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:

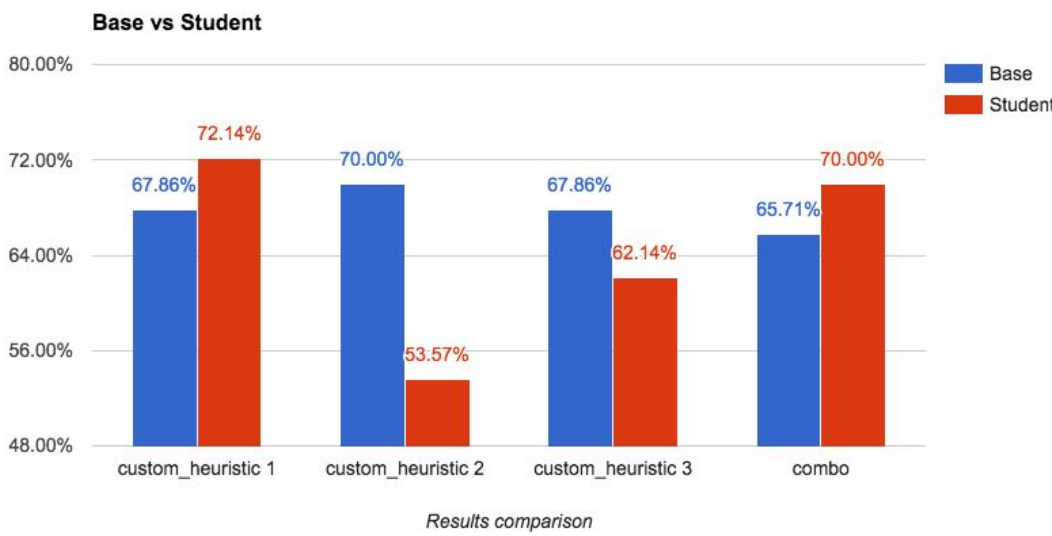
```
def custom_score(game, player):
   if game.is_loser(player):
      return float("-inf")

   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).

Results summary (in bar chart):



Notes:

1. y-axis: side-by-side winning percentage comparison between base-vs-student algorithms
2. x-axis: algorithms used in tournament
   2.1. custom_heuristic_1: # of legal moves for both player and opponent
   2.2. custom_heuristic_2: # of moves in center stage for both player and opponent
   2.3. custom_heuristic_3: # of distance between player and opponent
   2.4. combo: sum of all custom heuristic 1 to 3

Winning Ratio (in heat map):

	Random	MM_Null	MM_Open	MM_Improved	AB_Null	AB_Open	AB_Improved
base	9.00	3.00	1.22	1.86	2.33	1.22	1.00
custom_heuristic 1	19.00	9.00	2.33	0.82	3.00	1.86	1.86
custom_heuristic 2	1.50	0.82	1.22	1.00	1.50	1.22	1.00
custom_heuristic 3	9.00	2.33	1.50	1.22	2.33	1.00	0.67
combo	5.67	9.00	2.33	0.82	4.00	1.50	1.50

Notes:

1. The number above represents the winning ratio in x:1
   - For example: match 1 between student vs Random player using combo algorithm results in 17:3. The chart above shows 5.67 (=17/3).
2. gradient color codes:

- dark green: max values (most wins)
- yellow: 50% percentile (equal win-loss)
- dark red: min values (most losses)

Discussions:

- custom_heuristic_1 (strategy: # of legal moves) shows the most wins compared to other heuristics. Therefore, we suggest to use this algorithm in our tournament.
- Intuitively, it also makes sense why custom_heuristic_1 is the best strategy. As a player, we want to maximize our options in our future moves. Therefore, this strategy works well throughout the game.
- custom_heuristic_2 seeks legal moves toward the center stage. As with most board games, this general strategy is a good strategy at the beginning stage of the game. However, as the board is filled with previous moves, this heuristic function becomes less effective toward the middle and end game.
- custom_heuristic_3 is kind of similar to custom_heuristic_2. By moving away from the opponent, we expect our player to have more blank spaces to move. However, toward the middle and end game, this strategy is also becoming less effective.
- combo strategy (sum of all custom heuristic) fails to perform better than custom_heuristic_1. Adding more heuristic strategies does not necessarily improve the performance as the less effective strategies actually drag down the whole performance. In fact, we also have to be cautious with computing time to run all the heuristic functions in every move.