Heuristic functions

It is not always possible to build the full tree of a game to know the real utility of each decision. In those cases it is necessary to estimate the expected utility of a given position, that is usually as a far as we can get. This value will be calculated by using different features of that particular state. The score from the heuristic function will be used to decide if following that path has more chances to lead to a positive outcome than others.

In the code I have implemented three different heuristic functions. First, let's see a summary of their performance as percentage of victories from tournament.py.

Benchmarks

```
Match #   Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
                        Won | Lost   Won | Lost   Won | Lost   Won | Lost
    1       Random      10  |   0     9  |   1     9  |   1     9  |   1
    2       MM_Open      7  |   3     9  |   1     7  |   3     6  |   4
    3      MM_Center     9  |   1     8  |   2     7  |   3    10  |   0
    4     MM_Improved    7  |   3     8  |   2     7  |   3     8  |   2
    5       AB_Open      4  |   6     5  |   5     6  |   4     4  |   6
    6      AB_Center     7  |   3     6  |   4     6  |   4     5  |   5
    7     AB_Improved    3  |   7     5  |   5     3  |   7     5  |   5
```

Won/Lost Ratio:

- AB_Improved (Udacity): 67.1%
- AB_Custom: **71.4%**
- AB_Custom_2: 64.3%
- AB_Custom_3: 67.1%

From all heuristic functions, AB_Custom is the one with the best Won/Lost ratio, with 71.4% of victories. More than 3% better than the ones in second position.

Number of times defeated by an opponent:

- AB_Improved (Udacity): 2
- AB_Custom: **0**
- AB_Custom_2: 1
- AB_Custom_3: 1

AB_Custom is also the only heuristic function that has never got a negative ratio (more losses than wins)

against an opponent. 5 victories and 2 draws.

Best result against AB_Improved:

- AB_Custom: **Draw**
- AB_Custom_2: Lost
- AB_Custom_3: **Draw**

AB_Custom and AB_Custom_3 have the best results, both having a draw of 5 victories and losses.

The overall results shows that AB_Custom performs better than the other two custom functions, with higher chances to win and less defeats. The results are more consistent against each opponent. Finally, taking in account that it is also computationally simpler to calculate than AB_Custom_3, I choose **AB_Custom as the recommended function**.

AB_Custom (Squared opponent moves)

This heuristic function is based on the idea of the least squares loss function used in Machine Learning. Basically, opponent moves are a metric we want to reduce and the more it has less our chances to win.

This is the code:

```
own_moves = len(game.get_legal_moves(player))
opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
if opp_moves ==0:
    return float(own_moves)
return own_moves / opp_moves**2
```

It returns a score that is the ratio between our player's remaining moves and the opponent's moves squared. The more moves the opponent has, the least possibilities to choice that move. In case the opponent has no more available moves it just returns the number of the moves we have (mostly to avoid a division by zero).

AB_Custom_2 (Least opponent moves)

This function follows a similar goal as the previous one by trying to choose the move that gives the opponent the least amount of possible moves. But in this case it also takes our own player movements in account.

This is the code:

```
own_moves = len(game.get_legal_moves(player))
opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
if own_moves > opp_moves and opp_moves > 0:
    return 1/opp_moves
else:
    return float(own_moves)
```

The heuristic will choose that move where we get more future moves than the opponent and from those, the one where the opponent gets the smaller number of possible moves. So, the idea is to try to reduce opponent's moves faster.

AB_Custom_3 (Have the enemy near)

The idea is to choose, from all possible positive moves, the one that is closer to the opponent's position. Positive moves are the same as in previous heuristic, the ones where our player has more possible movements than the opponent.

This is the code:

```
own_moves = game.get_legal_moves(player)
opp_moves = game.get_legal_moves(game.get_opponent(player))
x, y = game.get_player_location(game.get_opponent(player))

distances = []

if own_moves > opp_moves and len(distances)>0 and opp_moves >0:
    for move in own_moves:
        my_x = move[0]
        my_y = move[1]
        dist = math.hypot(my_x-x, my_y-y)
        distances.append(dist)
    return sum(distances)/len(distances)
else:
    return float(len(own_moves))
```

First the best moves are chosen where "own*moves* > *opp*moves", and from those the one that is is closer to the opponent in euclidean distance returns better score and will be more likely to be chosen.