# Heuristic analysis for the game of isolation

*by A. Tkachenko for AIND, May 2017*

In this project our goal is to design a game-playing agent for the modified game of isolation where each player uses L-shaped movements. A part of this task involves developing evaluation functions to assess how favorable is a current board state from a player's perspective.

## Game options

Per tournament rules only 7x7 square boards are used. Each agent must play equal number of times as a 1st and as a 2nd player to eliminate potential 1st player advantage from affecting results.

## Sample size

The default number of matches in tournament.py (10 = 5 as a Player1 & 5 as a Player2) was found to be inadequate for comparing agent pairs where there is no huge difference in performance as evidenced by:

- Self-win rate of AB_improved agent, which often oscillated between 30 and 70%;
- Win rate of simpler default agents from sample_players.py, which occasionally had higher win rate against AB_improved

Doubling the number of games helped to reduce the noise to some degree, but still made comparisons difficult for agents with small difference in performance. As a result, quadruple number of games (40) was used for final performance comparisons.

## Win/lose conditions

All evaluated functions had win/lose condition checks in the beginning:

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

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

For the sake of brevity, they will be omitted in the following discussion and code snippets

## Evaluation functions

We have used the following functions:

1. weighted “improved score” function – a minor improvement over the original one that favors aggressive play when weight_opponent > weigh_player and vice versa.

NAME: **custom_score_4**

```
player_moves_tot = len(game.get_legal_moves(player))
opponent_moves_tot = len(game.get_legal_moves(game.get_opponent(player)))
weight_player=1
weight_opponent=1.5

return float(weight_player*player_moves_tot - weight_opponent*opponent_moves_tot)
```

2. “improve score” function accounting for the quality of each move (how close to the center they are). A move to the center cell (3,3) is worth 6 points. Anything else is 6 minus L1 distance from the center. The worst move (corner) is worth 6 - 3 - 3 = 0 points. This function was also weighted to favor more aggressive strategy

NAME: **custom_score**

```
player_moves=game.get_legal_moves(player)
oppo_moves=game.get_legal_moves(game.get_opponent(player))

player_move_score=sum([6-abs(move[0]-3) - abs(move[1]-3) for move in player_moves])
oppo_move_score=sum([6-abs(move[0]-3) - abs(move[1]-3) for move in oppo_moves])

return player_move_score - 1.5*oppo_move_score
```

3. “Chase / run away” evaluation function which rewards either following your opponent closely or running away from him (depending on the – or + sign in the “return” statement).

NAME: **custom_score_5**

```
player_loc=game.get_player_location(player)
oppo_loc=game.get_player_location(game.get_opponent(player))
return -(player_loc[0] - oppo_loc[0] + player_loc[1] - oppo_loc[1])
```

4. A family of functions assessing how close on average are player and/or the opponent to the unvisited cells on the board. It can use either L1 distance or L2 distance. In the interest of time only one variant was picked for detailed performance evaluation.
NAME: **custom_score_2**

```
def dist_func(pos0,pos1,pos2): #locations of: empty cell, player, and opponent respectively
    #difference between player's and opponent's L1 distances:
    return abs(pos1[0] - pos0[0]) + abs(pos1[1] - pos0[1]) - abs(pos2[0] - pos0[0]) - abs(pos2[1] - pos0[1])

    ###### Alternative versions of dist_func
    ##L1 dist(player, cell)
    #return abs(pos1[0] - pos0[0]) + abs(pos1[1] - pos0[1])

    ##L1 dist(opponent, cell) with a minus
    #return - abs(pos1[0] - pos0[0]) - abs(pos1[1] - pos0[1])

    ##difference between squares of player's and opponent's L2 distances to a cell
    #return (pos1[0] - pos0[0])**2 + (pos1[1] - pos0[1])**2 - (pos2[0] - pos0[0])**2 - (pos2[1] - pos0[1])**2

    ##difference between player's and opponent's L2 distances to a cell
    #return ((pos1[0] - pos0[0])**2 + (pos1[1] - pos0[1])**2)**0.5 - ((pos2[0] - pos0[0])**2 + (pos2[1] – pos0[1])**2)**0.5

    ##only counts (square of) player's L2 distnace to a given empty cell
    #return (pos1[0] - pos0[0])**2 + (pos1[1] - pos0[1])**2

    ##only counts (square of) opponent's L2 distance to a given empty cell (with an opposite sign)
    #return - (pos2[0] - pos0[0])**2 - (pos2[1] - pos0[1])**2

player_loc=game.get_player_location(player)
empty_spaces=game.get_blank_spaces()

oppo_loc=game.get_player_location(game.get_opponent(player))
distances=[dist_func(pos0, player_loc, oppo_loc) for pos0 in empty_spaces]
average_dist=sum(distances)/len(distances)
return -average_dist
```

5. “Combined” heuristic that uses fast function early in the game (aggressive quality-weighted “improve score” function) and another function for the late game (aver. proximity to the empty cells).
NAME: **custom_score_3**

```
def dist_func(pos0,pos1,pos2): #locations of: empty cell, player, and opponent respectively
    return abs(pos1[0] - pos0[0]) + abs(pos1[1] - pos0[1]) - abs(pos2[0] - pos0[0]) - abs(pos2[1] - pos0[1])

if len(game.get_blank_spaces())>20: #20 is our arbitrary threshold dividing early game and late game
    player_moves=game.get_legal_moves(player)
    oppo_moves=game.get_legal_moves(game.get_opponent(player))
    player_move_score=sum([6-abs(move[0]-3) - abs(move[1]-3) for move in player_moves])
    oppo_move_score=sum([6-abs(move[0]-3) - abs(move[1]-3) for move in oppo_moves])
    return player_move_score - 1.5*oppo_move_score
else:
    player_loc=game.get_player_location(player)
    empty_spaces=game.get_blank_spaces()
    oppo_loc=game.get_player_location(game.get_opponent(player))
    distances=[dist_func(pos0, player_loc, oppo_loc) for pos0 in empty_spaces]
    average_dist=sum(distances)/len(distances)
    return -average_dist
```

## Performance comparison

All 5 evaluation functions were tested against a standard set of opponents (see sample_players.py & tournament.py for more details). The results are combined below:

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

 Match #   Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3   AB_Custom_4  AB_Custom_5
                        Won | Lost   Won | Lost   Won | Lost   Won | Lost   Won | Lost   Won | Lost
    1       Random      37  |   3    38  |   2    37  |   3    38  |   2    36  |   4    36  |   4
    2       MM_Open     29  |  11    33  |   7    33  |   7    30  |  10    26  |  14    28  |  12
    3      MM_Center    32  |   8    31  |   9    36  |   4    30  |  10    38  |   2    32  |   8
    4     MM_Improved   29  |  11    31  |   9    27  |  13    28  |  12    27  |  13    25  |  15
    5       AB_Open     19  |  21    25  |  15    19  |  21    20  |  20    23  |  17    18  |  22
    6      AB_Center    25  |  15    23  |  17    18  |  22    21  |  19    19  |  21    19  |  21
    7     AB_Improved   19  |  21    24  |  16    13  |  27    22  |  18    17  |  23    18  |  22
--------------------------------------------------------------------------------------------------
           Win Rate:      67.9%        73.2%        65.4%        67.5%        66.4%        62.9%
```

As we can see, **custom_score** showed the best performance, consistently beating AB_Improved agent in ~60% of games and having an overall win rate of 70+%. **custom_score_2** won >60% games total but consistently lost >50% games to AB_improved. Combined heuristic **custom_score_3** had a slight edge against AB_improved and had the tournament win rate of ~67%. The last two heuristics had a win rate of 60-70% but still lost to AB_improved.

Results shown above indicate that **custom_score** evaluation function had the highest win rate and outperformed AB_improved agent. Another strong point of this heuristic is it's relative simplicity, which leads to low computation cost. As a result, an agent using this heuristic can explore the game tree to a greater depth in a fixed amount of time. Finally, despite it's simplicity it captures to a certain degree most of the key factors that determine how good a given game state is from a player's perspective: a) number of possible moves b) (un)likelyhood of being cornered in the future, estimated through the average proximity of available moves to the center of the board and c) the same information about the opponent, which allows to calculate our relative positional advantage.

In addition to that, this function is quite flexible. It has two adjustable parameters which allow optimizing our agent's behavior:

1. Relative weights assigned to player and opponent scores can be used to tweak how aggressive we want to be in a game (whether we want to improve our position or worsen opponent's).
2. The value of the center cell ("6" in the final version of this function) can be adjusted to give a stronger or weaker incentive to stay closer to the center of the board.

Such flexibility means that the function's scoring algorithm can be easily  adjusted without a major rewrite and a DOE can be done if needed to find the optimal values in general, against a specific type of opponent, or for a specific stage of the game (early game vs late game).

To improve results even further, a combined heuristic can be used, which utilizes a better late-game evaluation function. Such function can use a computationally expensive algorithm to do a more thorough scoring of a smaller amount of remaining valid moves. It may include, for example, detection of board partition and determination of the longest possible jump paths.