University of Alabama
College of Engineering

Homework #1

Student name: *Andrew Hankins*

Course: *Artificial Intelligence (CS 565)* – Professor: *Dr. Monica Anderson Herzog*
Due date: *February 8th, 2023*

Updated Python Files

achankins.py

```python
from src.strategies import Strategy
from src.piece import Piece
from src.compare_all_moves_strategy import CompareAllMoves

import src.weight

# Default features to be used in the weighting function

# ’number_occupied_spaces’: number_occupied_spaces,
# The number of spaces that your pieces are currently occupying
# NOTE: This does not include spaces with only a single piece on them

# ’opponents_taken_pieces’: opponents_taken_pieces,
# The number of opponent’s pieces we currently have taken
# NOTE: We would prefer to take as many pieces as possible.
# NOTE: Due to this, we want this parameter to strongly count towards the value

# ’sum_distances’: sum_distances,
# The sum of your pieces distances to the very end of the board

# ’sum_distances_opponent’: sum_distances_opponent,
# The sum of the opponents pieces distances to the very end of the board

# ’number_of_singles’: number_of_singles,
# The amount of spaces we occupy with only one piece on them

# ’sum_single_distance_away_from_home’: sum_single_distance_away_from_home,
# The sum of your single pieces distance to the very end of the board

# ’pieces_on_board’: pieces_on_board,
# The number of pieces that you currently have on the board.
# NOTE: We would prefer fewer pieces to be on the board because it means that
# some of the pieces have reached the very end.
# NOTE: Due to this, we want this parameter to count against the value

# ’sum_distances_to_endzone’: sum_distances_to_endzone,
# The sum of your pieces distances to the start of the endzone

class player1_achankins(CompareAllMoves):

    # Function that will evaluate the board
    def evaluate_board(self, myboard, colour):
        board_stats = self.assess_board(colour, myboard)
        weight_list = src.weight.weight

        # Attempt to normalize the features between a value of 0...1 and weight them
        board_value = float(weight_list[0]) * (board_stats[’sum_distances’] / 163.0) + \
                      float(weight_list[1]) * (board_stats[’number_of_singles’] / 7.0) + \
                      float(weight_list[2]) * (board_stats[’number_occupied_spaces’] / 7.0) + \
                      float(weight_list[3]) * (board_stats[’opponents_taken_pieces’] / 1.0) + \
                      float(weight_list[4]) * (board_stats[’sum_distances_to_endzone’] / 75.0) + \
                      float(weight_list[5]) * (board_stats[’sum_single_distance_away_from_home’] / 100.0) + \
                      float(weight_list[6]) * (board_stats[’pieces_on_board’] / 15.0) + \
                      float(weight_list[7]) * (board_stats[’sum_distances_opponent’] / 163.0)
        return board_value

class player2_achankins(CompareAllMoves):

    # Default features plus the new novel feature to be created

    def evaluate_board(self, myboard, colour):
        board_stats = self.assess_board(colour, myboard)
        return 0
```

compare_all_moves_strategy.py

```
from src.strategies import Strategy
from src.piece import Piece


class CompareAllMoves(Strategy):

    @staticmethod
    def get_difficulty():
        return "Hard"

    # Function that generates the features to be used when calculating the best
    # possible move.
    def assess_board(self, colour, myboard):
        # Get the current location of the pieces on the board
        pieces = myboard.get_pieces(colour)
        # Get the number of pieces on the board
        pieces_on_board = len(pieces)
        # Initialize the features that will be returned by the function
        sum_distances = 0
        number_of_singles = 0
        number_occupied_spaces = 0
        sum_single_distance_away_from_home = 0
        sum_distances_to_endzone = 0
        # Calculate the sum of the pieces distance to home and the sum of the
        # pieces distance to the endzone (last section of board)
        for piece in pieces:
            sum_distances = sum_distances + piece.spaces_to_home()
            if piece.spaces_to_home() > 6:
                sum_distances_to_endzone += piece.spaces_to_home() - 6
        # Get the number of single pieces, the sum of the single pieces distance
        # to home, and the number of occupied spaces.
        for location in range(1, 25):
            pieces = myboard.pieces_at(location)
            if len(pieces) != 0 and pieces[0].colour == colour:
                if len(pieces) == 1:
                    number_of_singles = number_of_singles + 1
                    sum_single_distance_away_from_home += 25 - pieces[0].spaces_to_home()
                elif len(pieces) > 1: # Not counting single spaces
                    number_occupied_spaces = number_occupied_spaces + 1
        # Get the number of piece’s we have taken from the opponent
        opponents_taken_pieces = len(myboard.get_taken_pieces(colour.other()))
        # Get the number of opponent’s pieces on the board
        opponent_pieces = myboard.get_pieces(colour.other())
        # Get the sum of the opponents pieces to their home
        sum_distances_opponent = 0
        for piece in opponent_pieces:
            sum_distances_opponent = sum_distances_opponent + piece.spaces_to_home()
        return {
            ’number_occupied_spaces’: number_occupied_spaces,
            ’opponents_taken_pieces’: opponents_taken_pieces,
            ’sum_distances’: sum_distances,
            ’sum_distances_opponent’: sum_distances_opponent,
            ’number_of_singles’: number_of_singles,
            ’sum_single_distance_away_from_home’: sum_single_distance_away_from_home,
            ’pieces_on_board’: pieces_on_board,
            ’sum_distances_to_endzone’: sum_distances_to_endzone,
        }

    # Function that will start the process to determine the best move, then
    # move the piece
    def move(self, board, colour, dice_roll, make_move, opponents_activity):

        # Determine the best move available
        result = self.move_recursively(board, colour, dice_roll)
        # If the roll is a double then the length will be 4
        not_a_double = len(dice_roll) == 2
        # If the roll is not a double then also check the dice in the reverse
        # order to ensure we currently have chosen the best possible move
        if not_a_double:
            new_dice_roll = dice_roll.copy()
            new_dice_roll.reverse()
            result_swapped = self.move_recursively(board, colour,
                                                   dice_rolls=new_dice_roll)
            if result_swapped[’best_value’] < result[’best_value’] and \
                    len(result_swapped[’best_moves’]) >= len(result[’best_moves’]):
                result = result_swapped

        # Make the best move(s)
        if len(result[’best_moves’]) != 0:
            for move in result[’best_moves’]:
                make_move(move[’piece_at’], move[’die_roll’])

    # Function that will recursively check for the best move
    def move_recursively(self, board, colour, dice_rolls):
        best_board_value = float(’inf’)
        best_pieces_to_move = []

        # Get the players current pieces
        pieces_to_try = [x.location for x in board.get_pieces(colour)]
        pieces_to_try = list(set(pieces_to_try))

        # Get one piece from each location to test
        valid_pieces = []
        for piece_location in pieces_to_try:
            valid_pieces.append(board.get_piece_at(piece_location))
```

compare_all_moves_strategy.py

```
        valid_pieces.sort(key=Piece.spaces_to_home, reverse=True)

        # Get the first dice roll
        dice_rolls_left = dice_rolls.copy()
        die_roll = dice_rolls_left.pop(0)

        # Iterate through each piece and test possible moves
        for piece in valid_pieces:
            if board.is_move_possible(piece, die_roll):
                board_copy = board.create_copy()
                new_piece = board_copy.get_piece_at(piece.location)
                board_copy.move_piece(new_piece, die_roll)
                if len(dice_rolls_left) > 0:
                    result = self.move_recursively(board_copy, colour, dice_rolls_left)
                    if len(result[’best_moves’]) == 0:
                        # we have done the best we can do
                        board_value = self.evaluate_board(board_copy, colour)
                        if board_value < best_board_value and len(best_pieces_to_move) < 2:
                            best_board_value = board_value
                            best_pieces_to_move = [{’die_roll’: die_roll, ’piece_at’: piece.location}]
                    elif result[’best_value’] < best_board_value:
                        new_best_moves_length = len(result[’best_moves’]) + 1
                        if new_best_moves_length >= len(best_pieces_to_move):
                            best_board_value = result[’best_value’]
                            move = {’die_roll’: die_roll, ’piece_at’: piece.location}
                            best_pieces_to_move = [move] + result[’best_moves’]
                else:
                    board_value = self.evaluate_board(board_copy, colour)
                    if board_value < best_board_value and len(best_pieces_to_move) < 2:
                        best_board_value = board_value
                        best_pieces_to_move = [{’die_roll’: die_roll, ’piece_at’: piece.location}]

        return {’best_value’: best_board_value,
                ’best_moves’: best_pieces_to_move}


class CompareAllMovesSimple(CompareAllMoves):

    def evaluate_board(self, myboard, colour):
        board_stats = self.assess_board(colour, myboard)

        board_value = board_stats[’sum_distances’] + 2 ∗ board_stats[’number_of_singles’] – \
                      board_stats[’number_occupied_spaces’] – board_stats[’opponents_taken_pieces’]
        return board_value


class CompareAllMovesWeightingDistance(CompareAllMoves):

    def evaluate_board(self, myboard, colour):
        board_stats = self.assess_board(colour, myboard)
        board_value = board_stats[’sum_distances’] – float(board_stats[’sum_distances_opponent’])/3 + \
                      2 ∗ board_stats[’number_of_singles’] – \
                      board_stats[’number_occupied_spaces’] – board_stats[’opponents_taken_pieces’]
        return board_value


class CompareAllMovesWeightingDistanceAndSingles(CompareAllMoves):

    def evaluate_board(self, myboard, colour):
        board_stats = self.assess_board(colour, myboard)

        board_value = board_stats[’sum_distances’] – float(board_stats[’sum_distances_opponent’])/3 + \
                      float(board_stats[’sum_single_distance_away_from_home’])/6 – \
                      board_stats[’number_occupied_spaces’] – board_stats[’opponents_taken_pieces’]
        return board_value


class CompareAllMovesWeightingDistanceAndSinglesWithEndGame(CompareAllMoves):

    def evaluate_board(self, myboard, colour):
        board_stats = self.assess_board(colour, myboard)

        board_value = board_stats[’sum_distances’] – float(board_stats[’sum_distances_opponent’]) / 3 + \
                      float(board_stats[’sum_single_distance_away_from_home’]) / 6 – \
                      board_stats[’number_occupied_spaces’] – board_stats[’opponents_taken_pieces’] + \
                      3 ∗ board_stats[’pieces_on_board’]

        return board_value


class CompareAllMovesWeightingDistanceAndSinglesWithEndGame2(CompareAllMoves):

    def evaluate_board(self, myboard, colour):
        board_stats = self.assess_board(colour, myboard)

        board_value = board_stats[’sum_distances’] – float(board_stats[’sum_distances_opponent’]) / 3 + \
                      float(board_stats[’sum_single_distance_away_from_home’]) / 6 – \
                      board_stats[’number_occupied_spaces’] – board_stats[’opponents_taken_pieces’] + \
                      3 ∗ board_stats[’pieces_on_board’] + float(board_stats[’sum_distances_to_endzone’]) / 6

        return board_value
```

Explanation of Novel Feature

Comparison of 5 Best Weighting Functions

Player Comparisons

□