Heuristics Analysis

Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
	Won	Lost	Won	Lost	Won	Lost	Won	Lost
Random	95	5	93	7	92	8	87	13
MM_Open	71	23	69	31	70	28	81	19
MM_Center	86	14	89	11	85	15	92	8
MM_Improved	66	34	72	28	73	27	70	30
AB_Open	54	46	43	57	51	49	47	53
AB_Center	60	40	57	43	61	39	51	49
AB_Improved	51	49	61	39	50	50	43	57
Win Rate:	**69.60%**		**69.14%**		**69.05%**		**67.29%**	

As part of the Isolation project, I've implemented three custom heuristic functions to evaluate a given state of a game for a player. The table above is a summary of the results my custom functions had competing against the different pre-defined agents.

My main goal when designing and evaluating my custom heuristic functions was to learn as much from actually playing the game as a human as possible, and find where I thought there were strategies that maximize my chances of winning. What I found, and what my custom functions mostly played off, was that a position in the corner bad minimized the given player's chance of winning. In a corner, there are less possible spaces around, so there are typically less possible moves. Ultimately, I want my player to maximize its number of moves. Part of my testing and analysis was testing my functions against the predefined agents (especially AB_Improved) and decide whether it had sufficient success.

My most successful custom heuristic, AB_Custom, had the highest winning percentage of my custom heuristics, and also performed the best against AB_Improved, winning an impressive 61% against the agent. This heuristic applies a weight to the scoring function depending on the possible legal moves for both the given player and its opponent. For example, if the given player has more possible corner moves than its opponent, a negative weight is applied, and vice-versa. The reasoning behind this is that if the given player has more corner moves, this presents a bad scenario for the given player because corner moves are bad. We want to minimize our score in this instance because we do not want our given player to end up in a corner. On the other hand, we add a positive weight if the given player's opponent has more corner moves because the opponent ending up in a corner is a good scenario for the given player. If both players have the same number of moves that end in a corner, no weight is applied, and the score is simply the difference between the given player and opponent's legal moves.

The second most successfully custom heuristic, AB_Custom_1, had an impressive overall winning percentage, but ended up coming out even against AB_Improved. This function, like the previous, is built on the disadvantage of a player ending up in a corner position of the board. If the given player is in a corner, the result is immediately the lowest possible score, negative infinity. Conversely, if the given player's opponent is in a corner, the result is immediately the highest possible score, positive infinity. If neither player is in a corner, the evaluation continues. The function will calculate the number

of moves that do not result in a corner position for each player, and will end up returning a score that is the difference between the given player's net moves and the opponent's net moves.

Lastly, my third custom heuristic, AB_Custom_2, further builds up the corner philosophy. The function first finds the difference between the given player's possible moves and the opponent's possible moves. With that result, the function results that plus the sum of the number of corner moves for the opponent. This essentially applies a positive weight to the open moves functions described for each possible corner move for the opponent. The more corner moves for the opponent, the greater weight applied.

Ultimately, I decided to go with AB_Custom as the function for my agent because it had the highest winning percentage of my custom heuristics, and also performed very well against AB_Improved. Additionally, I felt that the actual evaluation within the function was complex but not time-consuming, which allowed for greater search depth using iterative-deepening. Moreover, I found that applying the weights in the fashion I chose really gave the function great flexibility in its evaluation.