def Game(red=human.Strategy, black=ai.Strategy, init=None,
maxplies=8, verbose=False)

The red and black variables should be instantiated to 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=5). The init argument 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.

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

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

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. For example, when testing alpha-beta search, work with shallow searches from a board whose configuration you know and add statements that show you what is happening when a verbose flag is set to True. There are a number of predefined board configurations in boardlibrary. See the unit tests in checkerboard.py for an example of accessing them. You are free to add to these.

- When designing a utility function, test it on board configurations. An example can be seen in boardlibary.boards[“StrategyTest1”] which sets up a board where you can see if your utility function is doing well.

To turn in:

Submit checkers.py, ai.py and any other routines that you create. As always, turn in a print out and electronic versions. You may not modify the checkerboard.py class as it will prevent the grader from running your code, but if there is additional functionality that you wish to add, you are welcome to write functions that compute it, but remember that when we evaluate your ai.strategy with respect to an implementation of game that works with the stock code.