Heuristics analysis for isolation game

Sai Tai

In this project, we defined 3 different heuristics methods to try to reward those moves lead us to win the game and punish moves that lead us to lose the game. Because the the starting positions in the tournament are random, it is hard to get consistent results in one heuristics only. Thus, in this report we will try to focus on comparing 3 different heuristics methods and 2 different searches - alpha-beta search and iterative Deepening in figure out which is the better one than the others.

Environment States Of Isolation Game:

1. Fully observable
2. Deterministic
3. Discrete
4. Adversarial

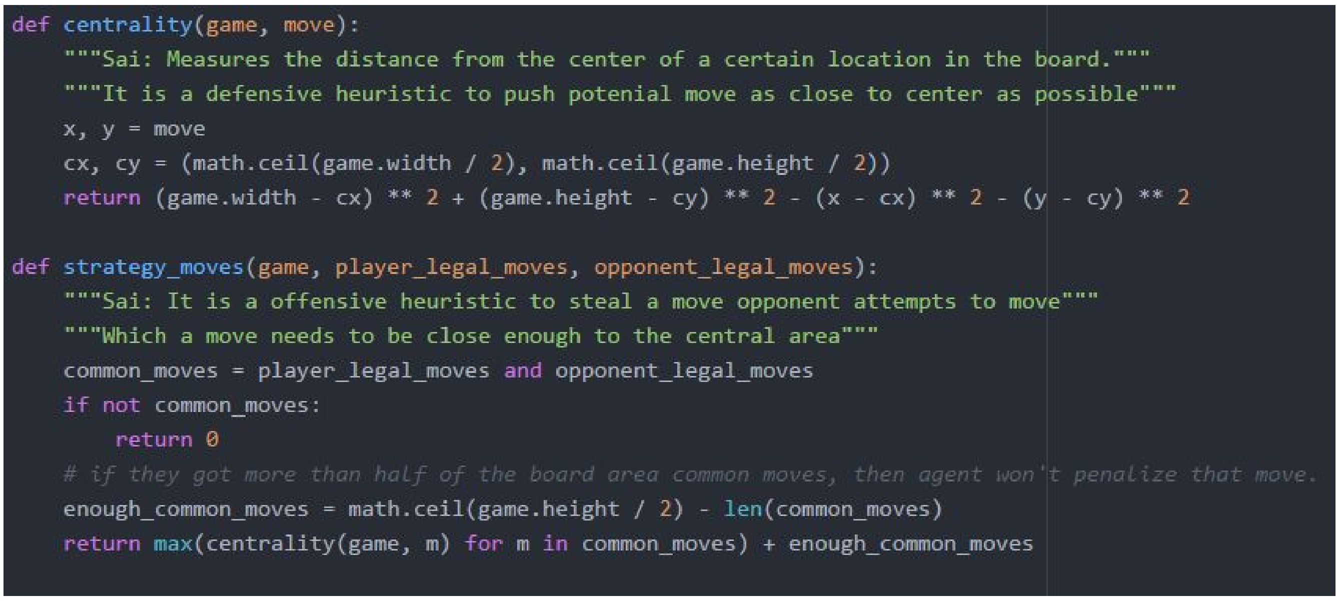
Define the characteristics of the game

Since a heuristic is some additional piece of information that we are gonna solve, we need to figure out the characteristics of the game then investigate how to use these characteristics to make our AI smarter. These are the information that was easily available to avoid long waiting time for each movement calculation.

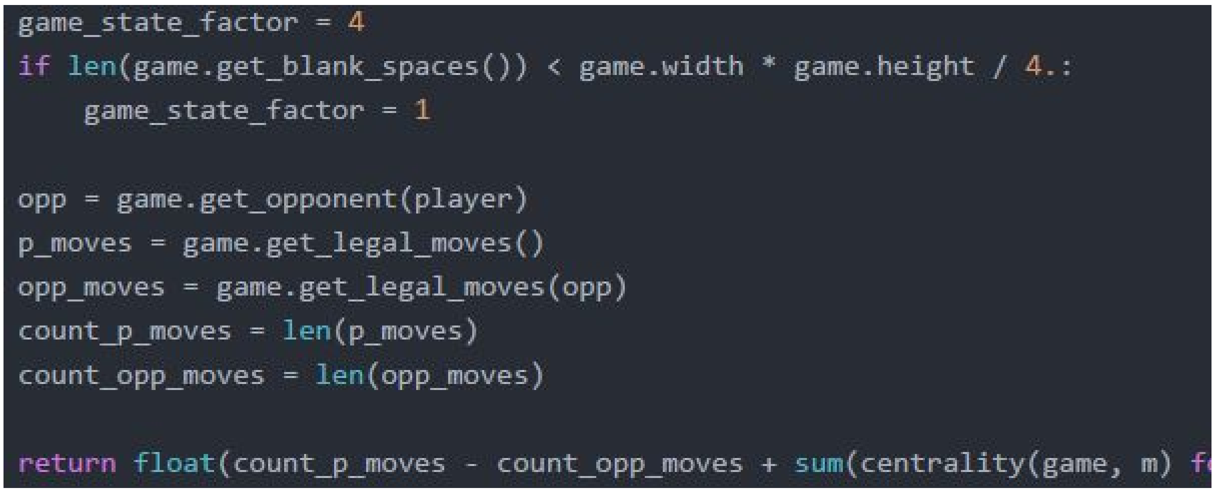
- Number of available moves to each player
    - Penalize moves that lead opponent to get fewer moves
    - Reward moves that lead player to get more moves
- Position of each player
    - Distance to the center (Especially important at the beginning of the game)
    - Distance to the four corner (Especially important at the ending of the game)
- Number of blank spaces on the board
    - Define the progress of the game

3 different heuristics, 3 different strategies

1. custom_score



```
def centrality(game, move):
    """Sai: Measures the distance from the center of a certain location in the board."""
    """It is a defensive heuristic to push potenial move as close to center as possible"""
    x, y = move
    cx, cy = (math.ceil(game.width / 2), math.ceil(game.height / 2))
    return (game.width - cx) ** 2 + (game.height - cy) ** 2 - (x - cx) ** 2 - (y - cy) ** 2

def strategy_moves(game, player_legal_moves, opponent_legal_moves):
    """Sai: It is a offensive heuristic to steal a move opponent attempts to move"""
    """Which a move needs to be close enough to the central area"""
    common_moves = player_legal_moves and opponent_legal_moves
    if not common_moves:
        return 0
    # if they got more than half of the board area common moves, then agent won't penalize that move.
    enough_common_moves = math.ceil(game.height / 2) - len(common_moves)
    return max(centrality(game, m) for m in common_moves) + enough_common_moves
```



```
game_state_factor = 4
if len(game.get_blank_spaces()) < game.width * game.height / 4.:
    game_state_factor = 1

opp = game.get_opponent(player)
p_moves = game.get_legal_moves()
opp_moves = game.get_legal_moves(opp)
count_p_moves = len(p_moves)
count_opp_moves = len(opp_moves)

return float(count_p_moves - count_opp_moves + sum(centrality(game, m) f
```

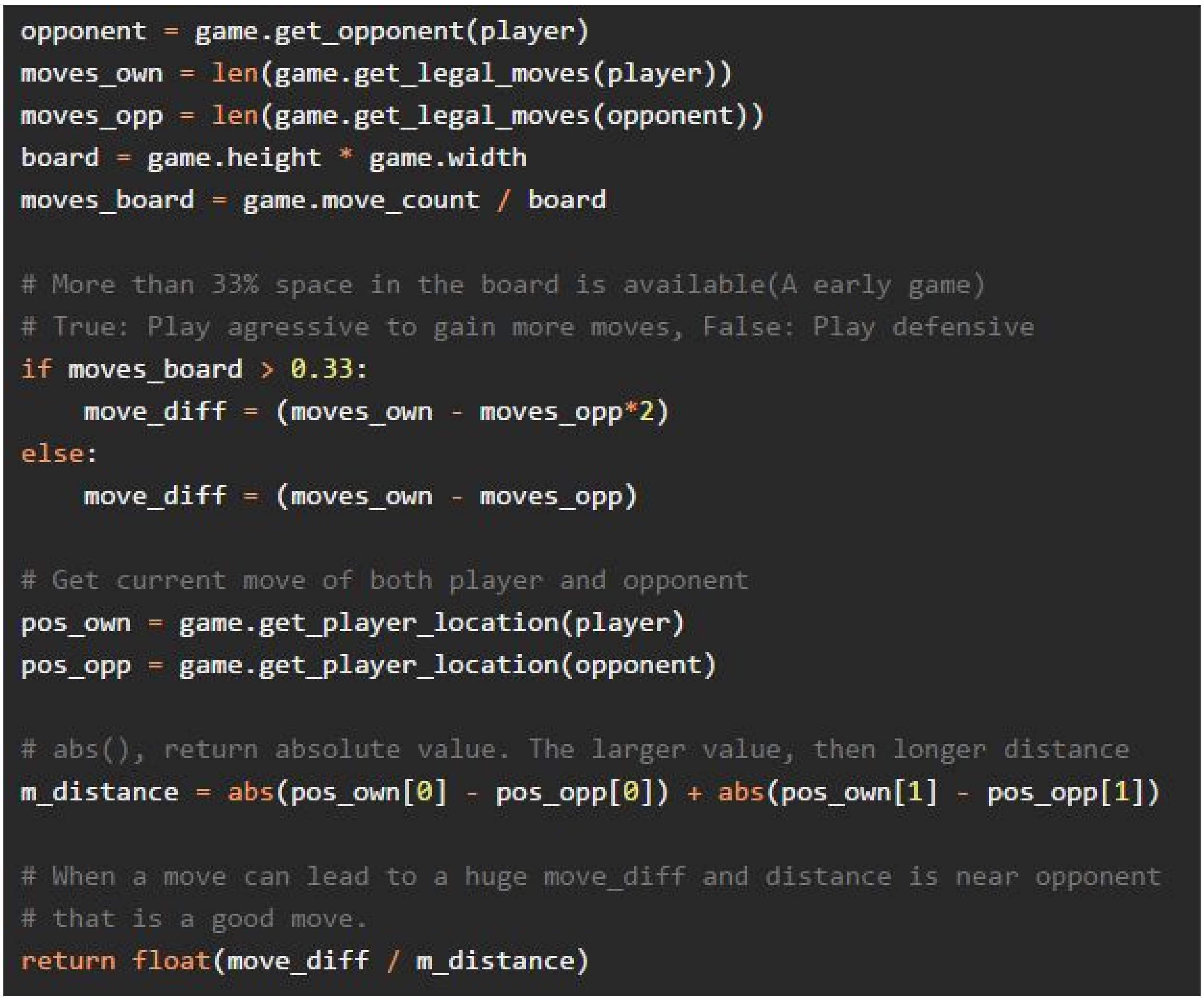
This heuristics tried to get a balance between being offensive and defensive. It tried to find a move leads player has more moves and opponent has fewer moves; a weighted sum of the centrality to encourage player moves towards center in early game; also a function called strategy_moves() to steal potential opponent's moves in the early game.

Formula:
count_p_moves - count_opp_moves
+ sum(centrality(game, m) for m in p_moves)*game_state_factor
+ strategy_moves(game, p_moves, opp_moves))

Pro: This heuristics should be the most balanced agent among these three agents I have created. (Technically)

Con: Longest response time, sometimes even forfeit a game during the time condition. Also, due to its complexity of this heuristics itself, it is so complicated to tune all those parameters in the formula to get a better performance. Even small changes can affect the result completely.

2. custom_score_2



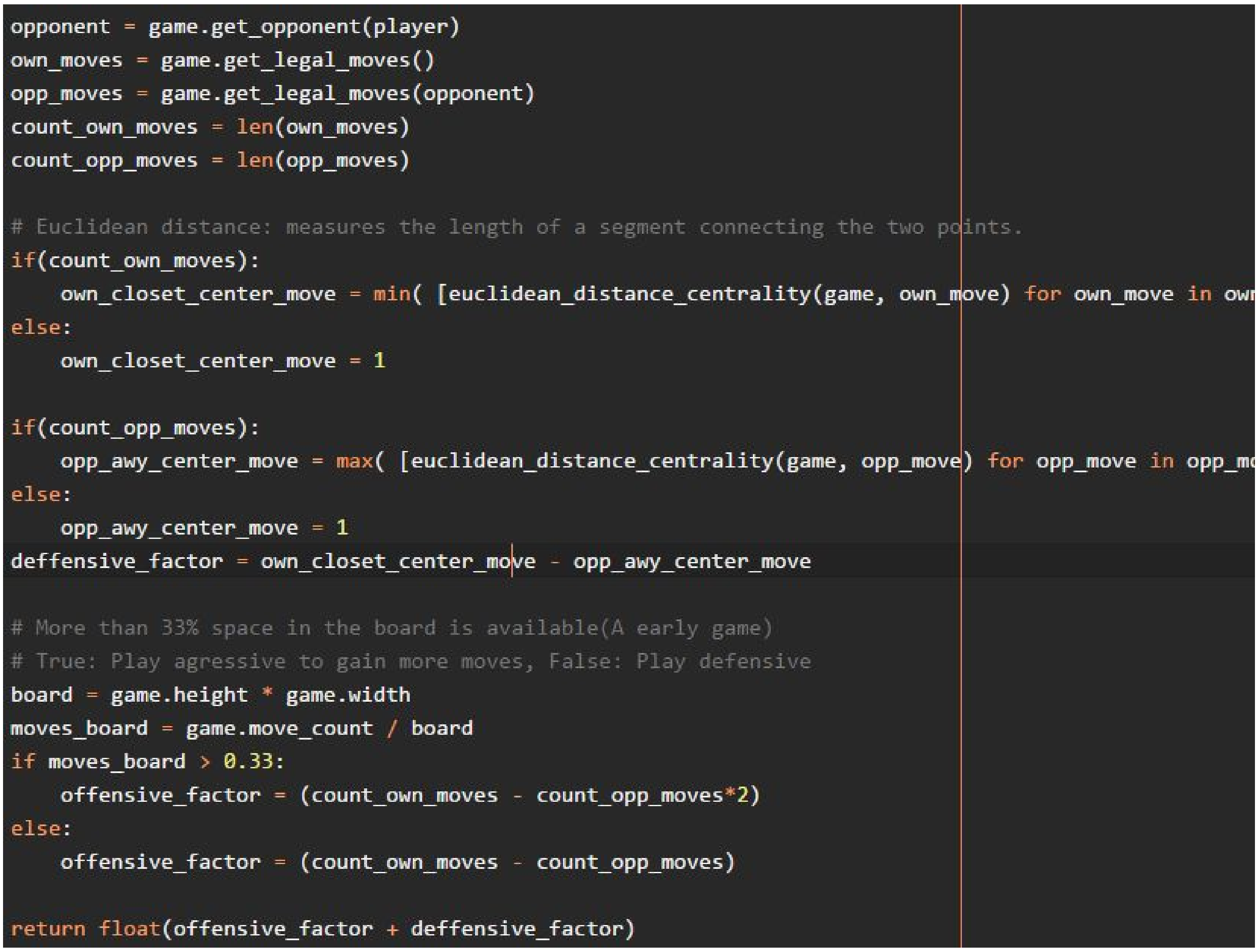
```python
opponent = game.get_opponent(player)
moves_own = len(game.get_legal_moves(player))
moves_opp = len(game.get_legal_moves(opponent))
board = game.height * game.width
moves_board = game.move_count / board

# More than 33% space in the board is available(A early game)
# True: Play agressive to gain more moves, False: Play defensive
if moves_board > 0.33:
    move_diff = (moves_own - moves_opp*2)
else:
    move_diff = (moves_own - moves_opp)

# Get current move of both player and opponent
pos_own = game.get_player_location(player)
pos_opp = game.get_player_location(opponent)

# abs(), return absolute value. The larger value, then longer distance
m_distance = abs(pos_own[0] - pos_opp[0]) + abs(pos_own[1] - pos_opp[1])

# When a move can lead to a huge move_diff and distance is near opponent
# that is a good move.
return float(move_diff / m_distance)
```

This heuristics attempts to motivate our player to play aggressive in a early game then play defensive in a late game. Also the distance of two players' move will affect how much this move will encourage to take, the closer they get, the stronger motivation.

Formula: ((moves_own - moves_opp) / abs(pos_own[0] - pos_opp[0]) + abs(pos_own[1] - pos_opp[1]))

Pro & Con: This heuristic to use a simpler theory than other 2 approach to get a quicker response time for iterating the search tree, but it doesn't has a complexity like other 2 heuristics to have a better strategy to face smart agent.

3. custom_score_3



```
opponent = game.get_opponent(player)
own_moves = game.get_legal_moves()
opp_moves = game.get_legal_moves(opponent)
count_own_moves = len(own_moves)
count_opp_moves = len(opp_moves)

# Euclidean distance: measures the length of a segment connecting the two points.
if(count_own_moves):
    own_closet_center_move = min( [euclidean_distance_centrality(game, own_move) for own_move in own
else:
    own_closet_center_move = 1

if(count_opp_moves):
    opp_awy_center_move = max( [euclidean_distance_centrality(game, opp_move) for opp_move in opp_mo
else:
    opp_awy_center_move = 1
deffensive_factor = own_closet_center_move - opp_awy_center_move

# More than 33% space in the board is available(A early game)
# True: Play agressive to gain more moves, False: Play defensive
board = game.height * game.width
moves_board = game.move_count / board
if moves_board > 0.33:
    offensive_factor = (count_own_moves - count_opp_moves*2)
else:
    offensive_factor = (count_own_moves - count_opp_moves)

return float(offensive_factor + deffensive_factor)
```

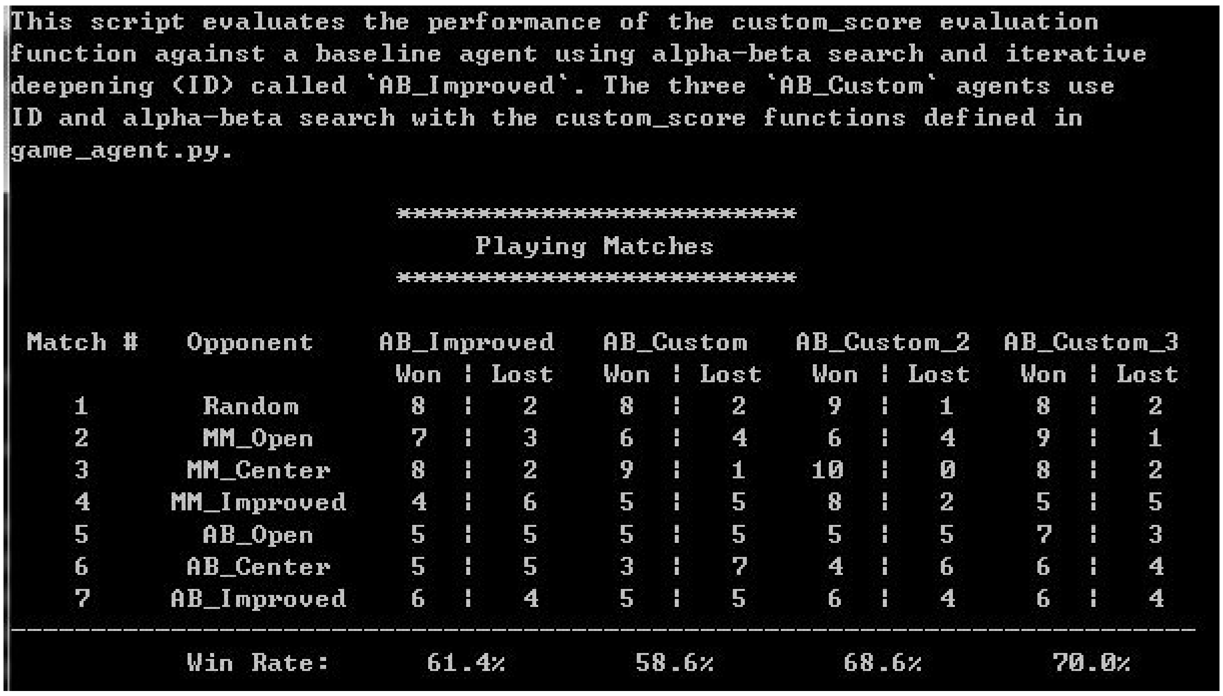
Half of the approach is almost the same as custom_score_2, the only difference is it uses both player legal moves to predict if that is a good move. We assume the definition of a good move which is a move can lead our agent to move toward center area and opponent to leave away from center area.

Formula: (count_own_moves - count_opp_moves*2) + (own_closet_center_move - opp_awy_center_move)

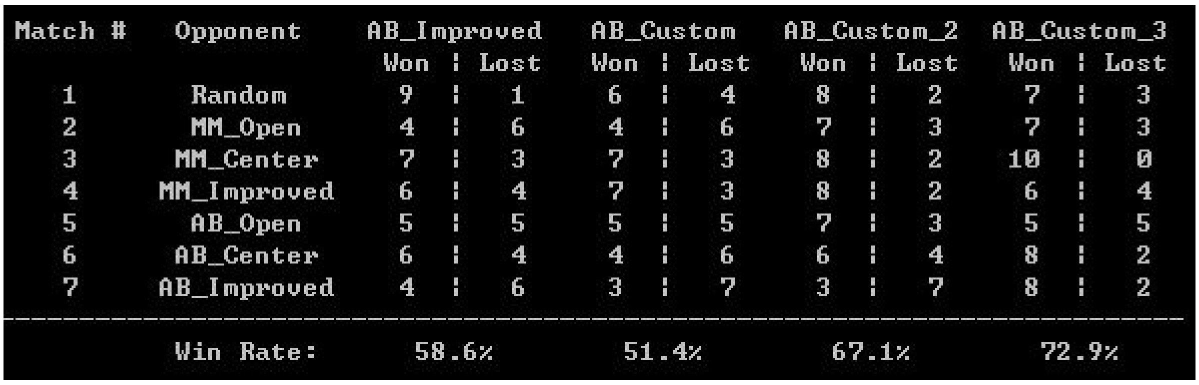
Pro: It performs the best performance among all these heuristics, especially when facing a smarter agent.

Con: It is not stable enough if we compare this heuristics to custom_score_2 when facing different type of opponent.

Result

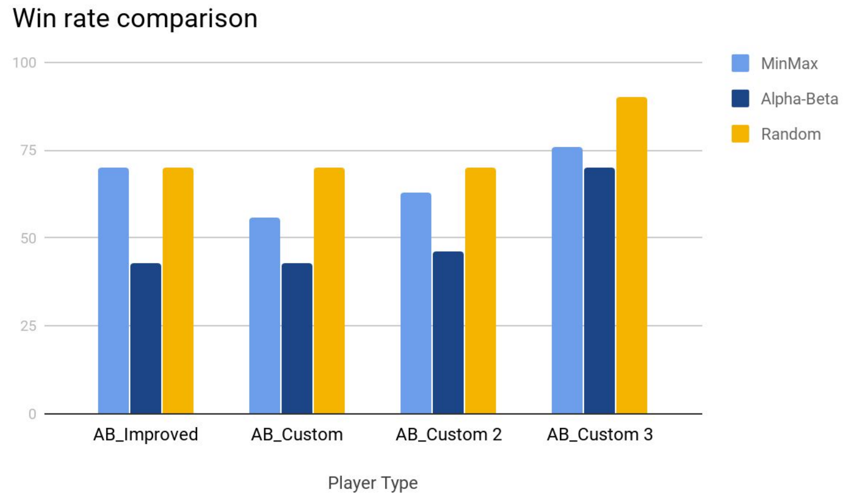


```
This script evaluates the performance of the custom_score evaluation
function against a baseline agent using alpha-beta search and iterative
deepening (ID) called `AB_Improved`. The three `AB_Custom` agents use
ID and alpha-beta search with the custom_score functions defined in
game_agent.py.

                        *************************
                             Playing Matches
                        *************************

 Match #   Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
                        Won | Lost   Won | Lost   Won | Lost   Won | Lost
    1       Random       8  |   2     8  |   2     9  |   1     8  |   2
    2       MM_Open      7  |   3     6  |   4     6  |   4     9  |   1
    3      MM_Center     8  |   2     9  |   1    10  |   0     8  |   2
    4     MM_Improved    4  |   6     5  |   5     8  |   2     5  |   5
    5       AB_Open      5  |   5     5  |   5     5  |   5     7  |   3
    6      AB_Center     5  |   5     3  |   7     4  |   6     6  |   4
    7     AB_Improved    6  |   4     5  |   5     6  |   4     6  |   4
--------------------------------------------------------------------------
           Win Rate:      61.4%        58.6%        68.6%        70.0%
```



```
 Match #   Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
                        Won | Lost   Won | Lost   Won | Lost   Won | Lost
    1       Random       9  |   1     6  |   4     8  |   2     7  |   3
    2       MM_Open      4  |   6     4  |   6     7  |   3     7  |   3
    3      MM_Center     7  |   3     7  |   3     8  |   2    10  |   0
    4     MM_Improved    6  |   4     7  |   3     8  |   2     6  |   4
    5       AB_Open      5  |   5     5  |   5     7  |   3     5  |   5
    6      AB_Center     6  |   4     4  |   6     6  |   4     8  |   2
    7     AB_Improved    4  |   6     3  |   7     3  |   7     8  |   2
--------------------------------------------------------------------------
           Win Rate:      58.6%        51.4%        67.1%        72.9%
```

AB_Custom performed badly in iterative deepening. One main factor is that it used too much time for evaluating its score rather than searching more potential good moves in the tree during Iterative Deepening. Maybe over optimization. **AB_Custom_2** performed stabilized results, demonstrating that the complexity of its function design is good enough for both min-max search and alpha-beta search. **AB_Custom_3** heuristics performs the best among the others. Thus we recommend using **AB_Custom3** heuristics.



Three justification for using this function:
1. Best performance among the others
2. Its complexity is good enough against smart agent and won't cause a long evaluation time.
3. Due to its formula, this function can easily replace its offensive & defensive factor to other better approach.