Approach

With my approach to providing an evaluation function, I iterated over a multitude of different approaches. After the initial batch of testing, more simplistic evaluation functions outperformed the more comprehensive evaluation functions and were subsequently scrapped. The three evaluation functions `AB_Custom()`, `AB_Custom_2()`, and `AB_Custom_3()` are the result of improving upon the simpler heuristics from that initial round of testing.

```
improved_score() from sample_players.py

    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 - opp_moves)
```

Each evaluation functions starts with `improved_score()` as a foundation and builds upon it by including additional variables based on features of the game state and modifiers to said variables (in the form of constants, exponents, etc.). This rationale stemmed from the poor performance of more comprehensive evaluation functions when tested against `improved_score()`. This revealed the computational trade-offs were not worth it, in this case, contributing to the emphasis of simplicity in further development of evalution functions.

Notable for its usage in two of the three custom evaluation functions are “tertiary moves”. We define tertiary moves as the set of legal moves A'_L available to our player in a gamestate s' after applying an action $a \in A_L$ to a gamestate s, where a is a move in the set A_L obtained from calling the function `game.get_legal_moves(player)`.

```
get_tertiary_moves(game, legal_moves) from game_agent.py

MOVEMENT_VECTORS = [(1, -2), (1, 2), (-2, -1), (-2, 1),
        (-1, -2), (-1, 2), (2, -1), (2, 1)]

def get_tertiary_moves(game, legal_moves):
    tertiary_moves = {(lm_y + v_y, lm_x + v_x)
                      for v_y, v_x in MOVEMENT_VECTORS
                      for lm_y, lm_x in legal_moves
                      if game.move_is_legal((lm_y + v_y, lm_x + v_x))}
    return tertiary_moves
```

The set of tertiary moves is determined by applying a set of vectors representing the legal movement of a Knight to the player’s position in s' and determining if that is valid. The validity is dependent solely upon whether the position is within the dimensions of the game board, and if the position is vacant – it does not account for the possiblity of the opponent moving to that position following s'.

Implementation

```
all custom_score() implementations include the following

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

As previously mentioned, each heuristic treats `improved_score()` as an initial starting point. With that, the above snippet is common to each of them meaning each treats terminal game states in the same way.

```
custom_score(game, player) from game_agent.py

    player_pos      = game.get_player_location(player)
    legal_moves     = game.get_legal_moves(player)
    tertiary_moves  = get_tertiary_moves(game, legal_moves)
    opponent        = game.get_opponent(player)
    opponent_pos    = game.get_player_location(opponent)
    opponent_moves  = game.get_legal_moves(opponent)
    return (len(legal_moves)
            + 0.5 * float(len(tertiary_moves)
                  / get_manhattan_distance(player_pos, opponent_pos))
            - len(opponent_moves))
```

With `custom_score()`, we have the most obvious case from which `improved_score()` served as the foundation. This heuristic is identical to `improved_score()`, while the number of tertiary moves grows in importance as the player and its opponent near each other. The constant `0.5` was determined through experimentation.

```
custom_score_2(game, player) from game_agent.py

    opponent        = game.get_opponent(player)
    player_pos      = game.get_player_location(player)
    opponent_pos    = game.get_player_location(opponent)

    return float(len(game.get_legal_moves(player))**2
                / (get_manhattan_distance(player_pos, opponent_pos)
                   + len(game.get_legal_moves(opponent))**2))
```

`custom_score_2()` works similarly but maintains an inverse relationship between the number of legal moves available to the player and the opponent. This heuristic servers as an attempt at maintaining proximity to the opponent after testing indicated the failings of other heuristics against the centering heuristic. Tertiary moves are not included so as to save computation time for iterative deepening.

```
custom_score_3(game, player) from game_agent.py

    player_pos       =  game.get_player_location(player)
    legal_moves      =  game.get_legal_moves(player)
    tertiary_moves   =  get_tertiary_moves(game, legal_moves)
    opponent         =  game.get_opponent(player)
    opponent_pos     =  game.get_player_location(opponent)
    return (float((0.5 * len(legal_moves) + len(tertiary_moves))
                 / get_manhattan_distance(player_pos, opponent_pos))
             - len(game.get_legal_moves(opponent)))
```

With `custom_score_3()`, tertiary moves are given precedence over the set of legal moves. The constant of `0.5` was arrived at through testing.

Results

Testing revealed inconsistent results hence the additional rounds, as well as incremental matches per round, of testing. Testing was initially limited to opponents utilizing alpha-beta pruning, with the exception of the random agent. Mini-max agents were added back in for the sake of thoroughness during what was intended to be a final round of testing, only to reveal further inconsistencies in terms of the winningest evaluation heuristic.

```
**********************************************************************
*                              LEGEND                                *
*   (of AB_Custom, AB_Custom_2, AB_Custom_3 wins against Opponent)   *
*   --------------------------------------------------------------   *
*                        *: 1st most wins                            *
*                        +: 2nd most wins                            *
**********************************************************************

                      *************************
                         Playing 150 Matches
                      *************************

                               Round 1

 Match #    Opponent     AB_Improved    AB_Custom    AB_Custom_2  AB_Custom_3
                          Won | Lost    Won | Lost    Won | Lost   Won | Lost
    1        Random       153 |   7     149 |  11    +151 |   9   *155 |   5
    2        AB_Open      85  |  75    *95  |  65    +81  |  79    78  |  82
    3       AB_Center     94  |  66    +94  |  66    *101 |  59    93  |  67
    4      AB_Improved    81  |  79    *88  |  72    +84  |  76    78  |  82
--------------------------------------------------------------------------
           Win Rate:       64.5%      *  66.6%      +  65.2%        63.1%
```

The first round of testing revealed that `custom_score()` was the most winningest evaluation heuristic, on average. With considering the results of all but those against the Random opponent, `AB_Custom` exhibits an even higher win rate than `AB_Custom_2` and `AB_Custom_3`.

```
                         *************************
                            Playing 200 Matches
                         *************************

                                 Round 2

 Match #    Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
                         Won | Lost   Won | Lost   Won | Lost   Won | Lost
    1        Random      187 |  13   +183 |  17   +183 |  17   *191 |   9
    2        AB_Open     110 |  90   +105 |  95   *114 |  86    97  |  103
    3       AB_Center    114 |  86    113 |  87   *117 |  83   +116 |  84
    4      AB_Improved   104 |  96   *102 |  98   +99  |  101   98  |  102
---------------------------------------------------------------------------
           Win Rate:       64.4%      +  62.9%     *  64.1%        62.8%

                                 Round 3

 Match #    Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
                         Won | Lost   Won | Lost   Won | Lost   Won | Lost
    1        Random      187 |  13   +187 |  13   +187 |  13   *190 |  10
    2        AB_Open     107 |  93    99  |  101  +101 |  99   *109 |  91
    3       AB_Center    118 |  82   *122 |  78    116 |  84   +117 |  83
    4      AB_Improved   99  |  101  +99  |  101  *101 |  99   +99  |  101
---------------------------------------------------------------------------
           Win Rate:       63.9%      +  63.4%        63.1%     *  64.4%

                                 Round 4

 Match #    Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
                         Won | Lost   Won | Lost   Won | Lost   Won | Lost
    1        Random      193 |   7    182 |  18   *193 |   7   +185 |  15
    2        MM_Open     158 |  42   +150 |  50    148 |  52   *157 |  43
    3       MM_Center    177 |  23    175 |  25   *179 |  21   +176 |  24
    4      MM_Improved   144 |  56   *150 |  50   +148 |  52    140 |  60
    5        AB_Open     104 |  96    97  |  103  *107 |  93   +102 |  98
    6       AB_Center    121 |  79   *119 |  81   +117 |  83    115 |  85
    7      AB_Improved   94  |  106  *112 |  88    103 |  97   +104 |  96
---------------------------------------------------------------------------
           Win Rate:       70.8%      +  70.4%     *  71.1%        69.9%

---------------------------------------------------------------------------
---------------------------------------------------------------------------
       Avg Win Rate
        over Rounds:                     65.6%     *  66.1%      + 65.7%
```

With increasing the number of matches played per round, and reintroducing the mini-max tree traversal, `AB_Custom_2` took the lead. Examining the results of each round, however, appears to delegitimize these results as indicative of the efficacy of the evaluation heuristics and, instead, to be the result of favorable RNG outcomes. Due to this, the number of matches was increased further for a final 5th round.

```
                  ************************
                     Playing 1000 Matches
                  ************************

                           Round 5

 Match #   Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
                        Won | Lost   Won | Lost   Won | Lost   Won | Lost
    1       Random      929 |  71    933 |  67   *937 |  63   +934 |  66
    2       MM_Open     759 |  241  *778 |  222   752 |  248  +755 |  245
    3      MM_Center    862 |  138   860 |  140  +870 |  130  *871 |  129
    4     MM_Improved   726 |  274  *760 |  240  +734 |  266   728 |  272
    5       AB_Open     525 |  475   542 |  458  *558 |  442  +550 |  450
    6      AB_Center    558 |  442  +561 |  439   558 |  442  *565 |  435
    7     AB_Improved   482 |  518  +503 |  497   489 |  511  *518 |  482
--------------------------------------------------------------------------
           Win Rate:      69.2%     *  70.5%       70.0%     +  70.3%

--------------------------------------------------------------------------
--------------------------------------------------------------------------
           Win Rate:                *  66.7%       66.0%     +  66.4%
      (excluding Random)
```

Surprisingly, all custom evaluation heuristics performed better than `AB_Improved`, albeit to a minor degree. Isolating the results of this final round, `AB_Custom` exhibits the highest win rate.

Selection

Based upon the results of Round 5, as well the win rates when excluding the Random agent – as I would not expect randomization to be the sole algorithm of any effective evaluation heuristic, I recommend the usage of the `AB_Custom` evaluation heuristic. This is due to its overall performance (its win rate) during Round 5's testing, it's win rate with excluding the Random agent for the rationale cited above, as well as it consistently either having the highest or second highest win rate over all five rounds of testing.