Project 2 - Game-Playing Agent Heuristic Analysis

When trying to come up with heuristics for my game-agent, I started from the cost function proposed in the lectures, specifically: `f(position) = own_moves(position) - opponent_moves(position)`
Moreover, I tried the *weighted* version of this function, giving a weight to `opponent_moves` and thus making it more aggressive. As it is shown in the below table, this did not work too well over the an entire game, although it does seem that at times, it could produce an advantage.

Therefore, my next approach (not documented in the table, but documented through the code) was to try a conditional approach and divide the game into multiple phases. I though that maybe initially we want to *corner* the opponent, moving on their possible future open positions (if possible), and further towards the end of the game we would want to have more freedom in moving away from the positions close to the opponent (i.e. not necessarily covering their future possible moves).Thus I tried addopting something similar to:

```
if the board is covered less than 20%:
    apply the aggressive weighted cost function
else:
    apply the regular f(position) # basically with weight
```

```
= 1
```

This did not work so well so I moved on, trying to look what else we would want to optimize. I tried looking a bit further, trying to optimize the function from one position to the next. Therefore, I initially tried to store the `# of own moves` from before selecting the new position (through *Iterative Deepening*). My cost function was: `g(position) = (own_moves(position) - own_moves(position - 1)) - opp_moves(position)`.
The reasoning behind this being that, at every new *step* (or position), we want the highest value of this function `g`. In order to achieve this, we would need to maximize the component `(own_moves(position) - own_moves(position - 1))` by making sure that at this *step* we get more moves that at the previous step (**yes, everyone can already see how badly I judged this...**), while also minimizing `opp_moves(position)`. Why was this a bad decision? Pretty simple - as we progress through the game, we will **naturally** have fewer and fewer open moves to choose from, so this function would start breaking.

But this "*fewer and fewer*" argument made me think of the opponent instead. As we progress through the game, we want **the opponent** to have fewer and fewer moves, from step to step (from position to position). Therefore, instead of keeping track of my own previous open moves, I started keeping track of the opponent's previous open moves: `h(position) = own_moves(position) + (opp_moves(position - 1) - opp_moves(position))`.

Therefore, now we try to maximize `h` by maximizing our own moves for this position (`own_moves`, naturally), **and** maximizing the component:
`(opp_moves(position - 1) - opp_moves(position))`
which, by maximizing it, means that we try to minimize the number of moves the opponent will have at this position **in comparison** to what they had in the previous position.

This also did not perform at too high rates, but I felt like I was getting closer. Similar to what I read about in the AlphaGo paper ("similar" might be a far term to use in this context), I thought of trying a hybrid approach, in which I would use a *conditional* approach as I explained above, along with this *previous moves count for opponent* strategy. Moreover, there was another interesting thing that I noticed while looking at win percentages (I highlighted it in the table as well):

> *My agents' win rates against **XX_Open** were horrible, over a variety of tested cost functions.*

Therefore, I came up with the third variation, a conditional approach, where I split the game into 3 phases:

1. In the very few starting moves, I go for the simplest function I know (one that has been beating my agents pretty constantly) - `own_moves(position)`, the simple count of my agent's moves
2. As we progress through the game, we want a function that keeps track of the ration of open moves that my agent has and the actual size of the board - we saw from a previous example

above that it really matters **knowing what are you counting in regards to what absolute value**. By that, I mean that it is useful to know that I have, for example, 5 open moves out of a board of 20 vs out of a board of 7 let's say. Indeed, this heuristic is also taken into account by the way I split the game phases, but still, the ratio helps. Along with this ratio, just like with the other functions, we want to minimize the opponent's number of moves. Therefore, a function that achieves this, while maximizing it when maximizing my agent's number of moves, is:
`F(position) = own_moves(position) / (game.width * game.height + opp_moves(position))`

3. As we get closer to the end game, we start to care about minimizing the opponent's number of moves in regards to their previous position's number of moves. In earlier stages of the game this might be harder to achieve due to the board being empty - it has a bigger impact when there are not that many open moves. Here we used function `h(position)` described above.

Notes: I defined the functions `weighted` because I also tried variations of weights along the heuristics, but they have been evaluated in accordance to what I described above.
The agents were evaluated on a *Mid-2012 MacBook Pro*: 2.5 GHz Intel Core i5, 8GB RAM.

Agent\Heuristic	ID_Improved	WMoveDiff(w=2)	WPrevMoveD
Random	100%	85%	85%

MM_Null	85%	75%	85%
MM_Open	65%	*40%*	*40%*
MM_Improved	60%	15%	60%
AB_Null	90%	75%	65%
AB_Open	50%	*25%*	*50%*
AB_Improved	50%	45%	50%
Overall	**71.43%**	**51.43%**	**62.14%**

In conclusion, I think there are other board parameters that could hold useful information in defining such cost functions, and I will keep tweaking with them in the following days/weeks.