N. Veiga (nohe.veiga@gmail.com)
Artificial Intelligence Nanodegree
Aug 2017

Heuristics analysis report

Approach

For this heuristics evaluation three different options have been chosen following two different approaches:

_Custom : based on Manhattan distance between agent and opponent, this function leverages the number of legal moves for the agent when the moves are far from the opponent.

_Custom_2 : based on the suggestions made from Thad on the lectures, this function is just a simple modification applying an arbitrary coefficient to the opponent's movement to make the chase more 'aggressive' by the agent. Due to the simple nature of this functions it was worth to give a try.

_Custom_3 : based on Manhattan distance between agent and opponent, the distance value is used as a bonus to reduce the opponent's moves (as _Custom_2) but penalizing more the opponent's moves when the agent plays long distances from the opponent. The inverse of the function has been used instead, without any significant results.

Based on improved_score() and center_score(), other modifications have been tried without significant success:

- Arbitrary pair/odd penalizing
- Change strategy depending on the remaining % blank_spaces/occupancy

Measurement of performance

Since most packages like logging, pdb or profile are not allowed, the performance measurement has been carried out by Terminal[1].
Number of matches played: 30 in two rounds.

Observations

Some important observations have been noted as a result of running several times the tournament file. One important factor that has been noted as an important key to improve the performance of the agent, is how the best_move variable is initialized in get_move() functions and minimax() and alphabeta() within their respective classes. Different ways to init have been used:

---

[1] python -m cProfile -s cumulative tournament.py

N. Veiga (nohe.veiga@gmail.com)
Artificial Intelligence Nanodegree

best_move = (-1, -1) → not a valid move
best_move = random.choice(legal_moves) → random choice
best_move = legal_moves[random.randint(0, len(legal_moves) - 1)] → random choice
best_move = legal_moves[0]

Easily the winning percentages in general may increase in 5-8% when best_move is initialized to a fixed position rather than random or not valid, there is an excellent discussion on Udacity's AIND-Term1 talking about this subject[2].

Another important factor noted as important is working with heuristic functions as simple as possible. Including loops increase significantly the amount of time for the evaluation and thus, the strategy may experience variations if the the function is more lightweight, thus these evaluations based on % blank spaces or pairs/odd, have been removed. In the next figure shows an example of the functions tested and finally discarded as the winning rates were below 60%.

```
def custom_score(game, player):
    """Calculate the heuristic value of a game state from the point of view
    of the given player.

    This should be the best heuristic function for your project submission.

    Note: this function should be called from within a Player instance as
    `self.score()` -- you should not need to call this function directly.

    Parameters
    ----------
    game : `isolation.Board`
        An instance of `isolation.Board` encoding the current state of the
        game (e.g., player locations and blocked cells).

    player : object
        A player instance in the current game (i.e., an object corresponding to
        one of the player objects `game.__player_1__` or `game.__player_2__`.)

    Returns
    -------
    float
        The heuristic value of the current game state to the specified player.
    """
    # TODO: finish this function!

    bonus = 0
    n_agent_moves   = len(game.get_legal_moves(player))
    n_opp_moves     = len(game.get_legal_moves(game.get_opponent(player)))
    n_blank_spaces  = len(game.get_blank_spaces())
    pcent_blank_spaces = int(n_blank_spaces / (game.width * game.height) * 100)


    player_loc  = game.get_player_location(player)
    opp_loc     = game.get_player_location(game.get_opponent(player))

    if pcent_blank_spaces <=75 :

        if manhattan_distance(player_loc, opp_loc) <= 3 :
            bonus = 12.0
        else:
            bonus = 2.0
    else:
            bonus = 1.0


    return float(n_agent_moves - bonus * n_opp_moves)

```

Figure 1. Example of a custom function discarded.

[2] https://discussions.udacity.com/t/sources-of-variation-in-tournaments-and-why-forfeits/268387

Data

Inasmuch as briefness is a valuable quality of the present exercise, for simplicity most of these experimental data obtained from different heuristics analysis in order to find the proposed custom functions, have been not included in the present document, but can be provided.

% Winning rates by Game Agent[3]

	AB_Improved	AB_Custom	AB_Custom_2	AB_Custom_3	timeouts
15/08 00:56	69.8	65.7	68.6	69.3	
15/08 09:37	66.9	70	72.9	67.4	466
15/08 11:11	66.2	65	69.8	66.2	202
Average	**67.63**	**66.90**	**70.43**	**67.63**	
Std Dev	**1.91**	**2.71**	**2.22**	**1.56**	

Figure 2. Winning % for each heuristic function.

Accumulated cpu time spent[4]

	AB_Improved cumtime	AB_Custom cumtime	AB_Custom_2 cumtime	AB_Custom_3 cumtime	Number of calls	Global time
15/08 00:56	504.800	359.181	307.393	374.046	3483039161	4816.040
15/08 09:37	363.954	291.653	251.425	299.598	3055927812	3417.628
15/08 11:11	499.093	339.061	304.941	360.232	3221515220	4661.095

Figure 3. Accumulated cpu time in secs spent by the function.

% time cpu spent for each function

	AB_Improved cumtime	AB_Custom cumtime	AB_Custom_2 cumtime	AB_Custom_3 cumtime
15/08 00:56	10.48164052	7.458015299	6.382692004	7.766671373
15/08 09:37	10.64931584	8.53378425	7.356710561	8.766255426
15/08 11:11	10.70763415	7.274277825	6.542260992	7.728484401
Average (% time)	**10.6128635**	**7.755359125**	**6.760554519**	**8.087137067**

Figure 4. % Time for each function vs the global time.

---

[3] Using tournament.py file provided with the files needed to this project
[4] Using Python cProfile - https://docs.python.org/3.6/library/profile.html

N. Veiga (nohe.veiga@gmail.com)
Artificial Intelligence Nanodegree
Aug 2017

Conclusions

Several tests have been run out, and an extremely high variability over the winning rates have been observed, so it is not worth getting into fine analytics. However in a roughly way, we can say that the performance of the custom functions is good enough for all 3 and the winning rates are over the 60%, and the performance versus the Improved function is better as data are fairly consistent. In the same way, the rates are much better for the Minimax agent rather than the AlphaBetaPlayer using iterative deepening alpha-beta search with heuristics (see the figure below).

Due to these variability is really difficult to draw a conclusion and pick one over the rest. However, based on the winning rates and the % time cpu spent, AB_Custom_2 and AB_Custom_3, seem to be the better choices, as the winning rates are high and the cost in time is lower compared to AB_Improved or AB_Custom.

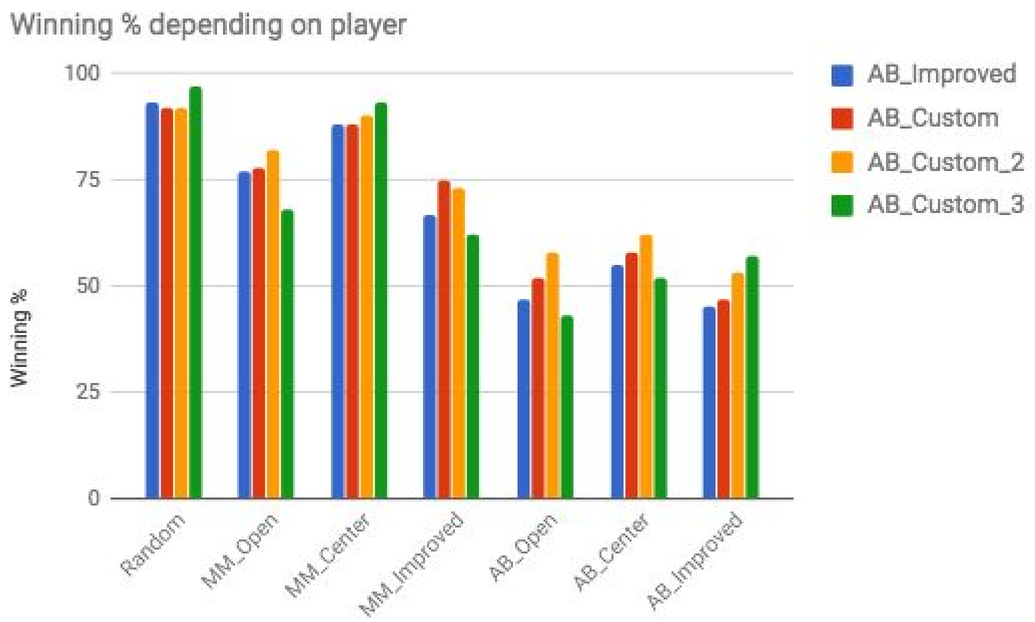

Figure 6. % Winning scores for each player using different heuristics.