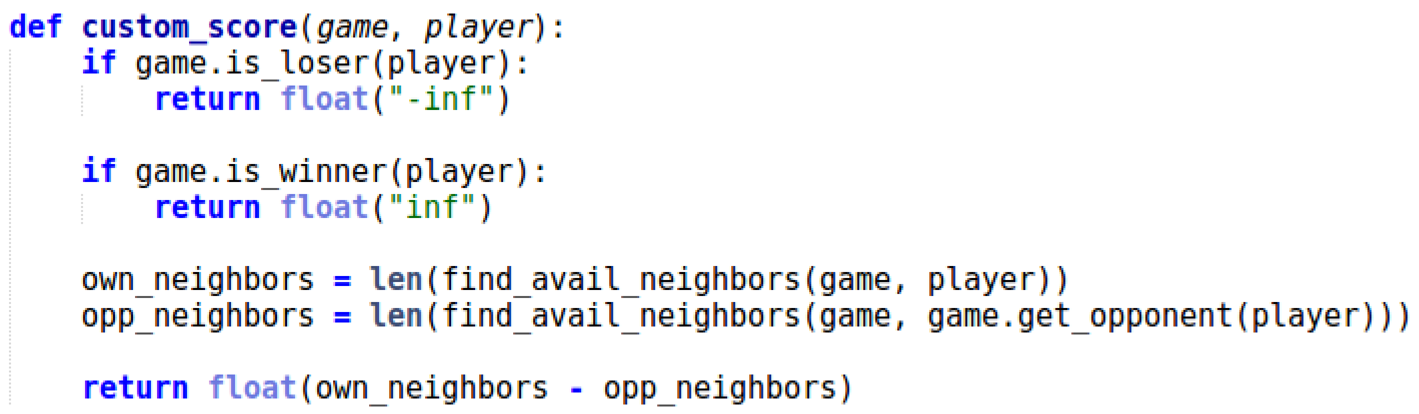
Heuristic Analysis for Isolation Game

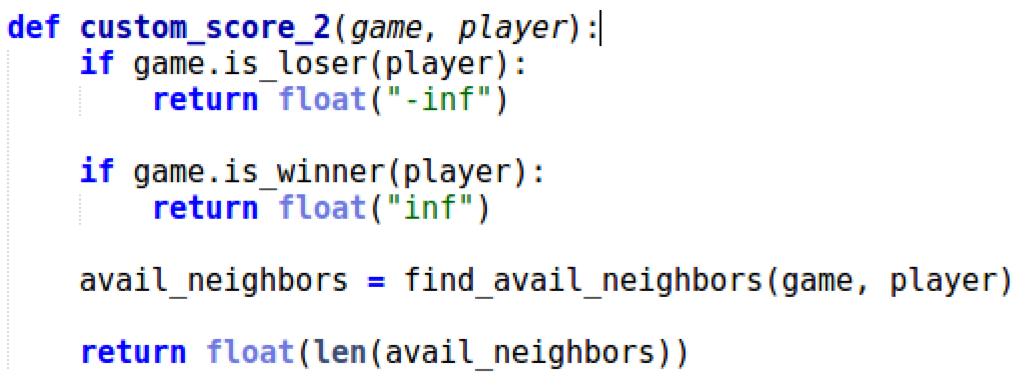
May 11, 2017

1. ANALYSIS OF HEURISTIC FUNCTIONS



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

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

    own_neighbors = len(find_avail_neighbors(game, player))
    opp_neighbors = len(find_avail_neighbors(game, game.get_opponent(player)))

    return float(own_neighbors - opp_neighbors)
```

Neighbors are defined as locations that are 1 row or 1 column away from the location of the specified player. custom_score calculates the difference between the number of the given player's neighbors and that of their opponent's. The given player is incentivized to move to locations that are adjacent to as many unoccupied spaces as possible while minimizing their opponent's number of unoccupied neighbors. This is because if the given player's neighbors are all occupied, most likely they're trapped and soon they'll lose.



```
def custom_score_2(game, player):
    if game.is_loser(player):
        return float("-inf")

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

    avail_neighbors = find_avail_neighbors(game, player)

    return float(len(avail_neighbors))
```

custom_score_2 calculates the number of the given player's neighbors and that of their opponent's. The given player is incentivized to move to locations that are adjacent to as many unoccupied spaces as possible. However, unlike custom_score, this function doesn't take into account opponent's neighbors. As a result, it is not as informative as the former. We can see in ***Table 1*** that custom_score_2 performs worse than custom_score.

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

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

    center_y, center_x = int(game.height/2), int(game.width/2)
    player_y, player_x = game.get_player_location(player)

    own_neighbors = len(find_avail_neighbors(game, player))
    opp_neighbors = len(find_avail_neighbors(game, game.get_opponent(player)))

    return -float((center_y - player_y)**2 + (center_x - player_x)**2) +
            float(own_neighbors - opp_neighbors)
```

The function considers both the distance of the player's location from the center and the difference between the number of the given player's neighbors and that of their opponent's. The player is incentivized to move to locations that are both close to the center and that are adjacent to as many unoccupied spaces as possible, while trying to minimize their opponent's number of unoccupied neighbors. However, interestingly, custom_score_3 performs slightly worse than custom_score. This could be because it takes more time to compute custom_score_3 and iterative deepening using this function would be slightly less likely to search as deeply as iterative deepening with custom_score does.

Table 1: Results from running 20 matches against each opponents

Match #	**Opponent**	**AB_Improved**	**AB_Custom**	**AB_Custom_2**	**AB_Custom_3**
		Won | Lost	Won | Lost	Won | Lost	Won | Lost
1	Random	31 | 9	35 | 5	33 | 7	36 | 4
2	MM_Open	19 | 21	27 | 13	25 | 15	22 | 18
3	MM_Center	25 | 15	28 | 12	25 | 15	26 | 14
4	MM_Improved	21 | 19	20 | 20	19 | 21	22 | 18
5	AB_Open	21 | 19	19 | 21	20 | 20	20 | 20
6	AB_Center	22 | 18	20 | 20	17 | 23	20 | 20
7	AB_Improved	18 | 22	20 | 20	18 | 22	20 | 20
	Win Rate:	56.1%	60.4%	56.1%	59.3%

RECOMMENDATIONS: I recommend using custom_score for several reasons:

- It takes into account information about both the given player and their opponent, whereas custom_score_2 only has information about the given player.
- It is slightly less complicated than custom_score_3. As the board size grows and/or the allowed search time shrinks, I believe it's important to have a simpler heuristic function so that we can search more deeply.
- Moreover, as shown in Section 2 below, get_move has already ensured that the given player occupies the center square if vacant. This is equivalent to the minimum value of the squared distance to the center, which is the best case scenario for the first term of custom_score_3 that calculates the squared distance to the center.

2. ANALYSIS OF ALPHABETAPLAYER'S GET_MOVE FUNCTION

In addition to the required implementation of iterative deepening for get_move in the AlphaBetaPlayer class, I have added a few features to enhance an AlphaBetaPlayer's performance. Although the features are not in custom_score, they play an important role in the player's performance. Therefore, it's worth mentioning them here.

a) If there is only one legal move available to a player, they should choose it, and avoid searching more deeply down the game tree, which is much more time consuming.

```
# If there is only 1 next legal move, return it and no need to do iterative deepening
if len(legal_moves) == 1:
    return legal_moves[0]
```

b) A player should occupy the center position if available, right from the beginning of the game.

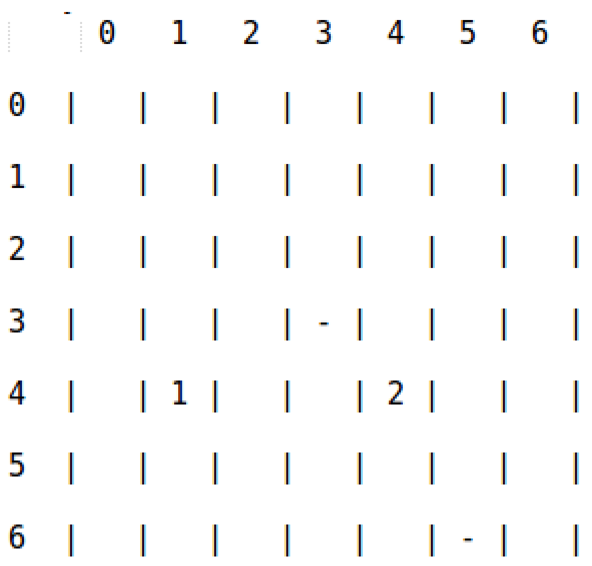
```
center_y, center_x = int(game.height/2), int(game.width/2)
# For its first move, player 1 should occupy the center position
if game.move_count == 0:
    return (center_y, center_x)
# If player 1 didn't occupy the center position it its first move, player 2 should do so
elif game.move_count == 1 and (center_y, center_x) in game.get_legal_moves():
    return (center_y, center_x)
```

c) As mentioned in the lecture, a player should reflect their opponent's move if possible. See ***Figure 1*** for an example.

```
# Reflect opponent's move if possible:
opp_r, opp_c = game.get_player_location(game._inactive_player)
own_next_r, own_next_c = game.height - 1 - opp_r, game.width - 1 - opp_c
if (own_next_r, own_next_c) in legal_moves:
    return (own_next_r, own_next_c)
```

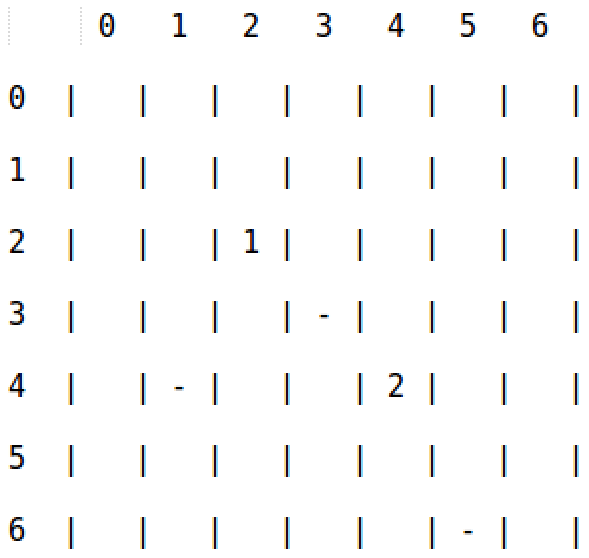
Figure 1 – Example of player 1 reflecting player 2's move

It’s player’s 1 turn to move. The game state before player 1 moves:



```
     0   1   2   3   4   5   6

0  |   |   |   |   |   |   |   |

1  |   |   |   |   |   |   |   |

2  |   |   |   |   |   |   |   |

3  |   |   |   | - |   |   |   |

4  |   | 1 |   |   | 2 |   |   |

5  |   |   |   |   |   |   |   |

6  |   |   |   |   |   | - |   |
```

The game state after player 1 moves to (2, 2), reflecting opponent's position at (4, 4):



```
     0   1   2   3   4   5   6

0  |   |   |   |   |   |   |   |

1  |   |   |   |   |   |   |   |

2  |   |   | 1 |   |   |   |   |

3  |   |   |   | - |   |   |   |

4  |   | - |   |   | 2 |   |   |

5  |   |   |   |   |   |   |   |

6  |   |   |   |   |   | - |   |
```