Explanation of the three heuristics I submitted

`custom_score` - Improved heuristic with 2/1 weights

`custom_score`, my best heuristic, was inspired by my observation that the Open and Improved algorithms seemed to perform about equally well when I ran them against each other using `tournament.py`. From this observation, I hypothesized that maximizing the number of possible own moves was more important than minimizing the number of possible moves for your opponent, although both are important. Based on this, I created a heuristic that treated the number of own moves as twice as important as the number of opponent moves, then maximized the spread between these two values just like the Improved heuristic.

`custom_score_2` - Warnsdorf's Rule

When brainstorming heuristics, one of the things I realized is that Isolation with knights is just an adversarial version of a Knight's Tour. One of the approaches listed on Wikipedia for solving a Knight's Tour with a computer is by repeatedly applying something called Warnsdorf's Rule. Warnsdorf's Rule is that when selecting the next board position for your knight you should always select the move that results in the fewest subsequent moves from that new position. I realized that this was essentially the opposite heuristic to the Open moves heuristic. So for my implementation of this heuristic I just returned the negation of the number of open moves from any given board position.

`custom_score_3` - Minimize opponent's open moves

This was inspired by the Open and Improved heuristics. After testing those heuristics, an obvious question seems to be "what happens if you only care about minimizing your opponent's possible moves?" To me this seemed equally valid a strategy as maximizing your own possible moves, and I was curious how it would compare to Open and Improved.

How I chose my best heuristic

I recommend my `custom_score` heuristic as a good heuristic for the following reasons:

- It performed the best in the tournament I ran, which played 100 matches against the best agents in `sample_players.py`.
- The computational cost is extremely low, which allows the search algorithm to search more of the state space. The slowest part of the heuristic calculation is finding the legal moves for both players, which runs in O(n), where n is the number of board spaces.
- The heuristic does not search the game tree to a depth beyond the current move. This means the heuristic is less "smart", but also ensures that it will always return a value quickly.
- Both the state of the active player and opponent are considered in the heuristic evaluation. The number of moves available to the active player is considered to be twice as important as the moves available to the opponent.

```
Match #   Opponent    AB_Improved    AB_Open     AB_Custom   AB_Custom_2  AB_Custom_3
                       Won | Lost   Won | Lost   Won | Lost   Won | Lost   Won | Lost
   1       AB_Open     48  |  52    54  |  46    54  |  46    43  |  57    49  |  51
   2      AB_Center    59  |  41    53  |  47    59  |  41    46  |  54    56  |  44
   3     AB_Improved   42  |  58    49  |  51    53  |  47    38  |  62    44  |  56
----------------------------------------------------------------------------------
          Win Rate:      49.7%        52.0%        55.3%        42.3%        49.7%
```