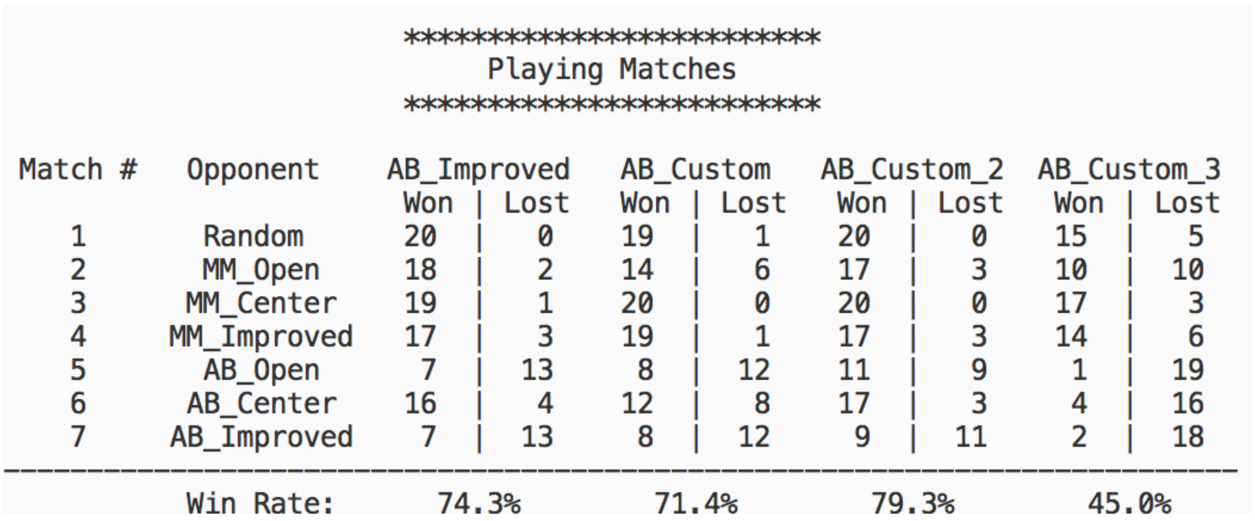
Isolation Heuristic Analysis

Evaluation functions are in integral part to any AI agent. My intent is to keep them fairly simple without adding any unnecessary complexity for complexity's sake.
During my trials I tried to establish a control metric that would return a random valuation for a board state. I wanted to make certain my measures were better than random. Initially I wasn't getting the results I had expected. My random function which would just return `random.random()`, was winning a majority of its games. It wasn't until someone on the slack channel had suggested I was reaching end-game, so It didn't matter what the valuation function returned as the is_winner/is_loser methods were being called.

For these evaluations I bumped the board size to 13 x 13 to give my valuation functions some playing time, along with increasing the matches count to provide a more continuous calculation to the win rate and was rather surprised as my previous best heuristic turned out to be inferior to my simplest one.

These runs take a considerable amount of time so for brevity I ran this extended run once. For the submission I will be swapping the AB_Custom & AB_Custom_2 functions.



```
                         *************************
                              Playing Matches
                         *************************

 Match #   Opponent    AB_Improved   AB_Custom    AB_Custom_2  AB_Custom_3
                        Won | Lost   Won | Lost   Won | Lost   Won | Lost
    1       Random      20  |   0    19  |   1    20  |   0    15  |   5
    2       MM_Open     18  |   2    14  |   6    17  |   3    10  |  10
    3      MM_Center    19  |   1    20  |   0    20  |   0    17  |   3
    4     MM_Improved   17  |   3    19  |   1    17  |   3    14  |   6
    5       AB_Open      7  |  13     8  |  12    11  |   9     1  |  19
    6      AB_Center    16  |   4    12  |   8    17  |   3     4  |  16
    7     AB_Improved    7  |  13     8  |  12     9  |  11     2  |  18
--------------------------------------------------------------------------
           Win Rate:      74.3%        71.4%        79.3%        45.0%
```

Worthy of noting is I do have some ideas on how to improve my win rates, at this juncture it's time to call it good enough for the project as there was a lot of time spent wading through the weeds to get it even working.

AB_Custom: 71.4% Win rate

```
def custom_score(game, player):

    if game.is_loser(player):
        return float("-inf")

    if game.is_winner(player):
        return float("inf")

    c = int(game.width / 2) + 1
    board_center = (c, c)

    pl_moves = len(game.get_legal_moves(player))
    op_moves = len(game.get_legal_moves(game.get_opponent(player)))

    avail_moves = game.get_legal_moves(player)
    total_moves = len(avail_moves)

    total_distance_from_ctr = 0

    for move in avail_moves:
        mx, my = move
        cx, cy = board_center[0], board_center[1]

        total_distance_from_ctr += np.sqrt((mx-cx)**2 + (my-cy)**2)

    # I want to penalize board states that have a lot of moves around the perimeter
    # So I'm multiplying the inverse of the mean distance to the center by the number
    # of moves we have.
    if total_moves > 0 and total_distance_from_ctr > 0:
        return (1. / float(total_distance_from_ctr / total_moves)) * float(pl_moves -
op_moves)
    else:
        return float("-inf")
```

This one didn't perform as good as I had hoped. I was trying a two-pronged approach to this one. I wanted to score a board states based on not only the number of moves my agent has available but want to weight the states that have center moves more favorably.

It works better than just looking at center moves entirely, but not as well as just the total number of moves in my customer_score_2.
I don't have the coding ability (yet) or time to test this; however, I hypothesize this function is discounting those game states where my agent can possibly win but most of the center moves have been taken.

AB_Custom_2: 79.3% Win rate

```
def custom_score_2(game, player):
    if game.is_loser(player):
        return float("-inf")

    if game.is_winner(player):
        return float("inf")

    pl_moves = len(game.get_legal_moves(player))
    op_moves = len(game.get_legal_moves(game.get_opponent(player)))

    return float(pl_moves - op_moves)
```

This performed the best out of the functions I put together and tested myself. It performs better than my expected best metric against the AB_Center sample player. Because of its simplicity, it's not under-valuing some scenarios.

AB_Custom_3 Win Rate 45%

```
def custom_score_3(game, player):
    if game.is_loser(player):
        return float("-inf")

    if game.is_winner(player):
        return float("inf")

    c = int(game.width / 2) + 1
    board_center = (c, c)

    avail_moves = game.get_legal_moves(player)
    total_moves = len(avail_moves)

    total_distance_from_ctr = 0

    for move in avail_moves:
        mx, my = move
        cx, cy = board_center[0], board_center[1]

        total_distance_from_ctr += np.sqrt((mx-cx)**2 + (my-cy)**2)

    if total_moves > 0 and total_distance_from_ctr > 0:
        return float(total_distance_from_ctr / total_moves)
    else:
        return float("-inf")
```

For the third one I wanted to see if a heuristic that doesn’t use a variant on total moves would be viable. Unfortunately, this one suffers the same short fall as my first custom_score.