Isolation Heuristic Summary

Yoon Namkung

Match results by tournament.py

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

Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
	Won	Lost	Won	Lost	Won	Lost	Won	Lost
Random	9	1	9	1	8	2	6	4
MM_Open	8	2	5	5	5	5	6	4
MM_Center	8	2	8	2	9	1	7	3
M_Improved	4	6	7	3	4	6	9	1
AB_Open	5	5	5	5	4	6	4	6
AB_Center	5	5	5	5	3	7	7	3
B_Improved	4	6	7	3	5	5	5	5
Win Rate:	61.4%		65.7%		54.3%		62.9%	

```
2.0 timeouts during the tournament -- make sure your agent handles
ly, and consider increasing the timeout margin for your agent.

forfeited 246.0 games while there were still legal moves available
```

The best win rate is 65.7% derived from AB_Custom and the least win rate is 54.3% from AB_Custom_2. Average win rate among Custom functions is 60.97%.

Recommendation from the result

- According to final result, AB_Custom shows the best result. Win rate is 65.7% which is the highest score among heuristics. Also, Ab_Custom shows that the highest performance among 4 in battle against AB_Improved. It seems that maintaining score value less than 1 is effective for 'move counting' algorithm. Lastly, it is still very simple to implement.

Heuristic 1

- First heuristic takes number of available moves of own player and opponent player. Then, it calculates difference between two values (player and opponent available moves) and divide the difference by total number of grid. The reason I divided it by total number of grid is that I was curious what will happen if I do not give big value for evaluation score. I guess it is better for this algorithm not to assign huge evaluation score as it shows this way overtakes AB_Improved about 4% win rate.

Implementation

```
if game.is_loser(player):
```

```
    return float("-inf")

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

init_num_moves = int(game.width * game.height)
own_moves = len(game.get_legal_moves(player))
opp_moves = len(game.get_legal_moves(game.get_opponent(player)))

return float((own_moves - opp_moves) / init_num_moves)
```

Heuristic 2

- Second heuristic calculates distance of each player's location from the center of the grid. Instead of using Euclidean distance, I used Manhattan distance for faster computation. Then, I chose the difference value between player and opponent's distance. However it seems that comparing player and opponent distance is not as effective as just calculating player's distance for evaluation score.

Implementation

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

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

w, h = game.width / 2., game.height / 2.
py, px = game.get_player_location(player)
oy, ox = game.get_player_location(game.get_opponent(player))

# Calcuate player's distance from center point using Manhattan distance
player_dist = float(abs(h-py) + abs(w-px))
opp_dist = float(abs(h-oy) + abs(w-ox))

return (player_dist - opp_dist)
```

Heuristic 3

- Third heuristic calculates distance between own player and opponent player using Euclidean distance. In this algorithm, it seems player and opponent plays hide and seek. Maximizer try to apart from other opponent and minimizer tries to catch other player. The result shows that maximizer was a little better for this game.

Implementation

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

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

```
py, px = game.get_player_location(player)
oy, ox = game.get_player_location(game.get_opponent(player))

return float((py-oy)**2 + (px-ox)**2)
```