HEURISTIC ANALYSIS

BUILDING A GAME PLAYING AGENT

INTRODUCTION

In this project, we implemented a game playing AI to win at the game Isolation, using adversarial search.

After studying multiple strategies, we defined several heuristics designed to outperform the ID\_improved agent provided by Udacity. Which is based on iterative deepening alpha-beta tree search.

All the heuristics functions used here are consist of combined evaluations of the positions of the players, the numbers of possible moves for each player and a given a state of the board.

This document provides the justification and description of each heuristic and compares their performances against the provided provided ID\_improved heuristic.

**By *Sean Alexander Frenn***

# Heuristic 1

To have a good evaluation function, we decided to focus on the victory for our player against its opponent by using a proportion heuristic which is a division between the number of available moves of the two players plus one (avoiding a division by zero). Comparing based on a ratio allows us to take in account only the relative difference between the players instead of a literal difference as used in AB\_Improved heuristic. The number of available moves of the opponent is multiplied by 1.5 to make a play more aggressive, making the agent follow the opponent.

We defined this heuristic while assuming that the immediate future state of our player is determined by the number of available moves of both agents. Multiplying some of our parameters listed above (tuning our parameters) when running multiple tests allow us to have a more performant heuristic function.

Surprisingly, we obtained a 100% win with this heuristic in several sets of matches.

# Heuristic 2

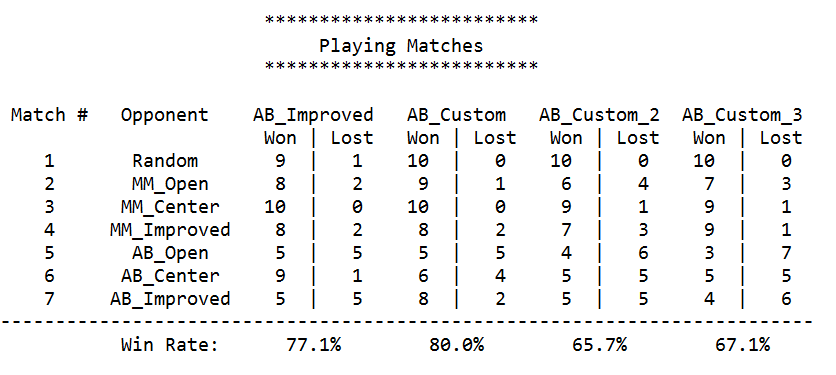
Here we combined all the possible atomic different elements we could think of to evaluate the score of our player against its opponent: the number of available moves for the player and the opponent, the number of common moves, the sum of the centrality of the player moves one state down the game tree, and the number of interfering moves that are common between the player and the opponent. Knowing the interfering moves, we can choose the best move of the opponent reducing its chances of winning the game. We keep cornering the opponent while preserving our good moves which seem to be a pretty good overall strategy for a win.

# Heuristic 3

In this heuristic, we focus on the level of the game state to decide whether we should evaluate according to the number of moves available to the players or the number of common moves available to the player and its opponent (making the game more defensive at the beginning and more aggressing towards its end). This technique allows us to corner to opponent near the end of the game increasing our chances of winning. In addition, we add a weighted score to perform an evaluation which makes the game more offensive than defensive even in the beginning of the game is necessary. We also check linear separations in the game (horizontally, vertically and diagonally) which can potentially let us know before the end of the game if the outcome of the game is known regardless of the number of moves left available to each player. This heuristic performs even better than the first heuristic some of the time.

# PERFORMANCE

Below are two representations of the performance of the heuristics. We can easily compare each two by looking at the comparison graph or directly reading in the table giving the different winning/losing rations over a total of 5 games per agent (or 10 matches between 2 agents).



# Conclusion

Heuristics defined in the game agent are the keys to the winning or losing of the agent.

We saw in our different tests that very good heuristics can be computed allowing the agent to maximize its chances of victory. And heuristics can be more or less performant given the agent they are playing with or against. However, the more complex the heuristic, the more computation time is required and the less deeper search we can perform due to the time limitation constraint we have. Therefore, we recommend evaluation functions that give quite good results within the given allowed time of computation.

To implement the explained heuristic above, refer to the file game\_agent.py

Improving the alphabeta algorithm allows us to drastically improve the performance of the game.

Here we implemented two different transposition table approaches to speed up the tree search.

One is a transposition table used to store the state of the game and the best move at that state for a given depth search. When exploring the game tree for a deeper depth, we restore the game state from the last saved depth and search from there.

The second one is a transposition table used to store the state of each game board after running an alphabeta search. Allowing us to avoid computing the nodes multiple times during a tree search.