Universiteit Leiden
Opleiding Informatica

Solving Jungle Checkers

Name: Bas van Boven

Date: 19/06/2014

1st supervisor: W.A. Kosters
2nd supervisor: J.K. Vis, J.N. van Rijn

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Jungle Checkers is a simplified version of the traditional Chinese board game Dou Shou Qi. First, we determine the space complexity of Jungle Checkers by counting the number of possible game positions. This number is approximately 2×10^{13}. Furthermore, we try to ultra-weakly solve Jungle Checkers by using the concept of a proof tree: both alpha-beta pruning and proof-number search algorithms are applied. Unfortunately, due to limited hardware resources, we did only manage to solve a tiny part of the game. However, using endgame databases constructed by retrograde analysis we are able to make a number of interesting observations about the game.

Contents

1 Introduction

We examine the board game *Jungle Checkers* [9]. Jungle Checkers is a simplified version of the traditional Chinese board game *Dou Shou Qi* [8] and was first academically used in a course on artificial intelligence at the UNSW School of Computer Science and Engineering [13]. The focus of this thesis is to conclude as much as possible about the *solution* of Jungle Checkers. In the field of Artificial Intelligence, three levels of solving a game are often distinguished [1]:

Ultra-weakly solved: the *game-theoretical value* (win, lose, or draw, seen from the perspective of the starting player) of the initial position of the game is determined.

Weakly solved: a strategy for obtaining the game-theoretical value of the starting position of the game is known (assuming reasonable time and computing resources).

Strongly solved: a strategy for obtaining the game-theoretical value of all possible positions of the game is known (assuming reasonable time and computing resources).

We are primarily concerned with the ultra-weak solution of Jungle Checkers. Although both Jungle Checkers and ultra-weak solutions are almost exclusively of interest to researchers in artificial intelligence, it could provide us with insights in Jungle Chess, which is — as it is a traditional board game — of greater interest.

In the next section, the game rules of Jungle Checkers will be explained and the space complexity of the game will be calculated. After the used approach is clarified, the techniques used for the forward and retrograde analysis will be explained. These techniques are similar to the techniques that were successfully used for solving Checkers [10]. At last, we derive some conclusions from the application of these techniques.
This bachelor thesis is supervised by W.A. Kosters, J.K. Vis and J.N. van Rijn from LIACS, the computer science department of Universiteit Leiden.

2 The game

In this section, the game rules of Jungle Checkers are explained.

2.1 Objective

At the start of the game, each player controls a total of four game pieces. The objective of the game is to place one of your own pieces in the *den* of the opponent or to eliminate all of the opponent's pieces.

2.2 The board

The game board of Jungle Checkers consists of 7×7 squares, see Figure 1. The columns or *files* are labeled `a`–`g` from left to right; the rows or *ranks* are numbered `1`–`7` from bottom to top.

```
7 . r . # . e .
6 . . t . d . .
5 . . . . . . .
4 . . . . . . .
3 . . . . . . .
2 . . D . T . .
1 . E . # . R .
  a b c d e f g
```

Figure 1: The game board with the game pieces in the initial position.

Two squares are special, namely, `d1` and `d7`. These squares are called *dens* and are denoted with `#`. Den `d1` is the den of the white player, and den `d7` is the den of the black player. All of the other squares are ordinary squares.

2.3 Turns and movement

The player that controls the white pieces (from here on simply called *white*) moves first, and from there on both players alternate moves (the other player called *black* controls the black pieces). It is not allowed to skip a move: each player must move a piece each turn. Each piece can move one square either

horizontally or vertically (which leads to both players having to choose out of a maximum of 16 possible moves each turn). It is not allowed for a piece to enter the own den.

2.4 The pieces

Both players control four different pieces, which represent animals. Capital letters are used to denote the white pieces, the lower case letters represent the black pieces. Each animal has a different (relative) strength. The strength of the pieces — listed from weak to strong — is:

1. `R`, `r` — Rat;
2. `D`, `d` — Dog;
3. `T`, `t` — Tiger;
4. `E`, `e` — Elephant.

The strength of an animal determines which of the opponent's animals it can *capture*. Capturing is defined as placing a piece on a square on which a piece of the other player is situated. The piece of the other player is then removed from the board. The general rule is that only pieces with the same or a higher strength may capture an opponent's piece. There is, however, one exception: the rat can — just like the spy in Stratego — capture the elephant (note the elephant can also capture the rat).

For the initial (fixed) placement of the pieces, see Figure 1.

2.5 Stalemate

A situation in which the active player can not play a legal move is called a stalemate: the game now ends in a draw. In Jungle Checkers, a stalemate occurs only when all pieces of a player are trapped by pieces of a higher rank of the opponent. A piece can only be trapped in the corner of the board, against the edge of the board or against the own den.

If both players play optimal, a stalemate will never occur. This is because instead of trapping (a) piece(s), it would always be possible to capture it/them (as all the trapping pieces need to be of higher rank than the trapped pieces).

2.6 Move notation

In this thesis, we use the following move notation:

- The first character of the move denotes the moving piece as explained in Section 2.4. An upper case letter represents a white piece while a lower case letter represents a black piece.
- If the move is a capture, this character is followed by a `x`.
- The final two characters represent the file and rank of the destination square, respectively.

3 Complexity

How difficult it is to solve a game depends for a great deal on two dimensions, as defined in [2]:

- *Decision complexity*: the difficulty to make correct decisions.
- *Space complexity*: the number of positions in the search space.

3.1 Decision complexity

The decision complexity of Jungle Checkers is difficult to determine. As the maximum number of candidate moves available in a certain position is 16, it seems unthinkable that the decision complexity is even comparable to the complexity of, for instance, Chess or Checkers. Furthermore, catching the opponent by surprise by doing an unexpected good move also seems unlikely, considering pieces can only move to adjacent spaces.

3.2 Space complexity

We can determine the number of positions in the search space of Jungle Checkers: the grand total is approximately 2×10^{13}, the exact number is computed over the course of this section. First of all, we split the problem into subproblems: we determine an upper bound for the unfinished positions, i.e., the positions in which a win or stalemate has not yet been established, and the won positions. Furthermore, we determine the exact number of stalemates.

Finally, we have to subtract some unreachable positions from the earlier determined upper bounds to get an exact result.

3.2.1 Unfinished positions

Unfinished positions are defined as positions in which a win or stalemate has not yet been established. The game board consists of 7×7 squares, but two squares are occupied by the dens and positions in which a piece is standing on a den are either a direct win for one of the two players, or invalid. That leaves 47 squares to consider. We have to make a separate calculation for each number of pieces remaining on the board:

- Two pieces remaining on the board: there is always both a white piece and a black piece on the board, which results in 16 different combinations of pieces. For the first piece, there are 47 squares on which it could be located, which leaves 46 squares for the second piece. Besides that, it could be white's or black's turn; so in total, there are $16 \times 47 \times 46 \times 2 = 69{,}184$ different positions.
- Three pieces remaining on the board: there are $\binom{4}{2} \times \binom{4}{1} \times 2 = 48$ possibilities to choose 3 out of 8 pieces. This results in $48 \times 47 \times 46 \times 45 \times 2 = 9{,}339{,}840$ different positions.
- Four pieces remaining on the board: there are $\binom{4}{2} \times \binom{4}{2} + \binom{4}{3} \times \binom{4}{1} \times 2 = 68$ possibilities to choose 4 out of 8 pieces. This results in $68 \times 47 \times 46 \times 45 \times 44 \times 2 = 582{,}183{,}360$ different positions.
- Five pieces remaining on the board: $\binom{8}{5} \times 47 \times 46 \times 45 \times 44 \times 43 \times 2 = 20{,}616{,}140{,}160$ different positions.
- Six pieces remaining on the board: $\binom{8}{6} \times 47 \times 46 \times 45 \times 44 \times 43 \times 42 \times 2 = 432{,}938{,}943{,}360$ different positions.
- Seven pieces remaining on the board: $\binom{8}{7} \times 47 \times 46 \times 45 \times 44 \times 43 \times 42 \times 41 \times 2 = 5{,}071{,}570{,}479{,}360$ different positions.
- All eight pieces remaining on the board: $\binom{8}{8} \times 47 \times 46 \times 45 \times 44 \times 43 \times 42 \times 41 \times 40 \times 2 = 25{,}357{,}852{,}396{,}800$ different positions.

Thus, in total, there are a maximum of 30,883,569,552,064 different unfinished positions.

3.2.2 The parity problem

A special property of the 8-piece positions is that by looking at the position of the game pieces, one can determine which player is the active player. In this section, we explain why this is the case.

First, we introduce the notion of white and black squares for Jungle Checkers: each board square is white or black, square `a1` is white, all horizontally or vertically adjacent squares of a white square are black and all horizontally or vertically adjacent squares of a black square are white.

Now consider the following: for every possible Jungle Checkers position in which all 8 pieces are still on the board, if the number of white squares occupied by a piece is even, white is the active player. Else, black is the active player. The induction below explains why:

1. In the starting position, the number of occupied white squares is even.
2. When white plays a non-capture move, the number of occupied white squares is incremented or decremented by 1 as one piece moves from a black to a white square or vice versa. When black plays a non-capture move, the number of occupied white squares is also incremented or decremented by 1. Thus, the parity of the number of occupied white squares changes with every non-capture move.

This does not hold for situations where a capture already took place: when white captures a black piece which is standing on a white square or black captures a white piece which is standing on a black square, the parity of the number of occupied white squares does not change with the move. For an example of such a situation, see Figure 2.

The conclusion is that every 8-piece Jungle Checkers position only exists with one out of two players being the active player. Thus, we can divide the number of possible 8-piece positions as calculated in Section 3.2.1 by a factor 2.

3.2.3 Stalemates

For further reducing the maximum number of unfinished positions, it is important to determine the number of stalemates in Jungle Checkers as stalemates

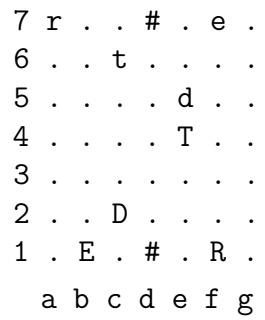

Figure 2: White is the active player and about to play `Txe5`. The parity of the number of occupied white squares does not change.

are currently counted as unfinished positions. In this section the white pieces will be denoted by `1`, `2`, `3` and `4`; the four black pieces will be denoted by `a`, `b`, `c` and `d`.

When counting positions we have to deal with a lot of factors. To maintain an overview we annotate all factors in subscript from here on. The meaning of the used annotations are as follows:

- $n_{\text{situations}}$: there are multiple versions of a certain situation.
- $n_{\text{positions}}$: there are multiple board positions a situation can occur in.
- $n_{\text{permutations}}$: a stalemate can be rearranged in multiple ways.
- $n_{\text{choosewhite}}$: multiple different combinations of white pieces exist.
- $n_{\text{chooseblack}}$: multiple different combinations of black pieces exist.
- n_{choose}: multiple different combinations of both black and white pieces exist.
- $n_{\text{choosefreepiece}}$: the number of different free pieces that can be considered.
- $n_{\text{orderwhite}}$: the white pieces can be rearranged in different ways.
- $n_{\text{orderblack}}$: the black pieces can be rearranged in different ways.
- n_{order}: both black and white pieces can be rearranged in several ways.

- $n_{\text{freepiece}}$: the number of board squares a free piece can be located on.
- 2_{color}: black and white pieces can be interchanged.

We first compute the number of positions where white is stalemated, which is of course equal to the number of positions where black is stalemated, hence the factor 2_{color}. We distinguish several cases.

Situation 1 A white piece is trapped by two black pieces. It can be piece `1`, trapped by `b` and `c`, or piece `2`, trapped by `c` and `d`. There are different variants of this situation. First note that `b` and `c` can be interchanged, and that `1` can be replaced with `2` (and simultaneously `b` with `d`); this causes a factor 4. Second, this construction can occur in the 4 corners, but also on 2 sides next to the white den, leading to a factor 6. Also, there might be other black pieces on the board: none, one or two. This situation in the bottom left corner of the board looks like:

```
  b
1 c
```

The number of positions is as follows:

- $4_{\text{permutations}} \times 6_{\text{positions}} \times 2_{\text{color}} = 48$ 3-piece stalemates.
- $4_{\text{permutations}} \times 6_{\text{positions}} \times \binom{2}{1}_{\text{choosefreepiece}} \times 44_{\text{freepiece}} \times 2_{\text{color}} = 4{,}224$ 4-piece stalemates.
- $4_{\text{permutations}} \times 6_{\text{positions}} \times 44_{\text{freepiece}} \times 43_{\text{freepiece}} \times 2_{\text{color}} = 90{,}816$ 5-piece stalemates.

Situation 2 Two white pieces are trapped by three black pieces. It concerns pieces `1`, and `2` or `3`, trapped by `b`, `c` and `d`. In the left figure, `c` and `d` can be interchanged, in the middle and right figure `b` and `c`. The construction can occur in the 4 corners and on 2 sides besides the white den, both horizontal and vertical (the latter only in the corners), leading to a factor 10. Also, there might be another black piece on the board. This situation in the bottom left corner of the board looks like:

```
b c      d b      d c
1 2 d    2 1 c    3 1 b
```

The number of positions is as follows:

- $3_{\text{situations}} \times 2_{\text{permutations}} \times 10_{\text{positions}} \times 2_{\text{color}} = 120$ 5-piece stalemates.
- $3_{\text{situations}} \times 2_{\text{permutations}} \times 10_{\text{positions}} \times 42_{\text{freepiece}} \times 2_{\text{color}} = 5{,}040$ 6-piece stalemates.

Situation 3 Three white pieces are trapped between a corner and the white den by three black pieces. It concerns pieces `1`, `2` and `3`, trapped by `b`, `c` and `d`. The pieces `1`, `2` and `3` allow for $3! = 6$ permutations. This construction can happen at 2 places. Also, there might be another black piece on the board. This situation in the bottom left corner of the board looks like:

```
b c d
1 2 3 #
```

The number of positions is as follows:

- $6_{\text{permutations}} \times 2_{\text{positions}} \times 2_{\text{color}} = 24$ 6-piece stalemates.
- $6_{\text{permutations}} \times 2_{\text{positions}} \times 41_{\text{freepiece}} \times 2_{\text{color}} = 984$ 7-piece stalemates.

Situation 4 Two white pieces are trapped between a corner and the white den by three black pieces. In this case, `1` and `2` can be interchanged (at the same time swapping `b` and `d`). The situation can occur at 2 locations. Also, there might be another black piece on the board. This situation in the bottom left corner of the board looks like:

```
b   d
1 c 2 #
```

The number of positions is as follows:

- $2_{\text{permutations}} \times 2_{\text{positions}} \times 2_{\text{color}} = 8$ 5-piece stalemates.
- $2_{\text{permutations}} \times 2_{\text{positions}} \times 42_{\text{freepiece}} \times 2_{\text{color}} = 336$ 6-piece stalemates.

Situation 5 Three white pieces are trapped in a corner by three black pieces. It concerns pieces `1`, `2` and `3`, trapped by `b`, `c` and `d`. There are 4 permutations (`4` can be `3` and `1` and `2` with `b` and `c` can be interchanged too). This construction can happen at 4 places. Also, there might be another black piece on the board. This situation in the bottom left corner of the board looks like:

```
b
1 c
3 2 d
```

The number of positions is as follows:

- $4_{\text{permutations}} \times 4_{\text{positions}} \times 2_{\text{color}} = 32$ 6-piece stalemates.
- $4_{\text{permutations}} \times 4_{\text{positions}} \times 41_{\text{freepiece}} \times 2_{\text{color}} = 1{,}312$ 7-piece stalemates.

Situation 6 Four white pieces are trapped between a corner and the white den by three black pieces. All 3 possibilities can occur at 2 positions. Also, there might be another black piece on the board. The 8-piece positions only exist in 1 out of 2 colors due to the parity problem, hence the factor 2_{color} is omitted there. This situation in the bottom left corner of the board looks like:

```
b           b           c
1 c d       1 d c       2 d b
4 2 3 #     4 3 2 #     4 3 1 #
```

The number of positions is as follows:

- $3_{\text{situations}} \times 2_{\text{positions}} \times 2_{\text{color}} = 12$ 7-piece stalemates.
- $3_{\text{situations}} \times 2_{\text{positions}} \times 40_{\text{freepiece}} = 240$ 8-piece stalemates.

3.2.4 Unreachable unfinished positions

There also exist some positions that are unreachable in Jungle Checkers. The reason for this is that there exist situations — such as stalemates — where a player is surrounded by the other player. Positions where the surrounded player is not the active player cannot exist because it is impossible for a player to surround itself. In this section, all these positions are explained. Furthermore, it should be noted that we explain all these situations for white being the surrounded player, but they all exist for black being the active player too, hence the factor 2_{color} in every situation.

Situation 1 A white piece is surrounded in a corner or next to a den by two black pieces. This is possible in 8 positions. There exist 3-, 4- and 5-piece positions of this situation because the 2 remaining black pieces can be present on the board too. This situation in the bottom left corner of the board looks like:

```
a
1 b
```

The number of positions is as follows:

- $8_{\text{positions}} \times \binom{4}{1}_{\text{choosewhite}} \times \binom{4}{2}_{\text{chooseblack}} \times 2!_{\text{orderblack}} \times 2_{\text{color}} = 768$ unreachable 3-piece positions.
- $8_{\text{positions}} \times \binom{4}{1}_{\text{choosewhite}} \times \binom{4}{3}_{\text{chooseblack}} \times 3!_{\text{orderblack}} \times 44_{\text{freepiece}} \times 2_{\text{color}} =$ 67,584 unreachable 4-piece positions.
- $8_{\text{positions}} \times \binom{4}{1}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 12_{\text{orderblack}} \times 44_{\text{freepiece}} \times 43_{\text{freepiece}} \times 2_{\text{color}} =$ 1,453,056 unreachable 5-piece positions.

Situation 2 A white piece is surrounded at a board side or next to a den by three black pieces. This is possible in 16 positions. There exist 4- and 5-piece positions of this situation because the remaining black piece can be present on the board too. This situation against the left side of the board looks like:

```
a
1 b
c
```

The number of positions is as follows:

- $16_{\text{positions}} \times \binom{4}{1}_{\text{choosewhite}} \times \binom{4}{3}_{\text{chooseblack}} \times 3!_{\text{orderblack}} \times 2_{\text{color}} =$ 3,072 unreachable 4-piece positions.
- $16_{\text{positions}} \times \binom{4}{1}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 4!_{\text{orderblack}} \times 43_{\text{freepiece}} \times 2_{\text{color}} =$ 132,096 unreachable 5-piece positions.

Situation 3 A white piece is surrounded by four black pieces. This is possible in 23 positions. This situation looks like:

```
  a
b 1 c
  d
```

The number of positions is as follows:

- $23_{\text{positions}} \times \binom{4}{1}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 4!_{\text{orderblack}} \times 2_{\text{color}} = 4{,}416$ unreachable 5-piece positions.

Situation 4 Two white pieces are surrounded in a corner or next to a den by three black pieces. This is possible in 12 positions. There exist 5- and 6-piece positions of this situation because the remaining black piece can be present on the board too. This situation in the bottom left corner of the board looks like:

```
a b
1 2 c
```

The number of positions is as follows:

- $12_{\text{positions}} \times \binom{4}{2}_{\text{choosewhite}} \times \binom{4}{3}_{\text{chooseblack}} \times 2!_{\text{orderwhite}} \times 3!_{\text{orderblack}} \times 2_{\text{color}} = 6{,}912$ unreachable 5-piece positions.
- $12_{\text{positions}} \times \binom{4}{2}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 2!_{\text{orderwhite}} \times 4!_{\text{orderblack}} \times 42_{\text{freepiece}} \times 2_{\text{color}} = 290{,}304$ unreachable 6-piece positions.

Situation 5 Two white pieces are surrounded between a den, a corner and three black pieces. This is possible in 4 positions. There exist 5- and 6-piece positions of this situation because the remaining black piece can be present on the board too. This situation in the bottom left corner of the board looks like:

```
a   b
1 c 2 #
```

The number of positions is as follows:

- $4_{\text{positions}} \times \binom{4}{2}_{\text{choosewhite}} \times \binom{4}{3}_{\text{chooseblack}} \times 2!_{\text{orderwhite}} \times 3!_{\text{orderblack}} \times 2_{\text{color}} = 2{,}304$ unreachable 5-piece positions.
- $4_{\text{positions}} \times \binom{4}{2}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 2!_{\text{orderwhite}} \times 4!_{\text{orderblack}} \times 42_{\text{freepiece}} \times 2_{\text{color}} = 96{,}768$ unreachable 6-piece positions.

Situation 6 Two white pieces are surrounded by four black pieces in a corner or right besides the den. Both situations are possible in 4 positions. This situation in the bottom left corner of the board and against the right side of the white den looks like:

```
a           a           a
1 b       b 1 c       b 2 c
c 2 d     d 2 #     d 1 #
```

The number of positions is as follows:

- $3_{\text{situations}} \times 4_{\text{positions}} \times \binom{4}{2}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 2!_{\text{orderwhite}} \times 4!_{\text{orderblack}} \times 2_{\text{color}} = 6{,}912$ unreachable 6-piece positions.

Situation 7 Three white pieces are surrounded by three black pieces in a corner. Both situations are possible in 4 positions. There exist 6- and 7-piece positions of this situation because the remaining black piece can be present on the board too. This situation in the bottom left corner of the board and against the right side of the white den looks like:

```
a
1 b         a b c
2 3 c       1 2 3 #
```

The number of positions is as follows:

- $2_{\text{situations}} \times 4_{\text{positions}} \times \binom{4}{3}_{\text{choosewhite}} \times \binom{4}{3}_{\text{chooseblack}} \times 3!_{\text{orderwhite}} \times 3!_{\text{orderblack}} \times 2_{\text{color}} = 9{,}216$ unreachable 6-piece positions.
- $2_{\text{situations}} \times 4_{\text{positions}} \times \binom{4}{3}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 3!_{\text{orderwhite}} \times 4!_{\text{orderblack}} \times 41_{\text{freepiece}} \times 2_{\text{color}} = 377{,}856$ unreachable 7-piece positions.

Situation 8 A white piece is surrounded by two black pieces, two times. This is possible in 36 different positions. This situation looks like:

```
a     +   c
1 b       2 d
```

The number of positions is as follows:

- $44_{\text{positions}} \times \binom{4}{2}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 2!_{\text{orderwhite}} \times 4!_{\text{orderblack}} \times 2_{\text{color}} = 25{,}344$ unreachable 6-piece positions.

Situation 9 Three white pieces are surrounded by four black pieces in various ways. All 4 situations are possible in 4 positions. This situation in the bottom left corner of the board and against the right side of the white den looks like:

```
  a          a            a          a          a
b 1 c      1 b c        b 1 c      2 c      1 b c
2 d 3 #    d 2 3 #    d 2 3 #      3 d      2 d 3 #
```

The number of positions is as follows:

- $5_{\text{situations}} \times 4_{\text{positions}} \times \binom{4}{3}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 3!_{\text{orderwhite}} \times 4!_{\text{orderblack}} \times 2_{\text{color}} = 23{,}040$ unreachable 7-piece positions.

Situation 10 Three white pieces are surrounded in a corner or next to a den by four black pieces. This is possible in 12 different positions. This situation in the bottom left corner of the board looks like:

```
  a
b 1 c
2 3 d
```

The number of positions is as follows:

- $12_{\text{positions}} \times \binom{4}{3}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 3!_{\text{orderwhite}} \times 4!_{\text{orderblack}} \times 2_{\text{color}} = 13{,}824$ unreachable 7-piece positions.

Situation 11 Four white pieces are surrounded between a corner and a den by three black pieces. Both situations are possible in 4 positions. There exist 7- and 8-piece positions of both situations because the remaining black piece can be present on the board too. The 8-piece positions only exist in 1 out of 2 colors due to the parity problem, hence the factor 2_{color} is omitted there. This situation in the bottom left corner of the board looks like:

```
  a          a
b 1 c      1 b c
2 3 4 #    2 3 4 #
```

The number of positions is as follows:

- $2_{\text{situations}} \times 4_{\text{positions}} \times \binom{4}{3}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 3!_{\text{orderwhite}} \times 4!_{\text{orderblack}} \times 2_{\text{color}} = 9{,}216$ unreachable 7-piece positions.
- $2_{\text{situations}} \times 4_{\text{positions}} \times \binom{4}{4}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 4!_{\text{orderwhite}} \times 4!_{\text{orderblack}} \times 40_{\text{freepiece}} = 184{,}320$ unreachable 8-piece positions.

Situation 12 Four white pieces are surrounded by four black pieces in various ways. All 6 situations are possible in 4 positions. The 8-piece positions only exist in 1 out of 2 colors due to the parity problem, hence the factor 2_{color} is omitted. This situation in the bottom left corner of the board and against the right side of the white den looks like:

```
a           a
1 b         1 b         a b
2 c         2 3 c       1 2 c
3 4 d       4 d         3 4 d #

    a       a b         a b
b c 1 d     1 2 c       1 2 c
2 3 4 #     d 3 4 #     3 d 4 #
```

The number of positions is as follows:

- $6_{\text{situations}} \times 4_{\text{positions}} \times \binom{4}{4}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 4!_{\text{orderwhite}} \times 4!_{\text{orderblack}} = 13{,}824$ unreachable 8-piece positions.

3.2.5 Won positions

Besides the unfinished positions and stalemates, there are won positions: positions in which a win has just been achieved by one of the players. There are two types of won positions: the positions in which one of the players has no material remaining on the board, and the positions in which a piece is standing on the den of the opponent.

Calculating the number of this first type of won positions is fairly trivial. We assume for now that white is the player with material remaining on the board. There are 4 different types of this situation:

- White has one piece remaining on the board: there are 4 different pieces to choose from, and 47 possible squares for this piece; so in total, there are $4 \times 47 = 188$ different positions.

- White has two pieces remaining on the board: choose 2 out of 4 different pieces, after which there are 47 possible squares for the first piece and 46 possible squares for the second piece. In total, there are $6 \times 47 \times 46 =$ 12,972 different positions.

- White has three pieces remaining on the board: $4 \times 47 \times 46 \times 45 =$ 389,160 different positions.

- White has four pieces remaining on the board: $1 \times 47 \times 46 \times 45 \times 44 =$ 4,280,760 different positions.

Besides that, all these positions are possible too if black is the player with pieces remaining on the board, which leads to a factor 2 to the positions above.

Then, the number of the second type of won positions: the type in which a piece is standing on the den of the opponent. There are 7 types of this situation, assuming again that white is the player that has won, i.e., has a piece standing on the den of the opponent:

- Two pieces remaining on the board: there is always both a white piece and a black piece on the board, which results in 16 different combinations of pieces. The opponent's piece is located on 1 of 47 possible board squares; so in total, there are $16 \times 47 = 752$ different positions.

- Three pieces remaining on the board: there are 72 combinations of three pieces, of which a specific white piece is standing on the den of the opponent (24 1 versus 2 combinations + 48 2 versus 1 combinations). The 2 other pieces can stand on 1 out of 47 and 1 out of 46 squares, respectively. In total, there are thus $72 \times 47 \times 46 =$ 155,664 different positions.

- Four pieces remaining on the board: there are 136 different combinations (16 1 versus 3 combinations + 72 2 versus 2 combinations + 48 3 versus 1 combinations). In total, there are thus $136 \times 47 \times 46 \times 45 =$ 13,231,440 different positions.

- Five pieces remaining on the board: there are 140 different combinations (4 1 versus 4 combinations + 48 2 versus 3 combinations + 72 3 versus 2 combinations + 16 4 versus 1 combinations). In total, there are thus $140 \times 47 \times 46 \times 45 \times 44 =$ 599,306,400 different positions.

- Six pieces remaining on the board: there are 84 different combinations (12 2 versus 4 combinations + 48 3 versus 3 combinations + 24 4 versus 2 combinations). In total, there are thus $84 \times 47 \times 46 \times 45 \times 44 \times 43 =$ 15,462,105,120 different positions.

- Seven pieces remaining on the board: there are 28 different combinations (12 3 versus 4 combinations + 16 4 versus 3 combinations). In total, there are thus $28 \times 47 \times 46 \times 45 \times 44 \times 43 \times 42 =$ 216,469,471,680 different positions.

- All eight pieces remaining on the board: 4 different white pieces can be placed on the opponent's den, resulting in $4 \times 47 \times 46 \times 45 \times 44 \times 43 \times 42 \times 41 =$ 1,267,892,619,840 different positions. However, the 8-piece won positions are also prone to the parity problem described earlier, which leads to a factor 2 reduction of the total number of possible positions. There are thus only 633,946,309,920 different positions.

Again, all these positions are possible too if black is the player that is standing on the den of the opponent, which leads to a factor 2 to the positions above.

3.2.6 Unreachable won positions

There also exist some won positions which are unreachable, which are explained in this section. Again, it should be noted that we explain all these situations for white being the surrounded player, but they all exist for black being the active player too, hence the factor 2_{color} in every situation.

Situation 1 A very unique situation: a white piece is standing on the black den, while this den is surrounded by three other pieces. The color of the surrounding pieces is not important, hence these 3 pieces are denoted with a $*$. There exist 4-, 5-, 6-, 7- and 8-piece positions of this situation because the remaining pieces can be present on the board too. However, half of the 8-piece positions are invalid due to the parity problem. This situation around the white den looks like:

```
  *
* # *
```

The number of positions is as follows:

- $68_{\text{choose}} \times 4!_{\text{order}} = 1{,}632$ unreachable 4-piece positions.
- $\binom{8}{5}_{\text{choose}} \times 5!_{\text{order}} \times 44_{\text{freepiece}} = 295{,}680$ unreachable 5-piece positions.
- $\binom{8}{6}_{\text{choose}} \times 360_{\text{order}} \times 44_{\text{freepiece}} \times 43_{\text{freepiece}} = 19{,}071{,}360$ unreachable 6-piece positions.
- $\binom{8}{7}_{\text{choose}} \times 840_{\text{order}} \times 44_{\text{freepiece}} \times 43_{\text{freepiece}} \times 42_{\text{freepiece}} = 533{,}998{,}080$ unreachable 7-piece positions.
- $(\binom{8}{8}_{\text{choose}} \times 1680_{\text{order}} \times 44_{\text{freepiece}} \times 43_{\text{freepiece}} \times 42_{\text{freepiece}} \times 41_{\text{freepiece}}) \div 2_{\text{parity}} = 2{,}736{,}740{,}160$ unreachable 8-piece positions.

Situation 2 A white piece is surrounded in a corner or next to a den by two black pieces, while a black piece is standing in the white den. This is possible in 8 positions. There exist 4- and 5-piece positions of this situation because the remaining black piece can be present on the board too. This situation in the bottom left corner of the board looks like:

```
a
1 b
```

The number of positions is as follows:

- $8_{\text{positions}} \times \binom{4}{1}_{\text{choosewhite}} \times \binom{4}{3}_{\text{chooseblack}} \times 3!_{\text{orderblack}} \times 2_{\text{color}} = 1{,}536$ unreachable 4-piece positions.
- $8_{\text{positions}} \times \binom{4}{1}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 4!_{\text{orderblack}} \times 44_{\text{freepiece}} \times 2_{\text{color}} = 67{,}584$ unreachable 5-piece positions.

Situation 3 A white piece is surrounded at a board side or next to a den by three black pieces, while a black piece is standing in the white den. This is possible in 16 positions. This situation against the left side of the board looks like:

```
a
1 b
c
```

The number of positions is as follows:

- $16_{\text{positions}} \times \binom{4}{1}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 4!_{\text{orderblack}} \times 2_{\text{color}} = 3{,}072$ unreachable 5-piece positions.

Situation 4 Two white pieces are surrounded in a corner or next to a den by three black pieces, while a black piece is standing in the white den. This is possible in 12 positions. This situation in the bottom left corner of the board looks like:

```
a b
1 2 c
```

The number of positions is as follows:

- $12_{\text{positions}} \times \binom{4}{2}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 2!_{\text{orderwhite}} \times 4!_{\text{orderblack}} \times 2_{\text{color}} =$ 6,912 unreachable 6-piece positions.

Situation 5 Two white pieces are surrounded between a den, a corner and three black pieces, while a black piece is standing in the white den. This is possible in 4 positions. This situation in the bottom left corner of the board looks like:

```
a   b
1 c 2 #
```

The number of positions is as follows:

- $4_{\text{positions}} \times \binom{4}{2}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 2!_{\text{orderwhite}} \times 4!_{\text{orderblack}} \times 2_{\text{color}} =$ 2,304 unreachable 6-piece positions.

Situation 6 Three white pieces are surrounded by three black pieces in a corner, while a black piece is standing in the white den. Both situations are possible in 4 positions. This situation in the bottom left corner of the board looks like:

```
a
1 b        a b c
2 3 c      1 2 3 #
```

The number of positions is as follows:

- $2_{\text{situations}} \times 4_{\text{positions}} \times \binom{4}{3}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 3!_{\text{orderwhite}} \times 4!_{\text{orderblack}} \times 2_{\text{color}} =$ 9,216 unreachable 7-piece positions.

Situation 7 Four white pieces are surrounded between a corner and a den by three black pieces, while a black piece is standing in the white den. Both situations are possible in 4 positions. The 8-piece positions only exist in 1 out of 2 colors due to the parity problem, hence the factor 2_{color} is omitted. This situation in the bottom left corner of the board looks like:

```
  a          a
b 1 c        1 b c
2 3 4 #      2 3 4 #
```

The number of positions is as follows:

- $2_{\text{situations}} \times 4_{\text{positions}} \times \binom{4}{4}_{\text{choosewhite}} \times \binom{4}{4}_{\text{chooseblack}} \times 4!_{\text{orderwhite}} \times 4!_{\text{orderblack}} =$ 4,608 unreachable 8-piece positions.

3.2.7 Conclusion

The determination of the space complexity is summarized in Table 1.

Up until 6 pieces remaining, we have verified our results with a position counting program and suspect there are no unidentified positions remaining.

Table 1: Number of positions of Jungle Checkers.

In total, there thus exist 19,934,340,958,800 $\approx 2 \times 10^{13}$ Jungle Checkers positions. This is way smaller than the search space of regular Checkers, which consists of approximately 5×10^{20} positions [11].

4 Approach

We try to determine the game-theoretic value of the starting position of Jungle Checkers by using the concept of a proof tree. In the — unrealistic — best case, for a game that is provably winning or losing, the proof-tree search would only have to look at the square root of the search space (around 5,600,000 positions): at losing or drawn positions all possible moves still have to be considered (since we need to be absolutely sure we can not do better), but at winning positions, we need only to look at one move in the ideal

case [11]. This essentially means that knowing which positions to look at will be the main difficulty.

For this research project, we try to ultra-weakly solve Jungle Checkers using a proof-tree search algorithm. We try to use this algorithm to find the game-theoretic value (win, loss or draw) corresponding to the starting position of the game. Both alpha-beta search (with and without iterative deepening) and proof-number search variants of proof-tree search were considered. This proof-tree search is also called *forward analysis* from here on.

To aid the proof-tree search, we have also built endgame databases using retrograde analysis [12]. These endgame databases contain the game-theoretic value of all positions with up to 5 pieces remaining on the board. When the proof-tree search arrives at a position with only 5 pieces remaining on the board, these databases are consulted. Now, the whole subtree of positions with 5 remaining pieces does not have to be solved, which should save a large amount of processing time.

Note that our approach is very similar to how Shaeffer et al. succeeded in solving regular Checkers [10].

5 Forward analysis

The basis for the forward analyzing engine is minimax search. This basis is optimized by using alpha-beta pruning and transposition tables. Furthermore, to prefer shorter wins over longer wins, depth correction was also implemented.

Using a variant of proof-number search as a basis for the forward analysis was also tried, but it does not seem to be as efficient as alpha-beta search in Jungle Checkers. The results of this experiment are also described in this section.

5.1 Minimax search

Minimax search is based on the minimax theorem, which was proved by John von Neumann in 1928 [6]. The theorem is valid for every two-person, zero-sum

game. A zero-sum game is defined as a game in which a player's gain or loss is balanced by the gain or loss of (an)other player(s). When one adds up the gains of all the involved players, the result is always zero. In other words: one player's loss is another player's gain. A two-player zero-sum game is therefore a game in which, when one player wins the game, the other player loses. It is trivial to see that Jungle Checkers is in fact a two-player zero-sum game: only one player can win and with each move a player can increase or decrease his/her distance to a win. This change in proximity is inversely proportional with the chance that the opponent wins the game.

In any board game, a player should always choose the move that leads to the best position possible: a player tries to maximize his own score. However, in a zero-sum game, this is the same as minimizing the score of the opponent. Therefore, it is possible to represent the balance between the two players as one game-theoretical value.

Minimax search is a type of search that tries to find the best possible move in a specified position, by recursively calculating the game-theoretical value of the positions resulting from every possible move. In each position, the best move (from the viewpoint of the active player) is now chosen.

In some games — like Jungle Checkers — the game tree is very deep, too deep to search entirely. Therefore, a fixed search depth is provided to the minimax search. If the algorithm is evaluating a position which is at this fixed depth, instead of searching deeper, an evaluation function is called which returns an estimation of the value of the position. We have experimented with different evaluation functions during the project. The evaluation function that was used in most of the computations is included in the Appendix.

When minimax search is implemented by switching signs on the recursive function call, this algorithm is called negamax. Negamax is arguably the cleanest implementation of minimax. Pseudocode implementation of negamax search [4] is listed in Algorithm 1.

```
1: procedure NEGAMAX(position, depth)
2:
3:     // if won, stalemate or in a leaf node, return score
4:     if WON(position) then return WIN
```

```
 5:     if STALEMATE(position) then return STALEMATE
 6:     if depth = 0 then return EVALUATE(position)
 7:
 8:     // start preparing for move sweep
 9:     moves := GENERATE_MOVES(position)
10:     best := −∞
11:
12:     // move sweep
13:     for all move of moves do
14:
15:         // try move
16:         POSITION.DO_MOVE(move)
17:         value := −NEGAMAX(position, depth − 1)
18:         POSITION.UNDO_MOVE(move)
19:
20:         // determine new value
21:         if value > best then best = value
22:
23:     // return best score
24:     return best
25:
26: end procedure
```

Algorithm 1: Negamax

5.2 Alpha-beta pruning

Negamax search can be improved further by implementing alpha-beta pruning. Alpha-beta pruning is a technique that prunes branches of the search tree (while still producing the same result as plain negamax search). Therefore, implementing alpha-beta pruning in negamax generally results in a significant time reduction. It could theoretically make the difference between considering the entire search space or only the part necessary for the proof tree, as described in Section 4.

The name alpha-beta pruning refers to two bounds that are passed along the recursive search. The α-bound represents the highest score that the maximizing player can be assured of and the β-bound represents the lowest score that the minimizing player can realize. Now, when a position/search node with value v is being considered, it (along with al its subtrees) can be pruned if $\alpha \leq v \leq \beta$ does not hold.

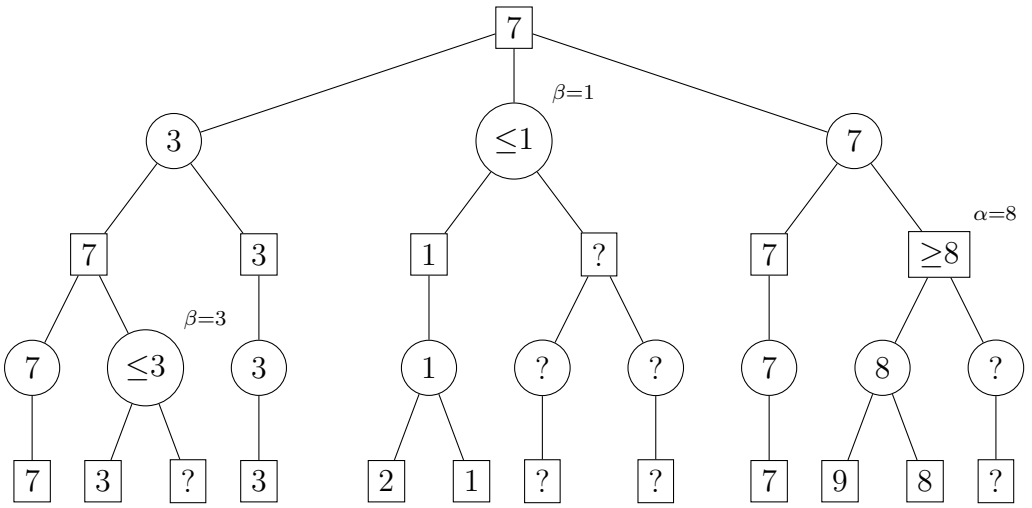

Figure 3: Example of a search tree; squares represent maximizing nodes and circles represent minimizing nodes. At cutoff nodes, the relevant bound is also given.

For an example of the effect of alpha-beta pruning, consider the search tree depicted in Figure 3. The nodes that are marked with a question mark are not visited, yet the value of the root node can be determined.

Furthermore, we also sort all candidate moves in each node to look at the most promising moves first, as determined by a quick evaluation function which looks at the direct development gain of the move and the possible material loss of the opponent. This evaluation function is listed in the appendix, and pseudocode implementation of negamax search with alpha-beta pruning [4] is listed in Algorithm 2.

1: **procedure** ALPHABETA($position, depth, \alpha, \beta$)
2:
3: `// if won, stalemate or in a leaf node, return score`
4: **if** WON($position$) **then return** WIN
5: **if** STALEMATE($position$) **then return** STALEMATE
6: **if** $depth = 0$ **then return** EVALUATE($position$)
7:

```
 8:     // start preparing for move sweep
 9:     moves := GENERATE_MOVES(position)
10:     moves := SORT_MOVES(moves)
11:     best := −∞
12:
13:     // move sweep
14:     for all move of moves do
15:
16:         // try move
17:         POSITION.DO_MOVE(move)
18:         value := −ALPHABETA(position, depth − 1, −β, −α)
19:         POSITION.UNDO_MOVE(move)
20:
21:         // determine new value and bounds
22:         if value > best then best = value
23:         if value > α then
24:             α := value
25:             if α ≥ β then break
26:
27:     // return best score
28:     return best
29:
30: end procedure
```

Algorithm 2: Alpha-beta pruning

5.3 Transposition tables

Consider a Jungle Checkers game position — for example, the initial configuration depicted in Figure 1 — and suppose the white player would like to move both his rat and his elephant forward. This can be done in two ways (supposing the opponent's move is the same in both cases): `1.Eb2 2.`x `3.Rf2` or `1.Rf2 2.`x `3.Eb2`. One could call `1.Rf2 2.`x `3.Eb2` a *transposition* of `1.Eb2 2.`x `3.Rf2`, and vice versa. In negamax search with alpha-beta pruning, positions like these will lead to calculating the value of the same subtree more than once, which can computationally be very expensive.

A solution to this problem is implementing *transposition tables* in alpha-beta search. Transposition tables are tables in which already analyzed positions are stored. In our implementation, the table is indexed by a key that is computed based on the game position (more specifically, we have implemented Zobrist hashing) [14]. After the value of a certain subtree is calculated, it is stored in

the transposition table. Before the search algorithm starts to compute the value of a subtree, the transposition table is consulted to see if it already contains an evaluation for the given position which is at least as deep as the requested search depth. This depth constraint is necessary since on a smaller search depth the game tree is not expanded far enough.

Note that there are so many possible game configurations in Jungle Checkers that the number of calculated configurations will usually quickly exceed the available capacity of the transposition table (which is dependent on the amount of main memory available). Thus, making use of transposition tables never fully eliminates the revisiting of nodes. This can also lead to repetitions being evaluated more than once in some occasions as we do not handle those specifically.

Pseudocode implementation of alpha-beta search with transposition table lookup is listed in Algorithm 3 [4].

```
 1: procedure ALPHABETA(position, depth, α, β)
 2:
 3:     // the original alpha value could be modified
 4:     alphaOrig := α
 5:
 6:     // fetch what is known already from the transposition table
 7:     entry := TT_FETCH(position)
 8:     if entry.type = EXACT then return entry.value
 9:     else if entry.type = LOWER then
10:         if entry.value > α then α := entry.value
11:     else if entry.type = UPPER then
12:         if entry.value < β then β := entry.value
13:
14:     // if alpha >= beta now holds, entry.value is the optimal result
15:     if α >= β then return entry.value
16:
17:     // if won, stalemate or in a leaf node, return score
18:     if WON(position) then return WIN
19:     if STALEMATE(position) then return STALEMATE
20:     if depth = 0 then return EVALUATE(position)
21:
22:     // start preparing for move sweep
23:     moves := GENERATE_MOVES(position)
24:     moves := SORT_MOVES(moves)
```

```
25:     best := −∞
26:     move_tt := 0
27:
28:     // move sweep
29:     for all move of moves do
30:
31:         // try move
32:         POSITION.DO_MOVE(move)
33:         value := −ALPHABETA(position, depth − 1, −β, −α)
34:         POSITION.UNDO_MOVE(move)
35:
36:         // determine new value and bounds
37:         if value > best then best = value
38:         if value > α then
39:             α := value
40:             move_tt := i
41:             if α ≥ β then break
42:
43:     // store found move in transposition table
44:     entry.depth := depth
45:     entry.value := best
46:     entry.move := move_tt
47:     if best < alphaOrig then entry.type = UPPER
48:     else if best >= beta then entry.type = LOWER
49:     else entry.type = EXACT
50:     TT_STORE(key, entry)
51:
52:     // return best score
53:     return best
54:
55: end procedure
```

Algorithm 3: Alpha-beta pruning with transposition table lookup.

5.4 Depth correction

Consider the position in Figure 4. If black is to move, it is evident that `eb1` is the optimal move: it can always lead to a win in three black moves. Another possible move is `df2`, but in this position black would need at least four moves.

The problem with positions like these is that alpha-beta search sees no difference between these two moves: they both lead to a guaranteed win, so each move receives the highest possible evaluation score. However, an algorithm

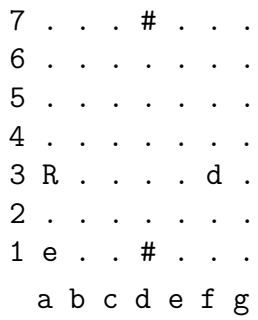

Figure 4: The game board with a position that is winning for black.

that does not prefer a quicker win might not necessarily get to actually realize a win because it is faced with the same choice between maximum evaluation scores in the next turn(s). This can also happen in the alpha-beta algorithm.

A solution to this problem is to subtract the node depth from the highest possible evaluation score (and, conversely, add the node depth to the lowest possible evaluation score in case of a loss). This way, shorter wins are chosen over longer wins. Note that we do not apply this addition or subtraction in the general case since that could mean that, theoretically, a promising position could be preferred above a very deep win.

A drawback of this strategy is that the execution time of alpha-beta search increases; previously, the search tree would get pruned rather quickly if one player could secure a win somewhere. In the new position, the search still tries to find a shorter win afterwards, which leads to the visiting of extra nodes. There seems to be nothing one can do to overcome this drawback. However, with this depth correction enabled, we can be sure that alpha-beta search follows the optimal line of play. Moreover, this can also result in a smaller proof tree.

Pseudocode implementation of alpha-beta search with transposition table lookup and depth correction is listed in Algorithm 4 [4].

```
1: procedure ALPHABETA(position, depth, α, β)
2:
3:     // the original alpha value could be modified
```

```
4:    alphaOrig := α
5:
6:    // fetch what is known already from the transposition table
7:    entry := TT_FETCH(position)
8:    if entry.type = EXACT then return entry.value
9:    else if entry.type = LOWER then
10:       if entry.value > α then α := entry.value
11:   else if entry.type = UPPER then
12:       if entry.value < β then β := entry.value
13:
14:   // if alpha >= beta now holds, entry.value is the optimal result
15:   if α >= β then return entry.value
16:
17:   // if won, stalemate or in a leaf node, return score
18:   if WON(position) then return WIN
19:   if STALEMATE(position) then return STALEMATE
20:   if depth = 0 then return EVALUATE(position)
21:
22:   // start preparing for move sweep
23:   moves := GENERATE_MOVES(position)
24:   moves := SORT_MOVES(moves)
25:   best := −∞
26:   move_tt := 0
27:
28:   // move sweep
29:   for all move of moves do
30:
31:      // try move
32:      POSITION.DO_MOVE(move)
33:      value := −ALPHABETA(position, depth − 1, −β, −α)
34:      POSITION.UNDO_MOVE(move)
35:
36:      // depth correction: prefer shorter wins
37:      if value > WINRANGE then value := value − 1
38:      else if value < LOSERANGE then value := value + 1
39:
40:      // determine new value and bounds
41:      if value > best then best = value
42:      if value > α then
43:         α := value
44:         move_tt := i
45:         if α ≥ β then break
46:
47:   // store found move in transposition table
```

```
48:     entry.depth := depth
49:     entry.value := best
50:     entry.move := move_tt
51:     if best < alphaOrig then entry.type = UPPER
52:     else if best >= β then entry.type = LOWER
53:     else entry.type = EXACT
54:     TT_STORE(key, entry)
55:
56:     // return best score
57:     return best
58:
59: end procedure
```

Algorithm 4: Alpha-beta pruning with transposition table lookup and depth correction.

The resulting algorithm is an optimized version of alpha-beta pruning which will always produce an exact answer. It also seems to have (mainly academic) uses outside the scope of this thesis as it could be used for solving other two-player zero-sum games.

5.5 Iterative deepening

It should be noted that we do not run alpha-beta search directly: instead, the search is called by an algorithm called *iterative deepening*: for a provided maximum search depth, alpha-beta search is called repeatedly, with each run incrementing the search depth until it reaches the depth limit. This way, we obtain some useful information during the forward analysis, such as the candidate move that is being considered and the heuristical evaluation score at every search depth.

5.6 Quiescence search

The last optimization of alpha-beta search we implemented is a solution to what is called the *horizon effect*: a situation in which the search determines that a certain move is optimal although the deeper consequences of this move are not beneficial. The depth limit is the cause of this problem: the search can be unable to expand far enough into the search tree to see these deeper consequences. To reduce this problem, a special function called *quiescence search* is called instead of the evaluation function when the depth limit is

reached. This function expands capture sequences at frontier nodes nonetheless, to get a better estimation of their value.

5.7 Proof-number search

Besides alpha-beta search, we have also implemented a variant of proof-number search [3] called depth-first proof-number search [7] as proof-tree search algorithm. The main difference between alpha-beta pruning and proof-number search is that where alpha-beta pruning uses an evaluation function to determine the visiting order of the nodes, proof-number search looks at the shape of the search tree. For all nodes, we keep track of proof and disproof numbers. These numbers represent lower bounds for the number of nodes that have to be visited in order to determine its game-theoretical value (this is called proving a node). The node chosen for expansion is always the node with the lowest proof number.

A downside of plain proof-number search is that it uses a lot of memory, as it has to store the whole search tree. Depth-first proof-number search is, as the name implies, a depth-first variant of proof-number search that tries to solve this problem. It also uses iterative deepening and makes use of transposition tables to store and retrieve the value of nodes. This has the added benefit of handling transpositions efficiently, like the transposition tables we used in alpha-beta pruning.

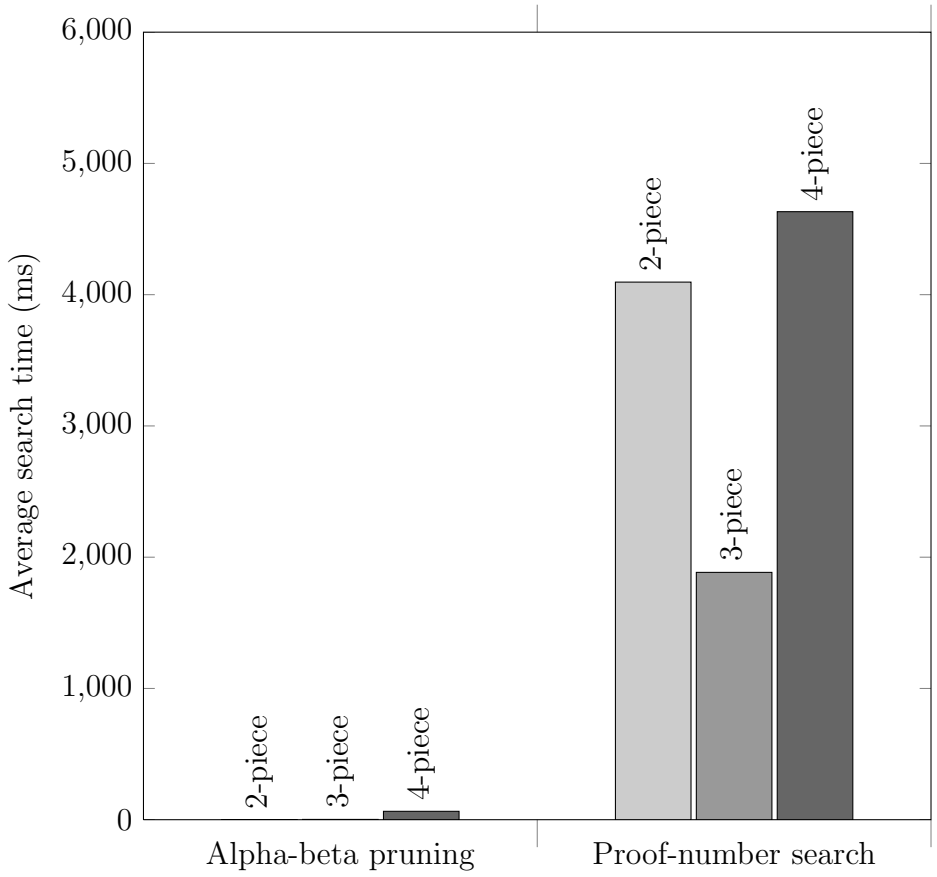

Figure 5: Comparison between the average alpha-beta search time and the average proof-number search time.

However, on a test set of Jungle Checkers positions we were able to solve, alpha-beta search performed a lot better than depth-first proof-number search. The results of the test run are displayed in Figure 5. The test set consists of randomly selected 2-, 3-, and 4-piece positions, 100 of each, none of which are draws. As can be seen proof-number search performed severely worse than alpha-beta pruning. The reason for this is that depth-first proof-number search suffers from a looping problem in domains where repetitions occur [5]: it is possible that the search enters a situation in which, for a certain amount of time, only a specific cycle of nodes are expanded repeatedly.

There is also another aspect that makes proof-number search less suitable

for Jungle Checkers. As for proof-number search the most important factor for determining the visiting order of nodes is the shape of the search tree, proof-number search should work best on games where the shape of different subtrees varies greatly; this is mainly the case in games where the play of one player can limit the number of options of the opponent [1]. Lines of Action is an example of such a game that has been used a lot within the context of proof-number search, but Checkers and Chess are also good examples of such games. In Jungle Checkers, situations like these do not occur very often, apart from the capturing of a piece of the opponent.

It also seems that for proof-number search, 3-piece positions are significantly easier to solve than 2-piece positions. We think the reason for this is that 3-piece positions have, on average, a shorter distance to win than 2-piece positions due to one player often having a significant advantage over the other player. In the test set, a 3-piece win was found on an average depth of 5.08 while a 4-piece win was found on an average depth of 6.75.

6 Endgame databases

To prevent the forward search from having to search too deep, *endgame databases* are implemented. In the case of Jungle Checkers, we define an *endgame* as a position in which only a few pieces still reside on the board. An endgame database stores, for every possible legal game position, if the game will result in either a win, a loss or a draw, provided that both players play optimally. These databases are then probed into by alpha-beta search, so that we can determine the game-theoretical value of these positions directly. The result is that big parts of the search tree can now be pruned.

Our tools can also generate more augmented versions of the endgame databases that include for each position the best move and the shortest number of moves leading to a win or the best counter play (if applicable). This might prove useful for future research.

6.1 Storage requirements

The endgame database that we construct consists of 2 bits per game position which denote the result of the corresponding position (win, loss or draw).

The positions are stored implicitly: a special function calculates the database index of the result, given a game position. This function is injective, but not surjective: there are indexes that do not have a corresponding position associated with them. The result is some internal overhead in the endgame database.

Besides this, we only store the positions in which white is the active player: the situations where black is the active player can be obtained from the databases by inverting the whole game situation, making use of the horizontal symmetry of the board.

The indexing function is defined as follows:

$$\sum_{i=0}^{\texttt{EGTB}-1} (\texttt{location}(\texttt{piece}[i]) \times \texttt{BOARD}^i \times \texttt{STORAGE}) + \texttt{conf} \times \texttt{BOARD}^{\texttt{EGTB}} \times \texttt{STORAGE}$$

in which:

- `EGTB` is a constant representing the number of pieces remaining on the board per position in this specific endgame database.
- `BOARD` is a constant representing the number of squares on the board ($7 \times 7 = 49$).
- `STORAGE` is a constant representing the number of bits we need to store all necessary information about a position (2).
- `piece[EGTB]` is an array that consists of the pieces remaining on the board, sorted in a specific order (white rat, black rat, white dog, black dog, white tiger, black tiger, white elephant, black elephant).
- `location(piece)` is a function that returns an index number corresponding to the board square `piece` is located on. The index of square `A7` is 0, the index of square `B7` is 1, and so forth. The dens also get an index number. The range of this function is thus 0–48.
- `conf` is an integer representing an index number corresponding to the combination of pieces remaining on the board. The combinations are all `EGTB`-subsets with at least one piece of each player on the board. These are sorted in lexicographical order, with an ordered set being {white rat, black rat, white dog, black dog, white tiger, black tiger, white elephant, black elephant}. The first index number is 0.

This indexing function leads to the storage requirements of our endgame database listed in Table 2. The second column denotes the number of positions that need to be stored: these are calculated in Section 3.2.1 (although these numbers are divided by two since we only store positions in which white is the active player). The third column denotes the size we would need to store this many positions in the hypothetical case that the hashing function would be perfect (2 bits per position). The fourth column denotes the actual necessary storage including the overhead the hashing function produces. This size can be calculated with the formula $\texttt{num\_combinations} \times 49^{\texttt{EGTB}} \times \texttt{STORAGE}$, in which `num_combinations` is the number of possible combinations of pieces, as calculated in Section 3.2.1.

Pieces remaining	**Positions stored**	**Size without overhead**	**Actual size**
2	34,592	8.6 KB	9.4 KB
3	4,669,920	1.2 MB	1.3 MB
4	291,091,680	72.8 MB	93.5 MB
5	10,308,070,080	2.6 GB	3.6 GB
6	216,469,471,680	54.1 GB	90.2 GB
7	2,535,785,239,680	633.9 GB	1.2 TB
8	12,678,926,198,400	3.2 TB	7.6 TB

Table 2: Storage requirements of the endgame database.

We have used this technique successfully to fit a complete 5-piece endgame database in 3.6 gigabyte of storage.

6.2 Retrograde analysis

Endgame databases are built using *retrograde analysis* [12]. The retrograde analysis algorithm uses a dynamic programming approach and works as follows:

1. Build an array in which every element corresponds to a game position.
2. Iterate over this array: for every element i.e. game position:
   (a) Generate all possible moves and for every possible move:

i. Determine if this move leads to a position which is already marked as a win. If so, mark the current position as winning.
ii. If not, determine if this move leads to a position which is not marked yet. If so, the current position could be a draw.

(b) If no possible moves lead to a win or a possible draw, mark the position as losing.

3. Repeat step 2 until no single marking changes in the whole array iteration. The unmarked positions are now certainly draws.

With this approach we have currently generated a 5-piece endgame database. In the nearby future we might even succeed in generating a 6-piece endgame database: our tools support this already.

6.3 Verification

We want to be as certain as possible that our endgame databases are correct. Therefore, we have written a tool which verifies the generated databases. It does this by calling alpha-beta search for each entry in the database and comparing the results of this search with the information that is contained in the database.

For wins and losses, this is a fairly trivial process. However, alpha-beta search is not able to check if a certain position is a draw. Therefore, for positions that result in a draw according to the endgame database, alpha-beta search is called with a very large initial search depth (127). If no certain win or loss is found by this search, we assume that the position is indeed a draw.

There is one more problem with this approach: the process is very time-consuming. We have verified the 2- and 3-piece databases, but verifying the 4- and 5-piece databases would be too time-consuming (although it is — in theory — possible since the process is highly parallelizable). However, since our approach and the used tools are the same for all the databases, we would expect no problems when performing such a verification.

7 Conclusions and Future Work

We have tried to determine the game-theoretical value of the begin position of Jungle Checkers with alpha-beta pruning (using a 5-piece endgame database and an evaluation function that is given in the appendix). We have visited more than 300,000,000,000 nodes (although these are not necessarily unique nodes: transposition table misses can occur which will result in calculating the value of certain nodes multiple times). As this search has not produced a result yet, it seems that finding the solution of Jungle Checkers falls outside the time and memory constraints of this research project.

We have, however, made a few interesting observations based on the 5-piece endgame databases and the analysis of some interesting positions:

- An endgame in which two pieces of the same strength remain on the board will never result in a draw. Either a piece can walk freely to the opponent's den, or a piece can force the other piece into a corner and capture it. In this last case, when the *Manhattan distance* (the sum of the horizontal and vertical components of the distance) between the pieces is odd, white can force black into a corner, whereas when the Manhattan distance between the pieces is even, black can force white into a corner.

- We have analyzed the 6-piece equal material situations with all the available material in the starting position. Out of 4 situations, 2 are winning for black (the games with both the elephants and rats on the board) and the other 2 are winning for white.

- We have also analyzed the 4-piece equal material situations with all the available material in the starting position. Out of 6 situations, 5 are winning for black (in 24 to 38 moves provided that both players play optimal). There is, however, an exception: with only the tigers and the elephants on the board, the game is a draw.

- If there would be no dens on the board, it would be impossible for white to win because when black always just steals the white move (an example of move stealing is `1.Tf2 2.tb6`, since pieces of equal strength are aligned diagonally against each other), every white capture is immediately followed by the corresponding black capture. Eventually

— unless white decides to stop capturing pieces — there will arise a situation where only a white and a black piece of equal strength remain on the board. A capture of equal material can only take place on the center square of the board because this is the only square such pieces can meet each other when taking move stealing into account. Considering white always arrives at this square first, it would thus be impossible for white to ever capture the last black piece. Therefore, the game would at least be a draw for black.

For now, Jungle Checkers remains only partially solved. There are a number of possible explanations for this. The first possibility is that our analyzing engine has not visited enough nodes yet to prove a win or a loss for the starting position. Generating a larger endgame database could perhaps significantly reduce the forward analyzing time in the future, although the amount of memory required for storing these bigger databases will not be insignificant. For storing the game-theoretical value (win, lose or draw) of all the 6-piece endgames we would need 90 GB of memory. This means that we also need to compute on a machine with more available main memory (all of our current experiments are run on a machine that has 16 GB of main memory).

Another possibility is that Jungle Checkers is a draw: to prove a draw, a very large part of the search space has to be visited, way larger than the nodes that we have evaluated right now. If this is the case, making our forward analysis parallelizable is probably a necessary step. This can also be regarded as future work.

References

[1] Allis, L.V., Searching for Solutions in Games and Artificial Intelligence, Ph.D. thesis, Maastricht University, 1994.

[2] Allis, L.V., van den Herik, H.J. and I.S. Herschberg, Which Games will Survive, *Proceedings of Heuristic Programming in Artificial Intelligence*, 232–243, 1991.

[3] Allis, L.V., van der Meulen, M. and van den Herik, H.J., Proof-Number Search, *Technical Reports in Computer Science*, CS91-01, 1991.

[4] Breuker, D., Memory versus Search in Games, Ph.D. thesis, Maastricht University, 1998.

[5] Kishimoto, A., Dealing with Infinite Loops, Underestimation, and Overestimation of Depth-First Proof-Number Search, *Proceedings of AAAI Conference on Artificial Intelligence*, 108–113, 2010.

[6] Kjeldsen, T.H., John von Neumanns Conception of the Minimax Theorem: A Journey Through Different Mathematical Contexts, *Archive for History of Exact Sciences*, 56(1):39–68, 2001.

[7] Nagai, A., A New Depth-First-Search Algorithm for AND/OR Trees, M.Sc. thesis, The University of Tokyo, 1999.

[8] van Rijn, J.N. and Vis, J.K., Complexity and Retrograde Analysis of the Game Dou Shou Qi, *Proceedings of 25th Benelux Conference on Artificial Intelligence*, 239–246, 2013.

[9] van Rijn, J.N. and Vis, J.K., Solving Jungle Checkers (project proposal), `www.liacs.nl/~snijssen/bachelorklas-2013-2014/jungle.pdf`, accessed April, 2014.

[10] Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P. and Sutphen, S., Checkers is Solved, *Science*, 317(5844):1518–1522, 2007.

[11] Schaeffer, J. and Lake, R., Solving the Game of Checkers, *Games of No Chance*, 29 119–133, 1996.

[12] Thompson, K., Retrograde Analysis of Certain Endgames, *ICCA Journal*, 9.3 131–139, 1986.

[13] UNSW School of Computer Science and Engineering, Animal Checkers, `www.cse.unsw.edu.au/~cs3411/07s1/hw3/`, accessed April, 2014.

[14] Zobrist, A.L., A New Hashing Method with Application for Game Playing, *ICCA Journal*, 13.2 69–73, 1970.

A Appendix

A.1 Implementation

We have limited ourselves here to the implementation of the most important parts of the forward analyzing engine: alpha-beta search, proof-number search, the evaluation function for alpha-beta search, the transposition table implementation for alpha-beta search and some helper functions, like the move sorting (alpha-beta search only) and the endgame database probing.

```
1  // ****************************************************************
2  // (C) Copyright 2014 Leiden Institute of Advanced Computer Science
3  // Universiteit Leiden
4  // All Rights Reserved
5  // ****************************************************************
6  // Jungle Checkers - a bachelor project by Bas van Boven
7  // ****************************************************************
8  // FILE:
9  //   alphabeta.cc
10 // DESCRIPTION:
11 //   Alpha-beta search method, makes use of transposition tables and
12 //   depth correction is also implemented.
13 // ****************************************************************
14
15 #include "../core/move.h"
16 #include "../core/position.h"
17 #include "../interface/retrograde_index.h"
18 #include "search.h"
19 #include "transposition.h"
20
21 #include <iostream>
22 using namespace std;
23
24 // alphabeta pruning
25 static int alphabeta(Position &position,
26                      int const depth,
27                      int       alpha,
28                      int       beta)
29 {
30
31   // save the original alpha value
32   int alphaOrig = alpha;
33
34   // take a look in the transposition table
35   Entry &entry = probe_tt(position.key());
36
37   // if a matching entry of sufficient depth has been found
38   if (entry.key == position.key() && entry.data.depth >= depth) {
39
40     // EXACT data type
41     if (entry.data.type == EXACT) {
42       return entry.data.value;
```

```
43      }
44      // LOWER data type
45      else if (entry.data.type == LOWER) {
46        if (entry.data.value > alpha) {
47          alpha = entry.data.value;
48        }
49      }
50      // UPPER data type
51      else if (entry.data.type == UPPER) {
52        if (entry.data.value < beta) {
53          beta = entry.data.value;
54        }
55      }
56
57      // if alpha >= beta now holds, entry.data.value is the optimal result
58      if (alpha >= beta) {
59        return entry.data.value;
60      }
61
62    }
63
64    // probe the EGTB if possible
65    if (USE_EGTB && pieces_board(position) == EGTB_PIECES) {
66      return probe_egtb(position);
67    }
68
69    // if won or we are in a leaf node, return appropriate score
70    if (position.is_won()) {
71      leaf_count++;
72      int const value = WIN;
73      return value;
74    }
75    if (depth == 0) {
76      leaf_count++;
77      // int const value = evaluate(position);
78      int const value = quiescence(position);
79      return value;
80    }
81
82    // start preparing for move sweep: generate moves
83    Move moves[MAX_MOVES];
84    int const count = position.generate_moves(moves);
85    sort_moves(moves, count, position.key());
86    int best = -INFINITY;
87    // move from transposition table is move 0
88    int move_tt = 0;
89
90    // if a stalemate is reached, return draw evaluation
91    if (count == 0) {
92      leaf_count++;
93      int const value = DRAW_STALEMATE;
94      return value;
95    }
96
97    // move sweep
98    for (int i = 0; i < count; ++i) {
99      // try move
```

```
100       position.do_move(moves[i]);
101       int value = -alphabeta(position, depth - 1, -beta, -alpha);
102       position.undo_move(moves[i]);
103       // if a win is detected deeper in the search, increment the distance
104       // to the win by one ply to prefer shorter wins
105       if (value > (-WIN-256)) {
106         // > 16127
107         value--;
108       }
109       else if (value < (WIN+256)) {
110         // < -16127
111         value++;
112       }
113       // determine new value & bounds
114       if (value > best) {
115         best = value;
116       }
117       if (value > alpha) {
118         alpha = value;
119         move_tt = i;
120         if (alpha >= beta) {
121           break;
122         }
123       }
124     }
125
126     // store found move in transposition table
127     if (best <= alphaOrig) {
128       entry = Entry(position.key(), depth, UPPER, best, &moves[move_tt]);
129     }
130     else if (best >= beta) {
131       entry = Entry(position.key(), depth, LOWER, best, &moves[move_tt]);
132     }
133     else {
134       entry = Entry(position.key(), depth, EXACT, best, &moves[move_tt]);
135     }
136
137     // return best score
138     return best;
139
140   } // alphabeta
141
142   // alphabeta pruning for the root of the tree
143   // we can safely ommit some checks and we don't want to take any
144   // chances with the best move as this is the one that we are going to
145   // play (theoretically it could always be retrieved from the
146   // transposition tables)
147   int alphabeta_root(Position &position,
148                      int const depth,
149                      int       alpha,
150                      int const beta,
151                      Move     &move)
152   {
153
154     // save the original alpha value
155     int alphaOrig = alpha;
156
```

```
157   // probe the EGTB if possible
158   if (USE_EGTB && pieces_board(position) == EGTB_PIECES) {
159     return probe_egtb(position);
160   }
161
162   // start preparing for move sweep: generate moves
163   Move moves[MAX_MOVES];
164   int const count = position.generate_moves(moves);
165   sort_moves(moves, count, position.key());
166   int best = -INFINITY;
167   // move from transposition table is move 0
168   int move_tt = 0;
169
170   // if a stalemate is reached, return draw evaluation
171   if (count == 0) {
172     int const value = DRAW_STALEMATE;
173     return value;
174   }
175
176   // move sweep
177   for (int i = 0; i < count; ++i) {
178     // try move
179     position.do_move(moves[i]);
180     int value = -alphabeta(position, depth - 1, -beta, -alpha);
181     position.undo_move(moves[i]);
182     // if a win is detected deeper in the search, increment the distance
183     // to the win by one ply to prefer shorter wins
184     /*if (value > (-WIN-256)) {
185       // > 16127
186       value--;
187     }
188     else if (value < (WIN+256)) {
189       // < -16127
190       value++;
191     }*/
192     // determine new value & bounds
193     if (value > best) {
194       best = value;
195     }
196     if (value > alpha) {
197       alpha = value;
198       move_tt = i;
199       if (alpha >= beta) {
200         break;
201       }
202     }
203   }
204
205   // probe the transposition table
206   Entry &entry = probe_tt(position.key());
207
208   // store found move in transposition table
209   move = moves[move_tt];
210   if (best <= alphaOrig) {
211     entry = Entry(position.key(), depth, UPPER, best, &moves[move_tt]);
212   }
213   else if (best >= beta) {
```

```
214       entry = Entry(position.key(), depth, LOWER, best, &moves[move_tt]);
215     }
216     else {
217       entry = Entry(position.key(), depth, EXACT, best, &moves[move_tt]);
218     }
219
220     // return best score
221     return best;
222
223   } // alphabeta_root
```

```
1   // ****************************************************************
2   // (C) Copyright 2014 Leiden Institute of Advanced Computer Science
3   // Universiteit Leiden
4   // All Rights Reserved
5   // ****************************************************************
6   // Jungle Checkers - a bachelor project by Bas van Boven
7   // ****************************************************************
8   // FILE:
9   //   proofnumber.cc
10  // DESCRIPTION:
11  //   An implementation of the Depth-first proof-number search
12  //   algorithm.
13  // ****************************************************************
14
15  #include "search.h"
16  #include "transposition.h"
17  #include "proofnumber.h"
18  #include "../core/move.h"
19  #include "../core/position.h"
20  #include "../core/move.h"
21  #include "../interface/retrograde_index.h"
22  #include <iostream>
23  using namespace std;
24
25  void PutInTT(PNnode node) {
26    // put node in transposition table
27
28    // probe the transposition table
29    PNEntry &entry = probe_pntt(node.position.key());
30    // store proof- and disproof numbers in transposition table
31    entry = PNEntry(node.position.key(), node.proof, node.disproof);
32  }
33
34  bool LookUpTT(PNnode node, uint64_t & proof, uint64_t & disproof) {
35    // update proof- and disproof numbers on transposition table hit
36
37    PNEntry &entry = probe_pntt(node.position.key());
38    //if a matching entry was found, update proof- and disproof numbers
39    if (entry.key == node.position.key()) {
40      proof = entry.proof;
41      disproof = entry.disproof;
42      return true;
43    }
44    // if no entry was found, return false
45    proof = 1;
46    disproof = 1;
```

```
47    return false;
48  }
49
50  bool IsTerminal (PNnode node) {
51    // determines if a node is terminal
52
53    if (node.position.is_won()) {
54      return true;
55    }
56    else if (USE_EGTB && pieces_board(node.position) == EGTB_PIECES) {
57      return true;
58    }
59    return false;
60  }
61
62  bool WinForCurrentNode(PNnode node) {
63    // true iff terminal node n is winning for the player corresponding to
64    // the current node
65
66    // if won, return appropriate score
67    if (node.position.is_won()) {
68      return false;
69    }
70
71    // probe the EGTB if possible
72    if (USE_EGTB && pieces_board(node.position) == EGTB_PIECES) {
73      int probevalue = probe_egtb(node.position);
74      if (probevalue == -WIN) {
75        return true;
76      }
77      else if (probevalue == WIN) {
78        return false;
79      }
80    }
81
82    // we should never be here
83    return false;
84  }
85
86  uint64_t ProofSum(PNnode node, Move moves[]) {
87    // calculate the sum of all the proof numbers of the children of node
88
89    uint64_t proof = 0;
90    for (int i = 0; i < node.numberOfChildren; i++) {
91      PNnode child = node;
92      child.position.do_move(moves[i]);
93      uint64_t lookupproof;
94      uint64_t lookupdisproof;
95      LookUpTT(child, lookupproof, lookupdisproof);
96      proof += lookupproof;
97    }
98    // prevent overflow
99    if (proof > PN_INFINITY) {
100     proof = PN_INFINITY;
101   }
102   return proof;
103 }
```

```
104
105 uint64_t DisproofMin(PNnode node, Move moves[]) {
106   // calculate the minimum of all the disproof numbers of the children of node
107
108   uint64_t disproof = PN_INFINITY;
109   for (int i = 0; i < node.numberOfChildren; i++) {
110     PNnode child = node;
111     child.position.do_move(moves[i]);
112     uint64_t lookupproof;
113     uint64_t lookupdisproof;
114     LookUpTT(child, lookupproof, lookupdisproof);
115     if (lookupdisproof < disproof) {
116       disproof = lookupdisproof;
117     }
118   }
119   return disproof;
120 }
121
122 void GenerateMoves(PNnode & node, PNnode (& children)[16]) {
123   // generates all children of node
124
125   Move moves[MAX_MOVES];
126   node.numberOfChildren = node.position.generate_moves(moves);
127   for (int i = 0; i < node.numberOfChildren; i++) {
128     node.position.do_move(moves[i]);
129     children[i].position = node.position;
130     node.position.undo_move(moves[i]);
131     children[i].proof = 1;
132     children[i].disproof = 1;
133   }
134 }
135
136 PNnode SelectChild(PNnode node, Move moves[], uint64_t & proof_c, uint64_t & disproof_2) {
137   // select the most promising child
138
139   PNnode best;
140   proof_c = PN_INFINITY;
141   uint64_t disproof_c = PN_INFINITY;
142   for (int i = 0; i < node.numberOfChildren; i++) {
143     PNnode child = node;
144     child.position.do_move(moves[i]);
145     uint64_t proof = 1;
146     uint64_t disproof = 1;
147     LookUpTT(child, proof, disproof);
148     // store the smallest and second smallest disproof in disproof_c and disproof_2
149     if (disproof < disproof_c) {
150       best = child;
151       disproof_2 = disproof_c;
152       proof_c = proof;
153       disproof_c = disproof;
154     }
155     else if (disproof < disproof_2) {
156       disproof_2 = disproof;
157     }
158     if (proof == PN_INFINITY) {
159       return best;
160     }
```

```
161     }
162     return best;
163   }
164
165   void MID(PNnode & node) {
166     // iterative deepening at each node
167
168     // TT lookup
169     uint64_t proof = 1;
170     uint64_t disproof = 1;
171     LookUpTT(node, proof, disproof);
172     if(proof >= node.proof || disproof >= node.disproof) {
173       // exceed thresholds
174       node.proof = proof;
175       node.disproof = disproof;
176       return;
177     }
178
179     // terminal node
180     if (IsTerminal(node)) {
181       if(WinForCurrentNode(node)) {
182         node.proof = 0;
183         node.disproof = PN_INFINITY;
184       }
185       else {
186         node.proof = PN_INFINITY;
187         node.disproof = 0;
188       }
189       PutInTT(node);
190       return;
191     }
192
193     // generate all moves
194     Move moves[MAX_MOVES];
195     node.numberOfChildren = node.position.generate_moves(moves);
196
197     // store larger proof and disproof numbers to detect repetitions
198     PutInTT(node);
199
200     // iterative deepening
201     while (node.proof > DisproofMin(node, moves) && node.disproof > ProofSum(node, moves)) {
202       uint64_t proof_c;
203       uint64_t disproof_2 = PN_INFINITY;
204       PNnode child = SelectChild(node,moves,proof_c,disproof_2);
205       // update thresholds
206       child.proof = node.disproof + proof_c - ProofSum(node, moves);
207       if (child.proof > PN_INFINITY) {child.proof = PN_INFINITY;}
208       if (node.proof > (disproof_2+1)) {
209         child.disproof = disproof_2 + 1;
210         if (child.disproof > PN_INFINITY) {child.disproof = PN_INFINITY;}
211       }
212       else {
213         child.disproof = node.proof;
214       }
215       MID(child);
216     }
217
```

```
218     // store search results
219     node.proof = DisproofMin(node, moves);
220     node.disproof = ProofSum(node, moves);
221     PutInTT(node);
222   }
223
224   int DFPN(Position & start) {
225     // set up for the root node
226
227     PNnode node;
228     node.position = start;
229     node.proof = PN_INFINITY;
230     node.disproof = PN_INFINITY;
231     MID(node);
232     if (node.disproof == PN_INFINITY) {
233       return 1;
234     }
235     else {
236       return 0;
237     }
238   }
```

```
1   // ****************************************************************
2   // (C) Copyright 2014 Leiden Institute of Advanced Computer Science
3   // Universiteit Leiden
4   // All Rights Reserved
5   // ****************************************************************
6   // Jungle Checkers - a bachelor project by Bas van Boven
7   // ****************************************************************
8   // FILE:
9   //   evaluation.h
10  // DESCRIPTION:
11  //   Defines for each piece a static development score for each
12  //   location of the board and a negative score when a piece is
13  //   captured (static relative piece value).
14  // ****************************************************************
15
16  #if !defined(__evaluation_h__)
17  #define __evaluation_h__
18
19  #include "../core/position.h"
20
21  static int const DEVELOPMENT[8][50] =
22  {
23    {
24        12, 13, 50, 100, 50, 13, 12,
25        11, 12, 13,  50, 13, 12, 11,
26        10, 11, 12,  13, 12, 11, 10,
27         9, 10, 11,  12, 11, 10,  9,
28         8,  9, 10,  11, 10,  9,  8,
29         7,  8,  9,  10,  9,  8,  7,
30         6,  7,  8,   9,  8,  7,  6,
31      -500
32    }, // WHITE_RAT
33    {
34        6,  7,  8,   9,  8,  7,  6,
35        7,  8,  9,  10,  9,  8,  7,
```

```
36          8,  9, 10,  11, 10,  9,  8,
37          9, 10, 11,  12, 11, 10,  9,
38         10, 11, 12,  13, 12, 11, 10,
39         11, 12, 13,  50, 13, 12, 11,
40         12, 13, 50, 100, 50, 13, 12,
41       -500
42     }, // BLACK_RAT
43     {
44         12, 13, 50, 100, 50, 13, 12,
45         11, 12, 13,  50, 13, 12, 11,
46         10, 11, 12,  13, 12, 11, 10,
47          9, 10, 11,  12, 11, 10,  9,
48          8,  9, 10,  11, 10,  9,  8,
49          7,  8,  9,  10,  9,  8,  7,
50          6,  7,  8,   9,  8,  7,  6,
51       -400
52     }, // WHITE_DOG
53     {
54          6,  7,  8,   9,  8,  7,  6,
55          7,  8,  9,  10,  9,  8,  7,
56          8,  9, 10,  11, 10,  9,  8,
57          9, 10, 11,  12, 11, 10,  9,
58         10, 11, 12,  13, 12, 11, 10,
59         11, 12, 13,  50, 13, 12, 11,
60         12, 13, 50, 100, 50, 13, 12,
61       -400
62     }, // BLACK_DOG
63     {
64         12, 13, 50, 100, 50, 13, 12,
65         11, 12, 13,  50, 13, 12, 11,
66         10, 11, 12,  13, 12, 11, 10,
67          9, 10, 11,  12, 11, 10,  9,
68          8,  9, 10,  11, 10,  9,  8,
69          7,  8,  9,  10,  9,  8,  7,
70          6,  7,  8,   9,  8,  7,  6,
71       -600
72     }, // WHITE_TIGER
73     {
74          6,  7,  8,   9,  8,  7,  6,
75          7,  8,  9,  10,  9,  8,  7,
76          8,  9, 10,  11, 10,  9,  8,
77          9, 10, 11,  12, 11, 10,  9,
78         10, 11, 12,  13, 12, 11, 10,
79         11, 12, 13,  50, 13, 12, 11,
80         12, 13, 50, 100, 50, 13, 12,
81       -600
82     }, // BLACK_TIGER
83     {
84         12, 13, 50, 100, 50, 13, 12,
85         11, 12, 13,  50, 13, 12, 11,
86         10, 11, 12,  13, 12, 11, 10,
87          9, 10, 11,  12, 11, 10,  9,
88          8,  9, 10,  11, 10,  9,  8,
89          7,  8,  9,  10,  9,  8,  7,
90          6,  7,  8,   9,  8,  7,  6,
91         -700
92     }, // WHITE_ELEPHANT
```

```
93    {
94         6,  7,  8,   9,  8,  7,  6,
95         7,  8,  9,  10,  9,  8,  7,
96         8,  9, 10,  11, 10,  9,  8,
97         9, 10, 11,  12, 11, 10,  9,
98        10, 11, 12,  13, 12, 11, 10,
99        11, 12, 13,  50, 13, 12, 11,
100       12, 13, 50, 100, 50, 13, 12,
101       -700
102   } // BLACK_ELEPHANT
103 }; // DEVELOPMENT
104
105 static inline int evaluate(Position const &position)
106 {
107   static int const factor[2] = {-1, 1};
108   return factor[position.is_white_turn()] *
109          (DEVELOPMENT[WHITE_RAT][position.piece(WHITE_RAT)] +
110          -DEVELOPMENT[BLACK_RAT][position.piece(BLACK_RAT)] +
111           DEVELOPMENT[WHITE_DOG][position.piece(WHITE_DOG)] +
112          -DEVELOPMENT[BLACK_DOG][position.piece(BLACK_DOG)] +
113           DEVELOPMENT[WHITE_TIGER][position.piece(WHITE_TIGER)] +
114          -DEVELOPMENT[BLACK_TIGER][position.piece(BLACK_TIGER)] +
115           DEVELOPMENT[WHITE_ELEPHANT][position.piece(WHITE_ELEPHANT)] +
116          -DEVELOPMENT[BLACK_ELEPHANT][position.piece(BLACK_ELEPHANT)]);
117 } // evaluate
118
119 #endif
```

```
1  // ****************************************************************
2  // (C) Copyright 2014 Leiden Institute of Advanced Computer Science
3  // Universiteit Leiden
4  // All Rights Reserved
5  // ****************************************************************
6  // Jungle Checkers - a bachelor project by Bas van Boven
7  // ****************************************************************
8  // FILE:
9  //   transposition.h
10 // DESCRIPTION:
11 //   Defines the transposition table structure for alpha-beta search.
12 // ****************************************************************
13
14 #if !defined(__transposition_h__)
15 #define __transposition_h__
16
17 #include "engine_types.h"
18
19 // TT entry types
20 static int8_t const LOWER = 0;
21 static int8_t const UPPER = 1;
22 static int8_t const EXACT = 2;
23
24 // Invalid depth
25 static int8_t const INVALID = -127;
26
27 // TT_SIZE *must* be a power of two
28 static int const TT_SIZE = 1 << 27;
29 static int const TT_MASK = TT_SIZE - 1;
```

```
30
31
32  class Entry
33  {
34    struct Data
35    {
36      int8_t  depth;
37      int8_t  type;
38      uint8_t piece;
39      uint8_t to;
40      int32_t value;
41    }; // Data
42
43    public:
44      explicit Entry(void);
45      explicit Entry(uint64_t const key,
46                     int const      depth,
47                     int const      type,
48                     int const      value,
49                     Move const*    move = 0);
50
51
52      uint64_t key;
53      union
54      {
55        uint64_t raw_data;
56        Data     data;
57      }; // data
58
59  }; // Entry
60
61
62  // The actual transposition table (in search.cc)
63  extern Entry tt[TT_SIZE];
64
65
66  inline Entry::Entry(void)
67  {
68    // intentionally empty
69  } // Entry::Entry
70
71  inline Entry::Entry(uint64_t const key,
72                      int const      depth,
73                      int const      type,
74                      int const      value,
75                      Move const*    move)
76  {
77    data.depth = static_cast<int8_t>(depth);
78    data.type = static_cast<int8_t>(type);
79    if (move != 0)
80    {
81      data.piece = static_cast<uint8_t>(move->piece);
82      data.to = static_cast<uint8_t>(move->to);
83    } // ifh
84    data.value = static_cast<int32_t>(value);
85    this->key = key;
86  } // Entry::Entry
```

```
87
88
89  static inline void clear_tt(void)
90  {
91    for (int i = 0; i < TT_SIZE; ++i)
92    {
93      tt[i].data.depth = INVALID;
94    } // for
95  } // clear_tt
96
97  static inline Entry &probe_tt(uint64_t const key)
98  {
99    return tt[key & TT_MASK];
100 } // probe_tt
101
102 #endif
```

```
1   // ****************************************************************
2   // (C) Copyright 2014 Leiden Institute of Advanced Computer Science
3   // Universiteit Leiden
4   // All Rights Reserved
5   // ****************************************************************
6   // Jungle Checkers - a bachelor project by Bas van Boven
7   // ****************************************************************
8   // FILE:
9   //   search.cc
10  // DESCRIPTION:
11  //   Search functions; mainly move sorting and endgame database
12  //   probing.
13  // ****************************************************************
14
15  #include "../interface/core_extensions.h"
16  #include "search.h"
17
18  uint64_t node_count  = 0;
19  uint64_t leaf_count  = 0;
20
21  Entry tt[TT_SIZE];
22
23  unsigned char * cbin;
24
25  int principal_variation(Position &position,
26                          int const depth,
27                          int const value,
28                          int       index,
29                          Move      variation[])
30  {
31    Entry &entry = probe_tt(position.key());
32    if (entry.key == position.key() &&
33        entry.data.depth == depth &&
34        entry.data.value == value)
35    {
36      Move move(entry.data.piece, CAPTURED, entry.data.to, NONE);
37      if (validate_move(position, move))
38      {
39        variation[index] = move;
40        position.do_move(move);
```

```
41        index = principal_variation(position,
42                                    depth - 1,
43                                    -value,
44                                    index + 1,
45                                    variation);
46        position.undo_move(move);
47      } // if
48    } // if
49    return index;
50  } // principal_variation
51
52  static inline int evaluate_move(Move const    &move,
53                                  uint64_t const key)
54  {
55    Entry &entry = probe_tt(key);
56    //static_cast<void>(key);
57    return -DEVELOPMENT[move.piece][move.from] +
58           DEVELOPMENT[move.piece][move.to] +
59           (move.enemy != NONE) * -DEVELOPMENT[move.enemy][CAPTURED] +
60           ((entry.key == key) * 4 * entry.data.value);
61  } // evaluate_move
62
63  void sort_moves(Move           moves[MAX_MOVES],
64                  int const      count,
65                  uint64_t const key)
66  {
67    int values[count];
68    values[0] = evaluate_move(moves[0], key);
69    for (int i = 1; i < count; ++i)
70    {
71      Move const temp = moves[i];
72      int const pivot = values[i] = evaluate_move(moves[i], key);
73      int j = i - 1;
74      int k = i;
75      while (j >= 0 && values[j] < pivot)
76      {
77        moves[k] = moves[j];
78        values[k] = values[j];
79        --j;
80        --k;
81      } // while
82      moves[k] = temp;
83      values[k] = pivot;
84    } // for
85    return;
86  } // sort_moves
87
88  int probe_egtb(Position &position) {
89    // fetches evaluation score (win, loss or draw) from EGTB
90
91    // invert position if neccessary
92    Position position_probe = position;
93    if (!position.is_white_turn()) {
94      position_probe.clear();
95      for(uint64_t i = 0; i < NUM_PIECES; i+=2) {
96        // get location from old pieces
97        uint32_t p1 = position.piece(i+1);
```

```
98        uint32_t p2 = position.piece(i);
99        // determine file and rank from old pieces
100       uint32_t p1_file = p1 % 7;
101       uint32_t p1_rank = p1 / 7;
102       uint32_t p2_file = p2 % 7;
103       uint32_t p2_rank = p2 / 7;
104       // inverse ranks
105       p1_rank = 6 - p1_rank;
106       p2_rank = 6 - p2_rank;
107       // build new location
108       p1 = p1_rank * 7 + p1_file;
109       p2 = p2_rank * 7 + p2_file;
110       // set new position
111       if(position.piece(i+1) != 49) {position_probe.set_piece(i,p1);}
112       if(position.piece(i) != 49) {position_probe.set_piece(i+1,p2);}
113     }
114   }
115   // fetch answer by using byte- and bit-indexes
116   int value = 0;
117   uint64_t index = position_index(position_probe);
118   uint64_t byte_index = index / 8;
119   unsigned char retrieve = cbin[byte_index];
120   unsigned char bit_index = (7 - (index % 8));
121   unsigned char bit_mask = 1 << bit_index;
122   bool calculated = (bool)((retrieve & bit_mask) >> bit_index);
123   bit_index = (7 - ((index % 8) + 1));
124   bit_mask = 1 << bit_index;
125   bool won = (bool)((retrieve & bit_mask) >> bit_index);
126   if (calculated && won) {value = WIN;}
127   else if (calculated && !won) {value = -WIN;}
128   // return value
129   return value;
130 } // egtb_fetch
131
132 void egtb_init() {
133   // fetches reads EGTB from file
134
135   // open file
136   FILE * pFile;
137   long lSize;
138   size_t result;
139   pFile = fopen (EGTB_LOCATION,"rb");
140   if (pFile==NULL) {cout << "ERROR: EGTB not found. Quitting..." << endl; exit (1);}
141   // obtain file size:
142   fseek (pFile , 0 , SEEK_END);
143   lSize = ftell (pFile);
144   rewind (pFile);
145   // allocate memory to contain the whole file:
146   cbin = (unsigned char*) malloc (sizeof(char)*lSize);
147   if (cbin == NULL) {fputs ("Memory error",stderr); exit (2);}
148   // copy the file into the cbin:
149   result = fread (cbin,1,lSize,pFile);
150   if ((unsigned)result != lSize) {fputs ("Reading error",stderr); exit (3);}
151   // terminate
152   fclose (pFile);
153   cerr << "EGTB load succeeded!" << endl << endl;
154 } // egtb_init
```

A.2 Search space of Jungle Checkers

We have also included Table 3, an augmented version of Table 1. It includes the base count of finished and unfinished positions (stalemates and invalid positions included), as well as the number of invalid positions and the total of valid positions.

Pieces	**Unreachable unfinished positions**	**Unfinished positions**	**Unreachable won positions**	**Won positions (no material)**	**Won positions (den)**	**Stalemates**	**Total reachable positions**
1	0	0	0	376	0	0	376
2	0	69,184	0	25,944	1,504	0	96,632
3	768	9,339,024	0	778,320	311,328	48	10,428,720
4	70,656	582,108,480	3,168	8,561,520	26,459,712	4,224	617,133,936
5	1,598,784	20,614,450,432	366,336	0	1,198,246,464	90,944	21,812,787,840
6	428,544	432,938,509,384	19,080,576	0	30,905,129,664	5,432	463,843,644,480
7	423,936	5,071,570,053,116	534,007,296	0	432,404,936,064	2,308	5,503,974,991,488
8	198,144	12,678,926,000,016	2,736,744,768	0	1,265,155,875,072	240	13,944,081,875,328
Total	2,720,832	18,204,640,529,636	3,290,202,144	9,366,160	1,729,690,959,808	103,196	19,934,340,958,800

Table 3: Overview of the Jungle Checkers search space, with the unreachable positions and the total position count included.