Heuristic Analysis

Victor Osório

I have choose 3 approaches for my heuristic. From there tree approaches, it was not possible to build a heuristic from the first. And from the last approach, I build two heuristics.
After that, I choose my default heuristic based on the execution of tournament. The results are shown at the end.

Choosing a position based on the position

For a first approach, I have try to build an heuristic based on the position of the player. These heuristic allow to the player avoid the borders and keep close to the middle. The results were not satisfactory, so I do not choose continue with this approach.

Choosing a position based on available moves

The next approach is thinking: how can I use the possible moves information better?
I can see that **Open Move Score** and **Improved Score** heuristics already use that information, but they treat the information as a plain number, not a set of position.
If we consider this information as a set and not just a number, we can extract more information from it.

Player Moves	Moves that the player can do
Opponents Moves	Moves that the opponent player can do
Common Moves	Moves that both players can do
Exclusive Opponent Moves	Moves that only the opponent can do

So, the first step for this heuristic is treat this data as a set, not as a list. Python gives to us a great library for sets, so it is easy to calculate common moves. With the lines below we can easily extract the four sets:

```
player_moves = set(game.get_legal_moves(player))
opponent_moves = set(game.get_legal_moves(game.get_opponent(player)))
common_moves = player_moves - opponent_moves
exclusive_opponent_move = opponent_moves - common_moves
```

Playing the game in a paper, I choose the strategy of avoid the `common_moves` at the begging and choosing this movements when we have fewer options.
Now we have to create a formula with these four sets for a better performance. With the information of common moves we have two different approaches. We can avoid positions

where both players have common moves. Or we can purposely steal opponent's moves. I'm using a `deep_factor` that avoid common moves in the begging of the game and look for positions with common moves in the end of the game.
I choose a faster way to calculate the score to do not waste time. As fast as the score is calculated, more node can be expanded in Iterative Search.

```
return float(len(player_moves) + len(common_moves) * game.move_count
                - len(common_moves) / (game.move_count + 1.0)
                - len(opponent_moves - common_moves))
```

Choosing a position based on possible movements

For our all heuristics, the Horizon Effect is still a problem. To handle that problem, in paper playing we always evaluate if the choose position is a final position or can lead us to other movements.
So, with this mindset I build the `custom_score_2` calculating a weight based on how many moves that position can give to the player.

```
def possible_move_weight(game, move):
    r, c = move
    directions = [(-2, -1), (-2, 1), (-1, -2), (-1, 2), (1, -2), (1, 2), (2, -1), (2, 1)]
    valid_moves = [(r + dr, c + dc) for dr, dc in directions
                   if game.move_is_legal((r + dr, c + dc))]
    return len(valid_moves)
```

With this weight, we can extend the given **Improved Score**:

```
def custom_score_2(game, player):
    if game.is_loser(player):
        return float("-inf")
    if game.is_winner(player):
        return float("inf")
    opponent_moves = game.get_legal_moves(game.get_opponent(player))
    player_moves = game.get_legal_moves(player)
    return float(len(player_moves) * sum([possible_move_weight(game, m) for m in
player_moves])
              - len(opponent_moves) * sum([possible_move_weight(game, m) for m in
opponent_moves]))
```

With this just looking if the current step has possible next steps gives to us a great gain in the Win Rate.

Improving custom_score_2

With the same information from custom_score_2, just looking for all possible movements from next step, we can build a better score.
For this score, we are not extending the **Improved Score**, but the **Open Move Score**. The possible_move_weight for the opponent position gives to us a correct information about the current state of the game, so this information can replace `opponent_moves`.
Now we can build a score that grow with the `possible_move_weight` of the player, and decrease with the `possible_move_weight` of the opponent.

```
def custom_score_3(game, player):
    opponent_moves = game.get_legal_moves(game.get_opponent(player))
    player_moves = game.get_legal_moves(player)
    return (len(player_moves) *
            sum([possible_move_weight(game, m) for m in player_moves]) /
            (1 + sum([possible_move_weight(game, m) for m in opponent_moves])))
```

With this simple change on simple score, we have a great gain on the Win Rate.

Results

The tournament execution present a high error. So when I was executing it I changed the numbers of Matches to 20. The results are listed below. In other matches there was different results from for all opponents. In some tests AB_Custom wins, in others AB_Custom_2 wins and in others AB_Custom_3 wins. But in all, my heuristics are better than AB_Improved.

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

Match #   Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
                       Won | Lost   Won | Lost   Won | Lost   Won | Lost
   1       Random       9  |   1     8  |   2     9  |   1    10  |   0
   2       MM_Open      8  |   2     8  |   2     7  |   3     9  |   1
   3      MM_Center     7  |   3     5  |   5     7  |   3    10  |   0
   4     MM_Improved    5  |   5     7  |   3     6  |   4     7  |   3
   5       AB_Open      3  |   7     5  |   5     7  |   3     5  |   5
   6      AB_Center     7  |   3     6  |   4     5  |   5     6  |   4
   7     AB_Improved    4  |   6     5  |   5     6  |   4     4  |   6
--------------------------------------------------------------------------
          Win Rate:      61.4%        62.9%        67.1%        72.9%
```

The chosen heuristic

I had chosen the `custom_score_3` for the following reasons:

- It has presented better performance in tournaments

- It is a simple to calculate heuristic. When the score take too long time calculating his value, it waste time from player searching the tree.
- It allow AlphaBetaPlayer search in more nodes per move.

In the table below it is presented a tournament with 40 matches each round.

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

Match #   Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
                       Won | Lost   Won | Lost   Won | Lost   Won | Lost
   1       AB_Open     26  |  14    22  |  18    22  |  18    23  |  17
   2      AB_Center    22  |  18    27  |  13    24  |  16    27  |  13
   3     AB_Improved   22  |  18    21  |  19    24  |  16    20  |  20
   4      AB_Custom    19  |  21    21  |  19    21  |  19    23  |  17
   5     AB_Custom_2   17  |  23    19  |  21    24  |  16    24  |  16
   6     AB_Custom_3   18  |  22    19  |  21    21  |  19    19  |  21
--------------------------------------------------------------------------
          Win Rate:      51.7%        53.8%        56.7%        56.7%
```

We can see in the image below that the `custom_score` is faster, but it `custom_score_3` has more information about the board and is faster than `custom_score_2`. So it has a better performance. If we can get the same information from the board more fast, we can have a better performance.

```
--- custom_score: 4.696846008300781e-05 seconds ---
--- custom_score_2: 0.0001251697540283203 seconds ---
--- custom_score_3: 0.00011920928955078125 seconds ---
```