Heuristic Scoring Function Analysis

James Mallett

Udacity AIND: Term 1 Project 2

Three custom scoring functions were implemented in this project in order to try and maximize the amount of times that the developed Alpha Beta game playing agent can win the game of Isolation against several different opponents; the results of these games are given in the following table.

Table 1: Heuristic Scoring Functions Performance

Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
	Won	Lost	Won	Lost	Won	Lost	Won	Lost
Random	9	1	10	0	8	2	10	0
MM_Open	8	2	5	5	8	2	8	2
MM_Center	8	2	7	3	8	2	8	2
MM_Improved	6	4	4	6	6	4	6	4
AB_Open	6	4	6	4	5	5	4	6
AB_Center	5	5	6	4	4	6	6	4
AB_Improved	4	6	6	4	3	7	5	5
Win Percentage:	65.70%		62.90%		60.00%		67.10%	

AB_Custom_1():

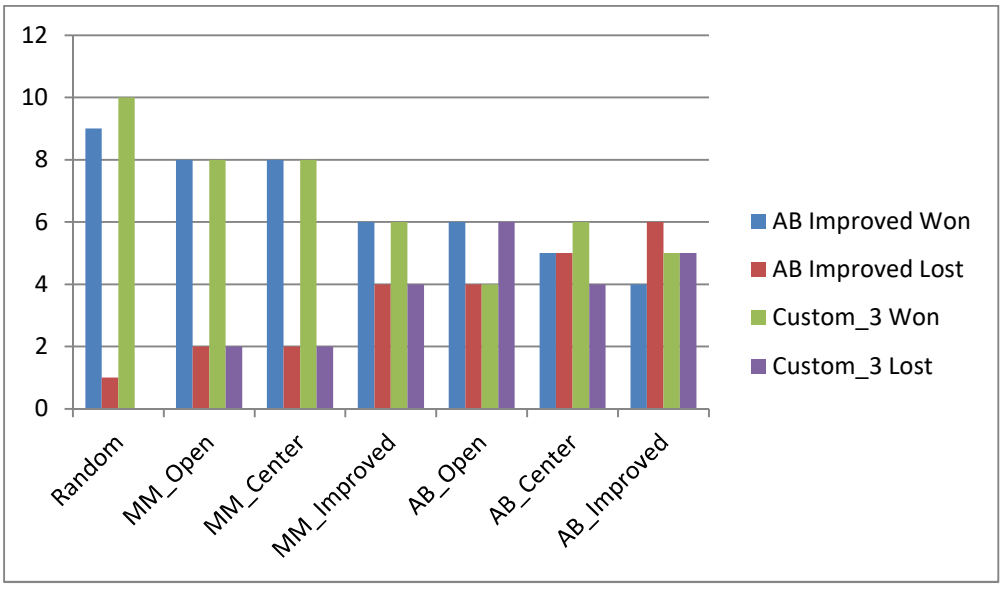
This function was centred on the principle of minimizing the number of moves the opponent has relative to the number of moves available to the user. It is a relatively simple function that is able to Win with a reasonable consistency, but has room for more complexity to be added which would allow for a further increase the win percentage.

AB_Custom_2():

This function was an experimental one, which produced a result that was not as good as the AB_Improved() baseline, but is still able to achieve a win percentage of 60% when playing against all of the various opponents. This function returns a score that is dependent on both the number of open spaces within a 2 block radius of the player, and the difference in moves as in AB_Custom_1() which is used as a weighting factor. Ultimately this function may be too computationally intensive to be feasible in a larger game, but is still able to produce good results on this scale.

AB_Custom_3():

This scoring function was able to achieve a win percentage of 67.1% against the various opponents and was the most successful of the three heuristic scoring functions. This function uses the distance of the player from the centre of the board as a component of the score and the difference between number of moves for the player and the opponent as a weighting factor. (1) It is therefore best to use this function because it was ultimately able to achieve the best win/loss ratio. (2) This scoring function also aims to keep the player as close to the centre of the board as possible, where it is less likely to get stuck in a corner, and uses the difference in moves of the player and opponent to ensure that having more moves than the opponent is also considered. (3) In terms of computational power, this function does not require large amounts, it does not contain any large loops or iterations and uses fairly straightforward arithmetic along with existing functions to generate a score for the proposed move.



When running a basic unit-test to assess the performance and processing times of the different scoring functions, which is shown at the end of this report, the following times to complete a game against a random player were observed:

Custom_score:	1.264s	Player Win
	1.547s	Player Win
	2.674s	Player Win
Custom_score_2:	2.384s	Player Win
	1.763s	Player Win
	2.665s	Player Win
Custom_score_3:	2.107s	Player Win
	2.110s	Player Win
	1.968s	Player Win

(4) So although Custom_score_3 took slightly longer on average compared to Custom_score, it uses a slightly more complex scoring function which could justify the additional processing time. It is also still shorter than Custom_score_2 which uses loops that could delay the game substantially.

All of these scoring functions could still be further improved by implementing symmetry in early game moves and having several moves for the start recommended more highly than other possible moves.

Tournament.py Result:

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

 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      8   |   2     10   |   0
    2        MM_Open      8   |   2       5   |   5      8   |   2      8   |   2
    3       MM_Center     8   |   2       7   |   3      8   |   2      8   |   2
    4     MM_Improved     6   |   4       4   |   6      6   |   4      6   |   4
    5        AB_Open      6   |   4       6   |   4      5   |   5      4   |   6
    6       AB_Center     5   |   5       6   |   4      4   |   6      6   |   4
    7     AB_Improved     4   |   6       6   |   4      3   |   7      5   |   5
----------------------------------------------------------------------------------
            Win Rate:      65.7%        62.9%        60.0%        67.1%

Your agents forfeited 243.0 games while there were still legal moves available to play.
```

UnitTest:

```python
class IsolationTest(unittest.TestCase):
    """Unit tests for isolation agents"""

    def setUp(self):
        reload(game_agent)
        # self.player1 = "Player1"
        # self.player2 = "Player2"
        self.player1 = game_agent.AlphaBetaPlayer()
        self.player2 = sample_players.RandomPlayer()
        self.game = isolation.Board(self.player1, self.player2)

    def test_example(self):
        # TODO: All methods must start with "test_"

        #Player 1 is active to start so he plays this move

        minmaxplayer=game_agent.AlphaBetaPlayer()
        RandomPlayer=sample_players.RandomPlayer()

        self.game.apply_move((2, 3))
        self.game.apply_move((0, 5))

        assert(self.player1 == self.game.active_player)

        winner, history, outcome = self.game.play()
        print("\nWinner: {}\nOutcome: {}".format(winner, outcome))
        print(self.game.to_string())
```