Heuristic Analysis Rebuttal

Jason McDonald

This analysis contains an extensive overhaul of my original work. My original analysis is posted here:

https://goo.gl/5Gdrvv

This submission will be my final submission.

My previous analysis was reject on the grounds that it did not meet the following requirement:

> **"The report makes a recommendation about which evaluation function should be used and justifies the recommendation with at least three reasons supported by the data."**

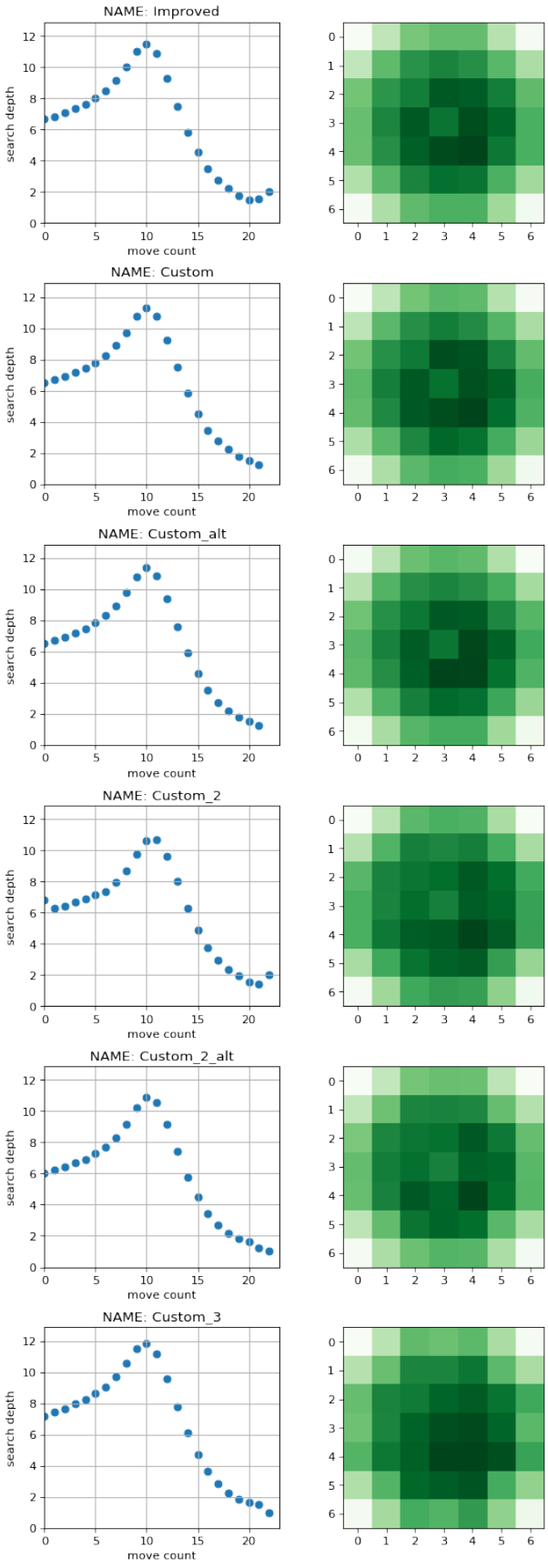
As I have previously stated, the data shows no such recommendation can be made. This specific example, isolation with a chess knight on a 7x7 board, lies outside the regime in which comparative heuristic analysis can be demonstrated with alpha-beta search and iterative deepening. The following data demonstrates this conclusion by showing win rates that are all within a percentage point. Any attempt to recommend a heuristic that does not involve a win rate for a game based on winning would be academically dishonest.

Match #	Opponent	Improved		Custom		Custom_alt		Custom_2		Custom_2_alt		Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	1964	36	1957	43	1937	63	1944	56	1961	39	1947	53
2	MM_Open	1623	377	1592	408	1598	402	1588	412	1600	400	1624	376
3	MM_Center	1894	106	1892	108	1890	110	1879	121	1885	115	1877	123
4	MM_Improved	1495	505	1531	469	1526	474	1474	526	1517	483	1501	499
5	AB_Open	1005	995	989	1011	988	1012	956	1044	992	1008	1006	994
6	AB_Center	1035	965	1029	971	1083	917	1014	986	1003	997	1064	936
7	AB_Improved	983	1017	1030	970	1030	970	940	1060	986	1014	970	1030
	Win Rate:	71.4%±0.8%		71.6%±0.8%		71.8%±0.8%		70.0%±0.8%		71.0%±0.8%		71.4%±0.8%	

Using a variety of custom score functions, I ran the tournament for this assignment. All of the score functions performed similarly. For this analysis I have included two score functions of some complexity ('Custom', 'Custom_2'), two score functions recommended by a Udactiy reviewer ('Custom_alt','Custom_2_alt') and one score function of low complexity ('Custom_3').

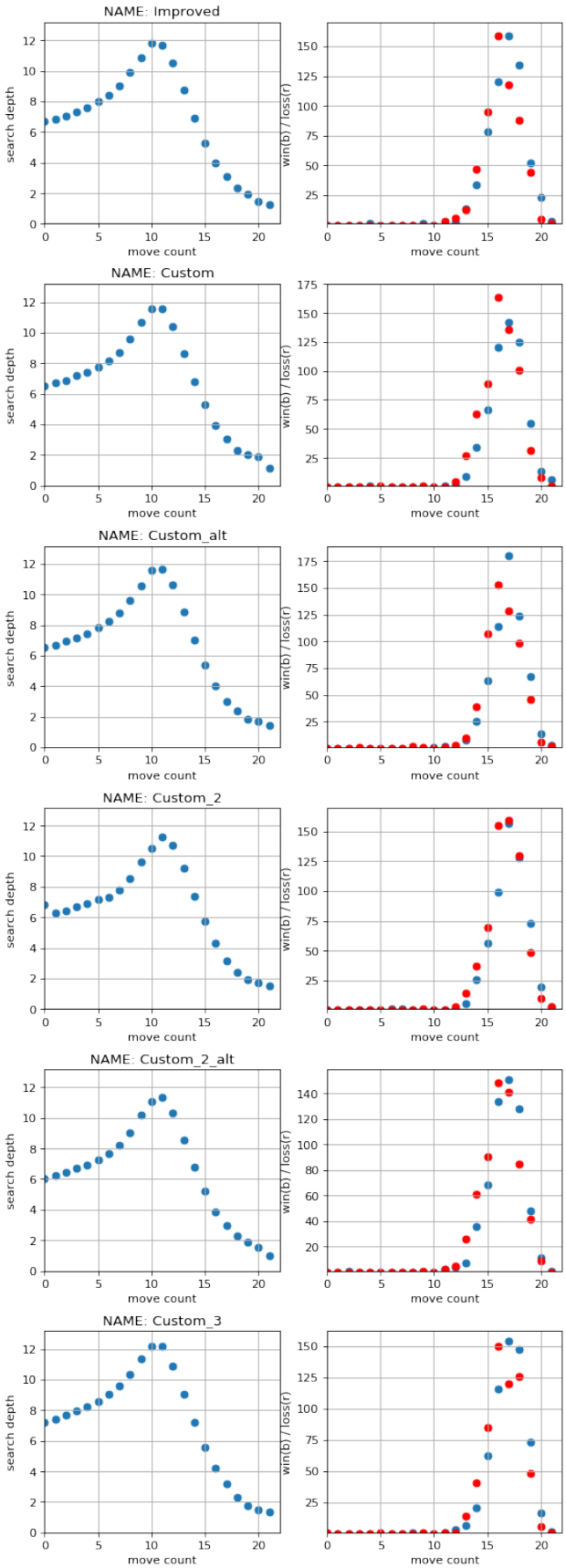
In the following pages I will show how heuristic functions fail to differentiate in this test and later I will show that heuristic analysis can be performed if we limit our test agents to minimax.

These plots show search depth as a function of move count for each score function and position occupation (green) averaged over all 14,000 games in the tournament. The search depth drop-off at move 10 for all score functions shows that alpha-beta is now searching to endgame for every move after move 10 and all score function return ±Infinity for win-loss scores. Thus, after move 10 all score functions perform EXACTLY the same and return EXACTLY the same values. This means two things. One, no heuristic scores are used after midgame. Two, any advantage conferred by a heuristic must be in the setup of the board state before endgame.



Match #	Opponent	Improved		Custom		Custom_alt		Custom_2		Custom_2_alt		Custom_3	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	AB_Open	209	191	211	189	194	206	190	210	195	205	196	204
2	AB_Center	209	191	187	213	204	196	193	207	204	196	208	192
3	AB_Improved	203	197	176	224	204	196	188	212	192	208	200	200
	Win Rate:	51.7%±2.9%		47.8%±2.9%		50.2%±2.9%		47.6%±2.9%		49.2%±2.9%		50.3%±2.9%	

We can further demonstrate the indifference to heuristic choice by running a tournament in which all agents use alpha-beta search and iterative deepening. The win rates then become about 50/50 showing little preference for a particular heuristic.



Finally, if we wish to differentiate among our heuristics we must place our agents into a scenario where they are compute limited or call limited for each move. In this case I've done a simple call limited scenario in which all agents are limited to minimax with a depth of three.

```
Match #   Opponent      Improved      Custom      Custom_alt    Custom_2    Custom_2_alt   Custom_3
                        Won  | Lost  Won  | Lost  Won  | Lost  Won  | Lost  Won  | Lost  Won  | Lost
    1       MM_Open     995  | 1005 1000  | 1000 1065  |  935 1161  |  839 1168  |  832  974  | 1026
    2      MM_Center   1672  |  328 1715  |  285 1720  |  280 1663  |  337 1688  |  312 1653  |  347
    3     MM_Improved 1001  |  999  977  | 1023 1005  |  995 1178  |  822 1056  |  944  947  | 1053
-------------------------------------------------------------------------------
           Win Rate:   61.1%±1.3%   61.5%±1.3%   63.2%±1.2%   66.7%±1.2%   65.2%±1.2%   59.6%±1.3%
```

From here we can begin to compare heuristics as we can see there is now a distribution of performance. However we must be sure to remember that all heuristics are context dependent and any adjustment to call count or a variation in compute time will change which scores do best.

The following pages detail the score functions used.

```
"""
My moves - Opp moves - Distance from center:

This custom score balances heuristic of maximizing the players moves and
minimizing the opponents moves with the heuristic of avoiding the corners
of the board.
"""
def custom_score(game, player):

# No win situation
    if game.is_loser(player):
        return float("-inf")
    # Best outcome
    if game.is_winner(player):
        return float("inf")

    # Tuning parameter
    p0 = 0.5

    # Improved Score
    own_len = float(len(game.get_legal_moves(player)))
    opp_len = float(len(game.get_legal_moves(game.get_opponent(player))))
    score = own_len - opp_len

    # Center Score
    w, h = game.width / 2., game.height / 2.
    y, x = game.get_player_location(player)
    dist = float(abs(h - y) + abs(w - x))
    score -= p0*(dist)

    # My moves - Opp moves - Distance from board center
    return score
```

```
# Suggestion from reviewer
def custom_score_alt(game, player):

    # No win situation
    if game.is_loser(player):
        return float("-inf")
    # Best outcome
    if game.is_winner(player):
        return float("inf")

    # get current move count
    move_count = game.move_count

    # count number of moves available
    own_moves = len(game.get_legal_moves(player))
    opp_moves = len(game.get_legal_moves(game.get_opponent(player)))

    # calculate weight
    w = 10 / (move_count + 1)

    # return weighted delta of available moves
    return float(own_moves - (w * opp_moves))
    return score

"""
Find a way out:

This custom score simply favors more move options and near
the end of the game it looks further and further out to find
paths with the most available outs
"""
def custom_score_2(game, player):

    if game.is_loser(player):
        return float("-inf")

    if game.is_winner(player):
        return float("inf")

    # calculate weight
    d = game.move_count // 10

    # Moves avalible to Knight
    directions = [(-2, -1), (-2, 1), (-1, -2), (-1, 2),
                  (1, -2), (1, 2), (2, -1), (2, 1)]
    blanks = game.get_blank_spaces() # blank space on the board
    own_moves = set(game.get_legal_moves(player))
    own_moves -= set(game.get_legal_moves(game.get_opponent(player))) # My
moves - Opp moves

    # Move Knight up to 'depth' moves away from initial position
    for _ in range(d):
        new_moves = []
        # Find all spaces avalible from current depth
        for r, c in own_moves:
            new_moves.extend([(r + dr, c + dc) for dr, dc in directions
                if (r + dr, c + dc) in blanks])
        #if not len(new_moves):
        #    break
        own_moves = set(new_moves)
    # Return a score that is total number of spaces that can be reached in
'depth' moves
    return float(len(own_moves))
```

```
# Suggestion from reviewer
def custom_score_2_alt(game,player):
    if game.is_loser(player):
        return float("-inf")

    if game.is_winner(player):
        return float("inf")

    score = .0
    total_spaces = game.width * game.height
    remaining_spaces = len(game.get_blank_spaces())
    coefficient = float(total_spaces - remaining_spaces) / float(total_spa
ces)

    my_moves = game.get_legal_moves(player)
    opponent_moves = game.get_legal_moves(game.get_opponent(player))

    for move in my_moves:
        isNearWall = 1 if (move[0] == 0 or move[0] == game.width - 1 or
            move[1] == 0 or move[1] == game.height - 1) else 0
        score += 1 - coefficient * isNearWall

    for move in opponent_moves:
        isNearWall = 1 if (move[0] == 0 or move[0] == game.width - 1 or
            move[1] == 0 or move[1] == game.height - 1) else 0
        score -= 1 - coefficient * isNearWall

    return score


"""
Fast - My moves — Move count
This score returns the number of move available to the player
and subtracts the number of moves in the game effectively penalizing
one move for each level searched
"""
def custom_score_3(game, player):

    if game.is_loser(player):
        return float("-inf")

    if game.is_winner(player):
        return float("inf")

    return float(len(game.get_legal_moves(player))- game.move_count)
```