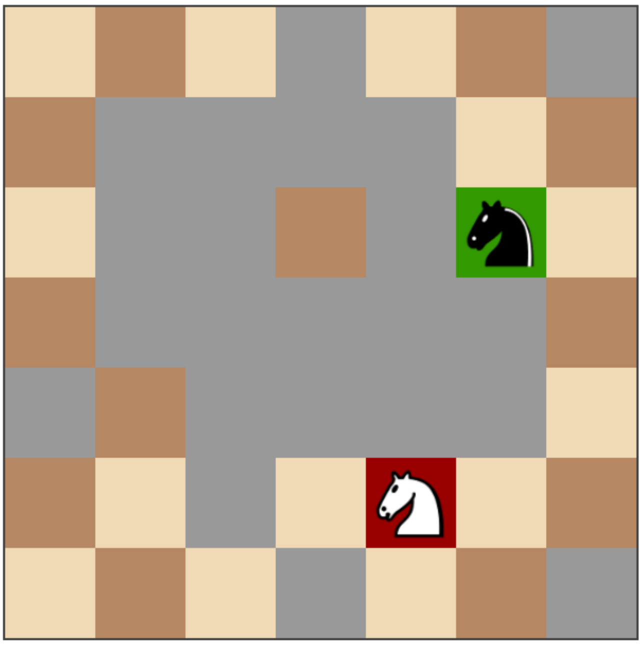
Custom score functions

In this version of Isolation, the player whose turn it is when both pieces are on squares of the same color has an advantage. The reason is that whenever there is a square that both players can reach (which means that they are both on a square of the opposite color), this player moves first. If moving to this square is good, the player can not only reach a good square, but he can also prevent his opponent from reaching a good square. The following position illustrates this.

Player1	Player2
(3,5)	(4,0)
(4,3)	(2,1)
(2,2)	(4,2)
(3,4)	(6,3)
(1,3)	(4,4)
(3,2)	(5,2)
(1,1)	(3,1)
(0,3)	(1,2)
(2,4)	(3,3)
(4,5)	(1,4)
(6,6)	(0,6)
(5,4)	(2,5)



White to move

Here (4, 6) is a good square. From there a few more moves along the bottom of the board will be available. As it is White's turn, he can move to this square, thus preventing Black from moving there.

In the following we call the player who has this advantage the *attacker*, and we call his opponent the *defender*. In general, a position where both players compete for the same squares favors the attacker. If the possible moves for both players do not overlap, the attacker's advantage is neutralized. custom_score_2 makes use of this observation.

custom_score_2: This heuristic increases the value from improved_score by 3 if there is a square which can be reached by both players in the next move. The value 3 was found by experimenting. This score function is about as good as improved_score.

custom_score_3: The score functions from the lecture and custom_score_2 only look at the squares that each player can reach in the next move. This function looks at the squares that each player can reach in the next k moves (for some $k \geq 2$). We write $s_k(p)$ for the number of squares that the player p can reach in the next k moves. Then custom_score_3 is calculated as

$$s_k(player) - s_k(opponent)$$

It turns out that 2 and 3 work best as values for k. In the implementation $k = 2$ is used. This score function takes more time than the functions from the lecture and custom_score_2. Its performance is also similar to that of improved_score.

custom_score: This score function combines the ideas from custom_score_2 and custom_score_3. We write $s_k(both)$ for the number of squares that can be reached by both players in the next k moves. As the attacker wants to reach a position where both players compete for the same squares and the defender wants to avoid this, we define the score function as

$$s_k(player) - s_k(opponent) + c \cdot s_k(both)$$

if *player* is the attacker and

$$s_k(player) - s_k(opponent) - c \cdot s_k(both)$$

if *player* is the defender.

c is a constant value. After experimenting with different values for k and c I chose $k = 2$ and $c = 0.5$. Although custom_score takes more time than the score functions from the lecture, it performs slightly better.

Tournament results

As a final test of the score functions, I used tournament.py to compare them. The opponents that are used for the comparison are AB_Open (Alphabeta with open_score), AB_Improved (Alphabeta with improved_score), MM_Improved (Minimax with improved_score), and MonteCarlo (a Monte Carlo search tree agent that I implemented).

The test agents consist of one agent for each of my custom score functions, and AB_Impr to allow a direct comparison with the custom score functions. The agent named Custom (see the file competition_agent.py) is an improved version of the alpha-beta algorithm that uses the same score function as AB_Custom. Custom implements a suggestion from the AIMA book: in each iteration step, the move that appeared best in the last step is considered first. Thus more branches can be pruned by the alpha-beta algorithm.

Here are the match results:

Match #	Opponent	AB_Custom		AB_Custom_2		AB_Custom_3		Custom		AB_Improved	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	AB_Open	59	41	58	42	55	45	67	33	51	49
2	AB_Improved	59	41	48	52	53	47	60	40	51	49
3	MM_Improved	88	12	77	23	84	16	87	13	83	17
4	MonteCarlo	78	22	70	30	70	30	76	24	76	24
	Win Rate:	71.0%		63.2%		65.5%		72.5%		65.2%	

The results show that my custom evaluation function (AB_Custom) is stronger than the others (AB_Custom_2, AB_Custom_3, and AB_Improved). Improving the alpha-beta algorithm by reordering the moves (Custom) yields even slightly better results.