Calvin Pugmire
CS 470, section 001

Programming Assignment (Adversarial Search II)

1.
Depth limit of 1: 0.1 seconds.
Depth limit of 2: 0.5 seconds.
Depth limit of 3: 1.5 seconds.
Depth limit of 4: 10 seconds.
Depth limit of 5: times out.

2.
Depth limit of 2.

3.
EM is much better about blocking potential losses earlier on.
EM toys with you more often.

4.
Depth limit of 1: 1st: win. 2nd: win.
Depth limit of 2: 1st: win. 2nd: win.
Depth limit of 3: 1st: win. 2nd: win.
Depth limit of 3: 1st: win. 2nd: win.
Depth limit of 4: 1st: win. 2nd: win.
Not in this case.

5.
Game 1: EM=1 vs AB=3: EM win.
Game 2: EM=2 vs AB=3: EM win.
Game 3: EM=3 vs AB=3: EM loss.
Game 5: EM=4 vs AB=3: EM loss.
Game 4: EM=3 vs AB=4: EM win.
Game 6: AB=1 vs EM=3: EM loss.
Game 7: AB=2 vs EM=3: EM win.
Game 8: AB=3 vs EM=3: EM loss.
Game 9: AB=4 vs EM=3: EM loss.
Game 0: AB=5 vs EM=3: EM loss.
Higher depth limit positively impacts AB vs EM.
Higher depth limit negatively impacts EM vs AB.

6.
6 hours.
Revising AB code.
I would revise AB code during the previous PA.

```
# Modified 10.3.2023 by Chris Archibald to
#  - incorporate MCTS with other code
#  - pass command line param string to each AI

import numpy as np



class AIPlayer:
    def __init__(self, player_number, name, ptype, param):
        self.player_number = player_number
        self.name = name
        self.type = ptype
        self.player_string = 'Player {}: '.format(player_number) + self.name
        self.other_player_number = 1 if player_number == 2 else 2


        # Parameters for the different agents


        self.depth_limit = 3  # default depth-limit - change if you desire
        # Alpha-beta
        # Example of using command line param to overwrite depth limit
        if self.type == 'ab' and param:
            self.depth_limit = int(param)


        # Expectimax
        # Example of using command line param to overwrite depth limit
        if self.type == 'expmax' and param:
            self.depth_limit = int(param)


        # MCTS
        self.max_iterations = 1000  # Default max-iterations for MCTS - change if you desire
        # Example of using command line param to overwrite max-iterations for MCTS
        if self.type == 'mcts' and param:
            self.max_iterations = int(param)

    def get_alpha_beta_move(self, board):
        """
        Given the current state of the board, return the next move based on
        the alpha-beta pruning algorithm


        This will play against either itself or a human player


        INPUTS:
        board - a numpy array containing the state of the board using the
                following encoding:
                - the board maintains its same two dimensions
                    - row 0 is the top of the board and so is
                      the last row filled
                - spaces that are unoccupied are marked as 0
                - spaces that are occupied by player 1 have a 1 in them
                - spaces that are occupied by player 2 have a 2 in them


        RETURNS:
        The 0 based index of the column that represents the next move
        """
        # moves = get_valid_moves(board)
        # best_move = np.random.choice(moves)


        # YOUR ALPHA-BETA CODE GOES HERE


        my_board = np.copy(board)


        value, best_move = self.max_value(my_board, self.depth_limit, float('-inf'), float('inf'))


        return best_move

    def max_value(self, board, limit, a, b):
        v = float('-inf')
```

```
        m = -1

        if is_winning_state(board, self.player_number) or is_winning_state(board, self.other_player_number) or (limit == 0):
            return self.evaluation_function(board), m

        limit -= 1

        for act in get_valid_moves(board):
            my_board = np.copy(board)
            make_move(my_board, act, self.player_number)
            new_v, new_m = self.min_value(my_board, limit, a, b)
            if v <= new_v:
                v = new_v
                m = act
            if v >= b:
                return v, m
            a = max(a, v)

        return v, m

    def min_value(self, board, limit, a, b):
        v = float('inf')
        m = -1

        if is_winning_state(board, self.player_number) or is_winning_state(board, self.other_player_number) or (limit == 0):
            return self.evaluation_function(board), m

        limit -= 1

        for act in get_valid_moves(board):
            my_board = np.copy(board)
            make_move(my_board, act, self.other_player_number)
            new_v, new_m = self.max_value(my_board, limit, a, b)
            if v >= new_v:
                v = new_v
                m = act
            if v <= a:
                return v, m
            b = min(b, v)

        return v, m

    def get_mcts_move(self, board):
        """
        Use MCTS to get the next move
        """

        # How many iterations of MCTS will we do?
        max_iterations = 1000  # Modify to work for you

        # Make the MCTS root node from the current board state
        root = MCTSNode(board, self.player_number, None)

        # Run our MCTS iterations
        for i in range(max_iterations):
            # Select + Expand
            cur_node = root.select()

            # Simulate + backpropate
            cur_node.simulate()

        # Print out the info from the root node
        root.print_node()
        print('MCTS chooses action', root.max_child())
        return root.max_child()

    def get_expectimax_move(self, board):
```

```
        """
        Given the current state of the board, return the next move based on
        the expectimax algorithm.

        This will play against the random player, who chooses any valid move
        with equal probability

        INPUTS:
        board - a numpy array containing the state of the board using the
                following encoding:
                - the board maintains its same two dimensions
                    - row 0 is the top of the board and so is
                      the last row filled
                - spaces that are unoccupied are marked as 0
                - spaces that are occupied by player 1 have a 1 in them
                - spaces that are occupied by player 2 have a 2 in them

        RETURNS:
        The 0 based index of the column that represents the next move
        """
        # moves = get_valid_moves(board)
        # best_move = np.random.choice(moves)

        # YOUR EXPECTIMAX CODE GOES HERE

        my_board = np.copy(board)

        value, best_move = self.player_value(my_board, self.depth_limit)

        if best_move == -1:
            best_move = get_valid_moves(board)[0]

        return best_move

    def player_value(self, board, limit):
        v = float('-inf')
        m = -1

        if is_winning_state(board, self.player_number) or is_winning_state(board, self.other_player_number) or (limit == 0):
            return self.evaluation_function(board), m

        limit -= 1

        for act in get_valid_moves(board):
            my_board = np.copy(board)
            make_move(my_board, act, self.player_number)
            new_v, new_m = self.chance_value(my_board, limit)
            if v <= new_v:
                v = new_v
                m = act

        return v, m

    def chance_value(self, board, limit):
        v = 0
        m = -1

        if is_winning_state(board, self.player_number) or is_winning_state(board, self.other_player_number) or (limit == 0):
            return self.evaluation_function(board), m

        limit -= 1

        chance = 1 / len(get_valid_moves(board))
        for act in get_valid_moves(board):
            my_board = np.copy(board)
            make_move(my_board, act, self.other_player_number)
            new_v, new_m = self.player_value(my_board, limit)
```

```
            v += new_v*chance

        return v, m

    def evaluation_function(self, board):
        """
        Given the current stat of the board, return the scalar value that
        represents the evaluation function for the current player

        INPUTS:
        board - a numpy array containing the state of the board using the
                following encoding:
                - the board maintains its same two dimensions
                  - row 0 is the top of the board and so is
                    the last row filled
                - spaces that are unoccupied are marked as 0
                - spaces that are occupied by player 1 have a 1 in them
                - spaces that are occupied by player 2 have a 2 in them

        RETURNS:
        The utility value for the current board
        """

        # YOUR EVALUATION FUNCTION GOES HERE

        #player_board = np.copy(board)
        #player_board[player_board == 0] = self.player_number
        #num_4s = num_in_state(player_board, '{0}{0}{0}{0}'.format(self.player_number))
        num_4s = num_in_state(board, '{0}{0}{0}{0}'.format(self.player_number))
        if num_4s > 0:
            return float('inf')
        num_4b = num_in_state(board, '{0}{0}{0}{1}'.format(self.other_player_number, self.player_number))
        num_4b += num_in_state(board, '{1}{0}{0}{0}'.format(self.other_player_number, self.player_number))
        num_4b += num_in_state(board, '{0}{0}{1}{0}'.format(self.other_player_number, self.player_number))
        num_4b += num_in_state(board, '{0}{1}{0}{0}'.format(self.other_player_number, self.player_number))
        num_3s = num_in_state(board, '{0}{0}{0}'.format(self.player_number))
        num_3b = num_in_state(board, '{0}{0}{1}'.format(self.other_player_number, self.player_number))
        num_3b += num_in_state(board, '{1}{0}{0}'.format(self.other_player_number, self.player_number))
        num_2s = num_in_state(board, '{0}{0}'.format(self.player_number))

        #enemy_board = np.copy(board)
        #enemy_board[enemy_board == 0] = self.other_player_number
        #op_num_4s = num_in_state(enemy_board, '{0}{0}{0}{0}'.format(self.other_player_number))
        op_num_4s = num_in_state(board, '{0}{0}{0}{0}'.format(self.other_player_number))
        if op_num_4s > 0:
            return float('-inf')
        op_num_4b = num_in_state(board, '{0}{0}{0}{1}'.format(self.player_number, self.other_player_number or 0))
        op_num_4b += num_in_state(board, '{1}{0}{0}{0}'.format(self.player_number, self.other_player_number or 0))
        op_num_4b += num_in_state(board, '{0}{0}{1}{0}'.format(self.player_number, self.other_player_number or 0))
        op_num_4b += num_in_state(board, '{0}{1}{0}{0}'.format(self.player_number, self.other_player_number or 0))
        op_num_3s = num_in_state(board, '{0}{0}{0}'.format(self.other_player_number))
        op_num_3b = num_in_state(board, '{0}{0}{1}'.format(self.player_number, self.other_player_number or 0))
        op_num_3b += num_in_state(board, '{1}{0}{0}'.format(self.player_number, self.other_player_number or 0))
        op_num_2s = num_in_state(board, '{0}{0}'.format(self.other_player_number))

        #return num_4s - op_num_4s
        return (10000*num_4b + 500*num_3s + 20*num_3b + 1*num_2s) - (10000*op_num_4b + 500*op_num_3s + 20*op_num_3b + 1*op_num_2s)

def num_in_state(board, player_win_str):
    to_str = lambda a: ''.join(a.astype(str))

    def check_horizontal(b):
        num = 0
        for row in b:
            num += to_str(row).count(player_win_str)
        return num

    def check_verticle(b):
        return check_horizontal(b.T)
```

```
    def check_diagonal(b):
        num = 0

        for op in [None, np.fliplr]:
            op_board = op(b) if op else b

            root_diag = np.diagonal(op_board, offset=0).astype(int)
            num += to_str(root_diag).count(player_win_str)

            for i in range(1, b.shape[1] - 3):
                for offset in [i, -i]:
                    diag = np.diagonal(op_board, offset=offset)
                    diag = to_str(diag.astype(int))
                    num += diag.count(player_win_str)

        return num

    return (check_horizontal(board) +
            check_verticle(board) +
            check_diagonal(board))



class RandomPlayer:
    def __init__(self, player_number):
        self.player_number = player_number
        self.type = 'random'
        self.name = 'random'
        self.player_string = 'Player {}: random'.format(player_number)

    def get_move(self, board):
        """
        Given the current board state select a random column from the available
        valid moves.

        INPUTS:
        board - a numpy array containing the state of the board using the
                following encoding:
                - the board maintains its same two dimensions
                    - row 0 is the top of the board and so is
                      the last row filled
                - spaces that are unoccupied are marked as 0
                - spaces that are occupied by player 1 have a 1 in them
                - spaces that are occupied by player 2 have a 2 in them

        RETURNS:
        The 0 based index of the column that represents the next move
        """
        valid_cols = []
        for col in range(board.shape[1]):
            if 0 in board[:, col]:
                valid_cols.append(col)

        return np.random.choice(valid_cols)



class HumanPlayer:
    def __init__(self, player_number):
        self.player_number = player_number
        self.type = 'human'
        self.name = 'human'
        self.player_string = 'Player {}: human'.format(player_number)

    def get_move(self, board):
        """
        Given the current board state returns the human input for next move

        INPUTS:
        board - a numpy array containing the state of the board using the
```

```
            following encoding:
            - the board maintains its same two dimensions
                - row 0 is the top of the board and so is
                  the last row filled
            - spaces that are unoccupied are marked as 0
            - spaces that are occupied by player 1 have a 1 in them
            - spaces that are occupied by player 2 have a 2 in them

        RETURNS:
        The 0 based index of the column that represents the next move
        """

        valid_cols = []
        for i, col in enumerate(board.T):
            if 0 in col:
                valid_cols.append(i)

        move = int(input('Enter your move, Human: '))

        while move not in valid_cols:
            print('Column full, choose from:{}'.format(valid_cols))
            move = int(input('Enter your move: '))

        return move


# CODE FOR MCTS
class MCTSNode:
    def __init__(self, board, player_number, parent):
        self.board = board
        self.player_number = player_number
        self.other_player_number = 1 if player_number == 2 else 2
        self.parent = parent
        self.moves = get_valid_moves(board)
        self.terminal = (len(self.moves) == 0) or is_winning_state(board, player_number) or is_winning_state(board,
                                                                                                             self.other_player_number)
        self.children = dict()
        for m in self.moves:
            self.children[m] = None

        # Set up stats for MCTS
        # Number of visits to this node
        self.n = 0

        # Total number of wins from this node (win = +1, loss = -1, tie = +0)
        # Note: these wins are from the perspective of the PARENT node of this node
        #       So, if self.player_number wins, that is -1, while if self.other_player_number wins
        #       that is a +1.  (Since parent will be using our UCB value to make choice)
        self.w = 0

        # c value to be used in the UCB calculation
        self.c = np.sqrt(2)

    def print_tree(self):
        # Debugging utility that will print the whole subtree starting at this node
        print("****")
        self.print_node(self)
        for m in self.moves:
            if self.children[m]:
                self.children[m].print_tree()
        print("****")

    def print_node(self):
        # Debugging utility that will print this node's information
        print('Total Node visits and wins: ', self.n, self.w)
        print('Children: ')
        for m in self.moves:
            if self.children[m] is None:
                print('   ', m, ' is None')
```

```
            else:
                print('   ', m, ':', self.children[m].n, self.children[m].w, 'UB: ',
                      self.children[m].upper_bound(self.n))

    def max_child(self):
        # Return the most visited child
        # This is used at the root node to make a final decision
        max_n = 0
        max_m = None

        for m in self.moves:
            if self.children[m].n > max_n:
                max_n = self.children[m].n
                max_m = m
        return max_m

    def upper_bound(self, N):
        # This function returns the UCB for this node
        # N is the number of samples for the parent node, to be used in UCB calculation

        # YOUR MCTS TASK 1 CODE GOES HERE

        # To do: return the UCB for this node (look in __init__ to see the values you can use)

        return 0

    def select(self):
        # This recursive function combines the selection and expansion steps of the MCTS algorithm
        # It will return either:
        # A terminal node, if this is the node selected
        # The new node added to the tree, if a leaf node is selected

        max_ub = -np.inf  # Track the best upper bound found so far
        max_child = None  # Track the best child found so far

        if self.terminal:
            # If this is a terminal node, then return it (the game is over)
            return self

        # For all of the children of this node
        for m in self.moves:
            if self.children[m] is None:
                # If this child doesn't exist, then create it and return it
                new_board = np.copy(self.board)  # Copy board/state for the new child
                make_move(new_board, m, self.player_number)  # Make the move in the state

                self.children[m] = MCTSNode(new_board, self.other_player_number, self)  # Create the child node
                return self.children[m]  # Return it

            # Child already exists, get it's UCB value
            current_ub = self.children[m].upper_bound(self.n)

            # Compare to previous best UCB
            if current_ub > max_ub:
                max_ub = current_ub
                max_child = m

        # Recursively return the select result for the best child
        return self.children[max_child].select()

    def simulate(self):
        # This function will simulate a random game from this node's state and then call back on its
        # parent with the result

        # YOUR MCTS TASK 2 CODE GOES HERE

        # Pseudocode in comments:
```

```
        ##################################
        # If this state is terminal (meaning the game is over) AND it is a winning state for self.other_player_number
        #   Then we are done and the result is 1 (since this is from parent's perspective)
        #
        # Else-if this state is terminal AND is a winning state for self.player_number
        #   Then we are done and the result is -1 (since this is from parent's perspective)
        #
        # Else-if this is not a terminal state (if it is terminal and a tie (no-one won, then result is 0))
        #   Then we need to perform the random rollout
        #     1. Make a copy of the board to modify
        #     2. Keep track of which player's turn it is (first turn is current nodes self.player_number)
        #     3. Until the game is over:
        #           3.1  Make a random move for the player who's turn it is
        #           3.2  Check to see if someone won or the game ended in a tie
        #                (Hint: you can check for a tie if there are no more valid moves)
        #           3.3  If the game is over, store the result
        #           3.4  If game is not over, change the player and continue the loop
        #
        # Update this node's total reward (self.w) and visit count (self.n) values to reflect this visit and result


        # Back-propagate this result
        # You do this by calling back on the parent of this node with the result of this simulation
        #   This should look like: self.parent.back(result)
        # Tip: you need to negate the result to account for the fact that the other player
        #   is the actor in the parent node, and so the scores will be from the opposite perspective
        pass

    def back(self, score):
        # This updates the stats for this node, then backpropagates things
        # to the parent (note the inverted score)
        self.n += 1
        self.w += score
        if self.parent is not None:
            self.parent.back(-score)  # Score inverted before passing along


# UTILITY FUNCTIONS

# This function will modify the board according to
# player_number moving into move column
def make_move(board, move, player_number):
    row = 0
    while row < 6 and board[row, move] == 0:
        row += 1
    board[row - 1, move] = player_number



# This function will return a list of valid moves for the given board
def get_valid_moves(board):
    valid_moves = []
    for c in range(7):
        if 0 in board[:, c]:
            valid_moves.append(c)
    return valid_moves



# This function returns true if player_num is winning on board
def is_winning_state(board, player_num):
    player_win_str = '{0}{0}{0}{0}'.format(player_num)
    to_str = lambda a: ''.join(a.astype(str))

    def check_horizontal(b):
        for row in b:
            if player_win_str in to_str(row):
                return True
        return False

    def check_verticle(b):
        return check_horizontal(b.T)
```

```
def check_diagonal(b):
    for op in [None, np.fliplr]:
        op_board = op(b) if op else b

        root_diag = np.diagonal(op_board, offset=0).astype(int)
        if player_win_str in to_str(root_diag):
            return True

        for i in range(1, b.shape[1] - 3):
            for offset in [i, -i]:
                diag = np.diagonal(op_board, offset=offset)
                diag = to_str(diag.astype(int))
                if player_win_str in diag:
                    return True

    return False

return (check_horizontal(board) or
        check_verticle(board) or
        check_diagonal(board))
```