Playing KnightThrough with Alpha-Beta

Fabian Braun, I6098238

January 24, 2015

Department of Knowledge Engineering
Maastricht University
Master's program: Artificial Intelligence
Course: Intelligent Search & Games
Teachers: Dr. Jos Uiterwijk, Dr. Mark Winands

Contents

1 Introduction **3**
1.1 KnightThrough . . . 3
1.2 Final Tournament . . . 4

2 Program Design **4**

3 Evaluation Function **5**
3.1 Evaluation Function ”Piece Count” . . . 5
3.2 Evaluation Function ”Leader Position” . . . 5
3.3 Evaluation Function ”Structure Quality” . . . 5
3.4 Evaluation Function ”Development” . . . 6
3.5 Evaluation Function ”ZickZack” . . . 7

4 Alpha-Beta **8**
4.1 Move ordering in version 5 and further . . . 9
4.2 Forward pruning alias ”Threatening move detection” . . . 10
4.3 Aggressive Forward Pruning in Version 7 . . . 10

5 Time Control **10**

6 Conclusion **12**

7 Appendix **13**
7.1 README file of developed program . . . 13

References **13**

1 Introduction

This report describes a bot program developed to play the game KnightThrough (Winands & Uiterwijk, n.d.-a). The development of this bot is the practical part of the course "Intelligent Search & Games" hold by Dr. Winands and Dr. Uiterwijk in fall 2014 at the Department of Knowledge Engineering, University Maastricht.

1.1 KnightThrough

KnightThrough is a zero-sum board game of perfect information. This means that at any time both players know the complete current game-state. Zero-sum expresses that one player's win means the loss of the other player. Furthermore the game is always finite and there are no draws possible. KnightThrough is played on an 8x8-board. Each player controls 16 pieces.

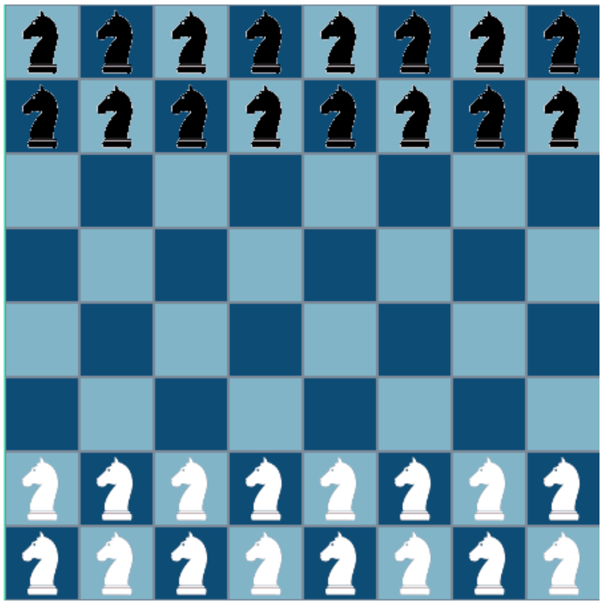

Figure 1: Initial board position

The goal of the game consists of reaching the opponent's ground line with one piece. The first player who achieves this goal wins. A move consists of moving a piece in the manner of a knight in chess. In contrast to chess it is not allowed to move a piece backwards. Otherwise the game would not be finite.
It is a course requirement that the bot is based on the alpha-beta-tree-search-algorithm.

1.2 Final Tournament

During the course a final tournament was announced, on which all student's bots should play KnightThrough against each other to determine the best bot. For the final tournament a time limit was introduced. The time limit is 15 minutes per player and match (Winands & Uiterwijk, n.d.-b). In conclusion a match can not last longer than 30 minutes.

2 Program Design

The bot is developed in the Java programming language. Basic tests have been implemented using JUnit. The graphical user interface (GUI) is based on Swing. To be able to measure the performance of different bots the engine is designed in such a way that playing bots can easily be changed over a configuration-file. Before the final tournament the following 10 different bots have been implemented:

- GUI player
- Random player
- Greedy player with configurable evaluation function
- Seven different Alpha-beta players with configurable evaluation function

For installation and execution instructions please see the attached README-file.

3 Evaluation Function

The alpha-beta-bots and also the greedy-bot allow a change of the evaluation function via configuration. The following evaluation functions have been developed:

3.1 Evaluation Function "Piece Count"

This basic evaluation function rates a board according to the following formula:
$rating = count(myPieces) - count(opponentsPieces)$
Although this evaluation is not very accurate it can be calculated very fast. This is a huge advantage as the evaluation method is executed on all investigated leaf nodes. All other evaluation functions use this feature as it is a meaningful statement over the quality of the current position.

3.2 Evaluation Function "Leader Position"

This evaluation function takes into account, how far the leading piece is developed. Development refers here to the index of the row of the leading piece. It is calculated from the point of view of the current player. In figure 2 the value for the white player is 3, as it's leading piece is in the 3rd row from white's viewpoint. The value for black is 5.

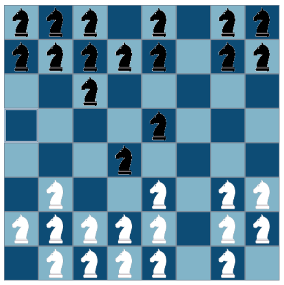
Figure 2: example position

The evaluation function does not take into account, when a leading piece is captured. This disadvantage has been accepted to increase the evaluation's velocity.

3.3 Evaluation Function "Structure Quality"

This evaluation function is computationally expensive. It evaluates how good the pieces of each player are working together. The evaluation function gives a high rating, when many pieces protect/attack the same square. The purpose of this evaluation function consists of creation a piece structure that is very robust against attacks from the opponent. A square which is attacked by several pieces is a safe position to move to.

The calculation is done as follows: First for each square is determined, how many pieces can move to this square. This number is saved. Figure 3 shows such a rating for the black player.

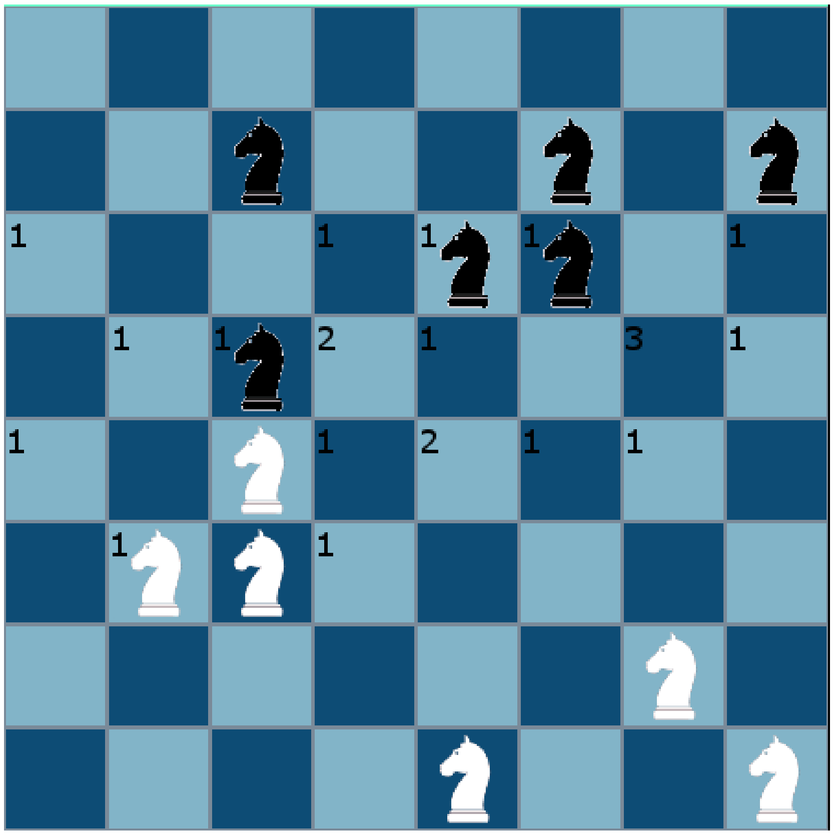

Figure 3: structure quality for black player

After the analysis of all squares the total score is determined. All squares are rated according to $count(attackingPieces)^2$. The sum of these values is the evaluation of the pieces of one player.
The rating for the black pieces in figure 3 accordingly would be: $(1+1+1+1+1+1+1+2^2+1+3^2+1+1+1+2^2+1+1+1+1) = 32$ The final evaluation value is determined by the difference of the ratings for each player in addition to the piece count value weighted with factor 1000.

3.4 Evaluation Function "Development"

The evaluation function "Development" tries to always develop the piece which is left most behind. The idea behind it consists of creating a defensive player that gains an advantage in the endgame because all pieces are developed. To achieve this the function keeps track on the least developed piece and evaluates a position better, when the development of the last pieces is better. This function is also combined with the piece-count-feature.

3.5 Evaluation Function "ZickZack"

This evaluation function was designed after some observations of games by the so far created bots. In these games one position turned out to be very strong and a compulsory loss for the opponent:

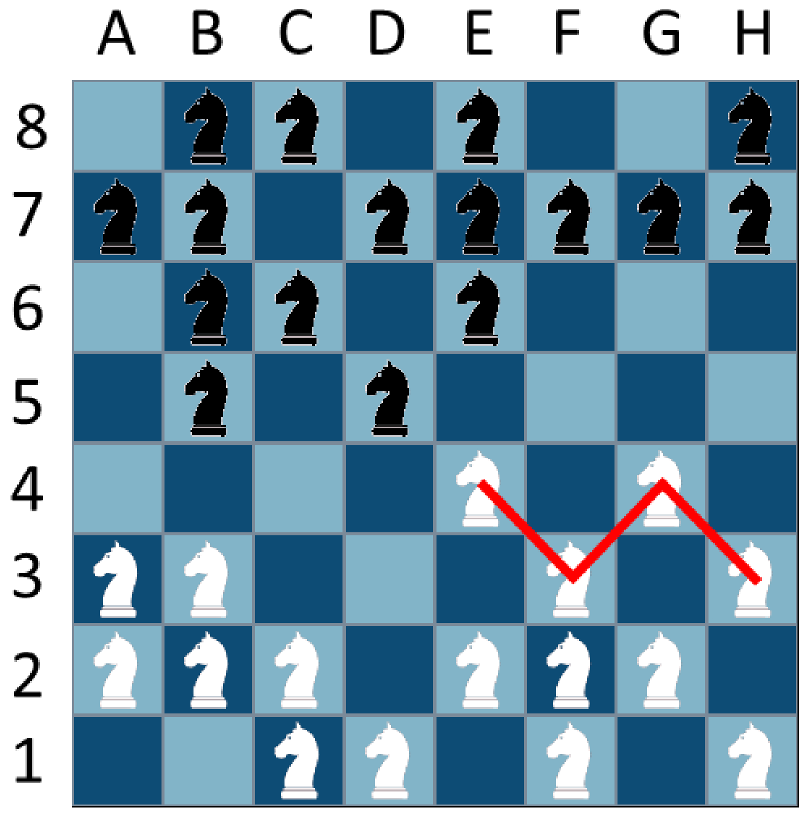

Figure 4: white to move - compulsory loss for black in 15 ply

The evaluation function was named "ZickZack" because of the structure of white's pieces on the right flank. The winning sequence is here:

1. white: G4 → H6 (sacrifice)
2. black: F7 → H6 (forced capture move) this move leads to G5 being less protected by black which finally is the reason why white wins
3. white: H3 → G5

4. black: H7 → G5 or E6 → G5 (forced)
5. white: E4 → G5
6. black: H7 → G5 or E6 → G5 (forced)
7. white: F3 → G5 threatening G5 → H7 at this point black has no piece to capture on G5 and the position is obviously lost. black can still postpone the loss.
8. black: B5 → A3
9. white: C2 → E3 (forced)
10. black: D5 → C3
11. white: D1 → C3 (forced)
12. black: any move, e.g. B6 → A4
13. white: G5 → H7
14. black: any move, e.g. A4 → B2
15. white: H7 → F8 (win)

The goal of evaluation function "ZickZack" consists of reaching this winning position. Therefore a position which contains the ZickZack-structure of pieces is assigned a high evaluation. Also parts of the ZickZack-structure already give a good rating. Unfortunately the winning sequence from above relies on the fact that the opponent does not have a piece on G8. Otherwise black can simply capture G8 → H6 instead of F7 → H6 and the situation is comfortable for black. Still it is not obvious that G8 → H6 is a better move than F7 → H6 in this position, so the strategy might still succeed. As the other evaluation functions this one also takes into account the piece count for each player as a dominating feature.

4 Alpha-Beta

The alpha-beta-tree-search algorithm is implemented relying on (Winands & Uiterwijk, n.d.-c) and (Winands & Uiterwijk, n.d.-d). As mentioned in chapter 2 seven different alpha-beta bots were implemented:

1. Static depth 5; no enhancements
2. Iterative deepening without estimation for next depth time; (slow) move ordering: capture moves first
3. Iterative deepening with basic breaking condition; (fast) move ordering: capture moves first; transposition tables

4. As 3. . Enhanced iterative deepening with breaking condition based on accurate time estimation; threatening move detection (forward pruning) in root

5. As 4. but with move ordering based on transposition table of previous depth

6. As 5. but threatening move detection (forward pruning) here at all tree nodes, not only on root

7. As 6. but with aggressive forward pruning

In the next chapters the enhancements are described in detail. Well known terms as iterative deepening and transposition tables are not explained further. This report focuses on the special enhancements which are applied for the specific KnightThrough-game.

4.1 Move ordering in version 5 and further

The move ordering is a combination of taking into account the transposition table (TT) of the previous depth and also if a move is a capture move.
The moves are ordered in the following manner: All capture moves and good moves according to the TT get an early spot in the list, whereas the other moves are positioned in the later part of the List.
To achieve a good performance of sorting the moves, the sorting algorithm is implemented in such a way that it does not always give the correct order. By this it is possible to shrink the complexity of the sorting to O(n). Each move is only ranked once. The incorrect order is not a disadvantage as an approximate ordering can already produce many prunings.
Pseudocode of the sorting algorithm:

```
sort(List unsortedMoves): sortedMoves
sortedMoves = new empty List
for all move: unsortedMoves do
  if move saved in TT then
    if saved evaluation value > half a piece advantage then
      add move to first position of sortedMoves
    else
      if saved evaluation value > 0 then
        add move to the end of the first third of sortedMoves
      else
        add move to last position of sortedMoves
      end if
    end if
  else
    if move is capture move then
      add move to first position of sortedMoves
    else
```

```
            add move to last position of sortedMoves
          end if
        end if
      end for
      return sortedMoves
```

4.2 Forward pruning alias "Threatening move detection"

To reduce the size of the search tree threatening moves are detected. "Threatening move" refers to one player being able to win in 1 ply. This occurs when one or more pieces of the player are located one or two rows from the ground-line of the opponent. In such a situation the only moves making sense are capturing the threatening piece. All other moves can be pruned immediately.
In version 4 and 5 this detection is only applied in the root. If there is only one move to capture the threatening piece, this move is performed immediately without considering the search tree. It is important to note that the move is then played in very short time. This saved time can be consumed for later moves.
In version 6 and 7 "Threatening move detection" is performed on all depths. This mechanism led to 2 to 3 times less nodes investigated for each move.

4.3 Aggressive Forward Pruning in Version 7

Version 7 is a try to generalize the pruning of version 6 to prune even more. The idea is here to only evaluate capture moves each time the opponent has a piece advantage.
This raises the pruning factor much more than in version 6. Version 7 has about 2 times less nodes to investigate than version 6. In some extreme cases version 7 searches even 9 times less nodes.
Unfortunately the pruning leads to bad play in the endgame. If the other player sacrifices a piece to be able to win later on, the v7 player only investigates capture moves by the opponent from that point onwards. It happens regularly that due to this the winning move for the opponent is not found.

5 Time Control

The final tournament in which the different bots should compete had a time limit of 15 minutes per player and per match. To distribute these 15 minutes optimally over the match a timing control is implemented in the alpha-beta-bots from version 4 onwards. This timing control takes into account the following aspects:

- an average game lasts about 14 turns or 28 ply.
- in the opening it is not worth spending a lot of time, as the first moves are not of great importance.

- when a win is detected during the endgame, the needed time decreases rapidly during the play-out of the winning line.
- when a loss is detected during the endgame, it is not important to spend a lot of time anymore.
- usually a winner is detected around ply 24

These observations led to the timing design shown in table 1.

turn number	spend % of remaining time
1	0.2
2	2.0
3	3.9
4	5.9
5	11.7
6	17.6
7	18.6
8	19.6
9	20.5
10..n	33.3

Table 1: time control

It is important to note that the percentage always refers to the *remaining* time. Because of that the needed time will never exceed the limit of 15 minutes. The search will always be interrupted when the time for the current move is over and the best result of the previous depth iteration is returned.
Additionally to the precalculated estimate per move the bots are able to decide dynamically if they try to reach the next iterative depth within the remaining time. E.g. if a bot has already consumed more than 50% of the time for a move after finishing the alpha-beta-tree-search on depth 8, then it does not make sense to start to search till depth 9 anymore. This search will probably not be finished in time.
The developed alpha-beta-bots from version 4 onwards estimate the time they are going to need for the next depth. For this the measured times of the previous two depths are taken as a basis to determine the time growth factor. With this growth factor the expected time for the next depth is calculated. When the time remaining is less than this estimate, the iterative deepening is cancelled.
During the tournament this approach gave good results. Figure 5 shows how the total consumed time evolved over the turns. The figure only shows games which lasted at least 24 ply.

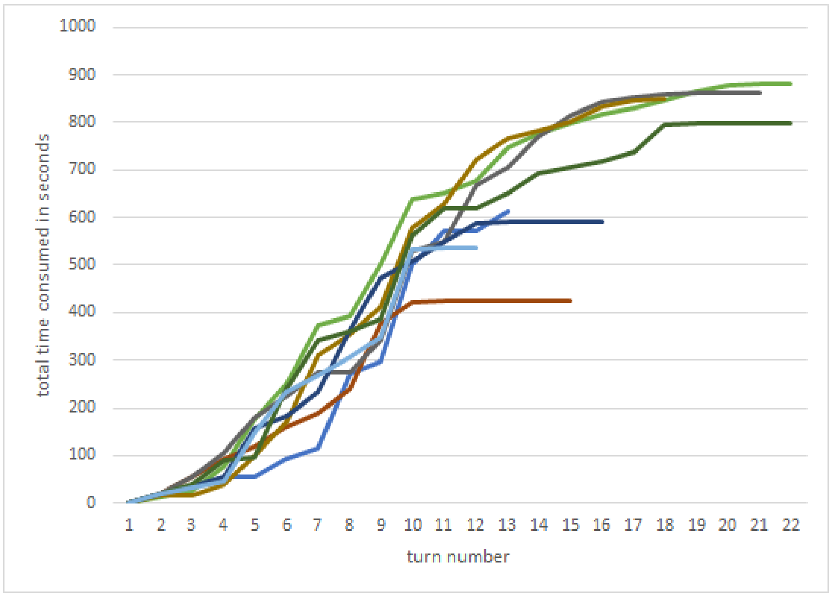

Figure 5: total time consumed. Each curve represents a different match

6 Conclusion

The bot "Alpha-Beta v6" turned out to give best results of the developed bots. This is not surprising as it contains the best version of all implemented enhancements. Only bot v7 has additional pruning compared to v6 but this enhancement led to a worse performance in the endgame as described above.
Regarding the evaluation functions it turned out that "Development" gave the best results. It ruled out other fast evaluation functions as "Piece Count" and "Leader Position" by gaining a better position till the endgame. "Structure Quality" was too expensive compared to "Development" which leads to a difference of 2 regarding the search depth. Therefore "Development" dominates against "Structure Quality". Evaluation Function "ZickZack" is able to win against "Development", but "Development" shows a more predictable behaviour against other evaluation functions. For this reason "Alpha-Beta v6" and evaluation function "Development" were chosen for the final tournament.
The bot achieved the second rank among 16 bots in the final tournament. It was beaten by a bot which was able to search 1-2 ply deeper due to operating-system-native compilation and highly optimised data structures. It also made use of a different evaluation function which led to less pieces being developed. In the end game it turned out to be better having few pieces developed disproportionately high than having all pieces developed equally.

7 Appendix

7.1 README file of developed program

```
########## AUTHOR ###########

Fabian Braun
21st of December 2014

####### INSTALLATION ########

To run this program a Java Runtime Environment in version 7 is required.
Please extract the .zip-archive.
It contains the following folder structure:
- randomTheLucky (parent folder)
    \- run.bat (shell script for execution on Windows OS)
    \- randomTheLucky.jar (runnable .jar-file, no program arguments required)
    \- save (directory for saved games, empty initially)
    \- resources (directory for configuration file and graphics)
        \- config.properties (configuration for the program)
        \- icon_up.png (image representing the player playing upwards)
        \- icon_down.png (image representing the player playing downwards)

## CONFIGURATION and START ##

To start the program run the .bat file.
The configuration can be done over the file config.properties.
The following values can be modified:
    - The starting player (upwards or downwards)
    - A file containing a saved game to be continued
    - Whether a game should be saved after each move
    - Whether the GUI should be shown also when to automatic bots play against each other
    - The client/bot to be used (per player)
    - The evaluation function to be used (per player)
    - The total time that a player may spend during the game (per player)
For more instructions see the configuration file.
```

References

Winands, D. M., & Uiterwijk, D. J. (n.d.-a). *Knightthrough project isg 2014.* lecture slides.

Winands, D. M., & Uiterwijk, D. J. (n.d.-b). *Knightthrough tournament.* lecture slides.

Winands, D. M., & Uiterwijk, D. J. (n.d.-c). *Lecture 2: Alpha-beta search.* lecture slides.

Winands, D. M., & Uiterwijk, D. J. (n.d.-d). *Lecture 3: Transposition tables and endgame databases.* lecture slides.