Isolation Heuristic Analysis

Knowing that the benchmark for the test agents that my agents were going to compete with used the the difference between each player's open moves as the heuristic for the evaluation function, I first focussed on the underlying features own_move and opp_moves, determined that wasn't the best relationship between those features, explored other features, centerness and openness, and determined optimal weighting for each feature by utilizing a grid search methodology. In the end, I was able to create an evaluation function, **AB_Custom**, that outpeforms the benchmark, **AB_Improved**, by 5.7% (3.9% more wins when NUM_MATCHES = 100).

Results at NUM_MATCHES = 100

Fig 1: Win Rates

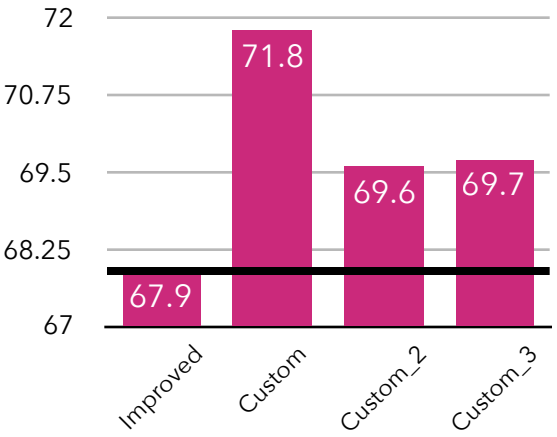
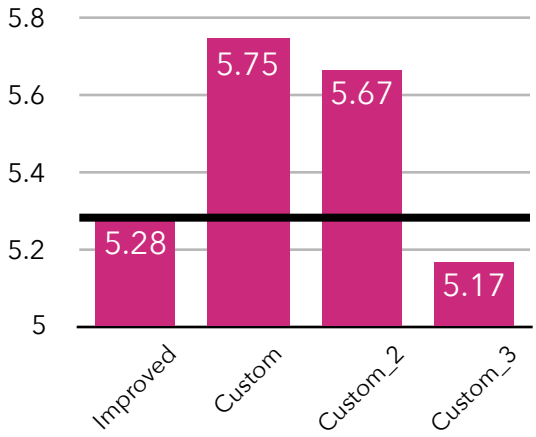


Fig 2: Average Search Timeout Depths



As described above, I initially realized that a weakness in the benchmark evaluation function, **AB_Improved**, is that it doesn't represent the advantage well when the opponent's open moves is near the boundaries. For example, when the player has 8 moves and the opponent has 0, the difference evaluation function would return a utility value of 8. In actuality, this should represent a terminal state with player as the winner and the evaluation function should return *infinity*. Likewise, at the other extreme, when both players have 8 moves, the difference function would return 0, but, given that player still has an ample choice in moves, this overestimates the negative position of player. Further, when the opponent player has more moves than the player, e.g. 6 for the opponent and 3 for the player, the difference equation yields -3, which doesn't seem right when the player still has 3 moves. A functional relationship between these two features utilizing a ratio more accurately describes the my intuition about the relative utility at different number of open moves. Additionally, it depresses the utility value when the player has more moves than the opponent, but the opponents still has more than 1 move—a scenario it seems more accurately portrays the player's advantage. Theoretically, this all seems sound, but should be reinforced by testing. Comparing

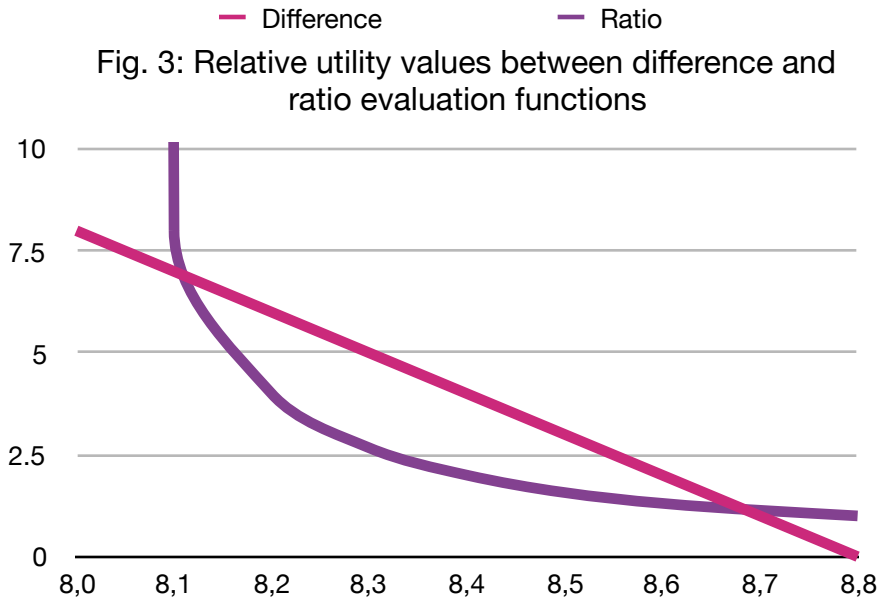


Fig. 3: Relative utility values between difference and ratio evaluation functions

AB_Improved with **AB_Custom_3,** which uses the simple ration as the evaluation function, you can see that this simple relational difference increases the performance significantly by 2.7% (1.8% more wins when NUM_MATCHES = 100). Note, one other difference between **AB_Improved** and **AB_Custom_3**, as is done with all of my custom evaluation functions, is that the function returns *positive infinity* immediately if the opponent has 0 open moves and *negative infinity* if the player has 0 open moves. This could also contribute to the increase in performance.

Grid Search

As mentioned above, a good deal of my exploration was spent considering different features of the game state and evaluating how important of a role they played in contributing to an optimized utility function. In order to more efficiently evaluation different weights for different features, I implemented a grid search function that dynamically created evaluation utilizing a different set of feature weights and then added agents using this functions to the tournament test agents. Using this approach allowed me to quickly uncover some relationships between certain features and zero in on optimal weights for each feature. Note that I used a modified tournament.py, tournament_mp.py, that incorporates multiprocessing to facilitate quicker feedback loops.

Search Depth Matters

Finally, as you can see in Figure 2, there is a strong correlation between how deep in the game tree a given agent is able to go before reaching the search timeout and the overall performance on the win_rate metric. During my grid search evaluations, I noticed that while a more complicated evaluation function might better capture the true utility of a game state, there was a tradeoff that prevented the agent from searching as far before the timeout. This led to poorer performing agents overall. Therefore, finding the most efficient evaluation function that best captured the utility became the name of the game.