Liam Tucker

CS 565 Assignment 1

Dr. Anderson

02/08/23

Additional Code

At the end of this report, I include the text of letucker.py, which contains my final weights, and the code of the function assess_board, which I modified to account for my custom attribute. I will submit alongside but not include in this report four additional .py files: determine_weights.py, custom_strategy.py, weight_experiment.py, and random_restart.py. I used each of these files in determining my optimal weights. I will briefly describe the function of each of these files.

determine_weights.py is a script I used as my main file. I create objects from my other custom classes in this script.

weight_experiment.py is nearly equivalent to experiment.py, which we were provided. My slight modifications only (1) returned a bool variable of which strategy won the experiment and (2) modified the print statements in the experiment.

custom_strategy.py contains my custom_player class. Each custom_player contains an array of weights as an attribute, used in its evaluate_board function. I also created functionality to run an experiment inside of the custom_player class, which entails functions for running a series of games, functions for creating new weights, and functions for determine which set of weights to keep.

random_restart.py contains the definition of the random_restart class, which is a more formalized way of running an experiment. It essentially implements a local search, using ideas from simulated annealing and random restart hill climbing, to formally determine candidates for the optimal weights.

Custom Attribute

I chose “largest_blockade” as my custom feature. Similar to “number_occupied_spaces”, which counts the number of spaces on which one’s player has two or more pieces (making it inaccessible to the opponent), “largest_blockade” is the value for the largest number of consecutive occupied spaces. This game factor will correlate with being in a good position (i.e. the larger “largest_blockade”, the smaller one’s utility) for a few reasons.

First, it accords with opening theory, which says that some of the best starting rolls are those which can create a blockade. Secondly, a large blockade restricts the opponent’s movement. When two or more consecutive rolls are inaccessible to the opponent, they will be less likely to be able to pass your blockade. A blockade is particularly effective when the opponent has two or more pieces who need to pass an opponent’s blockade, because for both pieces to pass the blockade together the opponent must roll a high enough value for one piece to pass the blockade on both dice one roll. The value of a blockade increases at a faster than linear rate (i.e. the difference between a blockade of 2 and a blockade of 3 is larger than the difference between a blockade of 1 and a blockade of 2), but squaring the largest_blockade seemed not to fit the form of other factors.

Finally, blockades have a snowball effect. That is to say, having a larger blockade makes it easier for one to add other pieces to the blockade. In turn, as spaces in the blockade have 3 or more pieces, it becomes easier to expand the blockade. By rewarding blockades in the weighting function, we increase the probability that our player builds a small or eventually large blockade. Most of the other attributes either describe the current state of the game (i.e. sum_distances, sum_distances_opponents) or the possibility for short-term gains (i.e. number_of_singles, opponents_taken_pieces). Blockades, on the other hand, are more of a long-term strategy; they don’t offer immediate improvements but create an environment where dominant positions could be reached. Thus, I hope that, rather than replace some of the predictive power of existing attributes, blockades will complement them.

Anecdotally, the best strategy my dad and I have developed when playing backgammon is to develop a large blockade. Somewhere between 25-50% of games involve one (or both) players establishing a large blockade (4+ spaces) at some point during the game, and the player to establish the large blockade wins roughly 75% of games.

Strategy Comparison

To develop these weights, I used an adapted form of random-restart hill climbing. I will describe the process for developing weights excluding my custom attribute, but the process was identical when including my custom attribute.

To begin, I created a custom_player class with an array of weights as an attribute. Its evaluation function is calculated by multiplying each weight in its weights array by the corresponding game state attribute. In roughly half or the roughly 20 restarts, the custom_player's initial weights were random values in the range [-1,1]. In the other half, the initial values were the weights used in the CompareAllMovesWeightingDistance strategy. When including my custom attribute, its weight starts at 0, as it is not a part of CompareAllMovesWeightingDistance.

Each restart consisted of 50 iterations. Each iteration began by generating an array of values in the range [-1,1] of the same length as the custom_player's weights attribute. I multiplied each value in the update_values array by a compression factor, which is 1 during the first iteration and 0.2 during the last. Then, a new custom_player is created whose weights are equal to the current custom_player's plus the update_values array. (If the updated weight is slated to be greater than 1, I instead move the old value X% of the way to 1, where X is a random value in the range [0, 1]. I do the same when the updated weight is slated to be less than -1).

The new player then plays a 269 game series against CompareAllMovesWeightingDistance. I chose 269 games because that's the minimum number of games where a strategy with a 55% true win percentage has at least a 95% chance of winning the series. The result of each series is recorded. If the new player won more games than the old player, then I keep the new player as the current player for the next iteration. Otherwise, the new player is deleted.

Initially, I implemented a form of simulated annealing in the iteration process. That is, with decreasing probability as the number of iterations increase and as the candidate strategy is worse, we may replace the current strategy with a worse one. Ultimately, I found that simulated annealing was not improving the final weights. This is largely because I updated each weight by independent random values in the range [-1,1] each iteration. The benefit of simulated annealing is in escaping local but not global minimums, but the algorithm I implemented was already capable of escaping local minimums.

After the 50 iterations are complete, I take the best strategy produced during this restart and record it. From this pool of champion weights, I selected the five best arrays of weights. The following tables list the weights for each of the final five contender strategies for each category (with or without my custom attribute), in addition to their success in longer series against ComapreAllMovesWeightingDistance and MoveFurthestBackStrategy:

Table 1: Details of Strategies Without Custom Attribute

Attribute	Strategy 1	Strategy 2	Strategy 3	Strategy 4	Strategy 5	Simple Ave
number_occupied_spaces	-0.579	-0.518	-0.926	-0.678	-0.556	-0.651
opponents_taken_pieces	-0.194	0.67	-1	-0.888	-0.49	-0.38
sum_distances	0.998	0.329	1	0.287	-0.327	0.457
sub_distances_opponent	-0.379	-0.971	-0.067	-0.459	-0.761	-0.527
number_of_singles	-0.49	0.942	0.202	0.678	0.37	0.34
sum_single_distance_away_from_home	0.092	0.396	0.149	0.665	0.825	0.425
pieces_on_board	0.937	0.082	1.227	0.589	0.662	0.699
sum_distances_to_endzone	0.056	0.138	0.154	0.119	0.12	0.117
Win percent against CompareAllMovesWeightingDistance (2501 games)	64.93%	58.02%	55.90%	59.10%	54.33%	58.74%

Win percent against MoveFurthestBackStrategy (2501 games)	95.48%	86.13%	85.65%	90.64%	85.25%	88.60%

Table 2: Details of Strategies With Custom Attribute

Attribute	Strategy 6	Strategy 7	Strategy 8	Strategy 9	Strategy 10	Simple Ave
number_occupied_spaces	-0.266	-0.395	-0.799	-0.971	-0.981	-0.682
opponents_taken_pieces	0.501	-0.238	-0.977	-0.985	-0.786	-0.497
sum_distances	0.289	0.947	0.295	0.736	0.711	0.596
sub_distances_opponent	-0.924	-0.677	-0.473	-0.968	-0.98	-0.804
number_of_singles	-0.946	-0.828	0.706	-0.56	0.392	-0.247
sum_single_distance_away_from_home	0.355	0.303	0.082	0.416	0.785	0.388
pieces_on_board	0.579	0.859	0.146	0.99	0.221	0.559
sum_distances_to_endzone	0.01	0.075	0.381	0.158	0.42	0.209
largest_blockade	0.897	0.272	-0.686	0.402	0.464	0.27
Win percent against CompareAllMovesWeightingDistance (2501 games)	62.02%	62.42%	62.02%	56.34%	57.01%	56.90%
Win percent against MoveFurthestBackStrategy (2501 games)	92.72%	93.52%	93.76%	92.28%	89.80%	90.68%

I also calculated the win percentage in a series for the weighted average of the five contender weights, weighted by wins in the 269 game series; they were very similar to the simple averages.

These weights generally performed slightly worse in this series of matches than they did during the iterative weight-determination process. This accords with my expectations; because I only chose the best weights from the weight-determination process, the chosen strategies are more likely to have overperformed during the 269 game sample, so their result in the 2501 game series will likely be slightly lower.

I chose the best weights against CompareAllMovesWeightingDistance as my final weights—Strategy 1 for my Player 1 and Strategy 7 for my Player 2. Both outperformed the best strategy provided to us, CompareAllMovesWeightingDistanceAndSinglesWithEndGame2, which won 62.26% of games in a 2501 game series against CompareAllMovesWeightingDistance and 93.24% against MoveFurthestBackStrategy. In a head to head 2501 game series against CompareAllMovesWeightingDistanceAndSinglesWithEndGame2, Strategy 1 won 53.06% of games, while Strategy 7 won 48.06% of games.

I am generally happy with these results. Given more time and computational power and the task of further improving my final weights, I would implement another hill-climbing algorithm, starting at my current optimal weights, with smaller steps and much larger series. A 269 game series isn't ideal; if Strategy A wins 65% of games against CompareAllMovesWeightingDistance and Strategy B wins 55% against ComapreAllMovesWeightingDifference, in a pair of 269 game series, there is a 2% chance Strategy B will win more games. That's not a huge probability, but when I have 50 iterations per restart, the odds of an inferior strategy winning by chance rise. However, I faced computational limitations. Each individual restart took roughly two hours of computational time on my laptop with 50 iterations of a 269 game series. Extending each series would decrease the number of restarts I could run. Given the resource limitations I face, I feel I chose a logical strategy for determining optimal weights.

I'd also like to address that my best strategy involving my custom attribute is less effective than my best strategy ignoring it. I have a few theories for why this may be. First, chance could be a factor. The optimal solution including largest_blockade as an attribute could be better than the optimal solution excluding it, but given a limited period to run my code, I may have failed to find the optimal solution by chance. Second, I designed an board state to track based on the way humans play backgammon; a computer who weights

various game state attributes by constant parameters may not be as effective at winning with a blockade as a human would be, which means a blockade would not be valuable. Third, much of the information for largest_blockade is already stored within number_occupied_spaces. (That is, a game state with a largest_blockade of four must have at least four, but at most seven, as the number_occupied_spaces attribute). This should create a dependency between number_occupied_spaces and largest_blockade where the optimal value of number_occupied_spaces is different depending on whether I include the largest_blockade attribute in this weighting (and vice versa). In other words, largest_blockade should offer more narrow improvements than a fully independent game state attribute, meaning we should expect a lower probability of optimally utilizing it in any given period of time.

Part 3

State 1: **12W14 14W19** (12W14 means the white piece at initial position 12 was moved to position 14)
Note that State 1 can also be reached by **12W17 17W19** or **17W19 12W17**
State changes

a. Opponent Roll [1,6]

Resulting state chg	**Utility value**
1.**13B7 7B6**	140.67
2. **13B7 15B14 OR 15B14 13B7**	140.67
3.**15B14 14B8 OR 15B9 9B8**	140.67

b. Opponent Roll [3,5]

Resulting state chg	**Utility value**
1._13B10 15B10 OR 15B10 13B10	141
2. **13B10 13B8 OR 13B8 13B10**	141
3. **15B10 10B7**	141

c. Opponent Roll [2,3]

Resulting state chg	**Utility value**
1._**13B11 13B10 OR 13B10 13B11**	140
2. **13B11 11B8**	140
3. **13B10 15B13 OR 15B13 13B10**	140

d. Opponent Roll [1,2]

Resulting state chg	**Utility value**
1._**13B11 11B10**	139.33
2. **13B11 15B14 OR 15B14 13B11**	139.33
Note that there is no third legal move which uses both rolls, and it is not legal to voluntarily forfeit a die move, so these are the only two possible moves.	

Average utility: 140.25.

State 2: **12W17 12W14** (can also be reached by **12W14 12W17**)
State changes

a. Opponent Roll [1,6]

Resulting state chg	**Utility value**
1.**13B7 7B6**	150.67
2. **13B7 15B14 OR 15B14 13B7**	180.67
3.**15B14 14B8**	180.67
4. **15B9 9B8**	150.67

b. Opponent Roll [3,5]

Resulting state chg	**Utility value**
1._13B10 15B10 OR 15B10 13B10	151
2. **13B10 13B8 OR 13B8 13B10**	151
3. **15B10 10B7**	151

c. Opponent Roll [2,3]

Resulting state chg	**Utility value**
1._**13B11 13B10 OR 13B10 13B11**	150
2. **13B11 11B8 OR 13B10 10B8**	150
3. **13B10 15B13 OR 15B13 13B10**	150

d. Opponent Roll [1,2]

Resulting state chg	**Utility value**
1._**13B11 11B10**	149.33
2. **13B11 15B14 OR 15B14 13B11**	179.33
Note that there is no third legal move which uses both rolls, and it is not legal to voluntarily forfeit a die move, so these are the only two possible moves.	

Average: 165.25

Under the conditions of this problem (only pieces beginning on spaces 7-18, only the dice rolls listed here), there are two possible distinct moves my player can make. Regardless of what my opponent rolls, I will be better off moving to State 1 than moving to State 2, assuming my opponent aims to minimize my utility with their move. Thus, my move of choice is **12W14 14W19**. (This matches with my intuition—what I would play in this position given the restrictions on which of my pieces I can move).

Source Code

The text of letucker.py and the function assess_board from compare_all_moves_strategy.py follow.

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

class player1_letucker(CompareAllMoves):


            # '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,


    def evaluate_board(self, myboard, colour):
        board_stats = self.assess_board(colour, myboard)
        params = [-0.578612945, -0.194372013, 0.998351437, -0.379138398, -0.48973386,
0.091891502, 0.937047711, 0.055684155]

        board_value = board_stats['number_occupied_spaces'] * params[0] + \
                      board_stats['opponents_taken_pieces'] * params[1] + \
                      board_stats['sum_distances'] * params[2] + \
                      board_stats['sum_distances_opponent'] * params[3] + \
                      board_stats['number_of_singles'] * params[4] + \
                      board_stats['sum_single_distance_away_from_home'] * params[5] + \
                      board_stats['pieces_on_board'] * params[6] + \
                      board_stats['sum_distances_to_endzone'] * params[7]
        return board_value

class player2_letucker(CompareAllMoves):


            # 'NOVEL_FEATURE': novel_feature_value,
            # '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,


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

        params = [-0.970537182, -0.984568516, 0.736305713, -0.96755707, -0.559878932,
        0.416247912, 0.990162973, 0.158104991, 0.401819125]

        board_value = board_stats['number_occupied_spaces'] * params[0] + \
                      board_stats['opponents_taken_pieces'] * params[1] + \
```

```
                board_stats['sum_distances'] * params[2] + \
                board_stats['sum_distances_opponent'] * params[3] + \
                board_stats['number_of_singles'] * params[4] + \
                board_stats['sum_single_distance_away_from_home'] * params[5] + \
                board_stats['pieces_on_board'] * params[6] + \
                board_stats['sum_distances_to_endzone'] * params[7] + \
                board_stats['largest_blockade'] * params[8]

    return board_value
```

```
def assess_board(self, colour, myboard):
        print("Inside asseess_board")
        pieces = myboard.get_pieces(colour)
        pieces_on_board = len(pieces)
        sum_distances = 0
        number_of_singles = 0
        number_occupied_spaces = 0
        sum_single_distance_away_from_home = 0
        sum_distances_to_endzone = 0
        longest_blockade = 0
        current_blockade = 0
        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
        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:
                    number_occupied_spaces = number_occupied_spaces + 1
            if len(pieces) > 1 and pieces[0].colour == colour:
                current_blockade += 1
                if longest_blockade > current_blockade:
                    longest_blockade = current_blockade
            else:
                current_blockade = 0

        opponents_taken_pieces = len(myboard.get_taken_pieces(colour.other()))
        opponent_pieces = myboard.get_pieces(colour.other())
        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,
            'largest_blockade': longest_blockade
        }
```