I tried many evaluation functions, beginning with the most basic: number of valid moves and number of valid moves minus opponent number of valid moves, that are actually implemented in the examples. Then, I began to try combinations of these functions.

I finished using the basic “actual minus opponent moves”, that showed best results, but trying to apply some spatial constraints: stay on center, stay close to opponent, and a combination depending on game state.

I also tried, as suggested on videos and by the reviewers, adding weights to the evaluation function. I tried to apply actual moves minus 2 * opponent moves. Since isolation is about taking off adversary space, this aggressive behavior could lead to better results. But in this specific board, I bet because the movement of the horse is “non blocking”, the results did not show a huge difference. Other types of moves (bishop, tower, queen), certainly would reward more the aggressive and the “stay close” by prisoning the adversary fast.

On the other hand, the idea behind applying spatial constraints was that, depending on the game, a good position on the board can lead a player to advantage. Since searching the entire game space is difficulty or impossible, this assumption can simplify the search.

The combination of constraints can be promising, but demands a design of constraints and an human made analysis of what constraint can lead a player to advantage depending on “global” state of game, using this kind of “table” to apply the correct constraint on the score function.

Running repeated times the matches, I got mixed results. Sometimes one function performs better, sometimes another, with no huge differences between each other. The random initialization seems to play a major role on results.

So, it demonstrates that the evaluation functions,including provided “improved_function” aren’t getting entirely the point, or, most probable, good openings matter more than evaluation functions in the first rounds of the game.

So, in resume:
1 – The number of valid moves minus number of opponent moves seeemed to perform better then other functions previously applied (stay on center, random, my moves, opponent moves)

2 – Staying with this function and trying to apply weighs showed mixed results.

3 – Applying spatial constraints also showed mixed results, but the mixed results itself show that initialization play an important role.

That said, *my proposal approach is to use activation function valid moves x opponent moves, running against itself many times in regression to learn best weights. Also, complement the utility function algorithm with an “opening” algorithm*, for first rounds of play. That would be done by repeatedly running the algorithm for every possible position on the first ‘n’ (n 2 to 6 probably) rounds and recording the positions that lead to more wins. Them, these “weights” should help to imply moves in the first rounds.

Perhaps, that complementary algorithm would be more generalizable than the constraint table mentioned before, could be “machine learned”, and would certainly scale more with the search space of the game – because the simulations would run before the actual game and results would be saved, simplifying the search on the time of the game, simply pruning alternatives known to be worse because of previous simulations. Remembering that even with simple games, on first rounds,

search spaces can be really huge, I think that pre-loaded maps would be a good approach to prune search space even before starting the search.

Description of the functions applied:

custom_score:
Returns actual player number of moves minus opponent number of moves divided by distance of center.

custom_score_2:
Returns opponent number of moves minus actual player number of moves divided by distance of opponent.

custom_score_3:
On begin of game, use 1, then start using 2.

Following, results of 5 consecutive results of running tournament on actual submitted code. I modified the headings of original test for more clarity, and added a Random Agent to test_agents, and a minimax that evaluates last level if reached timeout to cpu_agents:

```
Custom 3 (ran 3 times):
    if len(game.get_blank_spaces()) / (game.width * game.height) > 0.3:
        return custom_score_2(game, player)
    return custom_score(game, player)
```

Match #	CPU - TEST	Random		AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		TEST	CPU	TEST	CPU	TEST	CPU	TEST	CPU	TEST	CPU
1	Random	3	7	7	3	10	0	8	2	8	2
2	MM_Open	0	10	8	2	6	4	6	4	5	5
3	MM_Center	1	9	8	2	7	3	6	4	9	1
4	MM_Improved	1	9	5	5	5	5	4	6	5	5
5	MM_evtm_CS3	2	8	10	0	7	3	7	3	7	3
6	AB_Open	1	9	4	6	2	8	4	6	6	4
7	AB_Center	2	8	5	5	5	5	5	5	5	5
8	AB_Improved	1	9	3	7	2	8	3	7	5	5
	Win Rate:	13.8%		62.5%		55.0%		53.8%		62.5%	

Match #	CPU - TEST	Random		AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		TEST	CPU	TEST	CPU	TEST	CPU	TEST	CPU	TEST	CPU
1	Random	5	5	8	2	7	3	9	1	8	2
2	MM_Open	1	9	9	1	7	3	3	7	6	4
3	MM_Center	2	8	8	2	6	4	8	2	6	4
4	MM_Improved	2	8	8	2	4	6	3	7	7	3
5	MM_evtm_CS3	0	10	8	2	6	4	6	4	7	3
6	AB_Open	0	10	6	4	4	6	6	4	4	6
7	AB_Center	2	8	4	6	5	5	5	5	5	5
8	AB_Improved	2	8	4	6	5	5	3	7	2	8
	Win Rate:	17.5%		68.8%		55.0%		53.8%		56.2%	

```
Custom 3 (ran 1 time):
    if len(game.get_blank_spaces()) / (game.width * game.height) > 0.4:
        return custom_score(game, player)
    return custom_score_2(game, player)
```

Match #	CPU - TEST	Random		AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		TEST	CPU	TEST	CPU	TEST	CPU	TEST	CPU	TEST	CPU
1	Random	4	6	7	3	9	1	10	0	8	2

2	MM_Open	1	9	9	1	6	4	6	4	6	4
3	MM_Center	2	8	9	1	7	3	7	3	7	3
4	MM_Improved	1	9	6	4	7	3	3	7	5	5
5	MM_evtm_CS3	4	6	8	2	6	4	7	3	8	2
6	AB_Open	2	8	6	4	4	6	5	5	5	5
7	AB_Center	2	8	6	4	4	6	5	5	5	5
8	AB_Improved	0	10	6	4	2	8	3	7	4	6
	Win Rate:		20.0%		71.2%		56.2%		57.5%		60.0%

Custom 3 (ran 2 times):

```
    if len(game.get_blank_spaces()) / (game.width * game.height) > 0.2:
        return custom_score_2(game, player)
    return custom_score(game, player)
```

Match #	CPU - TEST	Random		AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		TEST	CPU	TEST	CPU	TEST	CPU	TEST	CPU	TEST	CPU
1	Random	6	4	8	2	8	2	10	0	8	2
2	MM_Open	1	9	6	4	7	3	6	4	4	6
3	MM_Center	2	8	7	3	7	3	7	3	6	4
4	MM_Improved	3	7	7	3	5	5	6	4	6	4
5	MM_evtm_CS3	0	10	7	3	7	3	7	3	8	2
6	AB_Open	2	8	4	6	3	7	6	4	3	7
7	AB_Center	3	7	5	5	5	5	4	6	5	5
8	AB_Improved	1	9	4	6	3	7	4	6	4	6
	Win Rate:	22.5%		60.0%		56.2%		62.5%		55.0%	

Match #	CPU - TEST	Random		AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		TEST	CPU	TEST	CPU	TEST	CPU	TEST	CPU	TEST	CPU
1	Random	4	6	9	1	7	3	8	2	9	1
2	MM_Open	0	10	7	3	5	5	7	3	6	4
3	MM_Center	3	7	7	3	7	3	7	3	8	2
4	MM_Improved	0	10	7	3	7	3	6	4	3	7
5	MM_evtm_CS3	0	10	7	3	10	0	5	5	7	3
6	AB_Open	4	6	4	6	3	7	5	5	5	5
7	AB_Center	3	7	4	6	6	4	6	4	4	6
8	AB_Improved	1	9	3	7	4	6	1	9	4	6
	Win Rate:	18.8%		60.0%		61.2%		56.2%		57.5%	

LAST RUN

Custom_score

```
    # Returns actual player number of moves minus
    # 2 times opponent number of moves
```

Custom_score_2

```
    # Returns actual player number of moves minus
    # 2 times opponent number of moves
    # divided by distance of oponent.
```

Custom_score_3

```
    if len(game.get_blank_spaces()) / (game.width * game.height) > 0.6:
        return custom_score_2(game, player)
    return custom_score(game, player)
```

****
Playing Matches
****

Match #	CPU - TEST	Random		AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		TEST	CPU	TEST	CPU	TEST	CPU	TEST	CPU	TEST	CPU
1	Random	6	4	10	0	9	1	9	1	8	2
2	MM_Open	0	10	10	0	4	6	6	4	6	4
3	MM_Center	2	8	9	1	7	3	9	1	7	3
4	MM_Improved	0	10	7	3	4	6	4	6	6	4
5	MM_evtm_CS3	1	9	7	3	6	4	8	2	9	1

6	AB_Open	1	9	5	5	5	8	2	6	4
7	AB_Center	1	9	8	2	6	4	7	3	6	4
8	AB_Improved	0	10	4	6	3	7	4	6	6	4
	Win Rate:	13.8%		75.0%		55.0%		68.8%		67.5%	

Match #	CPU - TEST	Random		AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3	
		TEST	CPU	TEST	CPU	TEST	CPU	TEST	CPU	TEST	CPU
1	Random	6	4	8	2	8	2	8	2	9	1
2	MM_Open	1	9	6	4	8	2	7	3	5	5
3	MM_Center	1	9	8	2	6	4	7	3	7	3
4	MM_Improved	1	9	6	4	8	2	6	4	6	4
5	MM_evtm_CS3	4	6	8	2	9	1	10	0	7	3
6	AB_Open	2	8	5	5	5	5	5	5	5	5
7	AB_Center	2	8	5	5	7	3	6	4	6	4
8	AB_Improved	2	8	4	6	3	7	3	7	5	5
	Win Rate:	23.8%		62.5%		67.5%		65.0%		62.5%	