Introduction

This is my analysis of the heuristics I came up with for the Isolation game in Udacity's Artificial Intelligence Nanodegree. The goal of the this project was to implement minimax search with alpha-beta pruning and iterative deepening and come up with 3 different heuristics in hopes that one of them consistently outperforms the "improved" heuristic from lectures. The ideas I came up with fall into three categories: positional heuristics, generalized versions of "improved," and combinations of the first two. Each heuristic was run using `tournament.py` with `NUM_MATCHES` set to 25 (so 100 total matches against each test agent). Also, I abbreviated the names of the test agents (e.g. MM_Null to MM_N) to save space on the tables. I hope that this doesn't cause any confusion.

First, here at the results I got for ID_Improved:

Random	MM_N	MM_O	MM_I	AB_N	AB_O	AB_I	Total
79%	72%	65%	55%	74%	65%	60%	67.14%

Positional Heuristics

The first idea was to value the squares in the center of the board more than those on the edges. I decided this because the edge squares have fewer moves in general than those in the middle. I used the ℓ_1-norm (or Manhattan distance) to compute the distance of each player from the center. The heuristic returned the difference of the two player's difference from the center. Also, it was simple to reverse that value, so I include those results here as well:

	Random	MM_N	MM_O	MM_I	AB_N	AB_O	AB_I	Total
Value Center	83%	71%	62%	56%	68%	57%	61%	65.43%
Value Edges	80%	73%	62%	51%	70%	50%	54%	62.86%

There does not seem to be a big difference in performance between these two strategies. However, valuing center position seems to do better against the alpha-beta test agents. There might also be a difference when we consider these two strategies for early moves. For example, if we can use a lot of the edge squares early in the game and leave ourselves with more squares in the center, then perhaps we would have an advantage. We will see how this worked out in a later section. These are `close_to_center_score` and `far_from_center_score` in the code.

The next positional heuristic I had in mind compared the distance of the two players' position. Similar to the above, there were two versions: one that valued being close to the opponent and one that valued be as far as possible from the opponent.

	Random	MM_N	MM_O	MM_I	AB_N	AB_O	AB_I	Total
Close	75%	69%	60%	55%	71%	56%	61%	63.86%
Far	81%	75%	55%	50%	71%	56%	61%	64.86%

These are `close_to_opponent_score` and `far_from_opponent_score` in the code. Neither of these strategies seems to be advantageous. In fact, we don't see much difference if we compare these results to the results of the "distance from center" heuristics. More importantly, none of these positional heuristics out performed ID_Improved.

Generalized Versions of the Improved Heuristic

Next I was determined to generalize the idea behind ID_Improved. I realized that ID_Improved only considered how many moves each player had in their very next move. So I devised a way (with the help of looking up knight movement analysis in chess) to take all successive moves into account. First, consider the following 5-by-5 Isolation board (X denotes a blocked square):

		X		X
X	X	1		
			X	
	X			
			2	

For each player we use breadth first search to find the shortest path to every other square, including the blocked squares.

Player 1

1	2	3	2	1
2	3	0	3	2
1	2	3	2	1
4	1	2	1	4
3	2	3	2	3

Player 2

3	2	3	2	3
2	3	2	3	2
3	4	1	2	1
4	1	2	3	4
3	2	3	0	3

Next, we take some $0 < r < 1$ and calculate r^n for each square:

Player 1

r^1	r^2	r^3	r^2	r^1
r^2	r^3	r^0	r^3	r^2
r^1	r^2	r^3	r^2	r^1
r^4	r^1	r^2	r^1	r^4
r^3	r^2	r^3	r^2	r^3

Player 2

r^3	r^2	r^3	r^2	r^3
r^2	r^3	r^2	r^3	r^2
r^3	r^4	r^1	r^2	r^1
r^4	r^1	r^2	r^3	r^4
r^3	r^2	r^3	r^0	r^3

Note that these only have to be calculated once and we can store these as a 2D-list in python. Now we take the unblocked squares and add their values for each player and subtract these totals. For example, if $r = 0.5$, then

Player 1

0.5	0.25		0.25	
		1	0.125	0.25
0.5	0.25	0.125		0.5
.0625		0.25	0.5	.0625
0.125	0.25	0.125		0.125

Player 2

0.125	0.25		0.25	
			0.125	0.25
0.125	.0625	0.5		0.5
.0625		0.125	0.125	.0625
0.125	0.25	0.125	1	0.125

So Player 1 has a score of 5.25, Player 2 has a score of 4.1875, and the heuristic will output $5.25 - 4.1875 = 1.0625$. We used the example of $r = 0.5$ here, but we can use whatever r we like as long as $0 < r < 1$. Below are the outcomes of some values of r:

r	Random	MM_N	MM_O	MM_I	AB_N	AB_O	AB_I	Total
$\mathbf{\frac{1}{2}}$	83%	76%	66%	71%	74%	65%	55%	70.00%
$\mathbf{\frac{1}{3}}$	94%	78%	70%	71%	70%	64%	70%	73.86%
$\mathbf{\frac{1}{4}}$	84%	81%	64%	68%	77%	59%	61%	70.57%
$\mathbf{\frac{1}{5}}$	83%	81%	69%	60%	74%	64%	60%	70.14%
$\mathbf{\frac{1}{6}}$	85%	78%	74%	66%	79%	67%	61%	72.86%
$\mathbf{\frac{1}{7}}$	84%	81%	65%	64%	81%	65%	72%	73.14%
$\mathbf{\frac{1}{8}}$	89%	70%	69%	64%	75%	70%	64%	71.57%
$\mathbf{\frac{1}{9}}$	81%	84%	66%	64%	78%	64%	62%	71.29%
$\mathbf{\frac{1}{10}}$	88%	89%	74%	73%	71%	70%	69%	76.29%

Regardless of the value of r, it is clear that this method is superior to ID_Improved. The value of r that stands out the most is $r = \frac{1}{10}$ because it performed the best overall. However, this value seems arbitrary and unsatisfying. This led me to the next version of this algorithm.

Instead of calculating the shortest paths on the game board as if it were empty and then removing the blocked squares, we should calculate the shortest path with the blocked squares considered at each step. This makes storing the 2D-list much more difficult, but it turns out that we can do without the storage. Using this idea, the shortest path arrays now look like:

Player 1

1	2	X	2	X
X	X	0	3	2
1	2	3	X	1
4	X	2	1	4
3	2	3	X	3

Player 2

5	2	X	2	X
X	X	X	3	2
3	4	1	X	1
2	X	2	3	2
3	2	3	0	3

Now instead of choosing $0 < r < 1$, add one to the length of the path and take the reciprocal:

Player 1

$\frac{1}{2}$	$\frac{1}{3}$	X	$\frac{1}{3}$	X
X	X	1	$\frac{1}{4}$	$\frac{1}{3}$
$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	X	$\frac{1}{2}$
$\frac{1}{5}$	X	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{1}{5}$
$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{4}$	X	$\frac{1}{4}$

Player 2

$\frac{1}{6}$	$\frac{1}{3}$	X	$\frac{1}{3}$	X
X	X	X	$\frac{1}{3}$	$\frac{1}{3}$
$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{2}$	X	$\frac{1}{2}$
$\frac{1}{3}$	X	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{3}$
$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{4}$	1	$\frac{1}{4}$

From here, we add the squares for each player and take the difference. For Player 1 we have a score of 6.65, and for Player 2 we have a score of 5.033. Therefore, the heuristic returns $6.65 - 5.033 = 1.62$. Below are the outcomes of this heuristic:

Random	MM_N	MM_O	MM_I	AB_N	AB_O	AB_I	Total
86%	88%	68%	68%	82%	67%	63%	74.57%

This heuristic performs better than ID_Improved and on the same scale as the first generalized versions of ID_Improved. However, because we avoid the need to choose an $0 < r < 1$, this heuristic will be the one we use going forward. Let's call this one `bfs_score`.

Combinations of the Two

The final set of heuristics we will analyze are a combination of `bfs_score` with the positional heuristics discussed above. One of these will be our final custom heuristic.

First, we will look at `close_to_opponent_score + bfs_score`:

Random	MM_N	MM_O	MM_I	AB_N	AB_O	AB_I	Total
85%	83%	73%	64%	79%	62%	62%	72.57%

Next, `far_from_center_score + bfs_score`:

Random	MM_N	MM_O	MM_I	AB_N	AB_O	AB_I	Total
87%	83%	63%	61%	77%	66%	53%	70.00%

`close_to_opponent_score + bfs_score`:

Random	MM_N	MM_O	MM_I	AB_N	AB_O	AB_I	Total
92%	86%	64%	71%	83%	75%	60%	75.86%

Lastly, `far_from_opponent_score + bfs_score`

Random	MM_N	MM_O	MM_I	AB_N	AB_O	AB_I	Total
87%	76%	62%	64%	81%	62%	53%	69.29%

The highest performing heuristic here is `close_to_opponent_score + bfs_score`. The performance is very close to `bfs_score` alone, but the newer version was able to beat the Random opponent 92% of the time. For these reasons, this is our `custom_score` in the code.

Conclusions

Overall, this project was interesting and fun to work on. If I had more time I would try more combinations of the heuristics with tuned scaling factors for each term. I would also try shifting the strategy as the game progressed. For example, we might try taking as many edge squares as we can in the beginning and shift to taking as many center squares as possible. Of course, we could also try the reverse as well.

It is worth noting two additional things regarding my work on this project:

1. None of the tests I reported on here timed out during the run. All of them (run on my MacBook Pro) were able to produce a move with `self.TIMER_THRESHOLD = 10.`
2. I was able to determine that if the agents were able to choose their first moves instead of begin positioned on the board randomly, then the second player is able to win almost every time. I say "almost" because I was not able to prove it mathematically. The proof would follow from the fact that the parity of the row and column of the player changes after every move. That is, if the row and column of the player's position are both odd, then after the next move one will remain odd and the other will become even.