Analysis of heuristics for the knight's isolation game

An analysis by Kris Roosen as requested by Udacity for the 2nd project in the AIND program.

The goal of this document is to analyze several heuristic scoring functions for the game *Isolation* for 2 players where the players move like knights in chess.

During the game, the player search for the best move in a limited amount of time by using the minimax algorithm with alpha beta pruning and iterative deepening. When a leaf node is reached or upon running out of time, nodes are being scored by the implemented heuristic function. A leaf node is a state where the game has ended and the player has either won or lost. Values for these states are defined as +infinity and –infinity, in that order. These states are checked for at the beginning of the scoring cycle in each proposed function below. Since the magic is in scoring non-terminal nodes, the winning or losing state are not further discussed.

The goal of this project is to find the best possible heuristic function for this specific game setup. The proposed functions apply to any rectangular n x m board where there are 2 adversarial players whose pieces move like knights in chess. The proposed function is compared to an identical algorithm where only the heuristic scoring function is different. This is done to rule out differences due to hardware setups etc. The scoring function (for a specific game state) to beat is called *improved_score* and is implemented as:

(number of legal moves for the active player) – (number of legal moves for the opponent)

One might quickly think about partitioning the board and forcing the opponent to end up in the smallest section. This is quite intuitive when a piece is moving like for example a queen in chess, however, this strategy is not intuitive at all for a knight. In my search for the best scoring function, I've tried following strategies:

- Adding weights to the *improved_score* function
- Calculating reach for player and opponent
- Distance to the center of the board
- Length of the player's and opponent's knight's tour
- A combination of these strategies

How to interpret the test results?

Results are obtained by running a tournament between a custom agent running the minimax algorithm with alpha beta pruning and iterative deepening using the i*mproved_score* function and several other agents with different scoring functions. In the second piece of the tournament, the same custom agent is running against the same other agents, but this time using the custom proposed scoring functions. To draw fair conclusions, the percentage of won games of the 2 custom agents should be compared. My custom heuristic scoring functions are considered to be better than *improved_score* if their win percentage is higher. Following is a table of opponent agents with their searching algorithms and scoring functions:

Name	Search algorithm	Scoring function
Random	Selects a random move	None
MM_Null	Regular minimax with iterative deepening	None
MM_Open	Regular minimax with iterative deepening	Number of legal moves for the player
MM_Improved	Regular minimax with iterative deepening	(Number of legal moves for the player) – (number of legal moves for the opponent)
AB_Null	Minimax with alpha beta pruning and iterative deepening	None
AB_Open	Minimax with alpha beta pruning and iterative deepening	Number of legal moves for the player
AB_Improved	Minimax with alpha beta pruning and iterative deepening	(Number of legal moves for the player) – (number of legal moves for the opponent)

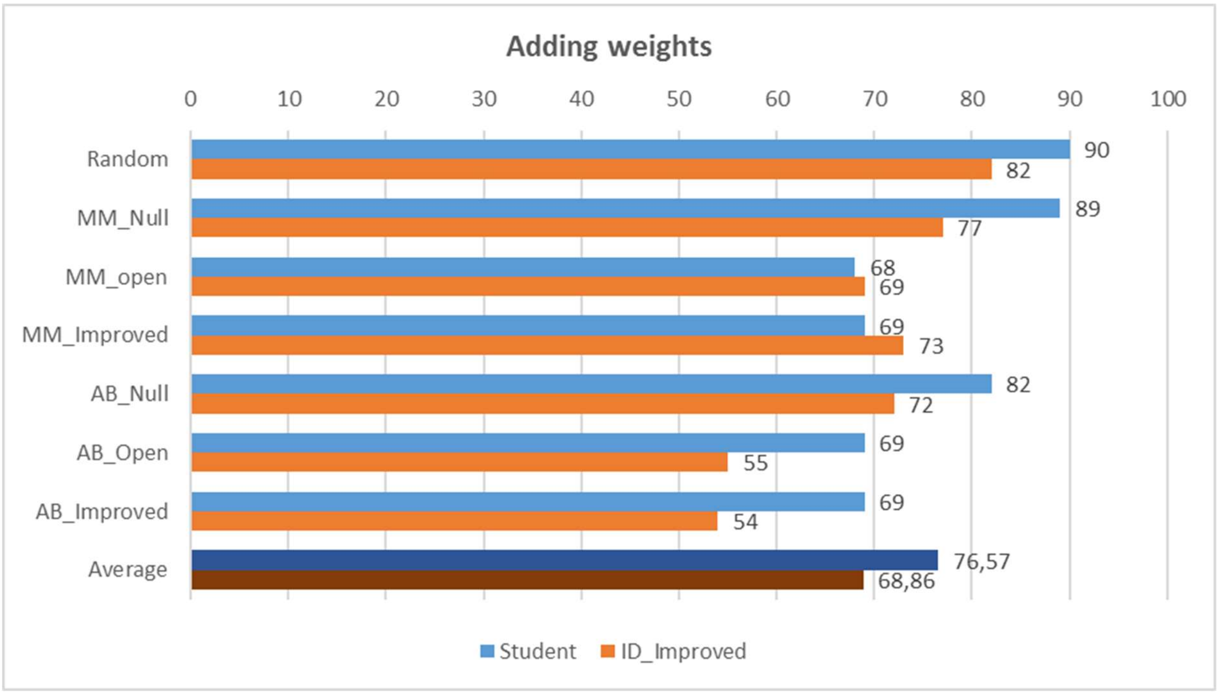
The custom benchmark agent is called 'ID_Improved' and my implementation is called 'Student'. All tests are run so that both agents are running against each opponent for 100 matches. The starting position of each player for each game is set at random. In the presented graphs the number of won matches out of 100 is shown for each agent/heuristic. On the last row, the average is shown.

Adding weights to the improved_score function

The first attempt to beat improved_score was to add more weight to the negative value that holds the number of legal moves for the opponent. This would cause the player to really chase after the opponent and block its moves. The weights are chosen so that the score is:

1 ** (number of legal moves for the active player) –* ***3*** ** (number of legal moves for the opponent)*

The results of running this are:



Conclusion:

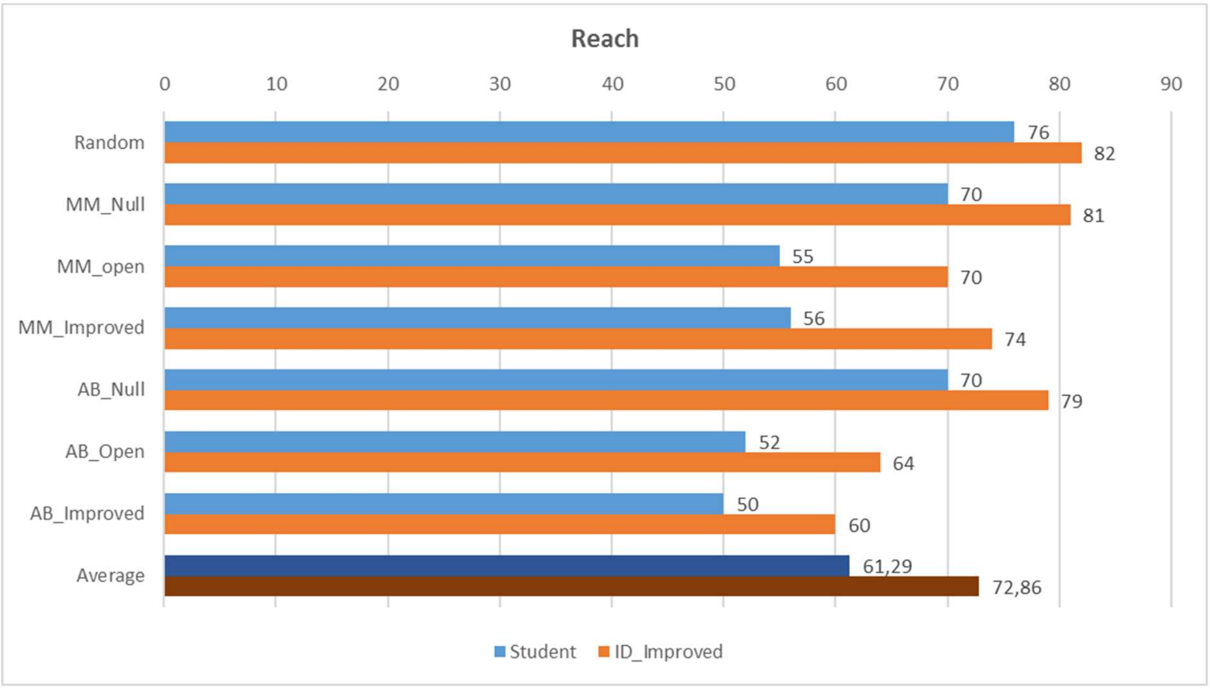
This approach is beating the improved_score function distinctively. However, it's not really a new scoring function, just a tuned version of the supplied function.

Calculating reach for player and opponent

The second attempt at beating the provided function was to map the reach of each player within 2 moves. Taking into account the already blocked positions, having a bigger reach than the opponent results in a higher score. The reach is calculated by counting the legal moves from the current position and the legal moves from each of those positions. Each position can only be counted once. The function is of the form:

1 * reach(player) – **3** * reach(opponent)

The results of running this are:



Conclusion:

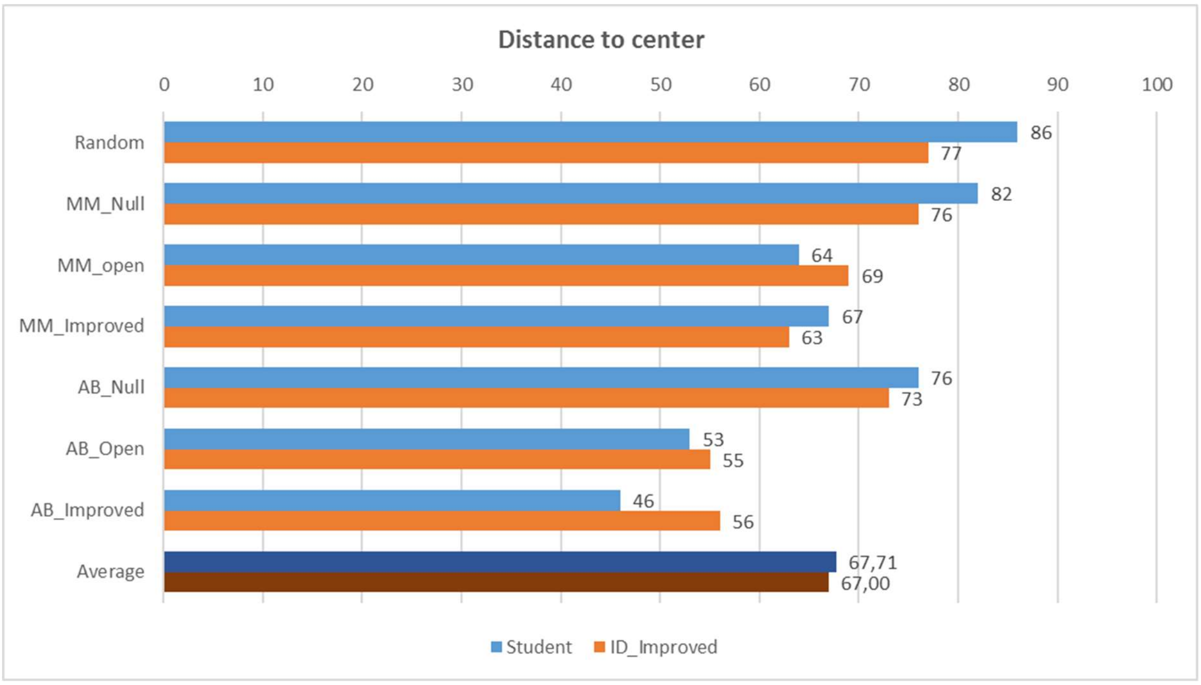
Several weights were tested with this function, but the performance is not good enough to beat the benchmark. One reason could be that for a small board, the defined reach has an insignificant result. The board size for these tests is 7 x 7. Another reason could be that this scoring function is trying to look 2 moves into the future. This is not enough to make a qualitative decision and doesn't take into account the opponent's moves. As the amount of blanc positions decreases, the reach also decreases. This isn't necessarily bad, but since that's not considered in the heuristic the score will still be lower.

Distance to the center of the board

In a 3rd attempt, I have tried to take into account the distance of each player to the center of the board. Around the edges there are less possible moves so it is easier to get stuck. The score consists of:

(Manhattan distance to center of player) – (Manhattan distance to center of opponent)

The results are:



Conclusion:

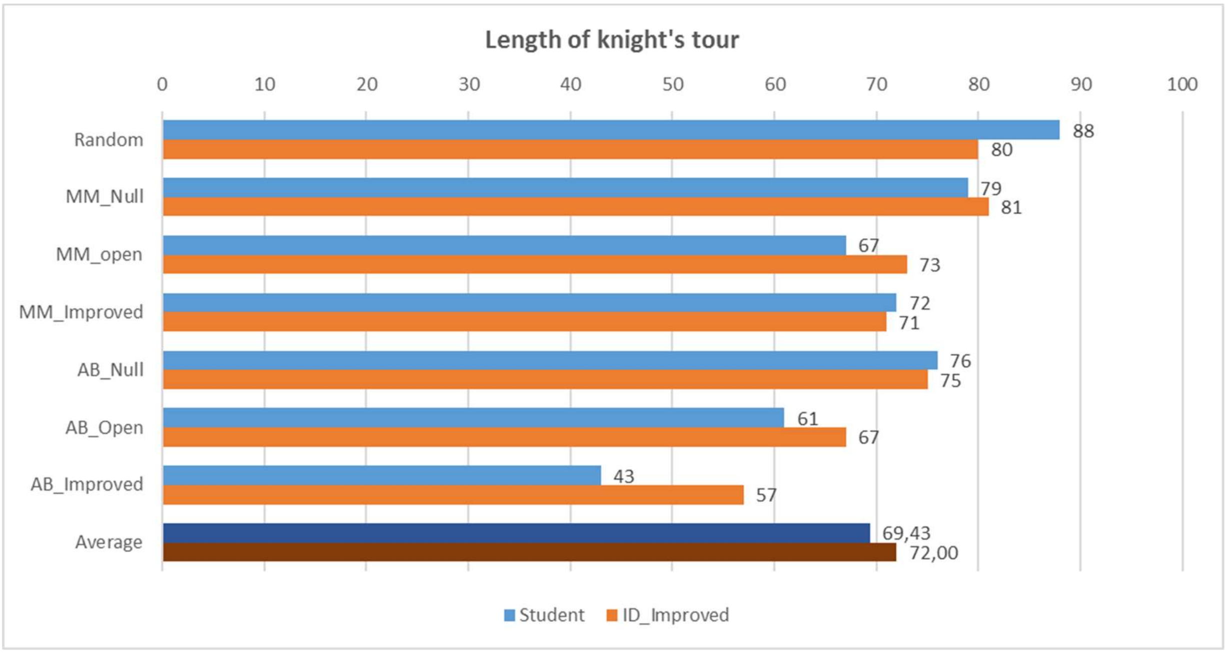
This doesn't seem to be a very good nor very bad scoring function. Being in the center gives you more options. However, since you can only step on each position once, it doesn't make sense to keep hanging around the center of the board. Eventually, as the board gets filled up, the edges of the board might have more possibilities than the center.

Length of the player's and opponent's knight's tour

For the 4th attempt, I went looking for existing heuristics for a knight. These can obviously be found at the game of chess. A knight's tour is defined as the path around a rectangular board where the knight passes each position exactly once. Since this is the longest possible path, it looks very promising. A distinction is made between open and closed tours. I've used the open tour where the knight doesn't need to end in the same position where it has started. In a way, by figuring out this tour for the player and the opponent, this game is being turned into a game of snake, where the goal is to block the opponent's path before you get stuck yourself. The scoring function is of the form:

1 * (length of knight's tour for player) – **3 *** (length of knight's tour for opponent)

The results are:



Conclusion:

For this setup, I expected a better result. The player with the longest path ahead of him, is in a better position. However, to find this longest path, it selects moves that result in positions with very little legal moves. This makes the method very vulnerable to being cornered in.

A combination of these strategies

After trying out the higher mentioned strategies with several weight distributions, I’ve picked the best candidates and combined them into a single scoring function. I ended up using the benchmark function with altered weights together with the knight’s tour function. The reason for this being that during the computation of the knight’s tour, moving to positions with the smallest number of legal moves is preferred. Since this can be a dangerous path to follow, the number of legal moves is brought in to balance this out.

Further improvements

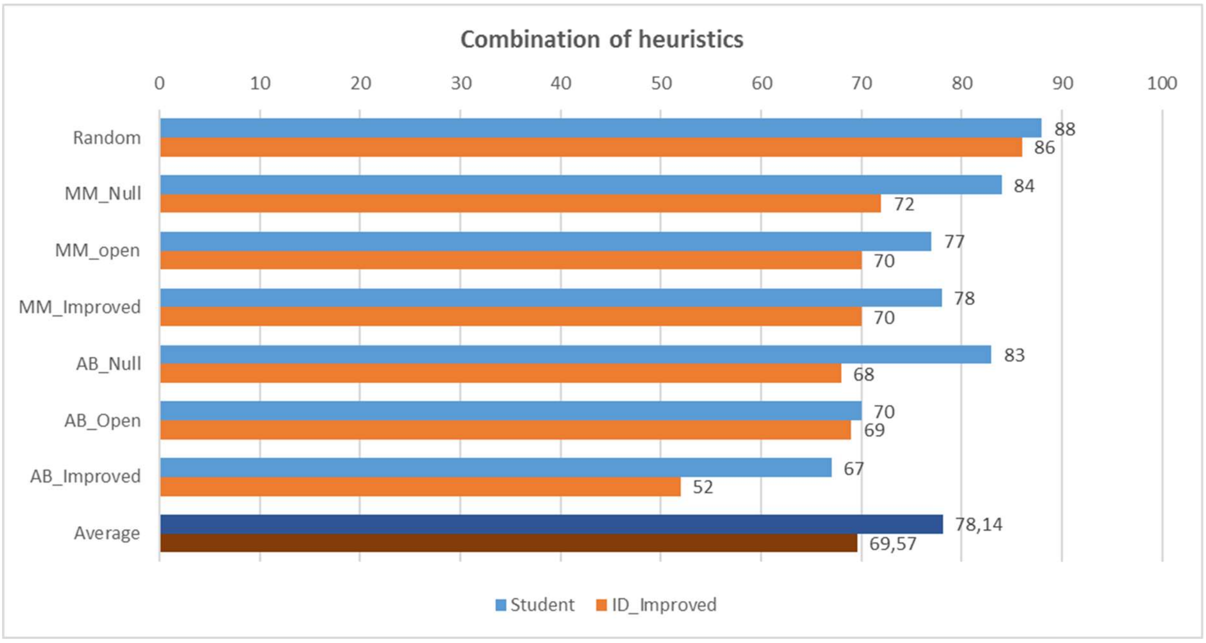
Several other improvements were implemented to the custom scoring function. First of all, the score of each heuristic strategy was normalized to enable the comparison of multiple functions as well as to enable adding weights without having individual scores out of proportion. A second improvement consisted of using different strategies in different stages of the game. Arbitrarily, the game stage is considered early if there are more than 15 empty positions left on the board. After that, the scoring computations are being made lighter so that the search depth can increase and we attempt to search all the way to a terminal node.

For this, a greedy version of the knight’s tour algorithm was coded up, based on Warnsdorff's Rule. Instead of finding the best (longest) knight’s tour, we always move to the position from where the least number of legal moves exist. This gives us a good approximation while requiring less computation.

The final algorithm I ended up using, is of the form:

```
If num_blank_spaces < 15:
        h1 = normalize(1 * num_legal_moves(player)          – 2 * num_legal_moves(opponent))
        h2 = normalize(1 * greedy_knights_tour(player)      – 2 * greedy_knights_tour(opponent))
        return 1 * h1 + 1 * h2
else:
        h1 = normalize(1 * num_legal_moves(player)          – 2 * num_legal_moves(opponent))
        h2 = normalize(1 * thorough_knights_tour(player)    – 2 * thorough _knights_tour(opponent))
        return 1 * h1 + 1 * h2
```

The results are:



In this case, it is particularly interesting to investigate the average and maximum search depth when exploring the game tree. The maximum search depth is measured over 100 matches and then averaged over all opponents that were faced.

	Average search depth	Maximum search depth
Student	5.0	14.00
ID_Improved	7.5	14.27

Conclusion:

From these results, I conclude that it's a very good idea to try to follow the longest possible remaining path, but simultaneously keeping enough options open to be able to escape when the opponent is intersecting the path. Another conclusion is that combining heuristics in a sensible way can achieve better results than the heuristics individually. Especially interesting to see is that the implemented strategy's average search depth is significantly lower than the bench mark but the maximum search depth is almost equal (this is reached in endgame). Taking into account that this implementation vastly outperforms the benchmark, we must conclude that it really pays off to think better about making good choices early on and to go for maximal search depth in the endgame.

Conclusion

From this project I have learned that it's not arbitrary at all to find good heuristics for even a simple game. Heuristics can be found intuitively, however there are many pitfalls and often better heuristics can be found after some detailed investigation.

Combining heuristics is a good way of getting an advantage while blocking their weaknesses mutually. In this study, I recommend using the combined heuristic of the knight's tour with the amount of currently legal moves. Some additional reasons to support this recommendation are:

- This heuristic thinks about the future path and about blocking the opponent's future paths as soon as possible, leaving him only with shorter future paths.
- This heuristic considers it bad to have only a small number of available moves while simultaneously trying to block the opponent's degree of freedom.
- Thinking carefully during the early game results in making better choices that lead to a brighter future. During the endgame however, the goal is to find a winning leaf node as soon as possible and go for that node. Here it is advised to make the scoring computations lighter and put more effort (time) into exploring the game tree as deep as possible.