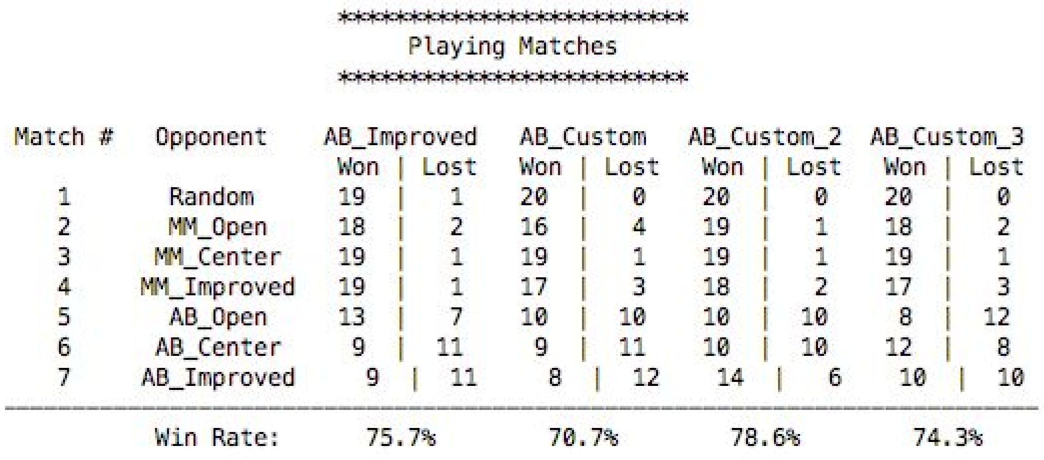
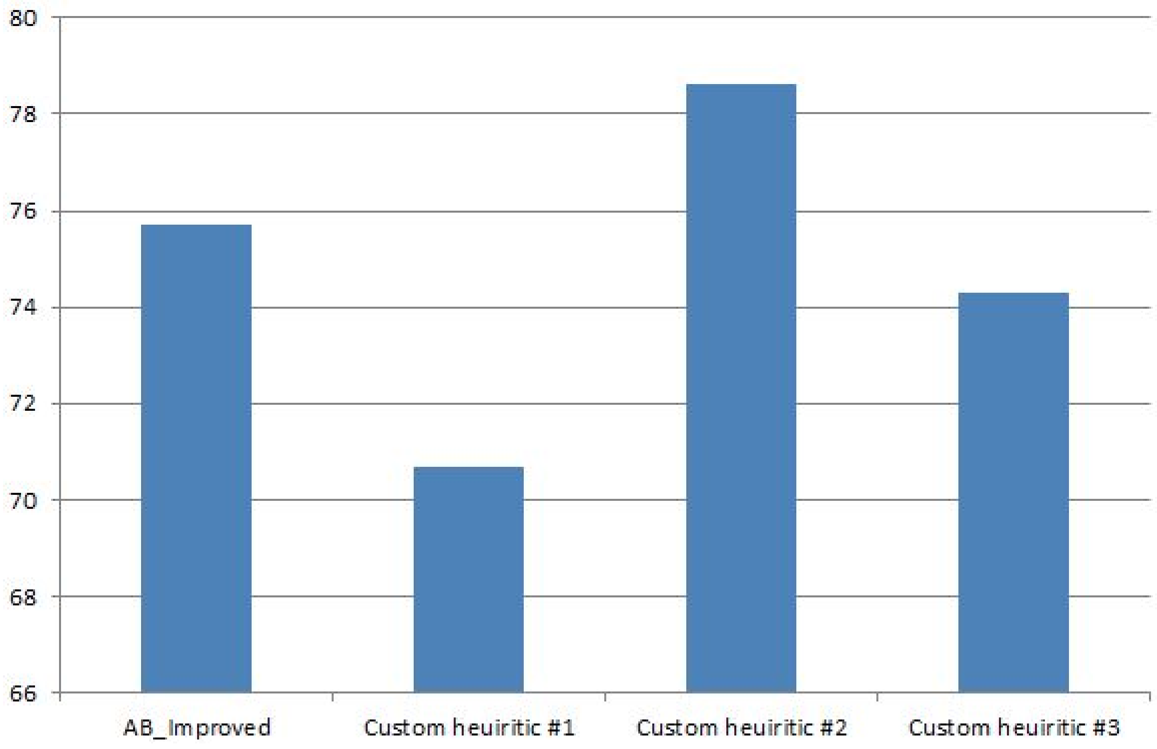
Comparison of all Heuristics:



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

Match #   Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
                       Won | Lost   Won | Lost   Won | Lost   Won | Lost
   1       Random      19  |   1    20  |   0    20  |   0    20  |   0
   2       MM_Open     18  |   2    16  |   4    19  |   1    18  |   2
   3      MM_Center    19  |   1    19  |   1    19  |   1    19  |   1
   4     MM_Improved   19  |   1    17  |   3    18  |   2    17  |   3
   5       AB_Open     13  |   7    10  |  10    10  |  10     8  |  12
   6      AB_Center     9  |  11     9  |  11    10  |  10    12  |   8
   7     AB_Improved    9  |  11     8  |  12    14  |   6    10  |  10
--------------------------------------------------------------------------
          Win Rate:      75.7%        70.7%        78.6%        74.3%
```

Winning rate Comparisons:



Custom Heuristic #1:

Implementation:

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

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

  my_location = game.get_player_location(player)
  opponent_location = game.get_player_location(game.get_opponent(player))
```

```
        return float(abs(my_location[0] - opponent_location[0]) + abs(my_location[1] - opponent_location[1]))
```

Analysis:

This heuristic demonstrates defensive technique to maximize the distance between opponent player. This is the best performing heuristic function of the three and it's simple and not very computationally heavy but its performance is poor when the number of matches increases when compared with #2 and #3 heuristics. More research has to be done on this heuristic for performance gains when playing against AB_Improved, AB_Center and AB_Open opponents.

Custom Heuristic #2:

Implementation:

```
def custom_score_2(game, player):
    if game.is_loser(player):
        return float("-inf")

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

    move_count = game.move_count

    w = 10 / (move_count + 1)

    # count number of moves available
    own_moves = len(game.get_legal_moves(player))
    opp_moves = len(game.get_legal_moves(game.get_opponent(player)))

    return float(own_moves - (w * opp_moves))
```

This heuristic weighted delta of available moves for each player and also it increases the importance of own_moves as the game progresses. As the number of matches increases, this heuristic evolved as number #1 with nearly 78.6% winning rate. It's not computationally intense as well, so I recommend this heuristic as my choice.

Custom Heuristic #3:

Implementation**:**

```
def custom_score_3(game, player):
    if game.is_loser(player):
        return float("-inf")

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

    score = .0
    total_spaces = game.width * game.height
    remaining_spaces = len(game.get_blank_spaces())
    coefficient = float(total_spaces - remaining_spaces) / float(total_spaces)

    my_moves = game.get_legal_moves(player)
    opponent_moves = game.get_legal_moves(game.get_opponent(player))
```

```
for move in my_moves:
    isNearWall = 1 if (move[0] == 0 or move[0] == game.width - 1 or
                    move[1] == 0 or move[1] == game.height - 1) else 0
    score += 1 - coefficient * isNearWall

for move in opponent_moves:
    isNearWall = 1 if (move[0] == 0 or move[0] == game.width - 1 or
                    move[1] == 0 or move[1] == game.height - 1) else 0
    score -= 1 - coefficient * isNearWall

return score
```

Analysis:

This heuristic determines if the player a near wall and thus possibly limiting moves resulting in a loss. It is still not well performing #2. Its very decent heuristic and its performance can be improved if more research is done.