1_0. What features of the game does your heuristic incorporate, and why do you think those features matter in evaluating states during search?

Every cell on the gaming board is represented by corresponding board index value as shown in Table1A. These board index values have been encoded in isolation.py by Isolation class. Both game playing agents' board locations can be tracked by their index value contained in game_state.locs[player id] from isolation.py before each CustomPlayer agent's turn. Knowing both players' board locations, a number of useful items can be coded in order to analyze every node/position. For example, a number of available moves in a game node can be obtained for both playing agents. Or indexes located in board corners or perimeter can be assigned specific values for later use in weighted heuristic functions, since these cells do not yield as many moves as the ones closer to the board center. Given that most cells have not been blocked yet, an agent can only make two or three moves from corner cells as opposed to as many as eight moves from center locations.

Table 1A. Board position index values as encoded in isolation.py

114	113	112	111	110	109	108	107	106	105	104
101	100	99	98	97	96	95	94	93	92	91
88	87	86	85	84	83	82	81	80	79	78
75	74	73	72	71	70	69	68	67	66	65
62	61	60	59	58	57	56	55	54	53	52
49	48	47	46	45	44	43	42	41	40	39
36	35	34	33	32	31	30	29	28	27	26
23	22	21	20	19	18	17	16	15	14	13
10	9	8	7	6	5	4	3	2	1	0

1.1 Use board index locations

1.1 Board index values used to identify less favourable board locations for the opposing game playing agent.

In order to minimize the number of moves for the opposing agent, certain nodes such as the ones containing edge or corner locations could be selected from where an agent is limited to two or four possible moves or even less as the gameboard becomes more populated. These positions are singled

out in is_opp_loc_on_edge() function where adversary agent's location is checked for membership in the following index arrays:

edge_cells = [112,111,110,109,108,107,106,88,78,75,65,62,52,49,39,36,26,8,7,6,5,4,3,2]

corner_cells = [114,113,101,105,104,91,23,10,9,13,1,0].

Opposing agent board index is available from state.locs[1-self.player_id] which is supplied by the game state.

Edge and corner locations are highlighted in blue and red respectively in Table 1B.

The corner spaces are assigned better scores compared to edge locations, since the former are more restrictive. In CustomPlayer.get_action() this heuristic is employed in combination with others when a node is evaluated.

Table 1B. Corner and edge cell locations on gaming board.

114	**113**	**112**	**111**	**110**	**109**	**108**	**107**	**106**	**105**	**104**
101	100	99	98	97	96	95	94	93	92	**91**
88	87	86	85	84	83	82	81	80	79	**78**
75	74	73	72	71	70	69	68	67	66	**65**
62	61	60	59	58	57	56	55	54	53	**52**
49	48	47	46	45	44	43	42	41	40	**39**
36	35	34	33	32	31	30	29	28	27	**26**
23	22	21	20	19	18	17	16	15	14	**13**
10	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

1.2 Use board index locations and base-10 game state to track the number of blank spaces in center of the board.

It is feasible to calculate how many blank spaces there are in certain board regions. Spaces in and around mid board are most valuable especially in the first 25-32 moves or so and they will be contested by both playing agents. One way to track number of blank board spaces is to read long-integer-game-state from Isolation.state[0], convert the integer to binary bitboard string, and then count number of '1' characters by iterating that string. To get a better idea about what board state appears like:

An example of board state in base-10 long integer format: 41523161203939122082683632224299007.

An example of bitboard string format:

'1111111111100111111111110011111111111001111111111100111110111110011111111111001111111111111000111111111111100111111111110'.

In a bitboard string, '1' characters represent available cells and '0' strings represent blocked or occupied cells. To find a specific board cell in a bitboard string, a single calculation is performed: cell = 114-I where cell is board index value and I is string character index. As was mentioned before, cells marked with '1' characters are blank locations. Once such a character is encountered, its board location is checked whether it is in the specified board regions such as center-west, or south-east, or north-west, etc.

Current game state in the form of a bitboard string can be obtained by importing DebugState() class from isolation.py and by executing the following code:

DebugState.from_state(state).bitboard_string.

In this version of CustomPlayer.get_action(), the bitboard string is derived from game state and returned by dec_to_bin() function.

Table 1C. Cell locations in center_west highlighted in red and in center_east highlighted in blue.

114	113	112	111	110	109	108	107	106	105	104
101	100	99	98	97	96	95	94	93	92	91
88	87	86	85	84	83	82	81	80	79	78
75	74	73	72	71	70	69	68	67	66	65
62	61	60	59	58	57	56	55	54	53	52
49	48	47	46	45	44	43	42	41	40	39
36	35	34	33	32	31	30	29	28	27	26
23	22	21	20	19	18	17	16	15	14	13
10	9	8	7	6	5	4	3	2	1	0

During opening stages of the game, the custom playing agent is directed towards center_west and center_east areas displayed in Table 1C. As the game progresses, the game state bitboard string is iterated every time CustomPlayer.get_action() is called and the number of blank spaces within both regions is calculated. If the board index value is a blank cell and it is contained in these center areas:

center_west = [86,85,84,73,72,71,60,59,58,47,46,45,34,33,32]

center_east = [82,81,80,69,68,67,56,55,54,43,42,41,30,29,28]

then the b_sectors dictionary gets updated and its first value is increased by one in the following code:

b_sectors['center_west'][0] += 1 for center_west quadrant or b_sectors['center_east'][0] += 1 for center_east.

In the beginning, there are 15 vacant cells in each region. However, the board center becomes less vacant after about 20-25 moves and as soon as 6 or 7 blank spaces left in each sector, the custom playing agent is directed toward board's perimeter.

1.3 Use board index locations and base-10 game state to track the number of blank spaces in corners of the board.

Same method described in previous section is employed to keep track of blank spaces in corner areas of the board.

Table 1D. Corner board sectors painted in red.

114	113	112	111	110	109	108	107	106	105	104
101	100	99	98	97	96	95	94	93	92	91
88	87	86	85	84	83	82	81	80	79	78
75	74	73	72	71	70	69	68	67	66	65
62	61	60	59	58	57	56	55	54	53	52
49	48	47	46	45	44	43	42	41	40	39
36	35	34	33	32	31	30	29	28	27	26
23	22	21	20	19	18	17	16	15	14	13
10	9	8	7	6	5	4	3	2	1	0

north_west

north_east

south_west

south_east

Four board sectors, labeled north_west, south_west, north_east, and south_east and illustrated in Table 1D, are monitored during mid and end game. Blank cells are iterated and counted from current game state and depending on how vacant these regions are, the custom playing agent is guided there accordingly.

1.4 Board index values used as a means in guiding custom playing agent toward vacant spaces.

Once most vacant sector is determined, the custom playing agent must be directed towards it or remain within it. All six board areas or sectors are assigned a single index value that serves as a location for the game playing agent to aim for. Sector location index values are stored in b_sectors[] second value. These values were selected because they are located close to the middle of each region of interest. For example, let's assume that on this game turn north_west corner is most vacant, meaning it holds the most blank cells compared to the other three corners. Then the target index value for the custom playing agent to reach becomes 87 which is read from this line of code b_sectors['north_west'] = [0, 87]. In case of south_west corner being most vacant area, target location becomes 35 (b_sectors['south_west'] = [0, 35]). Goal index values for all four corner regions are displayed in Table 1E.

Table 1E. Goal index values are printed in red. Blue cells constitute corner regions which are checked for blank spaces.

114	113	112	111	110	109	108	107	106	105	104
101	100	99	98	97	96	95	94	93	92	91
88	87	86	85	84	83	82	81	80	79	78
75	74	73	72	71	70	69	68	67	66	65
62	61	60	59	58	57	56	55	54	53	52
49	48	47	46	45	44	43	42	41	40	39
36	35	34	33	32	31	30	29	28	27	26
23	22	21	20	19	18	17	16	15	14	13
10	9	8	7	6	5	4	3	2	1	0

north_west

north_east

south_west

south_east

Custom playing agent guiding heuristic will be explained next. As soon as the most vacant area is determined, all nodes are evaluated based on board index distance heuristic and the node that gets the custom playing agent closer to the goal cell is selected. The main idea behind the node evaluation function is to simply measure the distance between custom playing agent's position and the goal board index value and subsequently determine the node with the closest distance.

Custom playing agent's location is tracked by state.locs[self.player_id].

Knowing board index values for both target cell and the agent's, distance between these two locations is computed in the following way:

first, both index values are converted to XY coordinates and then absolute values of X1 -X2 and Y1-Y2 are summed up to produce the distance. XY coordinate conversion is returned by ind_to_xy(ind) function and is implemented in the same way as ind2xy(index) DebugState class method in isolation.py.

Utilizing two different heuristic functions can be useful in game node evaluation. A function combining board distance heuristic and heuristic that finds less favorable positions for the opposing player, described in section 1.1, is employed during first 41 moves in this version of CustomPlayer.get_action() function. After 41 moves, nodes are evaluated by a combination of index distance and number of available moves for custom playing agent heuristics.

1.5 Utilizing game state's ply_count to deploy different heuristic functions throughout the game.

Ply or move count is available from game state tuple and it could be used to separate the game into several stages such as opening, mid, or end game. There are several reasons for this method.

First 31 moves are most demanding on system resources and the search rarely goes past depth 4 or 5, given 150 ms time limit for each move. Therefore, the shortest time limit is coded for that stage of the game.

For the next ten moves, time limit can be increased, since the game tree is reduced. And there is no need to check if center board areas are still available. At this point, most of mid board cells are more or less blocked.

My_moves heuristic becomes more important after 41 moves. And it is evaluated at this stage along with board distance heuristic. Time limit is increased as well which allows searching at additional depths.

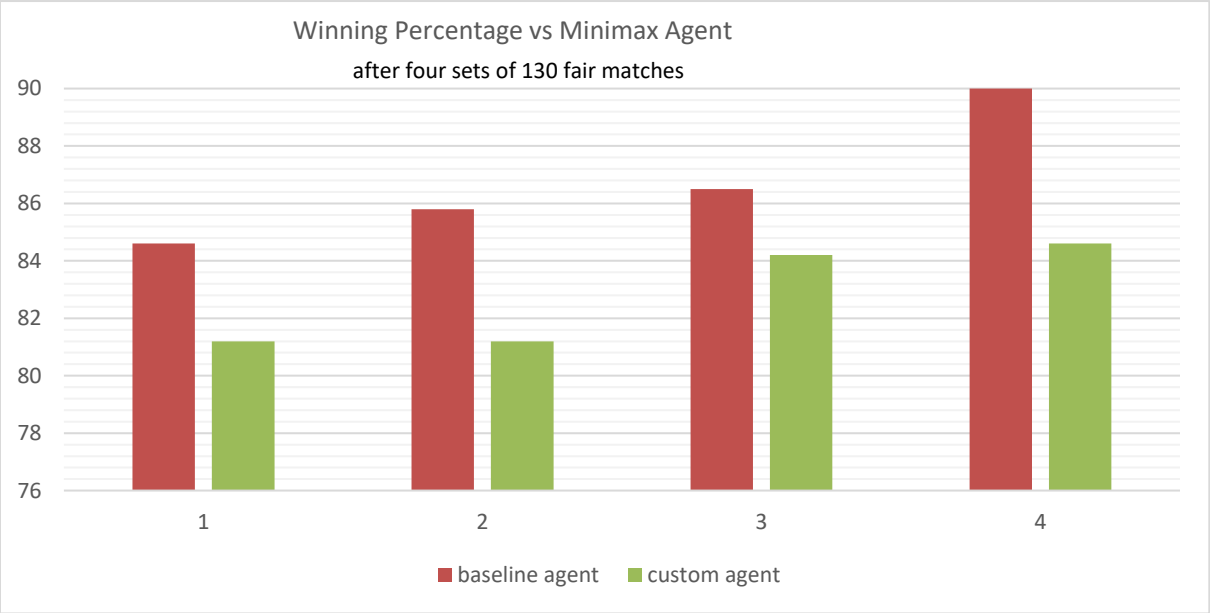
2_0. Evaluate effectiveness of your agent using the #my_moves - #opponent_moves heuristic as well as the effectiveness of your agent using your own custom heuristic.

Baseline heuristic agent outperformed custom playing agent after having played four sets of 130 fair matches against minimax agent. This is evident from experiment results illustrated in Chart 2A.

It is evident that #my_moves - #opponent_moves heuristic is elegant and very effective especially when it is deployed at search depths 7 and higher. Even better results could likely be obtained by assigning different weights to either my_moves or opponent_moves, depending on how populated the game

board is. For example, 3* my_moves - opponent_moves could be useful during end game, because finding nodes with more legal actions are more important in order not to get stuck in a dead-end position.

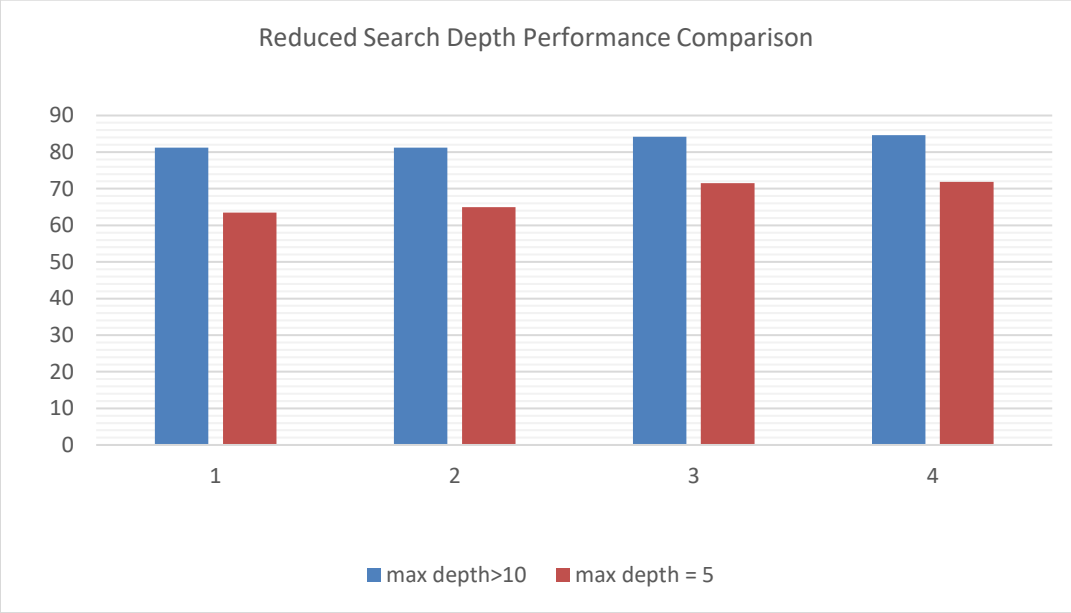
Chart 2A. Baseline and custom agent effectiveness comparison.



3_0. Analyze the search depth your agent achieves using your custom heuristic. Does search speed matter more or less than accuracy to the performance of your heuristic?

Greater search speed means reducing search depth which in turn means lesser winning rate or reduced performance provided that same heuristic functions have been utilized. This conclusion was made after running 130 fair matches with custom playing agent whose search depths were limited. Search depth was limited to 3 in the first 31 moves, 4 in the next ten, and 5 in the rest of moves after 41. Consequently, faster custom playing agent experienced lesser winning rate compared to the agent whose search was time limited instead of depth. Previous game playing results from 130 fair matches are stacked against the results for the agent whose search depth was restricted in Chart 3A.

Chart 3A. Search depths were fixed at 3,4, and 5 for the sake of experiment, causing poorer performance.



Experiment results from testing baseline heuristic lead to the same conclusion. Baseline agent achieves search depths of 8 during first 41 game moves and depths over 40 during the rest of the game. This alone gives the baseline agent up to 90 percent winning edge over opposing agent that runs the same heuristic but is limited to search depth 3.