Isolation Heuristic Analysis

Pengfei Cai

May 2018

Parameters used for Heuristic Test

When analyzing what consists of a good strategy to play the isolation game, I found that to gain as many moves as possible for the next step takes a predominant priority in the whole strategy, which in turn, may require the player to move as close as possible to the center to gain maximum possibilities and minimize opponent's options (max_comapre). At the same time, the player should also take initiative to block opponent's moves.

In general, the sums of values, which reflect the key evaluations mentioned above, weighted with different strategies are used to perform the heuristic analysis.

Table of Results

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

Match #   Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
                       Won | Lost   Won | Lost   Won | Lost   Won | Lost
   1       Random      10  |   0    10  |   0    10  |   0    10  |   0
   2       MM_Open      9  |   1     7  |   3     8  |   2     9  |   1
   3      MM_Center    10  |   0    10  |   0     9  |   1    10  |   0
   4     MM_Improved    8  |   2     9  |   1     7  |   3     7  |   3
   5       AB_Open      5  |   5     5  |   5     6  |   4     7  |   3
   6      AB_Center     4  |   6     5  |   5     6  |   4     5  |   5
   7     AB_Improved    6  |   4     3  |   7     5  |   5     5  |   5
------------------------------------------------------------------------
          Win Rate:      74.3%        70.0%        72.9%        75.7%
```

Heuristic 1

Implementation:

```
def custom_score(game, player):
    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)))

    w, h = game.width / 2., game.height / 2.
    y, x = game.get_player_location(player)
    y2, x2 = game.get_player_location(game.get_opponent(player))

    distance_to_center = float((h - y)**2 + (w - x)**2)
    opp_distance_to_center = float((h - y2)**2 + (w - x2)**2)
    return float((own_moves - opp_moves) + (opp_distance_to_center - distance_to_center)*0.634/(game.move_count))
```

Analysis:

This heuristic performs okay. The win rate isn't better than AB_Improved when it plays 10 games. This heuristic is quick to compute and involves additional information about the state of the board. How important the distance to the center is at any point in the game is controlled by the varying weight. The constant in this weight was obtained by random sampling in iterative plays against AB_Improved. The value of this constant could be improved to achieve better results.

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")

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

    pos_y, pos_x = game.get_player_location(player)
    number_of_boxes_to_center = abs(pos_x - math.ceil(game.width/2)) + \
        abs(pos_y - math.ceil(game.height/2)) - 1

    if percent_game_completed(0, 10, game):
        return 2*own_moves - 0.5*number_of_boxes_to_center
    elif percent_game_completed(10, 40, game):
        return float(3*own_moves - opp_moves - 0.5*number_of_boxes_to_center)
    else:
        return 2*own_moves - opp_moves
```

Analysis:

This heuristic performs adequately but is no better than AB_Improved. The idea of switching strategies based on a stage in the game is also a good one, but the time to switch the strategy and the optimal strategy to use during a game stage is difficult to find. Towards the end of the game,

we're not looking at the distance to the center because most moves will be away from the center and towards the walls anyway.

Heuristic 3

Implementation:

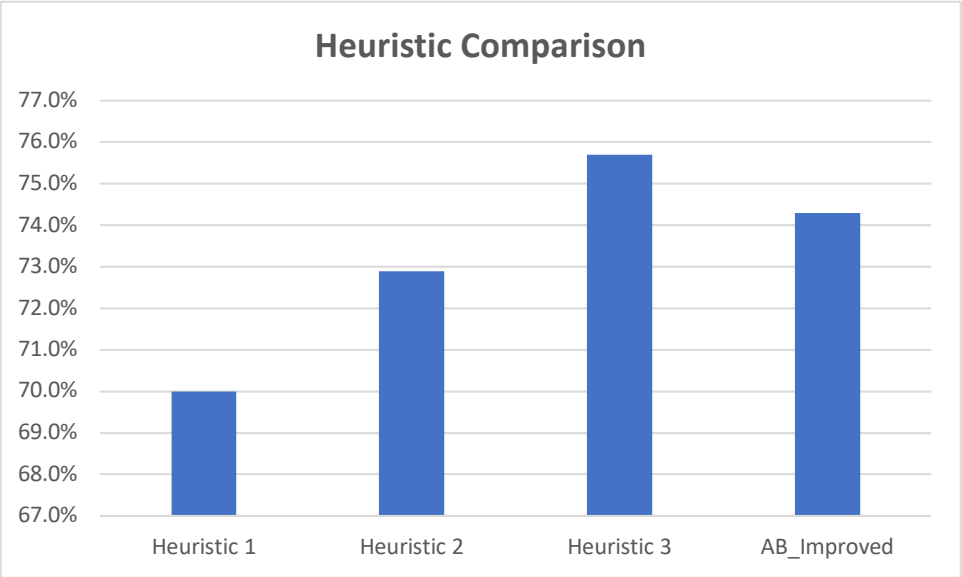
```
def custom_score_3(game, player):
    if game.is_loser(player):
        return float("-inf")

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

    my_moves = game.get_legal_moves(player)
    opponent_moves = game.get_legal_moves(game.get_opponent(player))
    own_moves = len(my_moves)
    opp_moves = len(opponent_moves)
    moves_so_far = 0
    for box in game._board_state:
        if box == 1:
            moves_so_far += 1

    w, h = game.get_player_location(game.get_opponent(player))
    y, x = game.get_player_location(player)
    distance_to_center = float((h - y)**2 + (w - x)**2)

    wall_boxes = [(x, y) for x in (0, game.width-1) for y in range(game.width)] + \
    [(x, y) for y in (0, game.height-1) for x in range(game.height)]

    penalty = 0
    if (x, y) in wall_boxes:
        penalty -= 1

    quality_of_move = 0
    for move in my_moves:
        y, x = move
        dist = float((h - y)**2 + (w - x)**2)
        if dist == 0:
            quality_of_move += 1
        else:
            quality_of_move += 1/dist
        if move in opponent_moves:
            quality_of_move -= 1

    if percent_game_completed(0, 40, game):
        return float(own_moves - opp_moves - distance_to_center + quality_of_move + penalty)
    else:
        return own_moves - 2*opp_moves
```

Analysis:

Clearly this heuristic performs better than AB_Improved. This heuristic takes multiple inputs to assess the quality of the board state. However, the heuristic is also more expensive to compute since we must go through future moves and compute the distance to the center for each. Again,

switching strategies based on the stage in the game might be a good idea, but it is difficult to predict when the switch strategies.

Conclusion



As illustrated in the bar chart above, the custom heuristic 3 gives the best result and outperforms AB_Improved. Aside from the variables included in the previous heuristics, this heuristic takes "quality" and "penalty" variables into the assessment of the board state. To evaluate the quality, the distance between the moves of both players in a round is assessed. Moreover, the penalty variable is used to include the negative effects of wall boxes around the player's moves. Therefore, I recommend this heuristic over the others.