Nicholas Beshouri
AIND Project 2 Heuristic Analysis

1. Custom Heuristics

1.1. `custom_score`

`custom_score` is a modification of `improved_score` that uses `own_moves` and `opp_moves` in a ratio instead of subtracting them: `len(own_moves) / max(len(opp_moves), 1e-6)`.

Using a ratio corrects a potential problem in `improved_score` where a state that was guaranteed to lead to a win (e.g. `own_moves = 2, opp_moves = 0, 2 - 0 = 2`) could be rated lower than a state that could potentially result in a loss (e.g. `own_moves = 5, opp_moves = 2, 5 - 2 = 3`). Dividing by `opp_moves`, or by a very small number when `opp_moves` is zero, means that states where the opponent has no remaining moves are always rated higher than ones that leave the opponent in the game.

In testing, `custom_score` generally out performed `improved_score`, but its performance varied considerably depending on the conditions of the game. In a 100,000 game trial where players were limited to a fixed search depth and allowed to choose their own starting positions, `custom_score` beat `improved_score` 62.653% of the time (Table 1). In a similar trial where starting positions were assigned randomly, `custom_score`'s win rate fell to 51.542%, suggesting `custom_score`'s advantages are most relevant in the early part of the game.

In a 2,000 game trial where both players were allowed to use iterative deepening and to choose their own starting positions, `custom_score` beat `improved_score` 55.0% of the time (Table 1). This decrease in performance over the fixed depth trial isn't surprising: the deeper players search, the earlier in the game they can see all the way to the terminal states, and the less relevant any difference in their heuristics becomes. As in the fixed depth trial, when I repeated the trial with random starting positions, `custom_score`'s win rate declined, in this case all the way to 49.95%.

1.2. `custom_score_2`

`custom_score_2` returns the same value as `custom_score` until the board is about half full and then returns the difference in the longest path each player could potentially follow from their current position: `longest_path(game, player) - longest_path(game, opponent)`.

In both fixed depth trials, `custom_score_2` improves on `custom_score` by a few percentage points. However, the high cost of computing walkable paths means that, in time limited iterative deepening trials, the player using `custom_score_2` wasn't able to search as deeply (Table 3) and so underperformed `custom_score`. This suggests that while `custom_score_2` is a better heuristic than `custom_score` in absolute terms, it isn't worth the extra time it takes to calculate it.

1.3. `custom_score_3`

`custom_score_3` starts with `custom_score`'s ratio and then, in the hope of chasing down and cornering the opponent, subtracts the normalized distance between the two players: `len(own_moves) / max(len(opp_moves), 1e-6) - norm_dis`.

`custom_score_3`'s performance was very similar to `custom_score`'s in all conditions.

2. Heuristic Recommendation

I recommend `custom_score` because:

- It performs a bit better than improved_score in some conditions.
- Unlike `custom_score_2`, it's fast to compute and so continued to perform well in time sensitive trials using iterative deepening.
- It's simpler and more general than `custom_score_3`, which doesn't significantly out perform it.

Note: In an earlier version of this analysis, I recommended `custom_score_3`, mostly do to its high performance in one test condition (iterative deepening without random starting positions). When I reran that trial in the process of studying average search depth, custom_score_3's advantage over custom_score disappeared and didn't reappear in repeated testing. I believe the earlier result to have been an anomaly and so have changed my recommendation.

Table 1. Wins Against `improved_score`

	Fixed Search Depth of 3 100,000 Games	Fixed Depth Random Starting Positions	Iterative Deepening 30 ms limit 2,000 Games	Iterative Deepening Random Starting Positions
`null_score`	20525 (20.525%)	20132 (20.132%)	853 (42.65%)	913 (45.65%)
`improved_score`	50513 (50.513%)	49819 (49.819%)	1014 (50.7%)	1002 (50.1%)
`custom_score`	62653 (62.653%)	51542 (51.542%)	1100 (55.0%)	999 (49.95%)
`custom_score_2`	65907 (65.907%)	53179 (53.179%)	908 (45.4%)	731 (36.55%)
`custom_score_3`	61026 (63.628%)	52230 (52.23%)	1107 (55.35%)	1001 (50.05%)

These data were produced using the simple testing function in Figure 1.

Table 2. Tournament Results

	AB_Improved	AB_Custom	AB_Custom_2	AB_Custom_3
Random	937 | 63	938 | 62	939 | 61	949 | 51
MM_Open	739 | 261	773 | 227	713 | 287	776 | 224
MM_Center	865 | 135	878 | 122	860 | 140	885 | 115
MM_Improved	718 | 282	742 | 258	670 | 330	751 | 249
AB_Open	527 | 473	550 | 450	410 | 590	547 | 453
AB_Center	589 | 411	585 | 415	466 | 534	579 | 421
AB_Improved	519 | 481	490 | 510	397 | 603	475 | 525
Win Rate:	69.9%	70.8%	63.6%	70.9%

These data were produced using the supplied tournament.py file, modified to increase the match count. The time limit for each move remained 150 milliseconds.

Table 3. Average Search Depth

	Search Depth in 30 ms Averaged Over 100 games
`null_score`	10.06
`improved_score`	9.117
`custom_score`	9.182
`custom_score_2`	7.930
`custom_score_3`	9.111

Tested on a 2.3 GHz 2012 MacBook Pro.

Figure 1. Simple Test Function

```
def ab_test(score_func, iterations=1000, iterative=True, time_limit=2000, random_first=False):
    player = AlphaBetaPlayer(score_fn=score_func, iterative=iterative)
    opponent = AlphaBetaPlayer(score_fn=improved_score, iterative=iterative)
    player_wins = 0
    opponent_wins = 0

    for i in range(iterations):
        # Alternate starting player.
        if i % 2:
            game = Board(player, opponent)
        else:
            game = Board(opponent, player)

        # If random_first flag is set, apply two random moves.
        if random_first:
            for _ in range(2):
                move = random.choice(game.get_legal_moves())
                game.apply_move(move)

        # Play the game and update the win counts.
        winner, history, outcome = game.play(time_limit=time_limit)
        if winner == player:
            player_wins += 1
        else:
            opponent_wins += 1

    return (player_wins, opponent_wins)
```