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.

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

Match #   Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
                      Won  | Lost   Won  | Lost   Won  | Lost   Won  | Lost
   1       Random      40  |   0    40  |   0    40  |   0    35  |   5
   2       MM_Open     34  |   6    38  |   2    34  |   6    14  |  26
   3      MM_Center    39  |   1    40  |   0    40  |   0    30  |  10
   4     MM_Improved   33  |   7    37  |   3    39  |   1    17  |  23
   5       AB_Open     21  |  19    29  |  11    33  |   7    10  |  30
   6      AB_Center    34  |   6    38  |   2    36  |   4    11  |  29
   7     AB_Improved   22  |  18    28  |  12    28  |  12     2  |  38
------------------------------------------------------------------------
          Win Rate:      79.6%        89.3%        89.3%        42.5%
```

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.

Heuristic 1

The code

AB_Custom: 89.3% Win rate.

```
def custom_score(game, player):
```

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

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


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

    pl_future_moves = float(sum([len(game.__get_moves__(move)) for move in pl_moves ]))
    op_future_moves = float(sum([len(game.__get_moves__(move)) for move in op_moves ]))

    return pl_future_moves - op_future_moves + len(pl_moves) - len(op_moves)
```

Summary

With the AB_Improved standard heuristic performing so well, I thought how I could capitalize on it further. I decided to incorporate this heuristic for subsequent board states.

It performed admirably against AB_Improved. My hypothesis is that this heuristic doesn't get trapped into losing board states due to its crude foresight; effectively, its more long-term focused than short term as AB_improved is. I don't personally have the coding ability to prove or disprove that hypothesis, however, this is only based on the intuition of the heuristic itself.

In closing

Of the metrics I devised, I recommend this one over the other two approaches. It's reasonably straight forward, performs consistently, and has head room for improvement. My second heuristic scored the same on win rate but involves an average Euclidean Distance amongst all moves that provided no real material benefit.

Heuristic 2

The code

AB_Custom_2: 89.3% Win rate

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

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

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

```
    board_center = (c, c)

    pl_options = game.get_legal_moves(player)
    op_optins = game.get_legal_moves(game.get_opponent(player))

    pl_moves = len(pl_options)
    op_moves = len(op_optins)

    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 PLUS the number of future moves from the board states that
    # terminate from the current board
    pl_future_moves = float(sum([len(game.__get_moves__(move)) for move in pl_options ]))
    op_future_moves = float(sum([len(game.__get_moves__(move)) for move in op_optins ]))

    if total_moves > 0 and total_distance_from_ctr > 0:
        return (1. / float(total_distance_from_ctr / total_moves)) * float(pl_future_moves - op_future_moves + pl_moves -
op_moves)
    else:
        return float("-inf")
```

Summary

This one did precisely as well as my first heuristic, not surprisingly. However, it did have a match of brilliance against the AB_Open opponent, but that advantage was limited to that opponent only. With further experimentation, this metric has promise and I believe could outperform my first metric, alas, I haven't coding skill or the time to do so efficiently. If I did, I would look at computing the average distance from the center for the players available moves for future board states with the future moves.

In closing

As it stands now, it's good and arbitrarily as good as my number one score, but can't recommend purely because of its additional complexity and no real material improvements. If we were to include future states mean distance from center for the available moves along with the future move numbers I think this would have a strong chance at being quite competitive.

Heuristic 2

The code

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")
```

Summary

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 is extremely myopic and doesn't consider future potential board states at all leaving it to be trapped quite frequently in losing board states.

In closing

The inclusion/testing of this heuristic led me to the conclusion that a forward-looking heuristic not a "nice to have" but a need to have to be even remotely competitive.