An evaluation function for a minimax Draughts player

Dennis van der Schagt (0814249)
Rob Wu (0787817)

March 9, 2015

Abstract

This report describes a minimax algorithm for an AI agent that plays International Draughts. This algorithm uses a alpha-beta search algorithm to find a good strategy for playing the game. Alpha-beta search is quite a standard algorithm in the field of Artificial Intelligence; the novelty that is presented in this report is the heuristic evaluation function.

1 The algorithm

1.1 Alpha-Beta search

An implementation of the alpha-beta search algorithm is shown below. Our player uses the alpha-beta search function with iterative deepening: `alphabeta()` is repeatedly invoked with `node`=*current game state*, $\alpha = -\infty$, $\beta = +\infty$ and a positive and a strictly increasing integer (`remainingDepth`) until the allotted time for the turn is used up. When the algorithm terminates, the move that results in a game state with the "best" heuristic value is returned. See section 1.2 for an elaboration of our heuristic value function (this also referred to in the pseudo-code at line 3).

```
   Input: node, remainingDepth, α, β
 1 if a player won in node or remainingDepth = 0 then
 2 |  return the heuristic value of node
 3 if it's white's turn in node then
 4 |  for each child of node do
 5 |  |  α:= maximum of α and alphabeta(child, remainingDepth - 1, α, β)
 6 |  |  if beta ≤ alpha then
 7 |  |  |  return β                                   /* Beta cut-off */
 8 |  return α
 9 else
10 |  for each child of node do
11 |  |  β:= minimum of β and alphabeta(child, remainingDepth - 1, α, β)
12 |  |  if α ≤ β then
13 |  |  |  return α                                  /* Alpha cut-off */
14 |  return β
                    Function alphabeta(node, remainingDepth, α, β)
```

1.2 Heuristic evaluation function

We intentionally kept our evaluation function simple and fast. Keeping it simple makes it easier to understand what is going on and thus easier to improve the magic numbers. Because of the simple evaluation function we also had an easier time fixing a bug we had in our alpha-beta algorithm. Keeping the function fast allows us to look forward as many turns as possible in the given time.

The evaluation function returns a single integer value which reflects the state of the game. A negative value means that black is likely going to win, whereas a positive value means that white will probably win. When the game state is such that one of the players is certainly going to win, the returned value is an extreme value. E.g. if black will certainly win from white, then the returned value is the maximum representable negative integer.

If it is not set in stone which player will win, then the score is calculated from the number of pieces on the board and their position. We assume that the number of pieces is the most important indicator of success, so this feature is the primary score component. To make sure that this primary choice is always respected irrespective of the position scores, the scores are chosen such that the sum of the position scores can never exceed the smallest unit of the piece count score.

The piece score modifier is used to value the pieces that are on the board. For this part we only look at the kind of piece, white or black, uncrowned or king. We give a value of 1000 to a white uncrowned piece. On advice of dr.ir. J.W. Wesselink we decided to value kings thrice as high as an uncrowned piece, so a white king is worth 3000. This makes sure our player will work toward getting a king without giving up too many uncrowned pieces for that goal. The values of the black pieces are the opposite of the values of the white pieces, as we want to have a symmetric evaluation function.

The position score modifier associates a value to a piece depending on their row position. The closer a piece is to the other side of the board, the higher its bonus score. This criteria exists to encourage reaching the other end in order to get a king, even when the AI agent cannot look that deep in the game tree. Because this criteria only exists to stimulate becoming a king, the bonus is only added to uncrowned pieces. Note that the piece count takes precedence over position, so the agent will not unconditionally move pieces forwards in an attempt to get a piece to the other end, because this action would negatively affect the piece count (i.e. blindly moving forward allows the opponent to kill a piece).

To avoid moving too many pieces to the other side of the board, the first row will get a higher bonus score than the first few rows, to encourage pieces to stay home. This allows them to prevent opposing pieces from being a king.

A way to implement this bonus system is by awarding one bonus point to a piece for moving one row forwards. A disadvantage of this mechanism is that it does not differentiate between a piece that is already close to the end, and a piece that is still at first rows. To distinguish between both situations, the list of bonus scores for each row is a Fibonacci sequence. The final list of bonus scores is shown in table 1. There are at most 20 pieces per player. The maximum score for a piece is 31, so the maximum bonus is not higher than $20 \times 31 = 620$, which is well below 1000 (the piece count score for an uncrowned piece).

row	score	comment
1	10	Staying home is preferable to moving one forward, in order to defend against incoming pieces from the opponent.
2	1	
3	2	= more than 1
4	3	= 1 + 2
5	5	= 2 + 3
6	8	...
7	13	
8	21	
9	31	
10	3000	A piece at the end is already a king. There's no need to associate a score with a piece at the last row, because this case does not occur. We fill in the table with score 3000 to show that this piece is a king.

Table 1: Bonus scores for piece position.

2 Results

Our player did fairly well in the tournament, especially considering the fact that the implementation of our evaluation function contained an error: We accidentally flipped the table of the bonus scores (table 1). Consequently, the intended effect of the position score was not applied (worse: the pieces at the first row were very sticky due to the incorrect bonus score of 3000). Because of that we lost one match of the group phase and consequently we did not qualify for the quarter finals. After finding and fixing the error we replayed all the matches, playing on the white side, as well as the black side. This time we won all those matches. Then we continued testing by playing against the winner of the tournament (player 42/Poro_Plank). We lose consistently when playing on the black side but when playing as white most of our games result in a win [1] or a draw [2].

Our player plays relatively well for the limited rules we use in the evaluation function. The number of moves that we can look ahead probably plays a more significant role. Throughout the game our player can look between 7 and 9 moves ahead (using 2 seconds each move). We did not optimize much in the alpha-beta search. If we would ever want to improve our player we could start by ordering the moves. In that way we could probably prune more of the tree and consequently look further ahead in the more interesting parts of the search tree.

3 Reproducing the results

Following are the steps required to reproduce our results:

- Download the tournament NetBeans project from `http://www.win.tue.nl/~wstahw/edu/2ID90/tournament-2015.zip` and unzip it
- Download our player plugin's NetBeans project from Peach and build it
- Copy the jar file from our player project to the plugins folder of the tournament project, thereby replacing our old player
- Run the tournament project
- Select our player (AlphaBetaPlayer) and one of the other players
- Run the competition

Keep in mind that results can vary with the time given per move. We ran all test games with 2 seconds per move.

4 References

Since we are no Draught experts at all, we read external sources such as `http://www.cs.columbia.edu/~devans/TIC/AB.html` to validate our approach. The take-away from that article is that the piece count is important, as well as the proximity of getting a king. This concept is similar to our implementation, which encouraged us to continue in this direction.

Since the alpha-beta search algorithm is quite standard, we looked up pseudo-code from `http://stackoverflow.com/a/15626976` and converted that to Java.

5 Contributions

Dennis and Rob contributed equally to this assignment, during the construction of the evaluation function, implementation of the program and writing this report.

6 Appendix

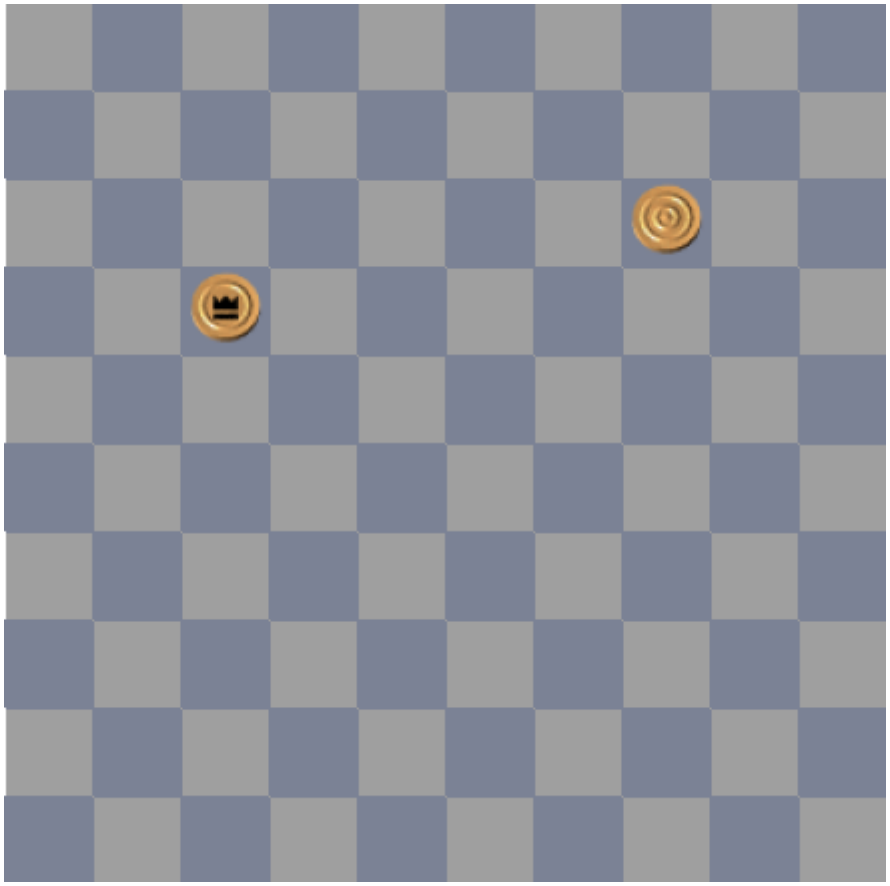

Figure 1: Final result of a game in which we played as white against the winner of the tournament.

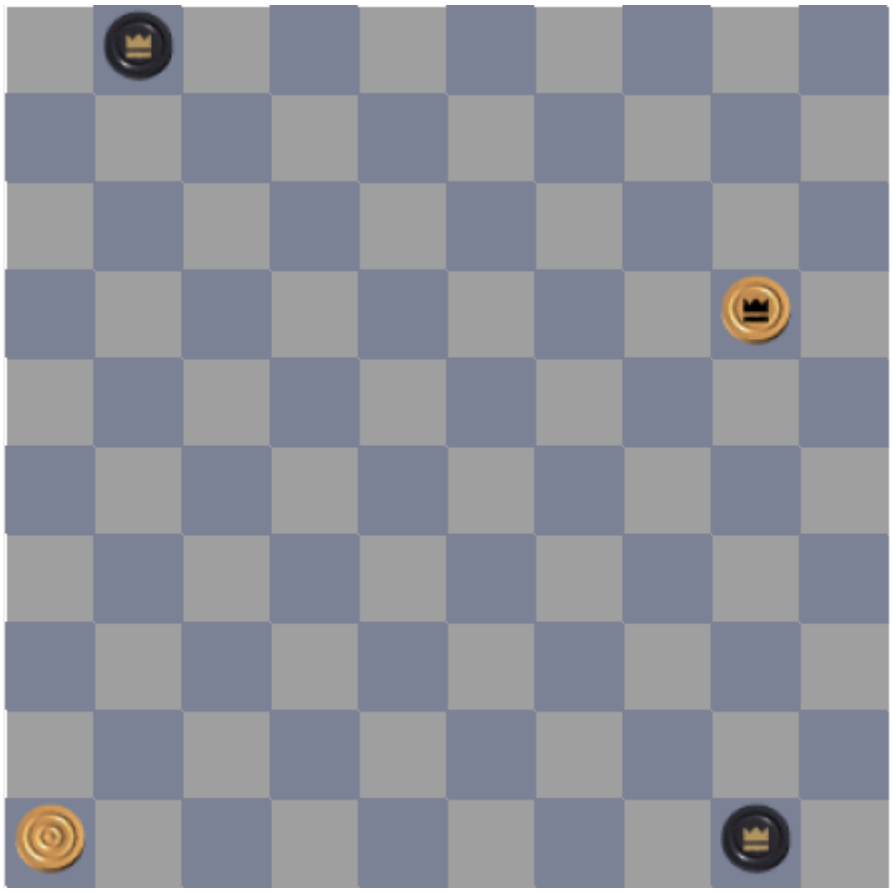

Figure 2: One of the many draws with the winner of the tournament.