A review of some evaluation heuristics for the game of Isolation

By Ezenwere I.K Emmanuel

Custom_score : the custom score uses the probability of winning as an evaluation heuristic for board states. When compared to AB_Improved it performed better with an average of 70.29% win rate while AB_Improved had a 68.29% Win rate. It was a slightly increased performance. Besides having the best performance compared to the rest, I choose this because I'm convinced that having a stochastic heuristic function is the right way to go, especially because we can't be 100% certain of the outcome of a game without searching to end-game.

Also I found it beautiful the consistency this function gives when we eventually search to end game, here's how:

Assuming we search to end game, we have S moves and our opponent is isolated and has no more moves left,
P(winning) = S/(S+0) = 1

While if we are isolated and our opponent has S moves then
P(Winning) = 0/(0+S) = 0

And that just makes logical sense.

Here's the program:

```
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)))
    win_probability = float(own_moves / (own_moves + opp_moves))

    return win_probability
```

Custom_score_2 : the custom score 2 square(my_moves) - square(opp_moves), ,this performed better than AB_Improved on average with an average win rate of 70.65% while AB_Improved had 68.29%. The motivation behind this heuristic was to magnify number of moves of the respective players in other to have a greater contrast between relative board advantage of each player.

```
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)))

    return float(own_moves**2 - opp_moves**2)
```

Custom_score_3 : the custom score 3 probability of winning / (my_moves - opp_moves), ,this performed slightly less than AB_Improved on average with an average win rate of 62.14% while AB_Improved had 68.29%.
The motivation behind this was to come up with a heuristic that uses some form of probability and the same algorithm for AB_Improved, (my_moves - opp_moves). This didn't perform so much as expected.

```
def custom_score_3(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)))
    score1 = float(own_moves / (own_moves + opp_moves))
    score2 = float(own_moves - opp_moves)

    if score1 == 0:
        return float("inf")
    else:
        return score2 / score1
```

Win rate 1	AB_Improved	AB_Custom	AB_Custom_2	AB_Custom_3
1	58.6	61.4	65.7	60.0
2	68.6	64.3	70.0	65.7
3	74.3	72.9	78.6	54.3
4	55.7	70.0	77.1	67.1
5	67.1	77.1	65.7	55.7
6	71.4	74.3	71.4	61.4
7	67.1	68.6	67.1	58.6
8	71.4	68.6	67.1	62.9
9	75.7	71.4	72.9	71.4
10	72.9	74.3	70.0	64.3

Average win rate after 10 batches	68.29	70.29	70.56	62.14

Remark:

I'm convinced that the right way to go about this is using deep learning; here's the proposed architecture, by using a deep neural network to generate some initial random function that maps an NxN (board dimension) matrix representation of the board to a single float, next using this same random function as our heuristic function we play the game a 1000 times, with cost function C = (100 - win_rate) reflecting the percentage of wins, next we back propagate and repeat this process until we can get C to converge to 0.

At this point we will have a representation that most accurately maps any given board state to the most accurate evaluation score.

I spent a majority of my time on this project designing such a neural network but I couldn't implement this eventually because I couldn't find out how to express backpropagation in this scenario where I don't have labels and input features beforehand.

I'm still working on this and I'm currently studying the Alpha Go paper to understand if and how they handled this challenge.