Udacity Artificial Intelligence Nanodegree

Project 2: Build a Game-Playing Agent (Heuristic functions)

In this deterministic, two-player isolation game with 7x7 board and L-shaped player movement, we implemented iterative deepening search, minimax and alpha-beta pruning algorithms, as well as three custom heuristic functions.

Our three custom heuristic functions are as follows:

1. "custom_heuristic_1" function
    - General strategy:
      The general rule of winning this isolation game is to maximize our player's future moves and minimize our opponent's future moves. This heuristic function calculates the difference of number of options in the next move for both our player and the opponent. To increase the "aggressiveness" of our player, we multiply opponent's number of moves by 2.
    - Implementation:

```
def custom_heuristic_1(game, player):
    """
    get aggressive moves
    """
    own_moves = len(game.get_legal_moves(player))
    opp_moves = len(game.get_legal_moves(game.get_opponent(player)))

    return float(own_moves - 2 * opp_moves)
```

    - Result:
      This heuristic function performs better in the tournament against our baseline model, ie. 72.14% (custom_heuristic_1) vs 67.86% (ID_Improved).

2. "custom_heuristic_2" function
    - General strategy:
      Our second heuristic function rewards our player to occupy the center stage, especially at the beginning of the game. Intuitively, moving to the center stage at the early stage of the game will improve our player's position with the most potential number of future moves.

    - Implementation:

```
def custom_heuristic_2(game, player):
    """
    legal moves within center stage
    """
    center_stage = [(2, 2), (2, 3), (2, 4),
                    (3, 2), (3, 3), (3, 4),
                    (4, 2), (4, 3), (4, 4)]

    own_moves = game.get_legal_moves(player)
    opp_moves = game.get_legal_moves(game.get_opponent(player))
```

```
    own_moves_duplicates = len([x for x in own_moves if center_stage.count(x) > 1])
    opp_moves_duplicates = len([x for x in opp_moves if center_stage.count(x) > 1])

    return float(own_moves_duplicates - opp_moves_duplicates)
```

- Result:
  This heuristic function actually performed worse against our baseline model, ie. 53.57% (custom_heuristic_2) vs 70% (ID_Improved).

3. "custom_heuristic_3" function
   - General strategy:
     Our last heuristic function calculates the distance between our player and the opponent. This heuristic function returns a score with the biggest distance between our player and opponent. The rationale is to to position our player away from the opponent so we can get more spaces to move later.

   - Implementation:
```
def custom_heuristic_3(game, player):
    """
    distance between player and opponent
    """
    own_location = game.get_player_location(player)
    opp_location = game.get_player_location(game.get_opponent(player))

    return float(abs(own_location[0]-opp_location[0] + abs(own_location[1]-opp_location[1])))
```

   - Result:
     This heuristic function performed just slightly worse against our baseline model, ie. 62.14% (custom_heuristic_3) vs 67.86% (ID_Improved).

4. Last, but not least, we combine the above 3 heuristic functions into our custom_score as follows:
   -
```
def custom_score(game, player):
    if game.is_loser(player):
        return float("-inf")

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

    return custom_heuristic_1(game, player) + custom_heuristic_2(game, player) +
custom_heuristic_3(game, player)
```
   - Result:
     The combination of the above 3 heuristic functions perform better than our baseline, ie. 70% (all heuristic functions) vs 65.71% (ID_Improved).