Universiteit Leiden
Computer Science

Retrograde Analysis and Proof Number Search

Applied to Jungle Checkers

Name: Michiel Sebastiaan Vos

Date: 24/02/2016

1st supervisor: Prof. Dr. A. (Aske) Plaat

2nd supervisor: Dr. W.A. (Walter) Kosters

MASTER'S THESIS

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

Retrograde Analysis and Proof Number Search Applied to Jungle Checkers

Michiel Sebastiaan Vos

February 24, 2016

Abstract

For the game of Jungle Checkers, we compute the seven-piece endgame tablebase, with a parallel variation of Retrograde Analysis. In order to do so, we create an efficient ranking algorithm, that takes into account mirrored positions and the prevention of invalid positions, which results in a maximum usage of 64 GB main memory. We apply Proof Number Search to Jungle Checkers to compute the game theoretical game of the start position and we encounter the Graph History Interaction (GHI) problem when used with a Transposition Table. Instead of using solutions for the GHI problem because they are not perfect, we prevented the GHI problem from happening. Unfortunately, this hampers the usage of the Transposition Table and as a result the search space becomes too big to traverse. We present an idea to compute the eight-piece endgame tablebase, in order to solve the game.

Contents

Chapter 1

Introduction

Jungle Checkers [29] is a two player zero-sum board game. Both players have each four pieces of different strength under control. The goal of the game is to reach the opponent's den with one of the pieces. The game has similarities with Stratego and checkers, but with fewer pieces.

We are interested in what would be the outcome of the game, if two players are playing perfectly. Perfect play means a player does not make mistakes and will do moves that lead to the best outcome of the game that is possible. It does not mean a perfect player will always win, because it is possible the opponent forces the player to do a non-winning move, because there are no better alternatives. When we know the outcome, we say the game of Jungle Checkers is weakly solved.

Good human players can see several moves ahead and can reason the best move, while in theory, computers can see many steps ahead and simulate all possible games, and find the perfect move. In practice, this is difficult because we do not have unlimited space and time.

Retrograde Analysis (RA) and Proof Number Search (PNS) are techniques that will minimize the run time of solving the game. These techniques will prevent doing duplicate calculations and will find an efficient order of calculations.

This thesis is written for the Master of Computer Science at Leiden University, supervised by Aske Plaat and Walter Kosters.

Problem Statement

The goal is to weakly solve the game. We use Proof Number Search to traverse the search tree and to look several moves ahead in order to determine a perfect move. To reduce the depth of the search tree, we calculate with Retrograde Analysis in advance the outcomes of the games with only seven pieces left on the board. This brings us to the problem statement: Can we solve Jungle Checkers with the seven-piece endgame tablebase and Proof Number Search?

1.1 Research questions

We have the following research questions:

What is an efficient ranking algorithm?

To answer the main research question, we need to enumerate all positions and have an efficient function that can map numbers to positions and visa versa. Trivial algorithms do not take into account transpositions, mirrored positions and prevention of invalid position.

How to generate the seven-piece endgame tablebase?

We solve all positions with seven-pieces on the board and store the game theoretical values in a tablebase. We use a parallel variation of Retrograde Analysis to speed up the computation and one bit per position to save memory.

How to apply Proof Number Search and resolve the Graph History Interaction problem?

While using Proof Number Search to solve the start position of the game, we encountered the Graph History Interaction (GHI) problem. The GHI problem arises when the game theoretical value of a position is stored in a transposition table and the value is incorrectly reused, because the history to the position is not taken into account. Our solution is a workaround and prevents the GHI from happening.

1.2 Contributions

We made the following contributions:

- a new ranking algorithm,
- the seven-piece endgame tablebase,
- a workaround for the Graph History Interaction problem,
- an idea for generating the eight-piece tablebase on a computer cluster.

1.3 Thesis overview

Chapter 2 describes the rules of the game and Chapter 3 the related work. In Chapter 4 a game rule is changed in order to get a binary objective. Chapter 5 describes the ranking algorithm which is used in the next two chapters, Chapter 6 the backward search with Retrograde Analysis and Chapter 7 the forward search with Proof Number Search. We conclude in Chapter 8 and give several pointers to future research.

Chapter 2

Game Rules

The game is played by two players, player white and player black. They control several pieces placed on a board. The objective of the game for a player is to win the game by entering the *den* of the opponent or capture all the pieces of the opponent.

2.1 Board

The board has seven rows and seven columns, dividing it into 49 cells. Each column has a letter assigned from a to g, from left to right. Each row has an index assigned from 1 to 7, from bottom to top. The label of the cell is the combination of the letter of the column and the index of the row, as can be seen in Figure 2.1.
There are two special cells. Cell `d1` is the white *den* and cell `d7` is the black *den*. They are denoted with a `#`.

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

Figure 2.1: Start position with all pieces.

2.2 Pieces

There are four kinds of pieces and they differ in strength. The pieces are (in order of strength from lowest to highest) rat (`r`), dog (`d`), tiger (`t`) and elephant (`e`). Both players have one of each, so in total there are eight pieces. The color of the pieces denotes the owner of the piece. An upper case character means the owner of the piece is white and a lower case character means the owner of the piece is black. See Figure 2.1 for the board with the start locations of the pieces.

2.3 Moves

A player can move one of his pieces to a horizontally or vertically adjacent cell. If the cell is already occupied by a piece of the opponent the piece will be *captured*. This is only possible if the strength of the piece of the player is equal to or higher than the strength of the piece of the opponent, or if the rat captures the elephant. A captured piece is removed from the board.
It is not possible to capture a piece of your own, enter your own *den* or skip a turn. The white player does the first move.

2.4 Draw

If the player to move does not have any possible moves, the game ends in a draw. With perfect play, stalemate never occurs [4]. A position is a repetition, if the position was already visited before during the game. We say a repetition is a draw, because there is no development in the game.

Chapter 3

Related Work

In 2013 Van Rijn and Vis [18, 19] wrote about the creation of an endgame tablebase for the game of Jungle Chess, as first step towards solving the game. They succeeded in generating an endgame tablebase for four pieces, where they found some interesting patterns, but solving the game was out of reach.

In 2014 Van Boven continued their work on endgame tablebases for his bachelor thesis [4], but focused on the simplified version of the game called Jungle Checkers. He succeeded in generating a 5-piece endgame tablebase, but visiting 300 billion nodes in the search tree was not sufficient to find the game-theoretical value. Generating the 6-piece endgame tablebase fell outside the scope of his research project, due to time and memory constraints.

In 2015 we generated the 6-piece endgame tablebase. The tablebase was generated in multiple parts, which required less main memory during the computation. In order to do so, a better ranking algorithm was used and several implementation improvements were achieved.

The following sections describe several essential algorithms and give pointers to more related work.

3.1 Ranking

For Retrograde Analysis (RA) and Proof Number Search (PNS) we need a bijective function that maps a position to a number. Zobrist [31] hashes a position to a number, but cannot do the inverse and is non-injective. A ranking [21] function maps a permutation or combination to a rank and an unranking function maps a rank to a permutation or combination. For Jungle Checkers a position is a permutation of locations where the pieces are on the board, given the pieces that are on the board and the player to move. Other terminologies include encoding and decoding, numbering permutations, enumeration schemes and integer representations.

In [3], the authors calculate the rank of a position for the game of Backgammon. A slow method is enumerating all position until the specific position is

matched. The number of non-matched positions so far equals the rank of the matched position. A fast method is calculating the number of combinations of checkers of the previous positions by using binomial coefficients. This works for Backgammon, because all checkers are equal and the permutation of the checkers on a point (the so called triangles on the board) does not matter.

The Canton pairing functions can combine two non-negative numbers into one. Using this function recursively it is possible to combine multiple numbers into a rank. The inverse is also possible. A disadvantage is that the function needs multiple math operators, with the square root operator being relatively slow.

Myrvold et al. [15] made a linear time algorithm, which recursively swaps values of a permutation. The disadvantage is that the permutations are not in lexicographic order.

The Gödel function is used by Thompson [28] for Chess and by Bal [2] for Awari.

The Lehmer code uses the factorial number system. A decimal number can be encoded to a factorial, which can be encoded to a permutation. We use this for the (un)ranking function of Jungle Checkers and it will be explained in detail in Chapter 5.

3.2 Retrograde Analysis

Retrograde Analysis (RA) [28] is a technique to find the game-theoretical value of the initial position by reasoning backwards from all terminal positions to the initial position of a game.

There are three phases:

1. All positions with black to move and black is mated, are marked as win for white.

2. For each position marked in phase 3 of the previous iteration, or phase 1 if this is the first time, mark the parent as a win for white.

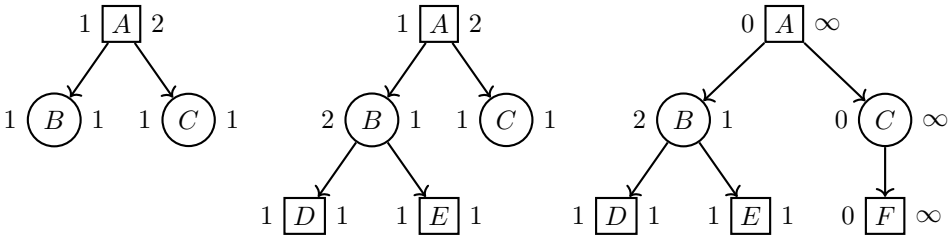
3. For each position marked in phase 2, if all siblings are wins for white, mark the parent as a win for white.

Repeat phase 2 and 3 until no more positions are marked. The positions not marked as win for white are illegal, loss for white or draws. The game-theoretical values of all positions are now known.

3.3 Proof Number Search

Proof Number Search [1] is a search algorithm for proving or disproving trees. By definition, a tree is *proven* when white is the winner, *disproven* if white it is not the winner and *unproven* when the winner it not known yet. These three definitions are also applicable to a node in the tree. Every node has two

numbers: proof and disproof. These values contain the amount of nodes that must be (dis)proven in order to (dis)proof the node. Initially they are both 1, because optimistically in order to (dis)proof the node, only the node itself must be (dis)proven. A proven node has proof number 0 and disproof number ∞. A disproof node has disproof number 0 and proof number ∞. A node where white is to move is called an OR node and where black is to move an AND node.



(a) Root node A is expanded.

(b) The (dis)proof numbers of node B and C are equal in (a). Ties are arbitrary chosen, we select B as MPN.

(c) Node C is the MPN, because the proof number of C (1) is less than B's (2) in (b). Node F is a terminal position.

Figure 3.1: The first three iterations of an example search with PNS.

Initially the tree consists only of the root node. There are three phases in an iteration:

selecting The Most Proving (or Promising [14]) Node (MPN) is selected. From the root a child is selected and this child selects a new child. This is repeated until a terminal[1] node is reached. An OR node selects a child with the smallest proof number and an AND node selects a child with the smallest disproof number.

expanding For the selected node, all child nodes are generated. Each child is evaluated. If a child's position is terminal and thus its node is (dis)proven, the (dis)proof numbers are set.

updating When the selected node is expanded, the (dis)proof numbers of the node and all its ancestors until the root must be updated. For an OR node, the proof number equals the minimal proof number of its children and the disproof number equals the sum of the disproof numbers of its children. For an AND node, the proof number equals the sum of the disproof numbers of its children and the disproof number equals the minimal disproof number of its children.

[1] A terminal node is a node without children. A terminal position is a position where the game is ended and the value is known. We use this definition also for positions with less than eight pieces, because thoses values are known for Jungle Checkers.

These three phases are repeated until the root has a (dis)proof number of zero. An advantage of this search algorithm is that it does not need an evaluation function.

We give an example of PNS, illustrated with Figure 3.1. In the first iteration the root node A is selected as MPN and is expanded. It has disproof number two, because in order to disprove node A, at least two other nodes (nodes B and C) must be disproven. In the second iteration node B is selected as MPN. It has proof number two, because in order to proof node B, at least two other nodes (nodes D and E) most be proven. In the third iteration node C is selected as MPN. The only child of C is node F, which is a terminal position and a proven node by game rules. The (dis)proof numbers of the ancestors are updated and the algorithm stops, because the root's proof number is zero.

Van den Herik and Winands [9] review several enhancements and variants of PNS, for example Depth-First PNS [16].

3.4 Transposition Table

In a search tree, a node is a transposition node of another node, if they are both related to the same position. The moves (or perhaps the ordering) leading to these nodes are different. To prevent solving the transposition of an already solved node, values of solved nodes are stored in a hash table. Before solving a node, the transposition table is checked to see if the node is not already solved and if it is, its value is used. Zobrist [31] is a popular hashing function for game positions and ranking functions are also suitable. Collisions are possible with non-injective hashing functions and can be solved by using a chained hash table or replacement strategies.

3.5 Graph History Interaction problem

In the game of chess, there is a rule called the threefold repetition rule. If a position occurs for the third time in a game, a player can call a draw. This means not only the position is needed to decide whether it is terminal and who the winner is, but also the previous positions before.

When a search algorithm is combined with a transposition table for a game where the history leading to a position is important, the Graph History Interaction (GHI) problem [17] can arise. Positions affected by repetitions are possibly wrongly stored in the transposition table and the values are wrongly reused in different places in the search tree where the position causing the repetition is not in the path.

3.5.1 An example

An example of the GHI problem is illustrated in Figure 3.2, taken from Kishimoto [10], which is a search graph for a checkmating puzzle. Suppose position E is visited during the search, with the path leading to E is A $\rightarrow$ B $\rightarrow$ E $\rightarrow$ H

→ E. Position E is a repetition and a loss is stored (in the transposition table) for position H, because in H, the only move does not lead to a checkmate. Next, position B is visited and a loss is stored, because child D is a terminal loss position and the value (loss) of child E is retrieved from the transposition table. At last, position F is visited and since H is stored as a loss, F is also stored as a loss, which in turn results in a loss for C. Position A is now considered a loss for A, because both children are losses. This is wrong, because the path A → C → F → H → E → G yields a win. This example shows, values stored in the transposition table cannot be blindly trusted, when the history can affect the value of a position.

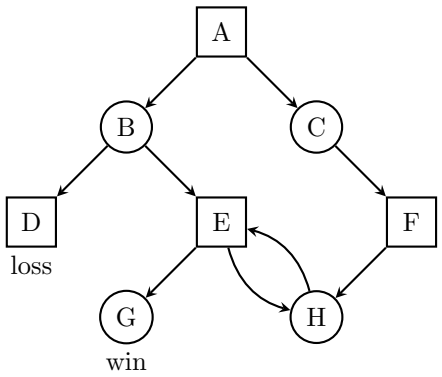


Figure 3.2: A graph where the GHI problem can occure.

3.5.2 Solutions

Kishimoto et al [11] made an overview of PNS algorithms and solutions to the GHI problem.

Campbell [7] describes two cases where the GHI problem can arise. Three relevant conclusions of his paper are:

1. He has a solution for the Draw-First case and not for the Draw-Last case.
2. He says "GHI does not appear to occur relatively infrequently."
3. The key in avoiding most occurrences of GHI appears to be iterative deepening.

For Jungle Checkers, a solution for the Draw-last case is required. In order to prove all positions, all "relatively infrequently" GHI problems must be solved and cannot be ignored. Seo [27] made an iterative deepening depth-first proof number search algorithm.

Breuker [6] made the Base Twin Algorithm and compared it to the algorithm of Schijf [26]. In the experiment both algorithms did not solve all chessmate

problems, mainly because Breuker limited the search. The limit was used to restrict the computation time, like in tournaments, and to restrict the memory usage, which was near the hardware limit.

Kishimoto [10] notes an issue with BTA. It does not work for the "current-player-loss scenario". Kishimoto stores for his algorithm the history of a position in the transposition table. The algorithm is compared to Nagai's solution [16]. Again, not all problems are solved.

In this research area, a "solution" to the GHI problem, does not mean that the algorithm can solve all problems or positions. For solving a game, the unsolved positions are most interesting and problematic. For computer players (bots), the solutions are useful, because perfect moves are not necessary. Some programs are ignoring the GHI problem and are using the computing power for searching for more positions instead of calculating correct values.

Chapter 4

Binary Value

In most zero-sum two player games the first player or the second player can win and some games can also end in a draw. In some cases it is desired to only have two possible outcomes, i.e., the game can only end in a win for black or white and not in a draw. In chessmate puzzles a player is only interested in being able to win and thus solve the puzzle. Some search algorithms can only prove and disprove trees [5]. Three outcomes can be stored in two bits and two outcomes can be stored in one bit. A value with only two possible outcomes is called a binary (game theoretical) value. Other terminologies are bounded value, worst(best)-case scenario, at-least-draw [13], binary question [25] and binary objective.

In order to make the outcome of a game binary in Jungle Checkers, the rules must change. There are two variations of the game. The first variant lets white win when the value in the original game would be a draw, white has the *advantage*. The second variant has the same logic, except white and black are interchanged. See Table 4.1 for an overview.

When white wins, even when white has the disadvantage, white would also win in the original game. When white wins with the advantage, the original game could end in a win for white or a draw. When the last case happens, the search must be repeated, but now with the advantage for black. If white now loses, the original game would be a draw and if white would win, the original would also be a win for white.

Losing with the advantage and winning with the disadvantage is impossible.

		Advantage	
		White	**Black**
Value	**White**	Draw or White	White
	Black	Black	Draw or Black
	Value with advantage	Value without advantage	

Table 4.1: The values with and without advantage.

In this thesis, white has the advantage.

Lincke [13] also used one bit to store the value, but works differently. One bit is stored in memory with the value as a win or an unknown. Two bits were stored on disk to store the value as win, loss, draw or unknown. The bits on the disks were only accessed when the value was not a win.

Chapter 5

Ranking Positions

For the algorithms in the following chapters, we need a function that maps a position to a number and visa versa. This function will be used many times and it is therefore important to be efficient. What is an efficient ranking algorithm?

5.1 Introduction

In the game tree, traversed by the search algorithm, every node is a position. When we need to compare nodes with each other, the rank is used as identifier. For Retrograde Analysis (RA), a tablebase is used to store the values of all the positions. The rank of a position is used as index, so we know which bits in the tablebase belongs to which positions. We can iterate over all positions by iterating over all ranks and use the unranking function to map the rank to a position.

For RA it is important to minimize the size of the tablebase, so it can fit into main memory. Solving only the positions that must be kept together in main memory for the algorithm to work can reduce the memory consumption. For example, in order to solve a position with n pieces, the positions with more than n pieces and less than $n-1$ are not required. We can divide the tablebase into subtablebases and generate the ones that are dependent of each other in sequential order and the ones that are independent of each other in parallel.

In Chapter 6 the generation of the subtablebases is described. This chapter describes how the ranking function works and how it is adapted to divide the subtablebases.

5.1.1 Divisions

The following divisions where made by Lake for checkers [12] and by Thompson for chess [28].

Number of pieces The $(n-1)$-piece tablebase can be generated before the n-piece tablebase. When all $(n-1)$-piece tablebases are generated, all n-piece tablebases can be generated in parallel.

For checkers it is possible to create more divisions. The tablebases can be divided by:

1. The number of black and white pieces, because it is not possible to uncapture a piece.
2. The number of kings and checkers. Checkers can only promote to kings and kings cannot degrade to checkers.
3. The leading (most advanced) checker, for each player. Checkers can only move forwards. Generating the subtablebase with all positions with the most leading checkers on row r can only begin when the subtablebase with all positions with the most leading checker on $r-1$ is completed. This idea can also applied to the second until nth most leading checker.

Symmetry In chess there are positions that are equal by symmetry. When the value of a position is calculated, it is not necessary to compute the values of positions that are symmetric to this one. There are two kinds of symmetry:

1. There is vertical symmetry. If the first column of the board is swapped with the last column and the second column with the second-last, etc., until all columns are swapped. The value of this position would not be different from the position before the swapping. We call such positions, mirrored positions.
2. There is symmetry by color. Swapping the colors of the pieces and the rows with the same method as above. with the columns and changing the player to move will result in a symmetrically equal position. We call this duplicate positions.

For Jungle Checkers we use the divisions by number of pieces and by both symmetries. Unfortunately, in Jungle Checkers moving backwards is not forbidden and the division by leading checker is not possible. Wu [30] has more divisions specific for chinese chess.

5.2 Method

This subsection describes the mapping from a rank to a position and visa versa.

Let n be the number of pieces and $r \in \{0, 1, \ldots, k!/((k-n)!-1)\}$, where k is the number of different locations a piece can be placed on the board. For Jungle Checkers, k equals 47, because seven rows multiplied by seven columns, minus the two dens produces 47 locations. Positions with a piece on the den are not needed to be enumerated, List `pieces` contains the pieces that are on the board, sorted in strength from low to high and within a strength by color,

with white first. List `loc` contains the locations of the pieces. The labeling of the locations is shown in Table 5.1 and the list with these locations is denoted by $L = (0, 1, \ldots, k-1)$. It holds that $\texttt{loc}_i \in L$ for $i \in \{0, 1, \ldots, n-1\}$ and $\texttt{loc}_i \neq \texttt{loc}_j$ if $i \neq j$.

5.2.1 Unranking

We start with describing the unranking function, which is divided into two parts. First, we convert the `rank` to a temporal list `loc_index` and secondly we convert the `loc_index` to the desired `loc` list. The Python function in Listing 1 receives a rank and returns the list `loc_index`. It holds that $\texttt{loc\_index}_i \in \{0, 1, \ldots, k-i-1\}$. If we use `loc_index` as the location list, multiple pieces can be placed on one location and piece i cannot be placed on a location from the set $\{k-i, k-1+1, \ldots, k-1\}$. This is not what we want and this problem is solved by converting `loc_index` to `loc` in Listing 2. In fact, `loc_index` is the factorial number representation of `loc`.

Let A_i be the set of locations of the pieces already placed on the board beforing placing piece i, so $A_i = \{\texttt{loc}_0, \texttt{loc}_1, \ldots, \texttt{loc}_{i-1}\}$, and let F_i be the list of locations with the locations of A_i omitted, denoted by $L \backslash A_i$, and let $F_i[j]$ be the j-th location of F_i. Now, $\texttt{loc}_i = F_i[\texttt{loc\_index}_i]$.

7	0	7	14	#	26	33	40
6	1	8	15	21	27	34	41
5	2	9	16	22	28	35	42
4	3	10	17	23	29	36	43
3	4	11	18	24	30	37	44
2	5	12	19	25	31	38	45
1	6	13	20	#	32	39	46
	a	b	c	d	e	f	g

Table 5.1: Order of locations.

Listing 1 `rank_to_index`

```
1 def rank_to_index(n, k, rank):
2   remainder = rank
3   loc_index = [0]*n # list of length n with zero values
4   for p in range(0, n):
5     loc_index[p] = remainder % (k-p)
6     remainder = remainder / (k-p)
7   return loc_index
```

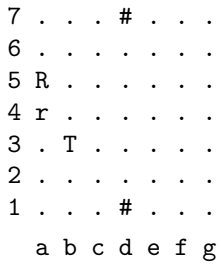


Figure 5.1: The lists belonging to this position with rank 19554 are `pieces` = (`R`, `r`, `T`), `loc` = $(2, 3, 11)$ and `loc_index` = $(2, 2, 9)$.

5.2.2 Ranking

The functions in Listing 1 and Listing 2 are respectively the inverse of the functions in Listing 3 and Listing 4.

This paragraph gives an example to clarify how the (un)ranking function works. Suppose we have the position given in Figure 5.1. When using function `rank_to_index` with rank 19554 we get `loc_index` = (2, 2, 9). A value v of $\texttt{loc\_index}_i$ means that the location of piece i is the v-th free location, accounting for the order of Table 5.1 and considering the previous pieces are already placed on the board. For the zeroth piece it means that $\texttt{loc}_0 = \texttt{loc\_index}_0 = 2$. For the first piece, $\texttt{loc\_index}_1$ equals 2, which means we must place the first piece on the second free location. Location 0 and 1 are free, but location 2 is already occupied by the zeroth piece, so the second free location is 3, hence $\texttt{loc}_1$ equals 3. The third piece has index 9 and the 9th free location is location 11, because location 2 and 3 are already occupied by piece zero and one. The result is the `loc` list $(2, 3, 11)$. The function `loc_to_index` does the inverse. The rank is the decimal representation of the factorial number representation and is calculated with the `loc_index` list as follows: $\texttt{rank} = 2 \cdot 47!/47! + 2 \cdot 47!/46! + 9 \cdot 47!/45! = 19554$.

Listing 2 `index_to_location`

```
1 def index_to_location(n, location_index):
2   loc = loc_index
3   for p in range(0, n):
4     for q in reversed(range(0, p)): # from the previous
5                                     # to the first
6       if loc_index[q] <= loc[p]
7         loc[p] += 1
8   return loc
```

5.2.3 Implementation

This subsection describes several implementations details.

Mirror

A board with the first piece on column `e`, `f` or `g`, is called a mirrored board. See Figure 5.2 for the mirrored board of the starting position (Figure 2.1). The theoretical game values of the original and the mirrored board are always equal, so it is unnecessary to calculate the values for both. When encountering a mirrored board, we mirror the board and continue with this board. This symmetry saves us the cost of calculating and storing nearly 45% of all positions. When enumerating all positions which are not mirrors, the first piece can only be put on the 26 locations, which are on a column on the left side of the board or on the center column.

If the first piece is in the middle column, we can apply the same procedure again, but now with the second piece. The idea works until the sixth piece, because there are only five locations between the two dens. The savings are decreasing and do not outweigh the disadvantage of slower computation, so we only use it for the first piece.

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

Figure 5.2: Mirror of the start position.

Duplicate

There are more positions which do not need to be calculated, since they can be derived from other positions. If a board of a position is vertically flipped, the ownership of all pieces are given to its opponent (black pieces become white,

Listing 3 `index_to_rank`

```
1 def index_to_rank(n, k, loc_index):
2   rank = 0
3   for p in range(0, n):
4     rank += loc_index[p] * factorial(k) / factorial(k-p)
```

white pieces become black) and the player to move is changed. The value of the new *duplicate* position is different from the *original* position before the changes. The answer of the question "Would the opponent win if he was in my situation?" is equal to the answer of the question "Can I win?".

A position is duplicate if black has less pieces or pieces lower in strength. When a duplicate position is encountered, the original position is retrieved and its rank is used. In Figure 5.3 the board of the duplicate of the start position is shown.

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

Figure 5.3: Duplicate board of the starting position.

Factorial

When $n = 7$, the function `index_to_rank` in Listing 3 uses 630 multiplications and divisions to rank only one position. Calculating $47!/(47-p)!$ is rather inefficient and can be rewritten as $\prod_{i=0}^{p-1}(47-i)$, which results in 21 multiplications.

The answers of the production function can be precalculated, stored and reused. In the implementation the whole function now uses $n-1$ additions, n multiplications and n lookups.

Increment rank

When enumerating all positions, the next position can be calculated by incrementing the rank and then using the unranking function. The time complexity

Listing 4 `location_to_index`

```
1 def location_to_index(n, location):
2   loc_index = loc
3
4   for p in reversed(range(0, n)): # from the last to the first
5     for q in reversed(range(0, p)): # from the previous
6                                     # to the first
7       if loc[p] >= loc[q]:
8         loc_index[p] -= 1
```

is $O(n^2)$, because of the complexity of the unranking function.

Another way, the one we have implemented, is to increment the `loc_index` list and use the `index_to_location` (Listing 2) function, which is $O(n)$. The procedure (Listing 5) looks similar as binary addition with carrying bits.

Listing 5 `increment_index`

```
1 def increment_index(n, k, loc_index):
2   loc_index[0] += 1
3   p = 1
4   while p < n and loc_index[p] == k-p:
5     loc_index[p] = 0
6     p += 1
```

5.3 Result

The algorithms in the next chapters spend most of their run time on using the ranking and unranking function. An efficient algorithm is critical. The ranking algorithm is efficient, because:

1. The complexity is dependent on the number of pieces on the board and not on the number of different positions.
2. Values of positions that can be derived from others and invalid positions are taken into account.
3. In combination with RA, subtablebases can be generated in parallel.
4. The implementation uses only n multiplications for ranking.

Chapter 6

Backward Search

With the (un)ranking algorithm from Chapter 5 explained, the generation of the seven-piece endgame tablebase can begin. When finished, the forward search can consult the endgame tablebase at the moment a piece is captured in order to save time by retrieving the theoretical game value, instead of expanding the search tree. This chapter will elaborate on how to generate the seven-piece endgame tablebase.

6.1 Introduction

When using forward search we start at the initial position of the game and try to find a sequence of perfect moves to a terminal position to get the game theoretical value. With backward search we start from all terminal positions (their values are determined by game rules), and calculate the values of the positions after undoing a move. This procedure can be repeated for the new valued positions and is known as Retrograde Analysis (RA).

The result of RA is an endgame tablebase, because in previous research these database were made for positions with a few pieces left on the board where the game is nearly finished.

In this chapter we show how to generate the seven-piece endgame tablebase. Besides the endgames, it also contains the positions in the middle games and all positions with one piece captured.

There are eight ways to select seven pieces out of eight pieces, which results in eight different sets of pieces, which we call material configurations. For each material configuration a seven-piece endgame subtablebase can be generated and the subtablebases can be generated in parallel, with the condition that the six-piece endgame tablebase is already generated. Four of the eight subtablebases contain duplicate positions and are not needed. The four relevant material configurations are `RrDdTtE`, `RrDdTEe`, `RrDTtEe` and `RDdTtEe`.

6.2 Method

Let `positions` be a list of size n containing all positions of a given material configuration and `changed` a Boolean which is set to true if at least one value is changed (and is initially set to true). The initial value of all positions is win for `WHITE`. At the start of each iteration (line 6), the Boolean `changed` is set to false. The unranking function is used to enumerate over all positions. When a value is changed to `BLACK`, it cannot change back and therefore positions with `BLACK` values are skipped. Positions with `WHITE` values are checked every iteration if the value can change to `BLACK`. This is not time-efficient, but it is space-efficient, because we save for each position a Boolean that indicates if a position is valued or not. There are two ways a value can change to `BLACK`:

1. It is black's turn and black can do a move to a position with a `BLACK` value where it is white's turn.

2. It is white's turn and white cannot do a move to a position with a `WHITE` value where it is black's turn.

After enumerating all positions, the iteration is finished. If no values have changed, the algorithm is finished and the values of all positions are known. Otherwise, the next iteration is started. The Python code of this method is shown in Listing 6.

Listing 6 `retrograde_analysis`

```
1  # n: number of positions
2  # value of positions are WHITE by default
3  def retrograde_analysis(n, positions):
4    changed = True
5    while changed:
6      changed = False
7      for rank in range(0, n):
8        p = positions[rank]
9        if p.value == WHITE:
10         if p.turn == BLACK:
11           if has_winning_move(p):
12             p.value = BLACK
13             changed = True
14         else p.turn == WHITE:
15           if not has_winning_move(p):
16             p.value = BLACK
17             changed = True
```

6.2.1 Parallelization

RA spends most of its computing time on ranking and unranking positions, which happens inside the `has_winning_move` function use in Listing 6. The procedure to rank a position is not dependent of other positions, which gives us a possibility to use parallelization, which could reduce the runtime.

In the for loop in line 7, all positions are enumerated, in sequential order. One way to use parallelization is to divide the positions and let each core, work on a division. We used the Intel® Xeon® Processor E5-2630 v3 [8], which has 8 physical cores and 8 virtual HyperThreading cores. Core $c \in \{0, 1, \ldots, 15\}$ works on division c which contains the positions with the rank in the range $\{c \cdot \frac{n}{16}, c \cdot \frac{n}{16} + 1, \ldots, (c+1) \cdot \frac{n}{16} - 1\}$. When the work of all divisions is done, the original for loop is finished and the iteration is ended.

For Jungle Checkers, this parallelization resulted in a reduction of the total run time, but in the worst-case scenario the algorithm could end in n (number of positions) iterations. To analyze the run time behaviour when scaling up from the sequential algorithm to the parallel version, first consider the case where only one core is used. Suppose every value is still `WHITE`, except the first position, which has value `BLACK`. The first iteration is started. The value of the first position is already `BLACK` and is skipped. Suppose the value of the p-th position becomes `BLACK`, because of the $(p-1)$-th position, until position $n-1$. The first iteration is finished and the second iteration does not change a value, because all positions are already `BLACK`. The algorithm finishes in two iterations.

In the next case, instead of using one core, we use parallelization to try to reduce the run time, by using n cores. In the worst-case scenario, the order of the cores finishing their work is from core $n-1$ to core 0, every iteration. This time, in the first iteration, only the value of the second position is changed. In every iteration i, only the value of position $i+1$ is changed, until iteration $n-1$. In this case, using parallelization did not result in a reduction of run time, because the sequential algorithm is finished in two iterations and the parallel version in n iterations.

6.3 Results

If a value of a position is changed (to `BLACK`) in iteration i with Retrograde Analysis, it means there is a position that needs i moves to reach a terminal position or a position with $n-1$ pieces, with a `BLACK` value. Figure 6.1 shows the percentage of positions that have a `BLACK` value, for each iterations, for each material configuration, where white had the *advantage*. For material configuration `RrDdTEe`, the algorithm is finished after 32 iterations, while material configuration `RrDTtEe` needs 44 iterations. After 10 iterations, the percentages are nearly converged and are close to their percentages of the last iteration, but there are still values changing and the algorithm must continue. The higher the strength of the (black) piece not on the board, the lower the percentage of positions with `BLACK` values. Although white has the *advantage* and a piece more,

the proportion between the positions with `BLACK` and `WHITE` values are not that different.

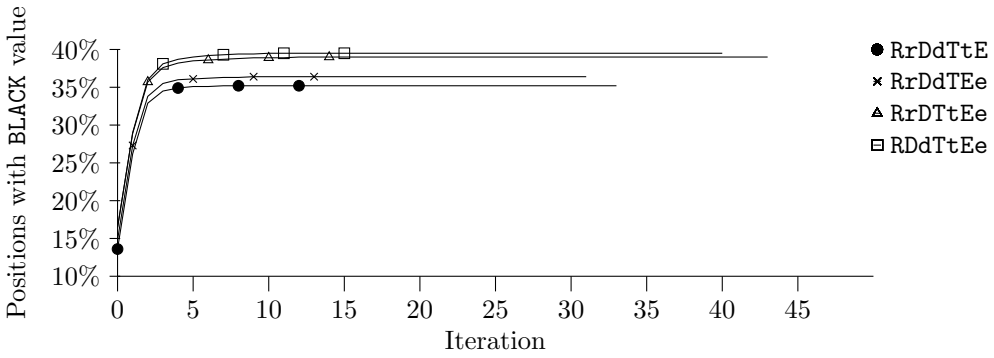


Figure 6.1: Percentage of postions with a `BLACK` value.

The complete seven-piece tablebase, together take more than 175 GB, which is exported to disk and is generated within 65 hours. The size is calculated as follows. There are $47{\cdot}46{\cdot}45{\cdot}44{\cdot}43{\cdot}42{\cdot}41 = 47!/(47-7)! \approx 3.2{\cdot}10^{11}$ possible ways to put seven pieces on a board. Due to vertical symmetry, we can ignore the positions with the first piece on the last three columns (or the last 21 locations), which makes $3.2 \cdot 10^{11} \cdot \frac{47-21}{47} \approx 1.8 \cdot 10^{11}$. This number must be multiplied by two, because we want to know the value of the positions when it is white's turn, but also for black's turn. For every position one bit is reserved for the value, which is why we need almost 44 GB main memory and another 16 GB to load the complete six-piece endgame tablebase. This fits nicely in the 64 GB main memory of a normal computing node from the so-called DAS-5 cluster. There are four material configuration, which makes $4 \cdot 44\,\text{GB} \approx 175\,\text{GB}$, the size of the complete seven-piece tablebase.

Chapter 7

Forward Search

RA is a backward approach which starts at the terminal positions and undoes moves until all positions are solved. In this chapter we start from the start position and do forward moves to promising positions that can solve the start position. Using the ranking algorithm from Section 5 and the seven-piece endgame tablebase from Section 6 we can speed up the search tree algorithm. In this chapter we apply the Proof Number Search (PNS) algorithm to Jungle Checker and we encounter the Graph History Interaction (GHI) problem.

7.1 Introduction

In a game search tree every node represents a position and every edge a move. Nodes are expanded in a certain order and the value is computed based on the values of their children. Initially, only the values of the terminal nodes are known, but during the search more and more values are being computed. When the value of the root is computed, the game is ultra weakly solved. For Jungle Checkers the values of positions with a piece captured are already computed and stored in the seven-piece endgame tablebase. The ranking algorithm is used to retrieve the related value of such positions.

We use PNS as search algorithm due to three reasons:

1. It does not need a knowledge based heuristic to guide the search [22]. A good heuristic function for Jungle Checkers is difficult to design, because we only need it when there are still eight piece on the board and with eight pieces on the board no player has an advantage based on the number or strength of pieces. Also, we are no expert players and it is hard to implement domain knowledge in an heuristic function.

2. It is a best-first search algorithm [9], where the search is guided by the structure of the tree, in order to traverse small subtrees first. In this way the root position is solved faster than when breadth-first or depth-first search is used.

3. Checkers was solved using Depth-First PNS [24] (DF-PNS). Instead of DF-PNS, we use the standard PNS, because DF-PNS suffers more from the GHI problem [9].

As discussed in Chapter 3, the known solutions for the GHI problem are not perfect and are not useful for solving games. Instead of solving the GHI problem, we detect when the GHI problem can arise and then choose to not use the Transposition Table (TT).

7.2 Method

The original PNS [11] algorithm is used in combination with a TT. A node (or position) is proven when the value is `WHITE` and disproven when the value is `BLACK`. The values of positions with a captured piece are retrieved from the seven-piece endgame tablebase, which were computed with backward search. Winning moves to terminal positions are preferred above moves to repeated positions, but not if it takes multiple moves to a terminal position and it only takes one move to a repeated position. The next subsections describe how and when the positions with computed values are stored into and retrieved from the transposition table.

Preventing the GHI problem

The GHI problem arises when the history of a position affects the value of a position and the value is wrongly stored into the TT. When the affected value is used for computing other values, those values are corrupted and also affected.

We prevent the GHI problem from happening by not storing values that are determined by the history of moves, i.e. the values of positions that are proven due to repetition are not stored. Repeated positions are draws and we remark that draws are winning for white, when white has the advantage, as described in Chapter 4. When a position is (dis)proven, but not by repetition, the value is stored in the TT, with the sequence of moves to reach the leaf node (a terminal position or a position with a captured piece). When a node is visited and its position is already in the TT, the path from the root to the node is compared with the sequence of moves to the leaf node, in order to check if there is a repeated position. If the path does not contain a repetition, the value from the TT is correct and can be used.

The next paragraphs describe in detail how the GHI problem is prevented.

Entry

An entry in the TT has four attributes: *rank*, *value*, *perfect move* and *next* entry. Since there are more positions than the size of the TT, collisions are possible. To still be able to save all positions, a chained hash table is used for the TT to deal with collisions and the attribute *next* points to the next entry if it exist. The attribute *perfect move* points to the entry with the position after the move,

unless the *perfect move* will lead to a terminal position, in which case it will be a null pointer. This will create a chain of moves, from a position to the leaf node. Not the history is stored, but the future.

There is also another kind of collision. There are entries with the same *rank* and hence the same index. However they have different *perfect moves*, resulting in different paths to a leaf node. If one entry cannot be used, because it will result in a repetition and thus a corrupted *value*, it could be that there is another entry, but with another path to a leaf node, without repetition.

Retrieving values

Nodes are evaluated as follows:

1. If the position is a terminal node, the *value* is determined by game rules.
2. If the position has a captured piece, the *value* is retrieved from the seven-piece endgame tablebase.
3. The *rank* is calculated and it is checked if there is an entry with the same index:
   (a) If the entry has the same *rank*.
      i. If there is no repetition in the path from the root via the node to a leaf node, by following the entries of the *perfect moves*, the *value* of the node is set with the *value* of the entry.
      ii. If there is a repetition, the *next* entry is checked.
   (b) If the entry has a different *rank*, the *next* entry is checked.
4. Otherwise, the *value* is still not determined or retrieved, the *value* stays unknown and the algorithm continues, with (dis)proof number 1 for this node.

Storing values

Suppose node m is proven, because it has no disproven children. When one of its children is proven due to repetition, it does not only affect the *value* of this child, but also the *value* of its parent, node m. Storing the *value* of m into the TT would result in corrupted *values* for other positions. Nodes with such values are called *affected*. This also applies when one of the children is proven due to an affected child. It also applies when proven and disproven are interchanged in the above description.

When a *value* is determined, it is only stored into the TT if the following conditions are met.

1. An entry with the same *rank*, *value* and *perfect move* does not exist.
2. The *value* of the position is not determined by a repetition.
3. The node is not affected by repetition.

When a node is (dis)proved, the children are removed from the tree, because they are not needed anymore and the allocated memory is released.

7.3 Results

For three pieces we computed the values of all positions and compared it with the values retrieved from the three-piece endgame tablebase. There were no differences found and we conclude the prevention of the GHI problem works.

Unfortunately, the search algorithm could not finish in our setup and the game theoretical value of the root position is still not known. The search could not continue, because the tree could not be expanded any further due to memory constraints on the current hardware.

Prior to termination there were approximately 10^9 nodes in the tree and 10^5 entries in the transposition table, with a negligible number of collisions. We expected a higher number of entries in the transposition table. It could be our prevention of the GHI problem, prevents the usage of the transposition table, because a lot of positions are affected by repetition.

Since there was a node found at depth 20 and the branching factor is 16, the number of different leaf nodes could in theory be $16^{20} \approx 10^{24}$. Even with the seven-piece endgame tablebase the search space is not reduced enough and we searched only a very small portion of the search space.

The ratio of the proof and disproof numbers gives an indication that the game theoretical value of the root will be a win for white, but there is no guarantee the ratio will not change. The development of the two numbers can be seen in Figure 7.1.

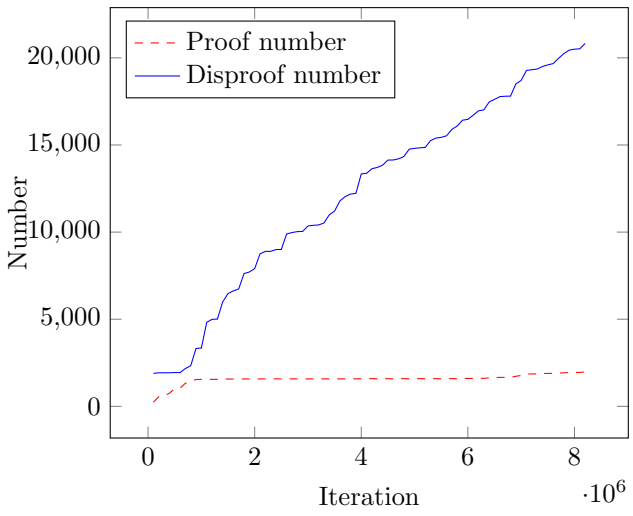


Figure 7.1: The development of the proof and disproof number of the root.

Chapter 8

Conclusion

Unfortunately, we did not succeeded in solving the game, but we made progress and are very close to reaching the goal. In the process we created a new efficient ranking algorithm, the seven-piece endgame tablebase and prevented the GHI problem from happening. In the next sections the research questions are listed and answered. We conclude with the problem statement and at last we give three ideas to solve the game for future research.

8.1 What is an efficient ranking algorithm?

The ranking algorithm for Jungle Checkers is efficient, because it takes into account mirrored positions, the prevention of invalid positions and it can be used for generating subtablebases in parallel.

8.2 How to generate the seven-piece endgame tablebase?

In combination with the ranking algorithm and a parallel variation of Retrograde Analysis, the seven-piece endgame tablebase is generated with only 64 GB of main memory. One subtablebase is generated in 44 iterations, which means there is at least one position, that turned out to be a win for black, after 44 moves.

8.3 How to apply Proof Number Search and resolve the Graph History Interaction problem?

The PNS algorithm can be applied to Jungle Checkers when a player has the advantage, which results in a binary objective. It can be combined with the

seven-piece endgame tablebase in order to retrieve a value of a position with a captured piece. Instead of solving the GHI problem, it is prevented by only storing and using values of position that are not determined or affected by repeated positions.

Problem Statement

Can we solve Jungle Checkers with the seven-piece endgame tablebase and Proof Number Search? The answer is no. The prevention of the GHI problem hampers the full usage of a Transposition Table, which resulted in not enough pruning of the search tree and prevented the desired reduction of the search space.

There are several reasons it is difficult to solve this game. The main problem, is that the rules do not forbid to move backwards, which can result in games with repeated positions, which yields a draw. The branching factor does not decrease after every move. The history to a position also affects the value of the position. The division of subtablebases cannot be extended. Solutions to the GHI problem are not perfect. The 44 iterations of RA for one subtablebase and the search tree depth of 22 indicate the game is played with slow progression and needs many moves to reach a terminal position, when both players are playing perfectly.

8.4 Future research

Since the game is not solved with the seven-piece endgame tablebase and Proof Number Search, this chapter describes several approaches towards solving Jungle Checkers as future research.

8.4.1 More memory and time

With Retrograde Analysis we succeeded in generating the seven-piece (endgame) tablebase. With more memory we can also generate the eight-piece tablebase.

For a board with seven pieces already placed, there are 40 locations left to place the eighth piece. One could think, the required main memory for the eight-piece tablebase is 40 times the required main memory of the seven-piece endgame tablebase.

For seven pieces a position is based on $(pieces, locations, turn)$, but for eight pieces the position is based on $(locations)$, because:

1. Without captured pieces, the eight pieces are known.
2. The turn can determined from the locations of the pieces by The Parity Problem [4].

The number of different positions can be divided by two. The required main memory is $26 \cdot 46!/40! \cdot 2 \cdot 40 \cdot 0.5$ bits ≈ 877 GB.

Generating a seven-piece subtablebase took approximately 41 times longer than generating a six-piece subtablebase. If we assume Retrograde Analysis does not need more iterations for the eighth-piece tablebase than the seven-piece tablebase, we can extrapolate the run time of generating the eight-piece tablebase, which will be estimated to be four months.

8.4.2 Changing rules

Changing the rules in such a way a piece cannot move vertically towards its own den has two advantages:

1. The branching factor decreases and pieces are moving faster towards each other, which will possibly lead to a less deep search tree before a piece is captured. The Graph History Interaction problem can still arise, so we are not sure if this advantage will lead to solving the game.

2. The subtablebases can be divided even further, based on the row of the piece vertically closest to the opponent's den, as is done for checkers [23]. This will lead to a much lower memory requirement and will make it possible to use Retrograde Analysis for generating the eight-piece tablebase.

It must be proven that changing the rule will not change the game theoretical value of the start position.

8.4.3 Cluster computing

We present an untested algorithm to generate the eight piece tablebase, where Retrograde Analysis is combined with the Transposition-Driven Scheduling [20] algorithm.

We use multiple computing nodes from a cluster to combine the memory in order to store the values of all positions with eight pieces. During the algorithm non-blocking messages are send and received by the nodes, which means the communication is only one way and there is no idle time because there is no need to wait for a response.

Suppose there are W computing nodes called workers and P positions with eight pieces. Each position has a game theoretical value and a counter. Each worker w_i stores the value and the counter of the positions with a rank in the set $\{i \cdot \frac{P}{W}, i \cdot \frac{P}{W} + 1, \ldots, (i+1) \cdot \frac{P}{W} - 1\}$ and is responsible for these positions.

Suppose position p is the parent of position q, meaning position p is the result after undoing a specific move from q and position q is the child of p. When position q is valuated, the worker of q called $w(q)$ can send a message to $w(p)$ with the rank of p and the value of q. The worker $w(p)$ can receive those messages.

The algorithm starts with letting the workers count the number of valid children for each position and store this number in the counter. This number is in the set $\{0, 16\}$ for Jungle Checkers and needs 5 bits to be stored.

Secondly, for all positions with seven pieces and all positions with eight pieces, where one piece is on one of the two dens, a message will be sent to their parents with the value retrieved from the seven-piece endgame tablebase or determined by game rules.

At last, the main part of the algorithm. When a worker receives a message with the rank of q and a value of p, it is checked if the value of position q can be determined, as described in Listing 7 and if that is the case, a message will be sent to all workers of all parents of q. A lot of messages will be sent and received.

The algorithms finishes when all counters of all positions are zero, but can be stopped when the value of the start position is determined. When the algorithm finishes, Jungle Checkers is solved.

Listing 7 `Retrograde Analysis with Transposition-Driven Scheduling`

```
# rank of position
# value of the child
def on_receive_message(rank, value):
  p = position[rank]
  if p.counter is not 0: # position it not yet valuated
    if winning(p, value): # child is winning for p
      p.counter = 0
      p.value = value
      send_messages(parents(p), value)
    else: # child is losing
      p.counter -= 1
      if p.counter is 0: # all children are losing
        p.value = value
        send_messages(parents(p), value)
```

Bibliography

[1] L.V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Rijksuniversiteit Limburg, 1994.

[2] H.E. Bal and L.V. Allis. Parallel Retrograde Analysis on a Distributed System. In *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, pages 73–73. IEEE, 1995.

[3] A. Benjamin and A.M. Ross. Enumerating Backgammon Positions: The Perfect hash. *INTERFACE*, 16(1):3–10, 1996.

[4] B. van Boven. Solving Jungle Checkers. Bachelor thesis, Leiden University, 2014.

[5] D.M. Breuker, L.V. Allis, and H.J. van den Herik. How to Mate: Applying Proof-Number Search. *Advances in Computer Chess*, 7:251–272, 1994.

[6] D.M. Breuker, H.J. van den Herik, J.W.H.M. Uiterwijk, and L.V. Allis. *A Solution to the GHI Problem for Best-First Search*. Springer, 1999.

[7] M. Campbell. The Graph-History Interaction: On Ignoring Position History. In *Proceedings of the 1985 ACM Annual Conference on The Range of Computing : Mid-80's Perspective: Mid-80's Perspective*, ACM '85, pages 278–280, New York, NY, USA, 1985. ACM.

[8] Dell. Intel® Xeon® Processor E5-2630 v3. `http://ark.intel.com/products/83356/Intel-Xeon-Processor-E5-2630-v3-20M-Cache-2_40-GHz`, 2014. [Online; accessed 2016-01-22].

[9] H.J. van den Herik and M.H.M. Winands. Proof-Number Search and its Variants. In *Oppositional Concepts in Computational Intelligence*, pages 91–118. Springer, 2008.

[10] A. Kishimoto and M. Müller. A Solution to the GHI Problem for Depth-First Proof-Number Search. *Information Sciences*, 175(4):296–314, 2005.

[11] A. Kishimoto, M.H.M. Winands, M. Müller, and J.T. Saito. Game-Tree Search Using Proof Numbers: The First Twenty Years. *ICGA Journal*, 35(3):131–156, 2012.

[12] R. Lake, J. Schaeffer, and P. Lu. *Solving Large Retrograde Analysis Problems Using a Network of Workstations*. Department of Computing Science, University of Alberta, 1993.

[13] T.R. Lincke. *Exploring the Computational Limits of Large Exhaustive Search Problems*. PhD thesis, Technische Wissenschaften ETH Zürich, 2002.

[14] M. Müller. Proof-Set Search. In *Computers and Games*, pages 88–107. Springer, 2003.

[15] W. Myrvold and F. Ruskey. Ranking and Unranking Permutations in Linear Time. *Information Processing Letters*, 79(6):281–284, 2001.

[16] A. Nagai. *DF-PN Algorithm for Searching AND/OR Trees and its Applications*. PhD thesis, Department of Information Science, University of Tokyo, 2002.

[17] A.J. Palay. *Searching with Probabilities*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1983.

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

[19] J.N. van Rijn and J.K. Vis. Endgame Analysis of Dou Shou Qi. *ICGA Journal*, 38(3):120–124, 2014.

[20] J.W. Romein, A. Plaat, H.E. Bal, and J. Schaeffer. Transposition Table Driven Work Scheduling in Distributed Search. *AAAI National Conference*, pages 725–731, 1999.

[21] F. Ruskey. *Combinatorial Generation*. University of Victoria, Victoria, BC, Canada, 2003. Working Version.

[22] M.P.D. Schadd, M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. van den Herik, and M.H.J. Bergsma. Best Play in Fanorona Leads to Draw. *New Mathematics and Natural Computation*, 4(03):369–387, 2008.

[23] J. Schaeffer, Y. Björnsson, N. Burch, R. Lake, P. Lu, and S. Sutphen. Building the Checkers 10-Piece Endgame Databases. In *Advances in Computer Games*, pages 193–210. Springer, 2004.

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

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

[26] M. Schijf, L.V. Allis, and J.W.H.M. Uiterwijk. Proof-Number Search and Transpositions. *ICCA Journal*, 17(2):63–74, 1994.

[27] M. Seo, H. Iida, and J.W.H.M. Uiterwijk. The PN*-Search Algorithm: Application to Tsume-shogi. *Artificial Intelligence*, 129(1):253–277, 2001.

[28] K. Thompson. Retrograde Analysis of Certain Endgames. *ICCA Journal*, 9(3):131–139, 1986.

[29] UNSW School of Computer Science and Engineering. Animal Checkers. `http://www.cse.unsw.edu.au/~cs3411/07s1/hw3`, 2007. [Online; accessed 2015-04-24].

[30] R. Wu and D.F. Beal. A Memory Efficient Retrograde Algorithm and its Application to Chinese Chess Endgames. *ICCA Journal*, 42:213–227, 2002.

[31] A.L. Zobrist. A New Hashing Method with Application for Game Playing. *ICCA Journal*, 13(2):69–73, 1970.