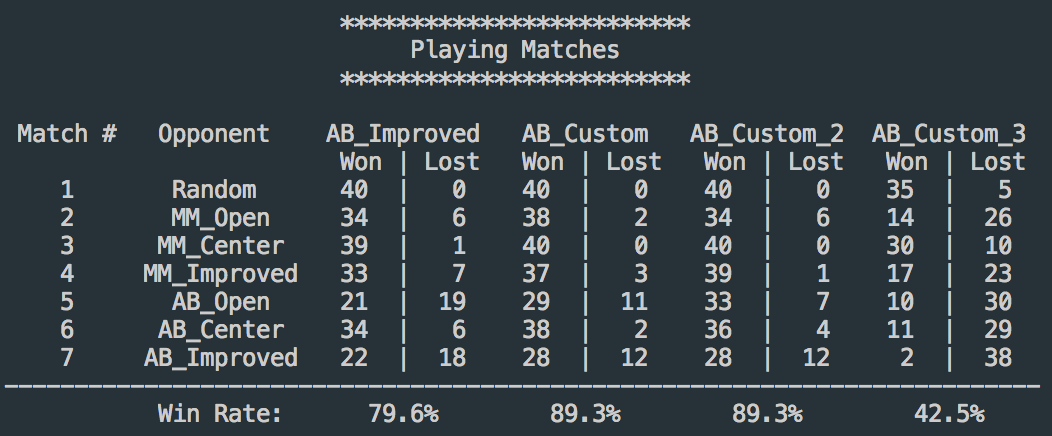
Isolation Heuristic Analysis

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.



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 out perform 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 it given its current state.

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.