heuristic_analysis

December 6, 2017

Lucas Sabbatini de Barros Fonseca
Artificial Intelligence Nanodegree

1 Overview

In this report I consider eight different heuristics for the AlphaBetaPlayer agent developed in game_agent.py file, combining them in a linear combination inside the functions custom_score(), custom_score_2() and custom_score3(). By varying the weights of these combinations, the effects of the heuristics on the overall score of the agent in multiple tournaments can be analysed, with the objective of finding a good combination of weights.

The heuristics themselves were chosen between a large number of evaluations found on online sources, github repositories and on the AIND lessons themselves, along with a few created or adapted by myself. The core focus of this study is to find good combinations of heuristics, instead of finding only one. Proper references are present inside function's documentations.

An example of the score functions that perform linear combination is given below. Documentations have been remove due to space economy.

```
In [1]: def custom_score(game, player):
            try:
                index_list = []
                weights = [0.2, 0.2, 0.0, 0.0, 0.0, 0.0, 0.0, 0.6]
                for i in range(len(weights)):
                    if weights[i] != 0.0:
                        index_list.append(i)
                # at the end, weights and index_list

                eval_vec = evaluate(game, player, index_list)
                value = 0
                for i in range(len(index_list)):
                    value += weights[index_list[i]]*eval_vec[i]

            except:
                print("Custom_scoreFuncitonError")
            return value
```

for efficiency purposes, instead of computing the dot product of weights and evaluation vector, the code runs a loop over the non-zero weights indexes.

2 Heuristic functions

Five heuristic (evaluation) functions have been chosen for the combination:

- actionMobility
- actionFocus
- my_moves_opp
- my_moves_2_opp
- distance_from_center
- reversed_distance_from_center
- squared_distance_from_center
- diff_my_moves_opp_moves_one_ply_lookahead

All these functions are handled through one function called evaluate, which uses the index_list created in custom_score to call functions and stores their values in a vector, returning it afterwards.

```
In [1]: def evaluate(game, player, index_list):
            try:
                eval_functions = [actionMobility,
                                  actionFocus,
                                  my_moves_opp,
                                  my_moves_2_opp,
                                  distance_from_center,
                                  reversed_distance_from_center,
                                  squared_distance_from_center,
                                  diff_my_moves_opp_moves_one_ply_lookahead]
                vec = []
                for idx in index_list:

                    vec.append(eval_functions[idx](game, player))
                    if player.time_left() < player.TIMER_THRESHOLD:
                        return [0.0]*len(index_list)
            except:
                print("EvaluateFunctionEror")

            return vec
```

3 Chosing weights

For choosing the weights, a series of tournaments were played with initial guessed values, and their values were updated in a trial and error approach. The tournament with which the heuristics were tested is not a good measure of their overall performance, since running the simulation multiple times with the same heuristics yields different performance values. This indicates that the performance of the agents is sensible to initial moves. 5th run shows this effect.

Computational and time requirements made it impossible to test a significant amount of combinations of heuristics. A more in depth discussion on this in the conclusion section.

The first four tournaments were used for studying the effect of each individual heuristic alone in the tournament, followed by testing all of them together and then a feel somewhat random

combinations. Around 60 tournaments have been played, but for space economy, a sample of ten trounaments is presented below:

- Figure 1

custom_score(), weights = [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
custom_score_2(), weights = [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
custom_score_3(), weights = [0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0]

- Figure 2

custom_score(), weights = [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0]
custom_score_2(), weights = [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0]
custom_score_3(), weights = [0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0]

- Figure 3

custom_score(), weights = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0]
custom_score_2(), weights = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0]
custom_score_3(), weights = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]

- Figure 4

custom_score(), weights = [1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
custom_score_2(), weights = [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0]
custom_score_3(), weights = [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0]

- Figure 5

custom_score(), weights = [0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0]
custom_score_2(), weights = [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.5]
custom_score_3(), weights = [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0]

- Figure 6

custom_score(), weights = [0.2, 0.0, 0.0, 0.2, 0.0, 0.0, 0.0, 0.6]
custom_score_2(), weights = [0.4, 0.2, 0.0, 0.2, 0.0, 0.0, 0.0, 0.6]
custom_score_3(), weights = [0.2, 0.0, 0.0, 0.2, 0.0, 0.0, 0.0, 0.6]

- Figure 7

custom_score(), weights = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0]
custom_score_2(), weights = [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0]
custom_score_3(), weights = [0.2, 0.2, 0.0, 0.0, 0.0, 0.0, 0.0, 0.6]

- Figure 8

custom_score(), weights = [1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0]
custom_score_2(), weights = [1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0]
custom_score_3(), weights = [1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]

- Figure 9

custom_score(), weights = [1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
custom_score_2(), weights = [1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0]
custom_score_3(), weights = [1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0]

- Figure 10

custom_score(), weights = [0.2, 0.2, 0.0, 0.0, 0.0, 0.0, 0.0, 0.6]
custom_score_2(), weights = [0.4, 0.2, 0.0, 0.0, 0.0, 0.0, 0.0, 0.6]
custom_score_3(), weights = [0.2, 0.0, 0.2, 0.0, 0.0, 0.0, 0.0, 0.6]

Control function improved_score performed with an win rate average of around 70%. For another evaluation to be considered better than it, the average should be higher than that. After around 50 tournaments were played with 100+ different sets of weights tested, three combinations were found to outperform the control function on average, but only by a couple of percentage points.

These combinations are presented below, and are set as final values of weight vectors for custom_score functions in the project.

custom_score(), weights = [0.4, 0.2, 0.0, 0.0, 0.0, 0.0, 0.0, 0.6]
custom_score_2(), weights = [0.2, 0.0, 0.0, 0.2, 0.0, 0.0, 0.0, 0.6]
custom_score_3(), weights = [1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]

Twelve data points were generated for each of these sets and the average of win percentage computed. They respectively yielded 73.21, 71.7 and 72.38 percent, while AB_improved had an average of 69.94% over 30 data points. A sample of results for these values are present in figures 11, 12 and 13, where each shows a tournament with repeated weights, correponding respectively to each set of weights.

It is worth noting here, as mentioned aboved and discussed in more depth later, that 50 tournaments for a linear combination of eight functions with cointinous possible values for the weights is too few compared to the state space of possible combinations, which means that the found sets are definitely not optimal.

A control test was also performed, turning only the weight corresponding to the equivalent improved_score function on, and the results are presented in figures 14 and 15.

4 Recommended evaluations

The most strong combination found involves three heuristics, them being actionMobility, actionFocus and diff_my_moves_opp_moves_one_ply_loockahead, and the set of weights below is recommended for the custom_score.

[0.4, 0.2, 0.0, 0.0, 0.0, 0.0, 0.0, 0.6]

actionMobility returns the number of possible moves the active agent has in a state, and has shown correlation to a good performance independently of the combination in question. As mentioned in lesson, this priority makes sense when evaluating nodes since for the game of isolation one loses when there's no available move left, so ranking these states higher forces the agent to look for open space.

diff_my_moves_opp_mvoes_one_ply_ahead has also shown to improve the agents performance independently of the combination of weights. This evaluation is similar to improved_score,

in the way that it returns the difference between player's and opponent's moves, but instead of calculating it over available moves from the current state being evaluated, it adds the number of available actions from all possible next positions for agent and opponent (sperately), and then return their difference.

The last heuristic, actionFocus, may seem counterintuitive since it includes the negative of actionMobility, but it actually helps in reducing the search space because it gives a measure of its narrowness, ranking states with fewer branches higher. Giving incentive to states with fewer moves is counterproductive in some ways (if used along with actionMobility), but since #my_moves is decreased from a fixed number, and the average of possible moves decreases down the game, its incentive to choose states with fewer possible moves increases with game depth, making it an adaptive strategy.

It is worth noting that this function reduces the search space throughout the game, not when selecting a single move, because the agent uses iterative deepening for the search. Considering that the time for selecting a move is constant in the whole game, narrower branches at the end of the game result in deeper search, making it possible to reach terminal states. Another interesting aspect of this function is that at the beginning of the game, it motivates the agent to select corners, since it returns larger values for fewer available moves. Its effect on the overall evaluation isn't strong though, after all, corners would yield smaller values for the other two heuristics.

Together, they create an agent that starts out giving high priority to branches with leaf nodes with most number of possible moves and that may lead to states with higher difference between agent's and opponent's moves. Further down the game its strategy slowly switches towards picking nodes with fewer available moves.

5 Discussion

As seen on tournaments, none of the chosen combinations has shown a significant improvement on performance against AB_improved, and it may be due to weak choice of weights and inefficiency on the overall architecture of the custom_score functions, since they require more computations than stand-alone evaluations. Also, TIME_THRESHOLD had to be set to 15.0 milliseconds.

Overall, actionMobility, my_moves_2_opp and diff_my_moves_opp_moves_one_ply_lookahead showed stronger, or at least more noticeble correlation to an increase in performance when activated, but with multiple evaluations and continuous possible values for the weights, the total number of possible combinations is simply too large to be analysed via brute force search, specially while varying the parameters by hand. So, while trying different weights and testing their effects may be useful for finding a good enough solution, the optimal solution for weights values won't be found this way.

Each evaluation function returns a different feature about the state currently being analysed, on the view of the current player. These features form a vector of features representing the state, and we want to find weights that map this vector to a good evaluation function. It's important to point out that the best strategy would involve adaptive weights, that instead of only combining individual features to evaluations, are sensible to the combination of features itself, to information about the game (depth in game), and opponent's strategy. To find adaptive weights, a simple linear combination would't be enough, and a function approximator capable of mapping more complex relations may need to be used.

Instead of guessing and testing values for the parameters, this agent could be coupled to a neural network model and a training algorithm, in a reinforcement learning process, so that a deeper function may be found.

List of Figures

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	9	1	10	0	10	0	9	1
2	MM_Open	8	2	8	2	8	2	6	4
3	MM_Center	10	0	7	3	8	2	8	2
4	MM_Improved	7	3	8	2	6	4	8	2
5	AB_Open	6	4	6	4	5	5	7	3
6	AB_Center	4	6	6	4	6	4	5	5
7	AB_Improved	7	3	6	4	5	5	4	6
	Win Rate:	72.9%		72.9%		68.6%		67.1%	

Figure 1

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	9	1	8	2	9	1	10	0
2	MM_Open	8	2	9	1	8	2	7	3
3	MM_Center	9	1	9	1	9	1	9	1
4	MM_Improved	7	3	8	2	6	4	7	3
5	AB_Open	6	4	5	5	6	4	3	7
6	AB_Center	3	7	6	4	4	6	6	4
7	AB_Improved	6	4	4	6	5	5	3	7
	Win Rate:	68.6%		70.0%		67.1%		64.3%	

Figure 2

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	10	0	10	0	10	0	9	1
2	MM_Open	9	1	8	2	6	4	6	4
3	MM_Center	7	3	10	0	9	1	7	3
4	MM_Improved	9	1	7	3	9	1	5	5
5	AB_Open	6	4	4	6	6	4	4	6
6	AB_Center	6	4	5	5	6	4	6	4
7	AB_Improved	5	5	5	5	4	6	5	5
	Win Rate:	74.3%		70.0%		71.4%		60.0%	

Figure 3

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	10	0	10	0	10	0	10	0
2	MM_Open	8	2	10	0	8	2	8	2
3	MM_Center	9	1	9	1	7	3	10	0
4	MM_Improved	8	2	7	3	6	4	6	4
5	AB_Open	6	4	6	4	4	6	6	4
6	AB_Center	5	5	7	3	5	5	6	4
7	AB_Improved	6	4	6	4	5	5	8	2
	Win Rate:	74.3%		78.6%		64.3%		77.1%	

Figure 4

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	9	1	10	0	10	0	10	0
2	MM_Open	9	1	10	0	9	1	7	3
3	MM_Center	9	1	8	2	8	2	8	2
4	MM_Improved	6	4	8	2	7	3	6	4
5	AB_Open	7	3	3	7	5	5	3	7
6	AB_Center	6	4	8	2	5	5	5	5
7	AB_Improved	6	4	4	6	6	4	8	2
	Win Rate:	74.3%		72.9%		71.4%		67.1%	

Figure 5

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	8	2	8	2	10	0	10	0
2	MM_Open	8	2	7	3	6	4	8	2
3	MM_Center	8	2	9	1	8	2	9	1
4	MM_Improved	8	2	6	4	6	4	8	2
5	AB_Open	7	3	4	6	7	3	5	5
6	AB_Center	7	3	7	3	7	3	4	6
7	AB_Improved	3	7	5	5	4	6	6	4
	Win Rate:	70.0%		65.7%		68.6%		71.4%	

There were 3.0 timeouts during the tournament -- make sure your agent handles searc
h timeout correctly, and consider increasing the timeout margin for your agent.

Figure 6

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	9	1	9	1	10	0	9	1
2	MM_Open	8	2	8	2	6	4	10	0
3	MM_Center	9	1	10	0	9	1	9	1
4	MM_Improved	8	2	7	3	10	0	7	3
5	AB_Open	6	4	6	4	7	3	5	5
6	AB_Center	5	5	6	4	5	5	5	5
7	AB_Improved	6	4	6	4	5	5	8	2
	Win Rate:	72.9%		74.3%		74.3%		75.7%	

Figure 7

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	10	0	10	0	9	1	10	0
2	MM_Open	9	1	8	2	8	2	7	3
3	MM_Center	8	2	10	0	10	0	9	1
4	MM_Improved	7	3	8	2	9	1	7	3
5	AB_Open	6	4	5	5	5	5	5	5
6	AB_Center	6	4	6	4	7	3	8	2
7	AB_Improved	6	4	7	3	4	6	4	6
	Win Rate:	74.3%		77.1%		74.3%		71.4%	

Figure 8

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	10	0	9	1	10	0	9	1
2	MM_Open	9	1	9	1	7	3	9	1
3	MM_Center	9	1	8	2	9	1	9	1
4	MM_Improved	6	4	6	4	6	4	6	4
5	AB_Open	3	7	5	5	7	3	5	5
6	AB_Center	6	4	2	8	3	7	6	4
7	AB_Improved	5	5	5	5	5	5	7	3
	Win Rate:	68.6%		62.9%		67.1%		72.9%	

Figure 9

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	10	0	10	0	9	1	9	1
2	MM_Open	6	4	9	1	9	1	7	3
3	MM_Center	9	1	9	1	9	1	9	1
4	MM_Improved	8	2	7	3	9	1	8	2
5	AB_Open	6	4	4	6	6	4	6	4
6	AB_Center	6	4	5	5	7	3	6	4
7	AB_Improved	4	6	4	6	7	3	5	5
	Win Rate:	70.0%		68.6%		80.0%		71.4%	

Figure 10

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	10	0	9	1	8	2	10	0
2	MM_Open	7	3	7	3	10	0	9	1
3	MM_Center	9	1	9	1	10	0	8	2
4	MM_Improved	5	5	8	2	7	3	8	2
5	AB_Open	4	6	7	3	5	5	6	4
6	AB_Center	5	5	7	3	3	7	6	4
7	AB_Improved	6	4	5	5	5	5	4	6
	Win Rate:	65.7%		74.3%		68.6%		72.9%	

Figure 11

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	9	1	10	0	10	0	8	2
2	MM_Open	8	2	7	3	8	2	8	2
3	MM_Center	10	0	9	1	6	4	9	1
4	MM_Improved	8	2	9	1	10	0	7	3
5	AB_Open	3	7	5	5	5	5	6	4
6	AB_Center	5	5	5	5	7	3	5	5
7	AB_Improved	5	5	5	5	6	4	3	7
	Win Rate:	68.6%		71.4%		74.3%		65.7%	

Figure 12

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	9	1	7	3	9	1	10	0
2	MM_Open	6	4	8	2	9	1	7	3
3	MM_Center	9	1	9	1	8	2	9	1
4	MM_Improved	8	2	7	3	8	2	8	2
5	AB_Open	7	3	5	5	6	4	4	6
6	AB_Center	6	4	6	4	6	4	5	5
7	AB_Improved	4	6	5	5	5	5	7	3
	Win Rate:	70.0%		67.1%		72.9%		71.4%	

Figure 13

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	10	0	10	0	10	0	9	1
2	MM_Open	8	2	4	6	9	1	6	4
3	MM_Center	8	2	9	1	8	2	9	1
4	MM_Improved	4	6	7	3	8	2	5	5
5	AB_Open	7	3	5	5	6	4	4	6
6	AB_Center	4	6	7	3	5	5	6	4
7	AB_Improved	5	5	6	4	4	6	6	4
	Win Rate:	65.7%		68.6%		71.4%		64.3%	

Figure 14

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	9	1	8	2	9	1	9	1
2	MM_Open	8	2	10	0	8	2	6	4
3	MM_Center	9	1	9	1	10	0	9	1
4	MM_Improved	7	3	7	3	6	4	8	2
5	AB_Open	6	4	4	6	4	6	7	3
6	AB_Center	6	4	7	3	5	5	7	3
7	AB_Improved	2	8	4	6	4	6	5	5
	Win Rate:	67.1%		70.0%		65.7%		72.9%	

Figure 15