Heuristics Analysis: AIND_Isolation

John Quinn, Udacity Artificial Intelligence Nano-Degree Candidate

N.B.: Throughout this report I have used (i) terms from the assignment materials and (ii) function names and data items included within my submitted code, game_agent.py. The former are well explained in the assignment materials, and the latter are sufficiently commented within my submitted code; detailed definitions of either within this report would be repetitive. I have, nonetheless, reiterated some such items and expounded upon others here for the sake of clarity.

Summary:

Within game_agent.py I have entered three custom heuristics functions, custom_score(), custom-score_2(), and custom_score_3(). The best heuristic function is, as required, custom_score(), and it is the one I suggest using.

The overall performance of custom_score() was 57.1%, higher than those of the other two functions. These three data can be found in the Win Rate line/row in the Playing Matches table generated by tournament.py and reproduced on the second page of this report. In that table, custom_score()'s results fall under the AB_Custom columns while those of the other two functions are listed in the columns to the right of those for custom_score().

custom_score(), also regularly beat what is likely the worthiest adversary, AB_Improved. In the ten games against this opponent, custom_score() won seven of those contests. custom-score_2() tied AB-Improved with five games apiece while custom_score_3() also won seven of these contests.

Against three adversaries, custom_score() posted narrow losses, losing six of ten contests. custom_score_2() and custom_score_3() performed comparably overall but with significant loss margins against certain opponents. In some matches the two lost seven or eight of the games. custom-score() proved more reliable against a variety of opponents than the other two.

I have, in addition, built custom_score() in a manner that allows for easy improvement. In particular, one can optimize the weight vector – see the next two pages of this report – with more sophisticated methods than trial-and-error and add or delete features.

Finally, custom-score() benefits from the experience gained building the other two functions, particularly custom_score_2(). Speed of execution is a key element to success for a heuristic function, and custom_score() is built for speed. custom_score_2() provides many data but suffers from phlegmatism and custom_score_3() is a bit simplistic.

Armed with superior technology vis-à-vis an opponent (e.g., player with a supercomputer and opponent with a standard-fare laptop), custom_score_2() would be a viable choice, but it is not the best choice when run on the same or comparable systems, the case in this assignment. I have left custom-score_2()'s six timeouts as a reminder of the foregone information that would otherwise be enjoyed via iterative deepening.

Playing Matches table:

The output of tournament.py is as follows:

```
                        *************************
                             Playing Matches
                        *************************

 Match #   Opponent    AB_Improved   AB_Custom   AB_Custom_2  AB_Custom_3
                        Won | Lost   Won | Lost   Won | Lost   Won | Lost
    1       Random       8  |   2    10  |   0     8  |   2     9  |   1
    2       MM_Open      6  |   4     4  |   6     6  |   4     2  |   8
    3      MM_Center     5  |   5     6  |   4     5  |   5     8  |   2
    4     MM_Improved    6  |   4     5  |   5     3  |   7     2  |   8
    5       AB_Open      4  |   6     4  |   6     4  |   6     5  |   5
    6      AB_Center     6  |   4     4  |   6     6  |   4     5  |   5
    7     AB_Improved    3  |   7     7  |   3     5  |   5     7  |   3
--------------------------------------------------------------------------
           Win Rate:      54.3%        57.1%        52.9%        54.3%
```

My Heuristic Functions – Descriptions and Analyses:

Explaining the three functions is best done in the following order: custom_score_3(), custom_score_2(), and then custom_score():   This follows the evolution of my approach to the assignment:

custom_score_3():

This was my first, naïve attempt at a heuristic function, which simply employs two features, move_count_difference and weighted_distance_from_center. The former is the difference between the number of the player's and the opponent's legal_moves at the outset. The latter is itself a heuristic measure.

This second feature is a judgmental approach to measuring the given player's distance from the center box. It starts with the base measure of vector magnitude but weights the longer differential by two since one part of a knight move must be two boxes long. This entire numerator is then divided by 3, the sum of the weights.

Finally, the two features are themselves weighted.   The first is accorded an order of magnitude higher than the second. If move_count_difference is not zero it is returned. Otherwise, the ultimate score returned is the dot product of the vector encapsulating the two component features and their weights.

This approach, particularly the final incarnation of the second feature and of the weight vector, was mostly trial-and-error. Along the way, I saw that simple formulae derived from methods discussed in lectures would likely lead to limited success and attempted a more sophisticated approach with custom_score_2().

custom_score_2():

This second heuristics function is quite a bit more involved than the first. Given the importance of speed/computational efficiency, especially when exhaustive search is impossible, I built a helper function, get_player_feature_values. This helper function returns the values of all features utilized in this heuristics function.

In that same vein, I continued, *inter alia*, to eschew numpy, utilizing faster, native Python for dot products for example. I did not even attempt matrix multiplication, as with a matrix with sufficient column space and/or row space the computational cost would be prohibitive, and the treasure trove derived from iterative deepening would be sullied.

This helper function, get_player_feature_values(), returns six integer features. These features are:

- legal_move_count: the number of coordinate tuples in p_legal_moves. This list, p_legal_moves, is merely the output of provided isolation.py's method, get_legal_moves.
- start_center_distance: the player's distance from the center box, measured from the player's initial coordinates, p_coordinates (before any p_legal_moves), via get_distances_from_center(). For extensibility and to limit redundant code, get_distances_from_center(), is another helper function I developed for intra-class calculation of what is called method center_score() in provided sample_player.py.
- total_next_moves: the sum of the number of legal_moves that the player can take after any original p_legal_moves (i.e., the number of next_moves).
- max_path_length: the length, measured in steps, of the longest path that the player can take from p_coordinates (i.e., the player's current Board coordinates) through any p_legal_move.
- path_count: that number of paths the player can take from p_coordinates through each p_legal_move
- min_center_dist: for any path in path_count the closest any node/box in that path is to the Board's center box measured by get_distances_from_center().

This second heuristic function obtains each of these six data for each of the two players from get_player_feature_values(). The return value improved_score() from provided sample_players.py is, of course, ***the*** measure; ultimately when that value is zero for a player that player loses. As such, custom_score_2() returns that value (i.e., the difference between the player's and the opponent's legal_move_counts) when it is not zero.

When that value is zero, however, the information gain is little; all one knows is that he or she has not won or lost. This is the equivalent of reporting a tie.

In this case, custom_player_2() first adds the data for the player and the data for the opponent. For the player, this sum includes the opponent's min_center_dist and vice-versa. For each adversary, each other datum is deemed "better" if larger, but the opposite is true for this one datum for such adversary. For that reason, this switching of sum inclusions was required.

This second heuristic function performs reasonably well, albeit with mixed results for certain opponents. Overall, custom_score_2() was inferior to custom_score(), 52.9% vis-à-vis 57.1%, and only tied AB_Improved five games apiece.

custom_score():

This final and best heuristic function reflects the knowledge that I gained in building the other two. In particular, it reflects speed of execution as essential and incorporates some of framework feature weights from the other two.

In deriving these weights, I started with visceral impressions of orders of magnitude. Clearly, legal_move_count was going to be most important, but when I assigned it a full order of magnitude above all other features the results were poor. After several trials with custom_score_2(), I noted that having one more move than one's opponent - legal_move_count – was more important than the number of legal second moves - total_next_moves – but only about one-half more important.

Distance from the center square at the outset - start_center_distance – counted but was roughly a full order of magnitude less important than total_next_moves. Such datum, of course, is even less relatively important than the most important, legal_move_count.

The other data seemed to encode some predictive value of the game result – winner or loser – but only marginally in comparison to the above three. For now, custom_score() includes them, but each with two orders of magnitude less than that of legal_move_count.

Ultimately, custom_score() returns improved (i.e., the difference between legal_move_count of the adversaries at the outset) if improved is not zero. Otherwise, custom_score() returns the dot product of the vector encapsulating the feature differences and their weight vector. This recognizes that improved is paramount but betters the results of improved by providing a useful coefficient, not simply a zero (i.e., a tie.)

Summary and Self-Critique:

My best heuristic function, custom_score(), performs well, but the utility of the other two is debatable. The entire approach to developing these heuristic functions is born of presumed domain knowledge, such as with isolation or chess, and trial-and-error. This has worked at least serviceably for this exercise, but one might approach the exercise, e.g., by coupling the selected features with machine learning. An appropriate classifier for an indicator variable – zero for a loss and one for a win – run over many simulations likely would have resulted in a more elegant and effective solution. This is likely the *de facto* go-to route for the weight vector elements.