Heuristic Analysis of Game-Playing Agent

# Introduction

This project is to finish the code in *game\_agent.py* from <https://github.com/udacity/AIND-Isolation>. The objective is to apply minimax, alphabeta, iterative deepening and heuristics learnt from the AIND course to build “smart” game-playing AI agents.

The code should be able to pass the tests from *sample\_player.py* and *test\_agents.py*. Performance of the heuristics can be tested via the *tournament.py*.

The original *tournament.py* performs 20 rounds of test for each match. In this test, I changed it to 80 for a more stable performance results. Time threshold in *game\_agnet.*py increased slightly to 15 ms. Other parameters remain as default.

# Study on the Baseline of Minimax, Alphabeta and Game Board

Below figure is the performance of ID\_Improved vs MM (Minimax) and AB (Alphabeta) with different heuristics. Random, MM\_Null and AB\_Null perform the worst. Because they either don’t employ any algorithm to find the “best” move, or the heuristic doesn’t take any effect as it’s almost impossible to win/lose the game in 5 moves.

AB is supposed to perform better than MM, but it’s difficult to tell this from below statistics.

ID\_Improved performs the best since it’s using iterative deepening and a better heuristic. In this test, the timeout threshold is set to 15 ms. Thus, it has 150 – 15 = 135 ms to find the “best” move. And usually iterative deepending can reach the depth of at least 7, which is greater than AB’s pre-defined 5. A greater timeout time should be able to improve ID\_Improved’s performance as it can reach a deeper level of search. But I won’t expect too much improvement as the branching factor of 8 is considered as too big.

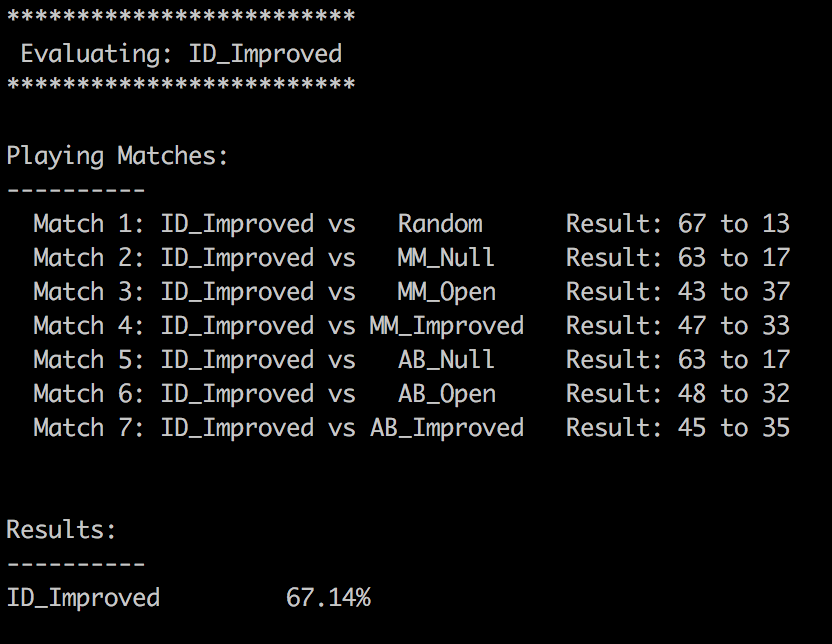


Figure 1. Baseline of ID\_Improved, 80 rounds

# The Three Custom Heuristics

Based on the figure 1, ID\_Improved performs the best because it evaluates not only the active player’s number of legal moves, but it also its opponent’s number of legal moves. The score function of the baseline is *“score = own\_move – opp\_move”.*

So the first heuristic will be exploring what will happen if we make the score function more “greedy”.

## Score = #own\_move – 1.5 \* #opp\_move

Assume there are 2 child nodes with (#own\_move, #opp\_move) value denoted as (5, 3) and (4, 2). In the baseline score, both will return score value of 2 and (4, 2) may be pruned if it’s visited after (5,3). By using this new heuristic function, node with (4, 2) will return a higher score value.

So, below performance result does show a slightly better result. From 67.14% to 70.36%.

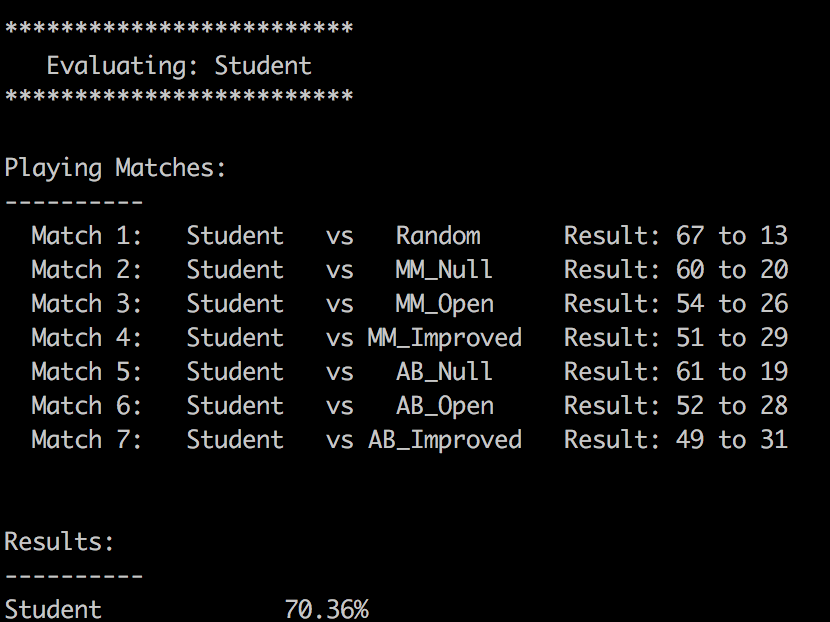


Figure . Result of Student, score = own\_move – 1.5\*opp\_move

And what if we make the score function even more “greedy”? The second try is using the similar way expect 1.5 becoming 3, so I categorize the both as one heuristic. Now *“score = #own\_move – 3 \*# opp\_move”* and the result also improves a little, from 70.36% to 72.68%.

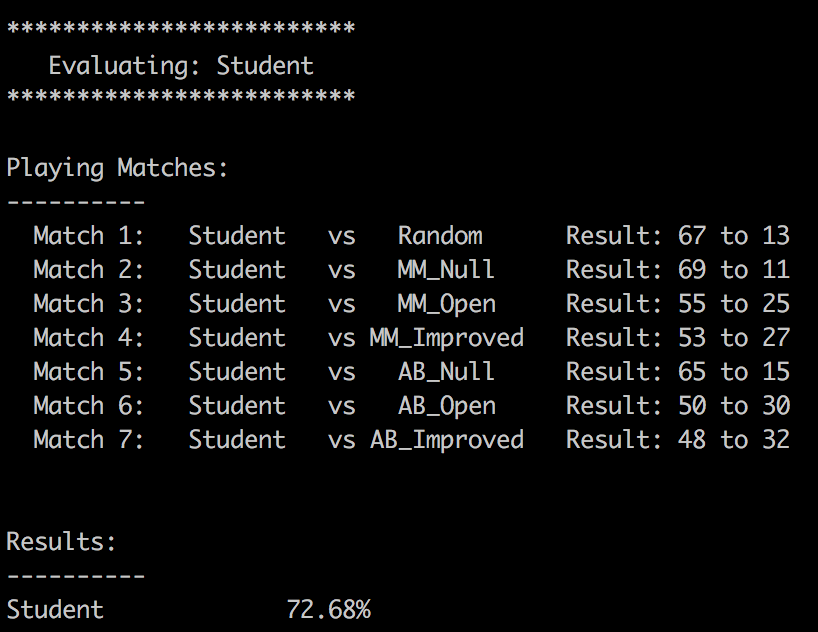


Figure . Result of Student, score = own\_move – 3\*opp\_move

Well, but making it more “greedy” doesn’t mean it always works. A further testing of *“score = #own\_move – 5\*#opp\_move”* didn’t produce a better result.

## Score = #empty neighbors

Another heuristic I can come out is calculating the number of empty neighbors.

The board has 7 \* 7 cells. Each player is making an “L” shape move. It came to my mind that what if the score is equal to the number of empty neighbors of current player. And below relationship must be satisfied.

*abs(row(neighbor) – row(player)) <=2, abs(col(neighbor) – col(player)) <=2*

The reason of choosing 2 is mainly because of the size of the board. If we choose a large number then in many situations the player and opponent share all the neighbors on the board.

The result of student is still greater than 50%, and it performs quite ok when the opponent is playing randomly or using a Null heuristic.

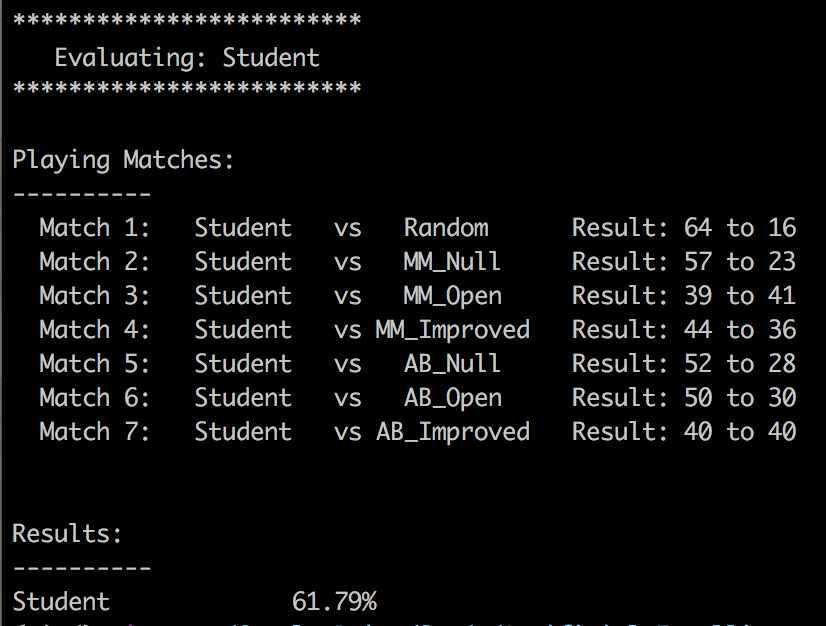
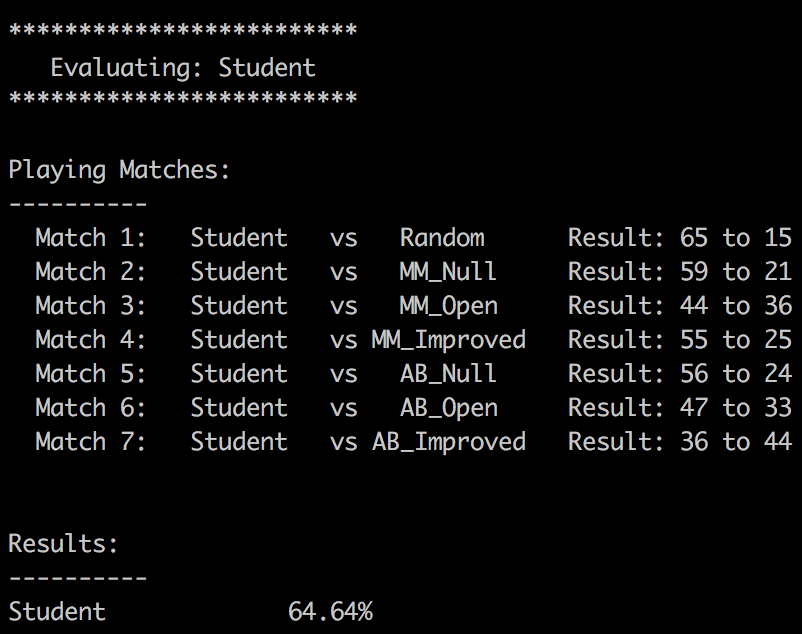


Figure . Result of Student, score = # neighbours

Based on all the performance results collected so far, we can observe that the performance of the agents Random, MM\_Null and AB\_Null are constantly low. Any valid score function can outperform them a lot. My guess is the agents MM\_Null and AB\_Null perform similarly to agent Random. Because the Null function only returns a valid score (either +inf, or –inf) when there is a winner or loser. But this doesn’t seem to happen often after 3 or 5 moves on a board with 7 \* 7 cells.

## Score = #empty neighbors - #opponent’s empty neighbors

The third heuristic function will not only check the player’s empty neighbors, but also its opponent’s empty neighbors. And it’s not a surprise to see it performs better than the previous one. However, not as good as the first custom heuristic.



# Summary and Conclusion

Below table contains all the performance statistics from the earlier section and the evaluation function of Student 1 is the recommended one.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Player**  **Agents** | **ID\_Improved** | **Student 1** | **Student 2** | **Student 3** |
| score = #own\_move - #opp\_move | score = #own\_move - 3 \* #opp\_move | score = #empty neighbors | score = #empty neighbors - #opp empty neighbors |
| **Random** | 83.8% | 83.8% | 80.0% | 81.3% |
| **MM\_Null** | 78.8% | 86.3% | 71.3% | 73.8% |
| **MM\_Open** | 53.8% | 68.8% | 48.8% | 55.0% |
| **MM\_Improved** | 58.8% | 66.3% | 55.0% | 68.8% |
| **AB\_Null** | 78.8% | 81.3% | 65.0% | 70.0% |
| **AB\_Open** | 60.0% | 62.5% | 62.5% | 58.8% |
| **AB\_Improved** | 56.3% | 60.0% | 50.0% | 45.0% |
| **Overall** | 67.1% | 72.7% | 61.8% | 64.6% |

The blue text shows that ID\_Improved and Students constantly outperform Random, MM\_Null and AB\_Null. Main reason in the project is the Null function almost never return a real valid score (+inf or –inf) because it’s not common to have a winner or loser in 5 moves on a 7 \* 7 board. Thus, all the 3 agents perform randomly.

Without any doubt, the student 1 shines the performance among all the testing agents and custom agents with different heuristics. And I can attribute to at least below 3 reasons.

1. The objective of the game is to win. Evaluating the score of oneself is not enough. Because it lacks the status of the opponent. Even if the evaluation function returns a good score of the player, it doesn’t mean the move won’t lead to an even better situation for the opponent. So, a good heuristic must be something that can quantify how much utility from the perspective of the player as opposed to the opponent. That’s also the reason why the Student 2 doesn’t produce a good result as it only evaluates the player’s score.
2. The coefficient 3 in the score function makes it more “greedy” than the ID\_improved. As explained earlier, some searches will benefit because of this extra “greedy”. It’s not always true, but in this project, it does work.
3. The evaluation function is cost effective and efficient. In this project, the time limit of each move is only 150 ms and I set a threshold of 15 ms to allow the program to return the result in time. A much too complicated evaluation function may cost too much time and leave less time to perform the minimax and alphabeta pruning. The search depth is also very important for the performance. Refer to the result of ID\_improved vs AB\_improved. The former one performs better is only because it’s using iterative deepening and usually can reach depth of at least 7, in comparison with the latter’s depth of 5.