Theory and practical strategies for efficient alpha-beta-searches in computer chess

Johannes Buchner

17th August 2005

Bachelorarbeit

Betreuer: Prof. Dr. Raúl Rojas

Fachbereich Mathematik/Informatik

Freie Universität Berlin

Abstract

The aim of this work is to give an overview of the theory of efficient alpha-beta-searches in computer chess and to explain the practical strategies implemented in the FUSc# chess program ([6]). The main focus is put on two topics: the first one is efficient move generation with a technique called "rotated bitboards" ([10]), and the second one is how to carry out efficient alpha-beta-searches by optimizing the move-ordering with static and dynamic heuristics. Practical experiments with FUSc# are used to verify the theoretical results in both questions.

In the first part, a short introduction to the basics of computer chess will help the reader to understand the structure of the FUSc# chess program. This includes a part explaining the "rotated bitboards" that are used for board-representation in Fusc#, as they are crucial for efficient move generation. In the second part, an overview of the theory of search algorithms used in computer chess is given. Many of the ideas still used in current chess programs date back to the very beginnings of computer science. This is shown by making references to the famous paper "Programming a Computer for Playing Chess" ([19]) by Claude Shannon, which was written as early as in 1950. However, there were also important improvements of the theoretical foundations of computer chess since then, the alpha-beta-algorithm beeing one of them. Although it has been subject to intensive studies since its discovery in the late 1950s, there has still been some relevant progress in the mid-90s, especially in understanding the relationship of alpha-beta (which is "depth-first") and "best-first"-algorithms like SSS* ([16]). An overview of these astonishing results is given, as they proove the traditional views on the subject to be quite wrong: It can be shown that SSS* is actually a special case of alpha-beta! After that follows a section on move ordering. One of the primary aims is to understand the influence of the move-ordering on the efficiency of alpha-beta searches, and additionally the idea behind some heuristics (static/dynamic) used for achieving a good move-ordering are explained.

The third part deals with the concrete implementation of some of the parts of our chess-program FUSc#. This includes a part on efficient move-generation using bitboards, and one which endables the reader to understand how the different heuristics for move-ordering (static/dynamic) are implemented in FUSc# as strategies in order to achieve efficient alpha-beta-searches. The forth part covers the practical experiments carried out with FUSc# in order to verify the theoretical results of part 2 and 3. A technique for verifying the correctness of move-generators in chess programm is presented and it is shown that the move-generator of FUSc# (that is based on "rotated-bitboards") works 100% correct. After that, the results of our experiments evaluating the different heuristics for achieving a good move-ordering are presented, and it is shown that those heuristics greatly improve the efficiency of the alpha-beta-searches carried out by FUSc#. In an outlook the current state-of-the-art of the FUSc#-chess program is described, and some ideas for further reseach projects are developed.

Contents

1 Basics of Computer Chess - Understanding the Structure of the FUSc#-Chess-Program

1.1 Introduction: Computer Chess as the Drosophilia for AI

Computer chess was the "drosophilia for AI" (e.g. according to [17]) till the mid-90s, after the defeat of world champion Kasparov in 1996 the interest has declined. Nevertheless, there is still a great interest in computer chess among the AI community as well as chess players in general. From a scientific point of view, the interest has shifted from the basic search algorithms to more advanced topics like machine learning, integration of perfect knowledge etc. ([9]). However, there were also some important improvements of the theoretical foundations of computer chess, concerning the alpha-beta-algorithm that forms the basis for nearly if not all successful chess programms, and especally in understanding the relationship between "best-first" algorithms like SSS* and alpha-beta ([16]).

In the first part of this work, a short introduction to the basics of computer chess will help the reader to understand the structure of the FUSc# chess program. This includes a part explaining the "rotated bitboards" that are used for board-representation in Fusc#, as they are crucial for efficient move generation, which is a necessary condition for efficient searches.

1.2 The Internals of the Fusc# chess program

1.2.1 Project History

FUSc# is the chess program developed by the "AG Schachprogrammierung" at the Free University in Berlin ([6]). It is written in C# and runs on the Microsoft .NET Framework ([12]). Here is an overview of the project history:

Date	Project Milestones
14th of october 2002	Foundation of the AG: decision for C#, .NET and OpenSource
1st of march 2003	first version (V 1.03): quiescent search, hashtables, heuristics, iterative search
1st of june 2003	first version playing on the internet (V 1.06) better evaluation
11th of june 2003	first official online-tournament: first victory!
14th of june 2003	Lange Nacht der Wissenschaften (V 1.07): documentation
January 2004	DarkFUSc#, version 0.1, rotated bitboards
July 2004	DarkFUSc#: new evaluation (incl. automatic classification of different types of chess positions)
July 2005	Lange Nacht der Wissenschaften (DarkFusc# 0.9): e.g. pondering
August 2005	DarkFusc# 1.0: better move ordering, more efficient search algorithm

The lack of performance of the old (pre-2004) FUSc# versions let us take the decision to do a complete rewrite of the FUSc#-Code in January 2004. The board-representation was changed to "rotated bitboards", which lead to a considerable performance boost (factor 2-3 according to some tests we did at that time). How this technique works is explained in detail in section 1.2.3.2.

However, still FUSc# is not as strong as other chess engines available. FUSc# is written in C# and runs on Microsoft Framework .NET, which means that the source-code of FUSc# is not compiled into machine language directly, but into the "Microsoft Intermediate Language" (MSIL), which is translated into machine language by the JIT-compiler ("Just in Time") of the .NET-Framework **at execution time**. Most other engines, on the contrary, are written in C/C++ or other langugages that are directly compiled into machine language. The compilers make intensive use of optimizations **at compiling time** (like gcc, see [13]), which offers much better possibilities for implementing more complex optimization. That's why a big part of the difference in performance can be explained by the use of different programming frameworks - .NET was not made for low-level high-performance applications in the first place, but for distributed computing, web services etc., and the .NET-Framework is also quite new on the market (version 1.1 is the most

recent stable one, with 2.0 beeing in beta planned to be released in autumn 2005), which let us hope that there will be better performance in the future.

Additionally, FUSc# was in the past mainly a research project to experiment with new ideas in chess programming like machine learning, neuronal networks (see [1], for example). That's why it is understandable that it can not compete with professional programs that were developed and tuned for many years by professional programmers and/or chess players, as our aim was not the fine-tuning of the move-generator or the search function in the first place. Nevertheless, the performance and chess skill of FUSc# have improved steadily over the last three years although the development of FUSc# was done by students in their free time, the team was changing often etc. It is has successfully playing at the FUSc#-servers for move than one year now, and has all features of modern uci[1]-engines as well as some interesting additions like a self-learning opening book. In the summer term 2005, the FUSc#-Team also organized a seminar on chess programming (lead by Marco Block and Prof. Raúl Rojas), where some of the techniques used in Fusc# as well as general chess programming topics were covered.

1.2.2 The Structure of a Computer Chess Program

As early as 1950, Claude Shannon published a much referenced paper on algorithms for playing chess ([19]). In this paper, many of the concepts that still form the basis of todays chess programs are developed. Shannon descibes the basic components of a chess program as follows (quote from [19], chapter 5):

```
The complete program [...]  consists on nine subprograms which we designate T_0, T_1,
..., T_8 and a master program T_9.  The basic functions of these programs are as follows:
```

- `T_0 - Makes move (a, b, c) in position P to obtain the resulting position.`
- `T_1 - Makes a list of the possible moves of a pawn at square (x, y) in position P.`
- `T_2, ..., T_6 - Similarly for other types of pieces:  knight, bishop, rook, queen and king.`
- `T_7 - Makes list of all possible moves in a given position.`
- `T_8 - Calculates the evaluating function f(P) for a given position P.`
- `T_9 - Master program; performs maximizing and minimizing calculation to determine proper move.`

Thus, a computer chess program consists of the following 3 major parts:

- a move generator (T_7, including T_1 - T_6)
- a search function (T_9, in the process of "minimaxing", there will be calls to T_0)
- an evaluation function (T_8)

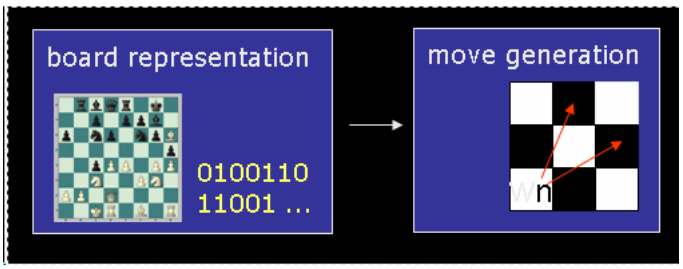
As chess programming got much attention in the history of artificial intelligence research, a wide variety of literature is available on the subject, both introductionary as well as covering more advanced topics. There are many good texts available descibing the detailed architecture of different chess programms (see e.g.[17]), and the aim of the following section is not to repeat the well-known facts on how chess programs work in general. Instead, we focus on the points that are important to understand the parts of the Fusc# chess program that will be used in our practical experiments in part 4:

[1] "unversal chess interface", the standard protocol for communicating with a chess engine and the successor of the older "winboard"-protocol

- the board representation (based on "rotated bitboards", which forms the basis for move-generation)
- the search algorithm (which will be tuned later on using different move-ordering heuristics, see 4.3)

1.2.3 Board Representation

All of the parts of a chess program descibed above are closely dependent on the way the chessboard is represented in the computer:



There exist several techniques for representing the chess board inside the computer in chess programms. The straight-forward-approach of just maintaining an array representing the 64 squares on a chessboard works fine, but has several drawbacks in move-generation as well as evaluation, which are frequently used in modern chess programs. Thus, the chess programming community has developed more advanced board representations, one of them beeing the bitboard representation.

1.2.3.1 The basic idea of bitboard representations The idea for the bitboard representation of the chessboard is based on the observation that modern CPUs are 64bit-processors, i.e. the length of a word in machine language is nowadays often 64bit. Those 64bit-words will correspond to the 64 squares on the chess board, and those "bitboards" (the name that is used for an unsigned int64) are used to represent various information about the position on the chessboard. The advantage of this representation lies in the availibility of very fast bit-manipulating operations on modern CPUs: On 64bit-machines, operations like AND, OR, NOT etc. can be executed on a 64bit "bitboard" in only one cycle. It is therefor to construct very efficient chess programms on the basis of the bitboard-approach, because, roughly speaking, the CPU operates on all 64bit "in parallel". Details how this works exactly will be given in sections 1.2.3.2 and 1.2.3.3.

In the history of computer chess, there were several authors who used variants of the bitboard-representation in their chess engines. As early as in the seventies, Slate and Atkin described the idea of using bitboards in their program "CHESS 4.5" (see [8], chapter 4). Another prominent programms that used this technique successfully is the former computer chess champion "Cray Blitz", written by Robert Hyatt, who continues to develop the program as an open-source project called "Crafty" ([3]). A third world-class chess engine using bitboards is DarkThought, developed at the university of Karlsruhe in the late 90s. Crafty and DarkThought were also the first programs that used an important refinement of the bitboard-representation called "rotated bitboards" (see section 1.2.3.3.4) . The author of DarkThought, Ernst A. Heinz, gives an overview of rotated bitboards as used in DarkThought (see [10]), which inspired much of our own developments.

1.2.3.2 Bitboards to represent a chess positions In each bitboard, a special information/property of the position can be encoded, where a "1" in the bitboard means the property is true for the given square, while a "0" means the property is not true. As an example, consider a bitboard "w_occ" that contains the information which square is occupied by a white piece - all squares corresponding to a "1" are occupied by a white piece, the others are not.

In order to represent a chess position, one "bitboard" is of course not enough - only the combination of several bitboards can contain the complete information of a position. Let's consider the following bitboards:

- one bitboard for each type of piece: "pawns", "knights", "bishops", "rooks", "queens", "kings"
- two bitboards "w_occ" and "b_occ" indicating which squares are occupied by what color
- a collection of bitboards encoding the occupied squares in a "rotated" manner (see 1.2.3.3.4)

In this representation, the white pawns can be obtained by "ANDing" the "pawns"-bitboard (in which the pawns of both colors are encoded) with the "w_occ"-bitboard:

```
white_pawns = pawns AND w_occ
```

Another example is computing the empty squares. For this, the white and black pieces are "ANDed", and then the bitwise complement ("NOT") is formed:

```
empty_squares = NOT (w_occ AND b_occ)
```

By following this idea and using the bitwise operations "AND", "OR", "NOT" etc., many more interesting information can be computed from the bitboards very efficiently.

1.2.3.3 The bitboard-approach towards move-generation The move-generation is used many times during the search-algorithms used in chess programs. Therfor, an efficient move-generation is needed. Based on the bitboard-approach, there exist different strategies for each of the piece-types in chess. One important concept is to compute bitboards of all possible moves (e.g. of a knight) from all the squares beforehand during the initialisation of the program, and store this information in a data-structure that provides efficient access to these pre-computed moves during the move-generation. For non-sliding pieces, this approach works straighforward, but for sliding-pieces some more tricks are needed. which are explained in the next section (1.2.3.3.4). But let's start with looking at generating moves for pawns, which uses a different but very elegant way of using the bitboard-representation.

1.2.3.3.1 Pawns The idea for generating pawn moves using bitboards is based on the "shift"-operations that exist on all microprocessors: by shifting the bitboard containing the white pawns to the left by 8 positions, the non-capturing moves of all (up to 8) white pawns can be generated simultaneously (this shifted bitboard has to be "ANDed" with the empty_squares in order to be valid)! For pawn captures, just shift to the left by 7 and 9 respectively, and "AND" with the black pieces. Although this looks amazingly fast on first sight, in practice some of the advantage of the parallel generation is lost when the moves must be put into a move list seperately (see section 3.1.2 for details). Maybe this could be avoided in some cases, and there are some ideas for future developments (see [2] for details).

1.2.3.3.2 Non-sliding pieces For non-sliding pieces like knight or king all possible moves from all the squares of a chess board are computed during the initialisation of the program and stored in arrays indexed by the from-field, i.e. there exist 2 arrays:

- knight_moves[from-field]
- king_moves[from-field]

In knight_moves[c1], for example, a bitboard that contains all possible "to-squares" for a knight standing on field "c1" is stored. During move generation, this bitboard can be "ANDed" with a bitboard containing the fields that are not occupied by own pieces (i.e. NOT(own_pieces))

to produce all knight moves from c1. But there are more possibilities: if knight_moves[c1] is "ANDed" with the opponents pieces, only capture-moves will be produced (this is needed very often e.g. during quiescence search). In general, very advanced move generation schemes are possible, e.g. "moves that attack the region of the opponent king" could be genrated by "ANDing" the possible to-squares with a bitboard that encode the fields near to the opponent king. These examples show the flexibility of the bitboard-approach.

Although this technique works fine for non-sliding pieces, there are diffculties when starting to think about sliding pieces, which will be covered in the next section.

1.2.3.3.3 Sliding pieces Computing "all possible moves from all squares" for sliding pieces is not as easy as for non-sliding pieces, because the possible moves for a sliding piece will depend on the configuration of the line/file/diagonal it is standing. For example, on a compleatly empty chessboard, a bishop standing in one of the corners will have plenty of moves, but in other positions with own pieces standing next to it and blocking its diagonal, there might be not even one move possible for the bishop. Therfor, the idea for bitboard-move-generation for sliding pieces is to compute all the possible moves for all squares **and** all configurations of the involved ranks/files/diagonals! For example, the rank-moves for a rook standing on a1 on an otherwise empty chessboard will be stored in "rank_moves[a1][00000001]", with the second index of the array beeing the configuration of the involved rank (i.e. 8 bits, with only "a1" beeing occupied as the rook is standing there itself). This works fine for rank-moves, because the necessary 8 bits for the respective rank can be easily obtained from the bitboard of the occupied pieces (this bitboard consists of 8 byte, and each of those corresponds to one rank). For file-moves of rooks and queens, and especially for the diagonal moves of bishops and queens, things turn out to be much more difficult: the necessary bits about the respective files/diagonals are spread all over the "occupied"-bitboard, they are not "in order", as they are for rank-moves. Here the idea of rotated bitboards helps out.

1.2.3.3.4 Rotated bitboards The idea of rotated bitboards is to store the bitboards that represents the "occupied squares" not only in the "normal" way, but also in a "rotated" manner. Therfor, the necessary bits representing files/diagonals are "in-order" in those rotated bitboards, as needed by the move-generation (see previous section). The "rotated bitboards" are updated incrementally during the search, i.e. when a move is done or undone. The following bitboards are maintained:

- board.occ, which represents the occupied squares in the"normal" representation
- board.occ_l90, the board flipped by 90° (for file moves)
- board.occ_a1h8, for diagonal moves in the direction of the a1h8-diagonal
- board.occ_a8h1, for diagonal moves in the direction of the a8h1-diagonal

A detailed description how these bitboards are used during move-generation can be found in sections 3.1.4. See the Appendix (6) for details about how the different rotated bitboards look like.

1.2.4 Search Algorithms

For the search in computer chess, Shannon ([19]) descibes 2 possibilities:

- type A strategy: seaching all possible moves from a position up to a given depth
- type B strategy: search only the "reasonable" moves, but with the possibility to search deeper, cause the tree is smaller

In the early days of computer chess, much hope was put into the "type B" strategy, but in practice, the "type A" programs (with some modifications) prooved to be more successful. It was just too hard to construct a good "plausible move generator" that generates only "reasonable" moves. (see e.g. [8], p.69-73). On the other hand, most chess programs today use some selective "forward pruning"[2] techniques (like nullmoves etc, see below), which one can argue lets them stop beeing a pure "type A"-program, but moves them a bit in direction of the "type B" programs. This combination of a basic full-width search, with several extensions and some selected forward pruning techniques has turned out to be most successful in practice and is also used in Fusc#. Concretely search algorithm in Fusc# uses the following techniques:

1. an alpha-beta search with an "aspiration window"
2. a basic quiescence search (assuring that the evaluation function is only applied in quiescent positions)
3. iterative deepening

For more details about "Aspiration window Alpha-Beta"[3] (AAB), please refer to [14].

For move ordering in Fusc#, static as well as dynamic heuristics are used. The following dynamic move ordering heuristcs have been implemented in the course of this work:

1. the killer heuristic
2. the history heuristic
3. the refutation heuristic

How these work exactly in FUSc# is constituting a major part of this work. For an explaination of the theory behind it see section 2.5. For an explanation of the concrete implementation in FUSc# please refer to section 3.2.

1.2.5 Other Aspects

Additionally to the ones described above, the following concepts are used in FUSc#:

- transposition tables (see [4])
- an evolutionary evaluation function (see [1])
- an evolutionary opening-book
- nullmove pruning

Some of the papers referenced above were written in the course of the seminar "Schachprogrammierung" (lead by Marco Block and Prof. Raúl Rojas in the summer term 2005 at the Free University Berlin) and can be downloaded from the appropriate section of the FUSc#-Homepage ([6]).

[2] this term is the complement to the "backward pruning" that is done by the alpha-beta algorithm described below

[3] the idea behind the aspiration window is that when doing interative deepening, the value of the bestmove for the next iternation is likely to be close to the value of the last iteration. By starting the alpha-beta-search at the root not with the normal bounds of alpha set to minus infinity and beta set to plus infinity, but with expected_value-100 and expected_value+100, for example, the search will be more efficient due to the better bounds provided. However, if the computed value falls outside this "alpha-beta-window", a re-search has to be done in order to obtain the correct value.

1.2.6 Fusc#-Server

There exists an online-server ([7]) where people can play against FUSc#. This server can also be used for testing-purposes of different versions of FUSc#, as it has been the case in the work "*Verwendung von Temporale-Differenz-Methoden im Schachmotor FUSc#,*" written by the FUSc#-Team member Marco Block. See [1] for more details on that topic.

2 Theoretical aspects of alpha-beta-searches

This section is not meant as an introduction to people completely new in the field of tree seaching in artificial intelligence (see [15] for a good book as an introduction to both the theoretical as well as the practical aspects of artificial intelligence, which is even containing an interesting part of the philosophical background of AI). In particular, this section does not provide an introduction to the alpha-beta algorithm. This has been done many times, and there exist excellent papers on the subject. A good overview is found in the article "Tree Seaching Algorithms" by H.Kaindl (see [14]). The aim of this section is to give a concise review of the state-of-the-art in chess programming theory. Both the historical roots as well as recent research is referenced, ranging from the beginnings of the 1950s till the end of the 1990s.

2.1 Foundations of Tree-Searching Algorithms in Computer Chess

In terms of mathematical game theory, chess is a "two player zero sum" game with "perfect information". Let us have a look at the parts of this definition:

- "two player" means there is a fixed number of two players (i.e. black and white in chess)
- "zero sum" means that a gain for one player equals the loss of the other (i.e. "both winning" is not possible, an advantage for white always means a disadvantage for black)
- "perfect information" means that both players have the complete relevant information of the game (i.e. the configuration of the board) available at all times of the game - there is no random or chance factor like e.g. in card games

Games like chess have always been an important application area for heuristic algorithms. The basic idea of such a heuristic search algorithm is to construct a tree (the "game tree") of possible moves for each player and find the best move by traversing this tree. As this tree grows exponentially, only for very simple games like "tic-tac-toe", it is possible to compute the whole tree and thus play perfectly. In computer chess it is not possible in practice to search the game tree until a "final" position (i.e. mate or stalemate) is reached, instead the search must terminate at a certain depth (the "horizon") and use a heuristical evalutation function in order to estimate of the value of the position. As early as 1950, Claude Shannon published a much referenced paper on algorithms for playing chess ([19]). In this paper, many of the concepts that still form the basis of todays chess programs are developed (also see section 1.2.2).

2.2 Minimax and Alpha-Beta

2.2.1 Minimax

We quote Shannon descibing the process of computing the minmax-value of a position. In the follwing lines, `f(P)` should denote the heuristical evaluation function (i.e. an integer) of a position P.

> "A strategy of play based on f(P) and operating one move deep is the following. Let M_1, M_2, M_3, ..., M_s be the moves that can be made in position P and let M_1P, M_2P, etc. denote symbolically the resulting positions when M_1, M_2, etc. are applied to P. Then one chooses the M_m which maximizes f(M_mP).

A deeper strategy would consider the opponent's replies. Let M_i1, M_i2, ..., M_is be the possible answers by Black, if White chooses move M_i. Black should play to minimize f(P). Furthermore, his choice occurs _after_ White's move. Thus, if White plays M_i Black may be assumed to play the M_ij such that f(M_ijM_iP) is a _minimum_. White should play his first move such that f is a maximum after Black chooses his best reply. Therefore, White should play to maximize on M_i the quantity min f(M_ijM_iP) M_ij "

The key assumption is that in a game between two players, MAX and MIN, MAX on the move always selects the *maximum* of the values for its children whereas MIN always takes the *minimum*. Applying this rule recursively over the entire game tree, a *minimax value* can be computed. However, this computation turns out to be very complex even for relatively small search depths. Assume a constant branching factor of w, then the tree for a depth d has as much as w^dnodes. In the game of chess, there is an average branching factor of about 40 ([8], p.61), which lets the tree grow extremely quickly (for middle-game positions, minimax searches of more than a few plies are beyond the computational power of todays computers). However, there were numerous improvements of classic minimax, one of the most important beeing the alpha-beta algorithm.

2.2.2 Alpha-Beta

The basic idea of the alpha-beta is to prune away "irrelevant" parts of the tree that will have no influece on the outcome of the search anyway. As minimax examines the whole game tree, it spends much time on searching compleately irrelevant positions. The Alpha-Beta-algorithms saves time by using the following idea: if it is already known that a certain move is worse than the current bestmove, do not spend time to compute exactly how much it is worse but continue with the next move in order to check it this one is better. Thus, big parts of the search-tree of minimax can be pruned away by obtaining so-called "alpha-cutoffs" and "beta-cutoffs". For more details and a descrition of the different flavours of alpha-beta-algorithms used in chess programms, a good overview can be found in the article "Tree Seaching Algorithms" by H.Kaindl (see [14]).

The efficiency of the alpha-beta-algorithm is depends heavily on the order in which it examines the moves (see section 2.4).

2.3 Depth First vs. Best-First

Minimax as well as the alpha-beta algorithm described above are depth-first algorithms, i.e. they search the game tree by exploring the first branch until the the deepest level (the horizon) is reached, and continue in a rigid left-to-right order. However, there exist other possibilities to compute the minimax-value of a position, like the "best-first"-strategy.

2.3.1 Traditional View on Best-First-Algorithms

The basic idea of the "best-first"-strategy is to search nodes that look most promising first. Therefor, the whole game tree is kept in memory (i.e. the storage requirements are higher than for plain alpha-beta), and the search uses all information avalailable to decide which path is most interesting to search next. The tree is not searched in a rigid left-to-right order, but the search "jumps" between different parts of the tree that are searched according to the prediction which parts are most promising. Another way of putting it is that while depth-first algorithms search the individual nodes in the game tree, the best-first-algorithms search for the best "minmax-solution tree" among all possible solution trees (for details, see [21]).

One example is the SSS*-algorthm, which was introduced in 1979 ([22]), and it was proven to be better than alpha-beta in the sense that it "never expands a node that alpha-beta does not expand", but there are cases where it expands cosiderably less nodes, according to some results obtained by several researchers in the 80s and early 90s. The widespread opinion was that best-first-algorithms like SSS* are more efficient than alpha-beta, but that they are unusable in practice

because of their enormous storage requirements. However, neither of the propositions of the last sentence is true, as is shown in the next paragraph.

2.3.2 SSS* as a Special Case of Alpha-Beta

Aske Plaat was able to show an astonishing fact in 1996: SSS* can be reformulated as a special case of alpha-beta! For this, the use of transposition tables (i.e. "memory-enhanched"-alpha-beta) is necessary, which leads to the short formula:

SSS*=alpha-beta+transposition tables

Contrary to earlier thoughts, Plaat was also able to show that in practice, SSS* does not need more memory than alpha-beta. He simply used his alpha-beta-reformulation of SSS*, called MT-SSS* (that he prooved would examine the nodes in the game tree in exactly the same order as SSS*), and showed that in practice, the memory-requirements were the same as with alpha-beta. But additionally, he was able to show that in game-playing practice, SSS* does not evaluate significantly less nodes than Alpha-Beta, given equal memory ressources. The theoretical and simulation results that had indicated the supremacy of SSS* compared to alpha-beta of the past were mostly based on "artificial trees, that lack essential properties of the as they are searched by actual game-playing programs" ([16], p. 5).

So the unexpected result of his research can be summarized with the sentence "the reasons for ignoring SSS* have been eliminated, but the reasons for using it are gone too" ([16], p.4). He even states that "we believe that SSS* should from now on be regarded as a footnote in the history of game tree search"[4].

2.4 The Influence of the Move-Ordering

It is easy to see that the alpha-beta-algorithm yields the same result (both the same bestmove as well as the same value for the bestmove) like the minmax-algorithm. However, the number of nodes visited can be quite different, and alpha-beta can perform much better that minmax. How much better it will perform depends on the order the moves are searched. In order to illustrate this, the two extreme cases are treated: the best case (i.e. the move ordering is perfect, best moves are always searched first) and the worst case (i.e. the worst moves are searched first).

2.4.1 Alpha-Beta (Worst Case)

In the worst-case, no cutoffs occur, cause the moves are searched in exactly the wrong order. Thus, the alpha-beta-algorithm degenerates to plain minmax, with complexity w^d.

2.4.2 Alpha-Beta (Best Case)

In the best case, the move ordering is perfect, i.e. best moves are always searched first and the number of cutoff is maximized. Slagle/Dixon ([20]) showed that the complexity of the best case depends on the fact whether d is even or odd:

- d even: $2w^{d/2} - 1$
- d odd: $w^{(d+1)/2} + w^{(d-1)/2} - 1$

This formula also has an influence on the growth rate of alpha-beta trees, which is not identical for the transition from odd to even depths compared to the transition from even to odd depths. Additionally, the difference between the best case and the worst case is getting bigger with increasing search depths (compare the practical results in section 4).

[4] Plaat develops a complete framework to give a unified view on the different depth-first alpha-beta algorithms as well as the best-first algorithms like SSS* and DUAL*. Additionally, he describes a new instance of this framework called "MTD(f)" which outperforms all known search algorithms both in computation time as well as on the amount of nodes visited.

2.4.3 Minimal Trees

Historically, the best case of alpha-beta has been treated as beeing the "minimal tree" that has to be searched by any tree search algorithm, thus beeing the asymtotic optimal case. However, Plaat has shown that this view is not correct: At first, due to transpositions, one should speak of a "minimal graph" instead of a "minimal tree". In his terminology, we have 2 graphs:

- the "Left-First-Minimal-Graph" (LFMG), which is best case of alpha-beta
- the "Real Minimal Graph" (RMG), which can be considerably less than the LFMG

The problem of the LFMG is that when it examines a cutoff, it can not be assured that this is the cutoff eliminating the biggest part of the search tree, and thus leading to the smallest search tree. The LFMG just takes the first cutoff it gets during the traversal of the search tree (which is done in a left-to-right manner, thus the name "left first"). However, there might be cutoffs that are missed by the LFMG that achieve the same result with less search effort. The *real* minimal graph (RMG) must always select the cutoff that leads to the smallest search tree. Due to the irregular branching factor observed in practical game trees, as well as due to transpositions, it is difficult if not impossible to compute the RMG in practice, cause the size of the subtrees is inter-dependet, as for example searching subtree A first might make it cheaper to search subtree B afterwards because of transpostions. Plaats summarizes his results with the follwing sentences: "In other words, minimizing the tree also implies maximizing the benefits of transpositions. Since there is no known method to predict the occurrence of transpositions, finding the minimal graph involves enumerating all possible sub-graphs that prove the minimax value, and thus computing the real minimal graph is a computationally infeasible problem for non-trivial search depths" ([16], p. 92).

But there are methods to approximate the RMG, leading to the approximate RMG (ARMG). Details are given in [16], and these results show that there is still more room for improvement of search efficiency than belived before, where the best-case of alpha-beta (the LFMG) was believed to be the theoretical optimum for search algorithms.

2.5 Heuristics for Achieving a Good Move-Ordering

As a good move-ordering is so crucial for achieving efficient alpha-beta-searches in computer chess (see section above), static as well as dynamic heuristics have been developed during the past decades. Static move ordering means that moves are sorted inside the move-generator according to some static heuristics that are independent from the position of the current node in the tree ("the move-generator only sees the position, and does not have any knowledge about the search tree). Dynamic move ordering tries to use information gathered during the search in order to decide which moves to search first. They are mostly domain-independent, i.e. they can be used for other games, too. We describe the following dynamic move ordering heuristcs here:

- the killer heuristic
- the history heuristic
- the refutation heuristic

How these work exactly in FUSc# is constituting a major part of this work. In this section, the basic idea of the heuristics is explained. For an explanation of the concrete implementation in FUSc# please refer to section 3.2.

2.5.1 Static Move Ordering

Inside the move-generator, each move is already assigned a value based on some static heuristics. These are different for capture and non-capture moves.

2.5.1.1 Capture Moves

In order to assgin a static value to a capture move, the capturing piece ("aggressor") as well as the captured piece ("victim") must be taken into consideration. A common strategy is the "MVV/LVA"-heurstic. At first, capture moves are sorted according to the "Most Valuable Victim" (MVV), then according to the "Least Valuable Aggressor" rule. How these rules are translated into the practive differs among the different chess programs, as some have a feature called "Static Exchange Evaluator" (SEE) that tries to statically evaltuate captures by considering how many attacks are there on the target fields. The aim is to be able to distinguish between winning and loosing captures already in the move generator, without any search! However, this turns out to be quite difficult, as normally only captures and re-captures on the same field are considered, and more advanced threats like mate or pins are not included. Also, the amount of information that is available in the move generator varies and is closely related to the data structures used in a chess program (e.g. some engines maintain a structure saving the number of attacks from/to a square incrementally).

2.5.1.2 Non-Capture Moves

In order to have an easy-to-compute hierachy for non-capture moves, the "piece-square-tables"[5] that are used in the evaluation-routine of Fusc# are also used to statically sort moves in the move-generator. Thus, if a piece moves from one square to a new square, that move is attributed the differences of the values of the squares according to the piece-square table for the moving piece. The formula is:

value = piece_square_table[piece][to] - piece_square_table[piece][from]

Take as an example a move where the knight moves from the edge of the board closer to the center. It will be attached a positive value, as the value of the from-square will be lower than the value of the to-square for the knight (which models the rule of thumb for chess that knights should not stand near the edges of the board, as it limits their number of moves).

2.5.2 The Killer Heuristic

The basic idea of the killer heuristic is that moves that have been good (i.e. causing a cutoff) at a certain depth should be tried again in the same depth. It was first described by Slate/Atkin in [8]. A simple implementation will just record one killer move at each ply, but already Slate/Atkin describe why this is not optimal. They suggest to keep 2 killer moves for each ply, and maintain a counter for both of them. Each time a cutoff occurs, the cutoff-move will be compared to the killer moves: if it corresponds to one of them, then the respective counter is incremented, if is not among them, then the killer-move with the lower counter is replaced by the cutoff-move. However, this heuristic can even be further improved, as we show in section 3.2.

2.5.3 The History Heuristic

The idea behind the histroy heuristic is that moves that have been good "on average over the whole tree" (i.e. that have good history value) should be tried earlier than others. For this purpose, a "history-value" is maintained for each possible move (normally, it is saved in an array indexed by the from and the to-square of the move) that is updated each time a cutoff occurs or a new best move is found. The history value is incremented by the distance to the horizon that a move was searched - the idea is that moves that are prooven to be good after a deep search should be awarded a bigger bonus that those that result from only shallow searches.

[5] A piece-square-table saves values for all squares for a specific piece in an array. These values indicate wheather it is generally considered to be good for the piece to stand on this field (i.e. higher value), or if it is a rather bad square for the piece. Static piece-square-tables are easy to implement, however there are also some ideas to modify them according to the current position on the board (e.g. analyse the postion before the search starts and fill the piece-square-tables appropriately).

2.5.4 The Refutation Heuristic

The idea of the refutation heuristic is that moves that have been good as an answer to a certain opponent's move should be saved and tried first in other parts of the tree if the last of the opponent's move has been the same. Thus, in the search tree, each time a cutoff occurs, this cutoff move is saved as refutation of the move that was made by the opponent one ply earlier. One example where this heuristic can be useful is a situation like the following: Imagine black has a piece standing on a square attacked by white, but black has his piece covered by another black piece. Now if black moves the piece that is covering the square, the other piece is left "enprise" and can be captured by white without any danger. In this situation, the capture would be the "refutation-move" for the move that moves the covering piece. Even though the general idea behind the refutation heuristic has been mentioned by other authors, the heuristic as we describe it here is to our knowledge only implemented in FUSc# (but e.g. some chess programs attribute a bonus to moves that capture the piece that was moved last by the opponent, which could be seen as a primitive try to "refutate" the last opponents move)

3 Practical strategies for efficient alpha-beta-searces - The FUSc# Source Code in Detail

As described in 1.2.1, FUSc# is the chess program developed by the "AG Schachprogrammierung" at the Free University in Berlin. It is written in C# and runs on the Microsoft .NET Framework (see [12]). In this section, a closer look will be taken to selected parts of the FUSc#-source-code, which can be downloaded from [6], as FUSc# is an open-source-project.

3.1 Prerequisites: Efficient Move Generation

This section aims to give an overview of how the move generator of FUSc# works. The movegenerator has been described in detail in [2]. At first an overview of move generation in FUSc# is given, then the generation of moves for the different pieces is explained. As explained in section 1.2.3.3, there are three main categories of piece-types for move generation:

1. Pawns (capturing/non-capturing moves)
2. Non-sliding pieces (knight, king)
3. Sliding pieces (bishop, rook, queen)

For each of these categories, one example is treated below (for the other pieces have a look at [2]). The steps involved in the move-generation in FUSc# are explained and illustrated by some snippets from the source-code. However, for these explainations, not the latest (fine-tuned, and therefor quite unreadable) version of the source-code of the FUSc#-move-generator will be used, but an earlier version where the concepts involved can be seen much clearer. Those concepts of course still form the basis of the move-generator of the latest versions of DarkFUSc# (our new engine, which can be downloaded from [5]). When we speak about "FUSc#" in the following section (like in the whole paper), we actually mean the current "DarkFUSc#"-engine, as beeing the latest member of the FUSc#-family (see 1.2.1 for more information about the FUSc# project history).

3.1.1 Overview of Move Generation in FUSc#

We will discuss now in detail how the move generator in FUSc# works. We will only deal with "movegen_w" that generates moves for white - there is a symmetrical routine for black, which is based on the same ideas and will not be treated here. The call for "movegen_w" is:

```
int movegen_w(Move[] movelist, ulong from_squares, ulong to_squares)
```

You can see that movegen_w expects 3 parameters:

- a "movelist" to store the generated moves in
- a bitboard (ulong is 64bit in .NET!) of "from_squares", which is normally "board.w_occ", i.e. all white pieces
- a bitboard of "to_squares", which is normally "~board.w_occ", i.e. the complement of all white pieces, but could also be e.g. "board.b_occ" to generate only capture moves

In the following sections we will discuss the move generation for some of the pieces in detail.

3.1.2 Pawns

In this work, only non-capturing moves for pawns will be considered (see [2] for the rest). Below you find the code-snippet from the FUSc#-move-generator that generates (one step) non-capturing pawn moves for white:

```
1  // WHITE PAWNS (one step)
2  pawn_fields_empty = ( (board.pawns & from_squares) << 8) & (~board.occ.ll);
3  tos = pawn_fields_empty & to_squares;
4  froms = tos >> 8;
5  while (from = GET_LSB(froms))
6  {
7  board.w_attacks |= from;
8  movelist[movenr].from = from;
9  movelist[movenr].to = GET_LSB(tos);
10 movelist[movenr].det.ll = 0;
11 movelist[movenr].det.ail.piece = PAWN;
12 movelist[movenr].det.ail.flags = 0;
13 movenr++;
14 CLEAR_LSB(tos);
15 CLEAR_LSB(froms);
16 };
```

In line 2, the idea is to compute a bitboard of all the "empty squares in front of white pawns". To get this, the pawns (standing on the "from_squares") are shifted to the left by 8 bits, and the result is "ANDed" with the complement of the "occupied" squares (found in "board.occ.ll"). Then, this "pawn_fields_empty" is "ANDed" with the "to_squares" in order to get the destination squares ("tos") for all the desired moves. The from-squares ("froms") for those moves can be obtained by shifting back the "tos" by again 8 bits. After line 4, all one-step non-capturing pawn moves (that origin from "from_squares" and head to "to_squares") have been genrated and are encoded in the two bitboards "froms" and "tos". In the while-loop in lines 5-16 those moved are put into the movelist indivudally. Therfor, the individual moves that correspond to the bits in the bitboard "froms" and "tos" must be obtained one-by-one. In line 5, the "Least Significant Bit" (LSB) of "froms" is extracted and saved in the bitboard "from", and in line 9 the same is done for the LSB in "tos" (it is saved in the bitboard "to"). These two bitboards, together with some additional information (like the piece that is moving) is then saved in the movelist (lines 7-12). In lines 13 and 14, the "Least Significant Bits" of "froms" and "tos" are cleared, as this was the move that has just been processed. If there is are bits left in "froms", then the next iteration of the while-loop will extract them, otherwise the generation of one-step non-capturing pawn moves is finished.

Two-step non-capturing pawn moves are generated similarily.

3.1.3 Non-Sliding pieces

As an example for non-sliding pieces, knight moves will be treated. Here is the code-snippet from the FUSc#-move-generator that generates moves for the white knight:

```
1  // WHITE KNIGHT
2  froms = board.knights & from_squares;
3  while (from = GET_LSB(froms))
4  {
5  from_nr = get_LSB_nr(from);
6  tos = knight_moves[from_nr] & to_squares;
7  while (to = GET_LSB(tos))
8  {
9  board.w_attacks |= from;
10 movelist[movenr].from = from;
11 movelist[movenr].to = to;
12 movelist[movenr].det.ll = 0;
13 movelist[movenr].det.ail.piece = KNIGHT;
14 movelist[movenr].det.ail.from_nr = from_nr;
15 movelist[movenr].det.ail.flags |= FROM_NR_COMPUTED;
16 if (board.b_occ & to) movelist[movenr].det.ail.flags |= NORMAL_CAPTURE;
17 movenr++;
18 CLEAR_LSB(tos);
19 };
20 CLEAR_LSB(froms);
21 };
```

Generating moves for the white knight starts in line 2, where "board.knights" (containing the knights of both colors) is "ANDed" with the "from_squares" (which normally contain all the white pieces). The result (a bitboard containing the white knights) is saved in "froms". In line 3 the LSB of "froms" is extracted and saved in the bitboard "from", which then only contains one bit set (at the position where the first white knight resides). Then, in line 5, the number of the bit set in "from" is computed by the routine "get_LSB_nr(from)" and saved in "from_nr". The "from_nr" is needed to index the array "knight_moves" in line 6 (this array contains all ever possible knight moves from the square that is given as index, see section 1.2.3.3.2). The destination squares for knight-moves ("tos") are computed by "ANDing" the "knight_moves[from_nr]" with the "to_squares", which could be all empty squares or all black pieces, e.g., if only the generation of certain types of moves is desired (capturing/non-capturing). After that, the generated moves are put in the movelist in lines 7-21.

3.1.4 Sliding pieces

As an example for non-sliding pieces, rook moves will be treated. Here is the code-snippet from the FUSc#-move-generator that generates moves for the white rook:

```
1 // WHITE ROOK
2 froms = board.rooks & from_squares;
3 while (from = GET_LSB(froms))
4 {
5 from_nr = get_LSB_nr(from);
6 rank_pattern = board.occ.byte[from_nr >> 3];
7 file_pattern = board.occ_l90.byte[l90_to_normal[from_nr] >> 3];
8 tos = (rank_moves[from_nr][rank_pattern] | file_moves[from_nr][file_pattern])
& to_squares;
```

```
9 while (to = GET_LSB(tos))
10 {
11 board.w_attacks |= from;
12 movelist[movenr].from = from;
13 movelist[movenr].to = to;
14 movelist[movenr].det.ll = 0;
15 movelist[movenr].det.ail.piece = ROOK;
16 movelist[movenr].det.ail.from_nr = from_nr;
17 movelist[movenr].det.ail.flags |= FROM_NR_COMPUTED;
18 if (board.b_occ & to) movelist[movenr].det.ail.flags |= NORMAL_CAPTURE;
19 movenr++;
20 CLEAR_LSB(tos);
21 };
22 CLEAR_LSB(froms);
23 };
```

For generating rook-moves, the idea of "rotated biboards" (section 1.2.3.3.4) comes into play. But at first, the white rooks are computed and extracted in lines 2-3, and the number of the square where the rook is standing is computed is line 5 and stored in "from_nr" (see previous sections for details). In lines 5 and 6 patterns of the rank and the file on which the rook is standing is saved in "rank_pattern" and "file_pattern" respectively. These patterns are 8-bit variables that are used to index the "rank_moves" and "file_moves"-arrays in line 8, in addition to "from_nr", containing the square where the rook is standing (see section 1.2.3.3.3 for details). In line 7, you can see how the idea of accessing the "rotated" representations of the occupied squares works in practice: The desired file-pattern is found in

"`board.occ_l90.byte[l90_to_normal[from_nr] >> 3]`"

Let's look at the individual parts of this expression:

- "board.occ_l90" contains a bitboard of the occupied squares, shifted by 90° to the left
- this bitboard consists of 8 bytes (i.e. 64bits), that can be accessed individually by "board.occ_l90.byte[0]" to "board.occ_l90.byte[7]"
- in order to get the correct byte-number, the "from_nr" is converted to the "l90"-square-nr by accessing the array "l90_to_normal" with index "from_nr" and shifted to the right by 3 bits
- this last shift can also be seen in line 6. When shifting "from_nr" (a number from 0...63) to the right by 3 bits, you will get the number of the byte where the bit corresponding to the "from_nr" resides

Thus, after line 6 and 7, you have the correct patterns stored in "rank_pattern" and "file_pattern". These are used to access the pre-computed "rank_moves" and "file_moves" arrays in line 8, where the bitboard of the possible destination squares for rook-moves ("tos") is computed. The individual moves are put in the movelist in lines 9-23 as decribed above.

3.2 Dynamic Move-Ordering in Fusc#

The current FUSc# source code contains the following dynamic move ordering heuristcs:

1. the killer heuristic
2. the history heuristic

3. the refutation heuristic

In this section, a brief description of the data structures used by each heuristic is given. If you are interested in more detials about these heuristics, e.g. how they are updated when a cutoff occurs, or in understanding the procedure of applying each of the heuristics to concretely sort the generated moves inside the search algorithm, please have a look at the FUSc#-source-code yourself, as describing the steps involved in a line-by-line fashion would go beyond the scope of this work.

3.2.1 Killer Heuristic

There exists two versions of the killer-heuristic in FUSc#: the "simple killer heuristic" and the "advanced killer heuristic" (see section 2.5.2).

3.2.1.1 Simple Killer Heuristic The simple killer heuristic maintains 2 killer moves for each ply, and saves a counter for both of them. Here is the appropriate data structure used in FUSc#:

```
public struct KillerMove
{
public Move killer1;
public sbyte counter1;
public Move killer2;
public sbyte counter2;
}

KillerMove[] s_killer_at_ply = new KillerMove[DFConstants.MAXDEPTH];
```

As an example, s_killer_at_ply[5].killer1 contains the first killer move at ply 5. Each time a cutoff occurs, the cutoff-move will be compared to the killer moves: if it corresponds to one of them, then the respective counter is incremented, if is not among them, then the killer-move with the lower counter is replaced by the cutoff-move.

3.2.1.2 Advanced Killer Heuristic The advanced killer heuristic uses the same data-structure as the simple killer heuristic to save 2 killer moves for each ply. However, additionally, a "killer-value" for each move is maintained, indexed by the "from" and the "to"-square (comparable to the way the "history-value" maintained for the history-heuristic, see below)

```
int[,,] s_killer_value = new int[200,64,64];
```

The killer-value of a move is stored in "s_killer_value[depth,from,to]" and is incremented each time a cutoff occurs for the move causing the cufoff. Therfor, the search keeps track in detail how often each move has caused a cutoff at a certain ply. This is more advanced than just maintaining counters for the two most frequent cutoff-moves that are saved as killer-moves in the simple killer heuristic (in the simple case, also the value of the counter for a killer-move is lost as soon as the move is replaced. This can lead to a situation where a very good killer move is still replaced frequently, because the simple killer heuristic has no way of saving "globally" how often each move has already caused a cutoff for each depth. This is exactly the idea behind the "s_killer_value"-array in the advanced killer heuristic).

3.2.2 History Heuristic

The data structures for the history heuristic are the following:

```
public struct HistoryMove
```

```
{
public Move history1;
public sbyte value1;
public Move history2;
public sbyte value2;
}

HistoryMove s_global_history;

// the history-value of a move is stored in s_history_value[from,to]
int[,] s_history_value = new int[64,64];
```

Unlike the killer-heuristics described above, the history-heuristic does not maintain history moves for each ply, but does save "global" history moves for the whole search tree. A "history-value" is maintained for each move in an array indexed by the "from" and the "to"-square. This history-value is augmented each time a cutoff occurs for the move causing the cutoff (see 2.5.3 for details).

3.2.3 Refutation Heuristic

For the refutation heuristic, the following data-structures are maintained during the search:

```
public struct RefutationMove
{
public Move move;
public bool exists;
};

RefutationMove[,] s_refutation = new RefutationMove[64,64];
```

Each time a cutoff occurs, the cutoff-move is saved as refutation of the move the opponent had done before. For this move, that was done before by the opponent, the flag "exists" is set to true, so that the refutation can be tried the next time the move occurs somewhere else in the search tree.

4 Practical Experiments with FUSc#

4.1 Verifying the Move-Generator of Fusc#

Constructing a basic move-generator is not too hard, since the basic rules for piece-movements in chess are manageable both in number and complexity. However, when also considering special moves like castling, promotion and en-passent and the huge number of possible chess positions there are some really tricky cases to handle - and the question arises how to make sure that the move-generator of one's chess program works 100% correct, even in awkward and seldom ocurring yet possible positions. "Manually" checking the move-lists of the program is possible for only a very limited number of positions - nevertheless it should of course be done in the process of developing a chess program, although it can always only be a first step. A more advanced method to verify the move-generator of a chess engine have been developed is to use the command "perft".

4.1.1 The "perft"-Idea

The basic idea is to implement a "perft"-command to the chess engine which will construct a minmax-tree untill a fixed depth and count all the generated nodes. This number can be compared to the number of nodes generated by the "perft"-command of other chess engines, and there

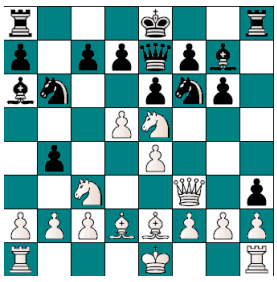
exist Web-Sites with both a collection of chess positions and the corresponding correct "perft"-numbers for several depths (see [11]). Of course, special attention should be given to positions involving "special moves" like castling, en-passent and promotion. One important point is that the search conducted by the "perft"-command must construct a plain minmax-tree without alpha-beta, transpositions tables, quiescence search, search extensions or any forward pruning techniques like null-moves.

4.1.2 Test Positions

4.1.2.1 The Start Postion The correct results for the "perft"-command at the start position are given in the following table. It is clear that for depth 1 (i.e. "move generation for white and counting the nodes") the result is 20, as there are 20 legalmoves for white in the start position in chess:

Depth	Perft(Depth)
1	20
2	400
3	8,902
4	197,281
5	4,865,609
6	119,060,324
7	3,195,901,860
8	84,998,978,956
9	2,439,530,234,167
10	69,352,859,712,417

4.1.2.2 A Middlegame Position The following position involves castling, en-passent and promotion (at least in higher depths) for both sides.
The FEN-Code is r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq -

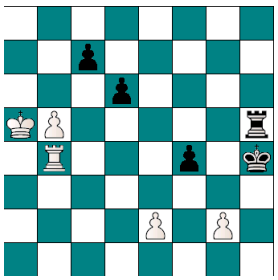


The correct results are:

Depth	Perft(Depth)
1	48
2	2039
3	97,862
4	4,085,603
5	193,690,690
6	8,031,647,685

4.1.2.3 An Endgame Position

Here is an endgame-poistion with FEN-code 8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - -



The correct results are:

Depth	Perft(Depth)
1	14
2	191
3	2,812
4	43,238
5	674,624
6	11,030,083
7	178,633,661

4.1.3 Results of Crafty and FUSc#

In order to test the move-generator of FUSc#, the three positions described above were loaded into the program and the "perft"-command was executed. In order to re-check the results, the experiment was also done with Crafty (version 1919p3, see [3]). The result is that the FUSc#-move-generator works 100% correct! See the Appendix (6.2) for the detailed output of both programs in the three positions.

4.2 Status of the Fusc#-Search-Algorithm before the Search-Experiments

4.2.1 Getting it Deterministic

Before starting the experiments on the influence on the different heuristics for achieving a good move ordering, the first thing to do was assuring that Fusc# plays 100% correct and deterministic, because otherwise, the results would be worthless. The correctness of the move generator was prooven in section 4.1. In order to get the Fusc#-search-algorithm 100% deterministic, we had to check the following things:

- the hash-table implementation: in the past, the hash-table algorithm had been initialized with random values. This has been changed so that we have a deterministic functioning of the hash-tables by either hard-coded values or values read by an initialization file (which can be changed by a special parameter in the source)
- In the beginning of our developments on Fusc#, we faced the problem that FUSc# was not able to learn from its mistakes (the evolutionary opening book was not implemented yet), so when somebody found a way to win against FUSc#, he or she could play this line again and again, and FUSc# would play exactly the same (bad) moves over and over. Therfor, we introduced a small random factor that was added to the score of the evaluation function, and thus changing the play of FUSc# a bit each time (we called this version RANDOMFUSC#). Of course, we had to switch it off before the experiments.

- the heuristics that we implemented for dynamic move-ordering have to be cleared every time before a search, so that no "old values" influence the search

In the end we could observe a 100% deterministic behaviour of the search of FUSc#.

4.2.2 Introducing a Framework for Conducting the Experiments

4.2.2.1 New Commands for an Automated Performing of the Tests In order to run automated tests, several new commands were introduced to FUSc#. They can be entered directly at the command prompt after FUSc# when started from the command-line. Normally, the engine understands only UCI[6]-commands, but for debugging and testing purposes, some additional commands were implemented. All of these start with "debug", one example is "debug board", which displays the current configuration of the boards, or "debug boards", which displays some of the bitboards that are used internally for board representation (see 1.2.3.2). A complete list of these additional commands can be obtained with "debug help". In the course of this work, we implemented the following new commands:

- "debug searchpositions positions.fen depth" starts a search on each of the positions contained in the file "positions.fen" to a fixed depth that is passed as second parameter. The positions must be given in the by their FEN-codes, a standard format for describing chess positions.
- "debug testfile commands.txt" passes the commands that are stored in the file "commands.txt" to the FUSc#-engine. Additionally, we implemented the possibility to specify a file with commands in the command line, e.g. by typing "Fusch commands.txt".
- "debug loadparams params.txt" loads the engine-parameters specified in the file "params.txt" (e.g. specific heuristics can be turned on/off), and thus configures the engine according to these parameters

4.2.2.2 Batch-Mode and Machine-Readable Output In order to do an automated testing of the FUSc# search algorithm based on the newly commands described above, it was necessary to implement a new special playing-mode we called "BatchMode" which should be set active when FUSc# is called not for normal game-playing purposes but for automated testing, e.g. from a Batch-file. The Batch-Mode assures absolute deterministic playing (see above), and disables unwanted features like e.g. pondering that are not useful when running automated tests. A second extension was a parameter defining if FUSc# should format its output in the normal human-readable manner, or rather in a "MachineReadable"-manner that can be better processed further when doing some test series. If the "MachineOutput"-parameter is set to "true", then FUSc# will only output the current search depth and the number of nodes searched in a csv[7]-like notation. This is much easier to process (e.g. for visualization of the data) than trying to parse the output that FUSc# generates when playing "normally".

4.2.3 Move Ordering in DarkFusc# 0.9

DarkFusc# 0.9 had only some very basic move ordering implemented, based on the transposition tables and static piece-square tables[8]. As shown in section 2.4, the move-ordering has a great influence on the efficiency of alpha-beta searches. The aim of our experiments was thus to implement more sophisticated heuristics for move-ordering (like killer-moves etc.) and to test which combination and priority of those heuristics prooves to be most successful in achieving the smallest tree in typical ches positions.

[6] "unversal chess interface", the standard protocol for communicating with a chess engine and the successor of the older "winboard"-protocol

[7] csv ("comma seperated values") is a standard for encoding tables and data in a very simple form and can be read and written by Mircosoft Excel, e.g.

[8] In the terminology used later in this section, the efficiency of the move-ordering of the old FUSc# was probably somewhere between the "static" and the "static+TT" version, as we did also major improvements to the static move ordering in the course of this work.

4.3 Measuring the Influence of the Different Heuristics

The following dynamic move ordering heuristcs have been implemented in the course of this work:

1. the killer heuristic
2. the history heuristic
3. the refutation heuristic

In this section, we descibe the experiments that were conducted in order to verify that the heuristics work correctly. An explaination of the theory behind it is given in section 2.5. For an explanation of the concrete implementation in FUSc# please refer to section 3.2.

4.3.1 General Experimental Setup

In order to gain some insights in how far the heuristics we implemented serve the goal of reducing the number of nodes that need to be searched, a number of fixed-depths searches were conducted by FUSc# in different positions, with the relevant heuristics beeing tuned on/off indivdually. Thus, what we did was basically counting nodes for different versions of FUSc# searching the same positions. This has a clear advantage: the results can be obtained without the need to evaluate if the "chess playing strength" of FUSc# got better or worse with the introduction of a new heuristic, which is much more difficult than just counting nodes.

Another question was whether to use the FUSc# search algorithm as it is used in normal games, or whether to modify/change it for the experiment. The second option would allow to do some modifications with the aim of getting clearer results by eliminating some parts of the complex set of functional units inside a chess programm that affect the performance of the engine. One option would be, for example, to reduce the evaluation function to evaluating just the material on the board, as this would save much time during the experiments, and one could argue that we are not trying to tune or modify the evaluation function in these experiments anyway, so there is no harm in disabling it. But the problem is that because of the complexity of the process, the side-effects of parts like e.g. the move-ordering-heuristics are not always clear: what if a heuristic that is best for the "material-only"-case performs significantly worse compared to the others when the normal (slower) evaluation function is used, e.g. because the information gained by a more fine-grained evaluation can be used better by another heuristic? That's why we decided in favour of the first option, i.e. to run the tests with the unmodified search-algorithm that is used for normal play. This assures that the heuristics presented here work in practice, i.e. the improvements we achieved are not only valid in the "testing world", but did improve the search (and thus the playing strength) of the FUSc# chess program in real life.

All the experiments were done workstation featuring an AMD Athlon 64 processor[9], where an evaluation copy of "Microsoft Windows XP Professional x64 Edition" was installed in order to test if FUSc# would profit from a native 64bit enviroment[10]. And indeed, the "nodes per second" (nps) were 2-3 higher when FUSc# was running on the 64bit platform than when running the same program on the same computer, but on a 32bit version of "Windows XP Professional". The reason for this lies in the bitboard-representation that is used for the board-representation in FUSc#, which heavily relies on 64bit-operations (see section 1.2.3.2).

4.3.2 Experiment 1

The aim of our first experiment was to evaluate the performance of the different move-ordering heuristics compared to a version of FUSc# that uses no move-ordering at all. A difficulty that

[9] Here is the exact hardware configuration used: AMD Athlon 64 3000+, 1024MB DDR-Ram, 2 x 200GB Maxtor Hard Discs (RAID 0), GeForce 6600 Graphics Adaptor

[10] In order to achieve a 64bit native runtime enviroment for FUSc#, also a 64bit version of the .NET-Framework (we used .NET 2.0 Beta 2) has to be installed. The source was compiled with "Visual Studio 2005 Beta 2" with "x64" as target platform

showed up in setting up the experiments is the following: there is an extreme difference in performance between the different versions (especially between the "no-heuristics-at-all and the "best"-version). And according to the theoretical results in 2.4, this difference gets even much larger with higher search dephts. Thus, a comparison with the "no-sort"-version was only possible in relatively small seach depths. For a comparison of the different heuristics among each other, however, we were able to reach much higher depths in our experiments (see 4.3.3).

4.3.2.1 Questions to be Answered As we described in section 3.2, we have implemented several move-ordering-heuristics in FUSc#, some of which were already described in the literature, but for others, to our knowledge, this has not been the case. Consequently, the main question was of course in how far those heuristics are useful to reduce the tree size of an alpha-beta-search.

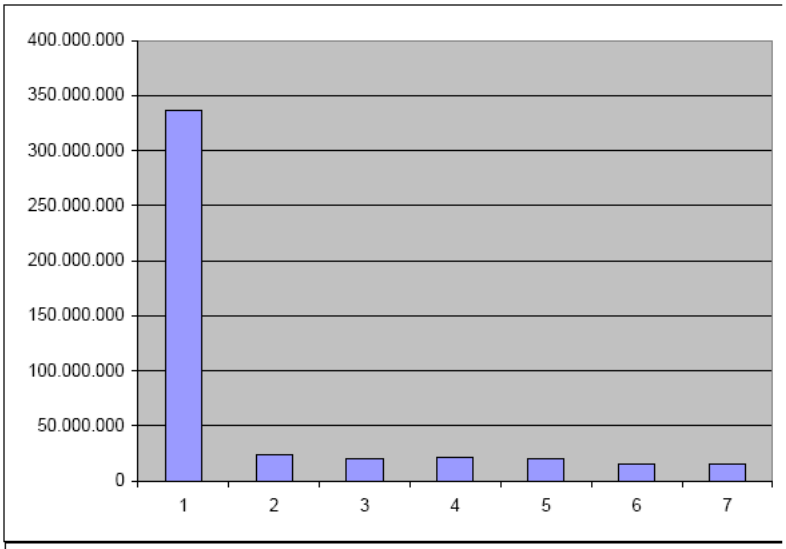
4.3.2.2 Experimental Setup A set of test positions was chosen: the LCT-II-Test[11]. It contains 35 positions, ranging from the early middlegame to the endgame taken from real games. In "Experiment 1", we compared the following versions of FUSc#:

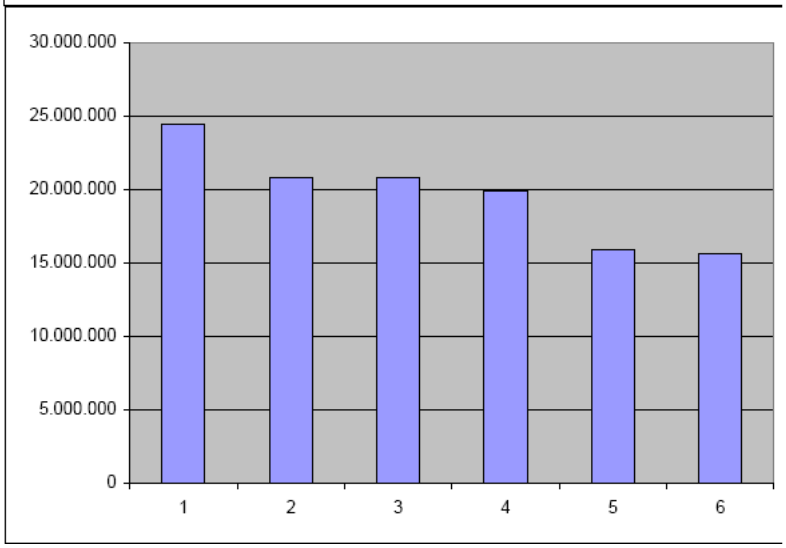
1. "no-sort", with all move-ordering heuristics turned off
2. "static", with static move-ordering, but all dynamic move-ordering heuristics turned off
3. "static+TT", with static move-ordering plus the move from the Transposition Table (if available)
4. "refuation", which equals version "3.", plus the refutation heuristic (it should actually be called "static+TT+refutation")
5. "history", with the history-heuristic turned on (i.e. "static+TT+history")
6. "killer", with the killer-heuristic[12] turned on (i.e. "static+TT+killer")
7. "all", with all the heuristics combined (i.e. "static+TT+refutation+history+killer")

4.3.2.3 Results Here is the diagram of the results:

[11] The LCT-II-Test is a widely used test suite inside the computer chess community for testing the strength of chess programs

[12] the killer-heuristic used in this experiment is actually the "killer-advanced" version. However, in this first experiment we did not aim at comparing the two versions of the killer-heristic we implemented in FUSc#, this was done later in "Experiment 2"





In the first diagramm, the versions "1."-"7." are shown, while in the second diagramm, the versions "2."-"7." (in the diagramm beeing labeled "1"-"6") are shown in a different scale for better comparison.

4.3.2.4 Interpretation of the results The main result is simple and clear: as expected, the move-ordering has a big influence on the efficiency of alpha-beta-searches. The newly implemented move-ordering heuristics in FUSc# are alltogether very successfull in reducing the tree size for typical alpha-beta searches, thus our primary aim of making FUSc#'s search algorithm more effiecient was successful. Already in depth 4, the version with all move-ordering heuristics turned off searched more than 20 times more nodes than the best version with all the heuristics combined. Also, all of the heuristics described above seems to contribute to the success of the combined "best case". except the "refutation"-heurstic, which searches slighly more nodes than when turned off. But one can make another observation: the combined "best case" is only slightly better than the best single-heuristic (which is the killer-heuristic in this case). There seem to be "side-effects" of the heuristics towards each other! Thus, the first experiment lead to the following questions:

- is the "refutation"-heuristic a contribution to achieving a good move-ordering? Up till now, the results were quite disappointing, as we expected an improvement of the nodes searched, but observed the number to even go up instead (although only by a few nodes). The question was whether this was also true for higher search depths, or if the picture would change when

going deeper.

- how do the "side-effects" of the heuristics influence the results of the "all-heuristics-cominded"-version for higher search depths? Will these "side-effects" increase, or will the combination of all heuristics turn out to be more efficient than each of the the single heuristics alone?

In order to answer these questions, there was a need to reach higher search depths than in the first experiment. Two more experiments were conducted, which are presented in the following two sections.

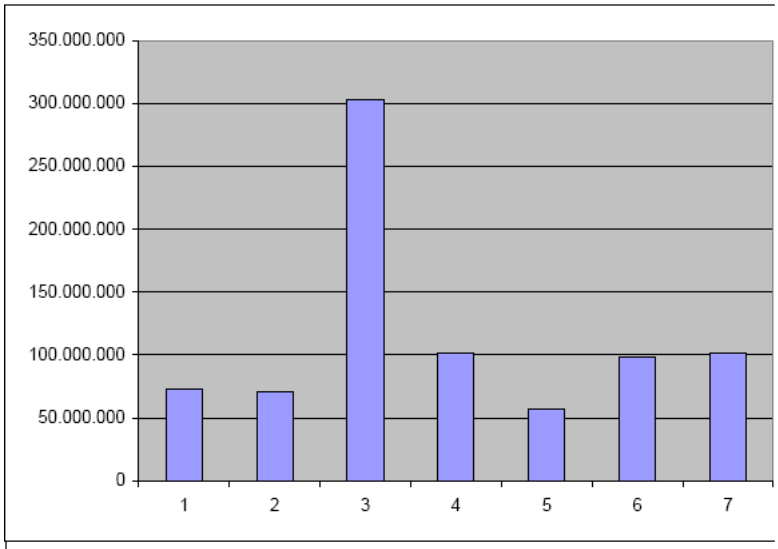
4.3.3 Experiment 2

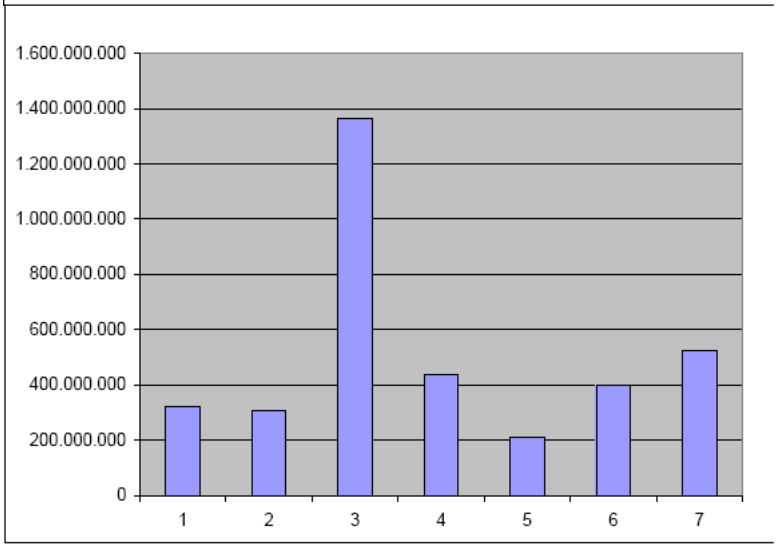
4.3.3.1 Questions to be Answered Additional to the questions that arose from "Experiment 1" (see above), we were interested in the question how the "advanced killer heuristic" that we developed in the course of this work would perform. However, the first experiment had told us that "side-effects" of the heuristics play an important role in practice, and consequently we were not so much interested in the performance of the heuristic when applied isolated, but rather in the performance of the heuristic when combined with the others. Therfor, we send 2 types of the "all-heuristics-combined"-version to the experiment: one with the simple killer heuristic turned on, and one with the advanced killer heuristic turned instead.

4.3.3.2 Experimental Setup It turned out that for conducting deeper searches in a reasonable time-frame, a smaller set of positions was needed. That's why for the "Experiment 2", a subset of 26 positions from the 35 "LCT-II"-positions used in "Experiment 1" was chosen in order to "go deep". The 9 positions with the hightes number of nodes were neglected, and the remaining positions were searched up to a depth of 9 plies. The complete test lasted more than 14 hours, with alone 10 hours for the last search depth 9. In "Experiment 2", we compared the following versions of FUSc#:

1. "all (killer-simple)", with all the heuristics combined (i.e. "static+TT+refutation+history+killer-simple")
2. "all (killer-advanced)", with all the heuristics combined (i.e. "static+TT+refutation+history+killer-advanced")
3. "static", with static move-ordering, but all dynamic move-ordering heuristics turned off
4. "static+TT", with static move-ordering plus the move from the Transposition Table
5. "history", with the history-heuristic turned on (i.e. "static+TT+history")
6. "killer", with the killer-advanced-heuristic turned on (i.e. "static+TT+killer-advanced")
7. "refuation", with the refutation-heuristic turned on (i.e. "static+TT+refutation")

4.3.3.3 Results Here is the diagram of the results:





In the first diagram, the results of the search till depth 8 is shown, whereas the second diagram shows the results of the search till depth 9.

4.3.3.4 Interpretation of the results The result of "Experiment 2" was ambivalent: On the one hand, beeing able to search so many postions untill a depth of 9 plies in a resonable time-frame was a big success for us, and this depth was only reached thanks to the new move-ordering heuristics. Also, the version applying the "advanced killer heuristic" performed clearly better than the one applying the "simple" one, which met our expectations. On the other hand, for the questions that arose from "Experiment 1", the answers were rather disappointing:

- the "refuation heuristic" was still performing very badly, especially in depth 9, where it searched considerably more nodes than the plain "static+TT"-version
- the "all-heuristics-combined"-versions (numbers "1" and "2" in the diagrams) were still outperformed by the isolated "advanced-killer-heuristic", and the difference seemed to grow larger with increasing search depths

One question that arose was if the fact that we reduced the set of positions searched in "Experiment 2" could have had a major influence on the relative performance of the heuristics. Maybe we removed just the positions that woud e.g. benefit from the refutation-heuristic in higher search depths! Thus, we scheduled a third experiment to find out.

4.3.4 Experiment 3

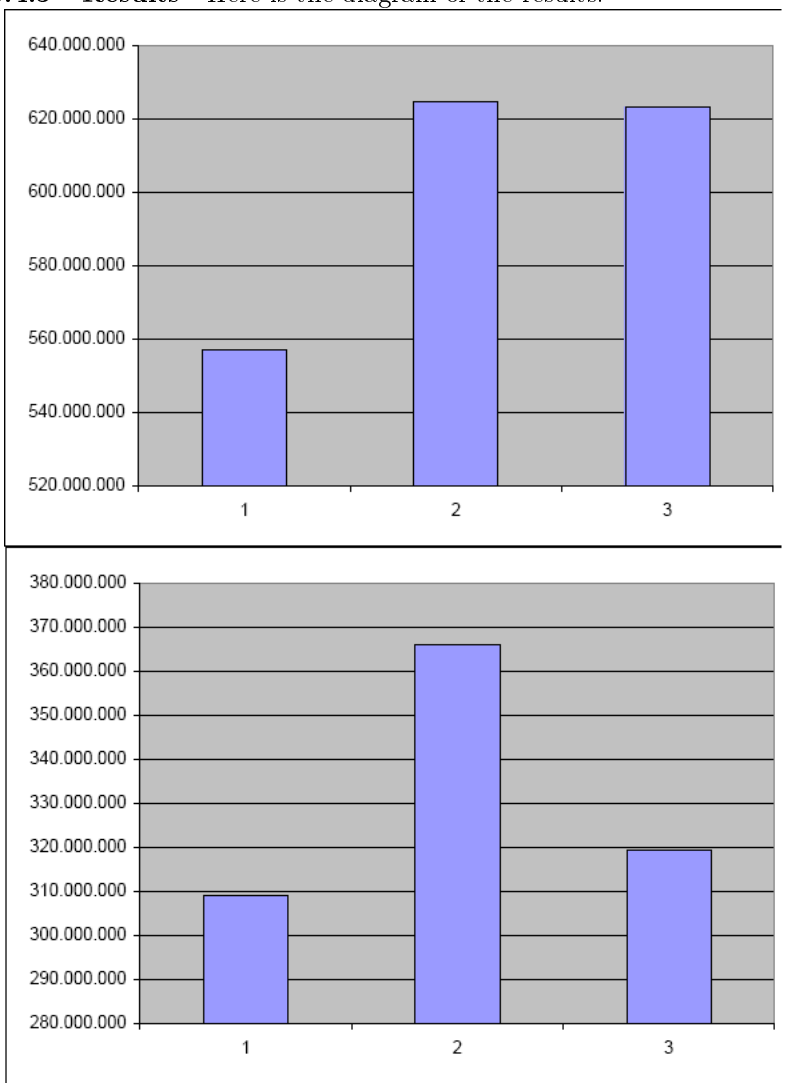
4.3.4.1 Experimental Setup In the "Experiment 3", we were going back to the full set of the 35 "LCT-II"-positions, but had only 3 versions of FUSc# running:

1. "killer-adv", with the killer-heuristic turned on (i.e. "static+TT+killer-advanced")
2. "all (killer-advanced)", with all the heuristics combined (i.e. "static+TT+refutation+history+killer-advanced")
3. "refuation", with the refutation-heuristic turned on (i.e. "static+TT+refutation")

A search depth of 7 plies was reached in ca. 8 hours of computing time.

4.3.4.2 Questions to be answered In "Experiment 3", we just wanted to find answers to the questions that arose after the finishing of "Experiment 2" (see above).

4.3.4.3 Results Here is the diagram of the results:



In the first diagram, the results of the search till depth 7 is shown, whereas the second diagram shows the results of the search till depth 6.

4.3.4.4 Interpretation of the Results For the two still open questions from "Experiment 2" (actually, they already arose in "Experiment 1"), we obtained the following results:

- the "refuation-heuristic" lead to a considerable reduction of the nodes searched at depth 6 and a slight improvement at depth 7
- the "killer-advanced"-heuristic seems to be the best of the currently implemented move-ordering heuristics, but only when implemented isolated from the others, as there seem to be many side-effects, occuring especially in higher search depths

The second point was a quite unexpected result, and needs further clarification (see next paragraph).

4.4 Interpretation of the Results of the three Search Experiments

The main result of our experiments is that we were successful in improving the move-ordering in FUSc#: "Experiment 1" showed that already in depth 4, the version with all move-ordering heuristics turned off searched more than 20 times more nodes than the best version with all the heuristics combined. Also, all of the heuristics we implemented in FUSc#, when applied alone, reduce the number of nodes searched - although there are special cases where certain heuristics fail, like e.g. the "refutation-heuristic" for smaller search depth, as we observed in "Experiment 1". But alone beeing able to search so many postions untill a depth of 9 plies in "Experiment 2" in a resonable time-frame was a big success for us, and this depth was only reached thanks to the new move-ordering heuristics. Also, the version applying the "advanced killer heuristic" performed clearly better than the one applying the "simple" one. However, the main challenge seems to be to control the "side-effects" that occur when the individual heuristics are combined: Our experiments 2 and 3 showed that it is not always the case that the "all-heuristics-combined"-version of the algorithm performs best, instead in our experiments the "killer-advanced"-heuristic was globally the best when applied alone, without the others! This is a quite astonishing result, and there has to be done further reasearch and testing in order to verify this hypothesis.

5 Conclusion/Future research

The aim of this work was to give an overview of the theory of efficient alpha-beta-searches in computer chess and to explain the practical strategies implemented in the FUSc# chess program. The main focus was put on two topics: the first one was efficient move generation with "rotated bitboards", and the second one was how to carry out efficient alpha-beta-searches by optimizing the move-ordering with static and dynamic heuristics. Pracical experiments with FUSc# were used to verify the theoretical results in both questions.

One of the challenges for optimizing the application of dynamic move ordering heuristics is that the heuristics tend to perform extremly different depending upon the exact conditions of the experiment: the result will depend on the selection of positions, on the chosen search depth, and probably on other factors that were not changed during our experiments but could as well be included in the set of parameters (like which variant of alpha-beta is used to perform the search, because it might be that one heuristic performs better with plain alpha-beta while beeing worse with MTD(f), for example). Also, there are even more parameters to tune concerning the move-ordering that just switching move-ordering-heuristcs on or off: one could fine-tune the exact order in which the heuristics are applied (e.g. "should the killer move come before the refutation move or afterwards?") or even include multiple variants of the heuristics (e.g. allowing captures in the killer-move slots or not). Alltogether, it seems to us that manually testing all of the different variants is not desirable, as this will need extremely much time, and is basically a "try-and-error" procedure. On the contrary, the search-algorithm of FUSc# seems to be at a point where a systematic testing of all the mentioned parameters would be needed in order to find out the best combination. This could be done either by massively running tests of different versions of FUSc#,

and tune the parameters manually according to the results. However, a more promising alternative seems to be trying to implement a framework where not only the testing but also the adjustment of the parameters is automatized. This could maybe even be combined with some approaches on machine learning, leading to a bunch of new research possibilities for the future.

Another aspect is that for doing large-scale experiments on search algorithms, much computing power is needed. An interesting perspective arises with the arrival of 64bit processors (like AMD64-chips, that are relatively cheaply available), because our experiments show that the .NET-Framework and FUSc# heavily profit from the 64bit enviroment (the "nps" were more than doubled without any source code changes!). Even more promising could be port of FUSc# to the new "64bit-Dualcore"-CPUs, although this step would need a rewrite of major parts of the program (e.g. conducting a parallel the alpha-beta-search). Nevertheless, this could be an interesting perspective, as there are many multiprocessor systems based on AMD64-processors (e.g. with AMD Opteron) available, that could form a base for future running enviroments of a "DeepFUSc#"-program

6 Appendix

6.1 Rotated bitboards in detail

This section should illustrate how the chess-board looks like in the "rotated bitboards". After the "normal" bitboard, the flipped bitboard (rotated to the left by 90°) as well as the two bitboards needed for move generation in direction of the two diagonals (the "a1h8" and the "a8h1"-bitboard) will be shown. For more information, please have a look at [10].

6.1.1 The normal bitboard

This is the "normal" bitboard, as used in many places in the program:

```
a8 b8 c8 d8 e8 f8 g8 h8
a7 b7 c7 d7 e7 f7 g7 h7
a6 b6 c6 d6 e6 f6 g6 h6
a5 b5 c5 d5 e5 f5 g5 h5
a4 b4 c4 d4 e4 f4 g4 h4
a3 b3 c3 d3 e3 f3 g3 h3
a2 b2 c2 d2 e2 f2 g2 h2
 a1 b1 c1 d1 e1 f1 g1 h1
```

6.1.2 The flipped bitboard ("l90")

The "flipped" bitboard is stored in "board.occ_l90" and used to generate moves along files for rooks and queens:

```
a8 a7 a6 a5 a4 a3 a2 a1
b8 b7 b6 b5 b4 b3 b2 b1
c8 c7 c6 c5 c4 c3 c2 c1
d8 d7 d6 d5 d4 d3 d2 d1
e8 e7 e6 e5 e4 e3 e2 e1
f8 f7 f6 f5 f4 f3 f2 f1
g8 g7 g6 g5 g4 g3 g2 g1
h8 h7 h6 h5 h4 h3 h2 h1
```

6.1.3 The a1h8 bitboard

The a1h8 bitboard is stored in "board.occ_a1h8" and used to generate diagonal moves in direction of the "a1h8-diagonal" for bishops and queens. Note that in this "compressed" representation, it must be assured that a piece can not "jump over the edge" of the chessboard and re-enter it on the other side because this would allow illegal moves. This must be taken care of during the initialisation of the bitboards, where all the legal diagonal moves in the direction of the a1h8-diagonal are encoded into the "a1h8_moves"-array. The edge of the board is marked with the symbol "|" in the following figure:

```
a8 | b1 c2 d3 e4 f5 g6 h7
a7 b8 | c1 d2 e3 f4 g5 h6
a6 b7 c8 | d1 e2 f3 g4 h5
a5 b6 c7 d8 | e1 f2 g3 h4
a4 b5 c6 d7 e8 | f1 g2 h3
a3 b4 c5 d6 e7 f8 | g1 h2
a2 b3 c4 d5 e6 f7 g8 | h1
 a1 b2 c3 d4 e5 f6 g7  h8
```

6.1.4 The a8h1 bitboard

The a1h8 bitboard is stored in "board.occ_a1h8" and used to generate diagonal moves in direction of the "a1h8-diagonal" for bishops and queens. The edge of the board is again marked with the symbol "|" in the following figure (see above):

```
a8 b7 c6 d5 e4 f3 g2   h1
a7 b6 c5 d4 e3 f2 g1 | h8
a6 b5 c4 d3 e2 f1 | g8 h7
a5 b4 c3 d2 e1 | f8 g7 h6
a4 b3 c2 d1 | e8 f7 g6 h5
a3 b2 c1 | d8 e7 f6 g5 h4
a2 b1 | c8 d7 e6 f5 g4 h3
 a1 | b8 c7 d6 e5 f4 g3 h2
```

6.2 "perft"-output for FUSc# and Crafty

As a proove for the correctness of the FUSc# move generator, the position described in section 4.1.2 are loaded into FUSc# and Crafty. Then the "perft"-command is executed. All the computed numbers turn out to be correct for both FUSc# and Crafty! For reference, you find the original output in the following two sections.

6.2.1 FUSc#

In FUSc#, the "perft"-command is implemented as "debug counodes", in order to be consistent with the other debugging commands (that can be obtained by entering "debug help" at the command prompt). Here is the output up to depth 5:

```
debug countnodes 1
Minmax-Suche bis Tiefe 1
Besuchte Knoten:  20
debug countnodes 2
```

```
Minmax-Suche bis Tiefe 2
Besuchte Knoten:  400
debug countnodes 3
Minmax-Suche bis Tiefe 3
Besuchte Knoten:  8902
debug countnodes 4
Minmax-Suche bis Tiefe 4
Besuchte Knoten:  197281
debug countnodes 5
Minmax-Suche bis Tiefe 5
Besuchte Knoten:  4865609
```

Now the middlegame-position:

```
position fen r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq -
debug countnodes 1
Minmax-Suche bis Tiefe 1
Besuchte Knoten:  48
debug countnodes 2
Minmax-Suche bis Tiefe 2
Besuchte Knoten:  2039
debug countnodes 3
Minmax-Suche bis Tiefe 3
Besuchte Knoten:  97862
debug countnodes 4
Minmax-Suche bis Tiefe 4
Besuchte Knoten:  4085603
```

And now the endgame-position:

```
position fen 8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - -
debug countnodes 1
Minmax-Suche bis Tiefe 1
Besuchte Knoten:  14
debug countnodes 2
Minmax-Suche bis Tiefe 2
Besuchte Knoten:  191
debug countnodes 3
Minmax-Suche bis Tiefe 3
Besuchte Knoten:  2812
debug countnodes 4
Minmax-Suche bis Tiefe 4
Besuchte Knoten:  43238
debug countnodes 5
Minmax-Suche bis Tiefe 5
Besuchte Knoten:  674624
debug countnodes 6
Minmax-Suche bis Tiefe 6
Besuchte Knoten:  11030083
```

6.2.2 Crafty

Here is the output of Crafty in the start-position:

```
White(1):  perft 1
total moves=20 time=0.00
White(1):  perft 2
total moves=400 time=0.00
White(1):  perft 3
total moves=8902 time=0.00
White(1):  perft 4
total moves=197281 time=0.26
White(1):  perft 5
total moves=4865609 time=6.66
White(1):  perft 6
total moves=119060324 time=283.79
```

Now the middlegame-position:

White(1): setboard r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq -
White(1): perft 1
total moves=48 time=0.00
White(1): perft 2
total moves=2039 time=0.00
White(1): perft 3
total moves=97862 time=0.10
White(1): perft 4
total moves=4085603 time=4.52

And now the endgame-position:

White(1): setboard 8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - -
White(1): perft 1
total moves=14 time=0.00
White(1): perft 2
total moves=191 time=0.00
White(1): perft 3
total moves=2812 time=0.00
White(1): perft 4
total moves=43238 time=0.04
White(1): perft 5
total moves=674624 time=0.86
White(1): perft 6
total moves=11030083 time=22.34

6.3 Detailed Result of the Experiments

In order to verify the diagrams shown in part 4.3, you will find some more detailed figures from the experiments described there. However, they represent only parts of the data that was retrieved during the experiments. The complete dataset will be made available for download, please follow the link from [6].

6.3.1 Experiment 1

position		no-sort	static	static+TT	refutation	history	killer
	1	666013	70868	38252	38083	37096	23858
	2	25134561	2743116	2246094	2228838	1500424	310604
	3	2220423	15099	14815	13820	13753	14798
	4	70676	21213	9869	9870	8999	8311
	5	226187	22715	4894	4898	5193	3226
	6	2991817	82051	54441	56897	55060	54137
	7	189035560	14595967	13527014	13508909	13740715	9993557
	8	24447290	4218618	862443	862868	746177	620125
	9	324827	36153	23225	22864	21877	16638
	10	169520	4522	2553	2553	2552	2558
	11	162010	170909	171274	165223	164612	172276
	12	625332	53319	48092	48273	42647	38003
	13	2788119	198120	154354	150014	150859	103770
	14	1378995	415566	228733	210729	219452	196556
	15	93352	13197	10051	9681	10248	11247
	16	30706892	135327	1530468	1522287	1735617	3367755
	17	12361841	168897	160628	155742	162884	115401
	18	24125984	46295	23543	23003	53341	23449
	19	1772323	295522	879701	1035831	356727	54053
	20	1935756	9987	9862	9834	9597	9780
	21	5900961	34774	34715	34435	84998	38418
	22	744122	53962	68379	66418	83349	63746
	23	19881	4189	3082	3082	2894	2665
	24	1867010	303460	166767	165449	241062	217918
	25	1935824	135967	190415	191344	192688	258632
	26	4575261	576440	293370	289710	260319	161596
	27	242	256	106	106	106	109
	28	2452	583	410	410	410	418
	29	2259	2207	1156	1127	1103	1732
	30	1817	1490	1269	1271	1239	1256
	31	16183	3438	3703	3831	3095	2783
	32	1298	788	771	778	743	748
	33	28424	6987	2430	2442	2415	2285
	34	899	282	282	282	282	282
	35	678	477	473	473	473	440
		336.334.789	24.442.761	20.767.634	20.841.375	19.913.006	15.893.130

6.3.2 Experiment 2

Here are the detailed results for depth 8:

position	all (k-simple)	(k-advanced)	static	static+TT	killer	history
1	1352155	1187726	3919219	1149535	1126695	1688392
2	1268675	1271228	13304784	1944809	1167277	1726543
3	1705323	2363923	5788682	4184839	2476661	4335617
4	2332305	2092778	20310221	8097080	1578073	4450866
5	9114574	8217224	23373500	3688378	4984372	3689574
6	2598672	2812409	19677097	9551647	2532081	7493000
7	3924715	3414087	9993140	4928461	3576105	5425097
8	2224532	2232869	3981948	2648470	2316373	1822473
9	1110318	1069188	21532804	3977964	1036996	2704092
10	6120309	6242437	21679025	7665557	4738700	7132774
11	1138502	1444377	4569959	2197540	817743	1872931
12	10420187	10043584	81544422	12086649	9775863	13605634
13	9730715	7996534	9168060	5862693	5334050	6542099
14	931580	917797	1426911	1194029	1066257	1064620
15	4266629	5200311	3658936	3648289	3085042	3818161
16	11210883	11283876	52726223	23651772	8960622	26314045
17	3157324	2377442	5233609	4499346	2349739	3980679
18	10868	9129	20920	8788	8804	6232
19	74221	65022	209607	203329	50499	165435
20	24642	20831	47096	27023	40239	30322
21	54247	59982	57066	25904	25626	22796
22	100040	98568	206662	131344	92294	138093
23	54756	54943	47580	42115	54018	37642
24	109123	140275	531828	180432	89205	190489
25	9240	5312	18174	5033	10356	5591
26	22570	22446	23601	27879	22581	33479
	73.067.105	70.644.298	303.051.074	101.628.905	57.316.271	98.296.676

Here are the detailed results for depth 9:

position	all (k-simple)	(k-advanced)	static	static+TT	killer
1	6393109	16307715	34932612	15974962	7863139
2	2297366	2227003	91254864	7503435	3173288
3	2780374	5752634	15792708	7147022	5379051
4	10905561	6885552	110736946	11386380	7967052
5	22449227	16523714	90002885	17146455	15691615
6	11776603	7244520	86122678	59618462	13488622
7	6837295	6645039	19625915	11569564	7172060
8	8438458	8204527	16797784	14341233	10699322
9	3437244	3225329	142976334	8742720	3153055
10	11717108	12271925	54428164	11914988	8561745
11	2481878	3127318	15541981	6792970	2215364
12	141741530	141313945	326304843	92107102	52428299
13	37835779	25396463	141565842	89869825	27241347
14	1308473	1314421	2576938	1635077	1380737
15	11662962	11800604	8068893	8130549	10553458
16	25451136	34591538	173071967	56837186	30186279
17	13997454	3937713	33289932	13172624	3826006
18	29666	13624	78596	38619	13994
19	213004	178674	797215	474527	113449
20	39216	45321	106620	48399	48800
21	77504	75996	162811	137774	96901
22	303364	238876	765785	272654	269023
23	135728	128225	58729	110417	118745
24	181234	210842	1343842	290453	148795
25	26793	7105	26058	7290	18812
26	102297	56143	104741	56186	82610
	322.620.363	307.724.766	1.366.535.683	435.326.873	211.891.568

6.3.3 Experiment 3

Here are the detailed results for depth 6:

position	killer-adv	all (k-adv)	refutation
1	350792	499312	939488
2	5148534	6907149	19473545
3	228852	214638	205086
4	87671	105309	91060
5	386668	472407	655485
6	412702	484208	542374
7	116403839	145870885	147736792
8	151443558	115337737	65694856
9	456723	433266	471663
10	146656	159940	161704
11	917621	918259	1306594
12	113880	131517	138074
13	565087	600884	1175343
14	6149903	20825955	36168534
15	74314	154097	136095
16	7651484	49409283	6529286
17	1431366	1848433	1530551
18	333099	396220	400750
19	4451399	6927632	13654973
20	96818	100605	95347
21	327096	557553	673921
22	454144	474921	357620
23	407192	445236	704754
24	4865943	4026651	13839349
25	2095513	4564885	3367524
26	3988777	4027457	3217257
27	1173	1164	1537
28	1753	1826	2051
29	8013	7822	6001
30	8061	7404	8512
31	11983	12978	14533
32	12111	13012	7882
33	47386	54280	86824
34	1551	1607	1509
35	2647	2282	2587
	309.084.309	365.996.814	319.399.461

Here are the detailed results for depth 7:

position	killer-adv	all (k-adv)	refutation
1	832874	882889	1652636
2	10909445	16731625	78963905
3	420538	423357	464431
4	847637	940743	1263415
5	837486	849514	1121403
6	1416320	1436583	1788875
7	225013842	280729870	275685477
8	208563323	184541686	106904453
9	1301301	1033557	1677763
10	235661	236229	247634
11	1410721	1347419	1655443
12	493651	562929	2915970
13	948012	1068876	1669033
14	44892576	27994485	43777143
15	228613	483775	409739
16	12074612	54275137	11163400
17	2570683	3176050	2710475
18	1466483	1292276	973621
19	10200310	12485849	21160474
20	138534	139618	142165
21	449795	806107	889562
22	2001656	2056112	3118403
23	793649	845289	1530221
24	6501815	5424942	14688936
25	10942717	12659385	27793976
26	11426837	11769389	18390208
27	4279	3962	3121
28	24716	32791	72834
29	25576	9195	13171
30	13138	14407	16189
31	21739	24026	40345
32	24563	28600	13433
33	69919	78862	151092
34	4849	3992	3483
35	13983	14300	16444
	557.121.853	624.403.826	623.088.873

References

[1] Block, Marco, *Verwendung von Temporale-Differenz-Methoden im Schachmotor FUSc#,* Diplomarbeit, Berlin, 2004

[2] Buchner, Johannes: *Rotated Bitboards in FUSc#*, Seminararbeit, Berlin, 2005

[3] Crafty-Homepage: *http://www.cis.uab.edu/info/faculty/hyatt/hyatt.html*

[4] Dill, Sebastian, *Transpositionstabellen*, Seminararbeit, Berlin, 2005

[5] Download of DarkFUSc# and FUSc#: *http://page.mi.fu-berlin.de/~fusch/download/*

[6] FUSc#-Homepage: *http://www.fuschmotor.de.vu*

[7] FUSc#-Server: *http://www.inf.fu-berlin.de/~fusch/*

[8] Frey, P.W. (editor), *Chess skill in man and machine*, Springer, 2nd edition 1983

[9] Heinz, Ernst A.*, Scalable search in computer chess,* Vieweg, 2000

[10] Heinz, Ernst A.: *How DarkThought plays chess*, http://supertech.lcs.mit.edu/~heinz/dt/node2.html

[11] Homepage of Peter McKenzie on Computer chess: *http://homepages.caverock.net.nz/~peter/perft.htm*

[12] Homepage Microsoft .NET: *http://www.microsoft.com/net/default.mspx*

[13] Homepage of gcc: *http://gcc.gnu.org/*

[14] Kaindl*, H., Tree Seaching Algorithms* (chapter 8 of [17]).

[15] Luger, George F.: *Artificial Intelligence, 2nd edition, 1993*

[16] Plaat, Aske: *RESEARCH, RE:SEARCH & RE-SEARCH*, Tinbergen Institute, 1996

[17] T.Antony Marsland, Jonathan Schaeffer (Editors), *Computers, Chess and Cognition, Springer, 1990*

[18] Reinefeld, Alexander, *Spielbaum Suchverfahren*, Informatik-Fachberichte 200, Springer Verlag, 1989.

[19] Shannon, Claude, *Programming a Computer for Playing* Chess, *Philosophical* Magazine, Ser.7, Vol. 41, No. 314 - March 1950.

[20] Slagle, James H. and Dixon, John K., *Experiments with some programs that search game trees*, Journal of the ACM, 16(2):p.189-207, April 1969

[21] Reinefeld, Alexander, *Spielbaum Suchverfahren*, Informatik-Fachberichte 200, Springer Verlag, 1989.

[22] Stockman, George C., *A minimax algorithm better than Alpha-Beta?*, Artificial Intelligence, 12(2):p179-196, 1979.