```python
"""
ai - search & strategy module
implement a concrete Strategy class and AlphaBetaSearch
"""
import abstractstrategy
from checkerboard import CheckerBoard as b
import math

class Strategy(abstractstrategy.Strategy):
    def __init__(self, maxplayer, game, maxplies):
        super(Strategy, self).__init__(maxplayer, game, maxplies)
        self.search_algorithm = AlphaBetaSearch(self, self.maxplayer, self.minplayer, self.maxplies, verbose=False)
        # todo: get this checked
        #####################################################
        # No need to repeat, because when you call (self.maxplayer, self.minplayer, self.maxplies) you'll access those values through the super
        '''
        self.maxplayer = maxplayer  # Not max player, other player thing
        self.game = game  # checkerboard class
        self.maxplies = maxplies  # use this for a cutoff
        '''
        #####################################################

    def play(self, board):  # Note: board is a checkerboard
        """"play - Make a move
                Given a board, return (newboard, action) where newboard is
                the result of having applied action to board and action is
                determined via a game tree search (e.g. minimax with alpha-beta
                pruning).
                """
        # search = AlphaBetaSearch(self, self.maxplayer, self.minplayer, self.maxplies, verbose=False)  # Moved this into the constructor of stategy
        # action = search.alpha_beta(board)  # todo Why is this method not being able to be called

        new_board = None
        chosen_action = self.search_algorithm.alpha_beta(board)
        new_board = board.move(chosen_action) # call a new board with the best utility value action
```

```
        return (new_board, chosen_action)

    def utility(self, board):
        # return the utility cost of the board being passed in
        utility_value = 0
        '''
        Ideas:
        Number of pawn
        Number of kings
        count total piece
        Dist to king
        Single jump
        Multiple Jumps


        below might cime in handy
        try:
            pidx = self.pawns.index(player)
        except ValueError:
            raise ValueError("Unknown player")

        # If we see any captures along the way, we will stop looking
        # for moves that do not capture as they will be filtered out
        # at the end.
        moves = []
        # Scan each square
        for r in range(self.rows):
            for c in range(self.coloffset[r], self.cols, self.step):
                piece = self.board[r][c]
                # If square contains pawn/king of player who will be moving
                if piece in self.players[pidx]:
                    # Determine types of moves that can be made
                    if piece == self.pawns[pidx]:
                        movepaths = self.pawnmoves[player]
                    else:
                        movepaths = self.kingmoves
                    # Generate moves based on possible directions
                    newmoves = self.genmoves(r, c, movepaths, pidx)
                    moves.extend(newmoves)
        '''
        return utility_value
```

```

class AlphaBetaSearch:
    '''
    prunes away branches that cannot possibly influence the final decision
    '''
    infinity = float('Infinity')
    negative_infinity = float('-Infinity')
    def __init__(self, strategy, maxplayer, minplayer, maxplies=3, verbose=False):
        self.strategy = strategy
        self.maxplayer = maxplayer
        self.minplayer = minplayer
        self.maxplies = maxplies
        self.verbose = verbose
        self.initial_alpha = self.negative_infinity  # alpha == best option for maximizer
        self.initial_beta = self.infinity

        # for returning the

    # todo create a cutoff
    # todo: learn how do the alpha beta on all possible actions
    def alpha_beta(self, state):
          # beta == best option for minimizer

        actions = state.get_actions(self.maxplayer)

        if actions:  # evaluate each actions using the utility method (self.utility(action)) and choose one action that the "max player" should make since were using an alpha beta max min algorithm
            for action in actions:
        v = self.max_value(state, self.initial_alpha, self.initial_beta)  # this is how the alpha and beta get passed down
        return action in state.get_actions(player) with value v

    def max_value(self, state, alpha, beta):
        if state.is_terminal()[0] is True:
            return self.strategy.utility(state)
        v = float('-Infinity')  # initial value of max_value
        for action in Actions(state):
            v = max(v, self.min_value(Result(state, action), alpha, beta))
            if v >= beta:
```

```
                return v
            alpha = max(alpha, v)
        return v

    def min_value(self, state, alpha, beta):
        if state.is_terminal()[0] is True:
            return self.strategy.utility(state)
        v = float('Infinity')  # initial value of min_value
        for action in state.get_actions(state):
            v = min(v, self.max_value(Result(state, action), alpha, beta))
            if v >= alpha:
                return v
            beta = min(beta, v)
        return v
```