AB_Custom:

Description:

First heuristic function returns number of the player's moves minus doubled number of opponent's moves.

Implementation:

```python
def custom_score(game, player):
    """Calculate the heuristic value of a game state from the point of view
    of the given player.

    This should be the best heuristic function for your project submission.

    Note: this function should be called from within a Player instance as
    `self.score()` -- you should not need to call this function directly.

    Parameters
    ----------
    game : `isolation.Board`
        An instance of `isolation.Board` encoding the current state of the
        game (e.g., player locations and blocked cells).

    player : object
        A player instance in the current game (i.e., an object corresponding to
        one of the player objects `game.__player_1__` or `game.__player_2__`.)

    Returns
    -------
    float
        The heuristic value of the current game state to the specified player.
    """
    return float(len(game.get_legal_moves(player))-2*len(game.get_legal_moves(game.get_opponent(player))))
```

Results & Analysis:

This scoring function is easy to implement and does not require significant computational power. The coring function formula is: "#_my_moves – 2*#_opponent_moves". The quotient "2" before the number of opponent moves makes the player to behave more aggressively and try to actively reduce the number of opponent's moves. The player equipped with this scoring function demonstrates decent performance and can occasionally outperform the AB_Improved heuristics. In 3 out of 10 game sessions, the winning percentage of AB_Custom was higher than results of AB_Improved function.

The key weakness of this function is its "blindness" to the positional situation of the game.

The screenshots below demonstrate the results of several matches.

Match #	Opponent	AB_Improved Won	AB_Improved Lost	AB_Custom Won	AB_Custom Lost
1	Random	10	0	9	1
2	MM_Open	5	5	7	3
3	MM_Center	8	2	6	4
4	MM_Improved	7	3	7	3
5	AB_Open	4	6	6	4
6	AB_Center	6	4	4	6
7	AB_Improved	5	5	3	7
	Win Rate:	64.3%		60.0%	

Match #	Opponent	AB_Improved Won	AB_Improved Lost	AB_Custom Won	AB_Custom Lost
1	Random	9	1	7	3
2	MM_Open	7	3	7	3
3	MM_Center	10	0	9	1
4	MM_Improved	5	5	7	3
5	AB_Open	4	6	5	5
6	AB_Center	6	4	6	4
7	AB_Improved	3	7	3	7
	Win Rate:	62.9%		62.9%	

Match #	Opponent	AB_Improved Won	AB_Improved Lost	AB_Custom Won	AB_Custom Lost
1	Random	10	0	10	0
2	MM_Open	8	2	7	3
3	MM_Center	6	4	10	0
4	MM_Improved	9	1	6	4
5	AB_Open	6	4	7	3
6	AB_Center	6	4	6	4
7	AB_Improved	6	4	7	3
	Win Rate:	72.9%		75.7%	

Match #	Opponent	AB_Improved Won	AB_Improved Lost	AB_Custom Won	AB_Custom Lost
1	Random	10	0	10	0
2	MM_Open	8	2	8	2
3	MM_Center	9	1	7	3
4	MM_Improved	6	4	6	4
5	AB_Open	3	7	2	8
6	AB_Center	6	4	2	8
7	AB_Improved	4	6	7	3
	Win Rate:	65.7%		60.0%	

AB_Custom_2:

Description:

The idea of this method is to analyze the game one step ahead. The algorithm is looping through the moves available in current situation and estimates the ratio of “#_my_moves” to “#_opponent_moves”. The function returns the average ratio for all available moves.

Implementation:

```python
def custom_score_2(game, player):
    """Calculate the heuristic value of a game state from the point of view
    of the given player.

    Note: this function should be called from within a Player instance as
    `self.score()` -- you should not need to call this function directly.

    Parameters
    ----------
    game : `isolation.Board`
        An instance of `isolation.Board` encoding the current state of the
        game (e.g., player locations and blocked cells).

    player : object
        A player instance in the current game (i.e., an object corresponding to
        one of the player objects `game.__player_1__` or `game.__player_2__`.)

    Returns
    -------
    float
        The heuristic value of the current game state to the specified player.
    """
    # TODO: finish this function!
    n = len(game.get_legal_moves(player))
    if(n==0):
        return float("-inf")
    s = 0
    for move in game.get_legal_moves(player):
        opponenet_moves = len(game.forecast_move(move).get_legal_moves(game.get_opponent(player)))
        if(opponenet_moves==0):
            return float("inf")
        else:
            s+=(len(game.forecast_move(move).get_legal_moves(player))/opponenet_moves)

    return float(s/n)
```

Results & Analysis:

Surprisingly, analyzing the game one step deeper does not improve the overall performance. In the series of ten matches this scoring function outperformed AB_Improved just once.

In this type of games it is more preferred to let the minimax algorithm to go one level deeper instead of incorporating the deepening functionality into the scoring function. It makes even less practical sense in this case since it decreases the performance.

Few examples of the game results:

Match #	Opponent	AB_Improved Won	AB_Improved Lost	AB_Custom Won	AB_Custom Lost	AB_Custom_2 Won	AB_Custom_2 Lost
1	Random	10	0	8	2	7	3
2	MM_Open	8	2	6	4	6	4
3	MM_Center	9	1	8	2	8	2
4	MM_Improved	6	4	6	4	3	7
5	AB_Open	3	7	7	3	4	6
6	AB_Center	4	6	5	5	4	6
7	AB_Improved	5	5	4	6	4	6
	Win Rate:	64.3%		62.9%		51.4%	

Match #	Opponent	AB_Improved Won	AB_Improved Lost	AB_Custom Won	AB_Custom Lost	AB_Custom_2 Won	AB_Custom_2 Lost
1	Random	8	2	8	2	10	0
2	MM_Open	8	2	7	3	7	3
3	MM_Center	7	3	9	1	7	3
4	MM_Improved	7	3	8	2	3	7
5	AB_Open	3	7	5	5	3	7
6	AB_Center	6	4	6	4	5	5
7	AB_Improved	7	3	6	4	4	6
	Win Rate:	65.7%		70.0%		55.7%	

Match #	Opponent	AB_Improved Won	AB_Improved Lost	AB_Custom Won	AB_Custom Lost	AB_Custom_2 Won	AB_Custom_2 Lost
1	Random	9	1	7	3	10	0
2	MM_Open	7	3	7	3	7	3
3	MM_Center	10	0	9	1	8	2
4	MM_Improved	5	5	7	3	7	3
5	AB_Open	4	6	5	5	5	5
6	AB_Center	6	4	6	4	6	4
7	AB_Improved	3	7	3	7	5	5
	Win Rate:	62.9%		62.9%		68.6%	

AB_Custom_3:

Description:

This scoring function combines the “aggressive” behavior of “AB_Custom” function with analysis of positional strength. The method finds a less occupied cluster on the board and estimates the distance between the player’s position and center of that cluster. It uses the formula from “AB_Custom” function as a basis (“#_my_moves – 2*#_opponent_moves”), with addition of a “positional component”.

The distance between the player and the center of less occupied cluster is defined as:

```
distance = abs(player_location[0]-cluster_center_x)
         + abs(player_location[1]-cluster_center_y)
```

Manhattan distance is used to reduce the computational load.

The positional score is defined as:

```
score = (game_height+game_width-distance)/(game_height+game_width)
```

The score is higher when the player is closer to the cluster's center.

The overall score is a simple sum between difference of legal moves of two opponents and the positional score.

In cases when several potential moves have the same estimate of move difference, the minimax algorithm will select the move which brings the player closer to the less occupied part of the board.

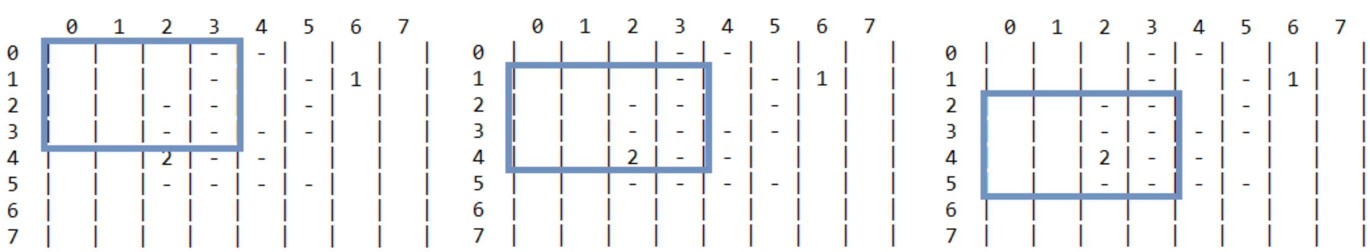
The positional score is calculated only when more than 20% of the board is occupied. Otherwise it returns the same value as "AB_Custom" function.

Less occupied cluster search:

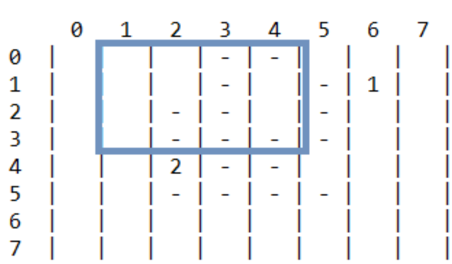
The cluster size is defined as a quarter of the board.

The algorithm consecutively analyses all clusters on the board and selects the less occupied one.

It starts in top left corner and moves down until the bottom border of the cluster is aligned with the bottom of the board.



Then, it shifts one position to the right and repeats the procedure.



In this example, the algorithm would select the cluster in the bottom right with the center in cell (6,6):

```
    0   1   2   3   4   5   6   7
0 |   |   |   | - | - |   |   |   |
1 |   |   |   | - |   | - | 1 |   |
2 |   |   | - | - |   | - |   |   |
3 |   |   | - | - | - | - |   |   |
4 |   |   | 2 | - | - |   |   |   |
5 |   |   | - | - | - | - |   |   |
6 |   |   |   |   |   |   |   |   |
7 |   |   |   |   |   |   |   |   |
```

Implementation:

```python
def custom_score_3(game, player):
    game_height = game.height
    game_width  = game.width
    brd = game._board_state
    if(game._board_state.count('1')<(game_height*game_width/5)):
       return float(len(game.get_legal_moves(player))-2*len(game.get_legal_moves(game.get_opponent(player))))

    cluster_height = round(game.height/2)
    cluster_width = round(game.width/2)
    l = len(game._board_state)-3-cluster_height - game_height*(cluster_width-1)

    i = 0
    column_number = 0
    min_open = game_height*game_width
    min_cluster_pos_x = -1
    min_cluster_pos_y = -1

    while i<=l:
        val_sum = 0
        for k in range(0,cluster_width):
            for n in range(0,cluster_height):
                j = i+n+k*game_height
                val = game._board_state[j]
                val_sum+=val

        if(val_sum<min_open):
            min_open=val_sum
            min_cluster_pos_x = i-column_number*game_height
            min_cluster_pos_y = column_number


        if(i==(game_height-cluster_height+game_height*column_number)):
            column_number+=1
            i+=cluster_width
        else:
            i+=1
    cluster_center_x = min_cluster_pos_x+int(cluster_height/2)
    cluster_center_y = min_cluster_pos_y+int(cluster_width/2)

    player_location = game.get_player_location(player)
    distance = abs(player_location[0]-cluster_center_x) + abs(player_location[1]-cluster_center_y)
    score = (game_height+game_width-distance)/(game_height+game_width)
    return float(len(game.get_legal_moves(player))-2*len(game.get_legal_moves(game.get_opponent(player)))+score)
```

Results:

The player agent which employs the custom_score_3 function performed better than “AB_Improved” in eight cases out of ten. Game results examples:

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	7	3	8	2	5	5	7	3
2	MM_Open	4	6	7	3	2	8	7	3
3	MM_Center	6	4	4	6	7	3	8	2
4	MM_Improved	6	4	8	2	4	6	5	5
5	AB_Open	8	2	4	6	3	7	3	7
6	AB_Center	5	5	6	4	7	3	7	3
7	AB_Improved	5	5	6	4	4	6	6	4
	Win Rate:	58.6%		61.4%		45.7%		61.4%	

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	10	0	8	2	7	3	10	0
2	MM_Open	8	2	6	4	6	4	8	2
3	MM_Center	9	1	8	2	8	2	10	0
4	MM_Improved	6	4	6	4	3	7	6	4
5	AB_Open	3	7	7	3	4	6	6	4
6	AB_Center	4	6	5	5	4	6	7	3
7	AB_Improved	5	5	4	6	4	6	6	4
	Win Rate:	64.3%		62.9%		51.4%		75.7%	

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	8	2	8	2	10	0	9	1
2	MM_Open	8	2	7	3	7	3	8	2
3	MM_Center	7	3	9	1	7	3	10	0
4	MM_Improved	7	3	8	2	3	7	7	3
5	AB_Open	3	7	5	5	3	7	6	4
6	AB_Center	6	4	6	4	5	5	6	4
7	AB_Improved	7	3	6	4	4	6	5	5
	Win Rate:	65.7%		70.0%		55.7%		72.9%	

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	10	0	5	5	5	5	8	2
2	MM_Open	7	3	7	3	6	4	9	1
3	MM_Center	8	2	6	4	7	3	9	1
4	MM_Improved	3	7	3	7	4	6	4	6
5	AB_Open	6	4	5	5	4	6	3	7
6	AB_Center	5	5	6	4	3	7	4	6
7	AB_Improved	6	4	7	3	6	4	4	6
	Win Rate:	64.3%		55.7%		50.0%		58.6%	

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	9	1	10	0	8	2	10	0
2	MM_Open	5	5	7	3	7	3	6	4
3	MM_Center	6	4	8	2	5	5	9	1
4	MM_Improved	4	6	5	5	5	5	5	5
5	AB_Open	5	5	3	7	5	5	7	3
6	AB_Center	5	5	6	4	4	6	5	5
7	AB_Improved	5	5	4	6	6	4	6	4
	Win Rate:	55.7%		61.4%		57.1%		68.6%	

Conclusion:

In circumstances when it is not recommended to analyze deeper levels of the game tree inside of the scoring function it makes sense to analyze the positional strength. Before creating this scoring functions, I reviewed several examples available online. Most implementations analyze position of the player relative to center, edges, or corners of the board without consideration of availability (openness) of that areas. In my “custom_score_3” function, I assumed that the most desired (less occupied) area may be anywhere on the board and it moves during the game.

The testing results show that the player agent, which actively attempts to reduce the number of opponent’s available moves while staying close to unoccupied regions, demonstrates stronger performance results and is able to beat the “AB_Improved” algorithm in most of the matches.