Analysis and Implementation of Lines of Action

M.H.M.Winands

Master's Thesis CS 00-03

Thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Science of Knowledge Engineering
in the Faculty of General Sciences
of the Universiteit Maastricht

Universiteit Maastricht
Institute for Knowledge and Agent Technology
Department of Computer Science
Maastricht, The Netherlands
August 2000

Contents

List of Tables

List of Figures

Preface

This report is my M.Sc. thesis. The research was performed at the computer science department of the Universiteit Maastricht, part of the research institute IKAT (Institute for Knowledge and Agent Technology). The subject of study is a two-person zero-sum game with perfect information. Specifically, we look at the game named LOA, which is too complex to be solved with present means. During my research LOA became a very exciting game to play. I wish to thank the following people for helping me to bring this thesis to a good end.

First of all, I want to thank my supervisors. The contribution of my supervisor prof. dr. Jaap van den Herik regarding his remarks on my program, especially the move generation, and his course Zoektechnieken, which gave me some new insights, was of great benefit. Also his criticising the contents of this report improved the readability of the text considerably. The most credits go to my daily advisor, dr. ir. Jos Uiterwijk, who had to endure my shifts of opinion, my program code and my report. We had many fruitful discussions, leading to very good suggestions. I think that without him the thesis would not have been what it is now.

Further, I thank Yngvi Björnsson and Darse Billings for reading this thesis. Their remarks were very useful. Their comments about the strong and weak points of the program MIA at the MSO will trigger me to improve the program considerably.

I am also grateful for the help of some LOA programmers / experts. Especially I want to thank Dave Dyer for sharing his LOA knowledge, particularly concerning the idea of quad counts. I want to thank Kerry Handscomb for giving me his articles regarding LOA. Also I thank Fred Kok for supplying some articles about LOA and testing the program.

Finally, I would like to thank the people who helped me out with the programming. I thank dr. David Sturgill for supplying an interface for board games. Also I want to thank my college buddy Patrick Hensgens for sharing his knowledge about the Java language.

Mark Winands
Valkenburg, August 2000

Abstract

The focus of the thesis is game playing. The problem domain is the game Lines of Action (LOA). The complexity of the game, some applicable search methods and evaluation functions are examined.

LOA is a two-person zero-sum game with perfect information. It is a connection game, albeit non-typical in comparison with Hex. It has also two interesting properties: *sudden death* and *convergence*. The following problem statement is considered:

What is the complexity of the game of Lines of Action?

The following complexities of the game LOA are computed: the state-space and game-tree complexity. The state-space complexity of LOA is $O(10^{24})$ and the game-tree complexity is $O(10^{56})$. This leads to the formulation of the second problem statement:

Is it possible to crack LOA?

The state of the art of computer techniques is that LOA is uncrackable. Because so far no structure is discovered in LOA that would result in the formulation of knowledge rules and solving the game, a program is developed that plays the game reasonably well.

Therefore, an alpha-beta depth-first search with iterative deepening is performed. Two aspects of constructing a search tree in LOA consume much time: move generation and detecting terminal nodes. Therefore moves are generated for a part incrementally. A heuristic is used showing that a great share of the possible positions in LOA are not terminal. This heuristic is based on quad counts and Euler numbers. Several move-ordering techniques are used. The move stored in the transposition table is always tried first. The history heuristic works very well in LOA. The explored game-specific ordering techniques are less successful. Killer moves do not work in combination with transposition tables. Transposition tables give a huge speed-up in LOA, but not as high as in chess. Windowing techniques are also used. PVS and null move both work fine. A curiosity of LOA is that its final positions still have a great deal of pieces remaining on the board. Endgame databases are not very useful in LOA. Although LOA is a connection game, capturing pieces is important. Therefore a quiescence search is performed, which looks only at capture moves, which destroy or create connections. Restricting the moves investigated to these types of capture moves, a very effective quiescence search has been implemented. The use of conventional search techniques works also well in the game of LOA.

Concentration, centralisation, solid formations, connections and (partial) blocking are important components in evaluation functions concerning LOA. There are three evaluation functions constructed, denoted *normal*, *quad* and *blocking*. They all consider the concentration and the centralisation of pieces on the board. The quad evaluator uses the quad count to determine the solidness and connectioness of the formations. The blocking evaluator looks at the mobility of pieces towards the neighbourhood of the centre of mass. The quad evaluator is the best of the three.

The program developed is called MIA (Maastricht In Action). It is competitive with world class LOA programs as YL and Mona. Quad count and quiescence search are responsible for MIA's strength.

Samenvatting

De focus van de thesis is zoektechnieken, toegespitst op spelen. Het probleem domein is het spel Lines of Action (LOA). De complexiteit van het spel, enige toepasbare zoekmethoden en evaluatiefuncties worden onderzocht.

LOA is een tweepersoons nulsom spel met perfecte informatie. Het is een verbindingsspel, maar niet standaard in vergelijking met Hex. Het heeft twee interessante eigenschappen: *sudden death* en *convergentie*. De volgende probleem stelling wordt beschouwd:

Wat is de complexiteit van het spel Lines of Action?

De volgende complexiteiten van het spel LOA worden berekend: de ruimte en de spelboom complexiteit. De ruimte complexiteit van LOA is $O(10^{24})$ en de spelboom complexiteit is $O(10^{56})$. Dit resulteert in de formulering van de tweede probleemstelling:

Is het mogelijk om LOA te kraken?

De huidige staat van computertechnieken is dat LOA onkraakbaar is. Omdat tot nu toe geen structuur in LOA is ontdekt, die zou resulteren in de formulering van kennisregels en het oplossen van het spel, wordt een programma ontwikkeld dat het spel redelijk speelt.

Hiervoor wordt een alpha-beta depth-first zoekalgoritme met iterative deepening uitgevoerd. Twee aspecten bij de constructie van een zoekboom in LOA kosten veel tijd: zettengeneratie en detectie van eindknopen. Daarom worden zetten deels incrementeel gegenereerd. Een heuristiek wordt gebruikt die in staat is een groot deel van de niet-terminale posities in LOA te herkennen. Deze heuristiek is gebaseerd op quad telling en Eulergetallen. Verscheidene ordeningstechnieken worden gebruikt. De opgeslagen zet in de transpositietabel wordt altijd als eerste onderzocht. De historie heuristiek werkt goed in LOA. De onderzochte spelspecifieke ordeningstechnieken zijn minder succesvol. Killer-zetten werken niet in combinatie met transpositietabellen. Transpositietabellen zorgen voor een grote versnelling in LOA, maar niet zo hoog als in schaken. Raamtechnieken worden ook gebruikt. PVS en nul-zet werken beide goed. Een curiositeit van LOA is dat zijn terminale posities nog steeds veel stukken bevatten. Eindspelbibliotheken zijn niet erg nuttig in LOA. Hoewel LOA een verbindingsspel is, zijn slagzetten belangrijk. Daarom wordt een quiescence search uitgevoerd, die alleen kijkt naar slagzetten, die verbindingen vernietigen of creëren. Door de onderzochte zetten te beperken met deze types van slagzetten is er een erg effectieve quiescence search geïmplementeerd. Het gebruik van conventionele zoektechnieken werkt ook goed in LOA.

Concentratie, centralisatie, solide formaties, verbindingen en (partiële) blokkades zijn belangrijke componenten voor evaluatiefuncties betreffende LOA. Er zijn drie evaluatiefuncties gemaakt, te weten *normal*, *quad* en *blocking*. Ze houden allemaal rekening met de concentratie en de centralisatie van de stukken op het bord. De quad evaluator gebruikt de quad telling om de soliditeit en verbondenheid van de formaties te determineren. De blocking evaluator kijkt naar de mobiliteit van de stukken richting de omgeving van het zwaartepunt. De quad evaluator is de beste van de drie.

Het ontwikkelde programma is genaamd MIA (Maastricht In Action). Het is concurrerend met sterke programma's als YL en Mona. De quad evaluator en quiescence search zijn verantwoordelijk voor MIA's kracht.

1 Introduction

The Good, the Bad and the Lines

1.1 Games

As long as civilisation exists, mankind has played board games. Games challenge the human intellect. Board games such as chess and draughts are interesting because they offer pure abstract competition, without the details of two armies going to war.

As long as games exist, mankind has invented different kinds of them. This is the reason, why we can categorise them. For example, we can distinguish games by the kind of information they provide or by the number of players. But we can also look at the theme of the game. Most of the older abstract board games are primarily war or territory based. Sixty years ago the first true connection game called Hex made its debut (Anshelevich, 2000). The objective of connection games is to group the pieces such that they form a kind of connection. What constitutes a connection really is defined in a game-dependent way. Connection subsequently became one of the great themes of twentieth-century abstract gaming; many prominent game inventors have made their contribution to this theme. Besides Hex, examples of connection games are Twixt (Thompson, 2000) and LOA.

As long as computers exist, mankind has tried to let them play non-trivial games. The challenge of a computer playing an intelligent game is a classic problem within the field of Artificial Intelligence (AI). Shannon (1950) and Turing (1953) were the first ones to describe a chess-playing program, ultimately leading to DEEP BLUE, that defeated the human World Champion (Schaeffer and Plaat, 1997). Samuel (1959) was the first who constructed a program, which played a reasonable game of checkers. Schaeffer's CHINOOK fulfilled Samuel's dream in 1995 by becoming World Champion (Schaeffer, 1997). There are many games where computers are superior. But humans are still in control in several games, like Backgammon and Go. There are five reasons why researchers are so interested in intelligent games:

1. Games provide an exact closed domain with well-defined rules, in contrast with real-world problems, which are often rather vague (Van den Herik, 1983).
2. Intelligent games are not easy. Learning the rules is mostly easy, but playing the games is hard (Minsky, 1968).
3. The domain of games is well suited for testing new ideas in problem solving. Some ideas have been used in mathematics, computer science and economics. Well-known examples are the travelling-salesman problem and chromosome matching (Nilsson, 1971).
4. By creating a machine, which plays an intelligent game, it may be possible to gain more insight into the way people reason. De Groot (1946), Newell and Simon (1972) are famous representatives of this.
5. It is fun to build programs, which can play intelligent games.

Game playing is to AI as Grand Prix motor racing is to car racing: although the specialised task and extreme competitive pressure lead one to design systems that do not look much like your garden-variety, general purpose intelligent system, many leading edge concepts and engineering ideas come from it. But, just as you would not expect a Grand Prix racing car to perform well on a dirty road, you should not expect advances in game playing to be translatable immediately into advances in less abstract and more society important domains (Russell and Norvig, 1995).

1.2 Lines of Action

Lines of Action (LOA) is a board game which is played on a checkers board. It is a two-person zero-sum game with perfect information. LOA is a connection game, albeit non-typical in comparison with Hex. Claude Soucie invented it around 1960. Sid Sackson (1969) described it in his first edition of *A Gamut of Games*. We use the rules described in the *second edition* of A Gamut of Games:

1. One player controls the twelve black pieces and the other the twelve white pieces. The black pieces are placed in two rows along the top and bottom of the board, while the white stones are placed in two files at the left and right of the board. The set-up of the game is shown in figure 1.1.
2. Black moves first.
3. Each turn, the player to move *has to* move one of his pieces, in a straight line, exactly as many squares as there are pieces of either colour *anywhere* along the line of movement. (These are the *Lines of Action*).
4. You may jump over your own pieces.
5. You may not jump over your opponent's pieces, but you can capture them by landing on them.
6. The object of the game is to turn your pieces into one connected unit. The first player to do so is the winner. The connections within the group may be either orthogonal or diagonal. For example, in figure 1.2 black has won because his pieces form one connected unit.
7. If one player is reduced by captures to a single piece, that is a win for this player.
8. If a move simultaneously creates a single connected unit for both the player moving and the opponent, the player moving wins.[1]

These are the official rules. Some situations were not covered by the original rules. For these situations, the rules of the Mind Sports Olympiad (MSO) are used:

9. If a player cannot move, this player loses.[2]
10. If a position with the same player to move occurs for the second time, the game is drawn.

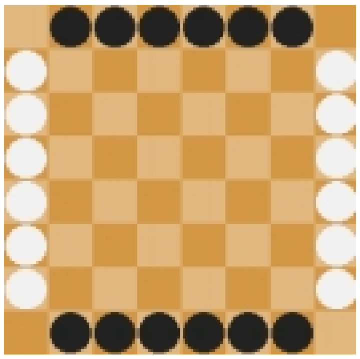


Figure 1.1: Board setup.

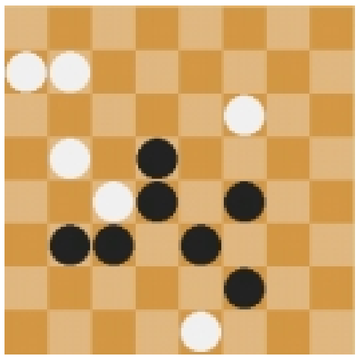


Figure 1.2: Terminal position.

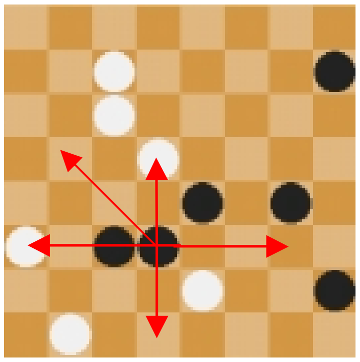


Figure 1.3: Movement of pieces.

The notation concerning the board and the moves, which we use in this thesis is the standard chess notation (see also appendix A). The possible moves of the black piece at **d3** in figure 1.3 are indicated by the arrows. It cannot move to **f1** because its path is blocked by an opposing piece. The move to **h7** is not allowed because a black piece is already there.

[1] In the first edition of A Gamut of Games, this situation was described as a draw.
[2] The use of this rule is disputable, because some LOA players think that the player has to pass.

Looking at the rules of the game we can see that LOA has the sudden-death property. A sudden-death game may end abruptly by the creation of one of a prespecified set of patterns. Grouping stones in one connected unit is the pattern to be formed in LOA. Another property of LOA is that it is a converging game (Allis, 1994). The initial position in LOA consists of 24 pieces, while during the game the number of pieces usually decreases. After the first few capture moves, the number of legal LOA positions decreases as the number of pieces decreases.

After the "release" in 1969, LOA already got some attention of AI researchers. The first program, playing LOA, was written at Stanford in a language called SAIL. It ran on the PDP-10 at the Stanford AI lab, circa 1975. Unfortunately the author of this program is unknown. Dave Dyer wrote also a LOA program, which was better than the Stanford program. In the eighties some Macintosh programs were developed. It was until the nineties that the first PC programs occurred. Nowadays several LOA programs exist like Mona, YL, Loa2D, LoaW etc. (Dyer, 2000).

LOA is *not* a game invented for some bored researchers, it is really played by people. First played at US universities, it is now played world wide thanks to the Internet. Several e-mail tournaments exist. The first world championship was held at the MSO in 1997, which was won by the Dutchman Fred Kok, the only grandmaster in the game of LOA. For more information about LOA see (Dyer, 2000).

1.3 Computer Game Playing

Computers can play games usually with an emphasis on one of the following two methods: knowledge-based methods or search-based methods. Abstract games are solvable in those two distinct ways.

If a truly knowledge-based approach is chosen, the game is solved using knowledge, not using any search. This is possible if all information of the initial state and the subsequent states necessary for solving the game is available. Mostly rules are defined for certain situations and stored in a rule base, but other representations of knowledge are also possible. There has to be sufficient space to store the knowledge, and of course discovering and representing the knowledge for solving the game should be feasible. A well-known example of a game solved only with knowledge is Nim (Bouton, 1901).

If a truly search-based method is used, the game is solved by search and not by knowledge. This is possible if there is sufficient time available to do a sufficiently large search of the state space. A well-known example of a game solved completely by search alone is Tic-Tac-Toe. Search-based methods always explore the game tree to solve the game. A game tree is a representation of the state space of the game. A node in the tree represents a position in the game; an edge represents a move. The root of the tree is a representation of the initial position. A terminal node is a position where the rules of the game determine whether the outcome is a win, draw or loss. A node is expanded by generating all successors of the position represented by the node. A node with at least one successor is defined as an interior node. Thus, a game tree is generated by expanding all the interior nodes. This process is repeated until all unexpanded nodes are terminal nodes. The game-theoretic value is the value of the initial position given that both players play optimally. This is done by using the minimax algorithm (Von Neumann and Morgenstern, 1944). Sometimes the game tree is therefore called a minimax tree, and the search conducted a minimax search. A minimal game tree is defined as a minimal part of the tree necessary to determine the game-theoretic value.

When the game is too large to be generated completely, a search tree is generated instead. This tree is only a part of the game tree. The root represents the position under consideration, and the other nodes are generated till a given depth. The nodes, which do not have children, are defined as leaves. Leaves include terminal nodes and nodes which are not yet expanded. In the leaf nodes an evaluation function is used to estimate the value of a certain position. This is the way to determine the best move for a given position according to the evaluator. Thus, by using an evaluator knowledge is applied to guide the search. How deep the search tree is explored is dependent on computer horsepower. This is the reason why this approach is called brute-force.

1.4 Problem statement

Two problem statements are considered:

What is the complexity of the game of Lines of Action?

We try to compute the state-space and game-tree complexity. If we know them, we can solve the next problem statement:

Is it possible to crack LOA?

You crack a game if you can retrieve the game-theoretical value of the initial position by doing a brute-force search (Allis *et al.*, 1991). We know now *what* the best move is for every faced opposition. If you also know *why*, the game is solved. Dependent on the results a program will be developed that either computes the game-theoretic value of the game or can play the game reasonably well, where we try to use endgame databases.

1.5 Outline of the thesis

The contents of the thesis is as follows.

Chapter 1 contains an introduction in games, especially in LOA, two problem statements and an outline of the thesis.

Chapter 2 tries to answer the first problem statement. The state-space and the game-tree complexity are computed. These complexities are compared with other games, which results in some conclusions.

Chapter 3 presents the methods to guide a depth-first search. First, some important notions and concepts about search are defined. A method is given to generate moves in LOA. Also a method is presented to detect terminal nodes in LOA. Next, the following heuristics are described: killer move, history heuristic, some game-specific move-ordering techniques, transposition tables, PVS and null move. The usefulness of these is explained. Also quiescence search is discussed. Finally, some heuristics not used are taken into consideration.

Chapter 4 presents some evaluation functions. First, we try to give some insight into playing the game of LOA. Some principles are noticed and discussed. Next, several evaluation functions are discussed based on these principles.

Chapter 5 gives some benchmark results. First, some characteristics of the game are examined. Next, the usefulness of several heuristics and evaluation functions are explored. Finally, some results against other programs are given.

The evaluation of the second problem statement, final conclusions, and future research are given in chapter 6.

2 Complexity of LOA

My name is complexity

If we want to develop some insight into the game of LOA, we have to calculate its complexity. Two distinct measures for complexity are used: the state-space complexity and the game-tree complexity. In this chapter we will examine those two.

2.1 State-space complexity

An upper bound of the state-space complexity for LOA can easily be derived. The computation is of the same kind as in checkers (Schaeffer and Lake, 1996). Let *B* be the number of black pieces and *W* the number of white pieces. There are at most twelve black and twelve white pieces. Each of the 64 squares on the board can be occupied by a piece. The number of positions having 24 pieces or less is derived in the following formula:

$$\sum_{B=1}^{12}\sum_{W=1}^{12} Num(B,W) - Num(1,1) \qquad \text{where} \quad Num(B,W) = \binom{64}{B}\binom{64-B}{W} \tag{2.1}$$

In the formula we have neglected all positions with two pieces or fewer, because they are impossible to reach. One of the players would have won before the last move was made. For the same reason we neglect all positions with only pieces of the same colour on the board. These neglections only lead to a reduction of 8.4×10^{12} positions.

# of pieces	# of positions
3	249,984
4	8,895,264
5	228,735,360
6	4,648,410,816
7	78,273,240,192
8	1,124,246,003,472
9	14,045,698,101,120
10	154,805,625,541,952
11	1,521,396,969,611,904
12	13,445,574,994,311,264
13	107,590,873,672,114,560
14	782,632,117,126,476,864
15	5,188,514,989,534,830,720
16	31,335,094,798,158,420,360
17	171,930,216,128,039,066,880
18	853,283,294,857,675,368,960
19	3,807,935,897,946,939,333,120
20	15,158,514,055,288,777,729,920
21	53,164,643,498,259,191,458,560
22	160,592,373,542,262,268,414,080
23	396,757,628,751,471,486,670,080
24	677,794,282,450,430,456,394,720
Total:	1,308,338,020,676,454,019,380,152

Table 2.1: State-space complexity.

In table 2.1, we have calculated the separate combinations per number of pieces using the same neglections as in equation 2.1. For example, the combination of three pieces is *Num(2,1)* + *Num(1,2).* If we sum those numbers together, we have eventually 1.3×10^{24} combinations. This number can also be obtained using equation 2.1. However, we have still included some positions, which are not reachable from the start (Dyer, 2000), meaning that the state-space complexity will be a little bit lower in reality.

Joseph Kisenwether pointed out the following two classes of unreachable positions. First, there are positions in which no piece has been captured and both white and black have all of their pieces united. This situation is shown in figure 2.1. The reason that this is unreachable is that whoever did not move last must have united his pieces on the move before the last move and the game would have ended then. Second, there are positions in which both white and black have all of their remaining pieces connected *and* every piece borders a piece of the other colour. This is unreachable for a similar reason to the class above: even if the last move was a capture, the player who made the preceding move would have completed a winning configuration and the game would have been finished. Such a position is depicted in figure 2.2.

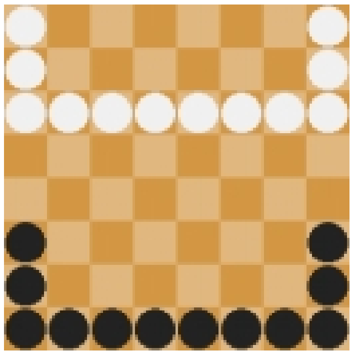


Figure 2.1: Unreachable terminal position I.

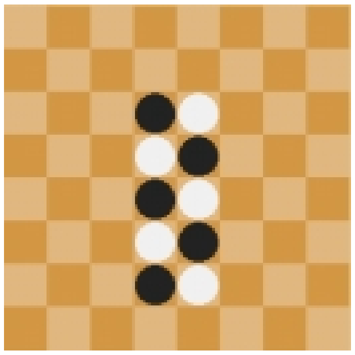


Figure 2.2: Unreachable terminal position II.

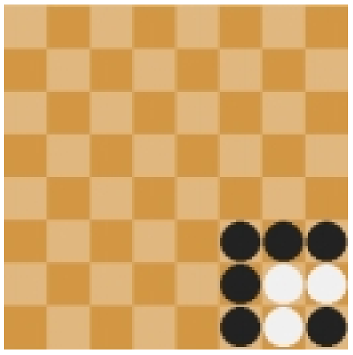


Figure 2.3: Unreachable terminal position III.

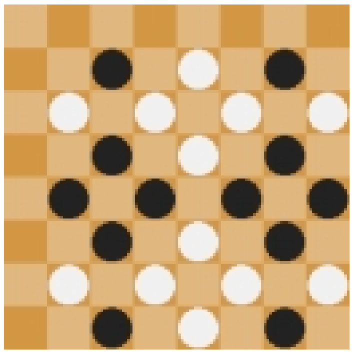


Figure 2.4: Stalemate position.

Jorge Gómez Arrausi discovered the following class of unreachable positions. These are positions in which one player is connected, but neither player could have created the connected position. For example, in figure 2.3, it is easy to generate all possible white predecessor positions, which have one of the white stones at some other position on the board, and possibly has a black stone where the white stone is now. In this position, there are exactly 330 such hypothetical white predecessors. Similarly, black must have created this position by capturing an isolated white stone, and since all black stones are adjacent to white, black could not have just created this position.

All these classes of impossible positions have in common that they are unreachable because the game would have already been ended. Joseph DeVicentis constructed a stalemate position, shown in figure 2.4, which is unreachable. In this position neither player can make a move. This position could only be reached by a capture move by white (**e8×e3**). But black would have had 13 pieces on the board, so this position could not arise in a real game.

It is possible that there are more classes of unreachable positions in LOA.

2.2 Game-tree complexity

The game-tree complexity can be approximated by the number of leaf nodes of the game tree with as depth the average game length (in ply), and as branching factor the average branching factor during the game. It is not hard to calculate the last. After 200 runs of the program playing against itself, we have determined the average branching factor at 30. In the beginning of the game the branching factor is 36. The branching factor will increase in the beginning, because we will get more moves per piece. But the branching factor will decrease due to capture moves and blocking. For more details see also chapter 5.

It is hard to determine the average game length. We only know that the minimal game length is 9. There is not a large game database with games at grandmaster level. The reason for this is that LOA is a young game and not played frequently at a serious level like chess, checkers, Go etc. The only way to estimate the average game length is to perform self-play experiments. Our program played 200 games against itself. The result is an average game length of 38. For more details see also chapter 5.

The formula to calculate the game-tree complexity is b^d, where b is the average branching factor and d is the average depth. The game-tree complexity for LOA is calculated as 10^{56}.

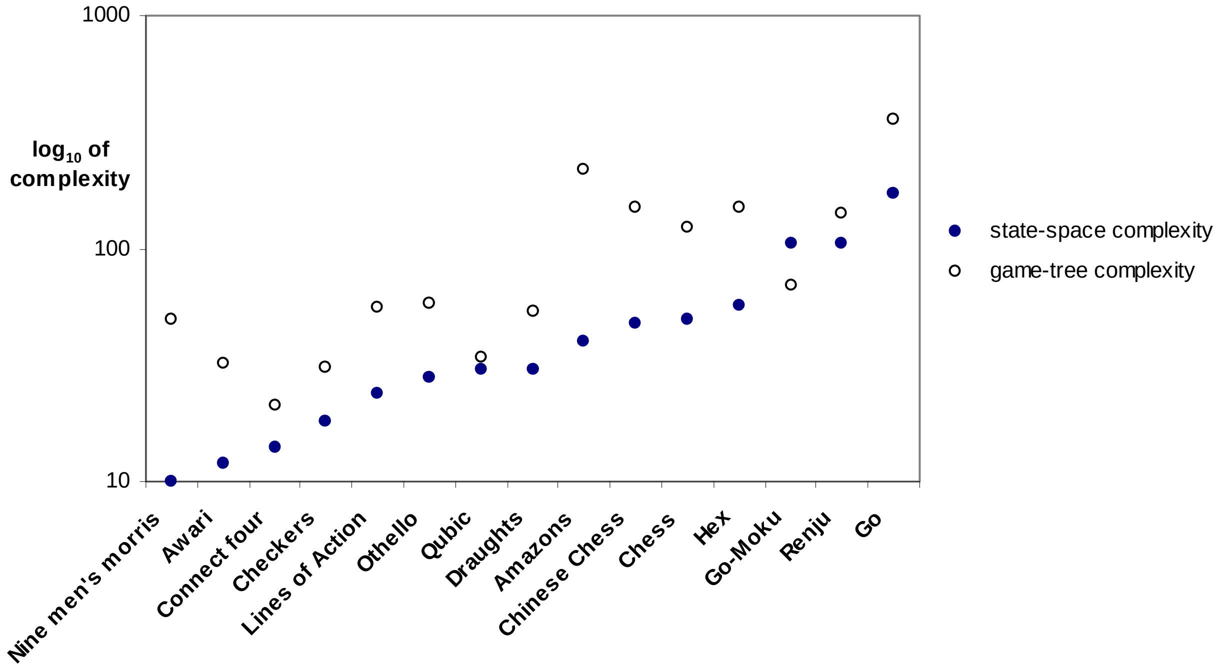


Figure 2.5: Estimated game complexities.

2.3 Chapter conclusions

In figure 2.5 we have compared the complexity of LOA with other games. This figure has been taken from Allis (1994). We have only added the complexity of Amazons, Hex and LOA, and gave another estimate of the game-tree complexity of Renju. The latter was done since the average game length of Renju is considerably larger than that of Go-Moku (Maltell, 2000) unlike Allis' point of view. If we have a closer look at figure 2.5, we see that LOA’s state-space complexity and game-tree complexity is higher than those of draughts but much lower than those of chess. We can compare LOA with Othello, whose state-space and game-tree complexity is slightly higher than LOA. Authors of Othello computer programs, using brute-force methods, have stated that their programs played at the level of the World Champion since 1980. Current programs outplayed their human opponents. Analogous to Othello, we can easily predict that LOA programs also will surpass humans soon. The chances that LOA will be solved in the near future are like Othello (extremely) remote. The state-space complexity rules out the option of full enumeration, while the game-tree complexity renders a full-depth forward search impossible. Only if a so far unidentified structure in the game is discovered, resulting in knowledge rules, which prove the value of nodes early in the game tree, maybe LOA will be solved. An argument that LOA can be solved is that it has the sudden-death property. For games of high complexity the sudden-death element in combination with a clear advantage for one of the players (black in LOA, see also appendix D for some numbers) may be the main property that allows the game to be solved. For example, Qubic and Go-Moku are solved because of this property.

Game-tree complexity is not necessarily a good measure for the complexity of the game. First, some games with high game-tree complexity are solved. For example, Go-Moku has a higher game-tree complexity than LOA, but has been solved. Second, the underlying structure of the problem domain of most games is a graph, not a tree. Some positions can be reached in several ways (see also chapter 3). Thus, in the computation of the game-tree complexity, we have included some board positions several times. The game-tree complexity sometimes overestimates the true complexity of the game. But the game-tree complexity is an estimate of the size of a minimax search tree, which often must be built to solve the game (Allis, 1994).

Normally, the state-space complexity is also not such a good measure for the complexity of the game. The same kind of reason can be used as above: Go-Moku has higher state-space complexity than LOA, but has been solved. However state-space complexity can give us some insight into the usefulness of endgame databases. This will be further explored in chapter 3.

3 Search

For a few ply more

In chapter one we defined the concept of game tree. We also noticed that it is impossible to examine the whole game tree for LOA. We can only search through the tree to a given depth. It is thus important to have search methods, which will guide the search efficiently and consistently. Ultimately our goal is to look as deep as possible, for some given position and time.

3.1 Tree construction

Before we can do a search, we need to generate the tree. We have to expand the leaf nodes. This is done by move generation. Also we he have to check if the visited nodes are terminal. In the next subsections we will discuss this matter.

3.1.1 Move generation

In a brute-force search, move generation is a time-consuming process. It is important to generate moves fast, resulting in a deeper investigation on average in the search process. Also generating moves in LOA is not trivial. Every piece can possibly move in any direction and the total number of squares that a piece can move is dependent on the pieces on the lines. Most programmers fall into the trap to generate the list of moves from scratch every time an interior node is visited. For example, if we look at the initial position, we can see that this is not a good idea. The black pieces on the first row have the horizontal line in common. If we have once calculated the number of pieces in a line, we do not have to do that again. We use look-up tables for the pieces in action in a line. There are four tables, one for the horizontal, one for the vertical, and two for the diagonal lines. In total there are 46 lines (8 + 8+ 15 + 15 = 46). When a move is played, most lines do not change. In case of a capture move only four lines change, and in case of a non-capture move six lines change. The changed lines are subtracted or added by unity. Thus, a part of our move generation is incrementally updated. This is a reduction of a large amount of work, but generation is still expensive. More than 30% of the time is spent on generating moves.

3.1.2 Detection of terminal nodes

In LOA it is not trivial to detect a terminal node. Remember that the game is over when a group of pieces with the same colour is connected. In most cases this is not easy to detect. Checking on a game-over situation is a time-consuming process in LOA, contrary to games like draughts and chess. The obvious solution is to update incrementally the count of pieces when moves are made. Next, count the number of pieces of an arbitrary group, using a depth-first algorithm. If this number equals the total number of stones of that colour, the game is over. However, the depth-first search necessary to retrieve the number of pieces is a time-consuming process.

Fortunately, there is a heuristic that can detect in 99% of the cases that a game definitely is not over. The solution lies in using bit quads, an OCR method. A quad is defined as a 2×2 array of bit cells (Gray, 1971). In LOA there are 81 possible bit quads for each side, including also bits quads covering only a part of the board along the edges. Considering rotational equivalence, there a six distinct types, depicted in figure 3.1:

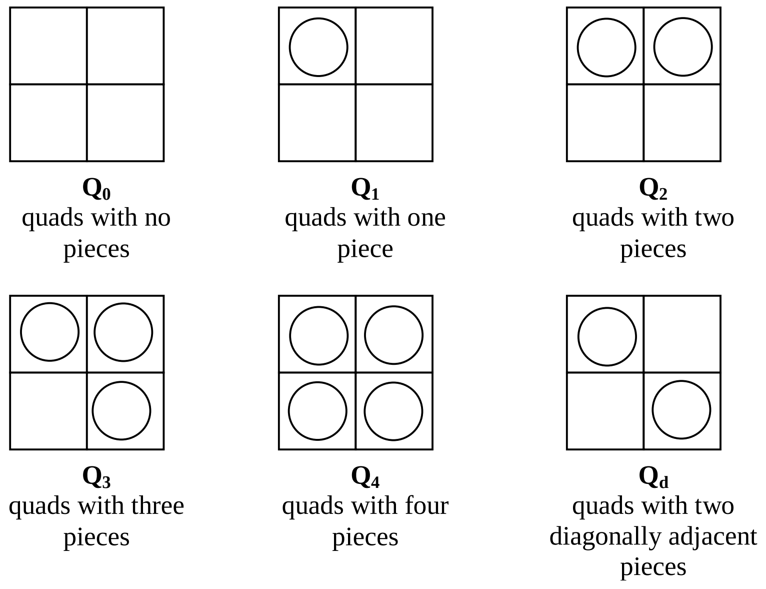


Figure 3.1: Different kinds of quads.

Notice that we have two kinds of quads with two pieces. One in which the pieces are diagonally adjacent, another in which they are orthogonally adjacent. If a quad only partially covers the board, the cells not covering the board are considered empty. Summing over all cell group counts gives us the Euler number *E* of the board as follows:

$$E = \left(\sum Q_1 - \sum Q_3 - 2\sum Q_d\right)/4 \tag{3.1}$$

For a motivation of this formula we refer to Gray (1971). The Euler number represents the number of connected groups minus the number of holes (interior regions). For example, in figure 3.2 the Euler number of black is three, because black has three connected units. The Euler number of white is one. White has two connected units, but the pieces are surrounding the region at **e5**. This is called a hole. Therefore the Euler number of white is equal to two connected units minus one hole. This example shows us that Euler numbers do not fully detect win positions. If the Euler number is greater than one, the game is definitely not over (Minsky and Papert, 1988). Only when there are at least as many holes as connected groups, we have to use another method. But fortunately holes in LOA are rare and using this heuristic is often sufficient to detect non-terminal positions.

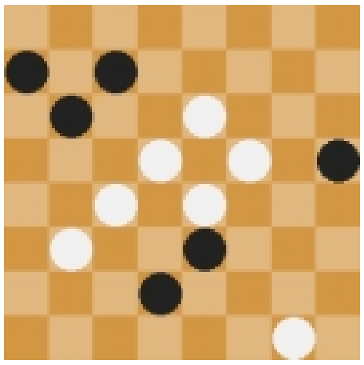


Figure 3.2: Position with hole.

An advantage of this method is that we can incrementally update the bit quads and their counts. There are mostly eight quads that change as the result of a move, with a maximum of twelve when a capture move occurs. Although the bookkeeping is not trivial, it is still cheap enough to outperform other methods.

Depth	Time (ms) with quads	Time (ms) without quads	Nodes
1	50	60	37
2	220	220	217
3	440	440	1,671
4	990	1,100	6,081
5	6,530	8,070	105,933
6	52,120	60,470	817,567

Table 3.1: Results of using quads.

In table 3.1 we see that using bit quads causes a speed up of 10 to 15 % at the start position of the game. An argument against the bit quad method is that its performance is low when the total number of stones is also low. But as we will see in chapter 5, LOA usually ends with a large amount of stones still on the board. The big gain is that we can use quad counts in the quiescence search and in the evaluation function also.

3.2 Searching

Since LOA has to examine a large tree, a *depth-first* search is commonly used. That is, the first branch to an immediate successor of the current node is recursively expanded until a leaf node is reached. The remaining branches are then considered as the search process goes back at shallower levels. Unfortunately, we do not have sufficient time to do a complete search. Commonly a modified version of this algorithm is used, called iterative-deepening search. This is a depth-first search with a depth limit. The tree is completely searched up to the given depth limit. If the search is completed, the limit is increased and a new search is started. The search process consists of several search trees with increasing depth. Because so many nodes are expanded multiple times, iterative deepening search may seem wasteful. But the overhead of this multiple expansion is rather small. To make this concrete for LOA, for $b = 30$ and $d = 4$, the number of nodes expanded in a depth-first minimax search is:

$$1+30 + 900 + 27,000 + 810,000 = 837,931$$

The total number of expansions in an iterative deepening is:

$$5 + 120 + 2,700 + 54,000 + 810,000 = 866,825$$

All together, the overhead is 3.4 %. The higher the branching factor is the lower the overhead. In chapter 5 we will see that the branching factor of LOA stays high during the game, so we have a little bit of overhead. But what is the advantage of iterative deepening?

First it helps to control the time to be spent for a move decision. It is hard to predict search times for a fixed depth search, because these vary significantly for distinct positions. Iterative deepening solves this stopping problem. After every iteration or during an iteration, it is possible to stop the search and produce the best move so far.

Second this approach allows us to store useful information to enable more alpha-beta cut-offs. In the next sections we will explore this further.

3.3 Alpha-beta

As the search of the game tree proceeds, the value of the best terminal node found so far might change. There are certain moves, which cannot affect the expected value of the tree. The alpha-beta algorithm takes advantage of this situation. John McCarthy conceived the idea in 1956, although he did not publish it. Brudno (1963) was the first to give this algorithm a thorough treatment. The alpha-beta algorithm is the core of most chess programs. Knuth and Moore (1975) proved its correctness. Although other alternatives were proposed in the past, none was better. We will use alpha-beta search in LOA. The alpha-beta algorithm uses lower (alpha) and upper (beta) bounds on the expected value of the tree. The bounds may be used to prove that certain moves cannot change the result of the search, and hence they can be cut-off. For a more detailed exploration of alpha-beta see (Russell and Norvig, 1995). The effectiveness of alpha-beta depends on the ordering in which the successors are examined. For an optimal ordering it turns out that alpha-beta only needs to examine $O(b^{d/2})$ nodes instead of $O(b^d)$. This means that the effective branching factor is $\sqrt{b}$ instead of b – for LOA, 5 instead of 30. This means that we can look ahead twice as far as with plain minimax.

3.4 Move ordering

If we know the best move for every position, we can decrease the search drastically with alpha-beta. But if we know what the best move is for every position, we do not have to search anymore. We have to create an ordering function, which tries plausible moves first that creates big cut-offs. But what are plausible moves? We determine those by heuristics. Two kinds of heuristics exist: game-specific and game-independent. If we use the first one, we take the evaluation function into account. For example, in LOA we can look first at moves going to the centre. Game-independent heuristics do not take characteristics of the evaluator into consideration, but their applicability can depend on the kind of game.

3.4.1 Killer moves

The killer heuristic is proposed by Huberman (1968). When searching the game tree, it stores the last move found, which was best or at least caused a pruning, at each depth. Whenever operating at a given depth, it first proposes the move so stored. The idea is that in many situations the same move of one's opponent will be his best move after any move. Normally the last pruning or best move is stored, but more killer moves can be stored. Pruning by the killer move is mostly due to information gathered at the same level in the same search subtree. The heuristic is cheap, for k moves at n plies, its memory cost is $k \times n$ entries in a move table. In figure 3.3 black has to move. If black performs one of the moves **a5-f5**, **a5-b6**, **b5-b6**, **d7×a4**, **d7×d4**, **d7-e6**, **d7-e7**, **e5-e6**, **g6-f6**, **g6-f5**, and **g6-f7** white's best response is always **c3×c5**. This move is the killer move. For more information about the killer move see also (Akl and Newborn, 1977).

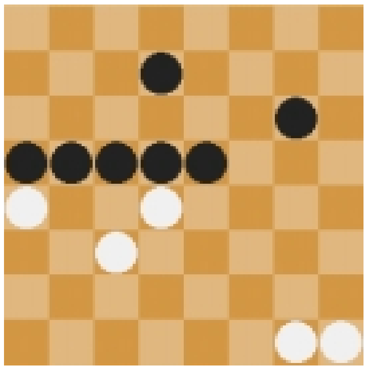


Figure 3.3: Killer move position.

3.4.2 History heuristic

This heuristic was invented by Schaeffer (1983). Unlike the killer heuristic, which only maintains a history of the one or two best killer moves at each ply, the history heuristic maintains a history for every legal move seen in the game tree. The best move at an interior node is the move which either causes an alpha-beta cut-off, or which causes the best score. With only a finite number of legal moves possible, it is possible to maintain a score for each move in two (white and black) tables. At every interior node in the search tree the history table entry for the best move found is incremented by 2^d, where d is the depth of the subtree searched under the node. When a new interior node is examined, moves are re-ordered by descending order of their scores. The scores in the tables are maintained during the whole game. They are divided by two for each new search process. They are only reset to zero at the beginning of a complete new game.

This heuristic does not cost very much memory. The history tables are defined as two tables with 4096 entries (64 *from_squares* × 64 *to_squares*), where each entry is 4 byte large. In the history table we have also defined moves which are illegal. The total memory cost is only 32K, a piece of cake for modern computers nowadays.

21	21	21	21	21	21	21	21
21	23	23	23	23	23	23	21
21	23	25	25	25	25	23	21
21	23	25	27	27	25	23	21
21	23	25	27	27	25	23	21
21	23	25	25	25	25	23	21
21	23	23	23	23	23	23	21
21	21	21	21	21	21	21	21

Figure 3.4: Number of possible moves for each square.

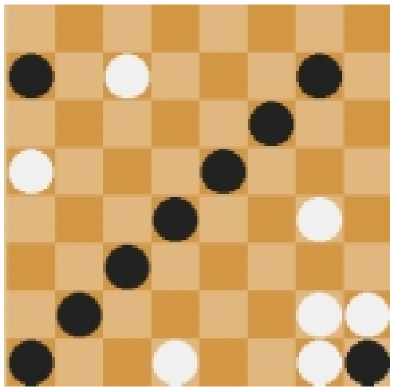


Figure 3.5: Rare move.

But a lot of the space reserved for the history table is wasted, because we reserve also space for illegal moves. Hartmann (1988) has calculated that in the game of chess 44 % of the possible moves is legal. In LOA, this number will be a bit lower because knight moves are not allowed. This means that only 1456 of the 4096 moves are legal, meaning that only 35.6 % of the entries in the table are used. In figure 3.4 the number of legal moves for each square is given. Also some moves are rare to happen in a real game. For example, moves going seven squares far are possible if and only if there are seven pieces of the same colour side by side on that line. In figure 3.5 it is possible to move **a1-h8**, but is very rare that in real game a position occurs where seven pieces of the same colour occupy a diagonal.

3.4.3 Game-specific ordering

In most game programs game-specific (evaluator-specific) techniques are used, when the other move ordering techniques have no preferences between some moves. In war-like games capture moves are mostly preferred. Although sometimes a capture move in LOA is tactically strong, in most situations, they would help the opponent. We have performed some experiments, but this move-ordering technique unfortunately turned out to be bad, even if we did not use other move ordering at all.

We also tried to prefer moves, which decrease their distances toward the centre of mass above those who do not. This turned out to be profitable, if we did not use another move-ordering technique, but it did worse in combination with the normal move-ordering techniques.

Finally, we defined an outsider move as a move going to a border field along the edges. Mostly this kind of moves is bad, and so non-outsider moves are preferred above outsider moves. In some situations this heuristic did give some extra cut-offs in combination with other techniques, but mostly it did worse.

At the end all game-specific ordering techniques were abandoned. It turned out to be that the game-independent move-ordering techniques were much better.

3.5 Windowing techniques

For most positions in LOA it is not possible to get a huge forced gain or loss at a next ply. So it seems unreasonable to set the initial bounds of alpha and beta at $-\infty$ and ∞ respectively. If we use some initial bounds (called a window), it is possible to get more cut-offs. But the disadvantage is that sometimes those bounds do not enclose the minimax value, and a re-search is needed. The problem is to find a window that provides overall more cut-offs including possible re-searches. There are several methods to do this.

3.5.1 Minimal-window principal variation search

One method to set the window is to close it as much as possible. This means that the beta value equals alpha +1. The basic idea behind this method is that it is cheaper to prove a subtree inferior, than to determine its exact value. Principal variation search (PVS) uses this concept. It has been shown that this method does well for bushy trees like in chess. Because the branching factor of LOA (30) is in the same range as in chess (35), we can safely presume PVS will work fine in LOA. Provided we have a good move-ordering mechanism, PVS reduces the size of the search tree. For a more detailed description of the algorithm, see also (Marsland, 1986).

3.5.2 Null-move search

The null move means changing who is to move without any other change to the game state. It is thus equivalent to passing, which is not allowed in LOA. The reason for doing a null move is that we hopefully can narrow our window resulting in a smaller search tree.

The null move is performed before any other move and it is searched at lower depth than we would do for other moves. The search depth of the null move is reduced by the factor R, which we have set at 2 in LOA. If the null move produces no cut-off or improvement of alpha, some unnecessary search has been done. This has to be avoided and in the following situations the null move is not performed:

- The previous move was a null move.
- The search is at the root of the tree.
- The side to move is at a considerable disadvantage, the chances for a cut-off or an improvement of alpha being low.

For a more detailed description of the algorithm see (Donninger, 1993).

We have assumed that doing a null move is a poor move: there is always a move that is better than the null move. That is the reason why we can safely use a null move. Problems occur when the null move is the best move. It is possible then that we get different outcomes for the search when doing null moves or not. This situation is called zugzwang (Uiterwijk and Van den Herik, 2000) and occurs also in LOA. In figure 3.6 black has to move. He can move the pieces on **f7**, **g7** and **h7**. First, if black moves **f7**, white can always play **f6-f7**. Next, if black tries to move **g7,** white can always play **f6-g7**. Finally, if black plays **h7**, white can always play **g8-h7**. If black moves, white will always win the game, but if he was able to pass white has no move to win the game instantly.

If the position is only two plies or less from the horizon, the null move is not performed. This is done to avoid zugzwang.

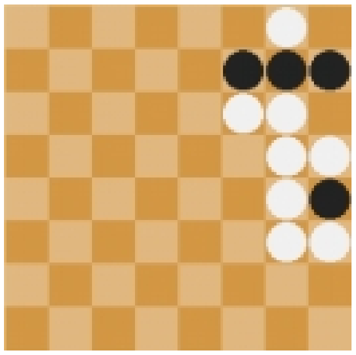


Figure 3.6: Zugzwang position.

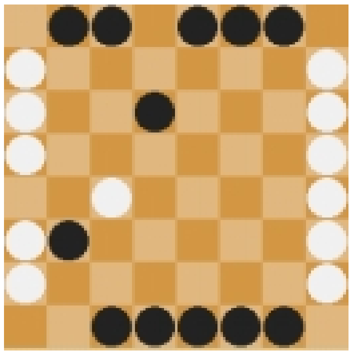


Figure 3.7: Position that can be reached by distinct move orders.

3.6 Transposition table

When we are searching for a good move, programs build large trees. Since a position can sometimes be arrived at by several distinct move sequences, the size of the search tree can be reduced considerably if the results of a position are stored. This happens also in LOA. The position showed in figure 3.7 can be reached by **1. b1-b3 a4-c4 2. d8-d6** or by **1. d8-d6 a4-c4 2. b1-b3**. Assume this position appears in a search tree. After exploring this position, we know its score and the best move. Because this position exist somewhere else in the tree it is beneficial to save the relevant information in a transposition table. In the next subsection we explain how we use transposition tables in LOA.

3.6.1 Hashing

We would like to save every position encountered in the search tree, but unfortunately this is not possible due to memory restrictions of most nowadays computers. Therefore a transposition table is implemented as a hash table using some hashing method.

In LOA there are only two different pieces (white and black) and 64 squares. For any combination of a piece and a square a random number is generated. Thus, in total 128 (64×2) random numbers are available. The hash value for a position is computed by doing an XOR operation on the numbers associated with the piece-square combination of that position. This method is called Zobrist-Hashing (Zobrist, 1970). It is not only fast, but it can also be done incrementally.

hashvalue (new position) = hashvalue (old position) XOR hashvalue (from_square_of_piece_moved) XOR hashvalue (to_square_of_piece_moved).

Normally we have to do only two operations in LOA. For capture moves, we have also to XOR the square of the captured piece.

If the transposition table consist of 2^n entries, the *n* lower-order bits of the hash value are used as a hash index. The remaining bits (the hash key) are used to distinguish among different positions mapping on the same hash index. In LOA we use a 63-bits hash value.

3.6.2 Use of a transposition table

The following traditional components are stored in the transposition table:

key	contains the hash key part of the hash value. There are 49 bits reserved for the key.
move	contains the best move in the position obtained from the search. There are 12 bits reserved for the best move.
score	contains the value of the position obtained from the search. The score can be an exact value, an upper bound or a lower bound. There are 16 bits reserved for the score.
flag	The flag indicates whether the score is an exact value, an upper bound or a lower bound. There are only two bits reserved for the flag.
depth	contains the depth of the subtree searched. There are 8 bits reserved for the depth.

Each entry in the transposition table is 11 bytes large. The key value is used to check if the retrieved board position is the same as the current. Sometimes using the key value can not detect that the retrieved board position is different. The danger exists that also the retrieved move is not valid. Therefore we check always if the from square is occupied by a piece of the moving side to prevent a breakdown of the program because we incrementally update the quads. This check is also used if the retrieved position is equal to the current one. We use the transposition table at three different levels:

1 The depth still to be searched is less than or equal to the depth retrieved from the table and the retrieved value is an exact value. The position does not have to be searched further and the retrieved value is returned. This is the main reason why transposition tables are used.

2 The depth still to be searched is less than or equal to the depth retrieved from the table and the retrieved value is not an exact value. The retrieved value can be used to adjust either the alpha value (if the retrieved value is a lower bound) or the beta value (if the retrieved value is an upper bound). Thus, we can use this value to adapt the alpha-beta window. A cut-off is possible, otherwise the retrieved move can be used as a first candidate, since it was considered best previously.

3 The depth still to be searched is greater than the depth retrieved from the table. The retrieved move is used as first move in the move ordering. Because iterative deepening is used, this situation occurs often. Iterative deepening and transposition tables are used in combination to speed up the alpha-beta search.

For a more detailed description of the algorithm, see (Marsland, 1986).

3.6.3 Probability of errors

Using a transposition table as a hash table introduces two types of errors. The first type of error is a type-1 error. A type-1 error occurs when two different positions have the same hash value. This mistake will not be recognised[1] and can lead to wrong evaluations in the search tree. This is something that we do not want to happen.

Let N be the number of distinguishable positions, and M be the number of different positions to be stored. The probability that this error will not happen is given by the following equation:

$$P(no\ errors) \approx e^{\frac{-M^2}{2N}} \tag{3.2}$$

As an example we consider our program, which searches 20000 nodes per second (nps). If it plays LOA using a total of two hours of thinking time, the number of nodes searched is 144×10^6. Assume that for about 30% of the nodes, an attempt is made to store them in the transposition table (Breuker, 1998). In this example, this is 43.2×10^6 nodes. If the hash value consists of 63 bits, the probability of at least one type-1 error is:

$$1 - e^{-\frac{43200000^2}{2\times 2^{63}}} \approx 1\times 10^{-4}$$

Because the transposition table is small, we cannot store all the positions occurred in the search. It happens that a position is to be stored in an entry, which is already occupied by another position. This is called a type-2 error or a collision. A choice has to be made which of the two involved positions should be stored in the transposition table. There are several replacement schemes (Breuker *et al.*, 1994), which tackle the collision problem. We use a TwoDeep scheme. This concept uses a two-level transposition table. Such a transposition table has two table positions per entry. If a collision occurs, the

[1] As we have seen in 3.6.2, we check if the piece to be shifted by the transposition table move really exists at that position. Thus, our program is able to detect sometimes this kind of mistake.

new position is always stored, and the one with the greatest depth of the remaining two is kept. Notice that for every new search process the transposition table is cleared.

3.7 Quiescence search

Mostly a certain depth limit is used as a cut-off test for the search. If the limit is reached the evaluation function is performed at that node. This approach can have some disastrous consequences because of the approximate nature of the evaluation function. Obviously a more sophisticated cut-off is needed. The evaluation function should only be applied to positions that are *quiescent*.

Tactical capture moves are responsible for swings in the evaluation function in LOA. A problem is that it is hard to define what a tactical capture move is because it depends on the board position. Mostly capture moves which destroy connections or form connections are considered as tactical. Therefore, we use a quiescence search that involves only that kind of moves. For example, if black moves **d7×d4** in figure 3.8, black will not only connect his pieces in the centre, but will also destroy white's connection. Remember that we can use the Euler-number approach to detect fast breaking up or forming connection capture moves. For example, if black's Euler number in figure 3.8 decreases, this means that more black pieces are connected. If the number increases, this means fewer pieces are connected. As we have seen in 3.1.2, the use of the Euler number to detect the number of isolated groups is not always correct. But it will not harm the use of the quiescence search.

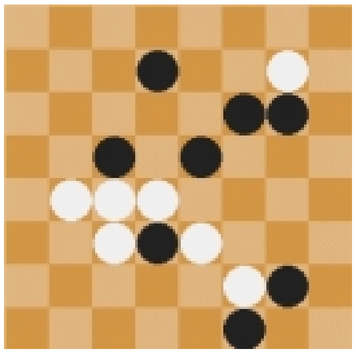


Figure 3.8: Tactical capture move.

In contrast to an exhaustive search, a quiescence search limits the set of moves to be considered and uses the evaluations of interior nodes as lower / upper bounds of the resulting search value. This mechanism works like the null move, but with the adjustment that no extra search is performed. Schrüfer calls this forward pruning although this is not the common term. For a detailed examination of the algorithm, see also (Schrüfer, 1989). Because the quiescence search is an alpha-beta search, move ordering would also be beneficial. Therefore we use for the quiescence search killer moves and history heuristic. Because quiescence search is a different search than the normal one, we also use separate history tables.

The use of a quiescence search in LOA is unconventional. The quiescence-search expansion, which only looks at capture moves that create or destroy connections, is a new concept. As we will see in chapter 5, this kind of quiescence search is effective.

3.8 Endgame databases

An *endgame* database stores all possible board positions for a given number. For each stored position the game-theoretic value is calculated. Because of the converging property of LOA an endgame database can be created. Schaeffer and Lake (1996) used an endgame database for checkers of 4.4×10^{11} positions with 8 or less stones. In LOA, this kind of database would only contain positions with maximally 7 pieces. This database would be 40 GB large. The problem is that in LOA most games end with more than ten pieces on the board (see also chapter 5). On the contrary with checkers and chess the (sub)goal of the game is not to capture enemy pieces. Most of the times you are doing your opponent a pleasure if you blindfoldly capture his pieces. Capture moves are used more tactically. For example, you capture an opposite piece if you can create a connected group or destroy an opposite connected group. Because capturing pieces is not the (sub)goal of the game, LOA ends with more pieces than chess and checkers. This is the reason for omitting the construction and use of endgame databases. For more information about endgame databases, see also (Van den Herik and Herschberg, 1985).

3.9 Other methods

In the literature there are several other alpha-beta enhancements. We will discuss them briefly in the rest of the section.

Forward pruning is a commonly-used technique. The idea is not to look further at certain moves, because they are considered bad. Mostly the problem is that we do not know for sure whether a certain move is bad. In old games like chess this kind of knowledge is reasonably developed, but in new games like LOA it is guesswork due to lack of experts.

The same problem we noticed before occurs also by constructing an *opening book*. This is a library of variations of the beginning phase of the game. In advanced chess programs this technique is commonly used. Although there exist a notion of good opening moves for LOA, which Handscomb (2000a) describes, this is not good enough to construct an opening book. There are too few grandmasters in LOA.

Another move-ordering technique is the *countermove heuristic* (Uiterwijk, 1992). This technique is based on the assumption that many moves have a "natural" response irrespective of the actual position in which the moves occur. The problem in LOA is that applicability of moves depends more on the board position than in chess.

3.10 Overall framework

In this chapter we have discussed several methods, some of which are used and some not. It is not clear in which order the used methods should best be performed. The pseudo code of Marsland (1986) and Donninger (1993) are combined. In which way alpha-beta, PVS, transposition tables, null move and tree construction are used is described by them. Before the null move is tried, the transposition table is used to prune a subtree or to narrow the window. As far as move ordering is concerned, the transposition move, if applicable, is always tried first. Next, the killer move is tried. All the other moves are ordered decreasingly according to their scores in the history table. If the scores are equal, non-outsider moves are preferred.

4 Evaluation functions

A fistful of evaluators

As we have seen in the previous chapters, a search is performed until a certain depth. Subsequently, the leaf nodes are given a value in accordance to the evaluation function. It is important that the evaluator gives us a good estimate of a position. In this chapter we will examine “LOA knowledge” and discuss the construction of the evaluation function.

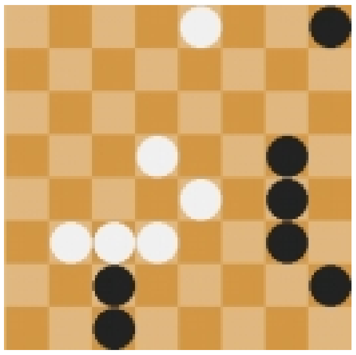


Figure 4.1: Threat position.

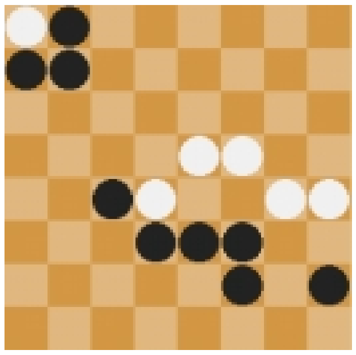


Figure 4.2: Blocked piece.

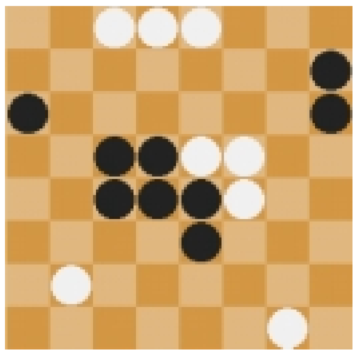


Figure 4.3: Solid formation.

4.1 General principles of LOA

Although knowledge about LOA is not well developed, there are some basic principles when playing LOA. Carl von Blixen (2000) has described on his homepages some guidelines, which we will take as a framework for developing an evaluation function.

- *Create a threat as quickly as possible.*

If a player can possibly connect all his pieces in one connected unit, this situation is called a *threat*. Also as the opposite party can prevent the connection to be made, the situation is still a threat, because the player *threats* to win. The first player to create a threat to win has a big advantage, because the opponent will have to break up his own formation to stop this threat. In figure 4.1 we see that white can win by the move **e8-e6**. Black can only prevent this by **g4×e4**, but he has to weaken his own formation.

- *Block opponent’s pieces*

Because one is not allowed to jump over opponent’s pieces, it is possible to *block* a piece. A piece is *completely* blocked, if it cannot move anywhere. Blocking a piece far away from the rest can be very effective. In figure 4.2 the opposite side has to waste many moves to first capture pieces around the blocked one at **a8** and try to connect again. During that time, black is likely able to create a threat. Even partial blocking can be very effective if it forces the opponent to take way around your pieces. Handscomb (2000a) describes that it is a common tactic to create a wall at the b- or g-file, or the 2nd or 7th row in the beginning phase of the game, which partially blocks pieces of the opposite side.

- *Connect pieces in multiple ways*

If pieces are connected in more than one direction it is harder for the opponent to break them up. In figure 4.3 black has created a rock solid formation in the centre of the board, which is hard to break up. The disadvantage is that it is hard to create this kind of formation fast. The danger exists that the opponent can make a threat first. Mostly it is good practise first to look if it is possible to connect a piece to a group of connected pieces and only secondly to see if a solid connection can be made.

These principles are good to have in mind when playing LOA. But they only say *what* is good in LOA, not *how* to *achieve* it. Some programmers have implemented a centralisation factor in their evaluator. Pieces in the centre are more awarded than pieces outside. Handscomb (2000b) argues that the benefits of centralisation in LOA are exaggerated. Because of the nature of the game, pieces huddled together in the middle are incapable of capturing each other. Even with only two pieces in a given line of action you need a distance of two squares for a capture, and if there are three pieces a considerable distance from the edge to the middle of the board is required. In figure 4.4 some pieces of white are scattered around the edges of the board. The white piece at **h4** can destroy black's formation in the middle by capturing the black piece **d4**. The white pieces at the edges of the board are acting as cruise missiles. This is not possible when all the pieces are in the centre.

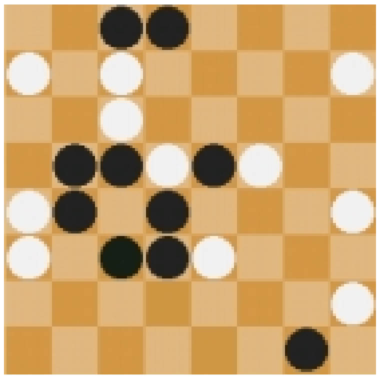


Figure 4.4: Roessner vs. Handscomb.

The last aspect we want to discuss in this section is capture moves. At first sight capturing opponent's pieces is not a good idea. The shortest game construction described in appendix B is created by capture moves, which reduces the pieces of one side such that it can quicker connect. However, it is not hard to see that capture moves in the endgame often are not bad. First, sometimes the only defence against a hostile threat is capturing a piece such that its formation is destroyed. Second, it also happens that the only way one is able to connect with its own formation is by a capture move. Third, a capture move is the only way to free one's blocked piece. Some authors argue that capturing pieces in the begin and middle game is also good. They think that a material advantage like in chess exists, because if a player has more pieces than his opponent, he has more chances to prevent hostile threats and create his own threats. The opponent has fewer possibilities to prevent threats. Whether material is really an advantage depends on the position of the pieces on the board. A material component in the evaluation function is dangerous in LOA, because the program might wildly capture pieces. A more subtle method has to be used, because sometimes a material disadvantage can be an advantage.

4.2 Loa evaluator

It is important that the evaluator is fast and a good estimator. The problem is to find those characteristics of LOA, which make the program play well. We have implemented three evaluators: *normal*, *quad* and *blocking*. The evaluator normal is also the core of quad and blocking. The range of every evaluator lies between –10000 and 10000.

4.2.1 Normal evaluator

This is the core of every evaluator implemented in the program. We have chosen a centre-of-mass approach. For each side, the centre of mass of the pieces at the board is computed. Next, we compute for each piece its distance to the centre of mass. The distance is measured as the minimal number of squares the piece has to travel going to the centre of mass. Subsequently, the average distance towards the centre of mass is calculated. Because only one piece can be in the centre of mass at the time, boards with few pieces are favoured. Therefore a recalculation is done which takes the number of pieces into account. The inverse of this average distance is defined as the concentration. The lower the average distance of the centre of mass, the higher the concentration. We favour boards where the pieces of the same colour are more in proximity of the centre of mass. For example, the centre of mass of black lies at **d5** and the centre of mass of white lies at **e5** in figure 4.5. It is not hard to see that black's average distance to the centre of mass is lower than white's. Black's formation is favoured above white's. This is according to real situation of the board. Notice that we do not look at connections. Because we favour boards where pieces are in the neighbourhood of each other, eventually they will be connected *in multiple ways*. This evaluator favours sometimes capture moves. For example, if **1. d1-d3** is played in figure 1.1, the program will respond with **a3×d3**. This move will not only improve its own concentration, but will also undermine the opponent's concentration, its pieces are still scattered around the board.

Also penalties are given for each piece at the edge, because they are easy to block. Finally, positions with a more centralised centre of mass are slightly favoured, preventing formations built at the edges of the board, which are easy to break up or to block.

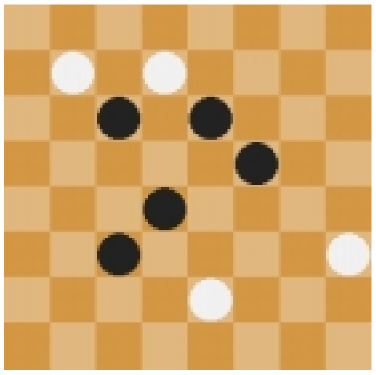


Figure 4.5: Concentrated pieces.

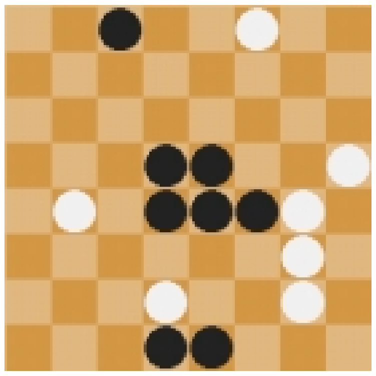


Figure 4.6: Solid centre formation.

4.2.2 Quad evaluator

This evaluator is called quad, because it uses the quad count. We can use the quad count in several ways. For example, we can easily compute the Euler number of each side. If we assume that holes do not occur, we have also computed the number of connected groups. It seems wise to take this factor into consideration. Because the goal of the game is to create one connected group, minimising the number of connected groups is maybe a subgoal of the game. However, at the start of the game this number is only two. Only bad moves can keep this number at two. Every sequence of normal moves in the beginning phase of the game will increase this number.

The previous evaluator has no notice of solid formations. Black's formation in the centre of the board is strong in figure 4.6. Formations with quads of three or four pieces are not possible to break up by a single capture. It seems reasonable to favour boards with many Q_3's and Q_4's. The danger exists that a lot of those quads are created outside the neighbourhood of the centre of mass. Therefore only Q_3's and Q_4's are rewarded, which lie at a distance of at most two of the centre of mass. Notice that this evaluator implicitly favours a material advantage.

4.2.3 Blocking evaluator

The problem with the previous two evaluators is that they are not able to detect blocking situations. For example, in figure 4.7 black has to move. If he does the move **f5-c5,** this will result in a better value of the evaluation function than doing **f5-e6**. But the piece at **c5** can not move any further to the centre of mass because of the white wall. However, the piece at **e6** is still able to move further. Therefore we have to take into account if pieces are partially blocked. If a piece outside the neighbourhood of the centre of mass cannot move a penalty is given. Also lower penalties are given for pieces, which have two moves or less. For pieces outside the region of the centre of mass is looked if they are able to move directly to the region of the centre of mass. For this evaluator we need to use our move generator. This is the reason why this evaluator is much slower than the others. Whether the benefit of better evaluations will outperform the disadvantage of searching less deep, will be explored in the next chapter. This evaluator has a notion of blocking but has its own faults: sometimes a piece can easily be freed by a capture move and be connected. For example, in figure 4.8 the black piece at **f8** is completely blocked. This board position is too negatively evaluated, because black can free this piece and win the game by **c4×f7**. Luckily, this kind of situation can easily be detected by the quiescence search, which is described in chapter 3.

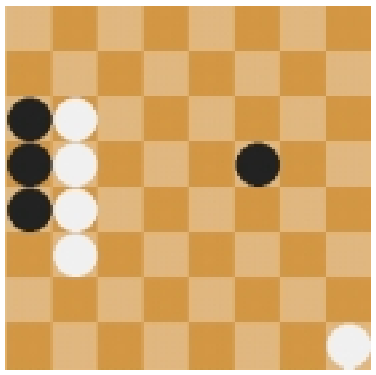


Figure 4.7: Wall position.

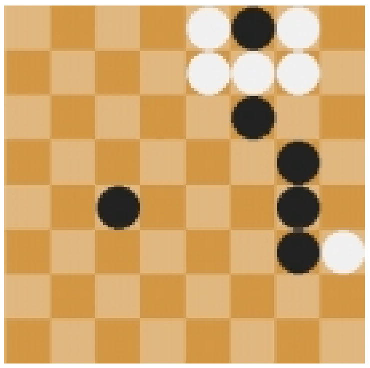


Figure 4.8: Pseudo blocking.

5 Testing and benchmarking

Once upon a time some tests

In this chapter some numerical results are given concerning the previous chapters. First, some characteristics of LOA are examined. Next, the usefulness of several alpha-beta enhancements is explored. Subsequently, the search extensions and evaluation functions are tested. Finally, the strength of the program is compared with others.

5.1 Properties of LOA

In this section some properties of LOA are given. The average branching factor, the development of the branching factor during the game, the average game length and the average total number of pieces in a final position is computed. For calculating these figures simulations were done. Our program played 200 times against itself on a P366 with 128 Mb SDRAM. The program used the configuration showed in table 5.1. Also a small random factor was added to each evaluation value.

Configuration	
Evaluator	Quad
Quiescence search	Yes
Transposition Table	Yes (2^{14} entries)
History heuristic	Yes
Killer Moves	Yes
Null Move	No
Time	5 sec. per move

Table 5.1: Game configuration.

5.1.1 Game length

In figure 5.1 the distribution of the game length in ply is given. We can see that the average game length is around 38 ply. This gives only an indication of the game length, it is *not* the definite number. Because the program played against itself, some disturbance could have been occurred. It is hard to verify this number, because there are no large LOA game databases. LOA experts guess the number is around 40, so this number gives a good indication. If we have a closer look at the figure we can see that the variation is high ($\sigma = 8.87$). We see that games longer than 65 ply are rare. Darse Billings noticed also in his simulations that long games are an oddity in LOA. This gives us an indication of an upper bound of the game length of LOA.

5.1.2 Total number of pieces at a final position

The significance of figure 5.2 is to show whether LOA often ends with a few pieces remaining at the board. If this occurs frequently an endgame database of seven pieces is profitable. A close look teaches us that games seldomly end with ten pieces or less. Thus, the use of an endgame database is small. The histogram shows us that games end normally with between 15 and 16 pieces. But again, these are not definite numbers because they are dependent on the evaluator and the search time.

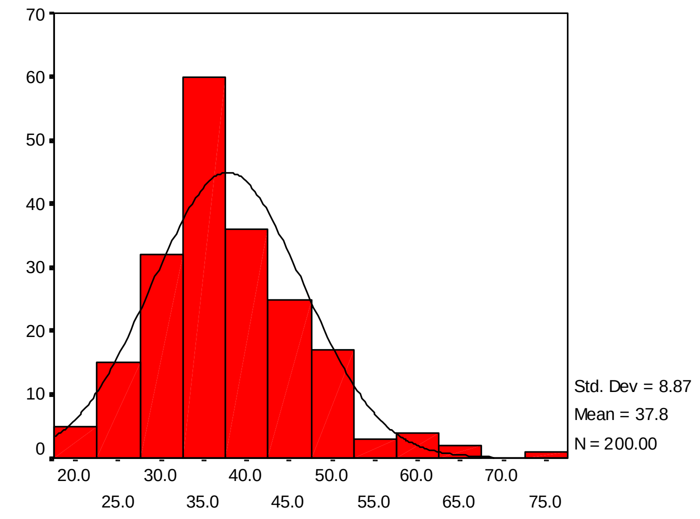


Figure 5.1: Game length in ply.

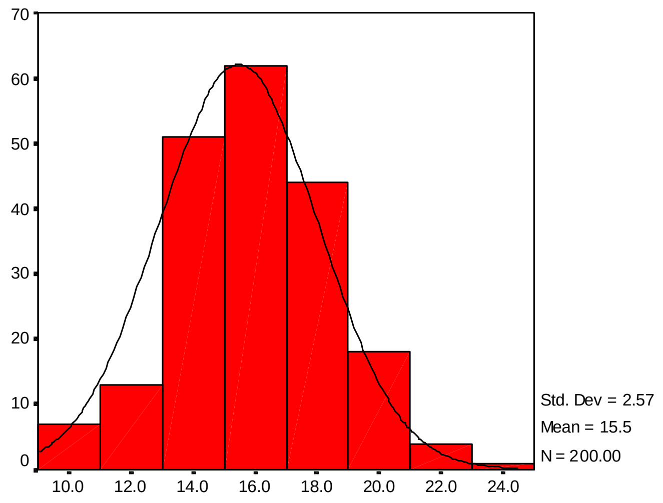


Figure 5.2: Number of pieces in a final position.

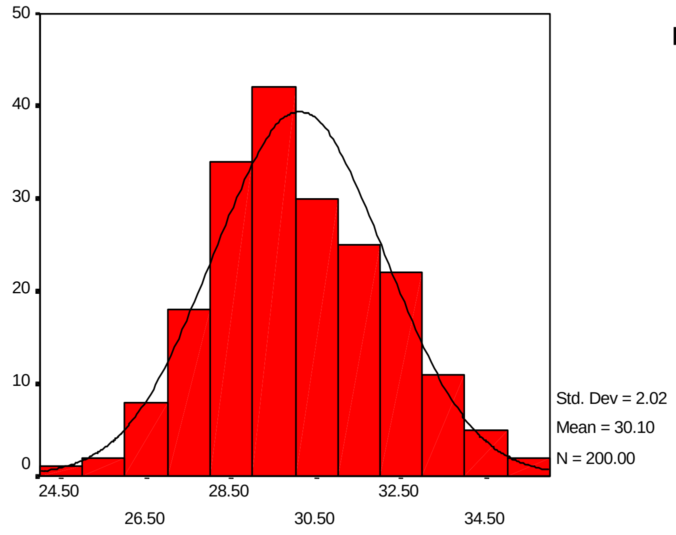


Figure 5.3: Average branching factor.

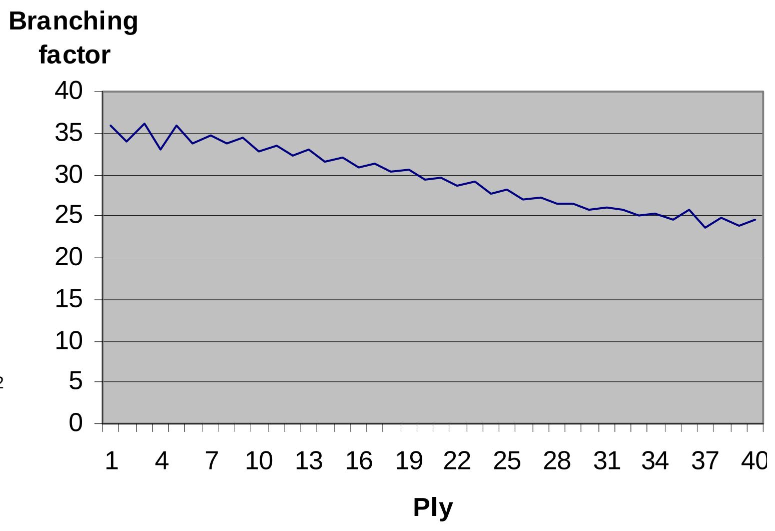


Figure 5.4: Development of the branching factor.

5.1.3 Branching factor

We calculated the average branching factor for *every* game. The distribution of these numbers is given in figure 5.3. The mean of the average branching factor is 30.1. Also the variation is low ($\sigma = 2.02$). This gives us confidence to take the average branching factor as 30. But the branching factor is dependent on the length of the game. We have already seen that we are not confident about this. What is the influence of the game length on the branching factor? In figure 5.4 the development in the game is given. In the beginning phase of the game the branching factor is around 35, but after nine plies the branching factor slowly decreases to 25. Because we do not expect that our game length computed in 5.1.1 is very different from the true game length, the change of the value of the branching factor will be low. Thus, if the game length is somewhat longer in reality, the average branching factor will still be around 30.

5.2 Search enhancements

In this section we are trying to get some insight into the usefulness of several alpha-beta enhancements. For all the experiments, the starting position was taken. The configuration represented in table 5.1 was used again. Except the transposition tables were 2^{19} entries large. Because we use a two-level scheme, we can store 2^{20} positions. In the tables given below, nodes, times and transposition table hits (TTHits) are cumulative counts. TTHits is defined as the number of times that a stored position was retrieved from the transposition table.

5.2.1 Transposition tables

In all the experiments the transposition tables are used as described in 3.6.2. In table 5.2, we can see that transposition tables gives a large speed-up. The usefulness of transposition tables improves when we go really deep. They are responsible for reducing the search-tree size in half at depth 7. This reduction by using transposition tables is comparable with the improvement noticed in a chess middle-game position (Uiterwijk, 1995).

	Without transposition table		**With transposition table**			
depth	# sec.	# nodes	# sec.	# nodes	# TTHits	% gain
1	0.0	58	0.0	58	0	0.0
2	0.1	1,436	0.1	1,344	3	6.4
3	0.7	4,292	0.7	3,975	41	7.4
4	1.3	18,738	1.3	11,841	195	36.8
5	8.5	151,899	6.1	105,241	2,027	30.7
6	58.8	1,059,405	34.1	579,640	19,764	45.2
7	513.4	9,242,547	224.4	4,039,648	116,005	56.3

Table 5.2: Results of using transposition tables.

There are two reasons why transposition tables work so well in LOA:

1) The retrieved positions have been found and stored in the same search. This is due to the fact that the same board position can be reached by different sequences of moves. These positions are transpositions. Normally, this is the first reason to use transposition tables. Some programmers argue that real transpositions are rare in LOA, because moving a piece will affect several other moves in that line. In LOA the independence between moves is smaller than in chess.

2) Because of iterative deepening, the same position will be reached in several iterations. The transposition table is a way of storing information about previously searched positions. When the same position is reached again, the search can be sped up or eliminated entirely by using this information. Many programmers believe this to be the reason why transposition tables work in LOA.

In the next experiment, showed in table 5.3, the transposition tables were cleared between the iterations. Looking at the table, we see that transpositions still give a significant reduction in the number of nodes searched. Since the gain roughly is half the gain when the tables are not cleared between iterations, it follows that both reasons given above to explain why transposition tables work well in LOA contribute equally.

	Without transposition table		With transposition table			
depth	# sec.	# nodes	# sec.	# nodes	# TTHits	% gain
1	0.0	58	0.0	58	0	0.0
2	0.1	1,436	0.1	1,399	2	2.6
3	0.7	4,292	0.7	3,918	2	8.7
4	1.3	18,738	1.2	15,273	91	18.5
5	8.5	151,899	7.5	139,420	1,390	8.2
6	58.8	1,059,405	51.3	921,917	18,561	13.0
7	513.4	9,242,547	384.1	6,914,145	176,899	25.2

Table 5.3: Results of using transposition tables, which are cleared between the iterations.

In the next experiment we have examined the influence of the size of the table on the total number of nodes to be searched. We did perform a 7-ply search at the start position. In figure 5.5 the relation is given between table size and the total number of nodes searched. As the table size increases, the number of nodes searched tends to converge to a constant. In other words, at some point no significant gain may be found by increasing the table size. According to Breuker (1998), this is caused by the larger percentage of tree nodes that can be retained in the transposition table. The probability of harmful collisions then greatly decreases. At a certain point the transposition table is sufficiently big to hold the entire search tree.

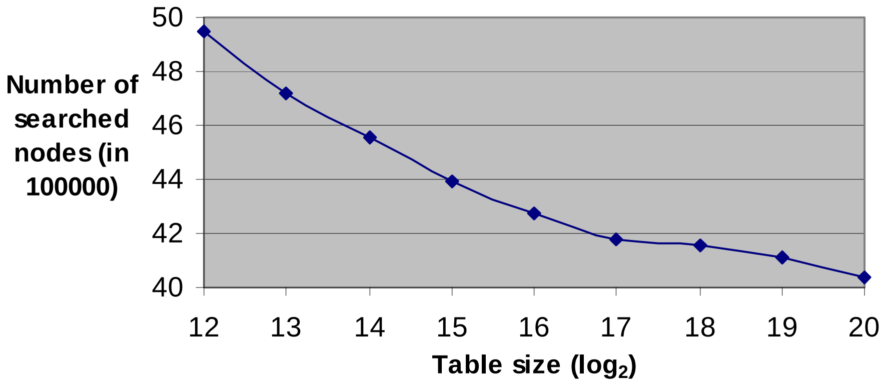


Figure 5.5: Comparing transposition tables with different sizes.

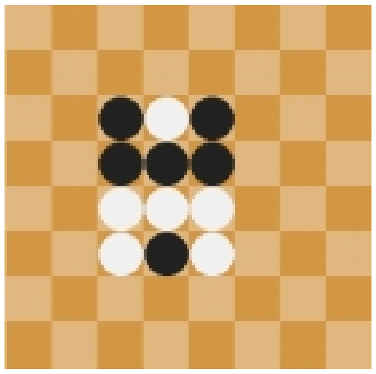


Figure 5.6: Endgame position.

The Bratko-Kopec positions in chess are famous, because due to transposition tables a search twice as deep can be performed. In figure 5.6 we present a LOA position where transposition tables are particularly useful. At depth nine a reduction of 85% is obtained, as is evident from table 5.4.

	Without transposition table		With transposition table			
depth	# sec.	# nodes	# sec.	# nodes	# TTHits	% gain
1	0.0	20	0.0	20	0	0.0
2	0.0	99	0.0	99	1	0.0
3	0.0	566	0.0	543	19	4.1
4	0.3	2,460	0.2	2,036	134	17.2
5	1.1	16,321	0.6	10,997	813	32.6
6	5.3	93,958	1.9	33,443	3,099	64.4
7	27.6	498,212	10.6	188,183	12,456	62.2
8	100.3	1,805,066	30.5	538,533	47,137	70.2
9	891.2	16,156,574	140.4	2,532,460	221,727	84.3

Table 5.4: Results of using transposition tables in the middle game.

5.2.2 PVS

Looking at table 5.5, we see that PVS gives us a considerable speed-up. The gain is not as big as by using transposition tables, but still useful. Although the gain due to PVS is changeable, it fluctuates around 20%. Thus, PVS provides that overall more nodes are pruned than re-searched.

	Without PVS		With PVS		
depth	# sec.	# nodes	# sec.	# nodes	% gain
1	0.0	58	0.0	58	0.0
2	0.4	1,531	0.1	1,344	12.2
3	0.8	5,502	0.7	3,975	27.8
4	1.5	15,987	1.3	11,841	25.9
5	8.9	160,531	6.1	105,241	34.4
6	77.1	668,656	34.1	579,640	13.3
7	289.5	5,211,899	224.4	4,039,648	22.4

Table 5.5: Results of using PVS.

5.2.3 Killer moves

If we have a look at table 5.6, we notice that killer moves do not make a difference when we go really deep. This can have several reasons. First, it could be that real killer moves do not often occur in LOA, as they do in chess. But another possibility could be that transposition tables have taken over the job. In the next experiment, we did not use transposition tables.

	Without killer move		With killer move		
depth	# sec.	# nodes	# sec.	# nodes	% gain
1	0.0	58	0.0	58	0.0
2	0.1	1,453	0.1	1,344	7.5
3	0.7	4,371	0.7	3,975	9.1
4	1.5	13,052	1.3	11,841	9.3
5	5.9	100,858	6.1	105,241	-4.3
6	33.9	574,248	34.1	579,640	-0.1
7	210.8	3,795,498	224.4	4,039,648	-6.4

Table 5.6: Results of using killer moves.

In table 5.7 we see that without transposition tables, killer moves are responsible for a gain of 39.8% at depth 7. Thus, killer moves are in principle useful in LOA. But they are dominated by transposition tables. Mostly the killer moves at the higher depths are responsible for big cut-offs. But because transposition moves are preferred above killer moves, they are only performed at lower depths. At those depths real killers are rare, which explains why killer moves can worsen things.

	Without killer move		With killer move		
depth	# sec.	# nodes	# sec.	# nodes	% gain
1	0.0	58	0.0	58	0.0
2	0.1	1,330	0.1	1,436	-8.0
3	0.8	4,171	0.7	4,292	-3.0
4	2.2	22,164	1.8	18,738	15.5
5	11.2	200,945	8.4	151,899	24.4
6	97.9	1,760,346	58.9	1,059,405	39.8

Table 5.7: Results of using killer moves without transposition tables.

5.2.4 History heuristic

Although the history heuristic is a simpler heuristic than transposition tables it works wonderfully good. The history heuristic is applied after the transposition and killer move. It is responsible for halving the search tree at depth 7 as seen from table 5.8, but its performance alters from depth to depth.

	Without history heuristic		**With history heuristic**		
depth	# sec.	# nodes	# sec.	# nodes	% gain
1	0.0	58	0.0	58	0.0
2	0.1	1,245	0.1	1,344	-8.0
3	0.8	4,337	0.7	3,975	8.3
4	1.4	13,211	1.3	11,841	10.3
5	12.4	222,400	6.1	105,241	52.7
6	49.3	891,115	34.1	579,640	34.9
7	433.4	7,797,603	224.4	4,039,648	51.8

Table 5.8: Results of using the history heuristic.

5.2.5 Outsider-move heuristic

In chapter 3 we have defined an outsider move as a move going to an edge field, these moves being examined as *last*. For testing if this heuristic is good, we compared plain alpha-beta with alpha-beta in combination with this move-ordering technique. In table 5.9, we notice that using this heuristic causes a reduction of 30%.

	Without any $\alpha\beta$ enhancements		**With only using outsider-move heuristic**		
depth	# sec.	# nodes	# sec.	# nodes	% gain
1	0.0	58	0.0	58	0.0
2	0.2	1,841	0.2	2,389	-29.8
3	0.9	13,347	0.9	13,574	-1.7
4	4.1	73,042	3.6	64,328	11.9
5	56.1	1,011,600	39.6	711,405	29.7
6	344.6	6,197,528	235.5	4,221,304	31.9

Table 5.9: Results of using only the outsider-move heuristic.

Using the outsider-move heuristic alone is a good thing to do. But it gives us no guarantee that it also works in combination with other methods. In table 5.10 we see that this heuristic provides a reduction in the earlier depths, but at depth 7 it causes extra work. At first sight it looks tempting to use the outsider move. But the history heuristic tables are empty in the beginning. During the game these tables are filled and by that way the program "learns" what good moves are. The usefulness of the outsider-move heuristic will diminish. This is the reason why the heuristic does not work at depth 7.

	Without outsider-move heuristic		**With outsider-move heuristic**		
depth	# sec.	# nodes	# sec.	# nodes	% gain
1	0.0	58	0.0	58	0.0
2	0.1	1,344	0.1	1,237	8.0
3	0.7	3,975	0.6	3,260	17.8
4	1.3	11,841	1.2	11,118	6.1
5	6.1	105,241	5.3	95,386	9.4
6	34.1	579,640	28.3	508,826	12.2
7	224.4	4,039,648	233.8	4,204,335	-4.1

Table 5.10: Results of using the outsider-move heuristic.

In the following experiment, we used the position achieved by the sequence of moves **d1-b3**, **h3-e3**, **b1-b4**. Because the program has pondered about the move **h3-e3**, it has built a search tree. The history table is now filled with scores. Looking at table 5.11, we see that the outsider-move heuristic has lost its positive effect. The history heuristic dominates the outsider-move heuristic.

	Without outsider-move heuristic		**With outsider-move heuristic**		
depth	# sec.	# nodes	# sec.	# nodes	% gain
1	0.0	71	0.0	59	16.9
2	0.1	676	0.1	667	1.3
3	0.3	3,334	0.3	3,402	-2.0
4	1.4	22,445	1.5	24,119	-7.5
5	3.5	64,697	3.5	65,165	-1.0
6	13.2	237,510	13.1	234,333	1.3

Table 5.11: Results of using the outsider-move heuristic after a couple of moves.

5.2.6 Null move

The last experiment is done with using the null move. The gain of null moves in the beginning phase of the game is not so big as the previous heuristics, but it is still profitable when the depth to be searched grows.

	Without null moves		**With null moves**		
depth	# sec.	# nodes	# sec.	# nodes	% gain
1	0.0	58	0.0	58	0.0
2	0.1	1,344	0.4	1,344	0.0
3	0.7	3,975	0.8	3,975	0.0
4	1.3	11,841	1.3	11,623	1.8
5	6.1	105,241	6.0	104,440	0.8
6	34.1	579,640	32.0	554,066	4.4
7	224.4	4,039,648	198.4	3,567,774	11.7

Table 5.12: Results of using null moves in the start position.

The results given in table 5.12, are typical for the beginning phase of the game, and hence do not give the null move full credit. In figure 5.7 a middle-game position is showed. If we now look at table 5.13 we notice that thanks to the null move a huge decrease of the number of nodes examined is accomplished.

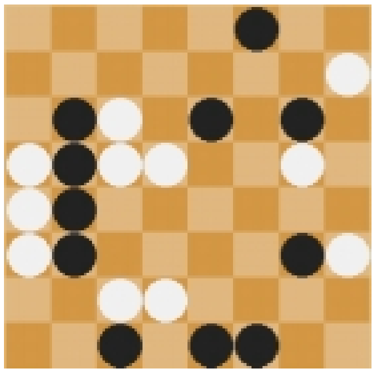


Figure 5.7: Handscomb vs. Thordsen.

In the middle game the negative effect of zugzwang does not occur often. Also in this board position no effects due to zugzwang were noticed.

	Without null moves		**With null moves**		
depth	# sec.	# nodes	# sec.	# nodes	% gain
1	0.0	58	0.0	58	0.0
2	0.0	594	0.0	594	0.0
3	0.2	2,508	0.2	2,508	0.0
4	0.6	9,767	0.5	8,024	17.8
5	2.2	40,843	1.4	24,129	40.9
6	11.4	204,952	6.3	111,983	45.3
7	46.8	841,617	13.5	242,397	71.2
8	179.2	3,223,975	58.0	1,053,489	67.3
9	1272.8	22,911,031	402.5	7,241,301	68.3

Table 5.13: Results of using null moves in the middle game.

Thanks to null move we can search roughly a ply deeper in the middle game. But the danger exists that due to null move defeats are not foreseen. Therefore, an experiment was conducted with the conditions described in table 5.1. The thinking time (5 sec.) was set low, so the advantage of searching a ply deeper due to null moves was lost. One player used the null move heuristic, the other did not use it. The player using the null moves won 104 of the 200 games, the other player won the rest. Thus, zugzwang has no negative effect. In a tournament using the null move will be an advantage because it will cause deeper searches.

5.3 Quiescence search

We have investigated whether performing a quiescence search is beneficial. Therefore we did some simulations. Two equal players using the quad evaluator played 200 times against each other. One player used quiescence search, the other did not. After 100 games, the players switched colour. If we look at the left side of table 5.14 we can see that using quiescence search is a great advantage. By using the quad evaluator both players look at connections. This is the reason why the player using the quiescence search wins. In the next experiment at the right both players used the normal evaluator. Both players do not value connections in their evaluator. The player using the quiescence search is still better off.

Quad Evaluator		**Normal Evaluator**	
Quiescence	No Quiescence	Quiescence	No Quiescence
125	71	134	65

Table 5.14: Comparing quiescence search with no quiescence search.

Doing a quiescence search costs some extra efforts, because we examine some more nodes. This means that the program might search less deep. In the next experiment we compared a search with and without quiescence search. Because of the quiescence search other values are returned at the leaf nodes, resulting in a different, possibly narrower, search tree. Looking at table 5.15 we see that the overhead noticed at earlier depths more or less disappears at depth 5 and beyond.

	Without quiescence search		With quiescence search		
depth	# sec.	# nodes	# sec.	# nodes	% overhead
1	0.0	37	0.0	58	56.8
2	0.1	250	0.1	1,344	437.6
3	0.5	1,951	0.7	3,975	103.7
4	1.1	6,389	1.3	11,841	85.3
5	6.1	103,924	6.1	105,241	1.4
6	34.2	614,255	34.1	579,640	-5.6
7	168.3	3,053,246	224.4	4,039,648	32.3

Table 5.15: Overhead of quiescence search.

5.4 Evaluators

In chapter 4 we have discussed the evaluators we have used. In this section they are evaluated against each other. Again we have used the configuration showed in table 5.1. All evaluators played against each other, switching sides halfly. The match results are given below.

Quad vs. Normal		Quad vs. Blocking		Normal vs. Blocking	
127	71	119	76	117	81

Table 5.16: Comparing the evaluators with each other.

We can conclude from the table that the quad evaluator is the best of the three. Interestingly, the quad evaluator defeats the blocking evaluator with no significantly higher numbers than the normal evaluator does. Because the normal evaluator defeats the blocking evaluator and is defeated by the quad evaluator, one should expect a glorious victory for the quad evaluator when playing against blocking. Possibly, both evaluators take advantage of the same weakness of the blocking evaluator.

When the normal and quad evaluators are used, the speed of the search performed is between 17000 and 18000 nodes per second (nps). If the blocking evaluator is used a speed of only 5000 nps is achieved. Thus, the blocking evaluator is a much slower evaluator, causing that normally roughly a ply less can be searched.

5.5 Results against other programs

Although the goal of the study is not to make the best LOA program in the world, we compared our program against others. We have only used recent programs playable at an IBM compatible PC. A description of the programs we used is given below.

- **Loa2D.** Benjamin Guihaire has written this program. The version currently used is 3.1, released in May 2000. The program is considered as lightweight. It has different evaluators, our program played against the *Normal* evaluator.
- **LoaW.** Dave Dyer and Ray Tayek have created this program. It was released in 1996. The program is written in C++.
- **Mona.** Darse Billings, who is a member of The University of Alberta GAMES Group, has created this program. The program was released in 2000. Mona has won of the best LOA players in the world. Still, the author of Mona feels it is too early to declare over-champion (significantly stronger than the human World Champion) status for Mona. The program is written in C.

- **YL.** Yngvi Björnsson, who is also a member of The University of Alberta GAMES Group, has constructed this program. This program is better than Mona. It searches deeper than Mona does. YL is written in C.

Our program MIA (Maastricht In Action) played with both colours against these programs. We used the same settings as showed in table 5.1, but this time with a thinking time of 30 sec. and the null move. The results are given below in the table 5.17. The games played are listed in appendix E.

MIA vs. Loa2D[1]	**MIA vs. LoaW**[2]
2-0	2-0
MIA vs. Mona[3]	**MIA vs. YL**[3]
2-0	0-2

Table 5.17: Comparing MIA with other programs.

Our program defeated Loa2D, LoaW and Mona. The first two are considered good-playing LOA programs, but humans can defeat them. MIA defeated Loa2D easily. The first victory against LoaW was also a walk through to the park. But, MIA came in serious trouble in the rematch, but won the match because it was able to first connect all its pieces in one group. Surprisingly, our program defeated Mona, which is considered as “world class” by its author. MIA was defeated but not slaughtered by YL. It was able to give a good opposition against YL. In the figures 5.8 and 5.9 we see the final positions of the matches MIA vs. YL and YL vs. MIA. Both times the final positions of MIA were not bad. MIA has built a solid formation in figure 5.8, but white was able to connect faster. MIA could not destroy the white connection, because YL huddled its white pieces against the black ones. In the return match MIA created four threats to win, which forced YL to do three times a capture move. This explains why black was able to build this formation in the middle and why white's pieces are disconnected.

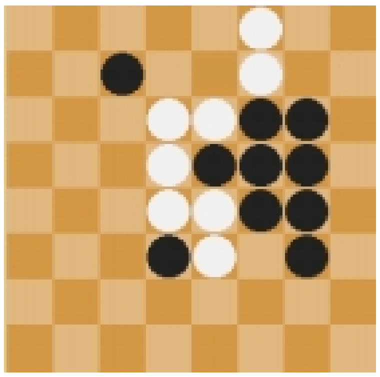


Figure 5.8: MIA vs. YL.

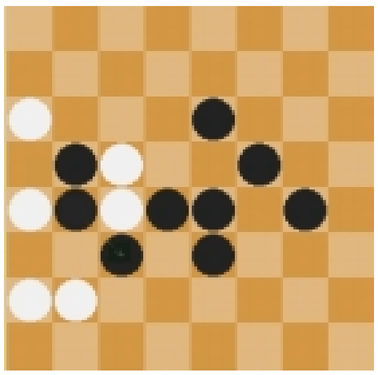


Figure 5.9: YL vs. MIA.

[1] In Loa2D it is not possible to set a certain thinking time, only a certain level. The level was set at 6 (hard), because it takes Loa2D approximately 30 sec. thinking time.
[2] In LoaW it is not possible to set a certain thinking time, only a certain level. The level of Loa2D was therefore set at 5 and the thinking time of MIA at 5 sec.
[3] Mona and YL do only run at the server of University of Alberta.

6 Conclusions

The ecstasy of conclusions

The evaluation of the problem statements, final conclusions, and future research are given in this chapter.

6.1 Problem statements revisited

In this thesis we have done research on the complexity of LOA for answering the problem statements mentioned in chapter 1.

What is the complexity of the game of Lines of Action?

The state-space complexity of LOA is 1.3×10^{24}. This is only an upper bound, because we have also calculated some unreachable positions. But we do not expect that the number will be much lower in reality and still be $O(10^{24})$. We have estimated the average branching factor at 30 and the game length at 38, which leads to a game-tree complexity of $O(10^{56})$. The state-space complexity and the game-tree complexity of LOA are comparable with those of Othello.

Is it possible to crack LOA?

Because the complexity of LOA is the same as Othello and this game is still not cracked, LOA will also not be cracked in the near future. Unless a certain knowledge pattern is found, LOA can not be solved. Because Othello programs have over-champion strength, this will probably also be the case for LOA programs soon. MIA is competitive with Mona and YL. Still it is too early to declare the over-champion status in LOA. According to the World Champion of 1999, Fred Kok, MIA is very good. His opinion is that computer programs will soon outperform human players. At the Mind Sports Computer Olympiad 2000, MIA got to know its place in LOA (see also appendix F).

6.2 General conclusions

Several conventional search enhancements were used. Killer moves exist in LOA and are only useful if transposition tables are not used. Transposition tables have taken over the job of killer moves in LOA. Transposition tables cause a reduction of 50% in the begin game and a reduction of 80% in some endgame positions. Thus, transposition tables are very useful in LOA. But situations like in chess where transposition tables cause a search twice as deep do not occur in LOA. In a search tree of a LOA position transpositions do occur but not so abundant as in some chess positions. Transpositions in LOA are considerably responsible for the reduction in the number of nodes examined due to transposition tables. The number of pruned nodes due to PVS is greater than the numbers of nodes re-searched. PVS is useful in LOA, but has not such a large effect as transposition tables. History tables work very well in LOA, their gain is comparable with that of transposition tables in the begin game. The gain of using null moves in the early phase of the game is small, but is huge in the middle game. Zugzwang does occur in LOA, and can have a negative effect when using null moves. But searching a ply deeper outweighs this. Thus, some conventional techniques do work in LOA. Endgame databases are not applicable in LOA, because there are too many pieces left in final positions. Constructing an endgame database of ten pieces would cost approximately 70 terabytes! The only game-specific move-ordering technique, which works, is the outsider-move heuristic described in chapter 3. But it lost its strength during the game when the history tables are filled.

Although LOA is a connection game, capturing pieces is a tactical concept. This is why the quiescence search described in chapter 3 is beneficial. LOA is not a trivial game and different evaluators can be constructed. A trade-off between search and knowledge has to be made. The less-knowledge-containing quad evaluator is better than the slow blocking evaluator. An evaluator should not only to be a good estimator but should also be fast.

6.3 Future research

In this section we do some recommendations for future research concerning the game of LOA.

Although we have calculated the complexities of LOA, these are still estimates. Concerning the state-space complexity, we have calculated some unreachable board positions in chapter 2. For example, we can do a Monte Carlo simulation to have a better number for the percentage of unreachable positions in LOA. Because LOA is played seriously more and more and LOA programs will also evolve, it is wise to estimate the average game length again. Because the level of play will increase, this figure will be more accurate.

The search methods can be more fine-tuned. For example, we use the null move during the whole game risking zugzwang. Why not disable null move in the endgame? The big question now is when the endgame starts. Also we can do more research concerning some parameters. The reduction factor R of the null move is set at two, because this is normally done in chess programs. But what is the effect of other values of R? There are lots of parameters in this thesis, which are arbitrarily chosen. We store one killer move, we divide the entries of the history tables by two during the search, we update the entries of the history tables by 2^d, etc. Not alone the used methods can be improved, some unused methods are maybe worth a try. We did not experiment with the countermove heuristic, because at first sight it seems useless in LOA, but we did also have some serious doubts about the null move, which was not justified after all. A challenge lies in the construction of an opening book in LOA. We hope that in the future more matches of a high level come available, such that it becomes easier to construct opening variants.

The evaluator is still a crude estimate of the goodness of a board position. Pattern matching can be used to detect good formations and blocking situations. Machine-learning methods can be examined for doing pattern matching in LOA. The techniques are promising, and have already achieved expert level in the field of Backgammon (Tesauro, 1994). They are currently under investigation in several other domains, including checkers, chess and Go. We would like to apply them in the game of LOA.

Appendix A: Notation

For people not familiar with the numbering of the board and the notation of the moves, this is explained in this appendix.

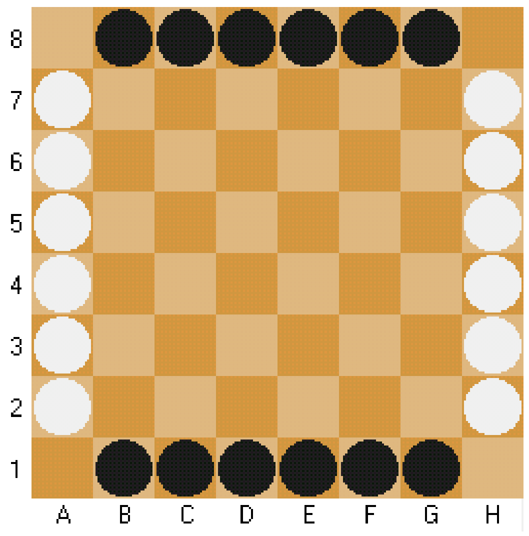
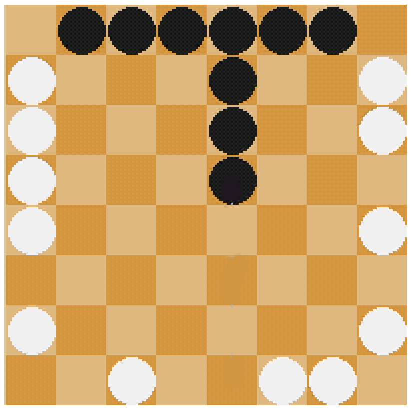


Figure A.1: Board numbering.

In figure A.1, the initial position is given. The horizontal lines are called rows and the vertical lines are called files. The names of the rows and files are determined by the numbers and letters, which stand at the edges of the board in figure A.1. For example, the square at the b-file and the first row is denoted as **b1.** Moving a piece from **b1** to **b3** is represented as **b1-b3**. Sometimes the short notation **b1b3** is used, but for the sake of clarity we use the full notation. A capture move is denoted by placing a × between the two fields. Thus, the notation of the four capture moves in this position is: **c1×a3**, **f1×h3**, **c8×a6** and **f8×h6**. In the short notation **:** is used between the fields.

Appendix B: Shortest LOA Game

The first shortest known game, was devised by Philip Cohen, and published in *November 1973 NostAlgia*. This ended with a suicide move by black. In December 1973, Mannis Charosh devised an improvement, which does not require the last move be a suicide play. This construction is given below.



Moves:

1	**d1-b3**	**h5-g4**
2	**b1-b4**	**g4×g1**
3	**b4-e7**	**a3×c1**
4	**b3-e6**	**h3×f1**
5	**e1-e5**	

Figure B.1: Shortest game construction.

Appendix C: Complexity of Games

In table C.1 the numbers of complexities are given, which are used in figure 2.5.

	State-space complexity ($\log_{10}$)	**Game-tree complexity ($\log_{10}$)**
Nine men's morris	10	50
Awari	12	32
Connect four	14	21
Checkers	18	31
Lines of Action	24	56
Othello	28	58
Qubic	30	34
Draughts	30	54
Amazons	40	220
Chinese Chess	48	150
Chess	50	123
Hex	57	150
Go-Moku	105	70
Renju	105	140
Go	172	360

Table C.1: Complexity of games.

Appendix D: Experimental Results

In section 5.1 the distribution of the average branching factor, game length and the number of pieces in a final position are given. In this appendix the figures of these distributions are given for the other simulations done in chapter 5. These figures do not differ very much of those in section 5.1. Also the number of times the black and the white side has won is given for each simulation.

Quiescence vs. No quiescence using both the quad evaluator

Quiescence vs.	No Quiescence
125	71

Black vs.	White
102	94

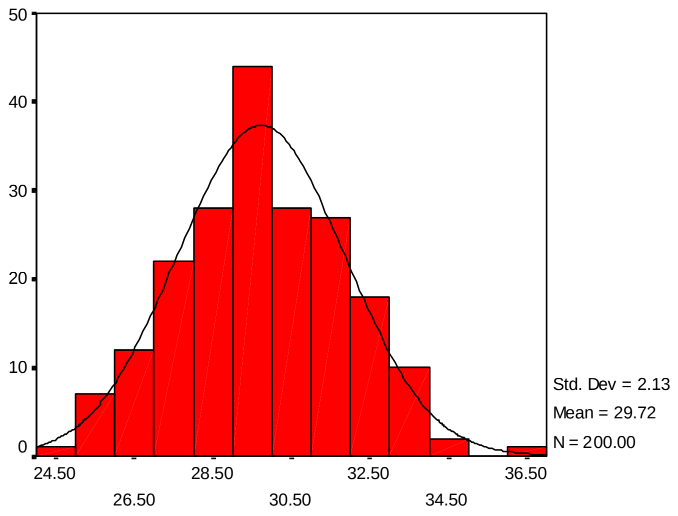
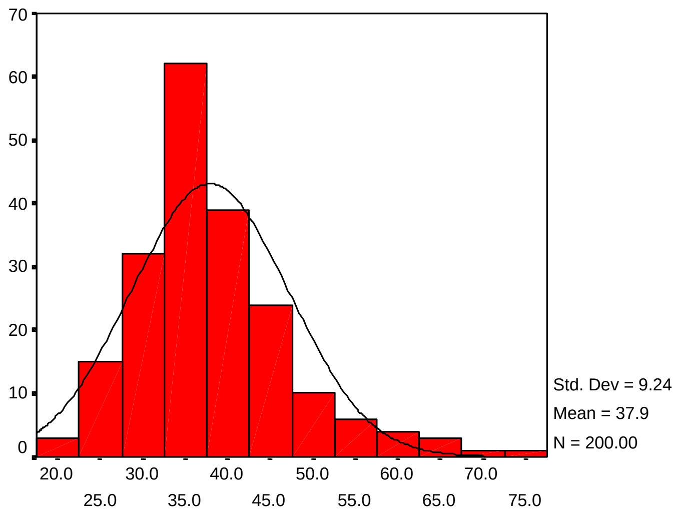


Figure D.1: Average branching factor.



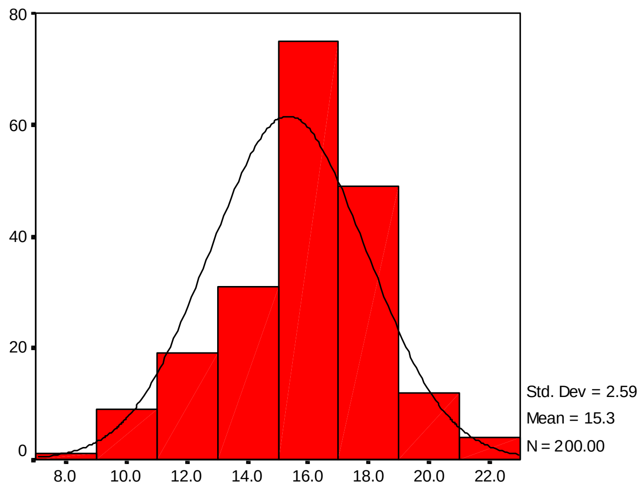


Figure D.2: Game length.

Figure D.3: Number of pieces in a final position.

Quiescence vs. No quiescence using both the normal evaluator

Quiescence	No Quiescence
134	65

Black	White
108	91

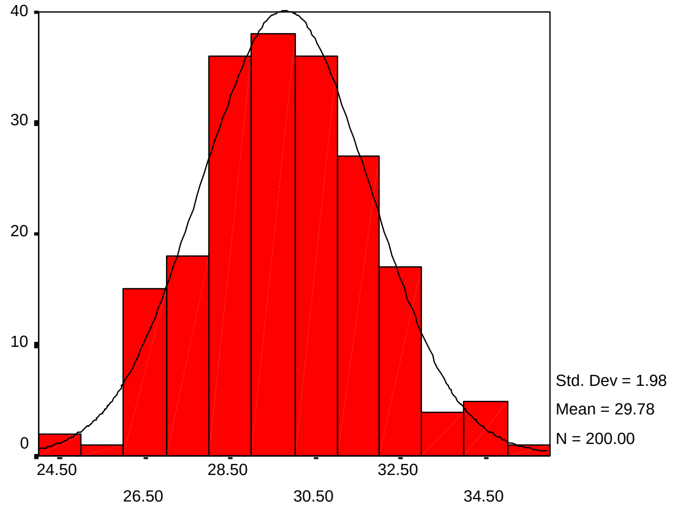


Figure D.4: Average branching factor.

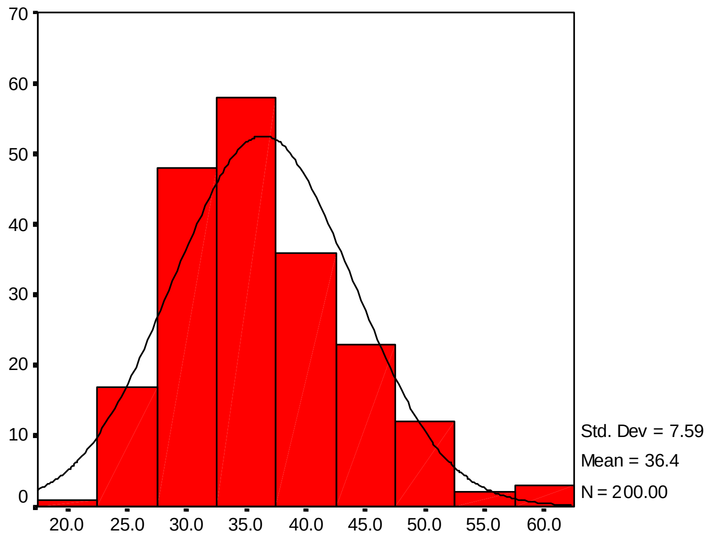


Figure D.5: Game length.

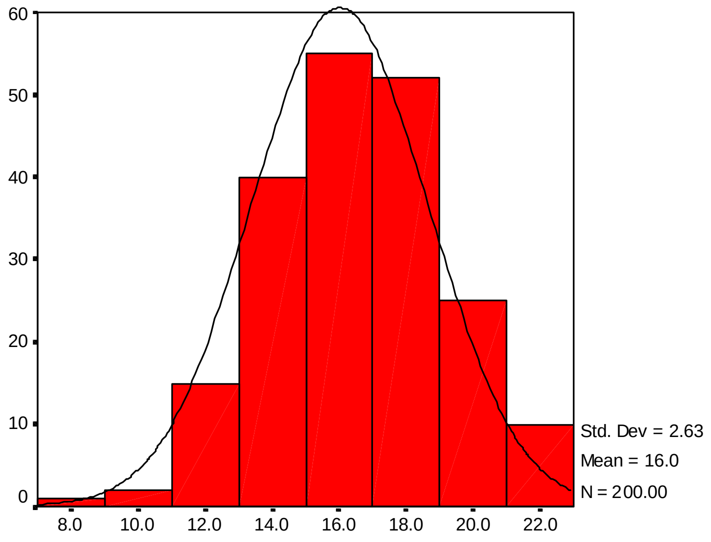


Figure D.6: Number of pieces in a final position.

Null move vs. No null move

Null move vs.	No Null Move
104	96

Black vs.	White
114	86

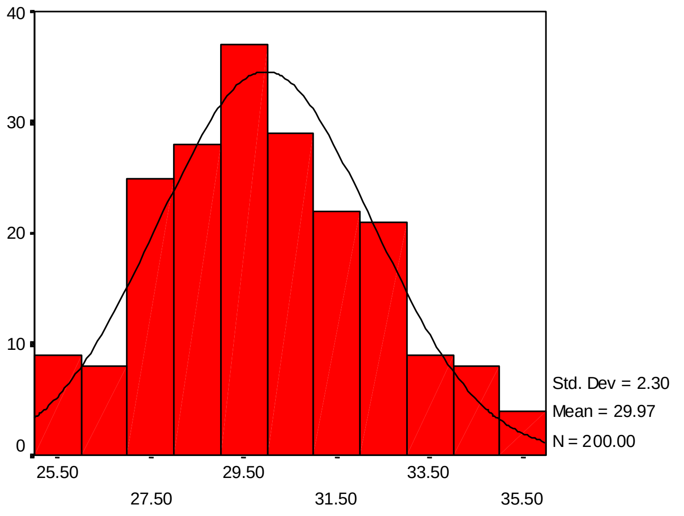


Figure D.7: Average branching factor.

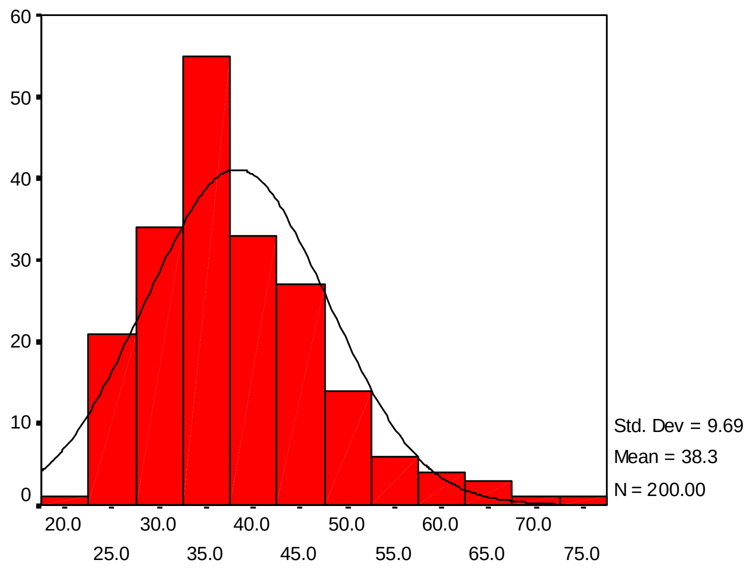


Figure D.8: Game length.

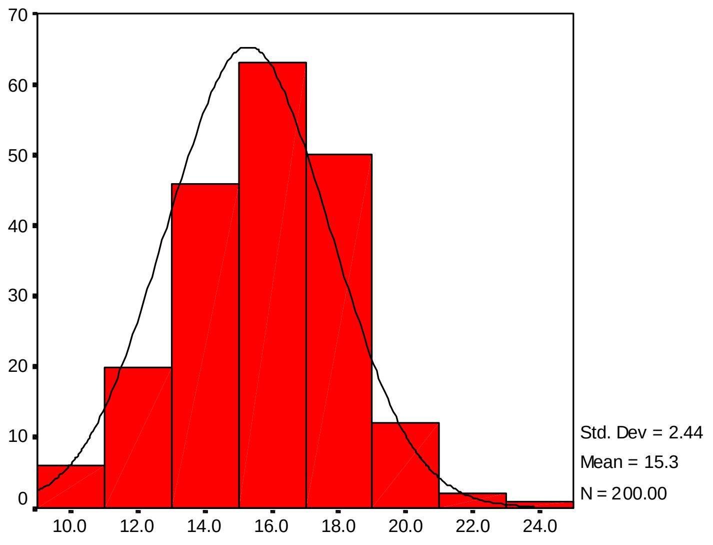


Figure D.9: Number of pieces in a final position.

Quad vs. Normal

Quad vs. Normal
127 71

Black vs. White
91 107

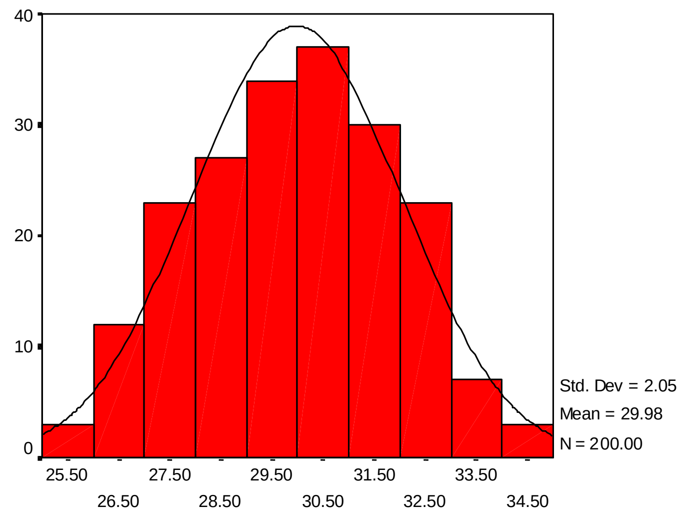
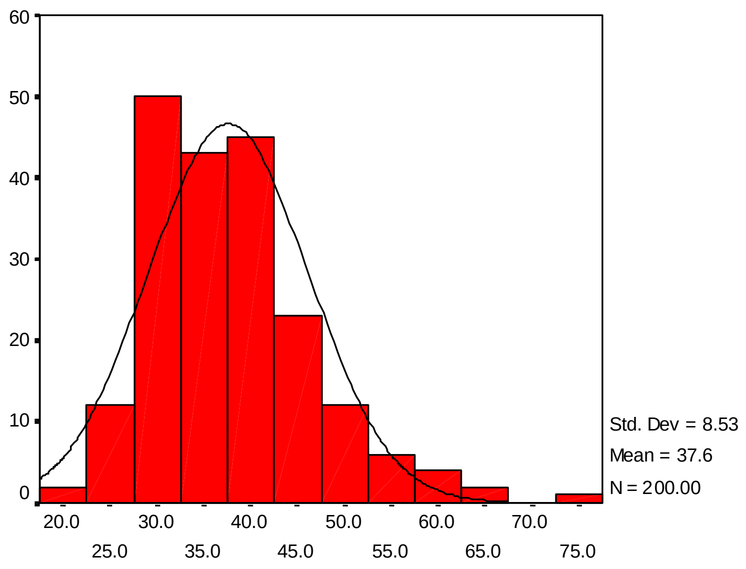


Figure D.10: Average branching factor.



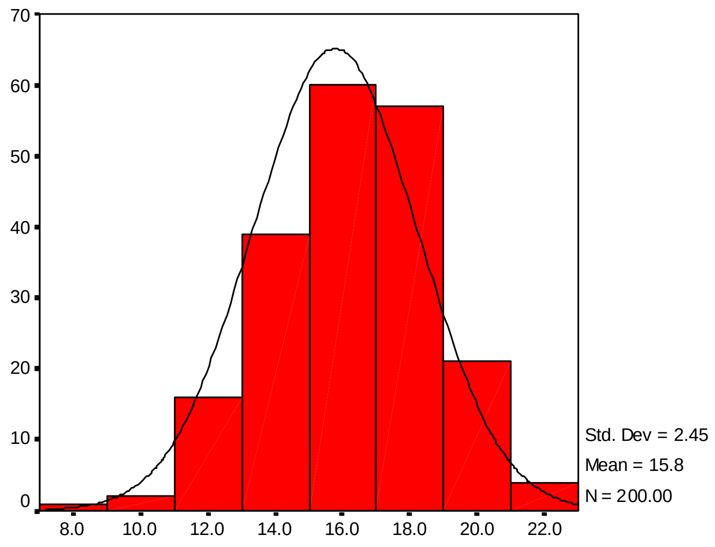


Figure D.11: Game length.

Figure D.12: Number of pieces in a final position.

Quad vs. Blocking

Quad vs.	Blocking
119	76

Black vs.	White
120	75

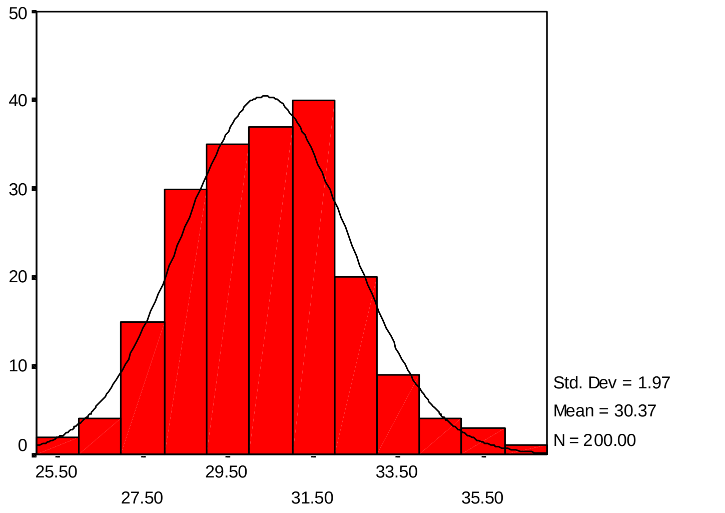


Figure D.13: Average branching factor.

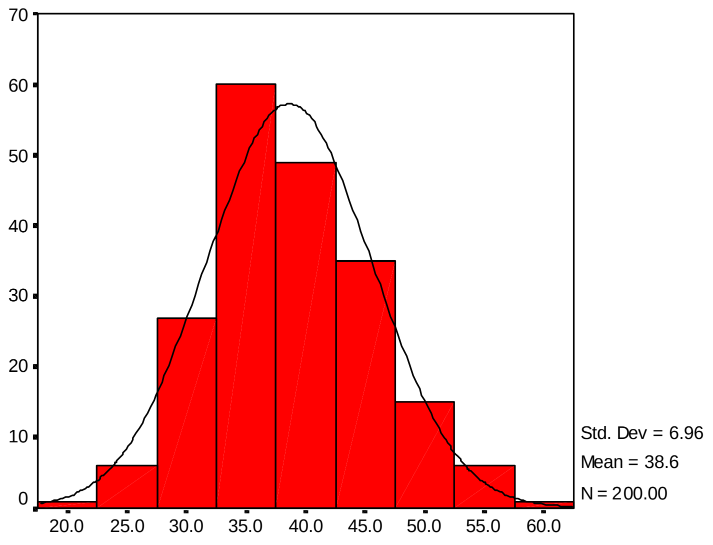


Figure D.14: Game length.

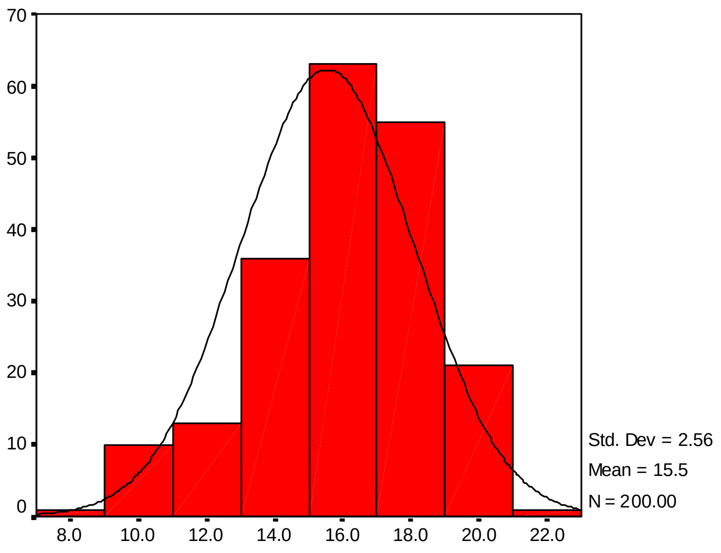


Figure D.15: Number of pieces in a final position.

Normal vs. Blocking

Normal vs.	Blocking
117	81

Black vs.	White
113	85

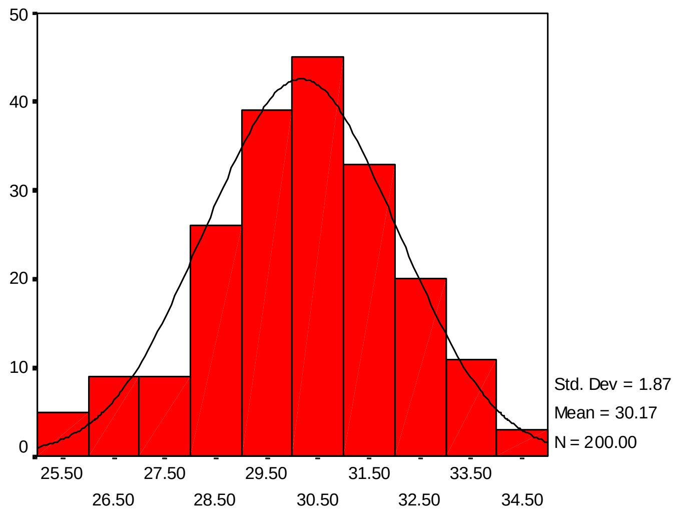
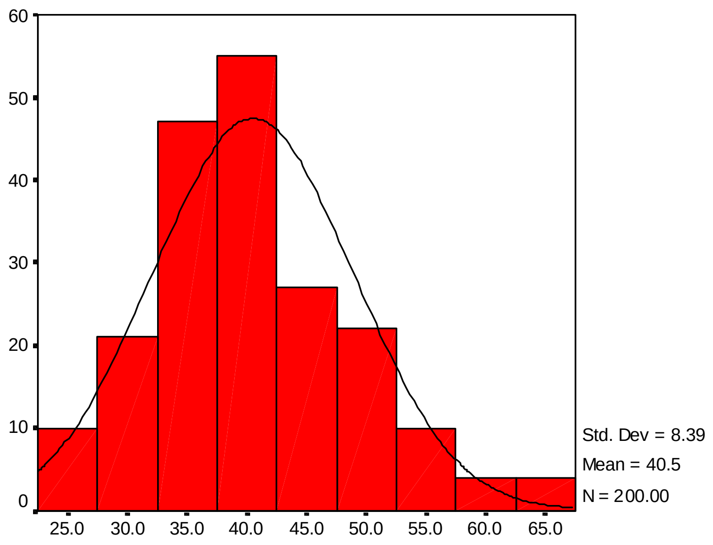


Figure D.16: Average branching factor.



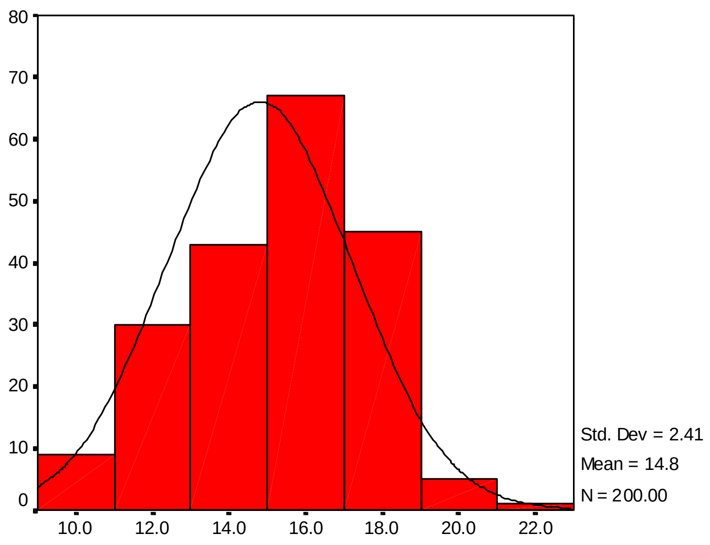


Figure D.17: Game length.

Figure D.18: Number of pieces in a final position.

Appendix E: Game Scores

The game scores of the matches played in section 5.5 are given in this appendix.

MIA vs. Loa2D			Loa2D vs. MIA		
1.	f1-f3	a3-d3	1.	c8-c6	h4-f2
2.	d1-g4	h2-f2	2.	g8-g6	h2-e2
3.	c8-f5	a7-d4	3.	b1-e4	a6-d3
4.	g8-g5	h6-f6	4.	c1-c3	a3-c5
5.	c1-e3	h5-f7	5.	e1-b4	a7-e3
6.	e1-e4	a6-c6	6.	f8×c5	a4-a1
7.	g1-h2	a4-d7	7.	d8-e7	a1×c3
8.	h2-f4	h7-e7	8.	e8×e3	a2-d2
9.	e3×e7	a2-c4	9.	b8-b6	h5-e5
10.	f3×c6	c4×c6	10.	f1-c1	a5-a4
11.	b1-b3	f6-d6	11.	g1-g3	a4×c6
12.	b3-d5	f7×d5	12.	c5×e5	c6×e4
13.	e8-e5	d5-e6	13.	c1×c3	h6×e3
14.	b8-e8	a5×e5	14.	b6×e3	e4-g4
15.	d8-g8	d3-b3	15.	e7-f6	d2-g2
16.	e6-g6	c6×g6	16.	f6-h6	d3×g6
17.	f4-h6	e5-g7	17.	d1-c2	g2-f3
18.	e8×g6	f2×f5	18.	h6-f4	h7-h5
19.	f8-f6	h3-f3			
20.	g8-h7				

Table E.1: MIA against Loa2D.

MIA vs. LoaW			LoaW vs. MIA		
1.	f1-f3	a2-c4	1.	b8-d6	a6×d6
2.	c1×c4	a6×c4	2.	g8-e6	h6×e6
3.	b1-d3	a7-c5	3.	g1-e3	a7-c5
4.	c8×c5	a3-a6	4.	c8×c5	a4-c4
5.	d1-d4	a5-c3	5.	c1×c4	a3-d3
6.	e1-g3	c4-e6	6.	d8-f6	h4×f6
7.	g1-g4	a6-d6	7.	b1-b2	h5-e5
8.	g8-g5	a4-c6	8.	e8-d7	h3×e3
9.	e8-g6	c6-e4	9.	f8-e8	h7-f5
10.	f8-f6	c3×c5	10.	e8-c6	h2-e2
11.	d8×h4	c5-c6	11.	f1-c1	e2×b2
12.	d3×d6	c6-c7	12.	d1-c2	e3-c3
13.	b8-b7	h2-g2	13.	c1-d2	a5-b6
14.	d4-c5	h7-e7	14.	d7-b5	f6-f4
15.	c5×c7	h6-g7	15.	e1-e4	b2×b5
16.	c7-c6	e4×g6	16.	e4-b4	f4×c4
17.	b7×e7		17.	c6-d5	d3-b3
			18.	c2-e4	e5-c7

Table E.2: MIA against LoaW.

MIA vs. Mona			Mona vs. MIA		
1.	f1-f3	h2-f4	1.	b8-b6	h6-e6
2.	b1-d3	h6-e3	2.	g1-g3	h5-f7
3.	d1-g4	h5-f5	3.	b1-d3	h4-f6
4.	e1-g3	a7-d4	4.	d8×f6	h7-e7
5.	g8-e6	e3×e6	5.	f8-g7	h2-f2
6.	e8×e6	d4-e5	6.	f1-c4	f2-c5
7.	f8-g7	h4-f6	7.	c1×c5	a4×c4
8.	d8×f6	h7-e4	8.	d3-d5	f7-b7
9.	g1-g5	a2-c4	9.	g8-d8	a3-d3
10.	e6×c4	a3×c1	10.	g7-e5	h3-g2
11.	c4-d5	a6-c4	11.	d8-d4	g2×d5
12.	c8-e8	e5×e8	12.	g3-e3	a2-b2
13.	b8-e5	e8×e5	13.	c8×e6	e7×c5
14.	d3×h3	e4-e6	14.	e6×a6	d3×a6
15.	d5-d6	c4×g4	15.	b6-b3	a5-b6
16.	d6-e7	a4-d4	16.	d1-b1	b2-c2
17.	h3-h4		17.	e8-e4	b7×b3

Table E.3: MIA against Mona.

MIA vs. YL			YL vs. MIA		
1.	f1-f3	h2-f4	1.	b1-b3	h3-e3
2.	b1-d3	h6-e3	2.	b8-b6	h7-f7
3.	d1-g4	h5-f5	3.	b3×e3	h6×e3
4.	e1-g3	h4-f6	4.	c1×e3	h4-f2
5.	b8-e5	a7-d4	5.	e1-e4	h2-e2
6.	d8×f6	a4-b3	6.	f1-g2	h5×e8
7.	f8-b8	a3-b4	7.	f8-d6	f7-d7
8.	f3-e4	b3-e6	8.	g1-g4	e8×e4
9.	b8×f4	a6-d6	9.	g2×e4	d7-b5
10.	c1-c3	h3-g2	10.	d1-d4	e2-b2
11.	g1-f1	g2×e4	11.	d4×b2	f2-c2
12.	f1×f5	a2-d5	12.	b2×b5	a7-a1
13.	g8-g5	h7-g7	13.	d8×a5	a1-b2
14.	e8-f7	a5-b6	14.	g8-e6	c2-c4
15.	c8-c6	b6-c7	15.	d6-b4	a3-c1
16.	c6-d7	g7-f8	16.	c8-f5	c1-c3
17.	c3-c5	b4-b5	17.	b6-d4	c3-c5
18.	c5×c7	b5×d7	18.	a5-c3	
19.	f7-g6	d7-f7			

Table E.4: MIA against YL.

Appendix F: Results at the MSO

MIA participated at the MSO of 2000. Unfortunately only two other programs participated: the strong LOA programs YL and Mona. MIA played four matches against each of them. The LOA programs had a total thinking time of 30 minutes for each match. The game scores are given below.

MIA vs. YL	**YL vs. MIA**	**MIA vs. YL**	**YL vs. MIA**
1. f8-f6 h4-f2	1. b1-b3 h3-e3	1. d1-b3 a5-c7	1. d1-b3 h3-e3
2. d8b6 h2-e2	2. b8-b6 h5-f7	2. b1-b4 a7-d7	2. b1-b4 a7-d4
3. b6×f2 a2-d2	3. b3×f7 h6-e6	3. g1-g3 h3-e6	3. c8-c6 h6-e6
4. f1-f4 h7-f5	4. c1×e3 h4-f2	4. g3×c7 h7-e7	4. c1×e3 h7-g6
5. c8×f5 h3×f5	5. c8×e6 h2-e2	5. e1-e5 h6×f8	5. b8-d6 h4-f2
6. e8-g6 a6-c4	6. d1-b3 a7-a1	6. c1-c4 f8×b4	6. e8-b5 h5-e5
7. g8-g5 c4×f4	7. d8-e7 a6-d3	7. b8-f4 a3-d6	7. d8-d5 h2-e2
8. c1-c2 e2-d3	8. e8×e2 h7-f5	8. f1-e1 a6×c4	8. g1-h2 e5-c3
9. b1-f1 h6-h4	9. b6×f2 a1-b2	9. e1×e6 a4×f4	9. h2-h3 c3-c1
10. g5-g2 a3-c3	10. f2×b2 a5-c5	10. e8-g6 b4-b6	10. f8-d8 c1×e3
11. g6-g3 d3×g3	11. g1-d4 f5-d5	11. b3-a3 b6-c5	11. d8×d4 f2-c2
12. f6-d4 h4×d4	12. g8-g7 d5-b5	12. g8×c4 h5×e5	12. g8×g6 a2-d2
13. b8-e5 a7-e3	13. b2×b5 a4-c4	13. g6×d6 h4-e4	13. e1-f2 c2-c4
14. e1-a1 f5-e6	14. e6-d6 a3-a1	14. a3-a5 e5-b5	14. c6×c4 e2-f3
15. a1-e1 a5×e1	15. e1-e5 a1-c1	15. d8-b8 f4-e3	15. h3-f5 e6-e4
16. f1×f4 e6-d5	16. d6×d3 c1-b2	16. b8-b6 a2-a4	16. f2×d2 f3×b3
17. c2-d3 a4-c2	17. g7-c3 b2×e2	17. a5-a7 e3-d2	17. c4-c3 a6×d6
18. d3-e2 h5×e2	18. f1-f4 e2-d1	18. c8-b7 h2-h1	18. f5-e6 e3×e6
19. MIA resigns	19. f4×c4 d1×b3	19. a7-b8 e7-d8	19. g6×e4 e6-c6
	20. e7-f6	20. e6×e4 d2×d6	20. f1-e1
		21. c4-d5 c5×c7	
		22. e4-a8 d6×b6	
		23. a8-a6 h1-e4	
		24. d5×b5	

Table F.1: MIA against YL at the MSO (1-3).

MIA vs. Mona	Mona vs. MIA	MIA vs. Mona	Mona vs. MIA
1. f8-f6 a5-c7	1 d1-b3 h3-e3	1. d1-b3 h4-f2	1. g1-g3 a3-d3
2. d8-g5 a7-d7	2 e1-g3 a7-d4	2. g8-d5 a5×d5	2. d8-b6 a4-c2
3. c1-f4 a4-c2	3 b1-b4 a2-c2	3. b1-b4 h2-e2	3. c8-c5 a2-d2
4. e1-g3 a2-d2	4 g1-g4 a5-c5	4. f8-c5 a7-c7	4. c5×c2 a5-c5
5. f6-f3 h2-e2	5 g8-g5 a4-c6	5. b8-b5 h7-f7	5. c2×c5 a7-c7
6. g8-g4 a6-d3	6 e8xc6 a6-c4	6. c8×a6 f7×b3	6. b8-b5 a6-a7
7. b1-b3 h7-e7	7 b8-b5 c2-e2	7. a6×e2 f2×c5	7. f1-f3 a7-d7
8. e8-g6 h3×f1	8 f1-d1 d4-e5	8. f1-c4 h3-g4	8. g8-f7 d3-g6
9. g6×d3 d7×d3	9 g4xc4 a3xc1	9. e1-e4 a3×c1	9. e1×h4 c7-f4
10. g1-f2 h4-h1	10 f8xc5 c1xg5	10. g1-e1 c1×e1	10. f8×f4 d2-d5
11. c8-b7 h1-e1	11 b3xe3 h7-g6	11. e8-c8 h6-g6	11. f7×d5 d7-f7
12. g5-e5 a3-c5	12 d8xg5 h2-f2	12. e2-e5 c7×e5	12. d1-g4 h7-g8
13. b8-b5 e1×e5	13 c5xf2 h5-f7	13. b4-b7 a2-b1	13. e8×g8 h5×d5
14. b7-b4 e5-d4	14 g3xg6 e2-g4	14. b7×b3 b1×e4	14. f3×d5 h2-g2
15. b3-c4 c7×f4	15 f2-f4 h4-g3	15. d8-b8 c5-b6	15. b1-d3 g2-f2
16. g3×d3 h6-e3	16 b5xe5 g3xe3	16. c8-c6 g6-d6	16. c5-f5 f2-e2
17. b4×d2 e7-e4	17 d1-f3 e3xe5	17. b8-a7 g4×c4	17. d3×g6 e2×g4
18. g4-g3 h5-h4	18 f3-d5 h6-h5	18. b3×b6	18. b6-e6 h3-g2
19. f3-h1 c5-b4	19 g6-f5 h5-h6		19. g8-h7 g2-e4
20. h1-f3 b4×d2	20 f5-e4 h6-f6		20. b5-e5 f7×h7
21. b5-b4 h4-g5	21 c8-c5		21. e6-c4 h6-g7
			22. c1-e3

Table F.2: MIA against Mona at the MSO (1-3).

YL was the strongest LOA program at the tournament. The tournament ranking is given below.

Rank	Program	Points
1	YL	6
2	Mona	4
3	Mia	2

Table F.3: Tournament ranking of MIA.

References

1. Akl, S.G. and Newborn, M.M. (1977). The Principal Continuation and the Killer Heuristic. *1977 ACM Annual Conference Proceedings*, pp. 466-473. ACM, Seattle.
2. Allis, L.V., Herik, H.J. van den and Herschberg, I.S. (1991). Which Games Will Survive? *Heuristic Programming in Artificial Intelligence 2: the second computer olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 232-243. Ellis Horwood, Chichester.
3. Allis, L.V. (1994). Searching for Solutions in Games and Artificial Intelligence. Ph.D. thesis Rijksuniversiteit Limburg, Maastricht, The Netherlands.
4. Anshelevich, V.V. (2000). The Game of Hex: An Automatic Theorem Proving Approach to Game Programming. To appear in the Proceedings of *AAAI-2000*.
5. Blixen, C. von (2000). Lines-of-Action. *http://www.student.nada.kth.se/~f89-cvb/loa.html*.
6. Bouton, C.L. (1901). Nim, a Game with a Complete Mathematical Theory. *Annals of Mathematics*, Vol. 2, No.3, pp. 33-39.
7. Breuker, D.M., Uiterwijk, J.W.H.M. and Herik, H.J. van den (1994). Replacement Schemes for Transposition Tables. *ICCA Journal*, Vol. 17, No. 4, pp. 183-193.
8. Breuker, D.M. (1998). Memory versus Search in Games. Ph.D. thesis. Universiteit Maastricht, Maastricht, The Netherlands.
9. Brudno, A.L. (1963). Bounds and Valuations for Abridging the Search of Estimates. *Problems of Cybernetics*. Vol. 10, pp. 225-241.
10. Donninger, C. (1993). Null Move and Deep Search: Selective-Search Heuristics for Obtuse Chess Programs. *ICCA Journal*, Vol. 16, No. 3, pp. 137-143.
11. Dyer, D. (2000). Lines of Action Homepage. *http://www.andromeda.com/people/ddyer/loa/loa.html*.
12. Gray, S.B. (1971). Local Properties of Binary Images in Two Dimensions. *IEEE Transactions on Computers*, Vol. C-20, No. 5, pp. 551-561.
13. Groot, A.D. de (1946). Het Denken van den Schaker, een Experimenteel-psychologische Studie. Ph.D. thesis, University of Amsterdam, Amsterdam, The Netherlands. In Dutch.
14. Handscomb, K. (2000a) Lines of Action Strategic Ideas – Part 1. *Abstract Games*. Vol. 1, No. 1.
15. Handscomb, K. (2000b) Lines of Action Strategic Ideas – Part 2. *Abstract Games*. Vol. 1, No. 2.
16. Hartmann, D. (1988). Butterfly Boards. *ICCA Journal*, Vol. 11, Nos. 2/3, pp. 64-71.
17. Herik, H.J. van den (1983). Computerschaak, Schaakwereld en Kunstmatige Intelligentie. Ph.D. thesis, Delft University of Technology. Academic Service, Den Haag, The Netherlands. In Dutch.
18. Herik, H.J. van den and Herschberg, I.S. (1985). The Construction of an Omniscient Endgame Data Base. *ICCA Journal*, Vol. 8, No. 2, pp. 66-87.
19. Huberman, B.J. (1968). A Program to Play Chess End Games. *Technical Report no. CS-106*. Ph.D. thesis. Stanford University, Computer Science Department, USA.
20. Knuth, D.E. and Moore, R.W. (1975). An Analysis of Alpha-beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293-326.
21. Maltell, T. (2000). Renju International Federation. *http://www.lemes.se/renju/*.
22. Marsland, T.A. (1986). A Review of Game-Tree Pruning. *ICCA Journal*, Vol. 9, No. 1, pp. 3-19.
23. Minsky, M. (1968). Semantic Information Processing. M.I.T. Press, Cambridge, MA, USA.
24. Minsky, M. and Papert, S. (1988). Perceptrons: An Introduction to Computational Geometry. M.I.T. Press, Cambridge, MA, USA.
25. Newell, A. and Simon, H.A. (1972). Human Problem Solving. Prentice-Hall Inc., Englewood Cliffs, NY, USA.
26. Nilsson, N.J. (1971). Problem-Solving Methods in Artificial Intelligence. McGraw-Hill Book Company, New York, NY, USA.

27. Russell, S. and Norvig, P. (1995). Artificial Intelligence: A Modern Approach. Prentice-Hall Inc., Englewood Cliffs, NJ, USA.
28. Sackson, S. (1969). A Gamut of Games. Random House, New York, NY, USA.
29. Samuel, A.L. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, Vol. 3, No. 3, pp. 210-229.
30. Schaeffer, J. (1983). The History Heuristic. *ICCA Journal*, Vol. 6, No. 3, pp. 16-19.
31. Schaeffer, J. and Lake, R. (1996). Solving the Game of Checkers. *Games of No Chance* (ed. J. Nowakowski). *MSRI Publications,* Vol. 29, pp. 119-133. Cambridge University Press, Cambridge, UK.
32. Schaeffer, J. (1997). One Jump Ahead: Challenging Human Supremacy in Checkers. Springer-Verlag, New York NY, USA.
33. Schaeffer, J. and Plaat, A. (1997). Kasparov versus Deep Blue: The Rematch. *ICCA Journal*, Vol. 20, No.2, pp. 95-101.
34. Shannon, C.E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, Vol. 41, No. 7, pp. 256-275.
35. Schrüfer, G. (1989). A Strategic Quiescence Search. *ICCA Journal*, Vol. 12, No. 1, pp. 3-9.
36. Tesauro, G. (1994). TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play. *Neural Computation*, Vol. 6, No. 2, pp. 215-219.
37. Thompson, M. (2000). TwixT. *http://home.flash.net/~markthom/html/twixt.html*.
38. Turing, A.M. (1953). Digital Computers Applied to Games. *Faster than Thought* (ed. B.V. Bowden), pp. 286-297. Pitman, London, UK.
39. Uiterwijk, J.W.H.M. (1992). The Countermove Heuristic. *ICCA Journal*, Vol. 15, No. 1, pp. 8-15.
40. Uiterwijk, J.W.H.M. (1995). Déjà vu. *Computerschaak*, Vol.15, No. 2, pp. 84-91. In Dutch.
41. Uiterwijk, J.W.H.M. and Herik, H.J. van den (2000). The Advantage of the Initiative. *Information Sciences*, Vol. 122, No. 1, pp. 43-58.
42. Von Neumann, J. and Morgenstern, O. (1944). Theory of Games and Economic Behavior. Princeton University Press, Princeton, NJ, USA.
43. Zobrist, A.L. (1970). A New Hashing Method with Application for Game Playing. Technical Report 88, Computer Science Department, The University of Wisconsin, Madison, WI, USA. Reprinted (1990) in *ICCA Journal*, Vol.13, No. 2, pp. 69-73.