Heuristic Analysis

Table of Contents

1. Summary

All of my heuristic evaluation functions are defined in *game_agent_utils/heuristics.py*. I developed and tested five heuristic evaluation functions. The best performer was the `*who_can_get_there_first_depth_n*` function, discussed below, which let the Student player consistently outperform the ID_Improved player in runs of *tournament.py* by 5-20 percentage points. Here is a small, representative sample of Student's performance versus ID_Improved, using this heuristic: 80% to 65%, 80% to 73%, and 84% to 64%.

I am recommending `*who_can_get_there_first_depth_n*` as the best heuristic evaluation function for three reasons:

- As indicated in section 2, it resulted in Student's best performance compared to ID_Improved in several runs of the *tournament.py* script. It typically earned Student 15 percentage points more than ID_Improved, whereas the second best heuristic, `*who_can_get_there_first*`, typically earned Student only 8 percentage points more than ID_Improved.
- As indicated in section 4, it allowed Student to perform well in head-to-head matches against ID_Improved, where Student won 65% of the matches. When I performed similar head-to-head matches using the simpler `*who_can_get_there_first*` heuristic, Student won only 50% of the matches.
- Compared to other heuristics, `*who_can_get_there_first_depth_n*` allowed Student to perform consistently strongly against the simpler players (Random, MM_Null, and AB_Null) usually beating them in 85% to 100% of matches. Other heuristics performed well against the simpler players, but not nearly as strongly as consistently winning 85% or more of the matches.

2. Analysis of Heuristics and their Tournament Performance

`improved_score_depth_n`

- **Description**: This is like the improved-score heuristic that ID_Improved uses, except that it considers not just the immediate squares available to a player but also the squares that are one, two, ..., up to `*max_depth*` moves away from the player in a breadth-first search of all blank spaces, starting from the player's current position. Each square that figures into a player's score is weighted by the number of moves (i.e., search depth) it takes for the player to get there. Therefore, this scoring favors players that can potentially make many subsequent moves in the future.
- **Performance**: This allowed Student to perform only marginally better than ID_Improved (e.g., 76% to 74%).

`who_can_get_there_first`

- **Description**: This too is like the improved-score heuristic, except that the active player's (the player with the next move) open moves cannot count towards the inactive player's open moves. It's as if the active player can take those squares first and block the inactive player from moving to them. Therefore, if Student is the active player and `*who_can_get_there_first*` returns a score greater than zero, then Student has more open moves than the opponent and can possibly be blocking several squares from the opponent. My intuition is that this heuristic is useful deep in the game tree, where there are not many moves left for each player; that is, it gives a good indication of when a player is about to isolate its opponent.
- **Performance**: This allowed Student to perform considerably better than ID_Improved (e.g., 75% to 67%).

`who_can_get_there_first_depth_n`

- **Description**: Similar in spirit to `*improved_score_depth_n*` and `*who_can_get_there_first*`. If a player can get to a square first in the breadth-first search, then that square cannot count as an open square for the opponent. This is how we determine who can get to a square first: a breadth-first search is conducted, where the active player (the one who has the next move in the game) first gets to move to all squares that are one move away. Then those squares are not available to the inactive player, who then gets to move to all open squares that are one move away from its current position, making those squares unavailable to the active player. Then the active player gets to move to all open squares that are two moves away, making those unavailable to the inactive player. And so on. The idea of this heuristic is that it estimates not only how many squares a player can block from the opponent in the next move, but how many it can block in the next N moves. My intuition is that this heuristic can "see" further down the game tree than `*who_can_get_there_first*` and is able to give a good indication that a player may be able to isolate its opponent in a few or maybe even several moves.

- **Performance**: With the `*max_depth*` parameter set to 5, this allowed Student to perform substantially better than ID_Improved, usually by 15 percentage points. See the opening paragraph for example statistics. (I used this heuristic function, typically, with the `*max_depth*` argument somewhere between 4 and 8 and saw good performance among all these values.) It's worth noting that although this heuristic allowed Student to perform up to 20 percentage points better than ID_Improved, Student's performance against the AB_Open and AB_Improved players was only marginally better than ID_Improved's performance; much of Student's advantage over ID_Improved came from playing simpler players.

`**bfs_max_depth_heuristic**`

- **Description**: Performs a breadth-first search for each player from the player's current position over all blank squares on the board. Outputs the difference between the maximum search depth that `*player*` and `*player*`'s opponent can achieve.
- **Performance**: This caused Student to perform poorly compared to ID_Improved (e.g., 65% to 69%).

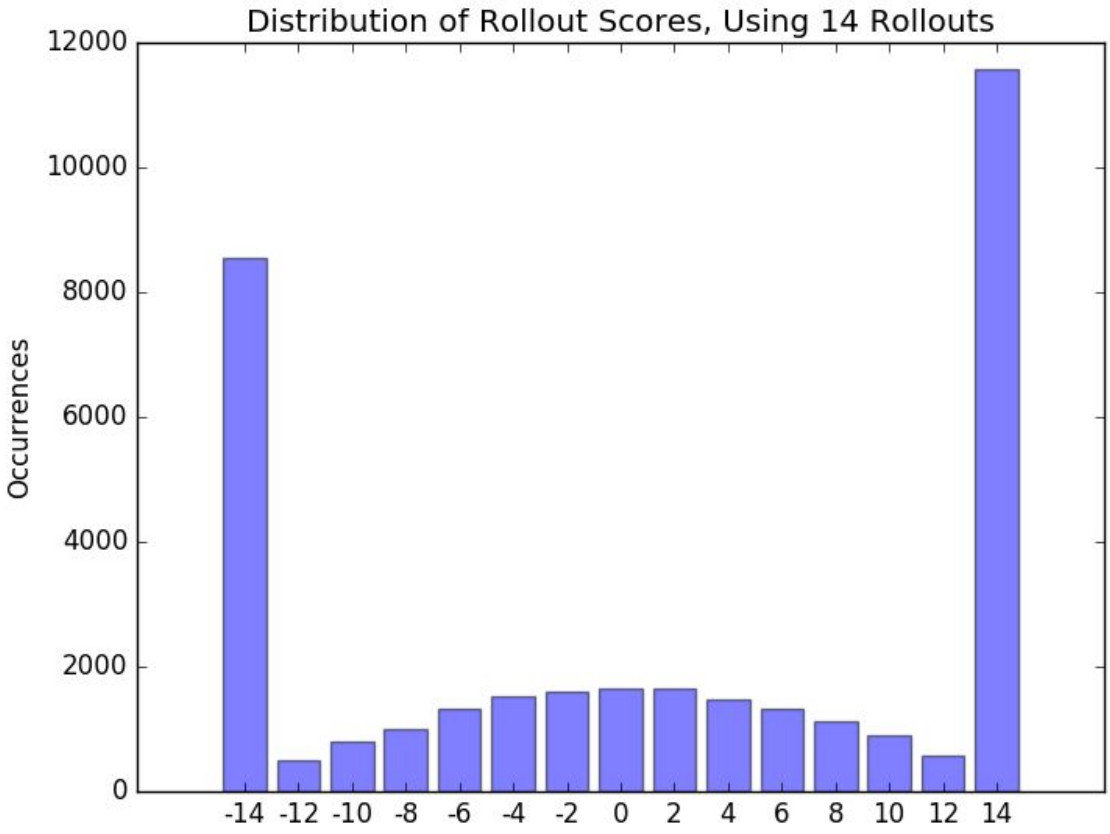
`**dfs_max_depth_heuristic**`

- **Description**: Like `*bfs_max_depth_heuristic*` but uses depth-first search instead of breadth-first search.
- **Performance**: This caused Student to perform poorly compared to ID_Improved (e.g., 63% to 70%).

3. Adding Pure Monte Carlo Rollouts

I was inspired to experiment with Monte Carlo rollouts after reading the AlphaGo paper, so I implemented pure Monte Carlo rollouts and gathered statistics on the distribution of rollout scores at various plies (i.e., game-tree depths). The statistics indicated a strong bimodal distribution toward all loses and all wins at ply 31 and beyond, indicating that once a game had reached ply 31, it was often the case that the winner was already determined and it was just a matter of time for the remaining moves to be played out.

The following figure illustrates the bimodal nature of game-tree nodes at ply 31 and deeper. I gathered statistics on the rollout scores of game nodes, using 14 rollouts at each node. If all rollouts from a node resulted in a win, then the node would get a rollout score of 14; if all loses, then a rollout score of -14; otherwise, the score would be somewhere in between. The figure shows that about half of the game-tree nodes result in a score of 14 or -14, indicating that the game winner is often determined by ply 31, even if more moves have to be made before formally reaching endgame.



Other statistics that I gathered showed that 65% of these Monte Carlo rollouts had to search only three or fewer plies before reaching endgame, showing that we're often near endgame when we conduct rollouts.

Therefore, I decided to add an optional parameter called `*use_rollouts*` to my custom player. If `*use_rollouts*` is true, then for board evaluations at ply 31 and deeper, my custom player would perform 14 simple Monte Carlo rollouts. Each rollout was worth 1 if my player won the rollout and -1 if it lost. The result of all 14 rollouts were summed, and this sum was then used in a weighted sum with the heuristic evaluation of the board, which allowed both the rollouts and the heuristic evaluation function to have a say in the value of a game state.

I ran the *tournament.py* script several times with both my Student player and my StudentRollouts player, which was configured the same as Student, except that `*use_rollouts*` was set to true. StudentRollouts often performed better than Student, usually by 5 percentage points.

4. Head-to-head: ID_Improved vs Student vs StudentRollouts

I was curious about how my game player would perform against ID_Improved in a head-to-head run of the *tournament.py* script, as well as how Student and StudentRollouts would perform head-to-head. I modified *tournament.py* so that Student, StudentRollouts, and ID_Improved would play each other twice:

- Surprisingly, Student beat StudentRollouts 11 to 9 both times they played.
- Student beat ID_Improved both times: 14 to 6 the first time and 12 to 8 the second.
- StudentRollouts beat ID_Improved both times. Surprisingly, the margin of victory was more narrow than when Student played ID_Improved. StudentRollouts won 11 to 9 both times it played ID_Improved.

Earlier results for StudentRollouts were promising, but these head-to-head results aren't quite as compelling. I will investigate possible weaknesses in StudentRollouts before deciding which player will represent me in the tournament for the February cohort.