New Architectures in Computer Chess

New Architectures in Computer Chess

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Tilburg,
op gezag van de rector magnificus,
prof. dr. Ph. Eijlander,
in het openbaar te verdedigen ten overstaan van een
door het college voor promoties aangewezen commissie
in de aula van de Universiteit
op woensdag 17 juni 2009 om 10.15 uur

door

Fritz Max Heinrich Reul
geboren op 30 september 1977 te Hanau, Duitsland

Promotor:	Prof. dr. H.J. van den Herik
Copromotor:	Dr. ir. J.W.H.M. Uiterwijk
Promotiecommissie:	Prof. dr. A.P.J. van den Bosch
	Prof. dr. A. de Bruin
	Prof. dr. H.C. Bunt
	Prof. dr. A.J. van Zanten
	Dr. U. Lorenz
	Dr. A. Plaat

 Dissertation Series No. 2009-16

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN 9789490122249

Printed by Gildeprint

Preface

About five years ago I completed my diploma project about computer chess at the University of Applied Sciences in Friedberg, Germany. Immediately afterwards I continued in 2004 with the R&D of my computer-chess engine Loop.

In 2005 I started my Ph.D. project "New Architectures in Computer Chess" at the Maastricht University. In the first year of my R&D I concentrated on the redesign of a computer-chess architecture for 32-bit computer environments. I developed more efficient search methods and more precise evaluation functions in order to carry out my R&D within the scope of a state-of-the-art computer-chess environment.

In fall 2006 the development of my new 32-bit computer-chess architecture was completed. With the opening book by Gerhard Sonnabend and the hardware provided by Clemens Keck my computer-chess engine Loop Leiden achieved the 2^{nd} place behind Rybka and before Hiarcs X MP at the 26^{th} Open Dutch Computer-Chess Championship 2006 in Leiden (NL) (see Appendix C.1).

In 2007, I started the next R&D phase on the basis of this computer-chess engine. The focus of this phase was on the development of a 64-bit computer-chess architecture that should be used within the scope of a quad-core computer-chess engine. The new 64-bit computer-chess engine, Loop Amsterdam, achieved the 3^{rd} place behind Rybka and Zappa but before Shredder at the 15^{th} World Computer-Chess Championship 2007 in Amsterdam (NL) (see Appendix C.2).

For the persons who continuously supported me some words of gratitude are appropriate. In particular, I am grateful to my supervisor Professor Jaap van den Herik who brought me with firm guidance to this success. He stimulated me to continue the R&D of my computer-chess engine Loop Chess and motivated me to write this thesis. In addition, he set up the contact to the company Nintendo in 2007 which implemented my source codes successfully in their commercial product Wii Chess. Moreover, I would like to thank my daily advisor dr. Jos Uiterwijk for the many crucial suggestions and tips during the reading of my thesis.

My thanks goes also to Dr. Christian Donninger (Chrilly) for his advisory input during the development of a parallel search engine and to Tord Romstad for his assistance in generating magic multipliers. Gerhard Sonnabend and Clemens Keck helped me in the experiments that lasted many months and particularly in the preparation for two important computer-chess championships, Great Work!

Finally, I am grateful to Viktoria for her endurance and for her help on translating the contents of the thesis. I feel great to my parents: in the thesis they see the results of their education.

Fritz Reul, 2009

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer chess is one of the oldest research areas in artificial intelligence [52, 55]. Although the playing strength of the best computer-chess engines has surpassed the playing strength of the human World Chess Champion, computer chess is still a challenging and intriguing research area.[1] The main issue is the complexity of chess knowledge. Therefore, the main research objective is: how can we develop new computer-chess architectures that adequately deal with the complex chess knowledge.

This thesis analyses and derives the most important requirements, objectives, rules, and theories for the development of computer-chess architectures. Apart from theoretical research and scientific scrutiny, the relation of our research to the computer-chess architecture is established by the implementation of algorithms. We do not consider any theory in its own preserve. Every theory or rule is always considered in a complex context in close relation to the other elements of a state-of-the-art computer-chess architecture.

This chapter is structured as follows. Section 1.1 introduces the notion of architectures in computer chess. Section 1.2 contains preliminary considerations. The problem statement and three research questions are given in Section 1.3. Section 1.4 discusses briefly the research methodology. Section 1.5 provides the background information of the engine. Section 1.6 gives the thesis overview.

1.1 Architectures in Computer Chess

The computer-chess architecture together with the design of search algorithms and the evaluation algorithms (and other high-level algorithms) [40] are the basis for a state-of-the-art and strong computer-chess engine. In this thesis the development of a computer-chess architecture should not begin until the most important requirements and objectives of the computer-chess engine have been exactly defined and determined. Computer-chess architectures that are developed only for a single objective have proven to be insufficient within the complex scope of a computer-chess engine. Consequently, it is not adequate

[1] In 1997 Kasparov was defeated by the chess machine Deep Blue which played on a supercomputer. In 2006 Kramnik was defeated by the computer-chess engine Deep Fritz.

to consider specific qualities of a computer-chess architecture separately and to draw conclusions from that. Hyatt [25] has already recognized this issue and formulated it as follows:

> *"If move generation and attack detection were the most time-consuming parts of a typical chess program, the 0x88 approach would be difficult to improve on."*[2]

Computer-Chess Architecture and Computer-Chess Engine

We remark that the elements of a typical computer-chess architecture have a deterministic origin and compute exact results, such as (1) move information (→ move generators), (2) attack information (→ attack detectors), (3) board information (→ doing and undoing moves), and (4) square information (→ static exchange evaluator). Furthermore, our computer-chess architecture is to be understood as a module implementing (1) the basic board representation, (2) the basic piece representation, and (3) the elementary algorithms. Further details and a comparison with other functions and modules of a computer-chess engine are discussed in Section 2.2.

There is a clear difference between a computer-chess engine and a computer-chess architecture. A computer-chess engine is the composition of (1) the computer-chess architecture, (2) the sequential and parallel search, (3) the static and interactive evaluation, (4) the transposition tables, (5) the move ordering algorithms, (6) the time management, and (7) the user interface. A computer-chess architecture composes all algorithms and data structures of a computer-chess engine as mentioned above.

Chess Knowledge

The complexity of chess knowledge increases disproportionately in relation to the game performance expected to be achieved through further development of a computer-chess engine. Whereas only some years ago (1985 to 1996) computer-chess engines, such as GNU-Chess [12], implemented a central move generator, nowadays at least four specific move generators are used (see, e.g., Fruit by Letouzey [38], Glaurung by Romstad [49]). This increase of complexity (in move generation) is also seen in the fields of (1) evaluation [16, 32], (2) search, and (3) parallelising and multi-core programming [20, 61].

A review of the version histories of known computer-chess engines clearly shows that the increase of complexity requires the use of stronger computer-chess architectures. We provide two examples. First, version 1.2.1 of Glaurung [58] was replaced by a clearly stronger version 2.1.[3] This stronger version implements a completely new computer-chess architecture with a new chess-program design. Second, Letouzey claimed to have developed a new computer-chess architecture in order to be able to improve his chess engine and to introduce an alternative approach to *Rotated Bitboards* [39].[4]

[2] For further information on the 0x88 approach refer to [42].

[3] Tord Romstad is the author of the open source computer-chess engine Glaurung.

[4] Fabien Letouzey is the author of the computer-chess engine Fruit which scored the 2^{nd} place at the 13th World Computer-Chess Championship, Reykjavik 2005.

These are only two examples, for which the research and development of a new computer-chess architecture can be justified. In general, we may state that (1) higher computing speed leads to a performance increase of the computer-chess engine without any trade-off, and that (2) the reduction of the CPU time ($\leq 50\%$) is approximately proportional to the measurable performance increase in ELO [59].

Obviously, straightforward and robust data structures of a computer-chess architecture reduce the test and debug phase of a computer-chess engine. The main advantage of such data structures lies in their simplicity and ease, and the possibility to implement specific and complex chess knowledge in a more compact and transparent way. Rajlich [45] defined chess knowledge very aptly as follows:[5]

> *"Chess knowledge wins games. If it does not, it is no knowledge."*

The computer-chess architectures, which are introduced and discussed in the following chapters, have been successfully tested. Apart from the countless private and public online computer-chess tournaments, the computer-chess architectures discussed in this thesis proved themselves by scoring the 2^{nd} place at the 26^{th} Open Dutch Computer-Chess Championship, Leiden (NL) 2006 and the 3^{rd} place at the 15^{th} World Computer-Chess Championship, Amsterdam (NL) 2007. The successes in the tournaments achieved by Loop Leiden 2006 and Loop Amsterdam 2007 are predominantly the result from the computer-chess architectures employed. The development of these computer-chess architectures lasted four years in total.

Without these technological developments further implementations of complex data structures ($\rightarrow$ parallelising and multi-core programming, chess knowledge, etc.) would hardly have been possible. In spite of that, only an excerpt from the most interesting and scientifically most valuable computer-chess architectures, technologies, and algorithms can be presented within the scope of this thesis. Further developments often differ only in minor details and are based on approaches similar to those introduced here.

1.2 Preliminary Considerations

The development of a computer-chess architecture is a rather laborious and extensive process. The main reason is the high complexity of the technologies, which are at the core of a computer-chess engine. Here, the main requirement is the harmonious interplay of these technologies. Thus, a computer-chess architecture can hardly be developed from scratch by a single person. As a result, technologies presented and developed in this thesis are also based on external developments and implementations. However, it is often not possible to trace back the initial sources and the original development of a particular technology. It goes without saying that we will credit all persons who are traceable, even by a posting only (see below).

[5] Vasik Rajlich is the author of the computer-chess engine Rybka. His computer-chess engine won the title of Computer-Chess World Champion twice, in Amsterdam 2007 and Beijing 2008.

In this thesis we predominantly refer to current discussions in computer-chess forums (→ *Winboard Forum* [2] and *Computer-Chess Club* [1]). Older literature and references, which indeed form a basis for the new development, are rarely used. This thesis also does not aim at the explicit consideration of known computer-chess architectures, such as Rotated Bitboards [11, 27] or the *0x88* representation [42]. Many a reference used in this thesis is not available in a scientifically elaborate form. This includes personal conversations with programmers [15, 39, 48] and the exchange of source codes as well as discussions via email. In this way the contents of this thesis can be regarded to be on a state-of-the-art level of the research and development in the field of the computer-chess architectures.

For reasons mentioned above, the number of the scientifically referred sources, such as books, book chapters, and journal publications is fewer than in an ordinary thesis. This thesis will make an attempt to refer as much as possible to researchers, who, owing to intensive development work, have not published their intellectual asset up to now.

1.3 Problem Statement and Research Questions

The introduction of Section 1.1 clearly stipulates which three kinds of requirements for the development of computer-chess architectures must be met.

1. **Unlimited implementation of chess knowledge.** The success of a computer-chess engine highly depends on the kind of implementation of complex data structures and chess knowledge.
2. **Higher computing speed.** The performance of the computer-chess architecture should be efficient. Here we note that a higher overall performance also results from a higher computing speed.
3. **Minimal overhead.** The data structures of the computer-chess architecture should be (1) as straightforward as possible and (2) as compact as possible during the implementation. It is important that the data of the computer-chess architecture are managed in a non-redundant way (at least, as much as possible) in order to minimise unnecessary overhead.

Based on these requirements, the following problem statement will lead us step by step through the development and analysis of new computer-chess architectures.

> **Problem statement:** *How can we develop new computer-chess architectures in such a way that computer-chess engines combine the requirements on knowledge expressiveness with a maximum of efficiency?*

The literature on computer chess is nowadays abundant, even in the subdomain of computer-chess architectures. So, all publications and research results in the latter field cannot be considered and analysed within the scope of this

thesis. The focus is therefore on the most outstanding and state-of-the-art issues. The different elements of a computer-chess engine (see Section 1.1) often cannot be separated clearly from each other due to their high complexity. For instance, the distribution of the search over several processors [20, 26] is based predominantly on the data structures of the computer-chess architecture, since information between split nodes [26] has to be exchanged and copied continuously. Many further developments in the field of a computer-chess engine are closely associated with the computer-chess architecture and are based on these basic data structures and algorithms.

The focus of this thesis is threefold: (1) on the development and analysis of a non-bitboard computer-chess architecture, (2) on the development and analysis of a computer-chess architecture based on magic multiplication and bitboards, and (3) on the development and analysis of a *static exchange evaluator* (SEE) with $\alpha\beta$-approach. The following three research questions will be answered in this thesis. They will be repeated in the chapters in which we address them. In these chapters we explain the notions used in full detail. Moreover, the research questions will be used to answer the problem statement.

> **Research question 1:** *To what extent can we develop non-bitboard computer-chess architectures, which are competitive in speed, simplicity, and ease of implementation?*

The first part of this thesis deals with the development and analysis of a complete computer-chess architecture. It is not based on only one data type, such as a 64-bit unsigned integer.[6] This computer-chess architecture is supposed to meet all important requirements. Findings gained here can simply be applied to other board games [7], such as Gothic Chess, 10×8 Capablanca Chess, Glinski's Hexagonal Chess, or computer Shogi [23] as there is no dependence between the dimension of a board and a specific integer data type.

> **Research question 2:** *To what extent is it possible to use hash functions and magic multiplications in order to examine bitboards in computer chess?*

In order to answer the second research question, a further computer-chess architecture must be developed. The implementation of this computer-chess architecture is based on a perfect mapping function and on 64-bit unsigned integers for the optimal use of the internal bandwidth [27].

> **Research question 3:** *How can we develop an $\alpha\beta$-approach in order to implement pruning conditions in the domain of static exchange evaluation?*

The development and analysis of an SEE is a challenging issue. The SEE is an important module of the computer-chess architecture for the qualitative and quantitative evaluation of moves and threatened squares. By means of the $\alpha\beta$-window, on which the static exchange evaluation is based, it is also possible to

[6] A 64-bit unsigned integer is the data type of the so-called *bitboard*.

implement pruning conditions. The benefit of these pruning conditions is the reduction of an iterative computation. The third research question deals with the theoretical elements and the implementation of different SEE algorithms.

1.4 Research Methodology

The research methodology is empirical. It consists of six phases. The first phase is characterised by collecting knowledge on methods and techniques. This is performed by reading, analysing, and validating to some extent the existing literature and even more by discussing several issues in internet computer-chess forums, such as the *Computer-Chess Club* [1] and the *Winboard Forum* [2]. The second phase is investigating the applicability of the methods and techniques of the first phase and attempting to estimate their relation to the methods and techniques used. The third phase is designing a new computer-chess architecture and analysing its possibility at a theoretical level.

In the fourth phase the implementation of the design takes place. Then in the fifth phase the implementation is tuned and tested by comparing it with the theoretical results. Finally, in the sixth phase the implementation is tested in practice, and an evaluation of the performance takes place. This phase is a real-life phase, since the computer-chess architecture is calibrated with its peers on the highest level, the world computer-chess championship.

1.5 Background of the Engine

The development of the computer-chess engine Loop started in 2004. The unlimited implementation of chess knowledge combined with high computing speed were the main goals of the new computer-chess engines. Apart from the implementation of state-of-the-art forward pruning techniques, such as *null move* [13, 14] and *late move reductions* [47], the development of new computer-chess architectures was the most important issue.

At the beginning, these new computer-chess architectures had to perform on 32-bit environments. The computer-chess engine Loop Leiden was the first chess engine to use detailed piece lists in the interplay with blocker loops. The main objective was to port this engine onto different state-of-the-art hardware environments without losses of performance. Furthermore, this new computer-chess architecture performed better than the widespread technology of Rotated Bitboards, even on 64-bit computer environments.

With the wide distribution of 64-bit computer environments, it was essential to develop further computer-chess architectures based on 64-bit unsigned integers (bitboards). These special computer-chess architectures were developed with the main objective, which was the maximum of performance on 64-bit environments. They are as straightforward as possible and rather compact during their implementation. But, as opposed to the platform-independent computer-chess architecture implemented in Loop Leiden, it is not possible to use these 64-bit technologies within the environment of 32-bit hardware and software systems without a loss of computing speed. The use of hash functions and magic

multiplications led to a quite efficient examination of bitboards in order to dispense with the rotated-bitboard approach. This resulted in the quite successful computer-chess engine Loop Amsterdam.

Additionally, the development of an algorithm for the evaluation of moves was essential in order to gain a better move ordering for further reduction of the branching factor. This requirement led to a quite interesting tool of the state-of-the-art computer-chess architecture: the static exchange evaluator. The results of this algorithm are quite precise. The application of the SEE algorithm is straightforward and the field of application is enormous (see Section 4.5).

All these techniques and algorithms are designed for the operation within multi-core computer-chess engines. Loop Leiden is a dual-core computer-chess engine. Loop Amsterdam is a quad-core computer-chess engine. Both computer-chess engines are based on an advanced shared hash table approach [61].

1.6 Thesis Overview

This thesis contains five chapters. Chapter 1 is a brief and general introduction of computer-chess architectures. Thereafter the problem statement and three research questions are formulated. In the course of the thesis every research question is discussed and answered in a separate chapter.

Chapter 2 answers the first research question. The development of a non-bitboard computer-chess architecture is carried out on an R&D basis of the computer-chess engine Loop Leiden 2006 and Loop Express.[7] At the beginning of this chapter, profile information of the computer-chess engine Loop will be discussed. The most important requirements and objectives of the computer-chess engine can be defined by a more precise evaluation of this profile analysis.

Chapter 3 answers the second research question. The development of the computer-chess architecture, based on bitboards and perfect hash functions requires the use of a particularly efficient mapping function. This chapter focuses on the theory and the development of the magic mapping function. With this mapping function a bit scan and hash algorithms for sliding pieces are developed. The two algorithms form the basis of the computer-chess architecture, where the information is stored in 64-bit unsigned integers. Finally, magic multipliers for a magic mapping function will be generated and examined by a suitable trial-and-error algorithm and a brute-force approach.

Chapter 4 answers the third research question. After the introduction of the SEE algorithm, recursive and iterative implementations will be examined. An $\alpha\beta$-window to control the evaluation is introduced on the basis of an iterative approach. Due to the new structure of the algorithm, the implementation of multiple pruning conditions is possible for the first time. Then, the efficiency of combined pruning conditions is analysed. Additionally, some typical applications of the SEE in the field of a computer-chess engine are introduced. An SEE is an important component of the state-of-the-art computer-chess architecture due to its structure and its deterministic computation.

[7] Loop Express is the computer-chess engine that was developed for Nintendo Wii Chess.

Chapter 5 contains the research conclusions and recommendations for future research.

This thesis contains three appendices. In Appendix A additional source codes, which are dealt with in the thesis, are listed. In Appendix B the magic multiplier sets for Bishops and Rooks according to the experiments in Section 3.8 are given. Appendix C provides two complete collections of computer-chess games played by LOOP LEIDEN at the 26^{th} Open Dutch Computer-Chess Championship, Leiden (NL) 2006 and LOOP AMSTERDAM at the 15^{th} World Computer-Chess Championship, Amsterdam (NL) 2007.

Chapter 2

Non-Bitboard Architectures

The main objective of Chapter 2 is to answer the first research question by a scientific R&D approach of the computer-chess engines Loop Leiden 2006 and Loop Express. Below we repeat the first research question.

> **Research question 1:** *To what extent can we develop non-bitboard computer-chess architectures, which are competitive in speed, simplicity, and ease of implementation?*

The development of the computer-chess architecture is mainly based on the experiences gained during the development of a computer-chess engine. A definition of the requirements and objectives is thus hardly possible without sufficient a priori experience (see Section 1.1). For this reason, the development of a computer-chess architecture in practice is developed mainly through adaptations and extensions. This mostly happens in cooperation with the development of the search and the evaluation components. The computer-chess architecture is adjusted to the remaining program structures in many steps which often overlap. The results of this incremental development are mostly complicated, redundant, and inconsistent data structures and algorithms. A computer-chess architecture, which is developed in such a way, is only rarely competitive in speed, simplicity, and ease of implementation.

The developer of the computer-Shogi engine Spear, Grimbergen [23, page 25] formulated this issue in the article "Using Bitboards for Move Generation in Shogi" quite precisely.[1]

> *"Quite a complicated piece of code and an area of the program that I rather do not touch even if there are some obvious optimizations that can be implemented."*

Although based on the experiences gained during the development of the Loop computer-chess β-engines 2005-2006, the 32-bit computer-chess architecture for Loop Leiden was written from scratch. One of the objectives of this redesigned engine was a strong and homogeneous data structure, that can also be used in

[1] Shogi's search algorithms are similar to the game of chess. However, the game has different pieces and rules, and is played on a 9×9 board.

the environment of a multi-core computer-chess engine (→ multiple threads or multiple processes). Therefore, we imposed three criteria on the used data structures.

1. **Competitiveness in speed.** A high computing performance is to be gained by efficient computations and minimal management overhead of the data. Besides the efficient move generation, attack detection, and mobility evaluation, the incremental expenditure of move simulations (→ do move, undo move) is to be minimised. Data structures with small redundancy are needed for that. Unlike in the classical approach of Rotated Bitboards [11, 27] the data about the chessboard are not managed in a redundant way.[2]

2. **Simplicity.** The basic data structures should be as straightforward as possible. All elementary storage access, on which the computer-chess architecture is based, is almost exclusively implemented by one-dimensional linear vectors. The most important computations, such as move generation, attack detection, and mobility evaluation, are realised by means of arithmetic operations (→ plus, minus, increment, and decrement).

3. **Ease of implementation.** The development of algorithms is based on fixed rules. For every kind of attack detection (→ direct attack, indirect attack, etc.) a specific *blocker loop* is used (see Section 2.5). All data which contain information about the chessboard (→ squares) are categorized unambiguously according to pieces and colours (→ white Pawn, ..., black King). The implementation is not based on specific integer data types (→ 64-bit unsigned integer), such as Rotated Bitboards or Magic Bitboards (see Chapter 3) [30], and is thus used more universally.[3] No relation between the size of the board and a specific integer data type exists.

Only when (1) the internal computer chessboard (see Section 2.2), (2) the detailed piece lists (see Section 2.3), and (3) the two blocker loops (see Section 2.5) are in harmonious interplay, a high-performance framework can be developed. This framework is not explicitly based on the use of bitboards and is thus implemented more flexibly as the size of the board with $n \times m > 64$ squares can be chosen almost arbitrarily. For this reason, the developed technologies can also be used in other chess variants [7] as for example Gothic Chess, 10×8 Capablanca Chess, Glinski's Hexagonal Chess, and in board games such as computer Shogi.

The chapter is organised as follows. In Section 2.1 an overview of known projects is given, in which the computer-chess architecture of Loop Leiden, to be introduced in this chapter, is implemented. In Section 2.2 the computer chessboard design for a non-bitboard computer-chess architecture is developed and

[2] Bitboards are also called bitsets or bitmaps. This data structure is used to represent the board in a piece-centric manner inside a computer-chess engine. Each bit represents a game position on the internal chessboard [35]. Rotated Bitboards make certain operations more efficient by rotating the bitboard positions by 90 degrees, 45 degrees, and 315 degrees.

[3] Magic Bitboards is a new and fast alternative to Rotated Bitboards. This technology uses a perfect hashing algorithm to index an attack bitboard look-up table. For more information see: `http://chessprogramming.wikispaces.com/Magic+Bitboards`

discussed. For this reason, the chessboard and the management of chessboard-related information play a central role. Based on the investigations in Section 2.3, the administration of information in piece lists is discussed. In Section 2.4 experiments with pieces in lists are carried out. In Section 2.5 two blocker loops are described, on which all move generations, attack detections, and mobility evaluations of sliding pieces in the computer-chess engine LOOP LEIDEN are based. Section 2.6 concludes with the experiment with loop iterations. A summary of the results backed up by an empirical experiment is presented in Section 2.7. Finally, the first research question is answered.

2.1 Implementation in Two External Projects

The non-bitboard computer-chess architecture, which will be introduced and examined in this chapter, has been used in two external projects since 2006. Below, the two projects and the implementations of the non-bitboard computer-chess architecture will be presented briefly.

2.1.1 The Chess Machine Hydra

The non-bitboard computer-chess architecture, on which LOOP LEIDEN is based, has been also implemented since 2006 in the chess machine HYDRA developed by Donninger *et al.*[4] Due to the simplicity and the ease of implementation, this computer-chess architecture could be implemented into the existing HYDRA project [17, 18, 19] in a short time. The chess machine could also execute software-based computations, such as move generation more efficiently. According to a statement by Donninger, the efficiency increase of this computer-chess architecture was slightly smaller in the chess machine HYDRA than in a pure computer-chess engine. Particularly, this is due to the parallel processing of the search tree by *field programmable arrays* (FPGAs) [5], whereby software-based implementations and optimisations are emphasized less.

2.1.2 Nintendo Wii Chess

The second implementation concerns the Nintendo Wii Project. The implementation of the non-bitboard computer-chess architecture from LOOP LEIDEN into the Nintendo project Wii Chess, 2007, proceeded without complications. The lower performance and the 32-bit bandwidth of Nintendo's main processor (codename "Broadway") were sufficient to implement the same computer-chess architecture, such as LOOP LEIDEN, without any restrictions.[5] Due to the smaller main memory, available for the chess engine on the Nintendo system, almost only memory-efficient one-dimensional look-up tables can be used for the computer-chess architecture. So, LOOP EXPRESS has smaller transposition

[4]For more information about the HYDRA project see: `http://www.hydrachess.com/`

[5]The main processor is based on the "PowerPC 750 CL"-architecture and was developed in a collaboration between IBM and Nintendo. For more information see: `http://www.mynintendo.de/wii/`

tables and smaller pawn hash tables (see Subsection 2.2.4). The implementation of the complete computer-chess engine Loop Express into the Wii Chess project lasted only a few weeks. After an extensive test phase, no problems were found within the computer-chess architecture.

2.2 Computer Chessboard Design

In this section the internal chessboard is developed. It is the basis for the non-bitboard computer-chess architecture. The computer chessboard is the central data structure of the computer-chess architecture as almost all computations (e.g., move generation and attack detection) have access to this data structure or their incremental manipulation (do move, undo move). Hence, the internal computer chessboard will be scrutinized in the further course of this chapter.

The most important issues of the internal computer chessboard design are: (1) the minimum size of the horizontal board border and (2) the management of *distances* and *increments* via surjective mapping. Surjective mapping means in this context the mapping of a pair of squares (from-square, to-square) into a linear vector. The detection of attacks, the evaluation of mobility, and the generation of moves of sliding pieces (→ captures, non-captures, check threats, and check evasions) are mainly based on the design of the internal computer chessboard. The more skillful the arrangement of the squares and the board border is, the more efficient and straightforward elementary computations can be executed.

A Profile Analysis

A profile analysis is carried out by means of Loop Leiden to obtain an overview of the time consumption of the most important functions of a computer-chess architecture. Below, we summarise (1) move generators, (2) attack detectors, and (3) functions for move management in a superordinate function. Thus, the time consumption of elementary functions of the computer-chess architecture can be measured sufficiently by this profile analysis.

In total, three different measurements were carried out at different game stages. All three measurements deliver rather similar results. For this reason, only the results of the first measurement are listed in Table 2.1. The position of the sample board was adopted from the Queen's Gambit (ECO = D53) in round 2 of the game Loop vs. Shredder at the 15^{th} World Computer-Chess Championship, Amsterdam (NL) 2007.[6] The position for the profile analysis was reached after 7 c×d5 e×d5 (see Appendix C.2).

The results of this measurement (see in Table 2.1) are subdivided into three main columns. In the first main column the function name and the number of the function calls are listed. The second main column is the sorting criterion of the profile analysis. The used CPU time in milliseconds (ms) of a function without substack is sorted in a descending order. In this case, the net CPU time of the single functions was measured. In contrast, the results of the measurement

[6] ECO: Encyclopedia of Chess Openings.

with substack are entered in the third main column. The gross CPU times contain the CPU times consumed in reality for all non-search functions; here all encapsulated function calls are considered. For instance, the functions for the position evaluation (→ `evaluate`) encapsulate further evaluation functions (material evaluation, pawn evaluation, king-safety evaluation, etc.).

A recursive search function consumes almost 100% of the entire CPU time as the complete computation is encapsulated in its substack. The search functions `search_w` (White's point of view) and `search_b` (Black's point of view) are the implementation of the $\alpha\beta$-*principal variation search* (PVS). The search functions `search_q_w` (White's point of view) and `search_q_b` (Black's point of view) are the implementation of the $\alpha\beta$-quiescence search [4].

The superordinate function `cca` (computer-chess architecture) encapsulates all functions of the computer-chess architecture. The actual CPU time consumed by the computer-chess architecture is the tare CPU time. The net CPU time indicates only the overhead of 4.6% for the explicit encapsulation in this experiment. Accordingly, the CPU time consumed in reality is approximately $51.7\% - 4.6\% = 47.1\%$. This result was confirmed by 46.9% and 42.1% in two further measurements for the middlegame and the endgame.

Profile Analysis: The Computer-Chess Architecture

		without substack (net)		with substack (gross)	
func name	func calls	time (ms)	rel (%)	time (ms)	rel (%)
`evaluate`	456,012	1,352	10.8	1,881	15.0
`search_w`	225,312	636	5.1	12,119	96.9
`cca`	3,401,726	575	4.6	6,463	51.7
`search_b`	184,500	493	3.9	12,155	97.2
`search_q_w`	244,021	476	3.8	4,829	38.6
`search_q_b`	210,814	425	3.4	4,923	39.4

Table 2.1: Profile analysis of the encapsulated computer-chess architecture.

The measured values confirm that the computer-chess architecture within the scope of a state-of-the-art computer-chess engine together with the evaluation functions consume most of the CPU time. Also large parts of the evaluation are based on the computer-chess architecture (recognition of threats, evaluation of mobility, etc.). Since the evaluation has a non-deterministic origin (see Section 1.1), this is to be considered separately.

This section contains four subsections. In Subsection 2.2.1 we will develop the computer chessboard representation. In Subsection 2.2.2 distances and increments within a computer-chess architecture will be defined. In Subsections 2.2.3 and 2.2.4 minimal board borders will be developed in order to introduce fast and memory-efficient one-dimensional look-up tables.

2.2.1 Computer Chessboard Representation

Below we discuss two computer-chessboard representations. First we review a chessboard representation with a border. Due to the extension of the internal board to an upper and lower as well as left and right board border, the border squares of the intrinsic 8×8 board do not have to be examined separately. If a piece exceeds the board border, this must be recognised quickly. In the 0x88-board computer-chess architecture this is gained by masking of the destination square with the hexadecimal number $88_{hex} = 136$. If the destination square belongs to the internal board ($square\ AND\ 88_{hex} = 0$), it must be checked additionally whether the destination square is empty.

This necessary condition ($square\ AND\ 88_{hex} = 0$) will be redundant, if the border of the internal board is examined as a separate *border piece*. As soon as a piece leaves the internal 8×8 board, it automatically comes to a collision with a border piece. If a sliding piece collides with a border piece, the sliding direction is then to be changed.

Second, a computer chessboard without a board border is only used in bitboard-based computer-chess architectures as the number of squares corresponds with the number of bits of 64-bit unsigned integers on the chessboard, the so-called *bitboard*. In some cases there are also non-bitboard developments which work without an explicit board border [60]. In most known implementations look-up tables are used when generating moves which are retrieved recursively. Especially with sliding pieces, the end of a sliding direction must be recognised easily so that the change from a sliding direction to the next one is executed with less computing time. These two-dimensional look-up tables are indeed elegant, however not particularly efficient.

In the computer-chess engine GNU-Chess and in older versions of Glaurung two-dimensional look-up tables are used for the generation of moves and the detection of attacks in relation to a borderless board. The source square of a piece and the destination square, which was retrieved last, are used to access the look-up table.

```
destination_sq = look_up_table(source_sq, destination_sq);
```

Only for starting this recursive mechanism the source square is also to be set onto the place of the destination square. An alternative would be the introduction of a one-dimensional look-up table to access the very first destination square, in order to start the recursive mechanism with the two-dimensional look-up table. If a sliding direction is blocked because of a collision, a further two-dimensional look-up table will be accessed, which, according to the same recursive mechanism, contains the first square of a new sliding direction. After first experiments with the Loop project in 2005 it turned out that this approach was not competitive in speed.

2.2.2 Distances and Increments

The relation between two squares is quite often evaluated during move generation, attack detection, and other elementary board operations. To let it happen in such a way that this evaluation can be carried out as fast as possible, the

searched information is stored in one-dimensional and two-dimensional look-up tables. The organisation of the information in one-dimensional look-up tables is only then possible when (1) the horizontal board border is sufficiently large and (2) only a relative relation between two squares is required. As soon as the relation between two squares has an absolute size (→ bijective mapping), no one-dimensional look-up tables can be used anymore. However, the one-dimensional look-up tables require considerably less memory and can also be used more efficiently since the address computation for their access is simpler.

For example, a look-up table for the management of distances or increments between two squares (→ file distance, rank distance, or square distance) is most straightforwardly to be realised by a one-dimensional look-up table. Concrete implementations will be presented in Subsection 2.2.4, after the dimensioning of the horizontal board borders is defined in Subsection 2.2.3. Below we define the basics for the dimensioning of the board border.

Distances

The information about distances is mainly used within the scope of evaluation. File distances are mostly used for the evaluation of attack patterns such as king-attack evaluation. The file distance describes the absolute distance between two files (see Equation 2.1) and is no metric with regard to sq_1 and sq_2 as $d_{file}(sq_1, sq_2) = 0 \Leftrightarrow sq_1 = sq_2$ does not apply [9, pages 602f.]. The rank distance, similarly to the file distance, describes the absolute distance between two ranks according to Equation 2.2. Combined file-rank distances, which are the sum of file distance and rank distance as in Equation 2.3, are not implemented in the computer-chess engine Loop Leiden. Access to look-up tables for square distances is especially relevant in the endgames to evaluate passed Pawns and the King-Pawn-King (KPK) endgames [4]. The square distance, as defined in Equation 2.4, is the maximum of file distance and rank distance, and is no metric, either.

Definition of Distances and their Bounds

$$0 \leq d_{file}(sq_1, sq_2) = |file(sq_1) - file(sq_2)| \leq 7 \quad (2.1)$$

$$0 \leq d_{rank}(sq_1, sq_2) = |rank(sq_1) - rank(sq_2)| \leq 7 \quad (2.2)$$

$$0 \leq d_{file\_rank}(sq_1, sq_2) = d_{file}(sq_1, sq_2) + d_{rank}(sq_1, sq_2) \leq 14 \quad (2.3)$$

$$0 \leq d_{square}(sq_1, sq_2) = max(d_{file}(sq_1, sq_2), d_{rank}(sq_1, sq_2)) \leq 7 \quad (2.4)$$

Increments

Increments are signed integers, which determine the increment between two squares, so that a piece can move from a source square to a destination square. This information is important especially in relation to the recognition of attacks within a non-bitboard computer-chess architecture. Increments only exist between a source square and a destination square, which can be reached by

one pseudo-legal move of any piece (→ Knight, King or sliding piece).[7] For the computation of an increment for sliding pieces the difference between two squares on the internal chessboard is divided by the square distance d_{square} according to Equation 2.5. For the computation of an increment for the Knight the difference between two squares on the internal chessboard is used according to Equation 2.6. In Equations 2.5 and 2.6 $sq_2 = destination\_square$ and $sq_1 = source\_square$.

Definition of Increments

$$sliding\_increment(sq_2, sq_1) = \frac{sq_2 - sq_1}{d_{square}(sq_2, sq_1)} \quad (2.5)$$

$$knight\_increment(sq_2, sq_1) = sq_2 - sq_1 \quad (2.6)$$

This relation between a source square and a destination square is calculated once and stored in one-dimensional or two-dimensional look-up tables. After the initialisation and during calculations the required information is retrieved again from the look-up tables. The information about increments is needed for the generation of check evasions and checking moves, for the recognition of attacks and the static exchange evaluation (see Chapter 4). According to the internal chessboard representation in Figure 2.1 square `h6` is represented as entry 115 and square `f5` as entry 98. This means that the increment corresponding to the move ♘f5–h6 would be 17.

2.2.3 Minimal Board Borders

For an adequate computer-chess architecture the determination of the size of the board border is important so that distances and increments can be managed as non-redundantly as possible. The uniqueness of a mapping can be achieved only by a minimum size of the horizontal board border. The sum of the right and left board borders must be greater than the difference between a square on the `a`-file and a square on the `h`-file on the same rank (see Equation 2.7).

Minimal Horizontal Board Border

$$border_{left} + border_{right} \geq d_{square}(sq_{a1}, sq_{h1}) = 7 \quad (2.7)$$

The difference between a square on `a`-file and a square on `h`-file within an `a1h8`-orientated board is seven squares. The right and left board borders must have a size of at least seven squares so that one-dimensional vectors can be addressed by a unique mapping. The vertical board border does not have any influence on the mapping function. In spite of that, the vertical board border (upper and lower board borders) should be minimal in order to obtain an internal board, which is as small as possible. The upper and lower board borders are at least

[7] A pseudo-legal move is a move which can be done, if no friendly piece is on the destination square and the sliding direction to the destination square is unoccupied.

2 squares large so that the Knight cannot move over the board border of the internal chessboard. After the determination of the board border, the internal chessboard has a minimum dimension of 15×12 squares which is shown in Figure 2.1.

165	166	167	168	169	170	171	172	173	174	175	176	177	178	179
150							...							164
135			$A8$	$B8$	$C8$	$D8$	$E8$	$F8$	$G8$	$H8$				149
120			$A7$	$B7$	$C7$	$D7$	$E7$	$F7$	$G7$	$H7$				134
105			$A6$	$B6$	$C6$	$D6$	$E6$	$F7$	$G6$	$H6$				119
90	...		$A5$	$B5$	$C5$	$D5$	$E5$	$F5$	$G5$	$H5$		...		104
75			$A4$	$B4$	$C4$	$D4$	$E4$	$F4$	$G4$	$H4$				89
60			$A3$	$B3$	$C3$	$D3$	$E3$	$F3$	$G3$	$H3$				74
45			$A2$	$B2$	$C2$	$D2$	$E2$	$F2$	$G2$	$H2$				59
30			$A1$	$B1$	$C1$	$D1$	$E1$	$F1$	$G1$	$H1$				44
15							...							29
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Figure 2.1: Internal chessboard with minimal borders.

2.2.4 One-dimensional Look-up Tables

In order to retrieve information about distances and increments from one-dimensional look-up tables a fast mapping function is required. This mapping function carries out a surjective mapping of two squares into the memory area of the corresponding look-up table. The mapping is calculated by the sum of a constant offset ($\rightarrow const\_offset$) combined with the difference of a source square ($\rightarrow sq_1$) and a destination square ($\rightarrow sq_2$). The constant offset must only guarantee that the calculated vector index is not negative ($\rightarrow index \geq 0$). $const\_offset + sq_{a1} - sq_{h8} \geq 0$ results from the non-negativity of the index. According to the board matrix in Figure 2.1 ($\rightarrow sq_{a1} = 33 \wedge sq_{h8} = 145$) we get $const\_offset \geq sq_{h8} - sq_{a1} = 112$. Access to the look-up table according to the calculated index is summarised in Equation 2.8. In spite of the asymmetric board border, it does not matter for the calculation of the constant offset, whether the left (see Figure 2.1) or the right border is three squares large.

Because of the seven-squares large board border, it is not possible to get from an edge square of the `h`-file over the board border to the next edge square of the `a`-file as $|sq_1 - sq_2| \geq 8$ applies to the minimum difference between these edge squares. Therefore, we deal with an surjective mapping. The smallest possible index can be $index = const\_offset + sq_{a1} - sq_{h8} = 0$ as in Equation 2.8. The largest possible index can be $index = const\_offset + sq_{h8} - sq_{a1} = 224$ according to Equation 2.8. It may be concluded that an surjective one-dimensional look-up table for a board as shown in Figure 2.1 has the memory capacity of $0 \leq index \leq 224$ and thus consists of 225 table entries.

Since only the relative relation between two squares to each other can be considered here, no absolute squares as a result value can be contained in such one-dimensional look-up tables. Moreover, absolute results must be managed in two-dimensional look-up tables. Almost all distances and increments can be managed by these one-dimensional surjective look-up tables within the scope of the computer-chess architecture.

The Unique Mapping Function for Vector Access

$$\begin{aligned} 0 \leq index &:= const\_offset + sq_1 - sq_2 \leq 224 \\ \Rightarrow result &:= vector(index) \end{aligned} \tag{2.8}$$

Four Examples

The look-up table with $225 = 15^2$ table entries for the internal chessboard of Figure 2.1 has a midpoint in a quadratic arrangement according to the examples 1 to 4 of Figure 2.2 in which all distances and increments are zero. This applies only then, when $sq_1 = sq_2$ ($\rightarrow source\_square = destination\_square$). Such a look-up table for retrieving information about the file distance or the square distance between two squares within a 12×15 chessboard with minimal board borders would look like as shown in example 1 and 2 of Figure 2.2. Whereas the matrices for distances are axis-symmetrical (see example 1 of Figure 2.2) or point-symmetrical (see example 2 of Figure 2.2), the matrices of increment look-up tables (see examples 3 and 4 of Figure 2.2) have an orientation without any symmetry. Source square and destination square must not be swapped therefore.

Look-up Table Access in Practice

The source-code listing of Figure 2.3 shows a straightforward algorithm for the verification of the legality of a move from the view of White. This algorithm is used in practice. It is implemented (apart from the processing of two special cases) in the computer-chess architecture of Loop Leiden and Loop Express. A king move and en passant move must be examined explicitly and are not implemented here. This algorithm requires, however, that the current position is legal and the King of the side, whose turn it is, is not in check. Consequently, this algorithm is only written for the incremental verification of the legality of moves. Precisely as for the incremental detection of check threats, one-dimensional increment look-up tables are used for this algorithm in order to examine sliding directions between the King and the moving piece. Thus, the algorithm in the source-code listing of Figure 2.3 is best suitable in order to demonstrate the use of look-up tables within the scope of the computer-chess architecture. The sample algorithm works in three steps and checks whether a pinned piece is trying to move out of its pinning line.

Step 1

In step 1 it is checked, whether a relation between the King square

```
king_square_w
```

and the source square

```
from=move_from(move)
```

Example 1: File Distance Matrix

$$\begin{bmatrix} 7 & 6 & 5 & \dots & 0 & \dots & 5 & 6 & 7 \\ 7 & 6 & 5 & \dots & 0 & \dots & 5 & 6 & 7 \\ 7 & 6 & 5 & \dots & 0 & \dots & 5 & 6 & 7 \\ \vdots & & \vdots & & \vdots & & \vdots & & \\ 7 & 6 & 5 & \dots & 0 & \dots & 5 & 6 & 7 \\ \vdots & & \vdots & & \vdots & & \vdots & & \\ 7 & 6 & 5 & \dots & 0 & \dots & 5 & 6 & 7 \\ 7 & 6 & 5 & \dots & 0 & \dots & 5 & 6 & 7 \\ 7 & 6 & 5 & \dots & 0 & \dots & 5 & 6 & 7 \end{bmatrix}$$

Example 2: Square Distance Matrix

$$\begin{bmatrix} 7 & 7 & 7 & \dots & 7 & \dots & 7 & 7 & 7 \\ 7 & 6 & 6 & \dots & 6 & \dots & 6 & 6 & 7 \\ 7 & 6 & 5 & \dots & 5 & \dots & 5 & 6 & 7 \\ \vdots & & \vdots & & \vdots & & \vdots & & \\ 7 & 6 & 5 & \dots & 0 & \dots & 5 & 6 & 7 \\ \vdots & & \vdots & & \vdots & & \vdots & & \\ 7 & 6 & 5 & \dots & 5 & \dots & 5 & 6 & 7 \\ 7 & 6 & 6 & \dots & 6 & \dots & 6 & 6 & 7 \\ 7 & 7 & 7 & \dots & 7 & \dots & 7 & 7 & 7 \end{bmatrix}$$

Example 3: Queen Increment Matrix

$$\begin{bmatrix} -14 & 0 & 0 & \dots & -15 & \dots & 0 & 0 & -16 \\ 0 & -14 & 0 & \dots & -15 & \dots & 0 & -16 & 0 \\ 0 & 0 & -14 & \dots & -15 & \dots & -16 & 0 & 0 \\ & \vdots & & & \vdots & & & \vdots & \\ 1 & 1 & 1 & \dots & 0 & \dots & -1 & -1 & -1 \\ & \vdots & & & \vdots & & & \vdots & \\ 0 & 0 & 16 & \dots & 15 & \dots & 14 & 0 & 0 \\ 0 & 16 & 0 & \dots & 15 & \dots & 0 & 14 & 0 \\ 16 & 0 & 0 & \dots & 15 & \dots & 0 & 0 & 14 \end{bmatrix}$$

Example 4: Knight Increment Matrix

$$\begin{bmatrix} 0 & \dots & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\ \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\ 0 & \dots & 0 & -29 & 0 & -31 & 0 & \dots & 0 \\ 0 & & -13 & 0 & 0 & 0 & -17 & & 0 \\ 0 & & 0 & 0 & 0 & 0 & 0 & & 0 \\ 0 & & 17 & 0 & 0 & 0 & 13 & & 0 \\ 0 & \dots & 0 & 31 & 0 & 29 & 0 & \dots & 0 \\ \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\ 0 & \dots & 0 & 0 & 0 & 0 & 0 & \dots & 0 \end{bmatrix}$$

Figure 2.2: Distance and increment matrices for surjective one-dimensional look-up tables.

exists at all. Only if a relation exists and the destination square

```
to=move_to(move)
```

is not on the same line, the opponent with a sliding piece can attack the friendly King.

Step 2

In step 2 it is to be checked, whether the line between the king square and the source square of the move is not occupied.

Step 3

Subsequently, it is merely checked in step 3, whether in the extension of this line an opposing

```
piece_is_black(piece(next))
```

sliding attacker

```
pseudo_attack(piece(next),next, king_square_w)
```

is hidden.

Look-up Table Access in Practice

```
bool move_is_legal_w(int move) {
  int from;
  int to;
  int inc;
  int next;
  // analyse king move and en passant move at first
  if (king_move(move) || en_passant_move(move)) {
    // insert special code for verification of legality here
  }
  // extract the move information
  from=move_from(move);
  to=move_to(move);
  inc=queen_increment(king_square_w, from);
  // step 1
  if (inc == 0) return true;
  if (inc == queen_increment(king_square_w, to)) return true;
  // step 2
  for (next=king_square_w+inc; piece(next) == NONE; next+=inc);
  if (next != from) return true;
  // step 3
  for (next+=inc; piece(next) == NONE; next+=inc);
  if (piece_is_black(piece(next)) == false) return true;
  if (pseudo_attack(piece(next),next, king_square_w)) return false;
  return true;
}
```

Figure 2.3: Source-code listing for legality verification of moves.

2.3 Managing Information with Piece Lists

In the beginning of this chapter the requirements (1) competitiveness in speed, (2) simplicity, and (3) ease of implementation were imposed on data structures of the computer-chess architecture. In this section we will develop and present data structures, which fulfil these requirements in theory and in particular in practice.

The main issues are (1) the chess pieces and (2) their management as well as representation. Subsequently, the technologies used in the computer-chess engine Loop Leiden and Loop Express are introduced. These technologies and their interplay with the internal computer-chessboard design from Section 2.2 will build the framework of a strong and platform-independent computer-chess architecture.

This section contains six subsections. In Subsection 2.3.1 pieces and piece flags are discussed. In Subsection 2.3.2 a variety of piece lists will be presented. Thereafter in Subsection 2.3.3, we will split up piece lists for light and dark squares. Scanning pieces will be the issue of Subsection 2.3.4. In Subsection 2.3.5 the incremental piece-list update while doing and undoing moves will be developed. Finally, in Subsection 2.3.6 we will examine a difficult phenomenon, while scanning pieces.

2.3.1 Pieces and Piece Flags

The pieces *Pawn*, *Knight*, *Bishop*, *Rook*, *Queen*, *King* and their colours *White* or *Black* are to be marked by unique flag disjunctions. In this way, negative integers for piece representation can be avoided. Moreover, every piece type can be recognised independently of its piece colour. With the help of flags it can also be marked precisely whether a piece is a sliding piece. The flags are arranged in ascending order depending on the material piece values. Indeed, it is more efficient to write a specific source code to deal with every piece type. The disjunction of flags, however, enables the use of a more general and a more compact source code. Below we discuss both issues: (1) basic flags and (2) flag disjunctions.

Basic Flags

Whereas Bishops slide in `a1h8`-direction and `h1a8`-direction, and Rooks slide in `a1h1`-direction and `a1a8`-direction, the Queen is a combination of the sliding as a Bishop and the sliding as a Rook.[8] The non-sliding characteristics of the Knight are unique. As only piece it can jump over other pieces. The properties of the Pawn of every player are also unique and are marked by specific flags. For identification of the colour and the board border (see Section 2.2) three further flags are required.

All following basic flags and flag disjunctions are indicated in the hexadecimal

[8] In this thesis the absolute information about the direction (a1h1, a1a8, a1h8, h1a8) is preferred to the relative information about the direction (horizontal, vertical, diagonal, antidiagonal) as the former is independent of the perspective on the board.

system. An empty square is characterised by $flag\_none = 00$. The pieces are designated in ascending order as following: $flag\_pawn = 01$, $flag\_knight = 02$, $flag\_bishop = 04$, $flag\_rook = 08$, and $flag\_king = 10$. The piece colour is characterised by $flag\_white = 20$ and $flag\_black = 40$. Finally, the board border is to be unambiguously indicated by $flag\_border = 80$.

Flag Disjunctions

With the help of a disjunction of basic flags we obtain unique flags for all white and black pieces. In addition, the order of the material value according to Kaufman [32] is considered. Regardless of the respective colour flag, the piece type can be unmasked by conjunction. Subsequently, all flag disjunctions for white pieces are defined as follows.

$$
\begin{aligned}
white\_pawn &= flag\_white \vee flag\_pawn = 20 \vee 01 = 21 \\
white\_knight &= flag\_white \vee flag\_knight = 20 \vee 02 = 22 \\
white\_bishop &= flag\_white \vee flag\_bishop = 20 \vee 04 = 24 \\
white\_rook &= flag\_white \vee flag\_rook = 20 \vee 08 = 28 \\
white\_queen &= flag\_white \vee flag\_bishop \vee flag\_rook = 20 \vee 04 \vee 08 = 2C \\
white\_king &= flag\_white \vee flag\_king = 20 \vee 10 = 30
\end{aligned}
$$

Piece flags for black pieces consist of these basic flags, too. The arrangement of pieces in ascending order has an advantage that pieces can be already compared in their material value rather simply. Considering these criteria, the pieces and the board border cannot be defined more compactly by flag disjunctions.

2.3.2 Sequential and Recursive Piece Lists

The use of piece lists in non-bitboard computer-chess architectures is the only possibility to access all pieces in arranged order in a time-saving way. Without the piece lists the complete board would have to be scanned for pieces.

> *"We certainly don't want to have to loop over all of the 64 possible squares."*
> *(Robert Hyatt [25])*

A separation of pieces and Pawns is just as obvious as the separation between white and black piece lists. The Pawns should be processed in the evaluation functions and move generators, separately from the pieces. The more detailed the piece lists are organised, the more specifically the qualities of pieces and Pawns can be examined.

With the detailed splitting of the piece lists into piece types, piece-specific information can be managed implicitly for every piece type and the Pawns, such as piece count or piece colour. The possibly detailed splitting of the piece lists is the most elegant solution in order to split the data and to organise the implicit management of the data in a non-redundant way. We note that this results in the fact that the computations, which retrieve and manipulate the data from

the piece lists, are better separated from each other with regard to the contents and clarity.

The constants and the piece lists are defined in the source-code listing of Figure 2.4. In the computer-chess architecture of Loop Leiden even more detailed piece lists are used (see Subsection 2.3.3). A further interesting possibility of the organised management of pieces and Pawns on the chessboard is to be realised by a doubly-linked list. In addition, the mode of operation is recursive: after the first piece has been retrieved from the list, the retrieved square becomes the index of the next piece in the list. By this method, a recursive forward linkage of pieces within the doubly-linked list is achieved. The reverse linkage occurs explicitly and only serves for managing the doubly-linked list while removing and adding pieces into the doubly-linked list.

Definitions of Piece Constants and Piece Lists

```
// definition of maximum possible piece constants
// in a legal chess game considering
// promotions and underpromotions
const int MAX_PAWNS=8;
const int MAX_KNIGHTS=2+8;
const int MAX_BISHOPS=2+8;
const int MAX_ROOKS=2+8;
const int MAX_QUEENS=1+8;
// definition of detailed piece lists
int pawn_list[MAX_PAWNS];
int knight_list[MAX_KNIGHTS];
int bishop_list[MAX_BISHOPS];
int rook_list[MAX_ROOKS];
int queen_list[MAX_QUEENS];
```

Figure 2.4: Source-code listing for the definition of detailed piece lists.

2.3.3 Light and Dark Squares

A Bishop on a light square will be designated as *light Bishop* from now on. Likewise, a Bishop on a dark square will be referred to as *dark Bishop*. The separation of Bishops on light squares and dark squares within the piece lists can be quite useful when developing algorithms for attack detection and evaluation patterns.

The light Bishop can only be located on the 32 light squares (→ `b1`, `d1`, ..., `e8`, `g8`) of the board. Thus, the unnecessary examination of the light Bishop will not occur, if, for example, an attack on a dark square is detected. Apart from higher efficiency and the possibility to develop algorithms more precisely, the information about the square colour of a Bishop is delivered implicitly and does not have to be investigated explicitly. As a result, the bishop pair cannot merely be defined by the existence of two Bishops, but by the existence of a light Bishop and a dark Bishop as a complementary pair. IM Kaufman mentioned this quality of the bishop pair in the context of the duplication of function without redundancy [32].

> *"With two bishops traveling on opposite colored squares there is no possibility of any duplication of function."*

For example, insufficient protection on squares of one colour is evaluated by certain pawn formations and single Bishops in evaluation patterns of the interactive piece evaluation. Here, a malus for the so-called *Melanpenie* or *Leukopenie* is assigned depending on additional criteria [33].[9]

The source-code listing of Figure 2.4 is extended by the splitting of the piece list for Bishops

```
int bishop_list[MAX_BISHOPS];
```

into one piece list for light Bishops and one for dark Bishops. In some computer-chess engines underpromotions were often not considered, such as in ARISTARCH by Zipproth or in RYBKA by Rajlich, in order to simplify move generators. In the real chess game only the knight promotion is relevant as this is not redundant to the queen promotion. The rook promotion and bishop promotion will be only interesting in very few endgames, if a threatening stalemate can be prevented by that. In this respect, the light Bishop and the dark Bishop should be managed in several piece lists and not in single integer variables (in analogy to the King) in order to be able to take into account these very rare underpromotions.

2.3.4 Forward | Reverse Piece-list Scan

The advantage of the fine division of information can be recognised in the sequential piece-list scan. If, for example, the piece counter for the opposing light Bishops is zero and the friendly King occupies a light square, an attack by the opposing Bishops will be impossible. Extracting of such trivial conditions leads to even higher efficiency in the algorithms for the recognition of attacks.

The king square is stored in a further integer. Of course, piece counters are also to be managed for every piece list. In total, we arrive at 6 piece lists with 6 piece counters for each White and Black, and also one integer for every king square. Access to these contiguous piece lists is realised by a sequential loop.

Piece lists can be processed forwards with a `for`-loop and reversed with a `while`-loop. In case of a `for`-loop, the first element of the list will be processed first. In case of a `while`-loop it is beneficial to select the last element first. The two possibilities of the piece-list scan are presented in the source-code listing of Figure 2.5. Attention is to be paid to the pre-decrementation in case of the reverse piece-list scan as decrementation is to be executed prior to access to the sequential piece list.

2.3.5 Incremental Piece-list Update

Below we discuss the incremental piece-list update and compare this straightforward technique with the Rotated Bitboards.

[9] Hans Kmoch describes insufficient occupation of the dark squares by Pawns and the dark Bishop as Black poverty (Melanpenie) and insufficient occupation of the light squares by Pawns and the light Bishop as White poverty (Leukopenie).

Forward Piece-list Scan

```
// example: scan white Rooks
for (pieces=0; pieces < white_rooks; pieces++) {
  source=white_rook_list[pieces];
  // insert special code for piece handling here
}
```

Reverse Piece-list Scan

```
// example: scan white light Bishops
pieces=white_light_bishops;
while (pieces > 0) {
  source=white_light_bishop_list[--pieces];
  // insert special code for piece handling here
}
```

Figure 2.5: Source-code listing for forward | reverse piece-list scan.

Doing and Undoing Moves

The incremental update of a contiguous piece list in the function for doing moves is only possible with the information of a further incremental index board. This index board contains the index for the corresponding piece list for every square, which is occupied by a piece. If, for example, a white Rook is on square `h7`, and this Rook is entered in the white rook list on the second position, $index = 1$ will appear in the index board on square `h7`.[10] Both the contiguous piece lists and piece counters as well as the index board must be updated in the function for doing moves as shown in the source-code listing of Figure 2.6.

In case of undoing a move, as shown in the source-code listing of Figure 2.6, the same steps are performed as with doing a move by swapping the source square ($\rightarrow$ from) and the destination square ($\rightarrow$ to).

Do a move in from $\rightarrow$ to direction

```
// example: white light Bishop is moving
index=white_index[from];
white_index[to]=index;
white_light_bishop_list[index]=to;
```

Undo a move in to $\rightarrow$ from direction

```
// example: white light Bishop is moving
index=white_index[to];
white_index[from]=index;
white_light_bishop_list[index]=from;
```

Figure 2.6: Source-code listing for piece-list update while doing a move.

[10] In C/C++ entries in vectors always start with the index zero.

Doing and Undoing Capture Moves

Doing a capture move must be performed explicitly in order to update incrementally the piece list and the piece counter of the opponent as well as the index board. In the source-code listing of Figure 2.7 the piece counter of Black is decremented, and the last piece of the piece list is copied on the place of the captured piece.

Undoing a capture move, as shown in the source-code listing of Figure 2.7, is performed in the same way as in the example of doing a capture move. The source square and the destination square are only swapped. The square of the captured piece is attached to the end of the piece list. The current position of the piece in the piece list is entered in the index board. Finally, the piece counters are incremented.

Do a capture move in from → to direction

```
// example: black Knight is captured
black_pieces--;
black_knights--;
index=black_index[to];
square=black_knight_list[black_knights];
black_knight_list[index]=square;
black_index[square]=index;
```

Undo a capture move in to → from direction

```
// example: black Knight is captured
black_knight_list[black_knights]=to;
black_index[to]=black_knights;
black_knights++;
black_pieces++;
```

Figure 2.7: Source-code listing for piece-list update while doing a capture move.

Incremental Piece-list Update versus Rotated Bitboards

The competitiveness in speed of a computer-chess architecture depends strongly on the redundancy of the data to be managed. Therefore, the computing time for the incremental update of piece lists, piece counters, and the index board while doing and undoing moves is compared with the incremental update of the non-rotated bitboard (→ `a1h1`-orientation) and the three rotated bitboards (90° left → `a1a8`-orientation, 45° right → `a1h8`-orientation, 45° left → `h1a8`-orientation) [27] within the scope of a Rotated Bitboard computer-chess architecture. These results depend on the sophistication of the bitboard implementation. Therefore, we implemented the additional rotated bitboards into the bitboard computer-chess engine of Loop Amsterdam in order to ensure convincing results.

The benchmark test is carried out on a 64-bit computer environment. Four chess positions are taken from the middlegame and the endgame respectively. The required computing time is measured, in order to do and undo all legal moves (capture moves and non-capture moves) 10^7 times of the corresponding

position. The chess positions are randomly selected and do not favour any of these two technologies.

In Table 2.2 the results of measurement of this benchmark test are summarised. In the first column the numbers of the benchmark test positions are entered. The second and third columns contain the number of legal capture moves ($\rightarrow$ *capt*) and non-capture moves ($\rightarrow$ *non-capt*) of the corresponding test position respectively. In the fourth and fifth columns the CPU times are entered for doing and undoing of $10^7 \times (capt\_moves + non\_capt\_moves)$ of the New Architectures and the Rotated Bitboards. In the sixth column the performance improvement is entered in per cent of the New Architectures from Loop Leiden in comparison with Rotated Bitboards.

Performance Comparison: 10^7 Iterations per Position					
	moves (n)		time (seconds)		rel (%)
position	capt	non-capt	New Architectures	Rotated Bitboards	improvement
middlegame					
1	3	53	18.2	21.4	17
2	3	39	14.7	17.2	17
3	5	41	15.3	17.9	17
4	3	40	14.9	17.6	18
endgame					
1	0	15	5.0	6.0	20
2	3	31	12.2	14.4	18
3	0	25	8.5	9.9	16
4	1	16	6.0	7.0	16

Table 2.2: Performance comparison of New Architectures and Rotated Bitboards.

Whereas the New Architectures with detailed piece lists from Loop Leiden can do and undo approximately $29.6 \times 10^6 \frac{moves}{second}$ in middlegame positions, the Rotated Bitboard technology can do and undo approximately $25.2 \times 10^6 \frac{moves}{second}$ in middlegame positions. The result of this measurement in endgame positions is almost identical. It may be concluded that the detailed piece list based technology of Loop Leiden could perform the basic board operations approximately 17% faster such as doing-undoing moves and capture moves than the Rotated Bitboards.

2.3.6 Random Piece Arrangement

The random piece arrangement in the piece lists is a difficult phenomenon while scanning the pieces (see Subsection 2.3.4). The evaluation and the SEE rarely produce different results because at least two sliding pieces (especially two Rooks) from the same piece list should aim at the same destination square. Below we will discuss this issue.

Due to the doing and undoing of capture moves, the contents of the piece lists are mixed time and again. By the capture of a piece, the last piece of the piece list is added to replace the captured piece. Then, the piece counter is decremented (see source-code listing of Figure 2.7). During the undoing of this capture move, the captured piece is placed at the end of the piece list, and the piece counter is incremented (see source-code listing of Figure 2.7). The contents of piece lists are continuously mixed by this algorithm. So, the order of the stored contents is changed, which can lead to differences in evaluation patterns and in static exchange evaluations especially in the context of hidden attackers.

The order of pieces in the piece list at any time during the computation is thus accidental and neither deterministic nor reproducible. The order while scanning the bits in the bitboard computer-chess architecture is, in contrast, always the same as active bits of a certain bitboard are sequentially scanned via bit scan forward or bit scan reverse (see Section 3.5).

The example position of Figure 2.8 is taken from a random measurement, and does not have any deeper chess background as the game is won by Black according to 1...♖a1 2 ♖d1 ♖×d1 3 ♕×d1 ♘e3.[11] Here, we only illustrate the problematic nature of the random piece arrangement in piece lists. The capture move 1...♕×d4 is to be examined more precisely in the context of a static exchange evaluation. The destination square `d4` is threatened by both white Rooks on square `d2` and square `c4`. The black Rook on square `a4` is a hidden attacker, and only prevented from moving to square `d4` by the white Rook on square `c4`. The order of captures of the two white Rooks plays a decisive role for the further static exchange evaluation. Only with 2 ♖c×d4 the hidden black attacker is activated, which changes the static exchange evaluation. The interested readers can compute themselves both static exchange evaluations after reading Chapter 4 and in particular the example in Section 4.4.

Up to now this problem has been reported by researchers only in context of Rooks. Theoretically, it can also occur with two light Bishops, two dark Bishops, or two Queens. In general, this phenomenon occurs only in cases where two sliding pieces from the same piece list aim at the same destination square, and at least one of these two pieces prevents an opposing sliding piece from moving to the same destination square. Romstad [50] observed a similar problem in the context of a passed Pawn asymmetry. However, in this case, the order of the bit scan caused the unintentional evaluation asymmetry.

2.4 Experiments with Pieces in Lists[12]

The number of pieces in the piece lists is measured during the computations. The measured results are statistically evaluated. In order to obtain a better understanding about the performance development in the different game stages of the chess game we perform two experiments (one for the middlegame and one for the endgame). The used test positions do not have any great influence on

[11] This position was analysed by Loop as follows (one variation reads): 4 ♘d3 ♕×d4 5 ♖×d4 ♘×d1 6 b×c6 ♘c3 7 ♖×g4 ♘×g4 8 ♗f3 ♘×f2 9 ♔×f2.

[12] The results of the following experiment were presented at the Ph.D. Day at the Maastricht University, 2005.

Static Exchange Evaluation Problem for ... ♕×d4

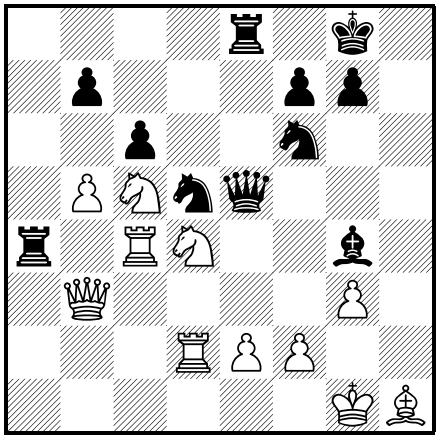


Figure 2.8: SEE problem with white Rooks in different piece-list orders.

the results provided that the number of pieces is almost identical. The average number of pieces in the piece list and their standard deviation are used for interpretation of these results.

This section contains two subsections. In Subsection 2.4.1 the experiment with pieces in lists will be described. In Subsection 2.4.2 the results will be discussed.

2.4.1 The Experiment

For the testing procedure, counters for the number of pieces in the piece lists are implemented at suitable places in the source code of the computer-chess architecture of Loop Leiden. In this way, it is possible to determine the statistical distribution of the number of pieces. In Table 2.3 average middlegame positions and endgame positions are taken as an experimental basis.[13] In the first column piece numbers for a piece list are entered in ascending order. In the second column the measured piece counters of middlegame positions deliver the information about how often a piece list contained n pieces. In the third column the relative proportion of pieces in the piece lists was computed. In the fourth and fifth columns the analogous experimental results are entered for a selection of endgame positions.

2.4.2 Results

In middlegame positions mostly $pieces \leq 2$ are entered in the piece list. $3 \leq pieces \leq 4$ per piece list was measured rarely. The occurrence of $pieces = 5$ per piece list is in turn a little more frequent, which is due to the separated piece lists

[13] All positions shortly after the opening are referred to as middlegame positions. All positions before the end of a game are designated as endgame positions.

Pieces in Lists				
	middlegame results		endgame results	
pieces (n)	abs	rel (p_{pieces})	abs	rel (p_{pieces})
0	13,500	0.11	173,900	0.57
1	63,100	0.53	100,700	0.33
2	34,600	0.29	23,200	0.08
3	2,070	0.02	2,900	0.01
4	2,400	0.02	4,000	0.01
5	3,900	0.03	1,800	0.01
6	90	0.00	0	0.00
7	0	0.00	0	0.00
8	0	0.00	0	0.00
sum	119,660	1.00	306,500	1.00

Table 2.3: Pieces in lists in middlegame and endgame phases.

for Pawns. In contrast, in the endgame approximately 90% of $pieces \leq 1$ are entered in the piece lists. The occurrence of $0 \leq pieces \leq 2$ in the middlegame applies even to approximately 93% of all piece lists. This makes us presume that the massive use of detailed piece lists especially with a decreasing number of pieces works more efficiently. For computation of the average value of pieces per piece list and the standard deviation the formulas from Equations 2.9 and 2.10 are used [9, pages 750f.].

Average Value and Standard Deviation for Pieces in Lists

$$\mu = \sum_{pieces=0}^{8} pieces \times p_{pieces} \tag{2.9}$$

$$\sigma = \sqrt{\sum_{pieces=0}^{8} (pieces - \mu)^2 \times p_{pieces}} \tag{2.10}$$

For computation of average values $\mu_{middlegame}$ and $\mu_{endgame}$ the number n of pieces in column 1 from Table 2.3 is to be multiplied with the relative frequencies from columns 3 and 5 as in Equation 2.9. The standard deviations $\sigma_{middlegame}$ and $\sigma_{endgame}$ are computed according to Equation 2.10. The average value in the middlegame is $\mu_{middlegame} = 1.45$ and the corresponding standard deviation is $\sigma_{middlegame} = 1.00$. The average number of pieces in lists in the endgame is with $\mu_{endgame} = 0.60$ clearly lower, which is due to the decreasing number of pieces. Thus, the standard deviation in the endgame $\sigma_{endgame} = 0.63$ is smaller. Accordingly, an acceleration of basic computations within a computer-chess architecture directly depends on the decreasing number of pieces.

2.5 Blocker Loops for Sliding Pieces

In Section 2.3 the management and organisation of information in piece lists have been examined. In this section the sliding qualities of Bishop, Rook, and Queen are analysed. The sliding of Bishop, Rook, or Queen is the most complex movement on the chessboard as a blocking piece can occupy every square of a sliding direction.

When generating non-capture moves, all accessible squares on the chessboard are entered in a move list. Only interposing squares of a sliding direction must be analyzed when recognizing a threat to a King.[14] When generating moves, the recognition of attacks and evaluation of mobility loops are used in order to analyse the sliding of Bishops, Rooks, and Queens. Almost every non-bitboard computer-chess architecture consists of hundreds of different `for`, `while`, and `do-while` loop constructions. Nevertheless, this tool mostly used in the non-bitboard computer-chess architecture has been hardly examined up to now. In this section (1) the *sliding-direction blocker loop* and (2) the *destination-source blocker loop* are introduced and examined. In spite of the variety of applications, these two blocker loops are sufficient to cover all computations in the context of sliding pieces within a state-of-the-art computer-chess engine such as Loop Leiden.[15]

This section contains two subsections. In Subsection 2.5.1 the sliding-direction blocker loop will be introduced. In Subsection 2.5.2 the destination-source blocker loop will be introduced.

2.5.1 The Sliding-direction Blocker Loop

The use of loops enables a sliding piece to slide over the chessboard. In this case, the piece slides from its source square in a direction until it collides with another piece or the board border (see Section 2.2). This collision is processed explicitly. Finally, this sliding process is repeated with the further sliding directions until all sliding directions of this piece are examined. A `h1a1` sliding-direction blocker loop with subsequent evaluation of the blocker squares is presented in the source-code listing of Figure 2.9.

2.5.2 The Destination-Source Blocker Loop

Quite often information is required, whether a piece can be attacked by another piece. From this information it is then possible to gain more complex knowledge about pinned pieces, x-ray attacks, etc. In this case only sliding pieces are analysed. It must be examined whether a destination square can be attacked by a sliding piece on a source square. In order to dispense with the additional condition, whether the destination square is already reached, the sliding direction is simply reversed. Thus, a sliding piece slides now from the destination square over the chessboard to its source square. The additional condition, whether the

[14] Interposing squares exist only in sliding directions since every interposing square can contain a blocking piece which blocks sliding from a source square to a destination square.

[15] The computer-chess architecture of Loop Leiden implements approximately 320 blocker loops.

Collision Detection while Sliding along a Direction

```
increment=INCREMENT_h1a1
// start the blocker loop in h1a1-direction
for (next_square=from+increment;
     piece(next_square) == NONE; next_square+=increment);

if (piece(next_square) & enemy) {
  // a capture is possible: insert code here to determine this case
}
```

Figure 2.9: Source-code listing for sliding and collision detection.

destination square is reached, is thus redundant within the blocker loop. At the latest, when reaching the source square the corresponding destination-source blocker loop is terminated as it comes to the collision with the piece itself.

A typical destination-source blocker loop from Loop Leiden is presented in the source-code listing of Figure 2.10. After retrieving the increments from a one-dimensional look-up table (see Subsection 2.2.4), a necessary condition is checked. Only when the source square is attainable from the destination square by a pseudo-legal move, the blocker loop will be launched.

Sliding from Destination Square to Source Square

```
// a piece is sliding in destination-source direction
increment=queen_increment(OFFSET+destination_square-source_square);
if (increment != 0) {
  for (next_square=destination_square+increment;
       piece(next_square) == NONE; next_square+=increment);

  if (next_square == source_square) {
    // a sliding piece on source_square can attack
       destination_square
  }
}
```

Figure 2.10: Source-code listing for sliding in destination-source direction.

2.6 Experiments with Loop Iterations[16]

In the same way as in the previous experiment in Section 2.4, the loop iterations within blocker loops are determined in the experiment below. The gained results are also used for the prediction and interpretation of the computer-chess architecture performance in ratio to the game phase. Of course, the following results of measurement are subject to small deviations. For this reason, comparisons between game phases are to be evaluated qualitatively.

[16]The results of the following experiment were presented at the Ph.D. Day at the Maastricht University, 2005.

This section contains two subsections. In Subsection 2.6.1 the experiment with loop iterations will be described. In Subsection 2.6.2 the results will be discussed.

2.6.1 The Experiment

For the testing procedure, counters for the number of loop iterations are implemented at suitable places in the source code of the computer-chess architecture of Loop Leiden. Thus some computations, that are based on blocker loops, can also be optimised specifically.

Middlegame positions and endgame positions were taken as an experimental basis for the results in Table 2.4. In the first column the number of possible loop iterations is entered in ascending order. In the second column the measured counters of middlegame positions provide the information on how often loops with n iterations were executed. In the third column the relative share of loops with n iterations $p_{iterations}$ was computed. In the fourth and fifth columns the analogous experimental results are entered for a selection of endgame positions.

Loop Iterations				
	middlegame results		endgame results	
iterations (n)	abs	rel ($p_{iterations}$)	abs	rel ($p_{iterations}$)
0	34,000	0.44	27,000	0.28
1	19,000	0.24	21,700	0.22
2	10,400	0.13	12,700	0.13
3	7,000	0.09	13,600	0.14
4	2,600	0.03	10,500	0.11
5	2,200	0.03	5,100	0.05
6	400	0.01	3,000	0.03
7	2,000	0.02	900	0.01
sum	77,200	1.00	93,800	1.00

Table 2.4: Loop iterations in middlegame and endgame phases.

2.6.2 Results

In middlegame positions the blocker loop is terminated in 90% of all computations after $iterations \leq 3$. Only in fewer than 4% of all computations the corresponding blocker loop is terminated after $6 \leq iterations \leq 7$. In endgame positions the blocker loop is terminated in 88% of all computations after $iterations \leq 4$. Due to the chessboard, which becomes increasingly empty, more iterations must be computed in the later phase of a chess game. The statistical evaluation of this experiment is carried out in the same way as the previous experiment in Section 2.4 according to the formulas for the average value in Equation 2.11 and the standard deviation in Equation 2.12 [9, pages 750f.].

Average Value and Standard Deviation for Loop Iterations

$$\mu = \sum_{iterations=0}^{7} iterations \times p_{iterations} \tag{2.11}$$

$$\sigma = \sqrt{\sum_{iterations=0}^{7} (iterations - \mu)^2 \times p_{iterations}} \tag{2.12}$$

For the computation of average values $\mu_{middlegame}$ and $\mu_{endgame}$ the number of iterations in column 1 from Table 2.4 is to be multiplied with the relative frequencies from columns 3 and 5 as in Equation 2.11. Standard deviations $\sigma_{middlegame}$ and $\sigma_{endgame}$ are computed according to Equation 2.12. The average value in the middlegame is $\mu_{middlegame} = 1.20$, and the standard deviation is $\sigma_{middlegame} = 1.00$. The average number of iterations in the endgame is 50% higher by $\mu_{endgame} = 1.80$, which depends on the chessboard, which becomes increasingly empty. The standard deviation $\sigma_{endgame} = 1.30$ is also a little higher in the endgame. The number of pieces decreases more strongly, and is contrasted to the increasing number of iterations within the blocker loops. In spite of that, a significant acceleration of the computation speed can be observed within the non-bitboard computer-chess architecture.

2.7 Answer to Research Question 1

In this chapter we introduced the internal computer chessboard in Section 2.2, the detailed piece lists in Section 2.3, and the blocker loops in Section 2.5. Only if these three techniques are in a harmonious interplay, a high-performance framework for the highest requirements on a computer-chess engine can be implemented. All algorithms for move generation, attack detection, mobility evaluation, and static exchange evaluation (see Chapter 4) from Loop Leiden are based on these three techniques.

The requirements for the development of a computer-chess architecture - (1) competitiveness in speed, (2) simplicity, and (3) ease of implementation - were defined at the beginning of this chapter. In the further course of this chapter the new computer-chess architecture of Loop Leiden is critically scrutinized and compared with Rotated Bitboards [27]. Below we examine these three requirements with respect to the implementations of Loop Leiden.

Competitiveness in Speed

The management overhead of data is quite small, since the information is managed in detailed piece lists and no explicit redundancy as with Rotated Bitboards is necessary. The incremental expenditure of doing and undoing moves (see Subection 2.3.5) is minimal as in each case only the corresponding piece list and the index board of White or Black must be updated (see Subsection 2.3.5). Doing and undoing capture moves is also highly efficient as only the

opposing piece list must be updated, and the corresponding piece counter must be decremented and incremented.[17]

Simplicity

As no absolute square information is required for addressing the look-up tables within a computer-chess architecture, it can be dispensed with the use of bijective two-dimensional vectors. Due to the seven-square horizontal board border, the implementation of more memory-efficient and more performance-efficient surjective one-dimensional vectors succeeded. For accessing the look-up table only the distance between source square and destination square is required. Moreover, only basic arithmetic operations are used (→ plus, minus, increment, and decrement) in the following cases: (1) while accessing and manipulating piece lists and (2) while executing blocker loops. Code implementations, such as code blocks (`for`- and `while`-loops, `if`- and `switch`-statements) and functions, are never nested deeper than three times due to the skilful use of the one-dimensional look-up tables, detailed piece lists, and blocker loops in the computer-chess architecture of Loop Leiden.

Ease of Implementation

Due to the detailed piece lists, a specific code must be written for every single piece (→ Pawn, Knight, light Bishop, etc.). Thus, the code can be optimised and simplified quite well. Redundant `if`- and `switch`-statements within move generators are completely excluded. The detailed piece counters can be used in evaluation patterns (recognition of the bishop pair, etc.). Unlike bitboard computer-chess architectures, the entire computer-chess architecture is competitive in speed on almost all software and hardware environments. Moreover, this architecture can simply be used in other chess variants [7] (Gothic Chess, 10×8 Capablanca Chess, Glinski's Hexagonal Chess, etc.) and board games, such as computer Shogi.

Empirical Results

A brute-force performance test follows these theoretical considerations. For this test two recursive brute-force algorithms (see source-code listings in Appendix A.1) are implemented in the computer-chess architecture of Loop Leiden and in an exactly analogous computer-chess architecture, which is based on Rotated Bitboards.

In the case of a brute-force recursion, all nodes of a tree are examined. Therefore, the results of the measurements are well comparable. Both recursive algorithms examine exactly the same number of nodes. The only difference is the order of the measured nodes. In the first algorithm (→ `basic_recursion`) all moves are done-undone sequentially. In the second algorithm (→ `advanced_recursion`) the moves are sorted a priori according to heuristic procedures. Thus, even more

[17] Loop Leiden: In the middlegame $\geq 50\%$ of all done-undone moves are capture moves because of the quiescence search. In the endgame the amount of capture moves decreases to $\leq 20\%$.

sophisticated move generators (→ move generators for check evasions, capture moves, etc.) and SEEs (see Chapter 4) are considered in the second brute-force algorithm. The second algorithm is close to a sophisticated implementation of an $\alpha\beta$-search algorithm without search window. Furthermore, the measurements are always reproducible and therefore well comparable.

A closed position from the chess game GridChess vs. Loop Amsterdam (→ 8 ♗d3 b6 9 ♗b2, see Appendix C.2) and an open position from the chess game Loop Amsterdam vs. Shredder (→ 45 g×f4 ♕d8 46 b×c5 ♖d7, see Appendix C.2) were used for this experiment. The measurements are carried out with both computer-chess architectures in the 32-bit mode and in the 64-bit mode. In each case the CPU times in milliseconds (ms) are measured for the basic recursion and the advanced recursion. In total, 16 results of measurement are summarised in Table 2.5.

Brute-Force Performance Tests				
	New Architectures		Rotated Bitboards	
	basic	advanced	basic	advanced
	closed position			
32-bit (ms)	97,930	144,070	136,250	191,170
64-bit (ms)	78,000	135,340	81,090	139,290
improvement (%)	25	6	68	37
	open position			
32-bit (ms)	39,290	72,200	73,510	101,400
64-bit (ms)	31,180	67,100	42,040	70,480
improvement (%)	26	7	74	43

Table 2.5: Brute-force performance tests with New Architectures and Rotated Bitboards.

As expected, the computer-chess architecture based on Rotated Bitboards profits enormously from 64-bit computer environments in this experiment, with a speed improvement of 37 to 74%. In spite of the almost ideal conditions on the 64-bit computer system, the performance of the Rotated Bitboards is 3 to 5% worse than the performance of the computer-chess architectures of Loop Leiden, called *New Architectures*. In the 32-bit mode Loop Leiden is 32 to 87% faster than the Rotated Bitboards. The high performance difference in open positions is particularly remarkable. For this reason, the move generators of Loop Leiden work especially efficiently in open positions, which depends on the use of the blocker loop from Section 2.5.

Conclusions Research Question 1

For research question one, we may conclude that it is possible to develop non-bitboard computer-chess architectures which are competetive in speed, simplicity, and ease of implementation. The extent to what this is possible is quite large, if we consider the performance of Loop Leiden in comparison with the best computer-chess engines (see tournament results in Appendix C.1). From

the empirical results given above we also may conclude that the implementation of sophisticated move generators, attack detectors, and static exchange evaluators for a state-of-the-art move ordering is more efficient than a computer-chess architecture based on Rotated Bitboards. Therefore, in the next chapter we will focus on the development of new techniques in relation to bitboards, but not in relation to Rotated Bitboards.

Future Research

Since the Rotated Bitboards are less efficient than the New Architectures from Loop Leiden in the 64-bit mode on a 8×8 board, the implementation of this technology in computer Shogi with its 9×9 board is especially interesting. Computer Shogi seems to offer even greater progress in the field of board architectures as the researchers have mainly experimented with the use of Bitboards so far [23].

Chapter 3

Magic Hash Functions for Bitboards

The main objective of Chapter 3 is to answer the second research question on the basis of the scientific research and development of the computer-chess engine Loop Amsterdam 2007. Below we repeat the second research question.

> **Research question 2:** *To what extent is it possible to use hash functions and magic multiplications in order to examine bitboards in computer chess?*

In this chapter the basics of the magic hash approach and magic hash functions will be examined. The use of the magic hash approach based on magic multiplications [30] has been discussed in different internet developer forums [36, 44] since its publication by Kannan. Many extensive publications, measurements, and hardware-independent developments followed, which are all based on the technology of the magic hash functions.

The approach of the magic hash functions is to be preferred to the alternative perfect hash functions by Fenner *et al.* [21], since their hash approach is less universal. In addition, according to Fenner *et al.* their perfect hash functions are not more efficient than Rotated Bitboards, but more space-efficient than magic hash tables. A magic hash algorithm for the management of bitboards as the compressed data media will consist of the following three elements [30].

1. **A magic hash function and a magic multiplier.** Both are used for (a) a magic bit scan or a magic hash algorithm that manage only information for a unique square or for (b) a magic hash function and a magic multiplier set for a magic hash algorithm that manage, for example sliding directions for several squares (see Section 3.6).

2. **A function for the initialisation of the hash table.** The function is based on the magic hash function and the magic multiplier or the magic multiplier set. This function calculates all conceivable bit combinations

(bitboards → input keys) and their solutions. Hash addresses are computed by the input keys via the magic hash function. Solutions are entered in the corresponding hash addresses.

3. **A function to access the initialised hash table.** The magic hash function generates the address for a magic bit scan in order to access the hash table. In contrast, for (combined) sliding directions for a magic hash algorithm the function must generate additional offsets in order to access the initialised hash table, since a contiguous memory space (a sub-hash table) is assigned to every square in the hash table.

For the implementation of an entire computer-chess architecture based on the magic hash, several independent magic hash algorithms, sub-hash tables, and hash tables to access the bit positions and 64-bit bitboards are to be developed.

The use of 64-bit unsigned integers (bitboards) within the scope of a computer-chess architecture is quite memory-efficient for the management of board related information. However, the use of bitboards has been redundant and inefficient up to now (see empirical results in Section 2.7) as board related information for the computation of sliding movements is to be managed in parallel many times (→ Rotated Bitboards). In order to examine bitboards efficiently and with minimum redundancy, fast and simple (perfect) hash functions must be developed.

This chapter is organised as follows. In Section 3.1 representation and orientation of bitboards are defined. In Section 3.2 the most important theoretical basics and relations between the magic hash function and the magic multiplication with powers of two and n-bit integers are introduced. In Section 3.3 the index mappings are derived for a general n-bit bit scan and compound sub-hash tables for (combined) sliding directions in order to address the hash tables with the unique magic index. In Section 3.4 a manual trial-and-error procedure for the generation of magic multipliers according to Kannan [30] is presented.

In Section 3.5 the magic hash for a *bit scan forward* (BSF) is developed. Its mode of operation can also be used for a *bit scan reverse* (BSR) that already implements all elements of an independent magic hash algorithm. The bit scan within a bitboard computer-chess engine is necessary to access bit positions in bitboards while e.g. generating or evaluating pieces, etc.

In Section 3.6 a magic hash for sliding directions on the basis of the theoretical foundations from Section 3.2 is developed. The implementation is, apart from a more complicated initialisation and the use of an offset-vector, quite similar to the implementation of a bit scan. Up to now, the generation of magic multipliers for magic hash algorithms has had to be managed only with appropriate trial-and-error approaches. In Section 3.7 a trial-and-error approach based on the research undertaken by Romstad [48] is developed in order to generate optimal magic multipliers (*optimal* according to Kannan [30, page 7]) for sliding directions. In Section 3.8 experiments are carried out with regard to the structure and the qualities of magic multipliers with (1) the trial-and-error algorithm and (2) the deterministic brute-force algorithm. The findings about magic hash functions for bitboard representation gained so far are summarized and discussed in Section 3.9 in order to answer the second research question.

3.1 Representation and Orientation of Bitboards

In the further course of this chapter all examples and derivations are represented in the binary system, since the logical structure, similar as in the hexadecimal system, can be visually better displayed than in other systems. Only with the operations left shift and right shift the bitboards with integers are shifted in the decimal system in order to be able to visualise the change of the positions of single active bits in the bitboards.

This section contains three subsections. The representation and orientation of 8-bit unsigned integers is dealt with in Subsection 3.1.1. The 16-bit unsigned integers (see Subsection 3.1.2) and 64-bit unsigned integers (see Subsection 3.1.3) can be displayed line by line as a matrix arranged in the binary system with the dimensions 4×4 or 8×8. A square shape of a bitboard is oriented as shown in Figure 3.1. The first element of this 64-bit bitboard corresponds with bit 56 and represents square `a8`. The last element of the 64-bit bitboard is equivalent to bit 7 and represents square `h1`.

56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	44	45	46	47
32	33	34	35	36	37	38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

Figure 3.1: The computer chessboard with a8h1-orientation.

3.1.1 8-bit Bitboards

The number one ($a = 1$) in the binary system stands for a single active bit and is thus a power of two. The multiplication ($a \times b$) of an integer (a) with a power of two ($b = 2^n$) is identical with the left shift operation ($a << n$). The following example in Figure 3.2 demonstrates how the multiplication of an 8-bit bitboard (8-bit unsigned integer, $0 \leq a \leq 2^8 - 1 = 255$) with a power of two ($b = 1000_{bin}$) is calculated. In this thesis a bitboard is always represented as an item with brackets around and with blanks between the bit positions. The orientation of such a 8-bit bitboard corresponds with the orientation of a rank of a chessboard from the white player's view. The representation of the 8-bit bitboard conforms to rank 1 of a chessboard from the white player's view. The representation of the bit positions of the 8-bit bitboard is thus, in contrast to the representation of an integer in the binary system, to be read from left to right.

1×8 **Representation and Orientation of 8-bit Bitboards**

$$\left| \begin{array}{cccccc} bit_0 & bit_1 & \dots & bit_6 & bit_7 \end{array} \right|$$

Multiplication and Left Shift of 8-bit Bitboards

$$\left| \begin{array}{cccccccc} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{array} \right| \times 1000_{bin} = \left| \begin{array}{cccccccc} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{array} \right|$$

Figure 3.2: Representation, orientation, and multiplication of an 8-bit unsigned integer.

3.1.2 16-bit Bitboards

It is the easiest way to represent a 16-bit bitboard (16-bit unsigned integer, $0 \leq a \leq 2^{16} - 1 = 65535$) as a square 4×4 bit matrix. In this case, the 4×4 bit matrix is to be oriented as a 4×4 chessboard from the white player's view. A multiplication with a power of two ($b = 1000_{bin}$) is carried out here as well. The result can be interpreted as a shift of the files of active bits to 3 binary positions (see Figure 3.3).

4×4 **Representation and Orientation of 16-bit Bitboards**

$$\left| \begin{array}{cccc} bit_{12} & bit_{13} & bit_{14} & bit_{15} \\ bit_8 & bit_9 & bit_{10} & bit_{11} \\ bit_4 & bit_5 & bit_6 & bit_7 \\ bit_0 & bit_1 & bit_2 & bit_3 \end{array} \right|$$

Multiplication and Left Shift of 16-bit Bitboards

$$\left| \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{array} \right| \times 1000_{bin} = \left| \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{array} \right| << 3_{dec} = \left| \begin{array}{cccc} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{array} \right|$$

Figure 3.3: Representation, orientation, and multiplication of a 16-bit unsigned integer.

3.1.3 64-bit Bitboards

The same as in the example in Figure 3.3, the `a`-file of a 64-bit bitboard displayed in the chessboard (64-bit unsigned integer, $0 \leq a \leq 2^{64} - 1 = 18446744073709551615$) can be transferred into the `h`-file by the multiplication with ($b = 10000000_{bin}$) (see Figure 3.4).

8×8 Representation and Orientation of 64-bit Bitboards

$$\begin{vmatrix} bit_{56} & bit_{57} & \cdots & bit_{62} & bit_{63} \\ bit_{48} & bit_{49} & \cdots & bit_{54} & bit_{55} \\ \vdots & & & \vdots & \\ bit_{8} & bit_{9} & \cdots & bit_{14} & bit_{15} \\ bit_{0} & bit_{1} & \cdots & bit_{6} & bit_{7} \end{vmatrix}$$

Multiplication and Left Shift of 64-bit Bitboards

$$\begin{vmatrix} 1 & 0 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \\ \vdots & & & \vdots & \\ 1 & 0 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \end{vmatrix} \times 10000000_{bin} = \begin{vmatrix} 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & \cdots & 0 & 1 \\ \vdots & & & \vdots & \\ 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & \cdots & 0 & 1 \end{vmatrix}$$

Figure 3.4: Representation, orientation, and multiplication of a 64-bit unsigned integer.

3.2 Hash Functions based on Magic Multiplications

An example of an n-bit magic hash function in Equation 3.1 demonstrates the general hash algorithm for the computation of a unique index. This unique index is needed to access the hash table in order to retrieve the searched information ($\rightarrow$ the result). For the computation of the unique index the two parameters *input* and *bits* are to be passed on to the magic hash function. The hash function processes these two variable parameters with the constant magic multiplier (*magic_multiplier*) and the constant number of bits (n). The amount of bits (n) is dependent on the data type of the first parameter (*input*). The second parameter (*bits*) describes the size of the hash table. $bits = log_2 n$ is used for the indexing of a single bit ($\rightarrow$ bit scan). The constant magic multiplier is from the same data type as the input key. For the processing of 64-bit input keys (*input*), as for example bitboards, a 64-bit magic multiplier is also to be used.

An Abstract *n*-bit Magic Hash Function

$$\begin{aligned} magic\_index(input, bits) &= \frac{input \times magic\_multiplier}{2^{n-bits}} \\ \Rightarrow result(magic\_index) &= hash\_table(magic\_index) \end{aligned} \tag{3.1}$$

Apart from the magic hash function and an n-bit magic multiplier, a further function for the initialisation of the hash table is required. The magic hash

function is called in this function in order to be able to process the magic index before the content of the hash table is addressed. This is quite an interesting case with the bit scan, since the same magic hash function and the same hash table are used for the bit scan forward and the bit scan reverse (see Section 3.5). The only difference in these two bit scans lies in the separation of a single active bit from a bitboard.

An additional offset is to be calculated for a magic hash algorithm for sliding directions, before the hash table can be addressed correctly (see Section 3.6). Here, access to the hash table is also separated from the computation of the magic index.

In the source-code listing of Figure 3.5 the declaration of the n-bit hash algorithm based on magic multiplications is displayed. The access to the magic hash, which Kannan mentions in his paper [30], is achieved by the function:

```
uint_n magic_hash_use(uint_n input, int bits);
```

The data type chosen here

```
uint_n
```

is to be defined explicitly as an unsigned integer with n bits.

Declaration of an n-bit Magic Hash Algorithm

```
uint_n magic_multiplier;
uint_n magic_hash_function(uint_n input, int bits);
uint_n magic_hash_initialise(void);
uint_n magic_hash_use(uint_n input, int bits);
```

Figure 3.5: Source-code listing for the declaration of an n-bit magic hash algorithm with its functions and the constant multiplier.

In the further course of this chapter we predominantly work with the unsigned integer data type $n = 8$ and $n = 64$. The reference to the chessboard can only be maintained with $n = 64$. Nevertheless, examples with $n = 8$ are easier to comprehend and to visualise as its representation is more compact.

This section contains three subsections. In Subsections 3.2.1 to 3.2.3 we will introduce the reader into the issues of the multiplication of bitboards with (1) a power of two and (2) an arbitrary n-bit unsigned integer.

3.2.1 The Magic Multiplication

In order to understand the magic multiplication more precisely, the splitting of a magic multiplier into its powers of two is helpful. Thus, it is possible to display the multiplication as a sum of powers of two. A multiplication with a power of two can also be written as a left shift according to Equation 3.2.

46 is an 8-bit magic multiplier, that is manually generated among further three 8-bit multipliers in Section 3.4. Through the splitting of the multiplier into powers of two and the use of the left shift operator the integer 46 can be written in the binary system.

The Left Shift

$$a \times 2^n = a << n \tag{3.2}$$

Splitting a Magic Multiplier in its Powers of Two

$$\begin{aligned} magic &= 46 \\ &= 2^1 + 2^2 + 2^3 + 2^5 \\ &= (1 << 1) + (1 << 2) + (1 << 3) + (1 << 5) \\ &= 00000010_{bin} + 00000100_{bin} + 00001000_{bin} + 00100000_{bin} \\ &= 00101110_{bin}. \end{aligned}$$

After the magic multiplier is split up into its powers of two, two different applications of the magic multiplication will be introduced.

1. **Power of two.** A magic multiplier is multiplied with a power of two $b = 2^i, 0 \leq i \leq n-1$. This operation is simply to be written by the left shift operator. The bits of the resulting bitboard are shifted to the left and the overflow is cut off. The so-called "index a 1 in a Computer Word" according to Leiserson *et al.* [37] is a more simple version of a hash algorithm, as there are only n-permutations after the isolation of a single bit in an n-bit word.

2. **Arbitrary n-bit integer.** A magic multiplier is multiplied by an arbitrary n-bit unsigned integer $0 \leq b \leq 2^n - 1$. This multiplication is more complicated to comprehend and to compute, since the magic multiplier is multiplied by a sum of powers of two, which cannot be explained with a basic shift operation. The indexing of arbitrary n-bit integers, i.e., multiple 1-bit computer words, with the DEBRUIJN approach has not been possible in the context of a computer-chess architecture up to now [37]. However, the bits are arranged in compound bit-patterns (diagonal, horizontal or vertical lines), so the number of the permutations decreases so far, so that the indexing is possible by means of magic multiplication.

In the further course both types of the magic multiplication will be examined in detail. Their specific applications in the environment of a computer-chess architecture will be introduced as well.

3.2.2 The Magic Multiplication by a Power of Two

The product of the magic multiplier and the power of two input key are used for the computation of an index for a bit scan in the context of a magic hash algorithm (see Section 3.5). Whether the input key of this hash function is the *least significant bit* (LSB) or the *most significant bit* (MSB), it does not affect the hash function, since both the LSB and MSB are powers of two. The

left shift and right shift of a magic multiplier with $n \in \{0, 1, 2, \ldots, 61, 62, 63\}$ in Equations 3.3 and 3.4 show the magic multiplication by a power of two by means of the left shift and right shift operation. The shift operations are already the most elementary forms of the multiplication of an unsigned integer (*magic*) with a power of two (a) or the reciprocal of a power of two (b).

Left Shift and Right Shift of a Magic Multiplier with n

$$n \in \{0, 1, 2, \ldots, 61, 62, 63\}$$

$$a = 2^n = 1 << n \quad \Rightarrow \quad magic \times a = magic << n \tag{3.3}$$

$$b = \frac{1}{a} = \frac{1}{2^n} = \frac{1}{1 << n} = 1 >> n \quad \Rightarrow \quad magic \times b = magic >> n \tag{3.4}$$

The product of the magic multiplier with all 64 possible powers of two is shown in the Equations 3.5. From exactly these products the indices to access a hash table for a magic bit scan will be computed in Section 3.5. Only bit positions, which are not shifted over bit position 63, are relevant for the further computing of the product. The lower bit positions are filled with zeros, the same as with the multiplication by integers. The filled zeros are to be considered when computing indices and are important for the precise addressing of the hash table. The associative, commutative, and distributive laws of mathematics apply to this multiplication by overflow within a ring [3].

Multiplication with all Possible Input Keys (Powers of Two)

$$\begin{array}{lcllllllll}
magic & \times & 2^0 & = & bit_{63} & bit_{62} & bit_{61} & \ldots & bit_2 & bit_1 & bit_0 \\
magic & \times & 2^1 & = & bit_{62} & bit_{61} & bit_{60} & \ldots & bit_1 & bit_0 & 0 \\
 & & \vdots & & & & & & & & \\
magic & \times & 2^{62} & = & bit_1 & bit_0 & 0 & \ldots & 0 & 0 & 0 \\
magic & \times & 2^{63} & = & bit_0 & 0 & 0 & \ldots & 0 & 0 & 0
\end{array} \tag{3.5}$$

3.2.3 The Magic Multiplication by an n-bit Integer

The product of the magic multiplier and the n-bit unsigned integer input key are also used, for example, to compute an index for a magic hash algorithm for sliding pieces (see Section 3.6). A further application field of the indexing of at most two 1's is the indexing of the positions of knight pairs [37, page 6], bishop pairs or rook pairs. However, the reduction of the hash table to a minimum size of 2^{12} entries fails. It is necessary instead, to allocate memory of 2^{15} entries in order to index arbitrary 64-bit masks with at least two active bits [37].

Before it will be possible to carry out the multiplication of the n-bit input key with an appropriate magic multiplier, the input key is to be scrutinized, as this consists of powers of two and as a result, the multiplication becomes considerably complicated. Unlike the scientific research undertaken by Leiserson *et al.*, only masks for sliding directions and no arbitrary "sparse multiple 1-bit computer

words" [37] will be discussed in this chapter. In this respect, the task can be performed easier, since the masks for sliding directions possess definitely fewer bit combinations than, for instance, a 64-bit unsigned integer input key with at least three randomly distributed active bits ($\rightarrow \frac{64!}{(64-3)!} = 249,984$ bit combinations).

The n-bit integer key is nothing else but an n-bit mask, or an "occupancy bitboard which has active bits on squares where pieces are placed and has inactive bits everywhere else" [30, page 6]. The precise relation between such a mask for filtering (combined) sliding directions and a magic hash algorithm will be examined in Section 3.6. At this stage, only the summation of a single shifted bitboard is to be analysed.

An input key of the hash function for sliding directions can be, for example a mask, that only possesses active bit positions along the (combined) sliding directions. The following 64-bit unsigned integer input key (*input*) is a possible input key of a Rook on square `d4`.

Example for a Possible 64-bit Input Key

$$input = \begin{vmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{vmatrix} = 2^{19} + 2^{25} + 2^{28} + 2^{30} + 2^{35} + 2^{51}$$

The n-bit integer input key (*input*) can be written generally as a sum of powers of two (see Equation 3.6). Here, the exponent p_i are 6-bit unsigned integers [30].

Generalized n-bit Integer Input Key

$$input = \sum_{i=0}^{n} 2^{p_i}, \quad 0 \leq p_i < p_{i+1} < 64 \tag{3.6}$$

The multiplication of the n-bit magic multiplier by the random n-bit input key can be split into the sum in Equation 3.7 due to commutativity and Equation 3.6 [30].

Since the representation of a detailed example of a 64-bit magic multiplication, based on a 64-bit input key and a 64-bit magic multiplier, is difficult to display due to the size of the bit matrix, we will focus on the 16-bit representation and orientation according to Subsection 3.1.2 in the example of Figure 3.6.

The magic product of $magic \times input$ cannot, unlike the Equations 3.5, be displayed in a simple way since a summation of component products must be carried out. Since, in this case, we compute with operations modulo 2^n, the

Definition of the Magic Product as a Summation of Leftshifts

$$\begin{aligned} magic\_product &= input \times magic \\ &= magic \times input \\ &= \sum_{i=0}^{n} (magic << p_i) \end{aligned} \tag{3.7}$$

Example for a 16-bit Magic Product

$$magic = \begin{vmatrix} bit_{12} & bit_{13} & bit_{14} & bit_{15} \\ bit_8 & bit_9 & bit_{10} & bit_{11} \\ bit_4 & bit_5 & bit_6 & bit_7 \\ bit_0 & bit_1 & bit_2 & bit_3 \end{vmatrix}, \quad input = \begin{vmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{vmatrix}$$

$$\begin{aligned} magic \times input &= magic \times (2^1 + 2^6) \\ &= magic << 1 + magic << 6 \\ &= \begin{vmatrix} bit_{11} & bit_{12} & bit_{13} & bit_{14} \\ bit_7 & bit_7 & bit_9 & bit_{10} \\ bit_3 & bit_4 & bit_5 & bit_6 \\ 0 & bit_0 & bit_1 & bit_2 \end{vmatrix} + \begin{vmatrix} bit_6 & bit_7 & bit_8 & bit_9 \\ bit_2 & bit_3 & bit_4 & bit_5 \\ 0 & 0 & bit_0 & bit_1 \\ 0 & 0 & 0 & 0 \end{vmatrix} \end{aligned}$$

Figure 3.6: Example for a 16-bit magic product.

summation of the shifted magic multiplier leads to carry effects and overflows. During the overflow at the multiplication by pure powers of two in Subsection 3.2.2, the overflow could be simply cut off, since carry effects complicate the computation of sums modulo 2^n.[1]

3.3 The Unique Magic Index

The computation of the magic index for the indexing of the corresponding hash table occurs via an appropriate hash function based on magic multiplications according to Section 3.2.[2] Both the hash table for a bit scan and the hash table for (combined) sliding directions are addressed via the magic hash function. The essential difference lies in the size of the hash table and the stored information.

A hash table for the management of bit positions for a bit scan is, of course, more memory-saving than a hash table for the management of bitboards, such as it is needed for a hash for sliding directions. On the one hand, there are considerably more combinations of bitboards for (combined) sliding directions. On the other hand, the specific memory requirement of a 64-bit bitboard ($\rightarrow$ 8-byte) is greater than that for the retention of bit positions ($\rightarrow$ 1-byte).

[1] We use modulo 2^n in the context of addition and multiplication of n-bit integers because computers have a fixed number of bits (e.g. 64-bit computer environments).

[2] A hash table in the context of computer-chess architectures can be a hash algorithm for a bit scan, sliding-direction hash for Bishops, Rooks or single lines like verticals, horizontals or diagonals.

This section contains two subsections. In Subsection 3.3.1 we will develop the index mapping for a bit scan. In Subsection 3.3.2 we will develop the index mapping for (combined) sliding directions.

3.3.1 Index Mapping for a Bit Scan

In the following, the computation of the magic index for a bit scan is presented. The implementation of an entire hash algorithm for a bit scan (forward | reverse) will be introduced later in Section 3.5. In this thesis the magic index is therefore called *unique magic index* or also *index mapping* [30]. The index mapping formulation refers, according to Kannan, to the magic product that is, strictly speaking, not an index mapping yet, since the complete mapping will be terminated by the division of the magic product with a power of two (see Equation 3.8).

In Equation 3.8 the computation of a unique magic index for a bit scan is presented. Without loss of generality, the implementation of the hash algorithm in Section 3.5 refers to $n = 64$-bit integer words. The general mapping function [30, pages 3-6], introduced here, has been already developed by Leiserson *et al.* [37] for n-bit and 8-bit unsigned integer words based on DEBRUIJN sequences. This straightforward and fast mapping function for n-bit unsigned integer words computes a unique magic index of the size $log_2 n$. As already shown in Equation 3.5, the unique magic index of an n-bit bit scan is a compound binary substring with a size of $log_2 n$ entries. In order to guarantee uniqueness, only the upper $log_2 n$ bits must be different after the multiplication of $input \times magic\_multiplier$. The hash index is equal to the unique magic index because the hash table is completely addressed by the very same magic multiplier and thus no additional offset has to be calculated.

The Magic Hash Index for a Bit Scan

$$magic\_index(input) := \frac{input \times magic\_multiplier}{2^{n-log_2 n}} \tag{3.8}$$

3.3.2 Index Mapping for Sliding Directions

The index mapping for (combined) sliding directions works as in Equation 3.8, provided that sliding directions are to be calculated for a single square. A complete magic hash algorithm for (combined) sliding directions consists of single sub-hash tables which contain bitboards of all 64 squares. Thus, an offset must be calculated before the computation of the unique magic index in order to address the correct sub-hash table.

For the computation of the move bitboard for a single square and a (combined) sliding direction all occupancy bits [30, page 6] must be extracted. A Rook on square `a1` can move onto maximum 14 squares. Pieces, that block the Rook in the case of sliding, can only occupy 12 out of 14 squares. Consequently, the border squares are not blocker squares for a sliding direction and therefore

do not have to be considered while computing the unique magic index. An enormous reduction of the hash table results from this because only $\frac{1}{2^2} = \frac{1}{4}$ of the memory capacity is required. In addition, the number of possible input keys decreases, which significantly simplifies the trial-and-error generation of magic multipliers (see Section 3.7). In total, there are 12 possible occupancy bits for a Rook on square `a1` and $2^{12} = 4096$ possible bit combinations for the input key.

In Figure 3.7 numbers of bits for the squares of a chessboard for the combined sliding direction of a Rook and a Bishop are summarised. They indicate a number of the occupancy bits for every single square. For example, exactly 11 bits are required for the sub-hash of a Rook $square \in B$ in order to manage all different bit combinations. The number of the occupancy bits is different, which is also to be considered while computing the offset index.

Number of Bits for the Unique Magic Indices for Sliding Rooks

$$\left.\begin{array}{cccccccc} C & B & B & B & B & B & B & C \\ B & A & A & A & A & A & A & B \\ B & A & A & A & A & A & A & B \\ B & A & A & A & A & A & A & B \\ B & A & A & A & A & A & A & B \\ B & A & A & A & A & A & A & B \\ B & A & A & A & A & A & A & B \\ C & B & B & B & B & B & B & C \end{array}\right\} \quad rook\_bits(square) = \begin{cases} 10 & : & square \in A \\ 11 & : & square \in B \\ 12 & : & square \in C \end{cases}$$

Number of Bits for the Unique Magic Indices for Sliding Bishops

$$\left.\begin{array}{cccccccc} B & A & A & A & A & A & A & B \\ A & A & A & A & A & A & A & A \\ A & A & C & C & C & C & A & A \\ A & A & C & D & D & C & A & A \\ A & A & C & D & D & C & A & A \\ A & A & C & C & C & C & A & A \\ A & A & A & A & A & A & A & A \\ B & A & A & A & A & A & A & B \end{array}\right\} \quad bishop\_bits(square) = \begin{cases} 5 & : & square \in A \\ 6 & : & square \in B \\ 7 & : & square \in C \\ 9 & : & square \in D \end{cases}$$

Figure 3.7: Number of bits for the unique magic indices for sliding Bishops and sliding Rooks on a chessboard.

The computation of the offset in Equation 3.10 is implemented via a linear vector. Therefore, offsets for the rook hash can be easily calculated as in Equations 3.9. All sub-hash tables are addressed with this offset. The unique magic index is calculated in the same way as already introduced in Equation 3.8. Merely the mapping of a single sub-hash depends on respective bits of a square. The hash index in Equation 3.12 is addressed in analogy with a two-dimensional vector [29, page 77] by addition of the offset and the magic index.

A less memory-efficient but a little more intuitive alternative for the managing of the hash table is realised by a *homogeneous array access* [30, page 6] (see Subsection 3.6.2). This memory management is not lockless [28, 43], and implements a one-dimensional vector with a constant offset or a two-dimensional

The Offset-Index Vector for a Rook Hash

$$\begin{aligned} offset\_index(a1) &= 0 \\ offset\_index(a2) &= 2^{12} = 4096 \\ offset\_index(a3) &= 2^{12} + 2^{11} = 6144 \\ &\vdots \\ offset\_index(g8) &= 2^{12} + 2^{11} + \ldots + 2^{11} = 96256 \\ offset\_index(h8) &= 2^{12} + 2^{11} + \ldots + 2^{11} + 2^{11} = 98304 \end{aligned} \tag{3.9}$$

vector. The *minimal array-access* for the offset calculation was introduced by Romstad, 2007 for the first time [48] (see Subsection 3.6.1). This approach proved itself to be as quick on a current 64-bit computer system as the homogeneous array access with constant offset.[3] However, hash entries would have to be allocated for every sub-hash $2^{12} = 4096$ ($\rightarrow$ rook hash) or $2^9 = 512$ ($\rightarrow$ bishop hash). 262144 hash entries ($\rightarrow$ 2048 kb) are needed for the rook hash with the constant offset and 102400 ($\rightarrow$ 800 kb) hash entries with the minimum offset. For the bishop hash the relation between the minimal array access 5248 ($\rightarrow$ 41 kb) and the homogeneous array access 32768 ($\rightarrow$ 256 kb) is even more favourable, amounting to approximately 80% of memory capacity.

The Magic Hash Index for (combined) Sliding Directions

$$offset\_index(square) = \sum_{i=a1}^{square-1} 2^{bits(i)} \tag{3.10}$$

$$magic\_index(input, square) = \frac{input \times magic(square)}{2^{64-bits(square)}} \tag{3.11}$$

$$hash\_index(input, square) = offset\_index(square) + magic\_index(input, square) \tag{3.12}$$

3.4 Construction of 8-bit Magic Multipliers

The magic multiplier in Equation 3.7 has been already defined in Section 3.2. A whole set of the magic multiplier is required for a more elaborate magic hash procedure, such as the magic hash for (combined) sliding directions (see Section 3.6). Preparatory for a magic hash for a bit scan, which implements only one magic multiplier, an 8-bit magic multiplier is constructed manually. This construction is quite similar to the construction of an 8-bit magic multiplier according to Kannan [30, page 3]. With the magic hash for a bit scan the mode of operation of the magic multiplication can be displayed clearly since the input key is only a power of two.

[3] A hash table with the constant offset would correspond exactly to a two-dimensional vector depending on a square.

Through the multiplication of the input key [30], which is an 8-bit power of two, with the magic multiplier $input\_key \times magic$ the magic multiplier is leftshifted. The subsequent division of the magic product by a power of two corresponds to a right shift. The computation of the unique magic index is the only purpose of this operation. Consequently, only $8 = 2^3 \rightarrow$ 3-bit sequences will be mapped into the bit position 1-3 as a unique magic index from the magic multiplier.

An 8-bit magic multiplier is developed manually in Table 3.8. Column 1 contains the input key. The input key is an 8-bit power of two, that is a bitboard with only one active bit. Column 2 contains the unique magic index. These column entries result from the 3-bit sequences which emerge by the construction of the magic multiplier. The magic multiplier in main columns 3 and 4 is again split up into its bits 1-8 and overflows 1-2. The 8-bit magic multiplier results from the compound 3-bit sequences, which are in each case a unique magic index.

8-bit Magic Multiplier												
		bit position								overflow		
input key	unique magic index	8	7	6	5	4	3	2	1	1	2	
$2^0 = 1$	0	0	0	0								
$2^1 = 2$	1		0	0	1							
$2^2 = 4$	2			0	1	0						
$2^3 = 8$	5				1	0	1					
$2^4 = 16$	3					0	1	1				
$2^5 = 32$	7						1	1	1			
$2^6 = 64$	6							1	1	0		
$2^7 = 128$	4								1	0	0	
	magic multiplier	0	0	0	1	0	1	1	1			

Figure 3.8: Manual construction of an 8-bit magic multiplier.

Via manual construction, there can be developed four ($\rightarrow 23, 2 \times 23 = 46, 29, 2 \times 29 = 58$) 8-bit magic multipliers. N-bit magic multipliers for a bit scan consist of $\frac{n}{2}$ active bits since all unique magic indices must be different and therefore the number of active bits is equal to the number of inactive bits. Moreover, magic multipliers for the indexing of a bit scan, that are DEBRUIJN sequences, start with $(log_2 n) - 1$ zeros [37, page 3, according to Steele (Sun Microsystems)].

These 4 magic multipliers can be verified with a basic brute-force algorithm, which goes through all numbers between 1 and $2^8 - 1$. With the same brute-force verification all $2^5 = 32$ magic multipliers for an unsigned integer 16-bit bit scan and all $2^{12} = 4096$ magic multipliers for an unsigned integer 32-bit bit scan can be detected by trial-and-error within a few minutes. The same brute-force approach would, of course, not deliver any results for the magic multipliers of a 64-bit bit scan, because the smallest 64-bit magic multiplier must have exactly 32 active bits in a non-contiguous pattern.

3.5 The Magic Hash Function for a Bit Scan

A particularly interesting implementation of a magic hash function is a bit scan [56]. A bit scan is supposed to return the position of the least significant bit (→ LSB) or most significant bit (→ MSB) in a bit matrix.[4] The isolation of the LSB in a bit matrix is to be solved in a more elegant and technically better way than the isolation of the MSB in a bit matrix. In order to generate unique magic indices with a suitable magic hash function (see Section 3.3), only bit matrices consisting of the isolated bit (→ power of two) are to be used. Therefore, the magic multiplier must generate unique magic indices only for these few 64 different bit matrices.

This section contains two subsections. In Subsection 3.5.1 the reader will be introduced in two competing bit scan implementations. In Subsection 3.5.2 we will develop and implement a corresponding magic hash algorithm for a bit scan.

3.5.1 Bit Scan Forward | Reverse

For the implementation of an efficient bit scan forward there are basically only two competing solutions: (1) the implementation via access to the native versions in hardware (→ bit scan intrinsic functions according to Microsoft) or (2) the construction of a suitable magic hash algorithm. Admittedly, native hardware implementations can be more efficient on a suitable computer environment. However, they are also quite specific and thus less portable between different computer platforms (→ x86, x64, *Extended Memory 64 Technology* (EM64T), *Itanium Processor Family* (IPF) [10]). Implementations in hardware via assembler instructions (→ BSF or BSR) or intrinsic functions, such as

```
unsigned char _BitScanForward(
  unsigned long * Index,
  unsigned long Mask);
unsigned char _BitScanReverse(
  unsigned long * Index,
  unsigned long Mask);
```

or in particular for 64-bit masks

```
unsigned char _BitScanForward64(
  unsigned long * Index,
  unsigned __int64 Mask);
unsigned char _BitScanReverse64(
  unsigned long * Index,
  unsigned __int64 Mask);
```

are not available on all computer environments. We give our implementation in Subsection 3.5.2.

[4] In the further course of this chapter the acronym LSB will be used for *least significant bit*, *rightmost bit*, or *low order bit*. The acronym MSB will be used for *most significant bit*, *leftmost bit*, or *high order bit*.

3.5.2 Bit Scan Forward Implementation

Our implementation of a bit scan forward via an entire magic hash algorithm consists of (1) a function for the initialisation of the hash table for a suitable magic multiplier, (2) the magic hash function for the computation of the unique magic indices, and (3) the bit scan forward function. The bit scan forward function retrieves the bit position after the isolation of the LSB with the magic hash function (see Section 3.3) and the initialised hash table.

The algorithm in the source-code listing of Figure 3.10 initialises a magic hash table

```
int magic_hash_table[64],
```

which contains all possible bit positions of an isolated bit of a 64-bit unsigned integer. Like in many programming languages, the first bit carries the bit position 0. From the bit matrix consisting of the isolated bit

```
uint64 isolated_bit
```

and the specific magic multiplier

```
const uint64 magic = HEX(0x218A392CD3D5DBF)
```

the magic hash function

```
int magic_function(uint64 isolated_bit, uint64 magic, int size)
```

computes the unique magic index:

```
int magic_index
```

Finally, the unique magic index addresses the hash table in order to store the information of the bit position. These steps are repeated for all 64 possible power of two bitboards in a loop. With this straightforward hash function further information about squares or bit positions on the chessboard can be managed easily in more extensive hash patterns. The source-code listing of a magic hash function is shown in Figure 3.9, which is based on the rules dealt with in Section 3.3.[5]

Perfect Bit Scan Hash Function

```
// generate a unique magic index based on the magic_multiplier
int magic_function(uint64 isolated_bit, uint64 magic, int size) {
  return (isolated_bit * magic) >> (64 - size);
}
```

Figure 3.9: Source-code listing for a magic hash function which maps an isolated bit to a unique magic index.

The bit scan forward function

```
int bit_scan_forward(uint64 bitboard)
```

[5]Thanks to Tord Romstad for this magic multiplier of a 64-bit bit scan.

Perfect Bit Scan Hash Initialisation

```
// initialise the look-up table for our magic bit scan
void initialise_bit_scan_forward(void) {
  uint64 input_key = 1;
  uint64 magic = HEX(0x218A392CD3D5DBF);
  // we need 6 bits to address a number 0-63
  int size = 6;
  int magic_index;
  int bit_position;
  for (bit_position = 0; bit_position < 64; bit_position++) {
    magic_index = magic_function(input_key, magic, size);
    magic_hash_table[magic_index] = bit_position;
    input_key = input_key << 1;
  }
}
```

Figure 3.10: Source-code listing for magic hash table initialisation for a magic bit scan.

in the source-code listing of Figure 3.11 can only be applied after the initialisation of the magic hash table:

```
int magic_hash_table[64]
```

The only parameter passed into this function

```
uint64 bitboard
```

is a bit matrix. The LSB is isolated from this bit matrix by means of the two's complement [51].[6] Finally, the unique magic index for access to the hash table is computed by calling the magic hash function from the source-code listing of Figure 3.9. Although the bit position is the only return value of this function, more extensive information with exactly the same algorithm can be stored.

Efficiency requirements for a bit scan, which should compete with a native implementation, are quite high. Access to the hash table via magic multiplication on a respective computer environment meets these high requirements and is, above all, very flexible [37]. Analogous implementations of hash tables can be used without caution within the scope of an algorithm, as long as information about the bit position is not required. Otherwise the indexing of a linear vector via the bit position is, of course, more reasonable.

3.6 Magic Hash Functions for Sliding Directions

The theory of the unique magic index was introduced in Section 3.3. Based on the calculation of the hash index for (combined) sliding directions according to Equations 3.10, 3.11, and 3.12, universal implementations with minimal array access (see Subsection 3.6.1) and homogeneous array access [30] (see Subsection 3.6.2) will be developed subsequently. Moreover, the technical differences of both array accesses will be compared.

[6] The isolation of the LSB could produce a compiler warning (Microsoft): "unary minus operator applied to unsigned type, result still unsigned".

Perfect Bit Scan Hash Implementation

```
// use bit scan forward function (hash table is initialised)
int bit_scan_forward(uint64 bitboard) {
  // the next line of code could produce a compiler warning:
  uint64 isolated_bit = bitboard & -bitboard;
  uint64 magic = HEX(0x218A392CD3D5DBF);
  int size = 6;
  int magic_index;
  magic_index = magic_function(isolated_bit, magic, size);
  int bit_position = magic_hash_table[magic_index];
  return bit_position;
}
```

Figure 3.11: Source-code listing for a 64-bit bit scan forward function.

3.6.1 The Minimal Array Access Implementation

The minimal array access will be dealt with in this subsection. The implementation contains both the function for the initialisation of the single sub-hash tables and the magic hash function for the minimal array access. The magic hash table for (combined) sliding directions is initialised in the source-code listing of Figure 3.12. The vector

```
int sliding_offset_index[64]
```

is initialised according to Equation 3.10 iteratively by summation of single offsets in an outer loop. The vector

```
int sliding_shift[64]
```

contains the difference between 64 and the number of bits $(64 - bits(square))$ according to the matrices of Figure 3.7. The inner loop initialises the sub-hash tables. The computation of the magic index is done according to Equation 3.11. The function

```
uint64 generate_blockers(int index, int bits, uint64 mask)
```

will be explained later in Subsection 3.7.3, within the context of the trial-and-error algorithm for computing magic multipliers. This function generates an input key for every possible index between 0 and offset $(0 \leq index < offset)$. The vector

```
uint64 sliding_bitboard[64]
```

contains the bit masks for blocker squares. The magic mapping for the computation of unique magic indices does not have to be written explicitly with brackets, since both the multiplication operator and the addition or subtraction operator have higher priority than shift operators.[7] The hash table for the initialisation of sub-hash tables according to Equation 3.12 is accessed via

```
sliding_move_hash_table[offset_index + magic_index].
```

[7] For more information about the priority of arithmetic operator see: `http://www.cplusplus.com/doc/tutorial/operators.html`, 2007.

Finally, the function

```
generate_sliding_attack(square, bitboard)
```

calculates the hash entry of the sub-hash table of the corresponding squares

```
uint64 bitboard
```

that fits the input key. The functions for the generation of sliding attacks are listed in Appendix A.2 of this thesis.

Initialisation of Magic (combined) Sliding Directions

```
void initialise_sliding_directions(void) {
  int offset_index = 0;
  // initialise the 64 sub-hash tables
  for (int square = 0; square < 64; square++) {
    // update the offset-index vector
    int offset = int(1 << 64 - sliding_shift[square]);
    sliding_offset_index[square] = offset_index;
    // loop through all possible occupancy bits
    for (i = 0; i < offset; i++) {
      uint64 bitboard = generate_blockers(i, 64-sliding_shift[
          square], sliding_bitboard[square]);
      magic_index = bitboard * sliding_magic[square] >>
          sliding_shift[square];
      sliding_move_hash_table[offset_index + magic_index] =
          generate_sliding_attack(square, bitboard);
    }
    offset_index+=offset;
  }
}
```

Figure 3.12: Source-code listing for magic hash table initialisation of magic (combined) sliding directions.

Access to the initialised hash table is presented in the source-code listing of Figure 3.13. The fitting sub-hash table is addressed by the input key of the square via the minimal array access. In addition to this offset, only the central bits from the input key

```
const uint64 blockers
```

are extracted as the edge squares are ignored in the magic multiplication [30].

3.6.2 The Homogeneous Array Access Implementation

The homogeneous array access implementation differs only insignificantly from the minimal array access implementation. While the hash table of the minimal array access implementation is to be implemented according to Subsection 3.3.2 as

```
uint64 minimal_hash_table[102400];  // 800 kb
```

the same hash table has to be defined as a two-dimensional vector [30, page 6] with:

Minimal Array Access Implementation

```
uint64 get_sliding_direction(const int square, const uint64
    blockers) {
  uint64 bitboard = blockers & sliding_mask[square];
  int offset_index = sliding_offset_index[square];
  int magic_index = bitboard * sliding_magic[square] >>
      sliding_shift[square];
  return sliding_move_hash_table[offset_index + magic_index];
}
```

Figure 3.13: Source-code listing for the magic hash table usage of magic (combined) sliding directions.

```
uint64 homogeneous_hash_table[64][4096];  // 2.0 mb
```

Further differences during initialisation and addressing of the hash table are obvious: as the offset-index vector is no longer applied (see Equation 3.9), both functions are simplified at the expense of the additional memory increase (see Subsection 3.3.2). Differences in time consumption of both implementations were not measurable in the scope of an entire computer-chess architecture (→ Loop Amsterdam) with accuracy of $\leq \pm 1\%$.

3.7 Generation of Magic Multipliers

In this section an algorithm for the generation of magic multipliers for the computation of the unique magic indices is presented. A magic multiplier is designated as optimally magic multiplier, when the computed unique magic indices consist of a minimum number of bits [30, page 7]. An optimal magic multiplier maps a bitboard (→ input key) into the smallest possible value range (→ output), whereby the size of the corresponding hash table is minimal as well.

The following algorithm for the generation of magic multipliers consists of several functions. The functions for the generation of masks (file masks, rank masks, and diagonal masks) and attacks (file attacks, rank attacks, and diagonal attacks) are not presented explicitly since their deterministic mode of operation is not part of the trial-and-error approach. These functions are listed in Appendix A.2 of this thesis.

The algorithm for the generation of optimal magic multipliers has a great number of parameters that affect the generation of magic multiplier and optimal magic multiplier. It is possible to determine exactly the number of ones, which a magic multiplier consists of. Likewise, the number of these one-bits can be arranged in a firmly stipulated interval.

This section contains four subsections. In Subsection 3.7.1 an algorithm for the generation of pseudo-random 64-bit numbers will be presented. These pseudo-random 64-bit numbers are necessary for the generation of magic multipliers. Only when a pseudo-random number maps all entries of a vector unequivocally, the pseudo-random number is regarded as a magic multiplier. In Subsection

3.7.2 bitboards will be mapped into unique indices by applying the magic function. The generation of possible blockers will be performed in Subsection 3.7.3. Finally, in Subsection 3.7.4 our trial-and-error function can be presented and discussed.

3.7.1 Generation of Pseudo-random 64-bit Numbers

The generation of pseudo-random 64-bit numbers in the function

```
uint64 random_uint64_one_bits(int one_bits)
```

in the source-code listing of Figure 3.14 is controlled via passing the only parameter:

```
int one_bits
```

The return value of this function is saved in the 64-bit unsigned integer

```
uint64 result
```

and contains exactly the indicated number of one-bits ($bit_i = 1$)

```
int one_bits
```

in pseudo-random order. The optimal number of one-bits can be determined in this way, in order to compute quickly a set of optimal magic multipliers for a given problem, such as magic rook hash. A set consists of 64 magic multipliers, since a single magic multiplier is required for every single square of the chessboard.

The function

```
int rand(void)
```

in the inner `while`-loop returns a pseudo-random integer in the range $0 \leq n \leq 32767$.[8]

3.7.2 Mapping Bitboards into Unique Magic Indices

During the generation of magic multipliers exactly the same magic function is required, which is used to access the magic hash table later in the computer-chess architecture. The input of the magic mapping function in the source-code listing of Figure 3.15 is a bitboard

```
uint64 bitboard
```

and the corresponding magic multiplier:

```
uint64 magic
```

The optional parameter

```
int bits
```

[8] Compare to `http://www.cplusplus.com/reference/clibrary/cstdlib/rand.html`.

Generator for Pseudo-random 64-bit Numbers with n One-Bits

```
uint64 random_uint64_one_bits(int one_bits) {
  uint64 power_of_two;
  uint64 result = 0;
  // repeat this process until all pseudo-random one-bits are
      generated
  for (int i = 0; i < one_bits; i++) {
    while (true) {
      power_of_two = uint64(1) << uint64(rand() & 63);
      if ((result & power_of_two) == 0) {
        result |= power_of_two;
        break;
      }
    }
  }
  return result;
}
```

Figure 3.14: Source-code listing for a generator for pseudo-random 64-bit multipliers with exact n one-bits.

is included for the purpose of generalisation of the magic mapping function. By passing the number of bits, it is determined into which value range the input key is mapped. The return value of this magic mapping function is a unique magic index for addressing the corresponding hash table.[9]

Magic Mapping Function

```
int magic_function(uint64 bitboard, uint64 magic, int bits) {
  return int((bitboard * magic) >> (64 - bits));
}
```

Figure 3.15: Source-code listing for a magic hash function to address the magic hash tables.

3.7.3 Generation of Possible Blockers

The function

```
uint64 generate_blockers(int index, int bits, uint64 mask)
```

in the source-code listing of Figure 3.16 generates all possible combinations of blockers for the sliding mask of a certain square. The sliding mask

```
uint64 mask
```

is a combination of sliding directions. This function generates, for example a bitboard for a combined file-rank mask (→ rook mask) of a respective square depending on the number of bits:

[9] The implementation of our magic mapping function of the source-code listing of Figure 3.15 is identical with the magic mapping function of the source-code listing of Figure 3.9.

```
int bits
```

Generator for Sliding Bit Combinations for a Square

```
uint64 generate_blockers(int index, int bits, uint64 mask) {
  int square;
  uint64 bitboard = 0;
  // loop through all possible bit combinations
  for (int i = 0; i < bits; i++) {
    square = find_first_bit_and_clear(mask);
    if (index & (1 << i)) bitboard |= square_to_bitboard(square);
  }
  return bitboard;
}
```

Figure 3.16: Source-code listing for generation of all possible bit combinations for a special square with several sliding directions.

3.7.4 Generation of Magic Multipliers

After the most important functions for the trial-and-error based algorithm for the generation of magic multipliers have been developed, the trial-and-error function can be presented now.

The function

```
uint64 find_magic_multiplier(int square, int one_bits)
```

in the source-code listing of Figure 3.17 receives as an input only one square (`a1-h8` → 0-63) and the exact number of one-bits, which the magic multiplier is supposed to consist of. The corresponding magic multiplier is the return value of this function. This function calls the two functions in the first part of the initialisation.

```
uint64 generate_mask(int square)
uint64 generate_attack(int square)
```

The source codes of these functions are listed in Appendix A.2 because they are of basic deterministic origin and do not concern the trial-and-error approach. The masks generated via the function call

```
uint64 generate_mask(int square)
```

consist of bit strings for single or combined sliding directions. The blocker vector

```
uint64 blockers[4096]
```

is initialised with all possible bit combinations of respective combined sliding directions. Furthermore, the entries of the blocker vector are passed on to the function for the generation of combined sliding attacks:

```
uint64 generate_attack(int square, uint64 blockers).
```

The vector for the management of bitboards with the combined sliding attacks of a respective square

```
uint64 solution[4096]
```

is also initialised in this initialisation loop. After the initialisation, an *infinite* trial-and-error iteration is implemented. In this case, the use of a temporary vector

```
uint64 used[4096]
```

is important. With the help of this temporary vector, possible collisions in the magic hash table are checked during each trail-and-error iteration. The generated pseudo-random 64-bit number

```
uint64 magic
```

consisting of a fixed number of one-bits

```
int one_bits
```

must now be able to produce a unique magic index for every entry of the blocker vector. If the calculated index

```
int index
```

leads to an *unintended collision* in the hash table, the trial-and-error iteration will be terminated then, and a new pseudo-random number will be generated.[10] Only when this pseudo-random number maps all entries of the blocker vector without unintended collision, the pseudo-random number is regarded as the magic multiplier. With this magic multiplier is it possible to address all entries of the solution vector.

3.8 Experiments with Magic Multiplier Sets[11]

The generation of magic multipliers for single squares of a chessboard is implemented via the trial-and-error algorithm dealt with in Section 3.7. As soon as all magic multipliers for the entire chessboard ($\rightarrow$ 64 squares) and a combination of sliding directions are computed, we obtain a complete magic multiplier set. Only two magic multiplier sets for the Bishops and the Rooks are required for the implementation of a magic-bitboard-based computer-chess architecture. The implementation of a magic multiplier set for the Queen occurs via arithmetic or logical operation ($\rightarrow$ arithmetic plus, logical or) of the bishop bitboard and the rook bitboard. The implementation of single sliding directions (`a1h1, a1a8, a1h8, h1a8`) occurs via unmasking ($\rightarrow$ logical and) of the respective bishop bitboard or rook bitboard. Some tests within the computer-chess architecture of Loop Amsterdam have shown that the explicit implementation of `a1h1`, ..., `h1a8` sliding directions does not lead to any measurable speed advantages, and thus does not have to be implemented explicitly. A small advantage in storage space and less redundant information in the magic hash tables are two positive side effects of this architecture.

[10] When at least two entries of a blocker vector are mapped into the same hash address and the solution bitboards are different, we speak about an unintended collision.

[11] The results of the following experiment were presented at the Ph.D. Day at the Tilburg University, 2008.

Trial-and-Error Algorithm for Generation of Magic Multipliers

```
uint64 find_magic_multiplier(int square, int one_bits) {
  uint64 blockers[4096];
  uint64 solution[4096];
  uint64 used[4096];
  uint64 mask;
  uint64 magic;
  bool failed;
  int index;
  int bits;
  int i;
  // initialise file, rank or diagonal masks and their bits
  mask = generate_mask(square);
  bits = count_bits(mask);
  // initialise a blockers vector and a solution vector
  for (i = 0; i < (1 << bits); i++) {
    blockers[i] = generate_blockers(i, bits, mask);
    solution[i] = generate_attack(square, blockers[i]);
  }
  // start the trial-and-error iteration
  while (true) {
    magic = random_uint64_one_bits(one_bits);
    for (i = 0; i < (1 << bits); i++) used[i] = 0;
    failed = false;
    for (i = 0; i < (1 << bits); i++) {
      index = magic_function(blockers[i], magic, bits);
      if (used[index] == 0) used[index] = solution[i];
      else if (used[index] != solution[i]) {
        failed = true;
        break;
      }
    }
    // did we find a magic multiplier for this square?
    if (failed == false) return magic;
  }
}
```

Figure 3.17: Source-code listing for a trial-and-error algorithm for generation of magic multipliers for sliding directions.

This section contains two subsections. In Subsection 3.8.1 magic multiplier sets with $n \geq 6$ bits for bishops and rooks are generated via the trial-and-error approach from Section 3.7. In Subsection 3.8.2 magic multiplier sets with $n \leq 6$ bits for bishops and rooks are generated by using a brute-force determinism.

3.8.1 Magic Multiplier Sets with $n \geq 6$ Bits

In Table 3.1 the runtime performance of the generation of two magic multiplier sets for Bishops and Rooks is listed. The number of the bits

```
int one_bits
```

is entered in the first column; by this instruction multipliers were generated via the function call:

```
uint64 random_uint64_one_bits(int one_bits)
```

For the generation of a magic multiplier set for Bishops 100 pseudo-random trial-and-error runs were executed in order to obtain a result of measurement as precise as possible. In the same way, 100 pseudo-random trial-and-error runs for the generation of the magic multiplier set for Rooks were carried out. The pseudo-random generator

```
int rand(void)
```

was initialised via the function call, prior to the generation of a new magic multiplier set,

```
void srand(unsigned int seed)
```

in order to exclude to a large extent falsifying random effects for this experiment.[12] The trial-and-error runs for every magic multiplier set were thus carried out each time with another sequence of pseudo-random 64-bit numbers. Frequently, it was observed that during the generation of more sophisticated magic rook multipliers the generation time of magic multipliers with a higher number of bits (e.g., $n \geq 13$) for corner squares ($\rightarrow$ `a1`, `h1`, `a8`, `h8`) was higher. This might lead to slight deviations in the results of measurement.

In columns 2 and 5 the CPU times in milliseconds (ms) for the generation of each of 100 magic multiplier sets are listed. In columns 3 and 6 the average CPU times for the generation of the magic multiplier sets are displayed. Column 4 and 7 show the relative CPU times that are required for the generation of magic multiplier sets with exactly n one-bits. In this case, the average CPU time for the generation of the magic multiplier set with the least number of bits was defined as 100%. At least 6 bits are needed in the trial-and-error experiment for an optimal magic multiplier set of a Bishop. At least 7 bits are needed in the trial-and-error experiment for an optimal magic multiplier set of a Rook. The generation of a magic multiplier set with $n \geq 15$ bits was not possible with this trial-and-error approach without additional filter conditions.

[12] For more information see: `http://www.cplusplus.com/reference/clibrary/cstdlib/srand.html`.

Generation of Magic Multiplier Sets with n Bits						
	Bishop-Sliding			Rook-Sliding		
	$n = 100$	$n = 1$		$n = 100$	$n = 1$	
n bits	time (ms)		relative (%)	time (ms)		relative (%)
6	30,650	306	100	n/a	n/a	n/a
7	14,480	144	47	264,060	2,640	100
8	9,370	93	31	162,900	1,629	62
9	8,430	84	28	157,020	1,570	59
10	9,320	93	30	163,990	1,639	62
11	9,200	92	30	190,300	1,903	72
12	11,140	111	36	227,310	2,273	86
13	10,120	101	33	397,060	3,970	150
14	16,230	162	53	538,120	5,381	204
15	27,120	271	88	660,940	6,609	250

Table 3.1: Generation of n-bit magic multiplier sets for bishop sliding and rook sliding via trail-and-error approach.

3.8.2 Magic Multiplier Sets with $n \leq 6$ Bits

In the trial-and-error function

```
uint64 find_magic_multiplier(int square, int one_bits)
```

in the source-code listing of Figure 3.17 it is possible to filter the generated pseudo-random numbers

```
magic = random_uint64_one_bits(one_bits)
```

before further processing. For this reason, the implementation of necessary filter conditions on a magic multiplier is reasonable. For example, a necessary condition can check qualities of the magic product.

```
magic_product = mask * random_number
```

The following filter condition

```
if (count_bits((magic_product) & 0xFFFF000000000000) < 8)
```

filters during the generation of the magic rook multiplier 41 to 50% of all pseudo-random numbers. During the trial-and-error search of the generation of the magic bishop multipliers even 39 to 67% of the pseudo-random numbers can be extracted in this way.[13] With this additional condition it is possible to generate magic bishop multipliers with up to 19 bits. During the generation of the magic rook multipliers no progress with regard to the number of bits could be achieved. The condition counts all active bits of the uppermost 16 bit positions of the magic product via masking. This condition works quite efficiently since these upper 16 bits are required for the computation of the unique magic index with the right shift.

[13] Thanks to Tord Romstad for the hint about filter conditions.

At this point, the question as to what happens while generating magic multipliers with $n \leq 6$ bits for Rooks and $n \leq 5$ bits for Bishops remains to be answered. Magic multipliers with a minimum number of bits cannot be efficiently generated with the trial-and-error approach from Section 3.7. With a basic brute-force determinism, that tests all bit combinations for 64-bit magic multipliers with $1 \leq n \leq 6$ bits, we can generate all possible magic multipliers in this way (see Appendix B). It is interesting to generate magic multiplier sets for Bishops and Rooks with this approach that produces the magic multiplier with the minimum number of bits for every square.

In Figure 3.18 the results of this experiment are summarised for the sliding Bishop optimal magic multiplier set. Each single element of this matrix describes the minimum number of bits which an optimal magic multiplier must have for one square. Therefore, a minimal magic multiplier set for Bishops consists of $3 \leq n \leq 6$ bits. The point symmetry of the number of bits in the centre of the chessboard is quite interesting. At least 6 bits are necessary for the magic multipliers of corner squares ($\rightarrow$ `a1, h1, a8, h8`) in order to map a unique magic index.

Minimal n Bits for Sliding Bishop Optimal Magic Multipliers

$3-4-bits$								$5-6-bits$							
–	–	4	3	3	4	–	–	6	5	–	–	–	–	5	6
–	–	4	3	3	4	–	–	5	5	–	–	–	–	5	5
–	–	–	4	4	–	–	–	5	5	5	–	–	5	5	5
–	–	–	4	–	–	–	–	5	5	5	–	5	5	5	5
–	–	–	–	4	–	–	–	5	5	5	5	–	5	5	5
–	–	–	4	4	–	–	–	5	5	5	–	–	5	5	5
–	–	4	3	3	4	–	–	5	5	–	–	–	–	5	5
–	–	4	3	3	4	–	–	6	5	–	–	–	–	5	6

Figure 3.18: Minimal number of n bits for the optimal magic multipliers for a sliding Bishop on a chessboard.

In Figure 3.19 the results of this experiment for the sliding rook optimal magic multiplier set are summarised. Unlike the matrix of a bishop set, no obvious symmetries exist here.

3.9 Answer to Research Question 2

In this chapter it was introduced how perfect hash functions can be used within the scope of a state-of-the-art computer-chess architecture. The requirements on the runtime efficiency of a respective hash function are high as almost the entire computer-chess architecture of a computer-chess engine is based on basic computation, such as scanning bits (see Section 3.5) or the generation of (combined) sliding attacks (see Section 3.6). These considerations bring us back to the second research question. Below we will give our conclusions and recom-

Minimal n Bits for Sliding Rook Optimal Magic Multipliers

$4-bits$								$5-6-bits$							
–	–	–	–	–	–	–	–	6	6	6	6	6	5	5	6
4	4	4	4	4	4	4	4	–	–	–	–	–	–	–	–
4	4	4	4	4	4	4	–	–	–	–	–	–	–	–	5
4	4	4	4	4	4	4	4	–	–	–	–	–	–	–	–
4	4	4	4	4	4	4	4	–	–	–	–	–	–	–	–
4	4	4	4	4	4	4	–	–	–	–	–	–	–	–	5
4	4	4	4	4	4	4	4	–	–	–	–	–	–	–	–
–	4	–	–	–	–	4	–	5	–	5	5	5	5	–	5

Figure 3.19: Minimal number of n bits for the optimal magic multipliers for a sliding Rook on a chessboard.

mendations for future developers.

In the last few years we have witnessed a trend towards the development of knowledge-based computer chess architectures, although this can hardly be proved empirically without an insight into source codes of the leading computer-chess engines. The trend towards implementing more knowledge at the cost of speed [15] has become quite popular among computer-chess engine developers. With regard to this trend and the continuously increasing complexity of the algorithms, the choice of a suitable computer-chess architecture at the beginning of a new development can be decisive for the later success and the further development. This trend is certainly, among other universally possible applications of Magic Bitboards [21, page 9], a main reason for the intensive developments of this technology by Kannan *et al.* [36, 44].

Conclusions Research Question 2

In order to be able to better understand the magic multiplication, it is necessary to gain a deeper understanding of multiplications of unsigned integers with overflow in mathematical rings (see Subsections 3.2.2 and 3.2.3). While the generation of magic multipliers for the multiplication with a power of two was possible even manually in Section 3.4, the generation of multipliers for the index mapping of integers in Section 3.8 was more extensive. The research of bit patterns at the end of Section 3.8 is only a first step in order to improve our understanding of this complex process. Yet, we may conclude that our magic hash functions for sliding directions with minimal and homogeneous array access (see Section 3.6) make full use of bitboards.

Future Research

In this chapter the sizes of the sub-hash tables have been defined according to the results presented in Subsection 3.3.2. The size of the hash table is exponentially dependent on the number of possible blocker squares (excluding the edge

squares). In fact, we may state that there are $2^{blocker\_squares}$ permutations for the input key. However, there are only maximal $4 \times 3 \times 3 \times 4 = 144$ different bitboards for a sliding Rook in one sub-hash table (see Table 3.2). This means that the same bitboards are addressed by different input keys in different hash entries.

In Table 3.2 the minimal necessary sizes of single sub-hash tables for a sliding Rook are summarised. In the first column and the last row the factors are entered that produce the 8×8 chessboard matrix when multiplied by each other. The orientation is to be contemplated from White's point of view (of course it is symmetric for Black's point of view). For the better understanding, the multiplication is split up in such a way that the sum of the factors always amounts to 14 ($\rightarrow$ 14 sliding squares).

Minimal n Bits for Sliding Rook Optimal Magic Multipliers								
$\times 7$	49	42	70	84	84	70	42	49
$\times 6 \times 1$	42	36	60	72	72	60	36	42
$\times 5 \times 2$	70	60	100	120	120	100	60	70
$\times 4 \times 3$	84	72	120	144	144	120	72	84
$\times 3 \times 4$	84	72	120	144	144	120	72	84
$\times 2 \times 5$	70	60	100	120	120	100	60	70
$\times 1 \times 6$	42	36	60	72	72	60	36	42
$\times 7$	49	42	70	84	84	70	42	49
	$\times 7$	$\times 6 \times 1$	$\times 5 \times 2$	$\times 4 \times 3$	$\times 3 \times 4$	$\times 2 \times 5$	$\times 1 \times 6$	$\times 7$

Table 3.2: Minimal number of n bits for optimal magic multipliers for a sliding Rook on a chessboard.

Table 3.2 could be an interesting approach for future research. Thus, a sliding Rook has $2^{12} = 4096$ different input keys on a corner square ($\rightarrow square \in C$ according to the equations in Figure 3.7). However, according to Table 3.2 there are only $7 \times 7 = 49$ different bitboard entries. Other squares show a more unfavourable relation, that is why corner squares and edge squares are the most interesting ones to examine. The reduction of the size of sub-hash tables is already a debated issue in internet developer forums [36, 44]. It is likely that it is impossible to penetrate in such minimal-size orders of the sub-hash tables.[14] However, via the generation of more appropriate magic multipliers, it could be possible to reduce the size of some sub-hash tables to around 1 to 2 bits.

[14] Hereby the next possible power of two is meant, which means that a minimal hash table of 256 entries must be chosen for 144 entries.

Chapter 4

Static Exchange Evaluation

The main objective of Chapter 4 is to answer the third research question on the basis of the R&D of a straightforward and quite efficient *static exchange evaluator* (SEE) for the computer-chess engine Loop Amsterdam 2007. Below we repeat the third research question.

> **Research question 3:** *How can we develop an $\alpha\beta$-approach in order to implement pruning conditions in the domain of static exchange evaluation?*

In this chapter the $\alpha\beta$-approach, which is the basis of a computer-chess search [4], will be applied to an iterative SEE. With the SEE, moves and in particular capture moves on a threatened destination square can be evaluated materially in an exact way. The SEE algorithm works according to deterministic rules and considers only threats and attacks during the evaluation of a move that aim at the destination square. Because the computation of threats and attacks is time consuming and the code is quite complicated, the development of the SEE is interesting for sophisticated implementations of chess knowledge in a state-of-the-art computer-chess architecture. The results of the SEE are quite precise. The application of the SEE algorithm is straightforward and the field of application is enormous (e.g. move ordering, evaluation, forward pruning, etc.) (see Section 4.5). Only with an iterative $\alpha\beta$-approach pruning conditions can be implemented into the SEE, whereby the procedure becomes clearly more efficient.

Current computer-chess engines use complicated algorithms for coding chess knowledge. The SEE is a good example of those algorithms which are currently in use. It is therefore an interesting algorithm to study, and since it is widely used, its improvements are relevant to the computer-chess community.

The applied data structures must be able to supply quickly all necessary information about threats, attack possibilities, and hidden attackers. This information is evaluated recursively or iteratively in further algorithmic steps. Due to the diverse possible applications of the SEE, it is interesting to scrutinize the complex algorithm and extensive possibilities of the implementation within the scope of a state-of-the-art computer-chess architecture.

In this chapter the major focus will not be on the details of the implementation of the SEE. Our scientific research discusses only those data structures and implementations which are related to the recursive and iterative computation within the $\alpha\beta$-SEE. Therefore, in this chapter no attention is paid to specific implementations of the SEE, such as a bit scan-free x-ray SEE [31].

In Section 4.1 the general SEE algorithm is introduced. Besides important definitions, two competitive alternatives of the implementation ($\rightarrow$ recursion or iteration) are developed and compared. In Section 4.2 the iterative $\alpha\beta$-approach and the pruning conditions resulting from that are investigated and further developed. In Section 4.3 the iterative implementation of the SEE with all pruning conditions is examined. It is based on the $\alpha\beta$-approach from Section 4.2.

The emphasis of Section 4.4 is on a comparison of the *quantitative α-pruning* with the new approach of the *qualitative β-pruning*.[1] In Section 4.5 some typical application fields of the SEE are presented, as they are implemented, for instance, in Loop Amsterdam, Loop Express, Glaurung (Romstad, 2008) or Fruit (Letouzey, 2005). Some empirical measurements are given in Section 4.6 with regard to the time consumption of the SEE in the state-of-the-art computer-chess engine Loop Amsterdam. In Section 4.7 an analysis of combined pruning conditions is provided. In Section 4.8 the results of our research are summarised. Different possibilities of the implementation based on the introduced technologies are evaluated and discussed. Finally, the third research question is answered.

4.1 The Static Exchange Evaluation Algorithm

The SEE computes the value of a square on a chessboard based on the balances of power that affect this square. The involved pieces determine these balances of power and possess particular qualities ($\rightarrow$ capture direction, x-ray attacks, promotion, etc.) that must be processed algorithmically in different ways. If several pieces of both parties (White and Black) simultaneously threaten the same square, different factors will have to be considered in the SEE. The result of the SEE is a material evaluation from the point of view of the side to move. The remaining squares of the chessboard, which are not affected by the SEE, are ignored. Therefore, the trivial $\alpha\beta$-tree of Figure 4.1 has no branching and no time-complex behaviour. At a left child the tree stops and at a right child the next capture move is done.

For proper readability we give a straightforward example (see Figure 4.2) that is used later on, too. In order to work with consistent terms in the further course of this chapter, different piece types and their performance influencing factors during a static exchange evaluation will be subsequently explained (see Subsection 4.1.1). Moreover, recursive algorithms and iterative algorithms will be introduced with the help of samples in Subsections 4.1.2 and 4.1.3, in order to evaluate algorithmic advantages and disadvantages.

[1] Kannan presented a bounded "alphabeta-like" SEE in the Winboard Forum, 2007 [31].

The Trivial $\alpha\beta$-Tree

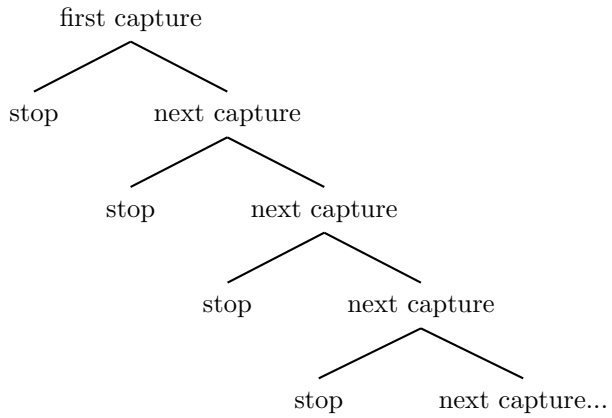


Figure 4.1: The trivial $\alpha\beta$-tree without time-complex behaviour for the SEE.

4.1.1 Definitions

The position of the sample chessboard in Figure 4.2 was extracted from a *Sicilian Opening* (ECO = B51).[2] This position occurred in round 4 between Loop Amsterdam and Rybka during the 15^{th} World Computer-Chess Championship, Amsterdam (NL) 2007 (see Appendix C.2) and is used to exemplify the following definitions.

Definition 1: The Blocker

A *blocker* is a piece which blocks another friendly or opposing sliding piece (slider) from arriving at a destination square. In Figure 4.2 the white Rook on square `d3` is the blocker of the white Rook on square `d1` in `a1a8`-direction and of the black Rook on square `d8` in `a8a1`-direction.

Definition 2: The Direct Attacker

A *direct attacker* can move directly onto a destination square. In Figure 4.2 the white Rook on square `d3` is a sliding direct attacker of the black Rook on square `d8`. If the white Rook moves onto square `e3`, the white Rook on square `d1` and the black Rook on square `d8` also become direct attackers concerning the respective opposing Rook.[3]

Definition 3: The Indirect Attacker (X-Ray Attacker)

An *indirect attacker* can also be designated as an *x-ray attacker* according to Hyatt, and is always a sliding piece, that is blocked by at least one blocker from

[2]ECO: Encyclopedia of Chess Openings.
[3]We do not explicitly take into account if an attacking piece is pinned.

moving onto a destination square.[4] Only when this blocker capitulates from the source-destination direction (but is not captured, otherwise a new blocker emerges), the indirect attacker becomes the direct attacker. Affected by the blocker on square `d3` in Figure 4.2, the white Rook on square `d1` is an indirect attacker of the black Rook on square `d8`. In reverse, the black Rook on square `d8` is the indirect attacker of the white Rook on square `d1`.

Definition 4: Material Value

If a direct attacker has a smaller material value than the attacked piece, then by the capture move a material profit is secure in any case, since the aggressor can lose only his attacker. In this case, one speaks according to Heinz [24] of a *aggressor-victim* relationship. The more favourable this relationship is for the aggressor ($aggressor \leq victim$), the more valuable the capture move is. For the possible case ($aggressor > victim$), it is necessary to analyse precisely the capture sequence via a static exchange evaluation. The piece values in this chapter for (1) Pawn, (2) Knight, (3) Bishop, (4) Rook, and (5) Queen are indicated in *centipawns* (CP) and have been adopted from Kaufman [32].

Multiple Threats on d-file: LOOP **vs.** RYBKA**, Amsterdam 2007**

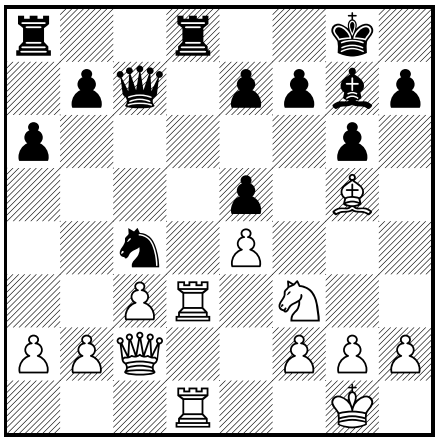


Figure 4.2: Direct attacker and blocker on square d3. Indirect attackers on square d1 and square d8.

4.1.2 Recursive Static Exchange Evaluation

"In order to understand recursion, one must first understand recursion."

The development of recursive algorithms is mostly more intuitive, simpler, and more time saving than the development of iterative algorithms for the same

[4] Indirect attackers are called *x-ray attacker* according to function names and comments within the open source computer-chess engine CRAFTY from Hyatt.

purpose.[5] Within a computer-chess architecture the use of recursive function calls is especially widespread in the field of sequential search algorithms. From the currently available open source computer-chess engines, only the engine SCORPIO by Shawul, Ethiopia, is based on an iterative approach and uses no recursive function calls for the *principal variation search* (PVS). Furthermore, the well-known computer-chess machine HYDRA by Donninger is based on an iterative search technique.[6]

Recursive Algorithms

Apart from the search algorithms for the PVS and the quiescence search, recursive brute-force performance tests are also often carried out in computer-chess engines during the development phase and the test phase. By these performance tests errors can quickly be traced in the basic computer-chess architecture during the development phase. With the help of public computer-chess engines, such as CRAFTY, this kind of performance tests can be carried out, which determine reliable comparative values. By means of these comparative values ($\rightarrow$ the computed nodes) faulty deviations can be found. In a later test phase and a later stage of the development of a computer-chess architecture these performance tests are a good way to measure and improve, for example, the computing speed of move generators and attack detections. The performance measurements of move generators within a search are quite inaccurate, since small changes in move generators mostly lead to changed game trees. The performance measurements should then hardly be comparable.

Source Code

The source-code listing of the recursive SEE in Figure 4.3 is to be read from White's point of view. The initialisation algorithm

```
int static_exchange_evaluation_w(const int from, const int to)
```

does not generate a call to a recursive procedure, as long as Black has not found any defender after the generation of all direct attackers with:

```
void generate_attackers(const int from, const int to)
```

The analysis with the return of the material value of the captured piece

```
int board_value(const int to)
```

can be terminated. As long as White moves its King and the condition

```
black_attackers >= 1
```

is fulfilled, the planned capture move would be invalid. The recursive SEE can be terminated early. The recursion from Black's point of view starts with the return of the square of the last capture move:

```
int recursion_b(const int from, const int to)
```

[5] lat. *recurrere* "run backward".

[6] For further information see the official HYDRA website: `http://www.hydrachess.com`

During the recursion the indirect attackers, which become activated by the function

```
void update_x_ray_attackers(const int next, const int to)
```

must be added to the available direct attackers. The recursion will be terminated, if (1) one side does not have any pieces which can move onto the destination square or (2) the King of a respective side belongs to the capture sequence and the opposing side has at least one further direct attacker.

Recursive Static Exchange Evaluation

```
int static_exchange_evaluation_w(const int from, const int to) {
  int value;
  // recursion for White
  generate_attackers(from, to);
  if (black_attackers() == 0) return board_value(to);
  if (board_piece(from) == WHITE_KING) return -INFINITE;
  value = recursion_b(from, to);
  return board_value(to) - value;
}

int recursion_b(const int from, const int to) {
  int value;
  int next;
  // recursion for Black
  next = next_black_attacker();
  update_x_ray_attackers(next, to);
  if (white_attackers() == 0) return board_value(from);
  if (board_piece(next) == BLACK_KING) return 0;
  value = recursion_w(next, to);
  if (board_value(from) - value < 0) return 0;
  return board_value(from) - value;
}

int recursion_w(const int from, const int to) {
  // recursion for White
  // same as for Black, but the function calls are vice versa
}
```

Figure 4.3: Source-code listing for a recursive SEE.

4.1.3 Iterative Static Exchange Evaluation

Iterative algorithms are often used for *iterative deepening* [22, 53] at the root of a sequential or parallel PVS within the computer-chess engine. Iterative algorithms store their local data in data structures, that are specifically organised for that purpose, and not in the stack as it is common in recursive procedures. These specific data structures of iterative algorithms are, unlike the stack, accessible from the outside. Although the management of these data structures must be developed by the programmer himself, it is, nevertheless, a powerful tool for synchronous and parallel manipulation of the data [20, 26].[7]

[7] Throughout the thesis words such as 'he', 'she', 'his', 'her', 'himself', and 'herself' should be interpreted gender-neutral unless when this is obviously incorrect from the context.

Iterative Algorithms

The time-complex behaviour and the branching factor of a computer-chess search engine with iterative deepening at the root of a sequential PVS is highly dependent on hash memory and heuristic procedures. Every iteration uses the information that is retrieved from the previous iterations in order to produce a search tree that is well sorted. Within the scope of a computer-chess search engine, there is, in addition to the iterative deepening in the root search, also the *internal iterative deepening* in the PVS, which is one of the best known iterative search techniques [46].

The splitting of the search tree in so-called split nodes is essential for parallel computer-chess search engines according to Hyatt's *Dynamic Tree Splitting* (DTS) procedures [26]. Likewise, the *Young Brothers Wait* (YBWC) algorithm is most efficient if implemented iteratively [61]. Dynamic jumping is only possible with synchronised and parallel manipulation of the data within the scope of an iterative approach. Such sophisticated parallel procedures must unlike ABDADA [61] or APHID [8] be developed iteratively, in order to enable a dynamic jumping in the search tree. This is quite complicated to achieve when recursion is involved in the search process.

Source Code

The source-code listing of the iterative SEE in Figure 4.4 is shown from White's point of view. The value of a pseudo move consisting of the move information

```
const int from
```

and

```
const int to
```

is to be computed in two steps with the linear vector:

```
int value_list[32]
```

Step 1: Forward-Iteration Loop

The generation of all direct attackers and the further initialisation works in analogy to the recursive algorithm in the source-code listing of Figure 4.3 which has been already described in Subsection 4.1.2. The value list

```
value_list[]
```

is initialised by the value of the captured piece

```
board_value(to)
```

and is described iteratively and alternately from Black's and White's points of view with the difference:

```
value_attacker - value_list[iteration-1]
```

When all iterations are terminated, the result of the capture sequence is entered on the last position of the value list:

```
value_list[iteration-1]
```

Step 2: Reverse-Iteration Loop

The result of the capture sequence after the last iteration must be evaluated with a further reverse iteration from the point of view of the respective side. The result of the capture sequence after iteration (i) must be compared with the result of the previous iteration $(i-1)$. The result of iteration $(i-1)$ will be replaced, if the new value from iteration (i) is better for the (in each case) active side [4, page 7-17]. The final result of the static exchange evaluation is computed iteratively and reversely by (1) comparison, (2) negation, and (3) copying of the material results up to the uppermost position of the value list:

```
value_list[0]
```

4.2 The Iterative $\alpha\beta$-Approach

The iterative SEE is, in contrast to the recursive implementation, more efficient, since no unnecessary recursive function calls and parameter passing must be done. The SEE algorithm computes the value of the capture sequence in accordance with the deterministic rules and thus does not require any heuristic procedures. The computations work with the constant $branching\ factor = 1$, since no branches within the "search tree" emerge (see Figure 4.1). The computations of an SEE are carried out according to the minimax theorem alternately from White's and Black's point of view. Due to the skilful use of an $\alpha\beta$-window for the forward-iteration loop, it is possible to dispense with the value list and the reverse-iteration loop for the evaluation of the value list from Subsection 4.1.3. The SEE algorithm is based on hard rules, as for example a move generator or an attack detector. It is of deterministic origin and returns, in contrast to a search algorithm or an evaluation function, exact results and no heuristic results. Therefore, this algorithm is at the core of a state-of-the-art computer-chess architecture.

This section contains four subsections. In Subsection 4.2.1 a short introduction of pruning conditions for the SEE is given. In Subsections 4.2.2 to 4.2.4 we will develop three pruning conditions which will be implemented into our iterative $\alpha\beta$-based SEE.

4.2.1 Pruning Conditions

The $\alpha\beta$-window shrinks after the initialisation with $\pm\infty$ by alternately White and Black, until the result of the SEE is computed. Apart from the precise application of the minimax principle, deterministic conditions depending on the lower bound α and the upper bound β are to be implemented, in order to avoid unnecessary iterations. Subsequently, three conditions will be introduced, which all in all lead to an economy of approximately 20% of all iterations and would not be implemented efficiently and comprehensibly without the iterative $\alpha\beta$-approach. With the qualitative β-pruning condition in Subsection 4.2.4 a

Iterative Static Exchange Evaluation with a Value List

```
int static_exchange_evaluation_w(const int from, const int to) {
  int iteration;
  int value_list[32];
  int next;
  int value_attacker;
  // iteration for White
  iteration=0;
  value_list[iteration] = board_value(to);
  next = from;
  generate_attackers(next, to);
  if (black_attackers() == 0) return board_value(to);
  value_attacker = board_value(from);
  // forward-iteration loop: (1) fill the value list
  while (true) {
    // iteration for Black
    iteration++;
    value_list[iteration]=value_attacker-value_list[iteration-1];
    next = next_black_attacker();
    update_x_ray_attackers(next, to);
    if (white_attackers() == 0) break;
    value_attacker = board_value(next);
    // iteration for White
    iteration++;
    value_list[iteration]=value_attacker-value_list[iteration-1];
    next = next_white_attacker();
    update_x_ray_attackers(next, to);
    if (black_attackers() == 0) break;
    value_attacker = board_value(next);
  }
  // reverse-iteration loop: (2) evaluate the value list
  while (true) {
    if (iteration == 0) return value_list[0];
    if (value_list[iteration] > -value_list[iteration-1]) {
      value_list[iteration-1] = -value_list[iteration];
    }
    iteration--;
  }
}
```

Figure 4.4: Source-code listing for an iterative SEE with a value list.

further new forward-pruning condition is derived, that terminates iterations as soon as $0 \notin [\alpha, \ldots, \beta]$ applies for the $\alpha\beta$-window of the SEE. The SEE returns quite precise results when applied in the field of move ordering compared with techniques such as *most valuable victim* (MVV) and *least valuable attacker* (LVA) according to Heinz [24].

4.2.2 The King-α-Pruning Condition

As already follows from the approach of the recursive SEE in Subsection 4.1.2, a King can only participate in a capture sequence if the other side does not have any direct attacker anymore. Otherwise the king move would be illegal, and the end of the recursion would be reached. In the approach of the iterative SEE with a value list discussed in Subsection 4.1.3 an illegal king move does not automatically trigger the termination of the forward-iteration loop. Only when the value list run through the reverse-iteration loop, the illegal king move can be considered.

In order to avoid these unnecessary iterations, which are executed by the illegal king move during a static exchange evaluation, an additional forward-pruning condition is to be implemented according to the $\alpha\beta$-approach. In the iterative $\alpha\beta$-approach lower bound α and upper bound β are always considered from White's point of view. This pruning procedure according to Equation 4.1 is thus called *king-α-pruning*, since during the breakup of the forward-iteration loop $return\_value := lower\_bound(white) = \alpha$ is returned. From Black's point of view the negated lower bound $return\_value := -lower\_bound(black) = -(-\beta) = \beta$ is returned.

The King-α-Pruning Condition

$$\begin{array}{lllll} \textbf{White:} & black\_attackers \geq 1 & \wedge & current\_attacker = white\_king \\ & & \Rightarrow & return\_value := \alpha \\ \textbf{Black:} & white\_attackers \geq 1 & \wedge & current\_attacker = black\_king \\ & & \Rightarrow & return\_value := \beta \end{array} \tag{4.1}$$

4.2.3 The Quantitative α-Pruning Condition

The $\alpha\beta$-window of the SEE decreases after the initialisation of $\alpha = -\infty$ and $\beta = board\_value(to)$ in the next iterations, until the pruning condition 4.1 or 4.2 is fulfilled or no further direct attackers are available anymore. The upper bound β can only be lowered in the iterations of the White. The lower bound α can only be raised in the iterations of the Black. A *quantitative α-pruning condition* from White's point of view can only be triggered, if the Black has a significant material win in the previous iteration $(i-1)$: $value_{i-1} + board\_value(next) \leq \alpha$. The material win from iteration $(i-1)$ should not be compensated for the recapture of the White in iteration (i). The quantitative α-pruning conditions for White and Black are summarised in Equation 4.2 from White's point of

view.

The Quantitative α-Pruning Condition (Fail-Hard | Fail-Soft)

$$\begin{aligned} \textbf{White:} \quad & value \leq \alpha \quad \Rightarrow \quad return\_value := \alpha \mid value \\ \textbf{Black:} \quad & value \geq \beta \quad \Rightarrow \quad return\_value := \beta \mid value \end{aligned} \tag{4.2}$$

Fail-Hard | Fail-Soft

In *fail-soft* $\alpha\beta$-approaches within the PVS only search results $\alpha \leq value \leq \beta$ are returned. In contrast, in *fail-hard* $\alpha\beta$-approaches within the PVS results in the value range $-mate + 1 \leq value \leq +mate - 2$ are returned. These two $\alpha\beta$-approaches can be applied to the iterative SEE. The distinction between fail-hard and fail-soft is irrelevant for accuracy and efficiency of the iterative $\alpha\beta$-SEE.

4.2.4 The Qualitative β-Pruning Condition

The SEE within the scope of a computer-chess architecture does not have to compute exact results, such as material modifications. In general, qualitative results are sufficient, so that any quantitative information about material modification by means of the SEE does not have to be considered. As long as $0 \in [\alpha, \ldots, \beta]$, no qualitative statements about the result of the SEE can be made, further iterations are necessary. Since the $\alpha\beta$-window is becoming increasingly smaller during the iterations, it is more probable that a qualitative β-pruning condition due to $0 \notin [\alpha, \ldots, \beta]$ is triggered. The qualitative β-pruning conditions for White and Black as well as the modifications of the upper bound β and the lower bound α in the iterations of White and Black from White's point of view are summarised in Equation 4.3.

The Qualitative β-Pruning Condition

$$\begin{aligned} \textbf{White:} \quad & value < \beta \quad \Rightarrow \quad \beta := value, \\ & \beta < 0 \quad \Rightarrow \quad return\_value := \beta \\ \textbf{Black:} \quad & value > \alpha \quad \Rightarrow \quad \alpha := value, \\ & \alpha > 0 \quad \Rightarrow \quad return\_value := \alpha \end{aligned} \tag{4.3}$$

Triggering a Qualitative β-Pruning Condition

In Figure 4.5 three iterations and the respective $\alpha\beta$-window in a horizontal value range $]-\infty, \ldots, \alpha = v_i, \ldots, 0, \ldots, \beta = v_k, \ldots, +\infty[$ are displayed. The $\alpha\beta$-window shrinks after iteration 1 and 2, alternately from the right ($\rightarrow$ upper bound) and from the left ($\rightarrow$ lower bound). After iteration 3, $\beta = v_k = v_2$.

As the upper bound is $\beta < 0$ and consequently $0 \notin [\alpha, \ldots, \beta]$, the qualitative β-pruning condition is fulfilled from White's point of view. The iteration can be terminated after iteration 3, since there already exists a qualitative static exchange evaluation.

Triggering a Qualitative β-Pruning Condition

White:	$-\infty$			$\ldots$			0	$\ldots$	v_0	$\ldots$	$+\infty$
				$\alpha\beta-window$							
Black:	$-\infty$	$\ldots$	v_1	$\ldots$			0	$\ldots$	v_0	$\ldots$	$+\infty$
			$\alpha\beta-window$								
White:	$-\infty$	$\ldots$	v_1	$\ldots$	v_2	$\ldots$	0	$\ldots$	v_0	$\ldots$	$+\infty$
			$\alpha\beta-window$								

Figure 4.5: The $\alpha\beta$-window and the static exchange evaluation iterations.

4.3 The SEE Implementation[8]

The approach of the iterative SEE from Subsection 4.1.3 is based on a value list, that is evaluated in a reverse iteration. Because of the use of the $\alpha\beta$-window according to Section 4.2 instead of a value list, the derived pruning conditions for White and Black, such as in Equations 4.1 to 4.3, can be implemented directly. The forward-iteration loop is terminated immediately, as soon as one of the pruning conditions is fulfilled or no direct attackers can move onto the destination square anymore. The algorithm in the source-code listing of Figure 4.6 computes the material modification of a move from White's point of view. Since the first iteration is executed directly after the initialisation, the actual forward-iteration loop starts with the Black. The initialisation of the lower bound $\alpha = -\infty$ is done in order to perform the special case of an illegal capture move of a king to a destination square which is defended by the opponent.

4.4 The $\alpha\beta$-SEE in Practice

The position of the sample board in Figure 4.7 was adopted from the *Sicilian Opening* (ECO = B52) in round 11 of the game Loop Amsterdam vs. Diep at the 15^{th} World Computer-Chess Championship, Amsterdam (NL) 2007. By means of this position, which emerged according to 21 ♕c1b2 a6 (see Appendix C.2), the mode of operation of the $\alpha\beta$-SEE can be shown quite well.

The static exchange evaluation of the capture move ♖×b6 should be computed iteratively, by neglecting the fast victory for Black according to 22. . . ♘×b6 23

[8]The following algorithm was presented at the Ph.D. Day at the Maastricht University, 2007.

Iterative SEE with αβ-Approach

```
int static_exchange_evaluation_w(const int from, const int to) {
  int value;
  int alpha;
  int beta;
  int next;
  int value_attacker;
  // iteration for White
  value = board_value(to);
  alpha = -INFINITE;
  beta = value;
  next = from;
  generate_attackers(next, to);
  if (black_attackers() == 0) return value;
  if (board_piece(from) == WHITE_KING) return alpha;
  value_attacker = board_value(next);
  // forward-iteration loop:
  // evaluate the value from White's point of view
  while (true) {
    // iteration for Black
    value -= value_attacker;
    if (value >= beta) return beta;
    if (value > alpha) alpha = value;
    if (alpha > 0) return alpha;
    next = next_black_attacker();
    update_x_ray_attackers(next, to);
    if (white_attackers() == 0) break;
    if (board_piece(next) == BLACK_KING) return beta;
    value_attacker = board_value(next);
    // iteration for White
    value += value_attacker;
    if (value <= alpha) return alpha;
    if (value < beta) beta = value;
    if (beta < 0) return beta;
    next = next_white_attacker();
    update_x_ray_attackers(next, to);
    if (black_attackers() == 0) break;
    if (board_piece(next) == WHITE_KING) return alpha;
    value_attacker = board_value(next);
  }
  if (value < alpha) return alpha;
  if (value > beta) return beta;
  return value;
}
```

Figure 4.6: Source-code listing for an iterative SEE with αβ-approach.

a5 ♘d7 24 ♘a2 ♘×e4 25 ♘×e4 ♕×e4.[9] Since square b6 is defended by two black pieces and the White has moved its indirect attacker on the 21st move to square b2, at least 3 forward iterations must be computed before the qualitative β-pruning is triggered.

Static Exchange Evaluation for ♖×b6: Loop vs. Diep, Amsterdam 2007

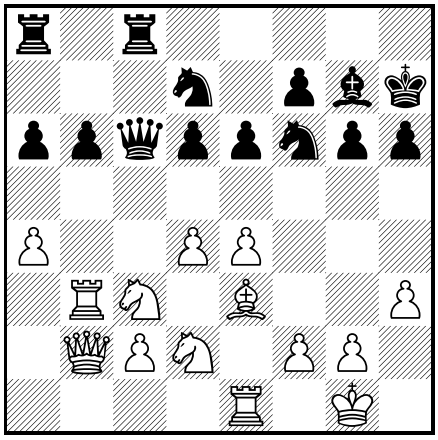


Figure 4.7: Qualitative β-pruning speeds up the iterative SEE. Loop's Rook on square b3 threatens Diep's Pawn on square b6.

Detailed Example

In the extracted positions in Figure 4.8 the piece constellations are shown prior to the respective iterations. Only direct attackers, indirect attackers (x-ray attackers), and the black Pawn on the destination square b6 are extracted. In order to display the mode of operation of the $\alpha\beta$ static exchange evaluation in this example, all iterations have been simulated. In every iteration the development of α, β and *value* is exactly split. In order to highlight precisely the right to move in the respective iteration, the positive sign is also added. Since no King is involved in the capture sequence, no king-α-pruning is triggered during the iterations. All following computations are to be seen from White's point of view.

Initialisation and Iteration 1

In Figure 4.8 (1) the white Rook on square b3 threatens the opposing black Pawn on square b6. The upper bound β and the current *value* are initialised with the maximally expected $material\_value(black\_pawn) = 100CP$. Lower bound α, that is an upper bound of Black, can only be raised in iteration 2 after the recapture of Black.

[9] One of the variations of this position as analysed by Loop.

$$\begin{aligned} value &:= material\_value(black\_pawn) = +100CP \\ \alpha &:= -\infty \\ \beta &:= value = +100CP \end{aligned}$$

Iteration 2

After the initialisation, the recapture of Black takes place in iteration 2, which is shown in Figure 4.8 (2). The update of *value* occurs via the subtraction of the $material\_value(white\_rook)$. The quantitative α-pruning condition for Black according to Equation 4.2 is not triggered. Qualitative-α-pruning as in Equation 4.3 is not triggered in this iteration either:

$$\begin{aligned} value &:= value - material\_value(white\_rook) = 100CP - 500CP = -400CP \\ &\quad\; value < \beta = +100CP \\ &\quad\; value > \alpha = -\infty \\ \alpha &:= value = -400CP \leq 0 \end{aligned}$$

Because of the move ...♘×b6, the indirect attacker for the move ♕×b6 is activated. Black does not have any indirect attackers for additional defence of square `b6`.

Iteration 3

The activated indirect attacker on square `b2` in Figure 4.8 (3) triggers the qualitative β-pruning condition in this iteration:

$$\begin{aligned} value &:= value + material\_value(black\_knight) = -400CP + 325CP = -75CP \\ &\quad\; value < \beta = +100CP \\ \beta &:= value = -75CP < 0 \Rightarrow \textit{Qualitative } \beta\textit{-Pruning} \end{aligned}$$

Iteration 4

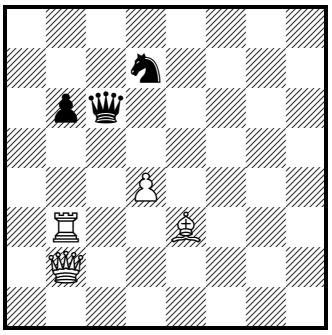
The board before the last iteration in Figure 4.8 (4) triggers no pruning condition with the move ...♕×b6. The iteration is terminated, since no further direct attackers are available concerning square `b6`. According to the algorithm in the source-code listing of Figure 4.6, the forward-iteration loop is terminated and the lower bound is returned as a final evaluation due to $value < \alpha$.

$$\begin{aligned} value &:= value - material\_value(white\_queen) = -75CP - 1000CP = -1075CP \\ &\quad\; value < \alpha = -400CP \Rightarrow return\_value = \alpha \end{aligned}$$

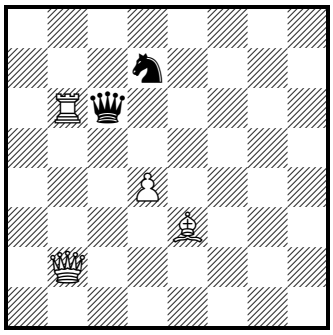
In Table 4.1 all 4 forward iterations, divided into qualitative and quantitative static exchange evaluations, are summarised. The asynchronous development

Static Exchange Evaluation: 1 ♖×b6 ♘×b6 2 ♕×b6 ♕×b6

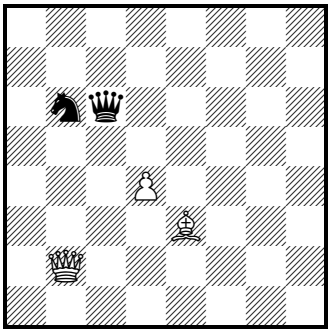
(1)



(2)



(3)



(4)

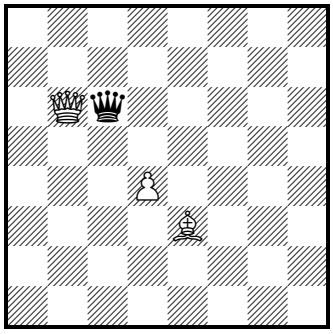


Figure 4.8: Loop's Rook on square b3 threatens Diep's Pawn on square b6. Three forward iterations have to be executed, at least.

of the $\alpha\beta$-window in the forward iterations of White and Black as well as the triggering of a break-up criterion or a pruning condition are comprehended rather simply in this manner.

$\alpha\beta$-Based Static Exchange Iterations						
	no pruning			qualitative β-pruning		
iteration	α	value	β	α	value	β
1	$-\infty$	+100	+100	$-\infty$	+100	+100
2	−400	−400	+100	−400	−400	+100
3	−400	−75	−75	−400	−75	−75
4	−400	−1075	−75	n/a	n/a	n/a

Table 4.1: Quantitative and qualitative $\alpha\beta$-SEE iterations.

4.5 SEE Applications

Over the past few years, the SEE has become a central element of a state-of-the-art computer-chess architecture. This algorithm is mainly used in the fields of (1) move ordering, (2) search, and (3) evaluation. Subsequently, some typical applications of the SEE in a computer-chess engine are introduced. The implementations originated from the engine Loop Amsterdam and are represented in a simplified or abbreviated way. Without loss of generality, optimisations and details, which are especially tuned for the playing style of Loop Amsterdam, are left out in the source-code listings.

This section contains four subsections. In Subsections 4.5.1 to 4.5.4 we will present four state-of-the-art applications for the SEE. Apart from the interactive pawn evaluation in Subsection 4.5.4, all SEE applications presented below are applied to the search engine of a computer-chess engine.

4.5.1 Selective Move Ordering

While selecting capture moves, it is possible to extract losing moves before launching the move loop in the PVS. These weak moves must be detected reliably. By means of the SEE, capture moves, which could cause a loss, can be scrutinized and evaluated. If the SEE considers many special cases and details, such as indirect attackers, en passant, and (under-) promotions, the extracting of capture moves will improve the move ordering significantly. Consequently, the search process works more efficiently, since fail-highs in *cut nodes* [34, 41] can be found faster and a further approximation to a *critical tree* or *minimal tree* [24, page 16] can be gained.

Source-Code Example

In the source-code listing of Figure 4.9, weak capture moves are extracted qualitatively. Capture moves, which are neither a promotion nor have another smaller

or equal material value as their captured piece

```
move_piece_captured(move)
```

are to be analysed additionally via the SEE. In this source-code listing the capture move will be estimated as good, if the static exchange evaluation is not negative ($\rightarrow value \geq 0$). One of the first open-source implementations of a qualitative move filter was published in 2005 in the project FRUIT by Letouzey [38]. This is called qualitative move filter because moves are filtered a priori according to the evaluated boolean result (true="good move" or false="bad move").

Source-Code Example: Qualitative Capture-Move Selection

```
bool capture_is_good_w(const int move) {
  if (move_captured(move)) {
    if (move_is_promotion(move)) return true;
    if (piece_value(move_piece_captured(move)) >=
        piece_value(move_piece(move))) return true;
  }
  else {
    if (static_exchange_evaluation_w(move) >= 0) return true;
  }
  return false;
}
```

Figure 4.9: Source-code listing for qualitative capture move selection with the use of SEE.

4.5.2 Selective α-Pruning

Many different selective forward-pruning procedures have been developed up to now. The best known techniques and procedures are *null move* by Donninger [13, 14] and *multi-cut $\alpha\beta$-Pruning* by Björnsson and Marsland [6]. The majority of forward-pruning procedures are based on β-pruning techniques and are aimed at finding a probable β-cutoff as early as possible. Selective α-pruning techniques, such as *late move reductions*, save the search of single subtrees and use information from *history tables* [62].[10] According to Romstad, late move reductions are based on the fact that in a computer-chess PVS with state-of-the-art move ordering, "a beta cutoff will usually occur either at the first node, or not at all" [47].

Source-Code Example

The source-code listing of Figure 4.10 shows the use of the SEE in a move loop during the PVS. Prior to the static exchange evaluation of the move

```
const int move
```

[10] *Late move reductions* is the correct term for this procedure, which is also well known as *history pruning*.

further necessary conditions must be verified first. The examination of these necessary conditions is quite simple and extracts already most of the candidate moves for a late move reduction (within the scope of Loop Amsterdam, approximately 60% of these nodes were extracted). The preconditions for a selective α-pruning can depend, according to Romstad, on various search parameters. While the SEE-condition for the selective α-pruning only validates the qualitative SEE-value $value < 0$ or $value \geq 0$, the preconditions can be tuned for the playing style of a computer-chess engine ($\rightarrow$ aggressive, offensive, defensive) or for the special requirements of a tournament ($\rightarrow$ opponents or time control).

Parameter Tuning

Parameter tuning within a computer-chess engine helps to adapt the playing style and the playing strength to special tournaments and test conditions. The combination of different preconditions requires precise setting, testing, and tuning. Subsequently, some reasonable criteria for preconditions are briefly presented. Assembling preconditions from these criteria is mostly quite difficult, that is why only the most interesting criteria are presented here: (1) the *node type* of the currently examined node according to Marsland and Popowich [41], (2) *done moves* is the number of the played moves of the current move loop, (3) *depth* is the distance between the transition from the PVS and the quiescence search, (4) *move*, the current move should be not an irreversible move ($\rightarrow$ capture move or pawn move), (5) *check*, and (6) *next check*, whether the friendly King is threatened or the opposing King is threatened by the move previously done.

Source-Code Example: Futility Filter for Selective α-Pruning

```
bool move_is_futile_w(const node_c & node, const int move) {
  if (node_is_pv(node) == false) {
    if (node_is_in_check(node) == false) {
      if (move_is_reversible(move) == true) {
        if (static_exchange_evaluation_w(move) < 0) return true;
      }
    }
  }
}
```

Figure 4.10: Source-code listing for selective α-pruning with static exchange evaluation to filter futile moves.

4.5.3 Fractional Search Extensions

A selective PVS is not only based on a balanced interplay of forward-pruning approaches and selective late move reductions [47], but also on selective search extensions. Extensions can be split exactly in units of $\frac{1}{4}$, $\frac{1}{12}$ or $\frac{1}{36}$ plies via fractional extensions and be controlled depending on further search parameters. When using fractional extensions, the divider should be chosen as skilfully as

possible. The more prime factors the divider has, the finer a ply can be split in multiple of $\frac{1}{2}, \frac{1}{3}, \frac{1}{4}$, etc. By using fractional extensions, it is possible to tune the search engine quite skilfully because the search tree is extended more precisely.

Nevertheless, too many unnecessary extensions are still triggered in practice, which leads to the explosion of a subtree. Independent of the weighting of a fractional extension ($\rightarrow$ the fraction of an extension), the necessity of an extension during the approximation to the critical tree or minimal tree [24] by means of the SEE should be verified additionally.

Source-Code Example

In the source-code listing of Figure 4.11 it is decided on a possible *pawn-push extension.*[11] Provided that the piece last moved is a pushing Pawn, a pawn-push extension will be triggered if no negative SEE-value is estimated for the Pawn's destination square. The negative SEE-value would result in the probable material loss of the respective Pawn.

Source-Code Example: Pawn-Push Extension Examination

```
bool extend_pawn_push_w(const int move) {
  if (piece_is_pawn(move_piece(move))) {
    if (is_7th_rank(move_to(move))) {
      if (static_exchange_evaluation_w(move) >= 0) return true;
    }
  }
  return false;
}
```

Figure 4.11: Source-code listing for examination of pawn-push extensions with static exchange evaluation.

4.5.4 Interactive Pawn Evaluation

Apart from the evaluation of the pawn structure, which is normally managed in special hash tables, the interactive interplay of all pieces on the chessboard should also be included in the evaluation of the Pawns.[12] If, for example, destination squares of moves are to be examined, the analysis of the squares of a passed Pawn will be quite interesting in the evaluation. Unlike the other pieces, the pawn structure is a compound, most important, and relatively inflexible framework. Pawn moves are irreversible and thus must be scrutinized during the interactive evaluation. A pawn structure once destroyed is hardly to be repaired again, and will have a great influence on the game.

"Pawns are the soul of chess", was the famous advice by Philidor in 1749.

[11] A pawn-push extension is, from White's point of view, the advancement of a Pawn onto the 6th or 7th rank.

[12] In contrast to the hashed pawn-structure evaluation, the dynamic interplay of all pieces is examined in the interactive evaluation.

4.6 Experiments with SEE Applications[13]

In Section 4.5, some interesting application fields of the SEE within a computer-chess architecture were presented. The aim of the following experiment is to find out, by means of the number of function calls of the SEE, in which application fields the SEE is predominantly called. A manual experimental setup seems to be more reasonable than a profile analysis (see Chapter 2). In order to measure the number of function calls, counters are implemented in the respective locations of the code for all applications.

As one could already presume prior to this experiment, the *selective move ordering* is among the most important application field of an SEE, as move ordering is especially carried out in the leaves of a search tree and the quiescence search. In Table 4.2 the results of this experiment are presented together with the counters. The experiment was carried out using common positions from the middlegame of a chess game. In column 2 the absolute number of the SEE-function calls is entered. In column 3 the relative share concerning the entire amount of the static exchange evaluations is shown. It is remarkable that the number of the function calls within the selective fractional search extensions is small. Since this approach of the extensions has an essential influence on the branching factor and thus also affects the time complexity of the search, the time consumption is to be neglected for static exchange evaluations within this application.

SEE Applications		
application	absolute (n)	relative (%)
selective move ordering	6,807,112	63
interactive pawn evaluation	2,225,768	21
selective α-pruning	1,672,752	16
fractional search extensions	53,972	< 1
sum of applications	10,759,604	100

Table 4.2: Experimental results about SEE applications.

4.7 Experiments with Pruning Conditions

The three pruning conditions discussed in Section 4.2 can be combined in all variations with each other. In the implementation of the source-code listing of Figure 4.6, all three derived pruning conditions are implemented according to Equations 4.1 to 4.3. King α-pruning works independently of the $\alpha\beta$-window and can thus be used independently of the quantitative α-pruning and qualitative β-pruning. From here on, the king α-pruning is called K_α, quantitative α-pruning is called QN_α and qualitative β-pruning is termed QL_β.

This section contains two subsections. In Subsections 4.7.1 and 4.7.2 we will analyse qualitative β-pruning by (1) counting the sum of iterations of combined

[13]The results of the following experiment were presented at the Ph.D. Day at the Maastricht University, 2007.

pruning conditions and (2) measuring the runtime performance of combined pruning conditions.

4.7.1 Qualitative β-Pruning in Action

In the first experiment the pruning conditions were combined with each other in order to find out in which iterations QL_β-pruning works most efficiently. In the first measurement all pruning conditions were enabled ($\rightarrow K_\alpha \cup QN_\alpha \cup QL_\beta$). In the second measurement QL_β was disabled. Out of theoretical considerations, it can be expected that QL_β can be used in iteration 3 for the first time, since at least 2 iterations are necessary in order to shrink the $\alpha\beta$-window sufficiently.

The experimental results in Table 4.3 show that in iteration 3, with 22% versus 14%, significantly more prunings are triggered, which is only to be attributed to QL_β-pruning. In columns 2 and 4 the absolute prunings are entered, i.e., in these iterations the static exchange evaluation was terminated due to a pruning. Therefore, it results that an evaluated summation of the prunings

$$sum\_of\_iterations = \sum_{iteration=1}^{8} iteration \times prunings(iteration)$$

is equal to the number of iterations. The sum of all prunings

$$sum\_of\_prunings = \sum_{iteration=1}^{8} prunings(iteration)$$

is only a control size to verify that both measurements have produced an equal number of prunings, though in different iterations.

Combining $\alpha\beta$-Pruning Conditions				
	$K_\alpha \cup QN_\alpha \cup QL_\beta$		$K_\alpha \cup QN_\alpha$	
iteration	absolute (n)	relative (%)	absolute (n)	relative (%)
1	1,587,815	31	1,587,815	31
2	1,878,661	37	1,878,661	37
3	1,125,784	22	690,402	14
4	417,542	8	751,490	15
5	50,658	1	123,425	2
6	16,658	< 1	42,378	1
7	710	< 1	2,961	< 1
8	189	< 1	898	< 1
sum of prunings	5,078,017	100	5,078,017	100
sum of iterations	10,752,377		11,321,581	

Table 4.3: Combining qualitative and quantitative $\alpha\beta$-pruning conditions.

4.7.2 Combined Pruning Conditions

The measurement of efficiency via the runtime performance of the SEE with combined pruning conditions is quite interesting. In Table 4.4 the results of this two-stage measurement are summarised. In the first stage of this measurement $n = 10$ SEE runs with every SEE call were processed in the computer-chess engine Loop Amsterdam. In the second stage of this test $n = 20$ SEE runs were carried out. In both measurements the runtime t_1 and t_2 were measured respectively. The difference in the runtime $t_2 - t_1$ describes the required time in milliseconds (ms) for the execution of $20 - 10 = 10$ SEE runs.

If the SEE algorithm with K_α-pruning is regarded to be 100%, the relative runtime of the different pruning combinations will be the following. The combination of all pruning approaches $K_\alpha \cup QN_\alpha \cup QL_\beta$ is, in contrast to the iterative SEE with K_α-pruning, approximately 19% more efficient concerning the performance behaviour in a computer-chess architecture.

Combining $\alpha\beta$-Pruning Conditions				
	$K_\alpha \cup QN_\alpha \cup QL_\beta$	$K_\alpha \cup QL_\beta$	$K_\alpha \cup QN_\alpha$	K_α
	time (ms)	time (ms)	time (ms)	time (ms)
t_1	10,620	10,890	11,600	11,840
t_2	15,530	16,010	17,500	17,890
$t_2 - t_1$	4,910	5,120	5,900	6,050
relative (%)	81	85	98	100

Table 4.4: Combining quantitative and qualitative $\alpha\beta$-pruning conditions.

4.8 Answer to Research Question 3

In this chapter we addressed the third research question. Apart from a recursive implementation, the emphasis was put on the iterative approach in the further course of this chapter. While using the $\alpha\beta$-window within the iterative implementation, we were able to control directly the static exchange evaluation during the computation. An iterative approach of the SEE with bounds has been already published by Kannan in 2007 in the Winboard Forum [31]. However, the pruning conditions were not completely implemented there. We developed pruning conditions depending on α and β, and combined them with each other. Their respective runtimes were examined in the computer-chess engine Loop Amsterdam. The use of the qualitative β-pruning condition caused the greatest rate increase in computing speed of the SEE, with approximately 15% in the experiment described in Section 4.7.

The details of the implementation of the SEE were neglected in this chapter, since both the Chapter 2 and Chapter 3 dealt with basic technologies for the implementation of computer-chess architectures. The details of the implementation of the SEE are similar, to a large extent, to those of a specific move generator (generation of checks, generation of non-capturing and non-promoting checks

(→ quite checks), or check evasions), since attacks, threats, and counter-attacks are also considered in these algorithms.

In Table 4.5 some excerpts from a profile analysis of the computer-chess engine Loop Amsterdam are taken. The test positions that were used represent all stages of an average chess game. A special 32-bit executable in debug mode has to be used for such a measurement.[14] The measured distribution of computing time of functions is only an approximate estimation.

Within the computer-chess engine Loop Amsterdam, two SEE-functions (→ `see_w` and `see_b`) for White and Black have together a time consumption of 10.4% + 3.1% = 13.5%. Therefore, the time consumption of the SEE, as measured in per cent, is only slightly smaller than the time consumption of all evaluation functions. The entire time consumption of the evaluation during the profile analysis is summarised in the right main column *with substack*.[15] Here, all function calls and sub-functions, which are above the main evaluation in the program stack, are considered.

Furthermore, Table 4.5 lists both capture-move generators (→ `gen_captures_w` and `gen_captures_b`) with the highest time consumption of this measurement for the sake of comparison. In total, their share of the entire time during a game played by Loop Amsterdam is 1.8%. This is quite small. The field of move generation still remains the subject of intense research and development ("Move Generation with Perfect Hash Functions", Fenner and Levene 2008 [21], "Not a Power of Two: Using Bitboards for Move Generation in Shogi", Grimbergen 2007 [23], "Magic Move-Bitboard Generation in Computer Chess", Kannan 2007 [30]), although there are functions within a state-of-the-art computer-chess architecture, which have similar requirements on the computer-chess architecture and are by far more interesting.

Profile Analysis: The Static Exchange Evaluator					
		without substack (net)		with substack (gross)	
func name	func calls	time (ms)	rel (%)	time (ms)	rel (%)
`evaluate`	721,993	3,438	6.8	7,796	15.5
`see_b`	677,255	2,451	4.9	5,231	10.4
`see_w`	223,268	734	1.5	1,578	3.1
`gen_captures_b`	202,416	482	1.0	482	1.0
`gen_captures_w`	165,089	381	0.8	381	0.8

Table 4.5: Profile analysis for SEEs in comparison with the evaluation function and the most time consuming move generators.

[14] The measurement is carried out on a single-core computer system with 2600 MHz using Microsoft Visual Studio 6.0, 32-bit.

[15] In the time measurement with substack all sub-functions are considered → gross. In the time measurement without substack all called sub-functions are neglected → net.

Conclusions Research Question 3

Our profile analysis in Table 4.5 gives a brief overview of the distribution of CPU times of all functions. We admit that the $\alpha\beta$-SEE consumes a considerable amount of the entire CPU time in the computer-chess engine Loop Amsterdam. Yet, we may conclude that the successful combination of king α-pruning, quantitative α-pruning, and qualitative β-pruning makes it possible to use the $\alpha\beta$-approach within this complex algorithm extensively in a computer-chess engine.

Future Research

In Section 4.5 the use of the SEE in computer-chess engines Loop Leiden, Loop Amsterdam, and Loop Express for Nintendo Wii has been introduced. We showed that there are only a few possible application fields of the SEE. For future research in the application field of the SEE, it would be most interesting to implement the SEE in (1) the move ordering of the quiescence search and (2) the interactive evaluation. An interesting idea for the implementation of the SEE is offered by Kannan. This idea was posted in the Winboard Forum in 2007 [31]. In his approach Kannan claims that "a piece behind another piece is always of a greater or equal value".

Chapter 5

Conclusions and Future Research

In Section 1.3 the requirements for the development of a computer-chess architecture were summarized. From these three requirements, (1) unlimited implementation of chess knowledge, (2) higher computing speed, and (3) minimal overhead the following problem statement has guided our research.

> **Problem statement:** *How can we develop computer-chess architectures in such a way that computer-chess engines combine the requirements on knowledge expressiveness with a maximum of efficiency?*

The problem statement was dealt with in three steps. For every step a research question was formulated. We approached the research questions with implementations and experiments, and dealt with them in an own chapter. All implementations were tested in the environment of the computer-chess engines Loop Leiden 2006, Loop Amsterdam 2007, and Loop Express. Moreover, they were optimised computer platform-independently. In this way, each idea was tested and evaluated in a fully functional state-of-the-art competitive computer-chess engine. All claims have been tested rigorously in practice.

In Section 5.1 the answers to the three research questions are summarized. In Section 5.2 the problem statement is answered. In Section 5.3 recommendations for future research follow, which could not be dealt within the framework of this thesis but may contribute to the field of computer-chess architectures and further computer-game architectures.

5.1 Conclusions on the Research Questions

In this section the answers to our three research questions stated in Section 1.3 are revised and placed in an overall context. Since every research question was dealt with in an own chapter, the scientific results about (1) non-bitboard computer-chess architectures (see Subsection 5.1.1), (2) magic hash functions

for bitboards and magic multiplications (see Subsection 5.1.2), and (3) static exchange evaluation (see Subsection 5.1.3) are provided in different subsections.

5.1.1 Non-Bitboard Architectures

A state-of-the-art computer-chess architecture must meet the requirements of (1) competitiveness in speed, (2) simplicity, and (3) ease of implementation. We set ourselves the task to fulfil these requirements optimally with elementary and highly efficient techniques without using 64-bit unsigned integers. This led to the first research question.

> **Research question 1:** *To what extent can we develop non-bitboard computer-chess architectures, which are competitive in speed, simplicity, and ease of implementation?*

As we have shown, a high-performance framework could be developed and tested in Chapter 2 due to the harmonious interplay of (1) the internal computer chessboard (see Section 2.2), (2) the detailed piece lists (see Section 2.3), and (3) the two blocker loops (see Section 2.5). This framework is explicitly not based on the use of bitboards. Therefore, it can be implemented more flexibly as the size of the board with $n \times m > 64$ squares can be chosen almost arbitrarily. Many chess variants [7] as for example, Gothic Chess, 10×8 Capablanca Chess, Glinski's Hexagonal Chess, and board games, such as computer Shogi, can use this technology.

In a 32-bit computer system a speed improvement of approximately 32 to 40% was measured in the environment of an advanced brute-force recursion (see Table 2.5). This is in contrast to the structurally identical computer-chess architecture based on Rotated Bitboards. In a 64-bit computer environment a speed improvement of approximately 3 to 5% was measured under the same conditions. The performance of the New Architectures was verified in this experiment by means of recursive brute-force performance algorithms (see source-code listings in Appendix A.1).

This computer-chess architecture was first implemented in the quite successful computer-chess engine Loop Leiden 2006. The computer-chess architecture was later implemented partially in Hydra and completely in Wii Chess by Nintendo (see Section 2.1). The high performance was just as important for these projects as the simplicity and ease of implementation in the following two environments: (1) the environment of a sophisticated computer-chess machine (→ Hydra) and (2) the environment of a commercial games console with the highest quality and security standards (→ Loop Express for Nintendo).

5.1.2 Magic Hash Functions for Bitboards

As we have shown, the use of 64-bit unsigned integers (bitboards) was quite memory-efficient for the management of board related information. The empirical results in Section 2.7 confirmed that the use of bitboards is redundant and inefficient. The reason is that board related information for the computation of sliding movements is to be managed in parallel many times (→ Rotated

Bitboards). For this reason, fast and simple (perfect) hash functions must be developed. Thus, bitboards could be examined efficiently and with minimum redundancy. This led to the second research question.

> **Research question 2:** *To what extent is it possible to use hash functions and magic multiplications in order to examine bitboards in computer chess?*

For the development of a simple and possibly fast (perfect) hash function, only simple arithmetic or Boolean operations can be used. In Chapter 3 we had an additional goal: indication of (1) arbitrary n-bit unsigned integers, such as multiple 1-bit computer words, and (2) compound bit-patterns (diagonal, horizontal, or vertical lines) via hash functions. Only in this way it is possible to develop (1) a challenging bit scan and (2) the movement of sliding pieces without computing redundant rotated bitboards.

These high requirements for a hash function can only be met through multiplications of the n-bit integers (input keys) by suitable multipliers, the so-called *magic multipliers*. The product of this multiplication contains the unique index to address the corresponding hash table. This operation is sufficiently fast on a 64-bit computer environment and offers furthermore sufficient space to examine different bitboards, such as multiple 1-bit computer words and compound bit-patterns via perfect mapping.

A complete computer-chess architecture based on hash functions and magic multiplications for the examination of bitboards is implemented in the computer-chess engine Loop Amsterdam. This engine was able to reach the 3^{rd} place at the 15^{th} World Computer-Chess Championship, Amsterdam (NL) 2007. An essential reason for the success of this 64-bit computer-chess engine was the use of highly sophisticated perfect hash functions and magic multipliers for the computation of compound bit-patterns (bitboards) via perfect hashing.

5.1.3 Static Exchange Evaluation

The *static exchange evaluator* (SEE) is an important tool for the qualitative and quantitative evaluation of moves and threatened squares within the scope of a state-of-the-art computer-chess architecture. The results of this deterministic approach are quite precise. The application of the SEE algorithm is straightforward, and the field of application is enormous (e.g., move ordering, evaluation, forward pruning, etc.). However, the pruning conditions can only be implemented into the SEE with an iterative $\alpha\beta$-approach. Then the procedure becomes clearly more efficient. The third research question resulted from this observation.

> **Research question 3:** *How can we develop an $\alpha\beta$-approach in order to implement pruning conditions in the domain of static exchange evaluation?*

It is hardly possible to prune a priori unnecessary capture-move sequences with a recursive algorithm or an iterative algorithm using a value list during the

static exchange evaluation. In Section 4.2 the iterative $\alpha\beta$-approach was introduced. Based on this approach, we developed three pruning conditions: (1) the King-α-pruning condition, (2) the quantitative α-pruning condition, and (3) the qualitative β-pruning condition.

Due to the further development of the iterative SEE as given in Section 4.1 and the implementation of the $\alpha\beta$-window, we could dispense with an additional value list. Since the evaluation of the value list via reverse iteration is unnecessary, the new algorithm is more compact and even more efficient without pruning conditions (K_α, QN_α, and QL_β). The implementation of the three pruning conditions in the iterative $\alpha\beta$-SEE proceeded without any difficulties.

Due to the combination of the quantitative α-pruning condition and the qualitative β-pruning condition, a performance increase of approximately 19% could be gained within the computer-chess engine Loop Amsterdam in the experiment in Section 4.7. From these results the new qualitative β-pruning condition has gained the highest pruning performance. However, we admit that the highly sophisticated SEE consumes with approximately 13% (see Table 4.5) clearly more CPU time than the most frequently used move generators (approximately 2%). The iterative $\alpha\beta$-based SEE is the most interesting deterministic algorithm within the 64-bit computer-chess architecture of Loop Amsterdam and the 32-bit computer-chess architecture of Loop Express.

5.2 Conclusions on the Problem Statement

In Section 1.3 we formulated the following problem statement.

> **Problem statement:** *How can we develop new computer-chess architectures in such a way that computer-chess engines combine the requirements on knowledge expressiveness with a maximum of efficiency?*

The computer-chess architectures and algorithms, that were developed and tested in this thesis, were successful in international computer-chess championships (see Appendix C). The implementation of complex chess knowledge with a maximum of efficiency has succeeded within the framework of the computer-chess architectures on 32-bit and 64-bit computer environments. Careful tuning and extensive testing were necessary to achieve a high level of performance.[1]

32-Bit and 64-Bit Computer Environments

The computer-chess architecture introduced in Chapter 2 is efficient on 32-bit and 64-bit computer environments. The implementation of complex chess knowledge within the computer-chess engine Loop Leiden was successful. During the 26^{th} Open Dutch Computer-Chess Championship (see Appendix C.1) Loop Leiden played with a performance of 1.8×10^6 to $3.1 \times 10^6 \frac{nodes}{seconds \times cores}$.

[1] Every algorithm and technology presented within this thesis were tested by independent β-testers more than 10,000 hours on different computer environments.

64-Bit Computer Environments

The use of hash functions and magic multiplications introduced in Chapter 3 is only competitive in speed on 64-bit computer environments. The implementation of complex chess knowledge within the computer-chess engine Loop Amsterdam was successful. During the 15^{th} World Computer-Chess Championship (see Appendix C.2) Loop Amsterdam played with a performance of 2.1×10^6 to $3.5 \times 10^6 \frac{nodes}{seconds \times cores}$.

5.3 Recommendations for Future Research

In this section three of the most interesting recommendations for future research are listed.

1. **Board games with $n \times m > 64$ squares.** The technologies and approaches which were developed in Chapter 2 can, in contrast to bitboards, also be applied to board games with $n \times m$ squares ($n \times m > 64$). An analysis of the performance of a 9×9 computer-Shogi architecture based on the New Architectures discussed in Chapter 2 in comparison with bitboards [23] would be quite interesting. Such an experiment would provide valuable findings, to what extent competitiveness in speed, simplicity, and ease of implementation of the New Architectures can be achieved in other board games.

2. **Magic hash functions.** A perfect hash function with magic multiplications for the examination of bitboards is a high-performance technology for computer-chess architectures. In order to be able to improve our understanding of the magic multiplication, it is necessary to gain a deeper understanding of multiplications of unsigned integers with overflow in mathematical rings (see Section 3.9). The research of bit patterns and the experiments with magic multiplier sets with $n \leq 6$ bits at the end of Section 3.8 was a first step in order to improve the understanding of this complex process. The reduction of the size of sub-hash tables for the examination of compound bit patterns to around 1 to 2 bits should be possible (see Section 3.9).

3. **Branching factor.** The precise results of a static exchange evaluation can be used for the reduction of the branching factor of a computer-chess search engine. Three of the four introduced applications of the SEE discussed in Section 4.5 were implemented in the computer-chess engine Loop Amsterdam to reduce the branching factor. It should be possible to improve the game performance with further and more skilful applications of the SEE within a sophisticated game-tree search and quiescence search through reduction of the branching factor.

   Finally, in Tables 5.1 and 5.2 the branching factors of some highly sophisticated computer-chess engines from 2006 are listed. The branching factors are computed as a quotient of the entire time after searching a new

ply and the entire time after searching the previous ply.[2]

$$branching\_factor = \frac{time(current\_ply)}{time(previous\_ply)}$$

The results of the quotient in Table 5.1 apply to the computer-chess engines tested in opening positions. In analogy, the results of the quotient in Table 5.2 apply to the computer-chess engines tested in middlegame positions. Both tables are arranged according to the average branching factor in ascending order in the last column. These results of measurement confirm our assumption, that stronger computer-chess engines and even stronger sub-versions of computer-chess engines (RYBKA 1.1 → RYBKA 1.2f) have a smaller branching factor.

Branching Factors of Computer-Chess Engines								
	branching factors (plies)							
chess engine	12	13	14	15	16	17	18	average
RYBKA 1.2f	1.8	1.6	2.2	2.2	2.1	n/a	n/a	1.96
RYBKA 1.1	1.8	1.9	2.0	2.2	2.7	n/a	n/a	2.09
SHREDDER 10 UCI	2.0	3.1	1.9	2.3	1.7	n/a	n/a	2.15
LOOP 2006	2.2	2.0	2.7	2.2	2.9	n/a	n/a	2.37
HIARCS x50 UCI	1.9	2.6	2.7	2.8	n/a	n/a	n/a	2.47

Table 5.1: Branching factors of computer-chess engines in opening positions.

Branching Factors of Computer-Chess Engines								
	branching factors (plies)							
chess engine	12	13	14	15	16	17	18	average
RYBKA 1.2f	1.7	1.7	1.6	2.3	1.8	1.8	1.9	1.81
RYBKA 1.1	2.0	1.8	1.8	2.3	2.0	2.0	1.9	1.96
LOOP 2006	n/a	2.3	1.8	2.0	2.1	2.1	2.2	2.07
SHREDDER 10 UCI	n/a	1.9	1.9	2.3	2.0	2.9	2.6	2.23
HIARCS x50 UCI	n/a	2.3	2.4	2.5	2.6	2.2	n/a	2.39

Table 5.2: Branching factors of computer-chess engines in middlegame positions.

[2]The results are derived from the experiment carried out by Gerhard Sonnabend, the main β-tester for all LOOP-engines since 2005.

References

[1] Computer Chess Club: Programming and Technical Discussions. Website, 2008. Available online at `http://talkchess.com/forum/index.php`. Last accessed February 15th 2009.

[2] Winboard Forum: Programming and Technical Discussions. Website, 2008. Available online at `http://www.open-aurec.com/wbforum/`. Last accessed June 17th 2008.

[3] R.B. Allenby. *Rings, Fields, and Groups: An Introduction to Abstract Algebra*. 2nd ed. Oxford, England: Oxford University Press, 1991.

[4] D.F. Beal. *The Nature of MINIMAX Search*. IKAT, 1999, Universiteit Maastricht, The Netherlands, 1999.

[5] J. Becker, M. Platzner, and S. Vernalde, editors. *Field Programmable Logic and Application, 14th International Conference, FPL 2004, Leuven, Belgium, August 30-September 1, 2004, Proceedings*, volume 3203 of *Lecture Notes in Computer Science*. Springer, 2004.

[6] Y. Björnsson and T.A. Marsland. Multi-Cut Alpha-Beta Pruning in Game-Tree Search. *Theoretical Computer Science*, 252(1-2):177–196, 2001.

[7] H.L. Bodlaender and D. Howe. The Chess Variant Pages. Website, 2008. Available online at `http://www.chessvariants.org/`. Last accessed February 15th 2009.

[8] M.G. Brockington and J. Schaeffer. APHID: Asynchronous Parallel Game-Tree Search. *J. Parallel Distrib. Comput.*, 60(2):247–273, 2000.

[9] I.N. Bronstein, K.A. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, Frankfurt am Main, Thun, 1999.

[10] Microsoft Corporation. 64-Bit Programmierung mit Visual C++. Technical report, March 2005. MSDN Library - Visual Studio 2005.

[11] S. Cracraft. Bitmap Move Generation in Chess. *ICCA Journal*, 7(3):146–153, 1984.

[12] S. Cracraft, J. Stanback, D. Baker, C. McCarthy, M. McGann, and C. Kong Sian. Chess. Website, 1985-1996. Available online at `http://www.gnu.org/software/chess/`. Last accessed February 15th 2009.

[13] C. Donninger. Null Move and Deep Search: Selective-Search Heuristics for Obtuse Chess Programs. *ICCA Journal*, 16(3):137–143, 1993.

[14] C. Donninger. Verified Null-Move Pruning. *ICCA Journal*, 25(3):153–161, 2002.

[15] C. Donninger. Discussion with Dr. Christian Donninger. 2006.

[16] C. Donninger. Von Bytes und Bauern. *Kaissiber Journal*, 28:10–17, Juli-September 2007.

[17] C. Donninger, A. Kure, and U. Lorenz. Parallel Brutus: The First Distributed, FPGA Accelerated Chess Program. In *IPDPS*, 2004. Proc. of 18th International Parallel and Distributed Processing Symposium (IPDPS), Santa Fe - NM USA, IEEE Computer Society.

[18] C. Donninger and U. Lorenz. The Chess Monster Hydra. In *Proc. of 14th International Conference on Field-Programmable Logic and Applications (FPL), Antwerp - Belgium, LNCS 3203*, pages 927–932, 2004.

[19] C. Donninger and U. Lorenz. The Hydra Project. *XilinX Journal (selected paper), Issue 53*, 2005.

[20] R. Feldmann, P. Mysliwietz, and B. Monien. Game Tree Search on a Massively Parallel System. In H. J. van den Herik, I. S. Herschberg, and J. W. H. M. Uiterwijk, editors, *Advances in Computer Chess*, pages 203–218, Maastricht, The Netherlands, 1994. University of Limburg.

[21] T. Fenner and M. Levene. Move Generation with Perfect Hash Functions. *ICGA Journal*, 31(1):3–12, 2008.

[22] V.S. Gordon and A. Reda. Trappy Minimax - using Iterative Deepening to Identify and Set Traps in Two-Player Games. In *CIG*, pages 205–210, 2006. Available online at `http://www.ecs.csus.edu/csc/pdf/techreports/trappy.pdf`. Last accessed February 15th 2009.

[23] R. Grimbergen. Not a Power of Two: Using Bitboards for Move Generation in Shogi. *ICGA Journal*, 30(1):25–34, 2007.

[24] E.A. Heinz. *Scalable Search in Computer Chess*. Friedr. Vieweg und Sohn Verlagsgesellschaft mbH, Braunschweig/Wiesbaden, 2000, 1999.

[25] R.M. Hyatt. Chess program board representations. Website. Available online at `http://www.cis.uab.edu/hyatt/boardrep.html`. Last accessed February 15th 2009.

[26] R.M. Hyatt. The DTS high-performance parallel tree search algorithm. Website. Available online at `http://www.cis.uab.edu/hyatt/search.html`. Last accessed February 15th 2009.

[27] R.M. Hyatt. Rotated Bitmaps, A New Twist on an Old Idea. *ICCA Journal*, 22(4):213–222, 1999.

[28] R.M. Hyatt and T. Mann. A lockless transposition table implementation for parallel search. *ICGA Journal*, 25(1):36–39, 2002. Available online at `http://www.cis.uab.edu/hyatt/hashing.html`. Last accessed February 15th 2009.

[29] R. Isernhagen. *Softwaretechnik in C und C++*. 1999 Carl Hanser Verlag München Wien, 1999.

[30] P. Kannan. Magic Move-Bitboard Generation in Computer Chess. 2007. Available online at `http://www.pradu.us/old/Nov27_2008/Buzz/research/magic/Bitboards.pdf`. Last accessed February 15th 2009.

[31] P. Kannan. SEE with magic bitboards. Website, January 2007. Available online at `http://www.open-aurec.com/wbforum/viewtopic.php?t=6104`. Last accessed June 17th 2008.

[32] L. Kaufman. The Evaluation of Material Imbalances. Website, March 1999. Available online at `http://home.comcast.net/~danheisman/Articles/evaluation_of_material_imbalance.htm`. Last accessed February 15th 2009.

[33] H. Kmoch. *Die Kunst der Bauernführung*, pages 37–39. Siegfried Engelhardt Verlag Berlin-Frohnau, 1967.

[34] D.E. Knuth and R.W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[35] M. Lefler, P. Kannan, G. Isenberg, P. Koziol, and Z. Wegner. Bitboards. Website, 2008. Available online at `http://chessprogramming.wikispaces.com/Bitboards`. Last accessed February 15th 2009.

[36] M. Lefler, P. Kannan, G. Isenberg, P. Koziol, and Z. Wegner. List of magics for bitboard move geneartion. Website, 2008. Available online at `http://chessprogramming.wikispaces.com/`. Last accessed February 15th 2009.

[37] C.E. Leiserson, H. Prokop, and K.H. Randall. Using de Bruijn Sequences to Index a 1 in a Computer Word. Website, July 1998. Available online at `http://supertech.csail.mit.edu/papers/debruijn.pdf`. Last accessed February 15th 2009.

[38] F. Letouzey. Fruit - pure playing strength. Website, 2005. Available online at `http://www.fruitchess.com/`. Last accessed February 15th 2009.

[39] F. Letouzey. E-mail Dialogue with Fabien Letouzey. 2006.

[40] V. Manohararajah. The Design of a Chess Program. *ICCA Journal*, 20(2):87–91, 1997.

[41] T.A. Marsland and F. Popowich. Parallel Game-Tree Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI*, 7(4):442–452, 1985.

[42] B. Moreland. Programming Topics. Website, 2003. Available online at `http://web.archive.org/web/20070707012511/http://www.brucemo.com/compchess/programming/index.htm`. Last accessed February 15th 2009.

[43] H. Nelson. Hash Tables in Cray Blitz. *ICCA Journal*, 8(1):3–13, 1995.

[44] G. Osborne and G. Isenberg. How to reduce the "bits" used in a magic number ca. Website, 2008. Available online at `http://64.68.157.89/forum/`. Last accessed June 17th 2008.

[45] V. Rajlich. Rybka v 1.0 Beta Readme. readme.txt, December 2005. Was available with the computer-chess engine Rybka 1.0 Beta.

[46] A. Reinefeld and T.A. Marsland. Enhanced Iterative-Deepening Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI*, 16(7):701–710, July 1994.

[47] T. Romstad. An Introduction to Late Move Reductions. Website, 2007. Available online at `http://www.glaurungchess.com/lmr.html`. Last accessed February 15th 2009.

[48] T. Romstad. Discussion with Tord Romstad. 2007.

[49] T. Romstad. Glaurung and Scatha. Website, 2008. Available online at `http://www.glaurungchess.com/`. Last accessed February 15th 2009.

[50] T. Romstad. SEE problem. Website, April 2008. Available online at `http://64.68.157.89/forum/viewtopic.php?t=20646`. Last accessed February 15th 2009.

[51] D. Rorvik. Programming - 2's Complement Representation for Signed Integers. Website, May 2003. Available online at `http://academic.evergreen.edu/projects/biophysics/technotes/program/2s_comp.htm`. Last accessed February 15th 2009.

[52] C.E. Shannon. Programming a Computer for Playing Chess. *Philosophical Magazine*, 41(7):256–275, 1950.

[53] D.J. Slate and L.R. Atkin. *Chess Skill in Man and Machine*, chapter 4. Chess 4.5 – Northwestern University Chess Program, pages 82–118. 1977.

[54] ICGA Team. 15th World Computer Chess Championship. Website, 2008. Available online at `http://www.grappa.univ-lille3.fr/icga/tournament.php?id=173`. Last accessed February 15th 2009.

[55] A.M. Turing. Digital Computers Applied to Games. In B.V. Bowden, editor, *Faster Than Thought*, pages 286–297, London, England, 1953. Pitman Publishing.

[56] unkown. Bitscan forward. Website, 2008. Available online at `http://chessprogramming.wikispaces.com/BitScan`. Last accessed February 15th 2009.

[57] T. van der Storm. 26th Open Dutch Computer Chess Championship. Website, 2008. Available online at `http://old.csvn.nl/docc06.html`. Last accessed February 15th 2009.

[58] H. van Kempen. Website, 2008. Available online at `http://www.husvankempen.de/nunn/`. Last accessed June 17th 2008.

[59] B. Vovk. How the Elo rating system works in chess. Website, 2008. Available online at `http://www.chesselo.com/`. Last accessed June 17th 2008.

[60] V. Vučković. The Compact Chessboard Representation. *ICGA Journal*, 31(3):157–164, 2009.

[61] J.C. Weill. The ABDADA Distributed Minimax Search Algorithm. *ICCA Journal*, 19(1):3–16, 1996. Available online at `http://www.recherche.enac.fr/~weill/publications/acm.ps.gz`. Last accessed February 15th 2009.

[62] M.H.M. Winands, E.C.D. van der Werf, H.J. van den Herik, and J.W.H.M. Uiterwijk. The Relative History Heuristic. In H.J. van den Herik, Y. Björnsson, and N.S. Netanyahu, editors, *Computers and Games*, volume 3846 of *Lecture Notes in Computer Science (LNCS)*, pages 262–272, Berlin, Germany, 2006. Springer-Verlag.

Appendix A

Source-Code Listings

This appendix contains two sections. In Section A.1 C/C++ source codes according to the brute-force performance experiments in Section 2.7 are given. In Section A.2 additional C/C++ source codes according to the trial-and-error algorithm in Section 3.7 are given for completeness.

A.1 Recursive Brute-Force Performance Algorithms

Below two recursive brute-force algorithms

```
uint64 basic_recursion(board_c & board, const int depth);
uint64 advanced_recursion(board_c & board, const int depth);
```

are listed. These algorithms were used for an experiment in Section 2.7. The reference to a board object is passed via the parameter:

```
board_c & board
```

The class

```
board_c
```

contains the information about the current chessboard (e.g. castle status, en passant square, etc.). The maximal recursion depth of both algorithms is controlled by the passed parameter:

```
const int depth
```

The additional identifier

```
undo_c undo
```

is needed in order to save information about the (1) en passant square, the (2) fifty move rule, and (3) for restoring hash keys. The only difference between these two algorithms is the order of doing/undoing moves. The generated moves in the first algorithm are done/undone sequentially. In the second algorithm the moves are ordered according to heuristic rules before they are done/undone. The class

```
sort_c
```

and the two function calls

```
void sort_initialize(board_c & board, sort_c & sort, int check);
int sort_move(board_c & board, sort_c & sort);
```

initialise and manage the move ordering and move selection.

Basic Brute-Force Recursion

```
uint64 basic_recursion(board_c & board, const int depth) {
  uint64 nodes;
  int moves;
  int move_list[256];
  undo_c undo;

  nodes=0;
  moves=generate_moves(board, move_list);

  for (int i=0; i < moves; i++) {
    int move=move_list[i];
    do_move(board, undo, move);
    if (!is_king_attacked(board)) {
      if (depth > 1)
        nodes+=basic_recursion(board, depth-1);
      else
        nodes++;
    }
    undo_move(board, undo, move);
  }
  return nodes;
}
```

Advanced Brute-Force Recursion

```
uint64 advanced_recursion(board_c & board, const int depth) {
  uint64 nodes;
  int move;
  int check;
  sort_c sort;
  undo_c undo;

  nodes=0;
  check=is_king_attacked(board);
  sort_initialize(board, sort, check);

  while ((move=sort_move(board, sort)) != MOVE_NONE) {
    do_move(board, undo, move);
    if (depth > 1)
      nodes+=advanced_recursion(board, depth-1);
    else
      nodes++;
    undo_move(board, undo, move);
  }
  return nodes;
}
```

A.2 Generation of Masks and Attacks

Below all additional source-codes listings for the generation of file masks and rank masks

```
uint64 generate_a1a8_mask(int square);
uint64 generate_a1h1_mask(int square);
```

as well as diagonal masks

```
uint64 generate_a1h8_mask(int square);
uint64 generate_h1a8_mask(int square);
```

are given. Furthermore, all additional source-codes listings for the generation of file attacks and rank attacks

```
uint64 generate_a1a8_attack(int square, uint64 blockers);
uint64 generate_a1h1_attack(int square, uint64 blockers);
```

as well as diagonal attacks

```
uint64 generate_a1h8_attack(int square, uint64 blockers);
uint64 generate_h1a8_attack(int square, uint64 blockers);
```

are given. For the generation of masks we need only to pass the parameter:

```
int square
```

For the generation of attacks the further parameter

```
uint64 blockers
```

is required in order to generate the corresponding attack mask for a given bitboard of blockers.

Generation of File Masks

```
uint64 generate_a1a8_mask(int square) {
  uint64 bitboard;
  int file;
  int rank;
  int r;

  bitboard = 0;

  file = square_file[square];
  rank = square_rank[square];

  for (r = rank + 1; r <= 6; r++)
    bitboard |= file_rank_to_bitboard(file, r);
  for (r = rank - 1; r >= 1; r--)
    bitboard |= file_rank_to_bitboard(file, r);

  return bitboard;
}
```

Generation of Rank Masks

```
uint64 generate_a1h1_mask(int square) {
  uint64 bitboard;
  int file;
  int rank;
  int f;

  bitboard = 0;

  file = square_file[square];
  rank = square_rank[square];

  for (f = file + 1; f <= 6; f++)
    bitboard |= file_rank_to_bitboard(f, rank);
  for (f = file - 1; f >= 1; f--)
    bitboard |= file_rank_to_bitboard(f, rank);

  return bitboard;
}
```

Generation of Diagonal Masks

```
uint64 generate_a1h8_mask(int square) {
  uint64 bitboard;
  int file;
  int rank;
  int f;
  int r;

  bitboard = 0;

  file = square_file[square];
  rank = square_rank[square];

  for (r = rank + 1, f = file + 1; r <= 6 && f <= 6; r++, f++)
    bitboard |= file_rank_to_bitboard(f, r);
  for (r = rank - 1, f = file - 1; r >= 1 && f >= 1; r--, f--)
    bitboard |= file_rank_to_bitboard(f, r);

  return bitboard;
}

uint64 generate_h1a8_mask(int square) {
  uint64 bitboard;
  int file;
  int rank;
  int f;
  int r;

  bitboard = 0;

  file = square_file[square];
  rank = square_rank[square];

  for (r = rank + 1, f = file - 1; r <= 6 && f >= 1; r++, f--)
    bitboard |= file_rank_to_bitboard(f, r);
  for (r = rank - 1, f = file + 1; r >= 1 && f <= 6; r--, f++)
    bitboard |= file_rank_to_bitboard(f, r);

  return bitboard;
}
```

Generation of Bishop and Rook Masks

```
uint64 generate_bishop_mask(int square) {
  return generate_a1h8_mask(square) |
         generate_h1a8_mask(square);
}

uint64 generate_rook_mask(int square) {
  return generate_a1a8_mask(square) |
         generate_a1h1_mask(square);
}
```

Generation of File Attacks

```
uint64 generate_a1a8_attack(int square, uint64 blockers) {
  uint64 bitboard;
  int file;
  int rank;
  int r;

  bitboard = 0;

  file = square_file[square];
  rank = square_rank[square];

  for (r = rank + 1; r <= 7; r++) {
    bitboard |= file_rank_to_bitboard(file, r);
    if (blockers & file_rank_to_bitboard(file, r)) break;
  }
  for (r = rank - 1; r >= 0; r--) {
    bitboard |= file_rank_to_bitboard(file, r);
    if (blockers & file_rank_to_bitboard(file, r)) break;
  }
  return bitboard;
}
```

Generation of Rank Attacks

```
uint64 generate_a1h1_attack(int square, uint64 blockers) {
  uint64 bitboard;
  int file;
  int rank;
  int f;

  bitboard = 0;

  file = square_file[square];
  rank = square_rank[square];

  for (f = file + 1; f <= 7; f++) {
    bitboard |= file_rank_to_bitboard(f, rank);
    if (blockers & file_rank_to_bitboard(f, rank)) break;
  }
  for (f = file - 1; f >= 0; f--) {
    bitboard |= file_rank_to_bitboard(f, rank);
    if (blockers & file_rank_to_bitboard(f, rank)) break;
  }
  return bitboard;
}
```

Generation of Diagonal Attacks

```
uint64 generate_a1h8_attack(int square, uint64 blockers) {
  uint64 bitboard;
  int file;
  int rank;
  int f;
  int r;

  bitboard = 0;

  file = square_file[square];
  rank = square_rank[square];

  for (r = rank + 1, f = file + 1; r <= 7 && f <= 7; r++, f++) {
    bitboard |= file_rank_to_bitboard(f, r);
    if (blockers & file_rank_to_bitboard(f, r)) break;
  }
  for (r = rank - 1, f = file - 1; r >= 0 && f >= 0; r--, f--) {
    bitboard |= file_rank_to_bitboard(f, r);
    if (blockers & file_rank_to_bitboard(f, r)) break;
  }
  return bitboard;
}

uint64 generate_h1a8_attack(int square, uint64 blockers) {
  uint64 bitboard;
  int file;
  int rank;
  int f;
  int r;

  bitboard = 0;

  file = square_file[square];
  rank = square_rank[square];

  for (r = rank + 1, f = file - 1; r <= 7 && f >= 0; r++, f--) {
    bitboard |= file_rank_to_bitboard(f, r);
    if (blockers & file_rank_to_bitboard(f, r)) break;
  }
  for (r = rank - 1, f = file + 1; r >= 0 && f <= 7; r--, f++) {
    bitboard |= file_rank_to_bitboard(f, r);
    if (blockers & file_rank_to_bitboard(f, r)) break;
  }
  return bitboard;
}
```

Generation of Bishop and Rook Attacks

```
uint64 generate_bishop_attack(int square, uint64 blockers) {
  return generate_a1h8_attack(square, blockers) |
         generate_h1a8_attack(square, blockers);
}

uint64 generate_rook_attack(int square, uint64 blockers) {
  return generate_a1a8_attack(square, blockers) |
         generate_a1h1_attack(square, blockers);
}
```

Appendix B

Magic Multipliers

In this appendix the magic multiplier sets according to the experiments in Section 3.8 for Bishops and Rooks are given in C/C++ code. Furthermore, the magic multipliers sets are given in hexadecimal notation partitioned into eight blocks for every rank (→ `rank1`, ..., `rank8`).

This appendix contains two sections. In Section B.1 the magic multipliers for Bishops are listed. In Section B.2 the magic multipliers for Rooks are listed.

B.1 Magic Multipliers for Bishops

Below the magic multiplier set with minimal n bits for Bishops is summarised. The appropriate square (→ `a1`, ..., `h8`) is given as a vector index. According to the results in Table 3.18, the number of active bits is added as C/C++ comment.

```
uint64 magic_multiplier_bishop[64];

magic_multiplier_bishop[a1]=0x0040100401004010; // 6 bits
magic_multiplier_bishop[b1]=0x0020080200802000; // 5 bits
magic_multiplier_bishop[c1]=0x0010040080200000; // 4 bits
magic_multiplier_bishop[d1]=0x0004040080000000; // 3 bits
magic_multiplier_bishop[e1]=0x0002021000000000; // 3 bits
magic_multiplier_bishop[f1]=0x0002080404000000; // 4 bits
magic_multiplier_bishop[g1]=0x0004040404040000; // 5 bits
magic_multiplier_bishop[h1]=0x0002020202020200; // 6 bits

magic_multiplier_bishop[a2]=0x0000401004010040; // 5 bits
magic_multiplier_bishop[b2]=0x0000200802008020; // 5 bits
magic_multiplier_bishop[c2]=0x0000100400802000; // 4 bits
magic_multiplier_bishop[d2]=0x0000040400800000; // 3 bits
magic_multiplier_bishop[e2]=0x0000020210000000; // 3 bits
magic_multiplier_bishop[f2]=0x0000020804040000; // 4 bits
magic_multiplier_bishop[g2]=0x0000040404040400; // 5 bits
magic_multiplier_bishop[h2]=0x0000020202020200; // 5 bits

magic_multiplier_bishop[a3]=0x0040002008020080; // 5 bits
magic_multiplier_bishop[b3]=0x0020001004010040; // 5 bits
magic_multiplier_bishop[c3]=0x0010000800802008; // 5 bits
magic_multiplier_bishop[d3]=0x0008000082004000; // 4 bits
```

```
magic_multiplier_bishop[e3]=0x0001000820080000; // 4 bits
magic_multiplier_bishop[f3]=0x0002000101010100; // 5 bits
magic_multiplier_bishop[g3]=0x0004000202020200; // 5 bits
magic_multiplier_bishop[h3]=0x0002000101010100; // 5 bits

magic_multiplier_bishop[a4]=0x0020200010040100; // 5 bits
magic_multiplier_bishop[b4]=0x0010100008020080; // 5 bits
magic_multiplier_bishop[c4]=0x0008080004004010; // 5 bits
magic_multiplier_bishop[d4]=0x0002008008008002; // 5 bits
magic_multiplier_bishop[e4]=0x0000840000802000; // 4 bits
magic_multiplier_bishop[f4]=0x0010010000808080; // 5 bits
magic_multiplier_bishop[g4]=0x0008020001010100; // 5 bits
magic_multiplier_bishop[h4]=0x0004010000808080; // 5 bits

magic_multiplier_bishop[a5]=0x0010101000080200; // 5 bits
magic_multiplier_bishop[b5]=0x0008080800040100; // 5 bits
magic_multiplier_bishop[c5]=0x0004040400020020; // 5 bits
magic_multiplier_bishop[d5]=0x0000020080080080; // 4 bits
magic_multiplier_bishop[e5]=0x0010020080001004; // 4 bits
magic_multiplier_bishop[f5]=0x0020080080004040; // 4 bits
magic_multiplier_bishop[g5]=0x0010040100008080; // 5 bits
magic_multiplier_bishop[h5]=0x0008020080004040; // 5 bits

magic_multiplier_bishop[a6]=0x0008080808000400; // 5 bits
magic_multiplier_bishop[b6]=0x0004040404000200; // 5 bits
magic_multiplier_bishop[c6]=0x0002020202000100; // 5 bits
magic_multiplier_bishop[d6]=0x0000004208000080; // 5 bits
magic_multiplier_bishop[e6]=0x0000080104000040; // 4 bits
magic_multiplier_bishop[f6]=0x0040100400400020; // 4 bits
magic_multiplier_bishop[g6]=0x0020080200800040; // 5 bits
magic_multiplier_bishop[h6]=0x0010040100400020; // 5 bits

magic_multiplier_bishop[a7]=0x0004040404040000; // 5 bits
magic_multiplier_bishop[b7]=0x0002020202020000; // 5 bits
magic_multiplier_bishop[c7]=0x0000020201040000; // 5 bits
magic_multiplier_bishop[d7]=0x0000000042020000; // 4 bits
magic_multiplier_bishop[e7]=0x0000001002020000; // 3 bits
magic_multiplier_bishop[f7]=0x0000401002008000; // 3 bits
magic_multiplier_bishop[g7]=0x0040100401004000; // 4 bits
magic_multiplier_bishop[h7]=0x0020080200802000; // 5 bits

magic_multiplier_bishop[a8]=0x0002020202020200; // 6 bits
magic_multiplier_bishop[b8]=0x0000020202020200; // 5 bits
magic_multiplier_bishop[c8]=0x0000000202010400; // 4 bits
magic_multiplier_bishop[d8]=0x0000000000420200; // 3 bits
magic_multiplier_bishop[e8]=0x0000000010020200; // 3 bits
magic_multiplier_bishop[f8]=0x0000004010020080; // 4 bits
magic_multiplier_bishop[g8]=0x0000401004010040; // 5 bits
magic_multiplier_bishop[h8]=0x0040100401004010; // 6 bits
```

B.2 Magic Multipliers for Rooks

Below the magic multiplier set with minimal n bits for Rooks is summarised. The appropriate square ($\rightarrow$ `a1`, ..., `h8`) is given as a vector index. According to the results in Table 3.19, the number of active bits is added as C/C++ comment.

```
uint64 magic_multiplier_rook[64];
```

```
magic_multiplier_rook[a1]=0x0080004000802010; // 5 bits
magic_multiplier_rook[b1]=0x0040002000100040; // 4 bits
magic_multiplier_rook[c1]=0x0080200010008008; // 5 bits
magic_multiplier_rook[d1]=0x0080100008008004; // 5 bits
magic_multiplier_rook[e1]=0x0080080004008002; // 5 bits
magic_multiplier_rook[f1]=0x0080040002008001; // 5 bits
magic_multiplier_rook[g1]=0x0080020001000080; // 4 bits
magic_multiplier_rook[h1]=0x0080010000402080; // 5 bits

magic_multiplier_rook[a2]=0x0000800040008020; // 4 bits
magic_multiplier_rook[b2]=0x0000400020100040; // 4 bits
magic_multiplier_rook[c2]=0x0000802000100080; // 4 bits
magic_multiplier_rook[d2]=0x0000801000080080; // 4 bits
magic_multiplier_rook[e2]=0x0000800800040080; // 4 bits
magic_multiplier_rook[f2]=0x0000800400020080; // 4 bits
magic_multiplier_rook[g2]=0x0001000200010004; // 4 bits
magic_multiplier_rook[h2]=0x0000800100004080; // 4 bits

magic_multiplier_rook[a3]=0x0000808000400020; // 4 bits
magic_multiplier_rook[b3]=0x0010004000200040; // 4 bits
magic_multiplier_rook[c3]=0x0000808020001000; // 4 bits
magic_multiplier_rook[d3]=0x0000808010000800; // 4 bits
magic_multiplier_rook[e3]=0x0000808008000400; // 4 bits
magic_multiplier_rook[f3]=0x0000808004000200; // 4 bits
magic_multiplier_rook[g3]=0x0000010100020004; // 4 bits
magic_multiplier_rook[h3]=0x0000020004008041; // 5 bits

magic_multiplier_rook[a4]=0x0000400080008020; // 4 bits
magic_multiplier_rook[b4]=0x0000200040100040; // 4 bits
magic_multiplier_rook[c4]=0x0000200080100080; // 4 bits
magic_multiplier_rook[d4]=0x0000100080080080; // 4 bits
magic_multiplier_rook[e4]=0x0000080080040080; // 4 bits
magic_multiplier_rook[f4]=0x0000040080020080; // 4 bits
magic_multiplier_rook[g4]=0x0001000100020004; // 4 bits
magic_multiplier_rook[h4]=0x0000800080004100; // 4 bits

magic_multiplier_rook[a5]=0x0000400080800020; // 4 bits
magic_multiplier_rook[b5]=0x0000201000400040; // 4 bits
magic_multiplier_rook[c5]=0x0000200080801000; // 4 bits
magic_multiplier_rook[d5]=0x0000100080800800; // 4 bits
magic_multiplier_rook[e5]=0x0000080080800400; // 4 bits
magic_multiplier_rook[f5]=0x0000040080800200; // 4 bits
magic_multiplier_rook[g5]=0x0000020001010004; // 4 bits
magic_multiplier_rook[h5]=0x0000800040800100; // 4 bits

magic_multiplier_rook[a6]=0x0000800040008020; // 4 bits
magic_multiplier_rook[b6]=0x0000400020008080; // 4 bits
magic_multiplier_rook[c6]=0x0000200010008080; // 4 bits
magic_multiplier_rook[d6]=0x0000100008008080; // 4 bits
magic_multiplier_rook[e6]=0x0000080004008080; // 4 bits
magic_multiplier_rook[f6]=0x0000040002008080; // 4 bits
magic_multiplier_rook[g6]=0x0001000200010004; // 4 bits
magic_multiplier_rook[h6]=0x0000040080420001; // 5 bits

magic_multiplier_rook[a7]=0x0080004000200040; // 4 bits
magic_multiplier_rook[b7]=0x0000400020100040; // 4 bits
magic_multiplier_rook[c7]=0x0000200010008080; // 4 bits
magic_multiplier_rook[d7]=0x0000100008008080; // 4 bits
magic_multiplier_rook[e7]=0x0000080004008080; // 4 bits
magic_multiplier_rook[f7]=0x0000040002008080; // 4 bits
magic_multiplier_rook[g7]=0x0000800200010080; // 4 bits
magic_multiplier_rook[h7]=0x0000800100004080; // 4 bits
```

```
magic_multiplier_rook[a8]=0x0000800100402011; // 6 bits
magic_multiplier_rook[b8]=0x0000400100802011; // 6 bits
magic_multiplier_rook[c8]=0x0000200100401009; // 6 bits
magic_multiplier_rook[d8]=0x0000100100200805; // 6 bits
magic_multiplier_rook[e8]=0x0001000800100403; // 6 bits
magic_multiplier_rook[f8]=0x0001000400080201; // 5 bits
magic_multiplier_rook[g8]=0x0001000200040081; // 5 bits
magic_multiplier_rook[h8]=0x0001000200804021; // 6 bits
```

Appendix C

Loop at Computer-Chess Tournaments

In this appendix the most important computer-chess tournaments where Loop participated between 2005 and 2008 are summarized. Additionally, in the games headings the opening variation, such as *Sicilian Najdorf* or *Queen's Gambit*, and their respective ECO codes are given. In the end of every game the final position is given. All analyses were carried out with the computer-chess engine Loop.

This appendix contains two sections. In Section C.1 the game data from the 26^{th} Open Dutch Computer-Chess Championship, Leiden (NL) 2006 are given. In Section C.2 the game data from the 15^{th} World Computer-Chess Championship 2007, Amsterdam (NL) are given.

C.1 26^{th} Open Dutch Computer-Chess Championship

The 26^{th} Open Dutch Computer-Chess Championship was held from 3^{rd} to 5^{th} of November 2006, in the Denksportcentrum Leiden. The playing tempo was 90 minutes per computer-chess engine. Clemens Keck was the operator of Loop Leiden and the book designer. Loop Leiden was installed on a dual-core Intel Conroe at 2×3000 MHz.

In Table C.1 the time schedule and the results of Loop Leiden are given. In Table C.2 the tournament results of the 26^{th} Open Dutch Computer-Chess Championship are summarized. Further information about this tournament and the final contingency table can be found at the CSVN website [57]. All games played by Loop Leiden in 9 rounds are chronologically listed below.

Time Schedule and the Results of Loop				
Round	Day	White	Black	Result
1	Friday	Rybka	Loop Leiden	1-0
2	Friday	Loop Leiden	ZZZZZZ	1-0
3	Friday	Loop Leiden	Deep Shredder	1-0
4	Saturday	The Baron	Loop Leiden	0-1
5	Saturday	Loop Leiden	Hiarcs X MP	0.5-0.5
6	Saturday	Fruit	Loop Leiden	0.5-0.5
7	Sunday	The King	Loop Leiden	0-1
8	Sunday	Loop Leiden	Deep Gandalf	1-0
9	Sunday	IsiChess MMX	Loop Leiden	0-1

Table C.1: Time schedule and the results of Loop at the 26^{th} Open Dutch Computer-Chess Championship 2006, Leiden (NL).

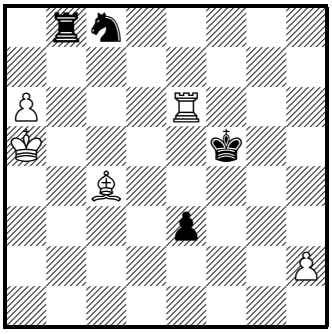
Tournament Result				
Rank	Program	Country	Games	Score
1	Rybka	Hungary	9	9.0
2	Loop Leiden	Germany	9	7.0
3	Hiarcs X MP	UK	9	6.5
4	Fruit	France	9	6.0
5	Deep Gandalf	Denmark	9	5.0
6	Deep Sjeng	Belgium	8	5.0
7	The Baron	Netherlands	8	5.0
8	The King	Netherlands	9	4.5
9	Deep Shredder	Germany	8	4.5
10	IsiChess MMX	Germany	8	4.5
11	XiniX	Netherlands	8	4.5
12	ANT	Netherlands	8	4.0
13	Hermann	Germany	8	3.5
14	Zzzzzz	Netherlands	8	2.0
15	Joker	Netherlands	8	1.0

Table C.2: Tournament result of the 26^{th} Open Dutch Computer-Chess Championship 2006, Leiden (NL).

Round 1: RYBKA - LOOP LEIDEN, *Sicilian Najdorf* (B90): Unusual White 6th moves, 6 ♗e3 ♘g4 and 6 ♗e3 e5

1 e4 c5 2 ♘f3 d6 3 d4 c×d4 4 ♘×d4 ♘f6 5 ♘c3 a6 6 ♗e3 e5 7 ♘b3 ♗e7 8 f3 ♗e6 9 ♕d2 O-O 10 O-O-O ♘bd7 11 g4 b5 12 g5 b4 13 ♘e2 ♘e8 14 f4 a5 15 f5 a4 16 ♘bd4 e×d4 17 ♘×d4 b3 18 ♔b1 b×c2+ 19 ♘×c2 ♗b3 20 a×b3 a×b3 21 ♘a3 ♘e5 22 ♕g2 ♕b8 23 f6 ♗d8 24 ♖d5 ♖a5 25 ♗e2 ♖×d5 26 e×d5 ♗b6 27 ♗f4 ♗c5 28 f×g7 ♔×g7 29 ♕g3 ♕a8 30 ♕×b3 ♗×a3 31 ♗×e5+ d×e5 32 ♕×a3 ♕×a3 33 b×a3 ♘d6 34 ♖c1 ♖d8 35 a4 ♘e4 36 ♗b5 ♖×d5 37 ♗c6 ♘d2+ 38 ♔b2 ♖d4 39 ♔c3 ♘c4 40 ♔b4 ♘d2+ 41 ♔b5 ♖d8 42 a5 ♘b3 43 ♖c4 ♖b8+ 44 ♔a4 ♘d4 45 ♗d5 ♖b1 46 ♖c3 f5 47 g×f6+ ♔×f6 48 a6 ♖b6 49 ♗c4 ♖b8 50 ♖h3 h5 51 ♖×h5 e4 52 ♖d5 ♘f5 53 ♔a5 e3 54 ♖d1 ♘e7 55 ♖d6+ ♔f5 56 ♖e6 ♘c8

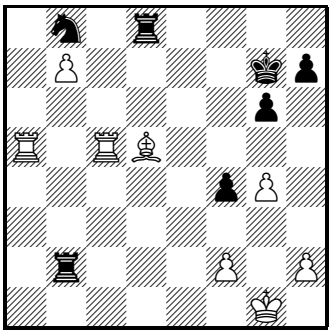
Final position: LOOP LEIDEN resigns → 1-0



Round 2: LOOP LEIDEN - ZZZZZZ

1 d4 ♘f6 2 c4 g6 3 g3 d5 4 c×d5 ♕×d5 5 ♘f3 ♗g7 6 ♘c3 ♕a5 7 ♗d2 ♕b6 8 e4 c6 9 ♗e2 ♗g4 10 ♘a4 ♕d8 11 e5 ♘e4 12 O-O b5 13 ♘c3 ♘×d2 14 ♕×d2 O-O 15 ♖fd1 f6 16 e×f6 e×f6 17 d5 b4 18 ♘a4 ♕a5 19 a3 ♕×a4 20 a×b4 ♕b3 21 ♘d4 ♕×d5 22 ♗×g4 f5 23 ♗f3 ♕×d4 24 ♕×d4 ♗×d4 25 ♖×d4 a5 26 b5 c5 27 ♖c4 ♖a7 28 b6 ♖a6 29 b7 f4 30 g4 ♖b6 31 ♖×c5 ♖×b2 32 ♗d5+ ♔g7 33 ♖a×a5 ♖d8

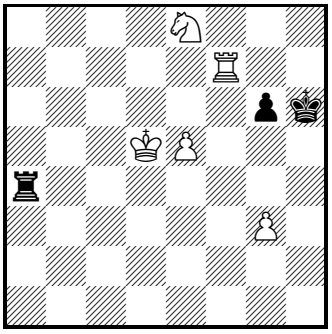
Final position: ZZZZZZ resigns → 1-0



Round 3: Loop Leiden - Deep Shredder, *Larsen Opening* (A01)

1 b3 e5 2 ♘c3 d5 3 d4 ♗b4 4 ♗b2 ♘f6 5 a3 e×d4 6 a×b4 d×c3 7 ♗×c3 O-O 8 e3 ♘e4 9 ♗b2 ♕h4 10 g3 ♕e7 11 ♕d4 ♘f6 12 b5 ♖e8 13 ♗d3 c5 14 ♕h4 ♘bd7 15 ♘f3 c4 16 ♗×f6 ♘×f6 17 b×c4 d×c4 18 ♕×c4 ♗h3 19 ♔e2 ♗g4 20 h3 ♗h5 21 ♖hd1 ♗g6 22 ♗×g6 h×g6 23 ♖d4 ♘e4 24 ♔f1 ♖ac8 25 ♕b3 ♖c5 26 ♘d2 ♘c3 27 ♖×a7 ♖ec8 28 ♖d3 ♕c7 29 e4 ♘×b5 30 ♖a4 ♕e7 31 h4 g5 32 h×g5 ♕×g5 33 ♖c4 ♖×c4 34 ♘×c4 ♕c5 35 ♘e3 ♕c6 36 ♕b4 ♖e8 37 f3 ♖a8 38 ♔g2 g6 39 ♖b3 ♘d6 40 ♖c3 ♕a6 41 ♖d3 ♘e8 42 ♔f2 ♖c8 43 ♖d7 b5 44 ♔e2 ♕e6 45 ♕×b5 ♘d6 46 ♕a4 ♔h7 47 ♕a7 ♔g8 48 ♔f2 ♕f6 49 ♔g2 ♔g7 50 ♘g4 ♖×c2+ 51 ♔h3 ♕×f3 52 ♕d4+ ♔h7 53 ♘f6+ ♔h6 54 ♖×d6 ♕g2+ 55 ♔g4 ♕e2+ 56 ♔f4 ♕f1+ 57 ♔e5 ♕b5+ 58 ♘d5 ♖c4 59 ♕d2+ ♔g7 60 ♕e2 ♕c5 61 ♖d7 ♕d4+ 62 ♔f4 ♖c1 63 ♖b7 ♖c5 64 ♖b4 ♕a1 65 ♕b2+ ♕×b2 66 ♖×b2 ♖c1 67 ♖b7 ♖c4 68 ♘c7 ♖c6 69 e5 ♔h6 70 ♘e8 ♖c4+ 71 ♔e3 ♖c5 72 ♔d4 ♖a5 73 ♖×f7 ♖a4+ 74 ♔d5

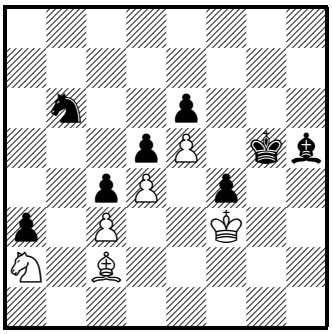
Final position: Deep Shredder resigns → 1-0



Round 4: The Baron - Loop Leiden, *Nimzowitsch-Larsen Opening* (A01)

1 d4 ♘f6 2 ♘f3 e6 3 ♗f4 d5 4 e3 c5 5 c3 ♘c6 6 ♘bd2 ♗d6 7 ♗g3 O-O 8 ♗d3 ♕e7 9 ♘e5 ♘d7 10 f4 f5 11 O-O ♕d8 12 ♘df3 ♘d×e5 13 f×e5 ♗e7 14 a3 c4 15 ♗e2 ♗d7 16 ♗f4 ♘a5 17 ♕e1 ♔h8 18 ♕g3 h6 19 ♕f2 ♕b6 20 ♗d1 g5 21 ♗g3 ♖g8 22 ♕c2 h5 23 ♘d2 h4 24 ♗f2 g4 25 g3 h×g3 26 ♗×g3 ♗g5 27 ♗f4 ♗×f4 28 e×f4 ♕d8 29 ♖f2 g3 30 h×g3 ♖×g3+ 31 ♔f1 ♕h4 32 ♗f3 ♖ag8 33 ♔e2 ♖h3 34 ♖af1 ♕×f4 35 ♕d1 ♕h6 36 ♖h1 ♖gg3 37 ♖×h3 ♖×h3 38 ♖g2 ♗b5 39 a4 ♖h1 40 ♖g8+ ♔×g8 41 ♕×h1 ♕×h1 42 ♗×h1 ♗×a4 43 ♘f3 ♗e8 44 ♘g5 ♗f7 45 ♔e3 ♘c6 46 ♘h3 ♔g7 47 ♘f4 ♔h6 48 ♗f3 ♘e7 49 ♗d1 b5 50 ♘h3 a5 51 ♔d2 b4 52 ♗a4 b×c3+ 53 b×c3 ♔h5 54 ♘g1 f4 55 ♔e2 ♘c8 56 ♔f3 ♘b6 57 ♗c2 a4 58 ♘e2 a3 59 ♘c1 ♔g5 60 ♘a2 ♗h5+

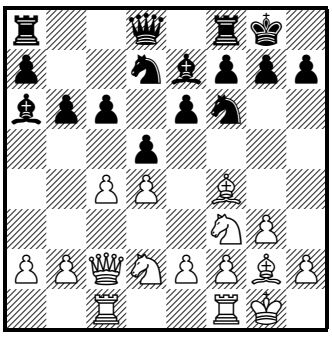
Final position: The Baron resigns → 0-1



Round 5: Loop Leiden - Hiarcs X MP, *Closed Catalan* (E06): Early deviations

1 d4 d5 2 c4 c6 3 ♘f3 e6 4 ♕c2 ♘f6 5 g3 ♗e7 6 ♗g2 O-O 7 O-O b6 8 ♗f4 ♗a6 9 ♘bd2 ♘bd7 10 ♖ac1 ♘h5 11 ♗e3 ♘hf6 12 ♗f4 ♘h5 13 ♗e3 ♘hf6 14 ♗f4

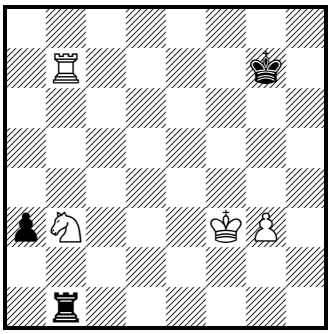
Final position: Drawn by threefold repetition → 0.5-0.5



Round 6: Fruit - Loop Leiden, *Semi-Slav: Meran System* (D48): 7...b5 8 ♗d3 a6

1 d4 ♘f6 2 c4 e6 3 ♘f3 d5 4 ♘c3 c6 5 e3 ♘bd7 6 ♗d3 d×c4 7 ♗×c4 b5 8 ♗d3 a6 9 e4 c5 10 d5 c4 11 ♗c2 ♕c7 12 d×e6 f×e6 13 O-O ♗b7 14 ♘g5 ♘c5 15 e5 ♕×e5 16 ♖e1 ♕d6 17 ♕×d6 ♗×d6 18 ♗e3 ♘d3 19 ♗×d3 c×d3 20 ♖ad1 b4 21 ♘a4 ♘g4 22 h3 ♗h2+ 23 ♔f1 ♘×e3+ 24 ♖×e3 O-O 25 g3 ♖ac8 26 ♘×e6 ♖f3 27 ♖d×d3 ♖c1+ 28 ♔e2 ♖×e3+ 29 ♖×e3 ♗c6 30 ♘ac5 ♖c2+ 31 ♔e1 ♖×b2 32 ♘d3 ♖b1+ 33 ♔d2 ♗g1 34 ♘d4 ♗d5 35 ♘f5 ♔f8 36 ♘d6 g6 37 ♖e8+ ♔g7 38 ♖e7+ ♔g8 39 h4 ♔f8 40 ♖c7 ♗e6 41 ♘e4 ♗f5 42 ♔e3 a5 43 h5 g×h5 44 ♖a7 ♗×e4 45 ♔×e4 ♗×f2 46 ♘×f2 ♖b2 47 ♘d3 ♖×a2 48 ♖×h7 b3 49 ♖×h5 a4 50 ♖b5 ♖g2 51 ♖b8+ ♔e7 52 ♔f3 ♖g1 53 ♘c5 a3 54 ♖b7+ ♔f6 55 ♖b6+ ♔g7 56 ♘×b3 ♖b1 57 ♖b7+

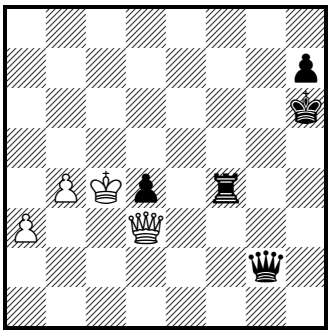
Final position: Insufficient material → 0.5-0.5



Round 7: The King - Loop Leiden, *Sicilian Najdorf* (B90): Unusual White 6th moves, 6 ♗e3 ♘g4 and 6 ♗e3 e5

1 e4 c5 2 ♘f3 d6 3 d4 c×d4 4 ♘×d4 ♘f6 5 ♘c3 a6 6 ♗e3 e5 7 ♘b3 ♗e7 8 f3 ♗e6 9 ♕d2 O-O 10 O-O-O ♘bd7 11 g4 b5 12 g5 ♘h5 13 ♘d5 ♗×d5 14 e×d5 ♕c7 15 ♔b1 ♘b6 16 ♘a5 ♘×d5 17 ♕×d5 ♕×a5 18 c4 ♖ab8 19 ♗d3 ♕d8 20 ♖hg1 b×c4 21 ♗×c4 a5 22 ♖g2 g6 23 ♕e4 ♔h8 24 f4 f6 25 ♖e2 ♕c8 26 ♖c2 ♕g4 27 ♗e2 ♕h3 28 f×e5 f×e5 29 ♗a7 ♖b4 30 ♕d5 a4 31 ♕a5 ♕h4 32 a3 ♖e4 33 ♕d5 ♘g7 34 ♗b5 ♖g4 35 ♗d7 ♖×g5 36 ♕c6 ♘f5 37 ♕×a4 ♖g4 38 ♕b3 ♘d4 39 ♕a4 ♖gf4 40 ♗×d4 e×d4 41 ♕c4 ♗f6 42 ♗c6 ♗g7 43 ♖g1 ♖4f5 44 ♕d3 ♗h6 45 ♗e4 ♖e5 46 ♗c6 ♖e3 47 ♕c4 ♕f6 48 ♖d1 d3 49 ♖g2 ♖c8 50 ♕b5 ♖e7 51 ♔a2 d2 52 ♗d5 ♖e1 53 ♖g×d2 ♗×d2 54 ♖×d2 ♕f4 55 ♕b7 ♖f8 56 ♖c2 ♕f5 57 ♕c6 ♕e5 58 ♖d2 ♖b8 59 ♕c2 ♖f1 60 ♕d3 ♕f6 61 ♕c4 ♖f8 62 h3 ♖f4 63 ♕d3 ♖e8 64 ♗c6 ♖d8 65 ♗d5 ♕e5 66 b4 ♖c8 67 ♔b3 ♖c1 68 ♗g2 ♕a1 69 ♖b2 ♔g7 70 ♗e4 ♖e1 71 ♕c3+ ♔h6 72 ♗f3 ♖b1 73 ♖×b1 ♕×b1+ 74 ♔a4 ♕f5 75 ♗g2 d5 76 ♕c5 ♕d7+ 77 ♔b3 d4 78 ♕c1 ♕e6+ 79 ♔a4 g5 80 h4 ♖×h4 81 ♕f1 ♖f4 82 ♕d3 g4 83 ♔b5 ♕e3 84 ♔c4 g3 85 ♗c6 ♕c1+ 86 ♔b5 ♕f1 87 ♔c4 g2 88 ♗×g2 ♕×g2

Final position: The King resigns → 0-1

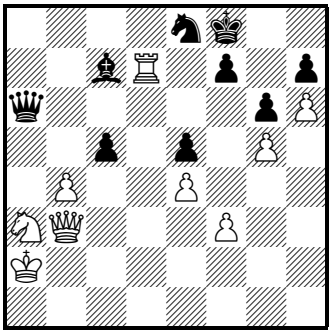


Round 8: Loop Leiden - Deep Gandalf, *Sicilian Najdorf* (B90): Unusual White 6th moves, 6 ♗e3 ♘g4 and 6 ♗e3 e5

1 e4 c5 2 ♘f3 d6 3 d4 c×d4 4 ♘×d4 ♘f6 5 ♘c3 a6 6 ♗e3 e5 7 ♘b3 ♗e6 8 f3 ♗e7 9 ♕d2 O-O 10 O-O-O a5 11 ♗b5 ♘c6 12 ♕e2 ♕c7

13 ♔b1 ♘a7 14 ♗a4 b5 15 ♗×b5 ♘×b5 16 ♘×b5 ♕b8 17 c4 ♖c8 18 ♖c1 ♖c6 19 ♘d2 ♗×c4 20 ♘×c4 ♕×b5 21 ♖he1 ♘e8 22 ♖c2 ♖b8 23 ♘a3 ♕b7 24 ♖×c6 ♕×c6 25 ♖c1 ♕b7 26 ♘c4 ♗d8 27 ♖d1 ♕c6 28 ♕d3 ♗c7 29 g4 ♖b4 30 ♖c1 ♕b7 31 ♖c2 ♕b8 32 h4 a4 33 h5 ♖b5 34 h6 g6 35 g5 ♗d8 36 ♔a1 ♖b4 37 ♕a3 ♕b7 38 ♗d2 ♖b5 39 ♕×a4 ♔f8 40 ♗e3 ♗e7 41 a3 ♗d8 42 b4 ♔e7 43 ♖d2 ♗c7 44 ♖d3 ♕c6 45 ♕b3 ♕b7 46 ♔a2 ♔f8 47 ♗c5 d×c5 48 ♖d7 ♖×b4 49 a×b4 ♕a6+ 50 ♘a3

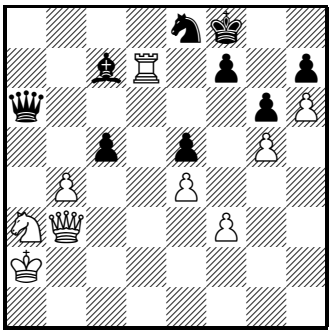
Final position: DEEP GANDALF resigns → 1-0



Round 9: ISICHESS MMX - LOOP LEIDEN, *Semi-Slav* (D46): 5 e3 ♘bd7 6 ♗d3, Black avoid the *Meran*

1 e4 c5 2 ♘f3 d6 3 d4 c×d4 4 ♘×d4 ♘f6 5 ♘c3 a6 6 ♗e3 e5 7 ♘b3 ♗e6 8 f3 ♗e7 9 ♕d2 O-O 10 O-O-O a5 11 ♗b5 ♘c6 12 ♕e2 ♕c7 13 ♔b1 ♘a7 14 ♗a4 b5 15 ♗×b5 ♘×b5 16 ♘×b5 ♕b8 17 c4 ♖c8 18 ♖c1 ♖c6 19 ♘d2 ♗×c4 20 ♘×c4 ♕×b5 21 ♖he1 ♘e8 22 ♖c2 ♖b8 23 ♘a3 ♕b7 24 ♖×c6 ♕×c6 25 ♖c1 ♕b7 26 ♘c4 ♗d8 27 ♖d1 ♕c6 28 ♕d3 ♗c7 29 g4 ♖b4 30 ♖c1 ♕b7 31 ♖c2 ♕b8 32 h4 a4 33 h5 ♖b5 34 h6 g6 35 g5 ♗d8 36 ♔a1 ♖b4 37 ♕a3 ♕b7 38 ♗d2 ♖b5 39 ♕×a4 ♔f8 40 ♗e3 ♗e7 41 a3 ♗d8 42 b4 ♔e7 43 ♖d2 ♗c7 44 ♖d3 ♕c6 45 ♕b3 ♕b7 46 ♔a2 ♔f8 47 ♗c5 d×c5 48 ♖d7 ♖×b4 49 a×b4 ♕a6+ 50 ♘a3

Final position: ISICHESS MMX resigns → 0-1



C.2 15^{th} World Computer-Chess Championship

The 15^{th} World Computer-Chess Championship was held from 11^{th} to 18^{th} of June 2007, in the Science Park Amsterdam (NL). The organiser was the International Computer Games Association (ICGA). The event together with computer games workshops, was sponsored by IBM, SARA Computing and Networking Services, and NCF (Foundation of National Computing Facilities).[1] Twelve programs took part in the round robin event. The playing tempo was 90 minutes per computer-chess engine. Clemens Keck was the operator of LOOP AMSTERDAM and the book designer. LOOP AMSTERDAM was installed on a quad-core Intel Woodcrest at 4×3000 MHz.

In Table C.3 the time schedule and the results of LOOP AMSTERDAM are given. In Table C.4 the tournament results of the 15^{th} World Computer-Chess Championship are summarized. Further information about this tournament and the final contingency table can be found at the ICGA website [54]. All games played by LOOP AMSTERDAM in 11 rounds are chronologically listed below.

Time Schedule and the Results of LOOP				
Round	Day	White	Black	Result
1	Monday	GRIDCHESS	LOOP AMSTERDAM	0.5-0.5
2	Tuesday	LOOP AMSTERDAM	SHREDDER	0-1
3	Tuesday	THE KING	LOOP AMSTERDAM	0-1
4	Wednesday	LOOP AMSTERDAM	RYBKA	0.5-0.5
5	Wednesday	ZAPPA	LOOP AMSTERDAM	1-0
6	Friday	MICRO-MAX	LOOP AMSTERDAM	0-1
7	Saturday	LOOP AMSTERDAM	JONNY	1-0
8	Saturday	ISICHESS	LOOP AMSTERDAM	0-1
9	Sunday	LOOP AMSTERDAM	THE BARON	0.5-0.5
10	Sunday	DEEP SJENG	LOOP AMSTERDAM	0-1
11	Monday	LOOP AMSTERDAM	DIEP	1-0

Table C.3: Time schedule and the results of LOOP at the 15^{th} World Computer-Chess Championship 2007, Amsterdam (NL).

[1] For more information see: `http://www.chessbase.com/newsdetail.asp?newsid=3936`

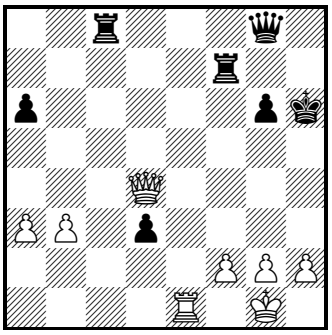
Result of the 15^{th} World Computer-Chess Championship				
Rank	Program	Country	Games	Score
1	Rybka	USA	11	10.0
2	Zappa	Turkey	11	9.0
3	Loop Amsterdam	Germany	11	7.5
4	Shredder	Germany	11	7.0
5	GridChess	Germany	11	7.0
6	Deep Sjeng	Belgium	11	6.0
7	Jonny	Germany	11	5.0
8	Diep	Netherlands	11	4.5
9	The Baron	Netherlands	11	4.0
10	IsiChess	Germany	11	3.5
11	The King	Netherlands	11	2.5
12	micro-Max	Netherlands	11	0.0

Table C.4: Tournament result of the 15^{th} World Computer-Chess Championship 2007, Amsterdam (NL).

Round 1: GridChess - Loop Amsterdam, *Queen's Gambit Declined* (D30): System without ♘c3

1 d4 d5 2 c4 c6 3 ♘f3 e6 4 ♕c2 ♘f6 5 e3 a6 6 ♘bd2 ♘bd7 7 b3 c5 8 ♗d3 b6 9 ♗b2 ♗b7 10 O-O ♖c8 11 c×d5 ♘×d5 12 ♕b1 c×d4 13 ♗×d4 ♗b4 14 a3 ♗×d2 15 ♘×d2 ♘c5 16 ♗e4 ♘×e4 17 ♕×e4 ♕e7 18 ♕e5 f6 19 ♕h5+ g6 20 ♕h6 e5 21 e4 e×d4 22 e×d5 ♔f7 23 ♘c4 ♗×d5 24 ♘×b6 ♖cd8 25 ♖fe1 ♕b7 26 ♘×d5 ♕×d5 27 ♖ac1 d3 28 ♖c7+ ♖d7 29 ♖c8 ♖×c8 30 ♕×h7+ ♔f8 31 ♕h8+ ♕g8 32 ♕×f6+ ♕f7 33 ♕h8+ ♕g8 34 ♕f6+ ♖f7 35 ♕d6+ ♔g7 36 ♕d4+ ♔h7 37 ♕h4+ ♔g7 38 ♕d4+ ♔h6

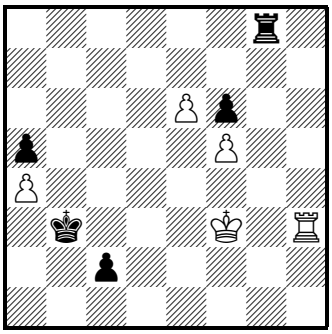
Final position: Drawn by threefold repetition → 0.5-0.5



Round 2: Loop Amsterdam - Shredder, *Queen's Gambit Declined* (D53): 4 ♗g5 ♗e7, Early deviations

1 d4 ♘f6 2 c4 e6 3 ♘c3 d5 4 ♘f3 ♗e7 5 ♗g5 h6 6 ♗×f6 ♗×f6 7 c×d5 e×d5 8 ♕b3 c6 9 e3 ♘d7 10 ♗d3 O-O 11 O-O ♖e8 12 ♖ae1 ♘b6 13 h3 ♕c7 14 ♖c1 ♗d8 15 ♖fe1 ♗e6 16 ♕c2 ♗e7 17 ♖e2 ♖ac8 18 ♖ce1 ♗f8 19 e4 d×e4 20 ♗×e4 ♕d7 21 ♗d3 ♖cd8 22 a3 ♕c8 23 b4 ♘d5 24 ♘×d5 ♗×d5 25 ♘e5 ♕c7 26 ♗h7+ ♔h8 27 ♗e4 ♔g8 28 h4 ♗×e4 29 ♖×e4 ♕d6 30 g3 ♕d5 31 ♘c4 ♖xe4 32 ♖×e4 ♕f5 33 ♕e2 g6 34 ♘a5 ♕d7 35 h5 g5 36 ♔g2 b6 37 ♘c4 ♗g7 38 ♘e5 ♕d5 39 ♕f3 ♗×e5 40 d×e5 ♖d7 41 ♕e2 c5 42 ♔h2 ♔f8 43 ♖c4 ♖e7 44 f4 g×f4 45 g×f4 ♕d8 46 b×c5 ♖d7 47 ♖c2 ♕h4+ 48 ♔g2 b×c5 49 ♖d2 ♖b7 50 ♕f2 ♕g4+ 51 ♕g3 ♕×h5 52 ♕f3 ♕×f3+ 53 ♔×f3 ♖c7 54 ♖h2 ♔g7 55 ♖g2+ ♔h7 56 ♖c2 c4 57 ♖c3 ♔g7 58 a4 h5 59 f5 ♔f8 60 ♔f4 ♔e7 61 ♔e3 ♖c8 62 ♔f3 f6 63 e6 ♔d6 64 ♔e4 a5 65 ♔d4 h4 66 ♔e4 ♖g8 67 ♔f4 ♔d5 68 ♖c2 ♔d4 69 ♖d2+ ♔c3 70 ♖d7 h3 71 ♔f3 ♔b2 72 ♖d2+ ♔b3 73 ♖h2 c3 74 ♖×h3 c2

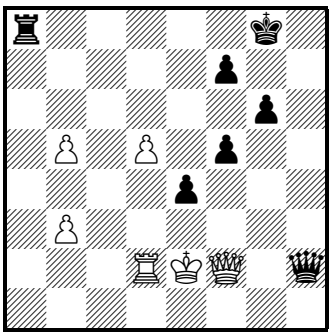
Final position: Loop Amsterdam resigns → 0-1



Round 3: The King - Loop Amsterdam, *Sicilian Najdorf* (B90): Unusual White 6th moves, 6 ♗e3 ♘g4 and 6 ♗e3 e5

1 e4 c5 2 ♘f3 d6 3 d4 c×d4 4 ♘×d4 ♘f6 5 ♘c3 a6 6 ♗e3 e5 7 ♘b3 ♗e6 8 ♕d2 ♗e7 9 f3 O-O 10 O-O-O ♘bd7 11 g4 b5 12 g5 b4 13 ♘e2 ♘e8 14 f4 a5 15 f5 ♗×b3 16 c×b3 a4 17 b×a4 ♖×a4 18 b3 ♖a5 19 ♔b1 d5 20 e×d5 ♗c5 21 ♗×c5 ♘×c5 22 ♕×b4 ♘d6 23 ♘c3 ♕a8 24 ♗b5 ♖c8 25 ♗c6 ♕a7 26 a4 ♖b8 27 ♗b5 ♘×b5 28 ♘×b5 ♖a×b5 29 a×b5 ♖a8 30 ♔c1 ♘d3+ 31 ♖×d3 ♕a1+ 32 ♔d2 ♕×h1 33 ♕c3 g6 34 ♔e3 ♕g1+ 35 ♔f3 ♕f1+ 36 ♔e3 ♕f4+ 37 ♔e2 g×f5 38 ♖d2 ♕×h2+ 39 ♔d1 ♕g1+ 40 ♔e2 e4 41 g6 h×g6 42 ♕d4 ♕h2+ 43 ♕f2

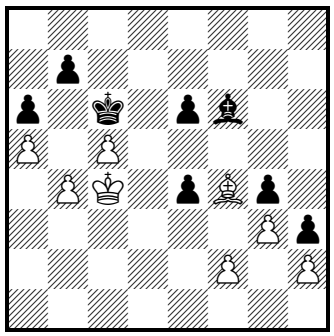
Final position: The King resigns → 0-1



Round 4: Loop Amsterdam - Rybka, *Sicilian* (B51): *Moscow Variation* (3 ♗b5+) without 3. . . ♗d7

1 e4 c5 2 ♘f3 d6 3 ♗b5+ ♘c6 4 O-O ♗d7 5 ♖e1 ♘f6 6 c3 a6 7 ♗a4 c4 8 d4 c×d3 9 ♕×d3 g6 10 ♘d4 ♘e5 11 ♗×d7+ ♕×d7 12 ♕c2 ♗g7 13 ♘d2 O-O 14 ♘2f3 ♖fd8 15 ♘×e5 d×e5 16 ♘f3 ♘e8 17 ♗g5 ♘d6 18 ♖ad1 ♕c7 19 ♖d3 ♘c4 20 ♖ed1 ♖×d3 21 ♖×d3 h6 22 ♗e3 f5 23 ♖d1 e6 24 ♗c1 ♖d8 25 ♖×d8+ ♕×d8 26 ♘d2 ♘×d2 27 ♕×d2 ♕×d2 28 ♗×d2 f×e4 29 ♔f1 ♔f7 30 ♔e2 ♔f6 31 c4 h5 32 b4 ♗f8 33 c5 g5 34 a4 g4 35 a5 ♔e7 36 ♗g5+ ♔d7 37 ♔e3 ♗g7 38 ♔×e4 ♔c7 39 ♔d3 e4+ 40 ♔c4 ♗e5 41 g3 ♔c6 42 ♗e3 ♗f6 43 ♗f4 h4 44 ♗d2 h3 45 ♗f4

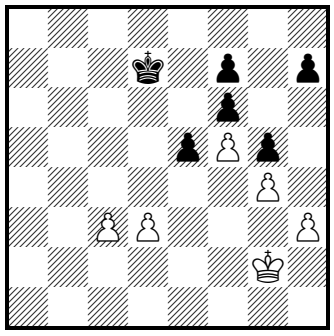
Final position: Drawn by threefold repetition → 0.5-0.5



Round 5: ZAPPA - LOOP AMSTERDAM, *Closed Sicilian* (B25): 3 g3 lines without early ♗e3

1 e4 c5 2 ♘c3 ♘c6 3 g3 g6 4 ♗g2 ♗g7 5 h3 ♘f6 6 d3 d6 7 ♘ge2 ♖b8 8 a4 ♗d7 9 O-O O-O 10 f4 ♔h8 11 ♗e3 ♕a5 12 f5 ♘g8 13 ♔h1 ♗e5 14 g4 ♘f6 15 ♗f4 ♘d4 16 ♗×e5 d×e5 17 ♘g3 ♕b6 18 ♖a2 ♕d6 19 ♕d2 g5 20 ♘d5 ♖g8 21 b3 ♖bd8 22 ♖aa1 ♗c6 23 ♘×f6 ♕×f6 24 ♘h5 ♕d6 25 ♖ab1 ♗e8 26 f6 ♖g6 27 b4 b6 28 a5 e×f6 29 b×c5 b×c5 30 ♖b7 ♖d7 31 ♖fb1 ♘c6 32 ♖7b5 ♘d4 33 ♖b8 ♖d8 34 ♖×d8 ♕×d8 35 ♖b7 ♘c6 36 ♕c3 ♘×a5 37 ♖×a7 ♘c6 38 ♖b7 ♕d4 39 ♕a3 ♕d6 40 ♕a6 ♖g8 41 ♕b6 ♖g6 42 ♕c7 ♕×c7 43 ♖×c7 ♔g8 44 ♘g3 ♔f8 45 c3 ♘e7 46 ♖×c5 ♗d7 47 ♘f5 ♘×f5 48 e×f5 ♖g8 49 ♖c7 ♔e7 50 ♗c6 ♖d8 51 ♔g2 ♔d6 52 ♖×d7+ ♖×d7 53 ♗×d7 ♔×d7

Final position: LOOP AMSTERDAM resigns → 1-0

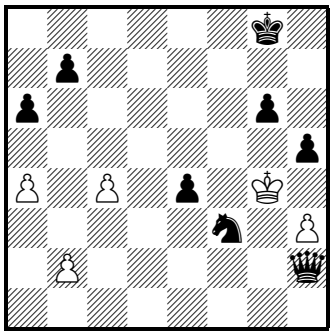


Round 6: MICRO-MAX - LOOP AMSTERDAM, *Sicilian* (B30): 2. . . ♘c6 3 ♗b5, lines without . . . g6

1 e4 c5 2 ♘c3 ♘c6 3 ♘f3 e5 4 ♗c4 ♗e7 5 ♘d5 ♘f6 6 ♘×e7 ♕×e7 7 ♕e2 O-O 8 O-O d6 9 c3 ♗e6 10 ♗×e6 ♕×e6 11 d3 h6 12 ♗e3 c4 13 h3 c×d3 14 ♕×d3 d5 15 e×d5 ♘×d5 16 ♕b5 a6 17 ♕c4 ♖ac8 18 ♗d2 ♖fd8 19 ♕e4 ♘f6 20 ♕e2 e4 21 ♘h2 ♘e5 22 ♖fe1 ♘d3 23 ♖eb1 ♘d5 24 ♘f1 f5 25 ♘g3 ♘f6 26 ♘f1 f4 27 a4 f3 28 g×f3 ♘e5 29 f4 ♘f3+ 30 ♔g2 ♖c5 31 ♘g3 ♖×d2 32 ♕e3 ♘h4+ 33 ♔h2 ♖h5 34 ♘×h5 ♘f3+ 35

♔g2 ♘×h5 36 ♖h1 ♕g6+ 37 ♔f1 ♘g3+ 38 f×g3 ♕×g3 39 ♕e2 ♕×f4 40 ♕×d2 ♕×d2 41 ♖d1 ♕×d1+ 42 ♔f2 ♕×h1 43 ♔g3 ♕h2+ 44 ♔g4 g6 45 c4 h5

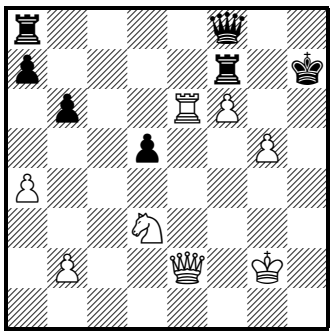
Final position: MICRO-MAX is mated → 0-1



Round 7: LOOP AMSTERDAM - JONNY, *Scotch Game* (C45)

1 e4 e5 2 ♘f3 ♘c6 3 d4 e×d4 4 ♘×d4 ♗c5 5 ♗e3 ♕f6 6 c3 ♕g6 7 ♘d2 ♘×d4 8 c×d4 ♗b6 9 h4 h5 10 a4 ♗a5 11 ♖c1 ♘f6 12 f3 d5 13 e5 ♘g8 14 ♔f2 c6 15 ♘b3 ♗c7 16 ♗d3 ♗f5 17 ♗×f5 ♕×f5 18 ♘c5 ♘e7 19 ♖e1 O-O 20 ♗g5 ♘g6 21 ♔g1 f6 22 g4 h×g4 23 f×g4 ♕c8 24 e×f6 g×f6 25 ♖e6 ♖f7 26 ♕e2 b6 27 ♘d3 ♗e5 28 ♖c×c6 ♗×d4+ 29 ♔g2 ♕f8 30 ♖cd6 f×g5 31 ♖×g6+ ♔h7 32 ♖de6 ♗f6 33 h×g5 ♔×g6 34 g×f6 ♔h7 35 g5

Final position: JONNY resigns → 1-0

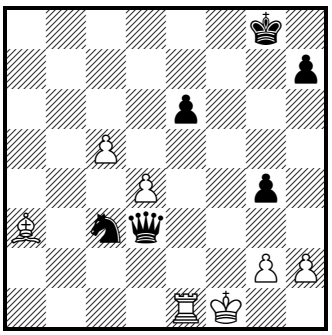


Round 8: ISICHESS - LOOP AMSTERDAM, *Queen's Gambit Declined* (D31): *Semi-Slav* without ... ♘f6

1 d4 d5 2 c4 e6 3 ♘c3 c6 4 ♘f3 d×c4 5 a4 ♗b4 6 e3 b5 7 ♗d2 a5 8 a×b5 ♗×c3 9 ♗×c3 c×b5 10 b3 ♗b7 11 b×c4 b4 12 ♗b2 ♘f6 13 ♗d3 ♘bd7 14 O-O O-O 15 ♖e1 ♘e4 16 ♕c2 f5 17 c5 ♗c6 18 ♗c4 ♕e7 19 ♗b3 g5 20 ♖ed1 g4 21 ♘e1 ♘df6 22 ♖a2 ♗d5 23 ♖da1 ♕c7 24 ♘d3 ♗×b3 25 ♕×b3 ♘d5 26 ♘e5 ♖a7 27 ♘c4 f4 28 ♕d3 f×e3 29 f×e3 ♕f7 30 ♖f1 ♕×f1+ 31 ♕×f1 ♖×f1+ 32 ♔×f1 a4 33 ♖a1 a3 34 ♗c1 ♘ec3 35

♘d6 ♘×e3+ 36 ♗×e3 b3 37 ♘c4 b2 38 ♖e1 ♖a4 39 ♘×a3 ♖×a3 40 ♗c1 b1♕ 41 ♗×a3 ♕d3+

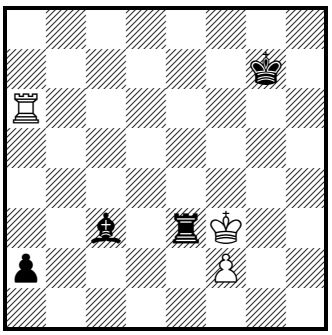
Final position: IsiChess resigns → 0-1



Round 9: Loop Amsterdam - The Baron, *French* (C13): Classical System: 4 ♗g5 ♗e7, *Alekhine-Chatard Attack*

1 e4 e6 2 d4 d5 3 ♘c3 ♘f6 4 ♗g5 d×e4 5 ♘×e4 ♗e7 6 ♗×f6 ♗×f6 7 ♘f3 O-O 8 ♗c4 ♘c6 9 c3 e5 10 d5 ♘b8 11 ♕e2 ♗f5 12 ♘g3 ♗g4 13 ♕e4 ♗×f3 14 ♕×f3 ♘d7 15 O-O ♘b6 16 ♗b3 a5 17 ♖ad1 a4 18 ♗c2 g6 19 ♖fe1 ♗g7 20 ♘e4 ♘c4 21 ♘c5 ♘×b2 22 ♖b1 a3 23 ♘×b7 ♕e7 24 ♖e4 f5 25 ♖b4 e4 26 ♕e3 ♕e5 27 ♘c5 ♕×c3 28 ♕×c3 ♗×c3 29 ♖b3 ♗d2 30 ♘e6 ♖fc8 31 g4 ♘c4 32 ♗d1 f4 33 ♔g2 f3+ 34 ♔g3 ♗a5 35 ♖b7 ♗c3 36 ♔f4 ♘d6 37 ♖7b3 ♗d2+ 38 ♔g3 ♔f7 39 h4 c6 40 ♖b6 ♗a5 41 ♘g5+ ♔e7 42 ♖×c6 ♖×c6 43 d×c6 h6 44 ♘h3 ♗c3 45 ♔h2 ♗e5+ 46 ♔g1 ♖a5 47 g5 h×g5 48 ♘×g5 ♖c5 49 ♗a4 ♖c4 50 ♗b3 ♖×c6 51 ♖e1 ♔f6 52 ♘h7+ ♔f5 53 ♘g5 ♖c5 54 ♗e6+ ♔f6 55 ♗b3 ♖b5 56 ♖b1 ♔e7 57 ♗c2 ♖c5 58 ♗×e4 ♘×e4 59 ♘×e4 ♖c2 60 ♖e1 ♔f8 61 ♘g5 ♗g3 62 ♖f1 ♗×h4 63 ♘×f3 ♗f6 64 ♖d1 ♖×a2 65 ♖d6 ♔g7 66 ♖a6 ♗c3 67 ♔g2 ♗b4 68 ♘e5 ♖c2 69 ♖×g6+ ♔h7 70 ♖a6 a2 71 ♘g4 ♔g7 72 ♘e3 ♖b2 73 ♘c4 ♖c2 74 ♘e3 ♖b2 75 ♘c4 ♖e2 76 ♘e3 ♗c3 77 ♔f3 ♖×e3+ 0.5-0.5

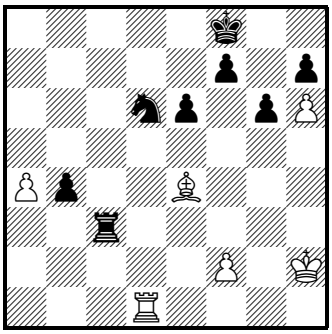
Final position: Insufficient material → 0.5-0.5



Round 10: DEEP SJENG - LOOP AMSTERDAM, *Semi-Slav* (D47): *Meran System*

1 d4 d5 2 c4 c6 3 ♘c3 ♘f6 4 e3 e6 5 ♘f3 ♘bd7 6 ♗d3 d×c4 7 ♗×c4 b5 8 ♗d3 ♗b7 9 e4 b4 10 ♘a4 c5 11 e5 ♘d5 12 O-O ♖c8 13 ♘×c5 ♘×c5 14 d×c5 ♗×c5 15 ♗d2 ♘e7 16 ♗b5+ ♗c6 17 ♗a6 ♖c7 18 ♕e2 O-O 19 ♖ad1 ♕a8 20 ♖c1 ♘g6 21 ♗e3 ♗b6 22 ♘e1 ♖d8 23 ♗d3 ♕b8 24 b3 a5 25 ♖c2 ♗×e3 26 ♕×e3 ♘e7 27 ♕c5 ♖dd7 28 ♕e3 ♘d5 29 ♕e4 g6 30 ♕f3 ♗b7 31 ♕g3 ♖×c2 32 ♘×c2 ♘c3 33 ♕e3 ♕d8 34 ♗c4 ♖d2 35 ♘e1 ♖×a2 36 ♘f3 ♗×f3 37 ♕×f3 ♕c7 38 ♕d3 ♕×e5 39 h4 ♘e2+ 40 ♔h1 ♔g7 41 g3 ♕c5 42 ♔g2 ♘d4 43 h5 ♘f5 44 ♔g1 ♘d6 45 ♗a6 a4 46 h6+ ♔f8 47 b×a4 ♖a3 48 ♕d1 ♖×g3+ 49 ♔h2 ♖c3 50 ♗d3 ♕d4 51 ♗e4 ♕×d1 52 ♖×d1

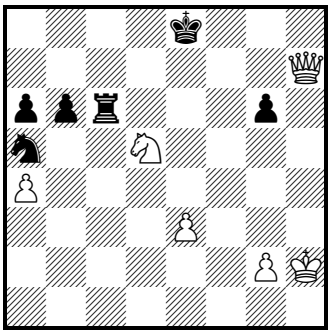
Final position: DEEP SJENG resigns → 0-1



Round 11: LOOP AMSTERDAM - DIEP

1 e4 c5 2 ♘f3 d6 3 ♗b5+ ♗d7 4 ♗×d7+ ♘×d7 5 O-O ♘gf6 6 ♘c3 g6 7 d3 ♗g7 8 h3 ♕b6 9 a4 O-O 10 ♗g5 ♕×b2 11 ♘b5 c4 12 d4 ♕b4 13 ♖e1 ♖fc8 14 ♗d2 c3 15 ♘×c3 ♕c4 16 ♖a3 e6 17 ♖b3 b6 18 ♗e3 h6 19 ♕c1 ♔h7 20 ♘d2 ♕c6 21 ♕b2 a6 22 ♕c1 d5 23 e5 ♘g8 24 ♘f3 ♘e7 25 ♕d1 ♕c7 26 ♗c1 ♕c4 27 ♖e3 ♘c6 28 h4 ♘a5 29 ♖b1 ♕c6 30 ♖d3 ♕b7 31 h5 ♖c7 32 h×g6+ f×g6 33 ♘e1 ♖c4 34 ♖h3 ♕c8 35 ♘e2 ♕e8 36 ♘f3 ♖ac8 37 ♗×h6 ♗×h6 38 ♘g5+ ♔g7 39 ♕d2 ♖×c2 40 ♘×e6+ ♔f7 41 ♕×h6 ♕×e6 42 ♘f4 ♕f5 43 ♖f3 ♘×e5 44 d×e5 ♖c1+ 45 ♖×c1 ♖×c1+ 46 ♔h2 ♕×e5 47 ♖e3 ♕×e3 48 f×e3 ♖c6 49 ♕h7+ ♔e8 50 ♘×d5

Final position: DIEP is mated in 14 moves (verified by LOOP AMSTERDAM): 50... ♔d8 51 e4 ♔c8 52 e5 ♖c5 53 ♕g8 ♔b7 54 ♕d8 ♘c4 55 e6 ♖c8 56 ♕d7+ ♔b8 57 e7 b5 58 e8Q ♖×e8 59 ♕×e8+ ♔b7 60 a×b5 a×b5 61 ♕×b5 → 1-0



Index

Summary

The thesis investigates the most important requirements, objectives, rules, and theories for the development of state-of-the-art computer-chess architectures. For this task we focus on the question how to develop new computer-chess architectures that make it possible to implement complex chess knowledge leading to a higher overall performance by a higher computing speed. Furthermore, the implemented data structures should be straightforward and compact in order to minimise unnecessary overhead.

The computer-chess architectures and the corresponding algorithms should perform on different hardware and software environments. The following problem statement guides our research.

> **Problem statement:** *How can we develop new computer-chess architectures in such a way that computer-chess engines combine the requirements on knowledge expressiveness with a maximum of efficiency?*

To answer the problem statement we formulated three research questions. They deal with (1) the development and analysis of a non-bitboard computer-chess architecture, (2) the development and analysis of a computer-chess architecture based on magic multiplication, and (3) the development and analysis of a static exchange evaluator (SEE) with $\alpha\beta$-approach. A precise formulation is given later in this summary.

Chapter 1 is a general introduction of computer-chess architectures. Our problem statement and the three research questions are formulated. Every research question is discussed and answered in an own chapter. In turn they seek to answer the problem statement.

Chapter 2 describes the development of a non-bitboard computer-chess architecture which is carried out on an R&D basis of the computer-chess engines Loop Leiden 2006 and Loop Express. One of the objectives of the new computer-chess architecture is a strong and homogeneous data structure that can also be used in the environment of a multi-core computer-chess engine. Therefore, we imposed three criteria on the used data structures, which are (1) competitiveness in speed, (2) simplicity, and (3) ease of implementation. This challenge has led us to the first research question.

> **Research question 1:** *To what extent can we develop non-bitboard computer-chess architectures, which are competitive in speed, simplicity, and ease of implementation?*

Although based on the experiences gained during the development of the Loop computer-chess β-engines 2005-2006, the 32-bit computer-chess architecture for Loop Leiden was written from scratch. We focus on the development of the chessboard and the management of chessboard-related information. Only if these new developments are in a harmonious interplay, a high-performance framework for the highest requirements on a computer-chess engine can be implemented.

These technologies have proven their performance within the computer-chess engine Loop Leiden at the 26^{th} Open Dutch Computer-Chess Championship, Leiden (NL) 2006. The engine was able to reach the 2^{nd} place. Furthermore, this non-bitboard computer-chess architecture has been used in two external projects, the Chess Machine Hydra and Nintendo Wii Chess, since 2006.

Chapter 3 focuses on the development of a complete computer-chess architecture based on hash functions and magic multiplications for the examination of bitboards. This has led us to the second research question.

> **Research question 2:** *To what extent is it possible to use hash functions and magic multiplications in order to examine bitboards in computer chess?*

In this chapter the basics of the magic hash approach and the magic hash functions are examined. In order to answer the second research question, an advanced version of our computer-chess architecture so far must be developed. The implementation of this computer-chess architecture is based on a perfect mapping function and on 64-bit unsigned integers (bitboards).

For the development of a well performing magic hash algorithm, only basic arithmetic and Boolean operations are used. Two main objectives of this chapter are the indication of (1) arbitrary n-bit unsigned integers, such as multiple 1-bit computer words, and (2) compound bit-patterns via hash functions. Only in this way it is possible to develop (1) a challenging bit scan and (2) the movement of sliding pieces without computing redundant rotated bitboards.

The new computer-chess architecture is implemented in the computer-chess engine Loop Amsterdam. This engine was able to reach the 3^{rd} place at the 15^{th} World Computer-Chess Championship, Amsterdam (NL) 2007. An essential reason for the success of this 64-bit computer-chess engine was the use of highly sophisticated perfect hash functions and magic multipliers for the computation of compound bit-patterns (bitboards) via perfect hashing.

Chapter 4 deals with the R&D of a straightforward and more efficient static exchange evaluator (SEE) for the computer-chess engine Loop Amsterdam.

The SEE is an important module of the computer-chess architecture for the evaluation of moves and threatened squares. When using an $\alpha\beta$-window it is

possible to implement efficient pruning conditions. The benefit of the pruning conditions is the reduction of an iterative computation. The third research question deals with the theoretical elements and the implementation of different SEE-algorithms.

> **Research question 3:** *How can we develop an $\alpha\beta$-approach in order to implement pruning conditions in the domain of static exchange evaluation?*

The development of an SEE is a challenging issue. After the introduction of the SEE algorithm, recursive and iterative implementations are examined. An $\alpha\beta$-window to control the evaluation is introduced on the basis of an iterative approach. Due to the new architecture of the algorithm, the implementation of pruning conditions is possible.

Some typical applications of the SEE in the field of a computer-chess engine are introduced. The applications range from move ordering and move selection to the control of search extensions and to the evaluation of passed-pawn structures. Due to the variety of possible applications of the SEE, it is interesting to scrutinize the complex algorithm and the extensive possibilities of an implementation within a state-of-the-art computer-chess architecture.

The last chapter of the thesis contains the research conclusions and recommendations for future research. Taking the answers to the three research questions into account, we are able to give an answer to our problem statement. All implementations were tested in the environment of the state-of-the-art computer-chess engines Loop Leiden 2006 and Loop Amsterdam 2007. So, every idea and technology is tested and evaluated in a competitive computer-chess program.

Samenvatting

Het proefschrift behandelt de belangrijkste voorwaarden, doelstellingen, regels en theoretische benadering voor de ontwikkeling van geavanceerde computerschaak-architecturen. Om dit te bereiken concentreren we ons op de vraag hoe we nieuwe computerschaak-architecturen kunnen ontwikkelen die het mogelijk maken om complexe schaakkennis te implementeren zodat betere spelprestaties worden verkregen mede als gevolg van een grotere computersnelheid. Bovendien moeten de geïmplementeerde gegevensstructuren eenvoudig en compact zijn om onnodige *overhead* te minimaliseren.

De computerschaak-architecturen en de daarbij behorende algoritmen moeten op verschillende hardware-platforms en in verschillende software-omgevingen functioneren. De onderstaande probleemstelling geeft richting aan ons onderzoek.

> **Probleemstelling:** *Hoe kunnen we nieuwe computerschaak-architecturen ontwikkelen zodat computerschaakprogramma's de voorwaarden die opgelegd zijn aan het formuleren van schaakkennis combineren met een maximale efficiëntie?*

Om deze probleemstelling te beantwoorden formuleren we drie onderzoeksvragen. Ze betreffen (1) de ontwikkeling en analyse van een niet-bitbord gebaseerde computerschaak-architectuur, (2) de ontwikkeling en analyse van een computerschaak-architectuur die is gebaseerd op magisch vermenigvuldigen, en (3) de ontwikeling en analyse van een statische afruilevaluator (*static exchange evaluator*, SEE) binnen een $\alpha\beta$-benadering. Een preciese formulering wordt verderop in deze samenvatting gegeven.

Hoofdstuk 1 is een algemene inleiding in computerschaak-architecturen. We formuleren onze probleemstelling en de drie onderzoeksvragen. Iedere onderzoeksvraag wordt behandeld en beantwoord in een apart hoofdstuk. Gezamenlijk zullen zij een antwoord geven op de probleemstelling.

Hoofdstuk 2 beschrijft de ontwikkeling van een niet-bitbord gebaseerde computerschaak-architectuur, die geïmplementeerd is in de computerschaak programma's Loop Leiden 2006 en Loop Express. Een van de doeleinden van de nieuwe computerschaak-architectuur is een krachtige en homogene gegevensstructuur die ook in een multi-core schaakprogramma gebruikt kan worden. Daarom hebben we drie eisen aan de te gebruiken gegevensstructuur opgelegd,

namelijk competitief zijn aangaande (1) snelheid, (2) eenvoud, en (3) implementatiegemak. Deze uitdaging bracht ons tot de eerste onderzoeksvraag.

> **Onderzoeksvraag 1:** *In hoeverre kunnen we niet-bitbord gebaseerde computerschaak-architecturen ontwikkelen, die competitief zijn wat betreft snelheid, eenvoud, en implementatiegemak?*

Hoewel het is gebaseerd op ervaringen met de ontwikkeling van de β-versies van Loop 2005-2006 computerschaakprogramma's is de 32-bits computerschaak-architectuur voor het Loop Leiden programma volledig opnieuw geschreven. We richten ons daarbij op de ontwikkeling van het schaakbord en het beheer van schaakbord-gerelateerde informatie. Enkel als deze nieuwe ontwikkelingen een harmonieus geheel vormen kan een raamwerk dat tot hoge spelprestaties leidt ontwikkeld worden, zodanig dat het voldoet aan de hoogste eisen van een computerschaakprogramma.

Deze technologieën hebben hun kracht bewezen in het computerschaakprogramma Loop Leiden tijdens het 26e Open Nederlands Computerschaak Kampioenschap, Leiden (NL) 2006. Het programma wist de tweede plaats te behalen. Verder is de niet-bitbord gebaseerde computerschaak-architectuur sinds 2006 gebruikt in twee externe projecten, te weten in de Schaak Machine Hydra en in Nintendo's Wii Chess.

Hoofdstuk 3 behandelt de ontwikkeling van een volledige computerschaak-architectuur die gebaseerd is op hash-functies en het magisch vermenigvuldigen van twee getallen bij het onderzoeken van bitborden. Dit heeft ons tot de tweede onderzoeksvraag gebracht.

> **Onderzoeksvraag 2:** *In hoeverre is het mogelijk om hash-functies en magisch vermenigvuldigen te gebruiken om bitborden in computerschaak te onderzoeken?*

In dit hoofdstuk onderzoeken we de grondslagen van de magische hash-benadering en van magische-hash functies. Om de tweede onderzoeksvraag te beantwoorden moet een geavanceerde computerschaak-architectuur ontwikkeld worden. De implementatie van deze computerschaak-architectuur is gebaseerd op een perfecte afbeeldingsfunctie en op 64-bits unsigned integers (bitborden).

Voor de ontwikkeling van een goed presterende magisch hash-algoritme zal alleen gebruik worden gemaakt van standaard rekenkundige en Booleaanse bewerkingen. De twee belangrijkste doelstellingen van dit hoofdstuk zijn het bepalen van (1) willekeurige n-bits unsigned integers, zoals meervoudige 1-bits computerwoorden, en (2) samengestelde bit-patronen met behulp van hash-functies. Alleen op deze manier is het mogelijk (1) om een uitdagende bit-scan te ontwikkelen, en (2) om de zetten van de lange-afstands stukken te bepalen zonder overbodige geroteerde bitborden te berekenen.

De nieuwe computerschaak-architectuur is geïmplementeerd in het computerschaak programma Loop Amsterdam. Dit programma wist de derde plaats te behalen op het 15e Wereld Computerschaak Kampioenschap, Amsterdam (NL)

2007. Een belangrijke reden voor het succes van dit 64-bits computerschaakprogramma was de berekening van samengestelde bit-patronen (bitborden) met behulp van perfecte hash-methoden.

Hoofdstuk 4 behandelt de implementatie van een eenvoudige en efficiënte statische afruilevaluator (SEE) voor het computerschaakprogramma Loop Amsterdam.

De SEE is een belangrijke module van de computerschaak-architectuur voor de evaluatie van zetten en van bedreigde velden. Bij gebruik van een $\alpha\beta$-raam is het mogelijk om efficiënte voorwaarden om te snoeien te bepalen. Het voordeel van het bepalen van deze voorwaarden tot snoeien is het reduceren van een iteratieve berekening. De derde onderzoeksvraag gaat over de theoretische bestanddelen en de implementatie van verschillende SEE-algoritmen.

> **Onderzoeksvraag 3:** *Hoe kunnen we een $\alpha\beta$-benadering ontwikkelen om voorwaarden voor snoeien op het gebied van statische afruilevaluatoren te implementeren?*

De ontwikkeling van een SEE is een uitdagende taak. Na een eerste introductie van het SEE-algoritme zullen er recursieve en iteratieve implementaties onderzocht worden. Een $\alpha\beta$-raam om de evaluatie te regelen wordt geïntroduceerd; het is gebaseerd op een iteratieve benadering. Dankzij de nieuwe architectuur van het algoritme is het mogelijk om de voorwaarden tot snoeien te implementeren.

Enkele typische toepassingen van de SEE op het gebied van een computerschaakprogramma worden besproken. De toepassingen bestrijken onderwerpen als zetordening en zettenselectie, het regelen van zetextensies, en de evaluatiestructuren voor een doorgebroken pion.

Door de gevarieerde toepassingen van de SEE is het interessant in detail te onderzoeken hoe het complexe algoritme precies werkt en wat de uitgebreide mogelijkheden van de implementatie binnen het domein van geavanceerde computerschaak-architecturen zijn.

Het laatste hoofdstuk van het proefschrift bevat de onderzoeksconclusies en de aanbevelingen voor toekomstig onderzoek. Op basis van de antwoorden op de drie onderzoeksvragen zijn we in staat een antwoord op de probleemstelling te geven. Alle implementaties zijn getest binnen de geavanceerde computerschaakprogramma's Loop Leiden 2006 en Loop Amsterdam 2007. Kortom, ieder idee en iedere technologie is getest en geëvalueerd in een competitief computerschaakprogramma.

Curriculum Vitae

Fritz Reul was born in Hanau, Germany, on September 30, 1977. From 1988 to 1997 he attended the Grimmelshausen Gymnasium in Gelnhausen, Germany. After the Abitur in 1997 he started studying Sports Science at the University of Frankfurt, Germany. In 1999 he started studying mathematics and applied computer sciences at the University of Applied Sciences Friedberg, Germany. There he deepened his knowledge about numerical algorithms and artificial intelligence. He improved his programming skills and focussed on the theory, the algorithms, and the implementation of two-player zero-sum perfect-information games. In 2004 he completed his diploma project about computer chess at the University of Applied Sciences in Friedberg. Immediately afterwards he continued in 2004 with the R&D of the computer-chess engine Loop.

In 2005 he started his Ph.D. project "New Architectures in Computer Chess" at the Maastricht University supervised by Jaap van den Herik and Jos Uiterwijk. In the first year of his R&D he concentrated on the redesign of a computer-chess architecture for 32-bit computer environments. In 2006 he developed the dual-core 32-bit computer-chess architecture Loop Leiden. In 2007 he developed the quad-core 64-bit computer-chess architecture Loop Amsterdam. In 2007 he licensed the C/C++ source codes of Loop for single-core 32-bit computer environments to Nintendo, which was implemented successfully in the commercial product Nintendo Wii Chess.

At the same time he worked as a lecturer of mathematics and computer sciences at the University of Applied Sciences in Friedberg. Furthermore, he participated with his computer-chess engines at the 26^{th} Open Dutch Computer-Chess Championship 2006, Leiden (NL) and the 15^{th} World Computer-Chess Championship 2007, Amsterdam (NL) scoring second and third ranks.

SIKS Dissertation Series

1998[1]

1 Johan van den Akker (CWI) *DEGAS - An Active, Temporal Database of Autonomous Objects*

2 Floris Wiesman (UM) *Information Retrieval by Graphically Browsing Meta-Information*

3 Ans Steuten (TUD) *A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective*

4 Dennis Breuker (UM) *Memory versus Search in Games*

5 Eduard Oskamp (RUL) *Computerondersteuning bij Straftoemeting*

1999

1 Mark Sloof (VU) *Physiology of Quality Change Modelling; Automated Modelling of Quality Change of Agricultural Products*

2 Rob Potharst (EUR) *Classification using Decision Trees and Neural Nets*

3 Don Beal (UM) *The Nature of Minimax Search*

4 Jacques Penders (UM) *The Practical Art of Moving Physical Objects*

5 Aldo de Moor (KUB) *Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems*

6 Niek Wijngaards (VU) *Re-Design of Compositional Systems*

7 David Spelt (UT) *Verification Support for Object Database Design*

8 Jacques Lenting (UM) *Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation*

2000

1 Frank Niessink (VU) *Perspectives on Improving Software Maintenance*

2 Koen Holtman (TU/e) *Prototyping of CMS Storage Management*

3 Carolien Metselaar (UvA) *Sociaal-organisatorische Gevolgen van Kennistechnologie; een Procesbenadering en Actorperspectief*

4 Geert de Haan (VU) *ETAG, A Formal Model of Competence Knowledge for User Interface Design*

[1]Abbreviations: SIKS – Dutch Research School for Information and Knowledge Systems; CWI – Centrum voor Wiskunde en Informatica, Amsterdam; EUR – Erasmus Universiteit, Rotterdam; KUB – Katholieke Universiteit Brabant, Tilburg; KUN – Katholieke Universiteit Nijmegen; RUG – Rijksuniversiteit Groningen; RUL – Rijksuniversiteit Leiden; FONS – Ferrologisch Onderzoeksinstituut Nederland/Sweden; RUN – Radboud Universiteit Nijmegen; TUD – Technische Universiteit Delft; TU/e – Technische Universiteit Eindhoven; UL – Universiteit Leiden; UM – Universiteit Maastricht; UT – Universiteit Twente, Enschede; UU – Universiteit Utrecht; UvA – Universiteit van Amsterdam; UvT – Universiteit van Tilburg; VU – Vrije Universiteit, Amsterdam.

5 Ruud van der Pol (UM) *Knowledge-Based Query Formulation in Information Retrieval*
6 Rogier van Eijk (UU) *Programming Languages for Agent Communication*
7 Niels Peek (UU) *Decision-Theoretic Planning of Clinical Patient Management*
8 Veerle Coupé (EUR) *Sensitivity Analyis of Decision-Theoretic Networks*
9 Florian Waas (CWI) *Principles of Probabilistic Query Optimization*
10 Niels Nes (CWI) *Image Database Management System Design Considerations, Algorithms and Architecture*
11 Jonas Karlsson (CWI) *Scalable Distributed Data Structures for Database Management*

2001

1 Silja Renooij (UU) *Qualitative Approaches to Quantifying Probabilistic Networks*
2 Koen Hindriks (UU) *Agent Programming Languages: Programming with Mental Models*
3 Maarten van Someren (UvA) *Learning as Problem Solving*
4 Evgueni Smirnov (UM) *Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets*
5 Jacco van Ossenbruggen (VU) *Processing Structured Hypermedia: A Matter of Style*
6 Martijn van Welie (VU) *Task-Based User Interface Design*
7 Bastiaan Schonhage (VU) *Diva: Architectural Perspectives on Information Visualization*
8 Pascal van Eck (VU) *A Compositional Semantic Structure for Multi-Agent Systems Dynamics*
9 Pieter Jan 't Hoen (RUL) *Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes*
10 Maarten Sierhuis (UvA) *Modeling and Simulating Work Practice BRAHMS: a Multiagent Modeling and Simulation Language for Work Practice Analysis and Design*
11 Tom van Engers (VU) *Knowledge Management: The Role of Mental Models in Business Systems Design*

2002

1 Nico Lassing (VU) *Architecture-Level Modifiability Analysis*
2 Roelof van Zwol (UT) *Modelling and Searching Web-based Document Collections*
3 Henk Ernst Blok (UT) *Database Optimization Aspects for Information Retrieval*
4 Juan Roberto Castelo Valdueza (UU) *The Discrete Acyclic Digraph Markov Model in Data Mining*
5 Radu Serban (VU) *The Private Cyberspace Modeling Electronic Environments Inhabited by Privacy-Concerned Agents*
6 Laurens Mommers (UL) *Applied Legal Epistemology; Building a Knowledge-based Ontology of the Legal Domain*
7 Peter Boncz (CWI) *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*
8 Jaap Gordijn (VU) *Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas*
9 Willem-Jan van den Heuvel (KUB) *Integrating Modern Business Applications with Objectified Legacy Systems*
10 Brian Sheppard (UM) *Towards Perfect Play of Scrabble*

11 Wouter Wijngaards (VU) *Agent Based Modelling of Dynamics: Biological and Organisational Applications*

12 Albrecht Schmidt (UvA) *Processing XML in Database Systems*

13 Hongjing Wu (TU/e) *A Reference Architecture for Adaptive Hypermedia Applications*

14 Wieke de Vries (UU) *Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems*

15 Rik Eshuis (UT) *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*

16 Pieter van Langen (VU) *The Anatomy of Design: Foundations, Models and Applications*

17 Stefan Manegold (UvA) *Understanding, Modeling, and Improving Main-Memory Database Performance*

2003

1 Heiner Stuckenschmidt (VU) *Ontology-Based Information Sharing in Weakly Structured Environments*

2 Jan Broersen (VU) *Modal Action Logics for Reasoning About Reactive Systems*

3 Martijn Schuemie (TUD) *Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy*

4 Milan Petkovič (UT) *Content-Based Video Retrieval Supported by Database Technology*

5 Jos Lehmann (UvA) *Causation in Artificial Intelligence and Law – A Modelling Approach*

6 Boris van Schooten (UT) *Development and Specification of Virtual Environments*

7 Machiel Jansen (UvA) *Formal Explorations of Knowledge Intensive Tasks*

8 Yong-Ping Ran (UM) *Repair-Based Scheduling*

9 Rens Kortmann (UM) *The Resolution of Visually Guided Behaviour*

10 Andreas Lincke (UT) *Electronic Business Negotiation: Some Experimental Studies on the Interaction between Medium, Innovation Context and Cult*

11 Simon Keizer (UT) *Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks*

12 Roeland Ordelman (UT) *Dutch Speech Recognition in Multimedia Information Retrieval*

13 Jeroen Donkers (UM) *Nosce Hostem – Searching with Opponent Models*

14 Stijn Hoppenbrouwers (KUN) *Freezing Language: Conceptualisation Processes across ICT-Supported Organisations*

15 Mathijs de Weerdt (TUD) *Plan Merging in Multi-Agent Systems*

16 Menzo Windhouwer (CWI) *Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouse*

17 David Jansen (UT) *Extensions of Statecharts with Probability, Time, and Stochastic Timing*

18 Levente Kocsis (UM) *Learning Search Decisions*

2004

1 Virginia Dignum (UU) *A Model for Organizational Interaction: Based on Agents, Founded in Logic*

2 Lai Xu (UvT) *Monitoring Multi-party Contracts for E-business*

3 Perry Groot (VU) *A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving*
4 Chris van Aart (UvA) *Organizational Principles for Multi-Agent Architectures*
5 Viara Popova (EUR) *Knowledge Discovery and Monotonicity*
6 Bart-Jan Hommes (TUD) *The Evaluation of Business Process Modeling Techniques*
7 Elise Boltjes (UM) *Voorbeeld$_{IG}$ Onderwijs; Voorbeeldgestuurd Onderwijs, een Opstap naar Abstract Denken, vooral voor Meisjes*
8 Joop Verbeek (UM) *Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale Politiële Gegevensuitwisseling en Digitale Expertise*
9 Martin Caminada (VU) *For the Sake of the Argument; Explorations into Argument-based Reasoning*
10 Suzanne Kabel (UvA) *Knowledge-rich Indexing of Learning-objects*
11 Michel Klein (VU) *Change Management for Distributed Ontologies*
12 The Duy Bui (UT) *Creating Emotions and Facial Expressions for Embodied Agents*
13 Wojciech Jamroga (UT) *Using Multiple Models of Reality: On Agents who Know how to Play*
14 Paul Harrenstein (UU) *Logic in Conflict. Logical Explorations in Strategic Equilibrium*
15 Arno Knobbe (UU) *Multi-Relational Data Mining*
16 Federico Divina (VU) *Hybrid Genetic Relational Search for Inductive Learning*
17 Mark Winands (UM) *Informed Search in Complex Games*
18 Vania Bessa Machado (UvA) *Supporting the Construction of Qualitative Knowledge Models*
19 Thijs Westerveld (UT) *Using generative probabilistic models for multimedia retrieval*
20 Madelon Evers (Nyenrode) *Learning from Design: facilitating multidisciplinary design teams*

2005

1 Floor Verdenius (UvA) *Methodological Aspects of Designing Induction-Based Applications*
2 Erik van der Werf (UM) *AI techniques for the game of Go*
3 Franc Grootjen (RUN) *A Pragmatic Approach to the Conceptualisation of Language*
4 Nirvana Meratnia (UT) *Towards Database Support for Moving Object data*
5 Gabriel Infante-Lopez (UvA) *Two-Level Probabilistic Grammars for Natural Language Parsing*
6 Pieter Spronck (UM) *Adaptive Game AI*
7 Flavius Frasincar (TU/e) *Hypermedia Presentation Generation for Semantic Web Information Systems*
8 Richard Vdovjak (TU/e) *A Model-driven Approach for Building Distributed Ontology-based Web Applications*
9 Jeen Broekstra (VU) *Storage, Querying and Inferencing for Semantic Web Languages*
10 Anders Bouwer (UvA) *Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments*
11 Elth Ogston (VU) *Agent Based Matchmaking and Clustering - A Decentralized Approach to Search*

12 Csaba Boer (EUR) *Distributed Simulation in Industry*
13 Fred Hamburg (UL) *Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen*
14 Borys Omelayenko (VU) *Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics*
15 Tibor Bosse (VU) *Analysis of the Dynamics of Cognitive Processes*
16 Joris Graaumans (UU) *Usability of XML Query Languages*
17 Boris Shishkov (TUD) *Software Specification Based on Re-usable Business Components*
18 Danielle Sent (UU) *Test-selection strategies for probabilistic networks*
19 Michel van Dartel (UM) *Situated Representation*
20 Cristina Coteanu (UL) *Cyber Consumer Law, State of the Art and Perspectives*
21 Wijnand Derks (UT) *Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics*

2006

1 Samuil Angelov (TU/e) *Foundations of B2B Electronic Contracting*
2 Cristina Chisalita (VU) *Contextual issues in the design and use of information technology in organizations*
3 Noor Christoph (UvA) *The role of metacognitive skills in learning to solve problems*
4 Marta Sabou (VU) *Building Web Service Ontologies*
5 Cees Pierik (UU) *Validation Techniques for Object-Oriented Proof Outlines*
6 Ziv Baida (VU) *Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling*
7 Marko Smiljanic (UT) *XML schema matching – balancing efficiency and effectiveness by means of clustering*
8 Eelco Herder (UT) *Forward, Back and Home Again - Analyzing User Behavior on the Web*
9 Mohamed Wahdan (UM) *Automatic Formulation of the Auditor's Opinion*
10 Ronny Siebes (VU) *Semantic Routing in Peer-to-Peer Systems*
11 Joeri van Ruth (UT) *Flattening Queries over Nested Data Types*
12 Bert Bongers (VU) *Interactivation - Towards an e-cology of people, our technological environment, and the arts*
13 Henk-Jan Lebbink (UU) *Dialogue and Decision Games for Information Exchanging Agents*
14 Johan Hoorn (VU) *Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change*
15 Rainer Malik (UU) *CONAN: Text Mining in the Biomedical Domain*
16 Carsten Riggelsen (UU) *Approximation Methods for Efficient Learning of Bayesian Networks*
17 Stacey Nagata (UU) *User Assistance for Multitasking with Interruptions on a Mobile Device*
18 Valentin Zhizhkun (UvA) *Graph transformation for Natural Language Processing*
19 Birna van Riemsdijk (UU) *Cognitive Agent Programming: A Semantic Approach*
20 Marina Velikova (UvT) *Monotone models for prediction in data mining*
21 Bas van Gils (RUN) *Aptness on the Web*
22 Paul de Vrieze (RUN) *Fundaments of Adaptive Personalisation*
23 Ion Juvina (UU) *Development of a Cognitive Model for Navigating on the Web*
24 Laura Hollink (VU) *Semantic Annotation for Retrieval of Visual Resources*

25 Madalina Drugan (UU) *Conditional log-likelihood MDL and Evolutionary MCMC*
26 Vojkan Mihajlovic (UT) *Score Region Algebra: A Flexible Framework for Structured Information Retrieval*
27 Stefano Bocconi (CWI) *Vox Populi: generating video documentaries from semantically annotated media repositories*
28 Borkur Sigurbjornsson (UvA) *Focused Information Access using XML Element Retrieval*

2007

1 Kees Leune (UvT) *Access Control and Service-Oriented Architectures*
2 Wouter Teepe (RUG) *Reconciling Information Exchange and Confidentiality: A Formal Approach*
3 Peter Mika (VU) *Social Networks and the Semantic Web*
4 Jurriaan van Diggelen (UU) *Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach*
5 Bart Schermer (UL) *Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance*
6 Gilad Mishne (UvA) *Applied Text Analytics for Blogs*
7 Natasa Jovanovic' (UT) *To Whom It May Concern - Addressee Identification in Face-to-Face Meetings*
8 Mark Hoogendoorn (VU) *Modeling of Change in Multi-Agent Organizations*
9 David Mobach (VU) *Agent-Based Mediated Service Negotiation*
10 Huib Aldewereld (UU) *Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols*
11 Natalia Stash (TU/e) *Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System*
12 Marcel van Gerven (RUN) *Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty*
13 Rutger Rienks (UT) *Meetings in Smart Environments; Implications of Progressing Technology*
14 Niek Bergboer (UM) *Context-Based Image Analysis*
15 Joyca Lacroix (UM) *NIM: a Situated Computational Memory Model*
16 Davide Grossi (UU) *Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems*
17 Theodore Charitos (UU) *Reasoning with Dynamic Networks in Practice*
18 Bart Orriens (UvT) *On the development and management of adaptive business collaborations*
19 David Levy (UM) *Intimate relationships with artificial partners*
20 Slinger Jansen (UU) *Customer Configuration Updating in a Software Supply Network*
21 Karianne Vermaas (UU) *Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005*
22 Zlatko Zlatev (UT) *Goal-oriented design of value and process models from patterns*
23 Peter Barna (TU/e) *Specification of Application Logic in Web Information Systems*
24 Georgina Ramírez Camps (CWI) *Structural Features in XML Retrieval*
25 Joost Schalken (VU) *Empirical Investigations in Software Process Improvement*

2008

1 Katalin Boer-Sorbán (EUR) *Agent-Based Simulation of Financial Markets: A modular, continuous-time approach*
2 Alexei Sharpanskykh (VU) *On Computer-Aided Methods for Modeling and Analysis of Organizations*
3 Vera Hollink (UvA) *Optimizing hierarchical menus: a usage-based approach*
4 Ander de Keijzer (UT) *Management of Uncertain Data - towards unattended integration*
5 Bela Mutschler (UT) *Modeling and simulating causal dependencies on process-aware information systems from a cost perspective*
6 Arjen Hommersom (RUN) *On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective*
7 Peter van Rosmalen (OU) *Supporting the tutor in the design and support of adaptive e-learning*
8 Janneke Bolt (UU) *Bayesian Networks: Aspects of Approximate Inference*
9 Christof van Nimwegen (UU) *The paradox of the guided user: assistance can be counter-effective*
10 Wauter Bosma (UT) *Discourse oriented Summarization*
11 Vera Kartseva (VU) *Designing Controls for Network Organizations: a Value-Based Approach*
12 Jozsef Farkas (RUN) *A Semiotically oriented Cognitive Model of Knowledge Representation*
13 Caterina Carraciolo (UvA) *Topic Driven Access to Scientific Handbooks*
14 Arthur van Bunningen (UT) *Context-Aware Querying; Better Answers with Less Effort*
15 Martijn van Otterlo (UT) *The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains*
16 Henriette van Vugt (VU) *Embodied Agents from a User's Perspective*
17 Martin Op't Land (TUD) *Applying Architecture and Ontology to the Splitting and Allying of Enterprises*
18 Guido de Croon (UM) *Adaptive Active Vision*
19 Henning Rode (UT) *From document to entity retrieval: improving precision and performance of focused text search*
20 Rex Arendsen (UvA) *Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met een overheid op de administratieve lasten van bedrijven*
21 Krisztian Balog (UvA) *People search in the enterprise*
22 Henk Koning (UU) *Communication of IT-architecture*
23 Stefan Visscher (UU) *Bayesian network models for the management of ventilator-associated pneumonia*
24 Zharko Aleksovski (VU) *Using background knowledge in ontology matching*
25 Geert Jonker (UU) *Efficient and Equitable exchange in air traffic management plan repair using spender-signed currency*
26 Marijn Huijbregts (UT) *Segmentation, diarization and speech transcription: surprise data unraveled*
27 Hubert Vogten (OU) *Design and implementation strategies for IMS learning design*
28 Ildikó Flesh (RUN) *On the use of independence relations in Bayesian networks*
29 Dennis Reidsma (UT) *Annotations and subjective machines - Of annotators, embodied agents, users, and other humans*

30 Wouter van Atteveldt (VU) *Semantic network analysis: techniques for extracting, representing and querying media content*

31 Loes Braun (UM) *Pro-active medical information retrieval*

32 Trung Hui (UT) *Toward affective dialogue management using partially observable Markov decision processes*

33 Frank Terpstra (UvA) *Scientific workflow design; theoretical and practical issues*

34 Jeroen De Knijf (UU) *Studies in Frequent Tree Mining*

35 Benjamin Torben-Nielsen (UvT) *Dendritic morphology: function shapes structure*

2009

1 Rasa Jurgenelaite (RUN) *Symmetric Causal Independence Models*

2 Willem Robert van Hage (VU) *Evaluating Ontology-Alignment Techniques*

3 Hans Stol (UvT) *A Framework for Evidence-based Policy Making Using IT*

4 Josephine Nabukenya (RUN) *Improving the Quality of Organisational Policy Making using Collaboration Engineering*

5 Sietse Overbeek (RUN) *Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality*

6 Muhammad Subianto (UU) *Understanding Classification*

7 Ronald Poppe (UT) *Discriminative Vision-Based Recovery and Recognition of Human Motion*

8 Volker Nannen (VU) *Evolutionary Agent-Based Policy Analysis in Dynamic Environments*

9 Benjamin Kanagwa (RUN) *Design, Discovery and Construction of Service-oriented Systems*

10 Jan Wielemaker (UVA) *Logic programming for knowledge-intensive interactive applications*

11 Alexander Boer (UVA) *Legal Theory, Sources of Law & the Semantic Web*

12 Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin) *Operating Guidelines for Services*

13 Steven de Jong (UM) *Fairness in Multi-Agent Systems*

14 Maksym Korotkiy (VU) *From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)*

15 Rinke Hoekstra (UVA) *Ontology Representation - Design Patterns and Ontologies that Make Sense*

16 Fritz Reul (UvT) *New Architectures in Computer Chess*

TiCC Ph.D. Series

1. Pashiera Barkhuysen
Audiovisual prosody in interaction
Promotor: M.G.J. Swerts, E.J. Krahmer
Tilburg, 3 October 2008

2. Ben Torben-Nielsen
Dendritic morphology: function shapes structure
Promotores: H.J. van den Herik, E.O. Postma
Copromotor: K.P. Tuyls
Tilburg, 3 December 2008

3. Hans Stol
A framework for evidence-based policy making using IT
Promotor: H.J. van den Herik
Tilburg, 21 January 2009

4. Jeroen Geertzen
Act recognition and prediction. Explorations in computational dialogue modelling
Promotor: H.C. Bunt
Copromotor: J.M.B. Terken
Tilburg, 11 February 2009

5. Sander Canisius
Structural prediction for natural language processing: a constraint satisfaction approach
Promotores: A.P.J. van den Bosch, W.M.P. Daelemans
Tilburg, 13 February 2009

6. Fritz Reul
New Architectures in Computer Chess
Promotor: H.J. van den Herik
Copromotor: J.W.H.M. Uiterwijk
Tilburg, 17 June 2009