Heuristic Analysis on Knight Version of Isolation

As part of fulfillment of project 2 in AIND, Udacity

(A detailed introduction and template code can be found in https://github.com/udacity/AIND-Isolation)

1 Introduction

We build adversarial search agents to play a deterministic, two-player game of perfect information on a 7x7 board. This game is a variation form of "Isolation" with the usual queen's movement replaced with knight's. Since 7 is a odd number, it is favorable for the first player to put his/her first piece in the center and play in rotational symmetry whenever possible. Geometric trick can also help us to reduce search space, for at least the branching factor is cut to one-eighth in the very beginning branch. However, these tricks are not our concern in this project and will be not discussed further. Our agents only implement MiniMax algorithm with Alpha-Beta pruning with iterative deepening for various heuristic score functions. There are 8 or them as shown in the following figure:

```python
def custom_score(game, player):
    # heuristic value: undeterministic variation of improved score
    if game.is_loser(player):
        return float("-inf")
    if game.is_winner(player):
        return float("inf")

    own_moves = len(game.get_legal_moves(player))
    opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
    weight = random.random()
    return float(weight*own_moves - (1-weight)*opp_moves)
```

```python
def custom_score_2(game, player):
    # heuristic value: defensive variation of improved score
    if game.is_loser(player):
        return float("-inf")
    if game.is_winner(player):
        return float("inf")

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


    return float(pow(own_moves-9, 3) - pow(opp_moves, 3))
```

```python
def custom_score_3(game, player):
    # heuristic value: look ahead score
    if game.is_loser(player):
        return float("-inf")
    if game.is_winner(player):
        return float("inf")

    def two_steps_ahead(game, player):
        r, c = game.get_player_location(player)
        directions = [(1, 3), (-2, 0), (-3, -1), (3, -3), (-1, -3), (-4, 2), (-3, 3), (1, -1), (4, -2), (2, -4), (-1, 1), (3, 1), (-3, -3), (2, 0), (0, 4), (1, -3), (4, 0), (1, 1), (0, -4), (-1, -1), (-3, 1), (-4, 0), (4, 2), (3, 3), (-4, -2), (-1, 3), (0, -2), (3, -1), (2, 4), (-2, -4), (-2, 4), (0, 2)]
        potential_two_steps_ahead_moves = [(r + dr, c + dc) for dr, dc in directions if game.move_is_legal((r + dr, c + dc))]
        return potential_two_steps_ahead_moves

    own_potential = len(two_steps_ahead(game, player))
    own_moves = len(game.get_legal_moves(player))
    opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
    return float(own_potential + pow(own_moves, 2) - pow(opp_moves, 3))
```

```python
def custom_score_4(game, player):
    # heuristic value: just focus on own's performance
    if game.is_loser(player):
        return float("-inf")
    if game.is_winner(player):
        return float("inf")

    own_moves = len(game.get_legal_moves(player))
    return float(own_moves)
```

```python
def custom_score_5(game, player):
    # heuristic value: inspired by Monte Carlo tree search
    if game.is_loser(player):
        return float("-inf")
    if game.is_winner(player):
        return float("inf")

    child_node = game.get_legal_moves()
    random_no = int(random.random()*len(child_node))
    further_game = game.forecast_move(child_node[random_no])
    return custom_score_4(further_game, player)
```

```python
def custom_score_6(game, player):
    # heuristic value: more greedy and aggressive variation of improved score
    if game.is_loser(player):
        return float("-inf")
    if game.is_winner(player):
        return float("inf")

    own_moves = len(game.get_legal_moves(player))
    opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
    return float(pow(own_moves, 3) - pow(opp_moves, 3))
```

```python
def custom_score_7(game, player):
    # heuristic value: more greedy and aggressive variation of custom_score_6
    if game.is_loser(player):
        return float("-inf")
    if game.is_winner(player):
        return float("inf")

    own_moves = len(game.get_legal_moves(player))
    opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
    return float(pow(own_moves, 5) - pow(opp_moves, 5))
```

```python
def custom_score_8(game, player):
    # heuristic value: less greedy but more defensive variation
    if game.is_loser(player):
        return float("-inf")
    if game.is_winner(player):
        return float("inf")

    own_moves = len(game.get_legal_moves(player))
    opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
    return float(pow(9-own_moves, 3) - pow(9-opp_moves, 3))
```

2 Rationale of Custom Scores

At each leaf node in the tree search, it is easy to give a score if that is a end-game state. But usually, it isn't. Thus, we have to depends on heuristics. We now compare the rationale behind different heuristics.

The first heuristic, custom_score, is a non-deterministic variation of improved score in the given template code. It is linear combination of number of own moves and additive inverse of number of opponent move with random weight. As in improved score, we want to maximize our own chance to proceed in the game and minimize opponent's chance to proceed (or in other words maximize the chance to terminate opponent's possible moves). The prior goal is more important in a defensive move, whereas the latter goal is more important in a aggressive move. However, it is hard to tell in each leaf node of our search tree, whether the move should be more defensive or aggressive. So this heuristic proposes a random linear combination of both.

The second heuristic, custom_score_2, is a deterministic variation of improved score in a defensive way. It follows two basic principles. First, the larger the number of own moves, the higher the score; the smaller the number of own moves, the higher the score. This is just the same as improved score. Second, but rate of change is non-constant. The change in score is not in a linear relationship with the number of moves with respect to partial change in number of moves for either side. With respect to number of own moves, the score is more sensitive when it is close to 0; but with respect to number of opponent moves, the score is less sensitive when it is close to 0. As a result, we should obtain a very defensive game agent. Why do we want to be more defensive? Because the movement of knight is hard to capture in even a few moves. Any seemingly aggressive moves might be easily escaped. Then why don't we try to build a defensive agent from start to end.

The third heuristic, custom_score_3, is a look-ahead-score. We count the number of potential two steps ahead moves from own move. Sum this number to the squre or number of own moves minus the cube of number of opponent moves. This is kind of doing one more depth of search, but via a faster estimation. This score consists of three terms, with each concern with the number of player's own next move, player's own move in two step ahead and opponent's immediate move. Rising power in two (out of three) components in custom_score_3 keep the three terms balanced, with roughly the same range. This is not as fast as the previous custom score, but would the estimation of one more depth compensate the time cost? We have to put it to tournament for testing.

The forth heuristic, custom_score_4, is greedy and simple. It just focus on player's own performance by counting the number of own moves. Since knight is so hard to predict, then we will use all the effort to evaluate own move. This score is in no doubt very fast. And in quite a number of cases (against Random, MM_Open, AB_Improved, see the figure in the next section), it over-performed improved score which consider opponent moves as well. Maybe everything is within sampling error; or maybe simplest is the best.

The fifth heuristic, custom_score_5, is inspired by Monte Carlo tree search (MCTS). At each leaf node of our search tree, we randomly select one child node, if there is any, and evaluate custom_score_4 on the child. This is custom_score_5 that may bring a bit more variation and randomness to the really simple custom_score_4. However, this is not the proper way to do MCTS. This interesting agent was built merely for fun.

Similar to custom_score_2, the sixth heuristic, custom_score_6, is another deterministic variation of improved score. The rate of change is non-constant. With respect to number of own moves, the score is more sensitive when it is close to 8 (that is the largest possible number of moves); and this is the same with respect to number of opponent moves. This is a testing agent that is thought to be more greedy and more aggressive than agent using improved score. The agent is strong but not as outstanding as custom_score_2.

Maybe agent using custom_score_6 isn't aggressive enough. We build a more aggressive one to see if it will be more powerful. The seventh heuristic, custom_score_7, builds a testing agent that is thought to be more greedy and more aggressive than agent using custom_score_6. The score depends a function with steeper derivative for a more rapid rate of change. It turn out that this agent is strong but not significantly stronger than custom_score_2 or custom_score_6. This is testing agent strengthen our belief that playing with the agile knight movement, being aggressive might not be obviously better.

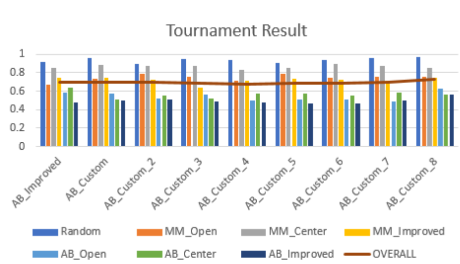
Finally, the last heuristic, custom_score_8, is another deterministic variation of improved score. In comparison to the very aggressive agent, we also want a agent that is less greedy but more defensive. So we change the score function again. With respect to number of own moves, the score is more sensitive when it is close to 0; and this is the same with respect to number of opponent moves. This should be more defensive than custom_score_2, and does sometimes over-perform custom_score_2, but not always.

3 Tournament Result

We run a tournament for the above heuristics and also the improved score against a random agent, three MiniMax agents and three Alpha-Beta pruning agents. The result is in the following figure:

```
In [1]: runfile('C:/Users/yeekatai/PycharmProjects/AIND-Isolation/tournament.py', wdir='C:/Users/yeekatai/PycharmProjects/AIND-Isolation')

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  AB_Custom_4  AB_Custom_5  AB_Custom_6  AB_Custom_7  AB_Custom_8
                        Won | Lost   Won | Lost   Won | Lost   Won | Lost   Won | Lost   Won | Lost   Won | Lost   Won | Lost   Won | Lost
    1       Random      92  |   8    96  |   4    89  |  11    95  |   5    94  |   6    91  |   9    94  |   6    96  |   4    97  |   3
    2       MM_Open     67  |  33    73  |  27    79  |  21    76  |  24    71  |  29    79  |  21    75  |  25    76  |  24    76  |  24
    3      MM_Center    85  |  15    88  |  12    87  |  13    87  |  13    83  |  17    85  |  15    89  |  11    87  |  13    85  |  15
    4     MM_Improved   74  |  26    75  |  25    72  |  28    64  |  36    71  |  29    73  |  27    72  |  28    71  |  29    75  |  25
    5       AB_Open     58  |  42    57  |  43    52  |  48    56  |  44    50  |  50    51  |  49    51  |  49    49  |  51    63  |  37
    6      AB_Center    64  |  36    51  |  49    55  |  45    52  |  48    57  |  43    57  |  43    55  |  45    59  |  41    56  |  44
    7     AB_Improved   48  |  52    50  |  50    51  |  49    49  |  51    48  |  52    47  |  53    47  |  53    50  |  50    56  |  44
--------------------------------------------------------------------------
           Win Rate:      69.7%        70.0%        69.3%        68.4%        67.7%        69.0%        69.0%        69.7%        72.6%

There were 1.0 timeouts during the tournament -- make sure your agent handles search timeout correctly, and consider increasing the timeou
margin for your agent.
```

Every Alpha-Beta pruning agents obviously over-performed random agent and the three MiniMax agents. And most of them are better than Alpha-Beta pruning agent evaluated with open_move_score or center_score in the template code. Performance between different heuristics might need more matches before we can confidently conclude which Alpha-Beta pruning agent is the best.In fact, Alpha-Beta pruning with iterative deepening seems to be the key factor for all strong agents. All of them had roughly 70% of win rate as shown in the graph. Maybe any heuristic, that make sense, will do reasonably well. (Otherwise, the problem may be due to the heuristic, such as AB_center that encourage player to move away from center...) Anyway, we believe that any three of our custom heuristics would good enough to submit.



However, it is required in the project rubric to make a recommendation about the best evaluation function. In this case, we have to recommend our eighth heuristic, custom_score_8, for the following reasons. First, AB_Custom_8 is one of the only two agents (the other is our second heuristic) achieving more than 50% win rate against every opponent. Second, it has the highest overall win rate. Third, it has highest win rate against 3 test agents: random, AB_open and AB_improved; this number 3 out of 7 is also the highest among all agents. Finally, due to the agile movement of knight, being aggressive might not bring too much benefit to an agent, so its defensive approach is more appropriate.

Disclaimer

The pronoun "we" means "I", the author, and "you", the reader throughout this note.