Assignment 2: Monte Carlo Tree Search and Othello (60 points)

In this assignment, you'll program a Monte Carlo Tree Search module for the board game Othello, also known as Reversi. The AI will decide which move is best using a combination of the UCT selection algorithm and random playouts. Code is also provided for you that will let you play against your AI if you like.

The rules of Othello are as follows:

1. The two player colors are white and black. The white player goes first.
2. You capture an opponent's pieces when they lie in a straight line between a piece you already had on the board and a piece you just played. (A straight line is left-right, up-down, or a 45 degree diagonal.)
3. You can only play a piece that would capture at least one piece. **If you have no legal moves, the turn is passed.**
4. The game is over when neither player has any legal moves left. Whoever controls the most pieces on the board at that point wins.

Something that is slightly unusual about Othello is the fact that a turn might be skipped if a player has no legal plays. You'll have to take that into account in your tree-building.

The AI is presumed to be white for this assignment; if you try the demo mode, you as the human will be playing black.

We'll use a string representation of the board (W for white, B for black, - for an empty space).

```
""" Final code implements Monte Carlo Tree Search for board game Othello."""

import copy
import sys
import numpy as np

NUM_COLS = 8
# With these constant values for players, flipping ownership is just a sign change
WHITE = 1
```

```
NOBODY = 0
BLACK = -1

TIE = 2  # An arbitrary enum for end-of-game

WHITE_TO_PLAY = True

# We'll sometimes iterate over this to look in all 8 directions from a particular square.
# The values are the "delta" differences in row, col from the original square.
# (Hence no (0,0), which would be the same square.)
DIRECTIONS = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]


def read_boardstring(boardstring):
    """Converts string representation of board to 2D numpy int array"""
    board = np.zeros((NUM_COLS, NUM_COLS))
    board_chars = {
        'W': WHITE,
        'B': BLACK,
        '-': NOBODY
    }
    row = 0
    for line in boardstring.splitlines():
        for col in range(NUM_COLS):
            board[row][col] = board_chars.get(line[col], NOBODY) # quietly ignore bad chars
        row += 1
    return board

def find_winner(board):
    """Return identity of winner, assuming game is over.

    Args:
        board (numpy 2D int array):  The othello board, with WHITE/BLACK/NOBODY in spaces

    Returns:
        int constant:  WHITE, BLACK, or TIE.
    """
    # Slick counting of values:  np.count_nonzero counts vals > 0, so pass in
```

```
    # board == WHITE to get 1 or 0 in the right spots
    white_count = np.count_nonzero(board == WHITE)
    black_count = np.count_nonzero(board == BLACK)
    if white_count > black_count:
        return WHITE
    if white_count < black_count:
        return BLACK
    return TIE

def generate_legal_moves(board, white_turn):
    """Returns a list of (row, col) tuples representing places to move.

    Args:
        board (numpy 2D int array):  The othello board
        white_turn (bool):  True if it's white's turn to play
    """

    legal_moves = []
    for row in range(NUM_COLS):
        for col in range(NUM_COLS):
            if board[row][col] != NOBODY:
                continue   # Occupied, so not legal for a move
            # Legal moves must capture something
            if can_capture(board, row, col, white_turn):
                legal_moves.append((row, col))
    return legal_moves

def can_capture(board, row, col, white_turn):
    """ Helper that checks capture in each of 8 directions.

    Args:
        board (numpy 2D int array) - othello board
        row (int) - row of move
        col (int) - col of move
        white_turn (bool) - True if it's white's turn
    Returns:
        True if capture is possible in any direction
    """
```

```
        for r_delta, c_delta in DIRECTIONS:
            if captures_in_dir(board, row, r_delta, col, c_delta, white_turn):
                return True
        return False

    def captures_in_dir(board, row, row_delta, col, col_delta, white_turn):
        """Returns True iff capture possible in direction described by delta parameters

        Args:
            board (numpy 2D int array) - othello board
            row (int) - row of original move
            row_delta (int) - modification needed to row to move in direction of capture
            col (int) - col of original move
            col_delta (int) - modification needed to col to move in direction of capture
            white_turn (bool) - True iff it's white's turn
        """

        # Can't capture if headed off the board
        if (row+row_delta < 0) or (row+row_delta >= NUM_COLS):
            return False
        if (col+col_delta < 0) or (col+col_delta >= NUM_COLS):
            return False

        # Can't capture if piece in that direction is not of appropriate color or missing
        enemy_color = BLACK if white_turn else WHITE
        if board[row+row_delta][col+col_delta] != enemy_color:
            return False

        # At least one enemy piece in this direction, so just need to scan until we
        # find a friendly piece (return True) or hit an empty spot or edge of board
        # (return False)
        friendly_color = WHITE if white_turn else BLACK
        scan_row = row + 2*row_delta # row of first scan position
        scan_col = col + 2*col_delta # col of first scan position
        while 0 <= scan_row < NUM_COLS and 0 <= scan_col < NUM_COLS:
            if board[scan_row][scan_col] == NOBODY:
                return False
            if board[scan_row][scan_col] == friendly_color:
```

```
                return True
            scan_row += row_delta
            scan_col += col_delta
        return False

    def capture(board, row, col, white_turn):
        """Destructively change a board to represent capturing a piece with a move at (row,col).

        The board's already a copy made specifically for the purpose of representing this move,
        so there's no point in copying it again.  We'll return the board anyway.

        Args:
            board (numpy 2D int array) - The Othello board - will be destructively modified
            row (int) - row of move
            col (int) - col of move
            white_turn (bool) - True iff it's white's turn
        Returns:
            The board, though this isn't necessary since it's destructively modified
        """

        # Check in each direction as to whether flips can happen -- if they can, start flipping
        enemy_color = BLACK if white_turn else WHITE
        for row_delta, col_delta in DIRECTIONS:
            if captures_in_dir(board, row, row_delta, col, col_delta, white_turn):
                flip_row = row + row_delta
                flip_col = col + col_delta
                while board[flip_row][flip_col] == enemy_color:
                    board[flip_row][flip_col] = -enemy_color
                    flip_row += row_delta
                    flip_col += col_delta
        return board

    def play_move(board, move, white_turn):
        """Handles the logic of putting down a new piece and flipping captured pieces.

        The board that is returned is a copy, so this is appropriate to use for search.

        Args:
```

```
        board (numpy 2D int array):  The othello board
        move ((int,int)):  A (row, col) pair for the move
        white_turn:  True iff it's white's turn
    Returns:
        board (numpy 2D int array)
    """
    new_board = copy.deepcopy(board)
    new_board[move[0]][move[1]] = WHITE if white_turn else BLACK
    new_board = capture(new_board, move[0], move[1], white_turn)
    return new_board

def evaluation_function(board):
    """Not used currently, but it could be used for smarter playouts"""

    # We could count with loops, but we're feeling fancy
    return np.count_nonzero(board == WHITE) - np.count_nonzero(board == BLACK)

def check_game_over(board):
    """Returns the current winner of the board - WHITE, BLACK, TIE, NOBODY"""

    # It's not over if either player still has legal moves
    white_legal_moves = generate_legal_moves(board, True)
    if white_legal_moves:  # Python idiom for checking for empty list
        return NOBODY
    black_legal_moves = generate_legal_moves(board, False)
    if black_legal_moves:
        return NOBODY
    # I guess the game's over
    return find_winner(board)


def print_board(board):
    """ Print board (and return None), for interactive mode"""
    print(board_to_string(board))

def board_to_string(board):
    printable = {
        -1: "B",
```

```
            0: "-",
            1: "W"
        }
        out = ""
        for row in range(NUM_COLS):
            line = ""
            for col in range(NUM_COLS):
                line += printable[board[row][col]]
            out += line + "\n"
        return out

    MCTS_ITERATIONS = 100

    def play():
        """Interactive play, for demo purposes.  Assume AI is white and goes first."""
        board = starting_board()
        while check_game_over(board) == NOBODY:
            # White turn (AI)
            legal_moves = generate_legal_moves(board, True)
            if legal_moves:  # (list is non-empty)
                print("Thinking...")
                best_move = MCTS_choice(board, True, MCTS_ITERATIONS)
                board = play_move(board, best_move, True)
                print_board(board)
                print("")
            else:
                print("White has no legal moves; skipping turn...")

            legal_moves = generate_legal_moves(board, False)
            if legal_moves:
                player_move = get_player_move(board, legal_moves)
                board = play_move(board, player_move, False)
                print_board(board)
            else:
                print("Black has no legal moves; skipping turn...")
        winner = find_winner(board)
        if winner == WHITE:
            print("White won!")
```

```
        elif winner == BLACK:
            print("Black won!")
        else:
            print("Tie!")

    def starting_board():
        """Returns a board with the traditional starting positions in Othello."""
        board = np.zeros((NUM_COLS, NUM_COLS))
        board[3][3] = WHITE
        board[3][4] = BLACK
        board[4][3] = BLACK
        board[4][4] = WHITE
        return board

    def get_player_move(board, legal_moves):
        """Print board with numbers for the legal move spaces, then get player choice of move

        Args:
            board (numpy 2D int array):  The Othello board.
            legal_moves (list of (int,int)):  List of legal (row,col) moves for human player
        Returns:
            (int, int) representation of the human player's choice
        """
        for row in range(NUM_COLS):
            line = ""
            for col in range(NUM_COLS):
                if board[row][col] == WHITE:
                    line += "W"
                elif board[row][col] == BLACK:
                    line += "B"
                else:
                    if (row, col) in legal_moves:
                        line += str(legal_moves.index((row, col)))
                    else:
                        line += "-"
            print(line)
        while True:
            # Bounce around this loop until a valid integer is received
```

```
        choice = input("Which move do you want to play? [0-" + str(len(legal_moves)-1) + "]")
        try:
            move_num = int(choice)
            if 0 <= move_num < len(legal_moves):
                return legal_moves[move_num]
            print("That wasn't one of the options.")
        except ValueError:
            print("Please enter an integer as your move choice.")
```

Use the following class for your MCTS tree.

```
class MCTSNode:
  def __init__(self, parent, move, board, white_turn):
    self.parent = parent
    self.children = []
    self.white_turn = white_turn
    self.move = move
    self.board = board
    self.playouts = 0
    self.wins = 0

  def __str__(self): # Can modify this for debugging purposes
    s = board_to_string(self.board)
    if self.move is not None:
      s += str(self.move[0]) + "," + str(self.move[1])
    s += "\n" + str(self.wins) + "/" + str(self.playouts) + "\n"
    return s
```

1 (6 points) Write a function UCT that, given a list of MCTSNodes, returns the MCTSNode with the biggest UCB1 value; this function will be used in the selection phase of MCTS. $UCB1(n) = \frac{U(n)}{N(n)} + C\sqrt{\frac{logN(parent(n))}{N(n)}}$ where $U(n)$ is the number of wins through node n, $N(n)$ is the number of playouts that went through node n, and $N(parent(n))$ refers to the number of playouts through the parent of N. C is an arbitrary constant that is tunable for performance, which we will set to $\sqrt{2}$. Create a helper function for the UCB1 calculation, since this is tested in the test code.

```
import math

def UCB1(node):
  #TODO
  return (node.wins/node.playouts) + math.sqrt(2) * math.sqrt((math.log(node.parent.playouts))/node.playouts)

def UCT(nodelist):
  # TODO
  max = None
  maxVal = 0
  for node in nodelist:
    eval = UCB1(node)
    if eval > maxVal:
      max = node
      maxVal = eval
  return max
```

```
# Tests for UCT, UCB1
clear_best_move = """--------
--------
--B-----
---BB---
---BW---
--------
--------
--------"""
# Create a tree corresponding to the three legal moves here.
# The win record will be 2/2, 1/2, 1/2.
my_root = MCTSNode(None, None, read_boardstring(clear_best_move), True)
my_root.playouts = 6
my_root.wins = 2
children_moves = generate_legal_moves(read_boardstring(clear_best_move),True)
children = []
```

```
for move in children_moves:
  new_board = play_move(my_root.board,move,True)
  node = MCTSNode(my_root,move,new_board,False)
  node.playouts = 2
  node.wins = 1
  children.append(node)
children[0].wins = 2 # The best move happens to be the first move listed
my_root.children = children
# Now test UCB1 on each, and UCT to select the best
for child in my_root.children:
  print(UCB1(child))  # Expect about 2.34 for first, 1.84 for other two
print(UCT(my_root.children)) # Expect the node with 2/2 wins
```

```
2.33856619904585
1.8385661990458504
1.8385661990458504
--------
-W------
--W-----
---WB---
---BW---
--------
--------
--------
1,1
2/2
```

2 (12 points) Now implement the selection phase of Monte Carlo Tree Search, in which the best child (highest UCB1 score) is selected until at least one child is missing, at which point we return that node that is missing a child. Note that in order to know whether there is an unexpanded child, you will need to use a board passed in as an argument, and actually play out moves.

The second return value should hold the list of possible moves, as we'll make use of it in the next step.

Othello-specific guidelines: it's possible there are no valid moves for the current player, in which case, play passes to the other player. In this particular function, you can trust node.white_turn is set correctly even if this two-turns-in-a-row event happens. But you should still check for the possibility of no moves for either player, in which case, you can return the current node.

```
def selection(root):
  # TODO
  # Othello edge case
  if (len(generate_legal_moves(root.board, True)) == 0 and len(generate_legal_moves(root.board, False))):
    return root, []

  # if node has no children, return node
  if len(root.children) == 0:
    return root, generate_legal_moves(root.board, root.white_turn)

  # list of children that have been explored
  explored_children = []
  for child in root.children:
    if child.playouts > 0:
      explored_children.append(child)
    else:
      # if there is an unexplored child, select the child
      return child, generate_legal_moves(child.board, child.white_turn)

  # check if all children have been explored (then return the UTC)
  best_child = UCT(root.children)
  return best_child, generate_legal_moves(best_child.board, best_child.white_turn)
```

```
# Test 1 for selection() - uses the tree from the last test section
node, children = selection(my_root)
print(node) # Expect the node with 2/2 wins
print(children) # Expect [(2, 3), (3, 2), (4, 5), (5, 4)]
```

```
--------
-W------
--W-----
---WB---
---BW---
--------
--------
--------
1,1
2/2

[(2, 3), (3, 2), (4, 5), (5, 4)]
```

```
# Test 2 for selection -- endgame
# Board has just one legal play for white
# (notice white piece in upper right corner)
filled_board = """BBBBBBBW
BBBBBBBB
BBBBBBBB
BBBBBBBB
BBBBBBBB
BBBBBBBB
BBBBBBBB
BBBBBBB-"""
my_root2 = MCTSNode(None, None, read_boardstring(filled_board), True)
my_root2.playouts = 2
my_root2.wins = 2
children_moves = generate_legal_moves(read_boardstring(filled_board),True)
children = []
for move in children_moves: # Should be only one
  new_board = play_move(my_root2.board,move,True)
  node = MCTSNode(my_root2,move,new_board,False)
  node.playouts = 2
  node.wins = 0
```

```
    children.append(node)
  my_root2.children = children
  # Selection should stop and return the node when there are no plays left
  node, children = selection(my_root2)
  print(node) # Expect the node with full board, white played
  print(children) # Expect empty list
```

```
BBBBBBBW
BBBBBBBW
BBBBBBBW
BBBBBBBW
BBBBBBBW
BBBBBBBW
BBBBBBBW
BBBBBBBW
7,7
0/2

[]
```

3 (12 points) Now implement the expansion step, adding a child that represents a new move to the node we singled out in the selection step, and returning that child. (If we found a node with no children in the previous step, just return that node.) Be sure to add the new node to the parent's list of children as well. You can determine the unexplored node to add arbitrarily - the first one in the list that isn't already a child will do fine. Return the new node.

```
def expansion(parent, possible_children):
  # TODO
  # if no possible children, return the node
  if len(possible_children) == 0:
    return parent

  move_board = play_move(parent.board, possible_children[0], not parent.white_turn)
  move = MCTSNode(parent, possible_children[0], move_board, not parent.white_turn)
  parent.children.append(move)
  return move
```

```
# Test 1 of expansion: check that node is added to parent's children
current_node, possible_children = selection(my_root)
new_node = expansion(current_node, possible_children)
print(new_node)
print(new_node.parent.children[-1]) # Should be the same node
```

```
--------
-W------
--WW----
---WB---
---BW---
--------
--------
--------
2,3
0/0

--------
-W------
--WW----
---WB---
---BW---
--------
--------
--------
2,3
0/0
```

```
# Test 2 of expansion: just return the selection node when no children
# can be added
current_node, possible_children = selection(my_root2)
node = expansion(current_node, possible_children) # Node with no children
print(node) # Should be a board with no moves left
```

```
BBBBBBBW
BBBBBBBW
BBBBBBBW
BBBBBBBW
```

```
BBBBBBBW
BBBBBBBW
BBBBBBBW
BBBBBBBW
7,7
0/2
```

4 (11 points) The next phase is simulation - play in the game proceeds quickly from that new node to the end of the game. Play random moves until the end, and determine who wins, returning a simple True for a white win and False for a black win. (You should also return False for a tie, which isn't a win.)

```
import random

def simulation(node):
  # TODO
  while check_game_over(node.board) == NOBODY:
    # while the game isn't over, just keep making random moves on the current board
    possible_moves = generate_legal_moves(node.board, node.white_turn)
    if len(possible_moves) == 0:
      node.white_turn = not node.white_turn
    else:
      random_move = random.choice(possible_moves)
      new_board = play_move(node.board, random_move, node.white_turn)
      node = MCTSNode(node,random_move, new_board, not node.white_turn)

  return find_winner(node.board)


# Should consistently return False since this is the nearly-all-black board
winner = simulation(my_root2)
print(winner)
```

```
-1
```

```
# Should return a mix of true and false
# (otherwise maybe you're not taking turns?)
```

```
winner = simulation(my_root)
print(winner)
```

```
1
```

5 (8 points) The last phase is backpropagation: updating the win/loss information for the new node and all its ancestors. Be sure to handle wins for each side correctly; counterintuitively, if white_turn is true for a node, then it's a child of a node where it's Black's turn, and Black wins should be recorded.

```
def backpropagation(node, white_win):
  # TODO
  if not node is None:
    # weird property where it is a win for a node where its white turn if black wins
    if not node.white_turn == white_win:
      node.wins += 1
    node.playouts += 1
    backpropagation(node.parent, white_win)
```

```
# Test backpropagation
current_node, possible_children = selection(my_root)
new_node = expansion(current_node, possible_children)
print('Before backpropagation')
print(new_node)
print(new_node.parent)
print(new_node.parent.parent)
backpropagation(new_node,True)
print('After backpropagation')
print(new_node) # White to go, no increase to wins
print(new_node.parent) # Black to go, expect wins increase by 1
print(new_node.parent.parent) # White to go, no increase to wins
```

```
Before backpropagation
--------
-W------
--WW----
---WB---
```

```
---BW---
--------
--------
--------
2,3
0/0

--------
-W------
--W-----
---WB---
---BW---
--------
--------
--------
1,1
2/2

--------
--------
--B-----
---BB---
---BW---
--------
--------
--------

2/6

After backpropagation
--------
-W------
--WW----
---WB---
---BW---
--------
--------
--------
2,3
0/1

--------
```

```
-W------
--W-----
---WB---
---BW---
--------
--------
--------
1,1
3/3
```

The final Monte Carlo Tree Search code that puts it all together is done for you. When you're ready, try playing a game against it with play(). You can adjust the thinking time of the AI by changing the MCTS_ITERATIONS constant.

```
def MCTS_choice(board, white_turn, iterations):
  start_node = MCTSNode(None,None,board,white_turn)
  for i in range(iterations):
    current_node, possible_children = selection(start_node)
    new_node = expansion(current_node, possible_children)
    white_win = simulation(new_node)
    backpropagation(new_node, white_win)
  # We look for the start node that has the most playouts -
  # not win % because this way favors nodes that have been tried quite a bit
  # (and are also good, or they wouldn't have been tried)
  max_playouts = 0
  best_child = None
  for child in start_node.children:
    if child.playouts > max_playouts:
      max_playouts = child.playouts
      best_child = child
  return best_child.move
```

```
play()
```

```
Thinking...
--------
--------
```

```
----W---
---WW---
---BW---
--------
--------
--------


--------
--------
---0W1--
---WW---
---BW2--
--------
--------
--------
--------
--------
----W---
---WW---
---BBB--
--------
--------
--------

Thinking...
--------
--------
----W---
---WW---
---WBB--
--W-----
--------
--------


--------
----0---
--12W---
---WW---
--3WBB--
--W-----
```

```
--------
--------
--------
--------
---BW---
---WB---
---WBB--
--W-----
--------
--------

Thinking...
```

Thought Questions

6, 3pts) Recall from lecture the simple evaluation function we could use for Othello of counting pieces on each side and finding the difference. Propose a way of turning this evaluation function into a probability (in the range [0,1]) of a move being selected during rollout. Every move should have nonzero probability, better moves should have higher probabilities, and the probabilities should sum to 1. (If you use the Internet or an AI to help with this problem, remember to cite your source.)

TODO One way that we can turn the evaluation function of counting pieces on each side and finding the difference into a probability of a move being selected at rollout is by keeping track of how many tiles would be flipped by a given move. Then each move could be assigned a probability of (how many tiles this specific move flips)/(total number of tiles that can be flipped by all possible moves). This formula would generate a higher probability for (better) moves that capture more pieces, and lower probabilities for (worse) moves that capture fewer pieces. The total of these probabilties would sum to 1 because each probability is generated by dividing the total number of pieces captured by the total number of pieces that could be captured by all moves.

7, 4pts) A slightly more complex evaluation function could count corner pieces and edge pieces for each side, and make these worth a different number of "points." Propose a local search approach to tuning the values of these edge and corner pieces in the evaluation function. Be clear on what the state is, what your proposed neighbors are, how you plan to get a value for the objective function, and how the local search method you've chosen generally proceeds. You can assume we're still using the evaluation function's result as part of rollouts in MCTS.

TODO One way to perform local search by tuning the values of edge and corner pieces in the evaluation function could be to add additional "weights" to moves that play on the edge and corners and prioritize moves that play corner and edge pieces. The state of the board will be the state of the board when it becomes the current player's turn. The neighbors to the current state of the board would be all the legal moves the current player could move. The objective function would be determined by taking the output of the evaluation function and then adding additional "points" to each neighbor based on how many new corner or edge pieces are added by the current move. Local search would then be performed by assigning each neighbor a probability of (points for this move)/(total points), and repeat the process until the time limit is reached or the game reaches an ending state.

8, 4pts) Suppose we replace our rollout evaluation function with a truly extensive evaluation function that also counts threatened pieces, legal places to play, and "safe" pieces that logically can't be captured. To keep the game running smoothly, we also impose a time limit on the search - it can't take more time than it took before. We also perform a local search to tune the values of each kind of piece in the extended evaluation function, but again, we only let it take as much time for the local search as it was spending before.

The AI now actually performs somewhat poorly. Give *two* different reasonable explanations for why the performance is now worse with a more complex evaluation function.

TODO One reason why the more complex evaluation function performs worse is due to the fact that isn't able to explore as many possibilities of game states. Since the evaluation function is much more complicated, that means it will take more time to evaluate each game state. However, we add a time constraint to the search so it can't take more time than it took before, which means that the AI actually isn't able to confidently explore game states that are better for it, leading to a worse performance.

Another reason why the more complex evaluation function performs worse is because the more complex evaluation function could focus too much on moves that have the best immediate benefit. Due to how comprehensive the evaluation function is, the local search would pick the best move looking 1 move ahead, but could actually be setting up the opponent with a move to take even more pieces (such as capturing all but 2 pieces in a row, giving the opponent an opportunity to re-capture the entire row). These are 2 reasons why a more complex evaluation function performs worse.