Intelligent Search & Games: KnightThrough Agent

Dennis Soemers, Student ID: I6037052

December 19, 2014

Maastricht University
Department of Knowledge Engineering
Maastricht, Netherlands
Course: Intelligent Search & Games
Course Tutors: Dr. Jos Uiterwijk & Dr. Mark Winands

1 Introduction

This report describes the implementation of an intelligent game-playing agent for the game of KnightThrough. This section first provides a brief explanation of the game's rules, and then an overview of the remainder of the report.

1.1 KnightThrough

KnightThrough is a variant of the game of BreakThrough (Kerry, 2001), where players control Knights instead of Pawns. It is a deterministic, perfect information, two-player game played on an 8×8 board, in which one player (the "White" player) controls White Knights, and the other player (the "Black" player) controls Black Knights. Each player starts the game with 16 Knights, with the initial state depicted in Figure 1.

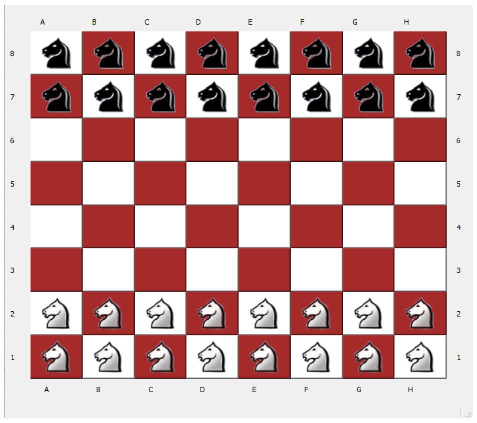

Figure 1: Initial Game State of KnightThrough

Knights are allowed to make similar moves as in Chess, but are restricted to those movements that take them closer towards the opponent's initial position. This means that White Knights must move up, and Black Knights must move down. Knights may not jump off the board, and they may not jump to squares already occupied by a piece of the same color. When a Knight lands on a square occupied by a piece of the opposing color, that piece is captured, again in the same way as in Chess.

A game of KnightThrough ends when one player no longer has any pieces (the player that still has pieces being the winner), or when one player has a Knight on the furthest row from that player's starting position (in which case that player wins). So the goal of the White player is to reach the top row with at least one Knight, and the goal of the Black player is to reach the bottom row with at least one Knight. Given these rules, the game is always guaranteed to end in a finite number of moves, and can therefore be labelled as a combinatorial game.

1.2 Overview

Section 2 provides details on the tools used for the implementation of a KnightThrough-playing agent, and assumptions that are made concerning the implementation in the remainder of the report. Section 3 discusses the representation of Game States and Moves, and the move generation of the agent. In Section 4, the search algorithm used by the agent is described. Section 5 explains the agent's evaluation function. In Section 6, two enhancements to the search engine are discussed. Experiments that were performed during development are described in Section 7. Finally, Section 8 concludes the report and provides ideas for future work.

2 Implementation Details

An intelligent agent named Ser Prunes-A-Lot (simply referred to as "the agent" in the remainder of this report) has been developed to play the game of KnightThrough. The agent has been written in C++ and compiled using the Visual C++ 2013 compiler, targeted for the x86 architecture. Version 5.3 of the Qt framework has been used for the development of the agent's GUI. The agent has only been tested on the Windows 7 operating system.

The agent as described in the remainder of this report uses the code from the "AspirationSearch.h" and "AspirationSearch.cpp" files in the attached source code for the AI engine. Other AI engines are included too, but they are all older implementations with fewer enhancements. These engines have no features that the AspirationSearch engine does not have, and are therefore not further described in this report.

Whenever a variable is described as having the integer type, it can be assumed to be a 32-bit integer (which is the default size of the "int" type with the used compiler settings). When a bigger or smaller size for integral types is used, this is specifically noted. Whenever the report mentions a collection of variables, the used data structure is the *std::vector* from the Standard Template Library, unless otherwise noted.

3 Game State & Moves

This section describes how Game States and Moves are represented in the agent's implementation. It also describes how, using these representations, the collection of legal moves is generated for any given game state.

3.1 Game State Representation

The agent defines a game state using five variables:

- *blackBitboard*: A 64-bit integer, where each of the 64 bits is uniquely mapped to one of the 64 squares on the game board. Any bit that is *set* (meaning that this bit is a 1, as opposed to a 0), means that the Black player has a Knight on the corresponding square. This representation is known as a *bitboard*.
- *whiteBitboard*: An equivalent variable to *blackBitboard* for the White player instead of the Black player.
- *numBlackKnights*: An integer representing the number of Knights that the Black player has. This number could be deduced from *blackBitboard* by scanning it for the number of set bits, but the choice was made to also keep this counter separately for more efficient access to the number of Knights that a player has.
- *numWhiteKnights*: An equivalent variable to *numBlackKnights* for the White player instead of the Black player.
- *currentPlayer*: A flag indicating which player is the player to move.

3.2 Move Representation

A move is defined by the following three variables:

- *from*: An integer denoting the square that the move moves away from.
- *to*: An integer denoting the square that the move moves towards.
- *captured*: A boolean variable indicating whether or not a move is a capture move.

The *from* and *to* variables should always be in the range $[0, 63]$. The *captured* flag is required to be able to correctly undo moves.

3.3 Move Generation

Upon initialization of the agent, for each of the two players, a table containing 64 entries is computed, where each entry contains a small collection (of at most 4 elements) containing the legal move targets for a certain starting square. The index used to access an entry in this table corresponds to the *from* square of a move, and each integer in the collection retrieved from the table corresponds to a legal *to* square.

The bitboard of the player to move can be scanned for the set bits to determine on which squares that player has Knights. These squares can then be used as index for the table of precomputed moves to obtain collections of legal move targets. Numbers with only the single bit corresponding to those move targets set can then be compared to the player's own bitboard to determine whether the moves are legal, and they can be compared to the opponent's bitboard to determine whether the moves are capture moves. Scanning for set bits in this procedure is done using *De Bruijn Multiplication* (*chessprogramming - BitScan*, n.d.).

The agent's move generator ensures that all capture moves are ordered before non-capture moves. All capture moves are ordered in such a way that capture moves farther away from that player's initial rows are considered first, and all non-capture moves are also ordered among each other to prefer moves closer towards the goal.

4 Search Algorithm

This section describes the search algorithm that the agent uses to search the game tree for the next move to play in any given game state.

4.1 Alpha-Beta Search

The core of the search algorithm is the Alpha-Beta Search algorithm. Pseudocode of this algorithm is given in Algorithm 1. When the agent is required to

Algorithm 1 Alpha-Beta Search

```
Require: Current game state $s$, $depth$, $\alpha$, $\beta$
 1: if $s$ is terminal or $depth = 0$ then
 2:     return $evaluate(s)$
 3: end if
 4: $score \leftarrow -\infty$
 5: for $m \in s.generateMoves()$ do
 6:     $s.applyMove(m)$
 7:     $value \leftarrow -$Alpha-Beta Search($s$, $depth - 1$, $-\beta$, $-\alpha$)
 8:     $s.undoMove(m)$
 9:     if $value > score$ then
10:         $score \leftarrow value$
11:     end if
12:     if $score > \alpha$ then
13:         $\alpha \leftarrow score$
14:     end if
15:     if $score \geq \beta$ then
16:         break
17:     end if
18: end for
19: return $score$
```

make a move, first an adapted version of Algorithm 1 is called that also keeps track of the move with the highest *score*, and returns the best move instead of returning a number. In the basic Alpha-Beta framework, this function is called with *depth* = the desired maximum search depth, $\alpha = -\infty$, and $beta = \infty$. The *evaluate*(*state*) function used in line 2 of Algorithm 1 is a heuristic evaluation function that is further described in Section 5.

4.2 Iterative Deepening

The basic Alpha-Beta Search algorithm as described above requires a maximum search depth to be provided. To avoid spending too little or too much time in any given turn, *Iterative Deepening* is used to determine the search depth. The idea of Iterative Deepening is to start with a search depth of 1, then start a new search with a search depth of 2, and continue in that way for a given amount of time. See Algorithm 2 for the pseudocode corresponding to this idea.

Algorithm 2 Iterative Deepening

```
Require: Current game state $s$
 1: $moveToPlay \leftarrow$ null
 2: $depth \leftarrow 0$
 3: $M \leftarrow s.generateMoves()$
 4: $score \leftarrow -\infty$
 5: while $timeAvailable()$ do
 6:    $depth \leftarrow depth + 1$
 7:    $(bestMove, score) \leftarrow startAlphaBeta(s, depth, -\infty, \infty)$
 8:    if $timeAvailable()$ then
 9:       if $score = win$ then
10:          return $bestMove$
11:       else if $score = loss$ then
12:          return $moveToPlay$
13:       else
14:          $moveToPlay \leftarrow bestMove$
15:       end if
16:       $M.sort()$
17:    end if
18: end while
19: return $moveToPlay$
```

In this pseudocode, it is assumed that the call to *startAlphaBeta*() in line 7 returns both the best move according to the search to the given depth, and the score corresponding to that move. In the actual implementation, the entire *startAlphaBeta*() function was manually inlined in the *Iterative Deepening* loop. Line 10 ensures that the entire iterative deepening procedure terminates as soon as a guaranteed win for the agent is detected. This means that the agent automatically prefers fast wins over slow wins, and does not waste any searching time once the win is guaranteed. Similarly, line 12 immediately terminates the search once a loss is guaranteed. The difference is that in this case, the best move of the previous search is returned, which means that the agent will attempt to delay the loss. Line 16 re-orders the moves available in the root node according to the scores found in the last search, which generally results in an improved move ordering for the next search with the new depth.

The agent was simply assigned 30 seconds of searching time per turn. This amount was determined experimentally, and provides a safe margin such that the agent never spends more than 15 minutes in a full game. More complex time management strategies were not implemented.

4.3 Aspiration Search

The *Iterative Deepening* algorithm as described above still starts every search with the largest possible search window of $(-\infty, \infty)$. Generally, a good estimate can be made of evaluations that are realistic to be obtainable, and therefore the initial window can be made smaller to increase the number of prunings. This idea is implemented using *Aspiration Search* (Winands, 2014).

In *Aspiration Search*, before starting an Alpha-Beta Search, two variables $guess$ and Δ are determined such that it is expected that the value propagated back to the root node lies in $\langle guess - \Delta, guess + \Delta \rangle$. When starting a search, α and β are respectively initialized to those two bounds. Suppose that a search returns a score to the root node equal to v. If $guess - \Delta < v < guess + \Delta$, the result is guaranteed to be correct and likely has been found more quickly than it would have been with a larger window. If $v \leq guess - \Delta$, the search needs to be restarted with the window $(-\infty, v)$. Similarly, if $v \geq guess + \Delta$, a re-search using the window (v, ∞) is required.

The values for $guess$ and Δ are dependent on the evaluation function described in Section 5, and have been experimentally determined. This is further elaborated on in Section 7.

5 Evaluation Function

The evaluation function used by the agent for evaluating the game state in any given leaf node is a simple linear combination of two features, named $\Delta material$ and $\Delta progression$. The value of $\Delta material$ in a game state is defined by the difference in the number of Knights between the two players. For example, if the White player is to move, and he has 2 more Knights than the Black player, then $\Delta material = 2$. If the Black player was to move in the same game state, a value of -2 would be assigned to $\Delta material$.

The $\Delta progression$ feature is defined as the difference in the distance between a player's side of the board and the row of the furthest travelled Knight. See Figure 2 for an example. In this state, the White player has $progression = 3$, because he has a Knight 3 rows above the bottom row. The Black player has $progression = 2$, because his furthest Knight is 2 rows below the top row. From the White player's perspective, this game state has $\Delta progression = 3 - 2 = 1$.

Given these two features, the final evaluation of any game state is given by Equation 1, with a weight of 100 for $\Delta material$ and a weight of 35 for $\Delta progression$.

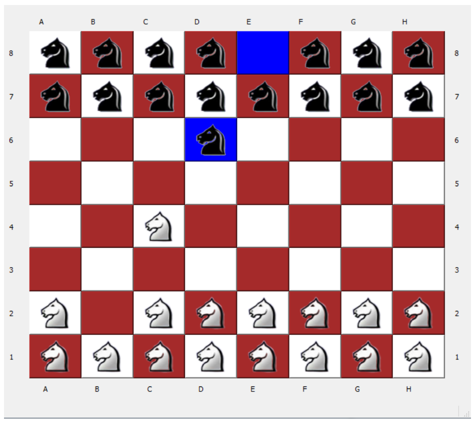

Figure 2: White player's *progression* = 3, Black player's *progression* = 2

$$score = 100 \times \Delta material + 35 \times \Delta progression. \tag{1}$$

In game states that are already terminal, or can be made terminal in a single move because the player to move already has a Knight within 2 rows of the goal row, Equation 1 is not used. In these situations a constant representing a win or a loss is returned depending on whether or not it is the evaluating player that has already won or can win in 1 move.

6 Enhancements

On top of the *Aspiration Search* algorithm described in Section 4, two enhancements have been implemented in the agent; a *Transposition Table* and the *Killer Move Heuristic.*

6.1 Transposition Table

A *Transposition Table* (Greenblatt, Eastlake, & Crocker, 1967) has been implemented to store important search information and avoid repeating searches for game states that can be reached through multiple different move sequences. The table is implemented as a dynamically allocated array with 2^{22} entries. Every entry in the table has room for 2 slots of data, where one slot contains the following variables:

- *bestMove*: The best move to play in this game state, as found by the search process that stored this data.
- *hashValue*: A 64-bit integer that can be used as an identification code for a game state.

- *value*: The value that was assigned to this game state by the search process that stored this data.
- *depth*: An 8-bit integer storing the depth to which the search tree was explored below the node corresponding to this data.
- *valueType* An 8-bit flag to indicate whether the *value* found was an exact value, a lower bound, or an upper bound.

To obtain a 64-bit hash code corresponding to a game state, the Zobrist hashing method (Zobrist, 1970) is used. The features that are used to construct this Zobrist hash code are all combinations of squares and player colors, and a single feature representing the player to move. This is important, because it is possible to reach certain game states in different ways, such that sometimes the Black player is to move and sometimes the White player is to move. Consider for example the move sequence $\langle E2 - G3, E7 - C6, G3 - F5, C6 - A5, F5 - G7\rangle$ and the sequence $\langle E2 - D4, G7 - E6, D4 - E6, E7 - C6, E6 - G7, C6 - A5\rangle$. After the first sequence the Black player is to move, after the second sequence the White player is to move, but apart from that difference the game states are equal.

Whenever a new node is reached during the search process, the first 22 bits of the hash code corresponding to that game state are used as an index into the *Transposition Table*. The remaining 42 bits are used to check which of the 2 slots in that table entry, if any, contain the correct game state. If the correct game state is found, the data is used to speed up the search process in the following way. If the value in the table is an exact value, it is returned immediately. If it is a lower or an upper bound, α or β can be set accordingly. If a search is still necessary, the best move as stored in the table is tried first.

Whenever a game state is evaluated, the hash code is used in the same way to index into the table again and add the new data. If both slots are already occupied, the replacement scheme *TwoDeep* (Breuker, Uiterwijk, & van den Herik, 1994) is used to determine which slot to replace.

6.2 Killer Move Heuristic

The second enhancement used by the agent is the *Killer Move Heuristic*. During a search process, a small table is stored in memory that stores the last 2 moves for each depth level that resulted in a pruning. When the search reaches another node on the same search depth, these 2 *Killer Moves* are tried after any move stored in the *Transposition Table*, but before any other moves.

This heuristic is based on the assumption that moves that were good enough to result in a pruning at the same depth, are likely to still be good moves, and therefore the heuristic improves move ordering. A small disadvantage is that the *Killer Moves* may not be legal moves in certain game states at the same depth, and therefore require a move legality check. Apart from the improved move ordering, another advantage is that, in any case where a move from the

Transposition Table or a *Killer Move* creates a pruning, it is not necessary to generate all the other moves.

7 Experiments

7.1 Aspiration Search Parameters

A small number of experiments has been carried out to determine good ways to determine values for the *guess* and Δ parameters of *Aspiration Search* as described in Section 4. Three different set-ups were tried. They were all tested by letting the *Aspiration Search* engine play against an *Iterative Deepening* engine. Both engines had a *Transposition Table*, but did not yet use the *Killer Move* heuristic. The game states were also not yet implemented using bitboards, but using a less advanced array-based representation.

The first set-up was to always set *guess* to the root node evaluation of the previous iteration during an *Iterative Deepening* process, the last evaluation of the previous search at the start of a new search, or 0 if no history was available. The Δ parameter was set to a constant value of 136. With these settings, the *Iterative Deepening* engine beat the *Aspiration Search* engine due to being able to search deeper. It was noted that the *Aspiration Search* engine often required re-searches in iterations using search depths in the range from 4 to 7.

Increasing Δ to 171 enabled the *Aspiration Search* engine engine to beat the *Iterative Deepening* engine. This noticeably reduced the number of required re-searches, but the increased window size also reduces the number of extra prunings given by *Aspiration Search*.

Finally, Δ was lowered to 100, but *guess* was changed every iteration by adding 141 or -141 to compensate for the odd-even effect often seen in Alpha-Beta search. The values 141 and -141 in this case were experimentally determined. With these settings, the number of re-searches was low (often only initially with a search depth of 1, and in the end-game when wins or losses were detected). The window size is also the smallest among all the tested set-ups. With these settings, the *Aspiration Search* engine was also capable of beating the *Iterative Deepening* engine.

7.2 Enhanced Evaluation Function

In an earlier version of the agent, where the Game State representation was still based on a matrix representing the board instead of the more compact bitboards, an enhanced evaluation function with a third feature was implemented and tested. This third feature, named *controlledProgression*, was similar to the *progression* feature described in Section 5, but only considered the progression of Knights on squares that could be attacked by at least as many allied Knights as the number of opposing attackers.

Using equal search depths, and a small weight of 1 (with the other two features having weights of 35 and 100), this evaluation function was capable of beating the simpler evaluation function of two features. However, at the time, this evaluation function was too expensive and reduced the search depth that was feasible within 30 seconds. Taking this search depth disadvantage into account, it was no longer able to beat the simpler evaluation function.

8 Conclusions & Future Work

This report describes the implementation of an intelligent agent capable of playing the game of KnightThrough. The search engine is based on the Alpha-Beta Search framework, enhanced with *Iterative Deepening* for simple time management, and *Aspiration Search*, a *Transposition Table* and the *Killer Move Heuristic* for reduction of the search space.

An enhanced evaluation function was tested earlier on in development, but at the time found not to provide enough benefit to compensate for the increased cost. The same enhancement could likely be implemented in a much more efficient manner using the bitboards that were used for the game state representation afterwards, but this has not yet been tested.

Another promising idea for future work would be to replace *Aspiration Search* with *Principal Variation Search* (Marsland & Campbell, 1982). Additionally, forward pruning techniques such as Multi-Cut (Björnsson & Marsland, 1999) could be tested. Finally, it would be useful to perform larger numbers of experiments with a larger variety of opponents to test exactly to what extent the discussed enhancements improve the performance in the game of KnightThrough.

References

Björnsson, Y., & Marsland, T. (1999). Multi-Cut Pruning in Alpha-Beta Search. In *Computers and Games* (pp. 15–24). Springer.

Breuker, D. M., Uiterwijk, J. W. H. M., & van den Herik, H. J. (1994). Replacement Schemes for Transposition Tables. *ICCA Journal*, *17*(4), 183–193.

chessprogramming - BitScan. (n.d.). https://chessprogramming.wikispaces.com/BitScan#Bitscan%20forward-De%20Bruijn%20Multiplication. (Accessed: 18-12-2014)

Greenblatt, R. D., Eastlake, D. E., & Crocker, S. D. (1967). The Greenblatt Chess Program. In *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference* (pp. 801–810).

Kerry, H. (2001). 8×8 Game Design Competition: The Winning Game: Breakthrough... and two other favorites. *Abstract Games Magazine*, *7*, 8–9.

Marsland, T., & Campbell, M. (1982). Parallel Search of Strongly Ordered Game Trees. *ACM Computing Surveys (CSUR)*, *14*(4), 533–551.

Winands, M. H. M. (2014). *Ordering & Windows 2014.* Lecture Slides. (Retrieved from: http://eleum.unimaas.nl)

Zobrist, A. L. (1970). A New Hashing Method With Application For Game Playing. *ICCA Journal*, *13*(2), 69–73.