Heuristic Analysis

By Michael Shum

Heuristics (in reverse order):

Heuristic #1 (AB_Custom_3): Available Moves

The first heuristic value we tried (and placed in Custom_3) simply returned the difference between available moves to the active player and the opponent player. Having more moves than your opponent should be beneficial, and this established a nice baseline evaluation method to compare vs AB_Improved.

Code:

```
if game.is_loser(player):
    return float('-inf')
elif game.is_winner(player):
    return float('inf')

my_moves = len(game.get_legal_moves(player))
opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
return float(my_moves - opp_moves)
```

Heuristic #2 (AB_Custom_2): Stay In The Center

The second heuristic value (Custom_2) made an attempt to differentiate each move by its position on the board. Given that having more available moves is beneficial, then moves along the walls naturally cut down on available options. Corners are particularly bad with fewer available moves even in an empty board. This evaluation function goes through all the available moves and assigns them a score with the center square worth the most (14 points), and corner moves worth the least (2 points).

Here is a visualization of the scores per position:

Row/Col	0	1	2	3	4	5	6
0	2	4	6	8	6	4	2
1	4	6	8	10	8	6	4
2	6	8	10	12	10	8	6
3	8	10	12	14	12	10	8
4	6	8	10	12	10	8	6
5	4	6	8	10	8	6	4
6	2	4	6	8	6	4	2

Code:

```
if game.is_loser(player):
    return float('-inf')
elif game.is_winner(player):
    return float('inf')

my_moves = game.get_legal_moves(player)
opp_moves = game.get_legal_moves(game.get_opponent(player))

my_score = 0
opp_score = 0

for move in my_moves:
    my_score += 14 - abs(move[0] - 3) * 2 - abs(move[1] - 3) * 2
for move in opp_moves:
    opp_score += 14 - abs(move[0] - 3) * 2 - abs(move[1] - 3) * 2

return float(my_score - opp_score)
```

Heuristic #3 (AB_Custom): Playing conservatively in the late game

The final heuristic we used (custom_score) takes the "Stay in the Center" evaluation function used in Custom_2 but differentiates between what stage in the game the player is in. In the early game, where less than 40% of the spaces have been taken, the evaluation function favours moves that are more centered but at a scale lower than Custom_2. However, in the later game, where at least 40% of the spaces have already been taken, the evaluation function plays more conservatively and weights moves that favour the center much higher than moves on the walls and corners.

Code:

```
if game.is_loser(player):
    return float('-inf')
elif game.is_winner(player):
    return float('inf')

my_moves = game.get_legal_moves(player)
opp_moves = game.get_legal_moves(game.get_opponent(player))

my_score = 0
opp_score = 0

#Early game, can be less conservative
if len(game.get_blank_spaces()) < 20:
    for move in my_moves:
        my_score += 7 - abs(move[0] - 3) - abs(move[1] - 3)
    for move in opp_moves:
        opp_score += 7 - abs(move[0] - 3) - abs(move[1] - 3)
else:
    for move in my_moves:
        my_score += 21 - abs(move[0] - 3) * 3 - abs(move[1] - 3) * 3
    for move in opp_moves:
        opp_score += 21 - abs(move[0] - 3) * 3 - abs(move[1] - 3) * 3

return float(my_score - opp_score)
```

Results:

Run #1:

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

Match #   Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
                      Won | Lost   Won | Lost   Won | Lost   Won | Lost
   1       Random      8  |  2     10  |  0     9  |  1     9  |  1
   2       MM_Open     8  |  2      7  |  3     8  |  2     6  |  4
   3      MM_Center    8  |  2      9  |  1     9  |  1     8  |  2
   4     MM_Improved   8  |  2      9  |  1     8  |  2     7  |  3
   5       AB_Open     5  |  5      6  |  4     4  |  6     4  |  6
   6      AB_Center    5  |  5      4  |  6     5  |  5     6  |  4
   7     AB_Improved   4  |  6      5  |  5     5  |  5     6  |  4
----------------------------------------------------------------------
          Win Rate:      65.7%        71.4%        68.6%        65.7%
```

Run #2:

```
                    **************************
                         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      9  |  1     10  |  0     10  |  0
   2       MM_Open     7  |  3      8  |  2     8  |  2     7  |  3
   3      MM_Center    8  |  2     10  |  0     9  |  1     8  |  2
   4     MM_Improved   7  |  3      9  |  1     6  |  4     6  |  4
   5       AB_Open     4  |  6      6  |  4     3  |  7     5  |  5
   6      AB_Center    5  |  5      5  |  5     6  |  4     6  |  4
   7     AB_Improved   4  |  6      5  |  5     7  |  3     6  |  4
----------------------------------------------------------------------
          Win Rate:      62.9%        74.3%        70.0%        68.6%
```

We ran tournament.py multiple times while building the evaluation functions, but these two final runs were made on the final custom score functions. It shows that combining weighted move scores with consideration for what point in the game the player is in, results in the best results with AB_Custom achieving the highest win rate in both runs.

AB_Custom outpaced AB_Improved by 5.7% and 11.4% respectively, with the other two heuristics generally doing better than AB_Custom as well.

Recommendation

We recommend using the AB_Custom evaluation function, which consistently had the best results in win rate, and strikes a good balance between a better heuristic while maintaining simplicity. Here are the three main reasons for choosing this evaluation function:

Win Rate:

The AB_Custom had a much higher win rate than AB_Improved and the other heuristics listed here. This was consistent through multiple runs, and through tweaking multipliers to make the impact of picking center squares vs walls/corners more pronounced.

Speed:

To allow iterative deepening to get as deep into the game tree as possible, we opted for a simpler heuristic that would be quick to execute and reduce the number of timeouts encountered. The heuristic runs in constant time, and the runtimes of AB_Custom and AB_Custom_3 are almost identical since the number of legal moves to each player is constant in big-O notation terms. Being able to get to the end of game trees is invaluable (represented by the infinite scores returned in win/loss states), and since the number of potential moves in knights isolation is much less than queens isolation, reaching end nodes is possible as long as the heuristic is quick enough.

Strategy:

This evaluation function was based on two main strategies: identifying high value positions and adopting different behaviours depending on what stage the game was in. By doing playtests and through analysis of the gameboard, moves closer to the center of the board were shown to be more valuable since it gave the most future flexibility and resulted in the deepest game tree branches. Intuitively, moves in the center of the board which allowed for 8 potential moves, were much better than moves in the corners which allowed for only 2 potential moves. Weighting each move proved to be much more effective than treating all moves the same. Playing more conservatively in the second half of the game, by emphasizing moves in the center of the board, also improved this evaluation function considerably versus evaluation functions that did not change behavior depending on board state.