Custom score #1

Description:

This heuristic to designed to improve the performance of the original AB_Improved evaluation function, thus be able to search deeper in the tree.

Implementation:

With the original AB_Improved implementation, after the game gets all the legal moves for a player, it shuffles the moves. Since we only care about the number of moves in the evaluation function, shuffling the redundant.

Game result:

After striping the shuffling from the original implementation, the performance improvement is significant. For a 400 game run, the improved version runs 24% faster than the original function (from 0.026ms to 0.019ms). For each game running on my laptop, the agent needs to evaluate ~20k times of the game state, the performance improvement is able to give custom_score an extra 2400 evaluation comparing to the original AB_Improved agent.

Custom_score is able to beat AB_Improved for most of the time. The agent is tested against AB_Improved with a series of 3 tournament (800 games each) added up to 2400 games. The improved heuristic won 2 of the 3 tournaments.

Custom score #2

Description:

The hypothesis of custom score #2 is that I believe different position of the cell has different strategic value to the winning of the game. So in this heuristic, for each cell in the game, a score is computed based on the correlation of winning the game and it is then added to the result of custom score #1 heuristic for the final score.

Implementation:

For the game with a certain size, a list of float numbers are created to store the score for each cell, every cell starts with score 0. I run the game between two AB_Improved agent for ~400 times, for each game, if player 1 wins, every move of player 1 will get a +1 in the score list and every move of player 2 will get a -1. After the game run, each number in the list is divided by number of games.

```
scores = [
  -0.01, -0.02, -0.01, 0.04, -0.05, -0.02, -0.07,
  -0.05, 0.03, 0.09, 0.07, -0.03, 0.00, -0.02,
  0.08, 0.04, 0.03, 0.02, 0.05, 0.08, 0.01,
  0.07, 0.06, 0.12, 0.19, 0.03, 0.05, -0.04,
  0.03, 0.07, 0.04, 0.08, 0.09, -0.02, -0.06,
  -0.10, -0.02, 0.11, -0.03, -0.02, -0.03, -0.07,
  0.01, -0.07, -0.04, 0.04, -0.04, -0.07, -0.12]
```

So, clearly some positions are more likely to lead to a winning game. At this stage, I'm able to get the equation for the final score:

Y = custom_score_#1 + score[current_move] * X

In order to find the perfect X for the equation, a second round of test run is conducted. A series of 60-game run is tested against X value from [0, 20) with interval of 2.

X	Winning rate (%)
0	48.3
2	50.8
4	54.2
6	47.5
8	50.8
10	52.5
12	51.7
14	52.5
16	44.2
18	42.5

There's a pretty clear bell curve for the value of X peaked at 4. To make it even more precise, a 3rd round of series game for x value from [3, 5) is tested again.

X	Winning rate (%)
3.0	45.0
3.2	45.0
3.4	48.3
3.6	51.7
3.8	45.0
4.0	60
4.2	48.3
4.4	50.0
4.6	56.7
4.8	43.4

From the result, I picked 4.0 as the x for the final equation. Notice that you actually see lower winning rate with the same band of X from table 2 than in table 1, this indicates that 60 games isn't enough to reach statistical significance. To get more reliable scores, a new test with significant more games is recommended. Another limitation is that, this methodology is limited to 7*7 board only, different size of the game board needs to be trained separately.

Game result:

After more than 3000 game run, custom_score_#2 consistently beats the original AB_Improved with a small margin (1%-4%), which proves the success of this heuristic.

Custom score #3

Description:

The goal of custom_score is trying to monitor the density of the game board while playing and trying to push the opponent to the denser area of the game.

After every game ends, it's easily noticeable that the occupied cells are oftentimes not evenly distributed in the game board, with the center of the density skewed towards one side of the game. Based on this observation, an experiment is conducted to test whether it's feasible to push the opponent to the denser corner of the game and thus limit the number of legal moves

for the opponent. However, the result shows this strategy ineffective against AB_Improved and led to a big loss of the game.

Implementation:

K-means clustering is used to identify the center of the current occupied cells for the entire game board. The target corner is identified based on the center. In addition, because the size of the game board is small, outlier cells might skew the density center significantly. To address this problem, DBScan is used to remove the outliers before piping the cells to k-means. The evaluation function returns the combination of custom_score_#1 and the distance between the current opponent to the target corner.

Game result:

The performance of custom_score is pretty bad with a 40% win rate against AB_Improved for 400 games. Because AB_Improved is always aim for the moves that give it more legal_moves in the future and corner is almost 100% of times opposite of that, custom_score_3 is never able to successfully lock it towards the corner. Plus computing k-means is time consuming, which limit the depth of the search tree.