Heuristic Analysis

Custom Score

(weighted_precomputed_move_advantage_with_initiative)

This heuristic is similar to the improved_score, except for two pretty straight forward changes: weighting moves and accounting for initiative

Weight

1) Not all destination squares are created equal. Originally I weighted each destination square by the number of exits that it had.

For example, the red square can reach 6 squares, but 2 of them are already covered, so it's value would be 4.

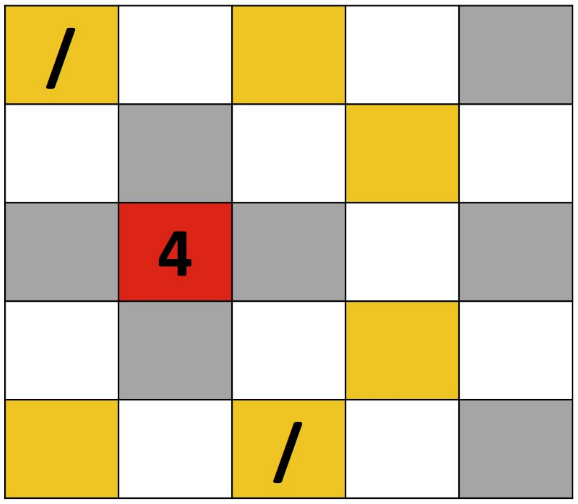
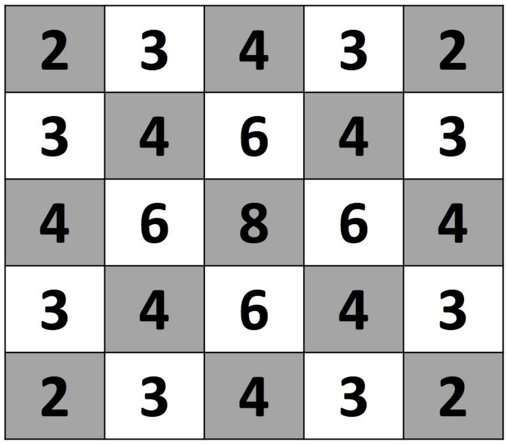


Figure 1: Exits nodes from the red square accounting for already taken squares

However, determining the number of exits on the current board turned out to be too computationally expensive, so instead I pre-computed based on a clear board. This is a good predictor, since corner nodes are guaranteed to have much fewer destinations than center nodes. The precomputed values (on a 5x5) would appear as follows:



2	3	4	3	2
3	4	6	4	3
4	6	8	6	4
3	4	6	4	3
2	3	4	3	2

Figure 2: Pre-computed exit nodes on a clear 5x5 board.

Thus this defines a weighting function: $weight(a) = \frac{exits(a)}{8}$

Using this pre-computation, the weighting improved score algorithm was written as follows:

```
weighted_move_advantage(game, player, opponent):
  if game.is_winner(player) return ∞
  if game.is_loser(player) return -∞
  weighted_moves = (p) => sum(game.moves.map(weight))
  return weighted_moves(player) - weighted_moves(opponent)
```

Listing 1: Weighted move advantage

Initiative

The second change it to realize that when a player does not have the initiative, next moves that he shares with his opponent are not as valuable, since they could be taken. A fractional weighting is selected to dis-advantage moves shared by the opponent. In practice we chose a weight of zero, meaning the player w/o initiative does not count moves that it’s opponent could take.

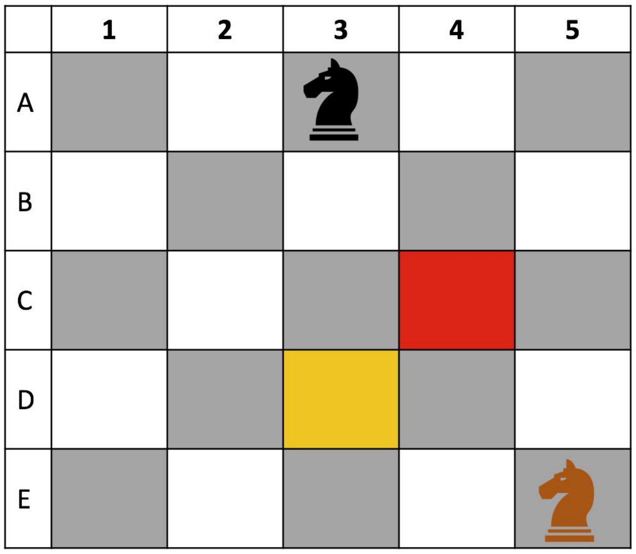


Figure 3: The orange knight does not get to count C4 since his opponent has initiative and can move there. D3 is his only counted move.

Results

Together these two weights drive the player towards moves that are away from the edge, towards moves that reduce the opponent's options when the player has initiative, and away from the opponent when the player does not.

A brief note on methodology
The tournament.py evaluation didn't seem rigorous enough to me to include here due to it's lower play count. Instead I devised a system to run 1000 games in parallel on AWS Lambda. Thus runs in about 20 seconds, and gives me far more accurate responses. However, I only ran comparisons between the improved_score function and the score function being discussed, so the full grid given in tournament.py that includes performance against the dumber bots is not included here. I think these numbers are much more rigorous than a sample size of 10.

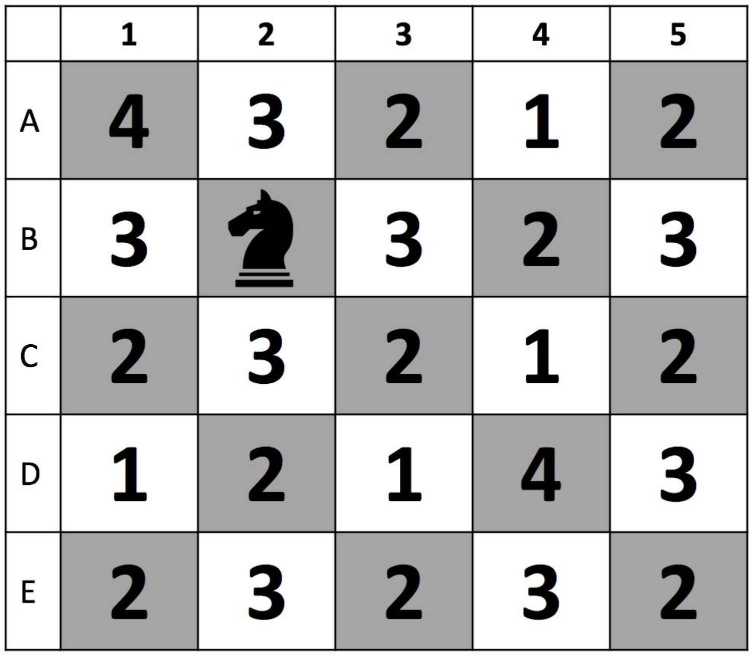
Evaluating the results of 1000 game plays against the custom_score function, this metric delivered won 543 times against an improved metric agent. That's a 54% win rate. It's not stellar, but it is better. This is slightly more expensive, causing the average search depth to decrease from 7.89 plies per move, to 7.62 plies.

Custom Score 2

(reachable_diff)

This heuristic's goal is to maximize the portion of the board that the player is able to reach while minimizing the opponents. It's a logical extension of the above heuristic. Where the above only looks at the **next** square the player can reach, this algorithm looks at **all** nodes the player can reach from it's current location. It essentially finishes playing out the game in single-player.

For example, the reachability, on an open 5x5 board, for a knight in B2, is as follows:



	1	2	3	4	5
A	4	3	2	1	2
B	3	♞	3	2	3
C	2	3	2	1	2
D	1	2	1	4	3
E	2	3	2	3	2

Figure 4: Reachability for a knight on B2 on an open 5x5 board

We used reachability on the current board, rather than pre-computing on an open board. We can see from this that reachability is simply the graph-distance between the knight and each square. It is computed using Disjkstra's algorithm. Using this reachability function, we define a new score function as follows:

$$score(p) \;=\; \sum_{s \,\in\, S} \frac{exits(s)}{2^{reachabilty(p,s)-1}}$$ where S is each square

on the board

This function defines the value of any square as the number of exit nodes it has times the inverse second power of it's distance minus 1. Thus squares that are one move away continue to have their same value as the custom_score function. However squares 2 moves away will count for half as much, and 3 squares away has $\frac{1}{4}$th as much, etc. In practice, the Disjkstra's algorithm was depth-limited to distances of 3 nodes away. Like the first function, the difference of these scores between the players is computed.

Results

In theory, this score function is at least as powerful as the original custom_score function. However, it performed much worse due to it's extra computational effort. While the improved_score function explored and average of 8.0 plies per move in 1000 games, this score metric explored only 5.0 plies. That's not surprising, since this method looks 3 moves ahead (though in single player search, rather than adversarial search). This less depth, resulted in only a 49% win ratio against the improved_score function.

Custom Score 3

(opposed_reachable_diff)

This score function is much like the above (and will ultimately fail for the same reason) but takes an extra step to takes in account opponent interaction a bit more, and accounts better for initiative.

This score function's only difference is that it seeds the Dijkstra's algorithm not just with the player's position, but also the opponents. The player with initiative is seeded first, and is given priority in the Heap over the player w/o initiative for equivalent distance nodes. This means that the player only counts squares that it can reach **before** it's opponent can reach them.

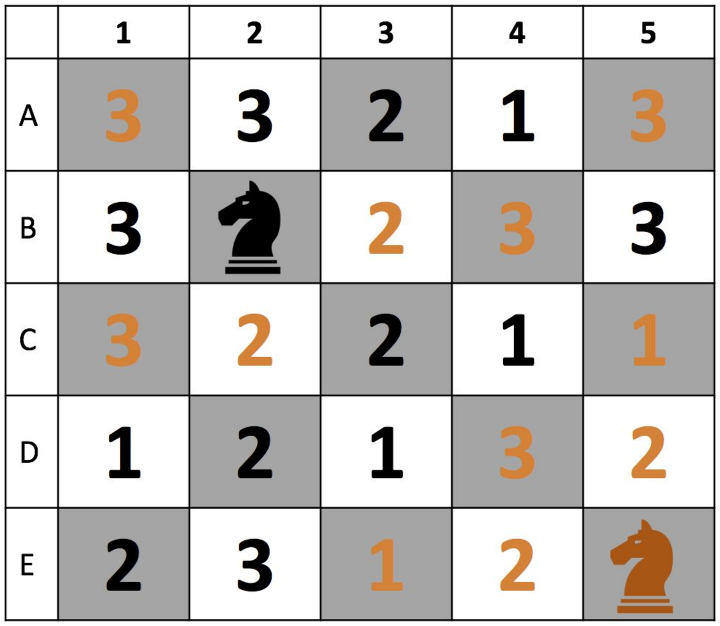


Figure 4: The orange knight has initiative. Orange numbers are the squares that the orange knight can reach first and their distances. Black numbers are the same for the black knight.

This yields two benefits.
1) The player with initiative, though starting from a weaker position, can actually out-play it's opponent because he'll take away some of his opponents second moves. The oppposibility models this better.
2) Since the Disjkstra's algo is seeded with both players, it is equivalent for both players, and needs only be run once rather than twice (once from each player's perspective)

Results

This method failed for the same reason the second did, doing only slightly better. It won 50% of it's games in a 1000 game match against the improved_score. It's earch depth was slightly better (5.3) since the Dijkstra's algorithm needs to only be run once rather than twice.

Summary of results

	Win % vs improved_score	Avg. depth
custom_score	54%	7.62
custom_score_2	49%	5.0
custom_score_3	50%	5.3

Interpretations

Isolation, where players move like knights, does not offer much opportunity for blocking entire sections of the board from the opponent. I suspect that the custom_score_2 and custom_score_3 functions would do much better in traditional isolation, where strategic play is much more important, and determining reachability is vital.

In the Knight variation however, search depth is king since carving out sections of the board for yourself is near impossible. As such, a simple, easy-to-compute score function is going to perform better than more sophisticated score functions that look for such opportunities. The custom_score function both had the highest accuracy and allowed for the deepest level of search. It’s also a marked improvement over the improved_score function because it considers initiative and weights destination squares.