

#### BSc Thesis

# Talvmenni

A chess-playing program capable of utilizing JavaSpaces™ for parallel processing

Eyðun Lamhauge og Eyðun Nielsen

NVDRit 2004:18

Faculty of Science and Technology University of the Faroe Islands

# Heiti / Title Talvmenni

Eitt talvforrit ið megnar at nýta JavaSpaces<sup>™</sup> til samútrokningar

#### Talvmenni

A chess-playing program capable of utilizing JavaSpaces<sup>™</sup> for parallel processing

Høvundar / Authors Eyðun Lamhauge og Eyðun Nielsen

Vegleiðari / Supervisor Kurt Madsen, Fróðskaparsetur Føroya

Ábyrgdarvegleiðari / Responsible Martin Zachariassen, Fróðskaparsetur Føroya

Supervisor

Ritslag / Report Type BSc verkætlan, teldufrøði

BSc Thesis, computer science

Latið inn / Submitted 20<sup>th</sup> of December 2004

**NVD**Rit

2004:18

© Náttúruvísindadeildin og høvundarnir 2004

Útgevari / Publisher Náttúruvísindadeildin, Fróðskaparsetur Føroya Bústaður / Address Nóatún 3, FO 100 Tórshavn, Føroyar (Faroe Islands)

Postrúm / P.O. box 2109, FO 165 Argir, Føroyar (Faroe Islands)

# **Samandráttur**

Í hesari ritgerð verður fyrst greitt frá bygnaðinum í einum talvforriti. Ymiskir dátubygnaðir, leitialgoritmur og háttir at umboða talvvitan innan í talvforritum verða kannað. Í sambandi við hesa verkæltan bleiv eitt talvforrit framleitt, ið fekk heitið *Talvmenni*.

Síðani verður kannað, um JavaSpaces<sup>™</sup> tøknin er egnað sum samútrokningarumhvørvið hjá *Talvmenni*, og ein samútrokningaralgoritma varð ment, ið varð nevnd *TOSCA*. Mátingar vóru gjørdar, har avrik frá samleiting við JavaSpaces verða samanborin við avrik frá vanligari leiting. Úrslitini av kanningunum vóru lutvíst tileggjandi, og benda á at JavaSpaces megnar væl at gagnnýta økta teldutilfeingi, men hevur høgt samskiftisroki.

# **Abstract**

Firstly, this thesis identifies and describes the elements of a chess-playing program. Different data structures are examined, as well as modern search algorithms, and some of the most used approaches for representing chess knowledge within the program are described. *Talvmenni*, a program capable of playing the game of chess was developed during this research.

Secondly, the suitability of JavaSpaces as an environment for parallelizing the processing of *Talvmenni* is examined, and *TOSCA*, a parallel algorithm is introduced. Empirical results comparing the performance of parallel search with JavaSpaces to performance of sequential search are reported. The results of the experiments are partly encouraging, and indicate that JavaSpaces do scale very well with additional computer resources, but also contain a high ratio of communication overhead.

# **Acknowledgments**

First of all, thanks to our supervisor Kurt Madsen for his revisions, sharp comments and enthusiasm throughout this endeavor.

Thanks to Sigrun E. Petersen for proof reading of the thesis. Our gratitude to Lis Berg for her tolerance during the last eight months, and for providing us with the best possible work-environment.

Finally, we would both like to thank our families for their never-ending patience during the course of this, for us, addictive project. This thesis would not have been possible without their support.

# **Table of Contents**

Samandráttur	iii
Abstract	iv
Acknowledgments	V
Table of Contents	Vi
List of Figures	ix
Chapter 0: Introduction	
Problem Statement	1
Outline of this Thesis	2
Chapter 1: Background and Related Work	4
1.1 The Game Of Chess	4
1.1.1 The Initial Position	5
1.1.2 Rules for Piece Movement	6
1.1.3 Special Rules	7
1.2 Computer Chess Development	9
1.2.1 Computer Chess and Artificial Intelligence	9
1.2.2 A Brief History Overview	11
1.2.3 Sequential Computer Chess	14
1.3 Parallel Search	16
1.3.1 The Need for Speed	16
1.3.2 Sources of Parallelism	17
1.3.3 Parallel Architectures	19 21
1.3.4 The Tuple Space Model	21
1.4 Talvmenni ver. 0.1	22
1.4.1 Development Tools	24
Chapter 2: Data structures	25
2.1 Board Representation	25
2.1.1 Offset Representation	26
2.2 Bitboards	29
2.2.1 Pregenerated Bitboards	30
2.2.2 Rotated Bitboards	34
2.2.3 Talvmenni Bitboard Iterator	35
2.2.4 Talvmenni Bitboard Masks	38
2.3 Move Generation	42
2.3.1 Full Move Generation	43
2.3.2 Perft Calculation	45
2.4 Chapter Summary	46

Chapter 3: Representing Chess Knowledge	47
3.1 Introduction	47
3.2 Position Evaluation	48
3.2.1 Material Balance	48 49
3.2.2 King Safety 3.2.3 Mobility and Development	49 50
3.2.4 Pawn Structure	50
3.3 Opening Book	53
3.4 Hash Tables	54
3.4.1 Transposition Tables	54
3.5 Endgame Databases	56
3.6 Chapter Summary	57
Chapter 4: Sequential Search	58
4.1 Game Trees	58
4.2 Minimax	60
4.2.1 Negamax	62
4.3 Alpha-Beta Pruning	63
4.4 Alpha-Beta Enhancements	65
4.4.1 Move Ordering	65
4.4.2 History Heuristic and Killer Moves	66
4.4.3 Iterative Deepening	67
4.5 Minimal Window Search	68
4.5.1 Principal Variation Search 4.5.2 MTD(f)	68 69
4.6 Horizon Effect	70
4.6.1 Quiescence Search	71
4.7 The Null-Move Heuristic	72
4.8 Chanter Summary	73

Chapter 5: Distributed Search Strategy	74
5.1 Introduction	74
5.2 Parallel Performance	75
5.2.1 Search Overhead	76
5.2.2 Communication Overhead	77
5.2.3 Syncronization Overhead	77
5.3 Parallel Algorithms	78
5.3.1 The Young Brother Wait Concept	78
5.3.2 APHID	80
5.4 Jini and JavaSpaces	82
5.4.1 The Jini Architecture	83
5.4.2 JavaSpaces	84
5.4.3 The Replicated Worker Pattern	86
5.5 TOSCA	87
5.5.1 Parallel Strategies	88
5.6 Chapter Summary	90
Chapter 6: Tests & Experiments	91
6.1 Methodology	91
6.1.1 Hardware	92
6.2 Test Results	93
6.2.1 Test 1: Perft Tests	93
6.2.2 Test 2: Play against Humans	96
6.2.3 Test 3: Play against Computers	97
6.2.4 Further Remarks	98
6.3 Parallel Speedup	99
Chapter 7: Conclusions	101
Appendices	
A: List of Abbreviations	104
B: Screenshots	105
C: Games from the Test Tournament	113
D: Bit-wise Operations	114
E: Resources CD	115
References	116

References

# **List of Figures**

1.1 The initial position 1.2 A comparison of the rating development of the human chess	5 12
champions and the best chess computers  1.3 Horizon axis shows number of positions examined in three	16
minutes in a logarithmic scale and vertical axis shows rating points 1.4 Multiprocessor systems	19
2.1 Move generation for pawns	26
2.2 The 0x88 board representation	27
2.3 A bitboard mapping for all white pawns	29
2.4 A unit-test mapping the bit string for G7	30
2.5 The construction of the constant _1 representing rank 1	
2.6 Unit-tests for file C and diagonal a2-g8	32
2.7 'Maximum' movepatterns is highlighted in red, for a	33
Knight and Queen placed on the square d4	
2.8 The complete code for the bitboard-iterator	35
2.9 deBruijn method in the class Square.Util	36
2.10 Example-position showing occupancy for white queen	
2.11 Chess-men on white queen's move-path	39
2.12 Pre-generated masks for obtaining the path from north-east to	39
south-west towards 'd4'	
2.13 Chessmen masks out with north-west to south-east path	40
towards 'd4' mask	
2.14 Finding reachable path for white queen on 'd4'	40
2.15 All reachable paths for white queen on 'd4'	41
2.16 Reachable path & Legal moves	44
3.1 Deep Blue – Kasparov. 1996	51
3.2 Knowledge vs. speed	52
3.3 Positions can be reached by different move ordering	55
oto i oblitorio cari de reactica dy amerene move oracimig	00
4.1 Example of a subset of a game-tree	59
4.2 Pseudo minimax algorithm	61
4.3 An example of a minimax-search	61
4.4 Pseudo negamax algorithm	62
4.5 An example of an alpha-beta search	64
4.6 MTD(f)	69
	77
5.1 Parallel overheads	77
5.2 An APHID search tree	80 83
5.3 Technology stack	
5.4 Operations on a JavaSpace	85
6.1 Test position no.1	93
6.2 A comparison of Perft values for test position no. 1	94
6.3 Test position no.2	94
6.4 A comparison of Perft values for test position no. 2	95
6.5 Test position no.3	95
6.6 A comparison of Perft values for test position no. 3	95
6.7 Talvmenni Rating Development	96
6.8 Tournament table for the test tournament	98
6.9 Sequential search vs. TOSCA with one worker	99
6.10 Speedups for different search depths	100



# **Problem Statements**

Game playing is one of the oldest areas in artificial intelligence (AI), and for the public advances especially in chess-playing programs has been symbolic for the progress of the whole subject of AI.

In addition, developing a chess-playing program requires the application of theoretical concepts and algorithms that can be applied in other real-world practical situations. Computer chess programming has therefore become accepted as a useful test environment for AI research.

#### **Problem Statement 1:**

• Identify and describe the elements of a program capable of playing the game of chess and develop a chess engine implemented in Java.

In chess programs, speed is considered a key determinant of the program's playing strength The faster the computer processes positions the deeper it can search into the game-tree. One approach to increasing the speed of chess programs that has received a lot of attention is *parallelism*.

Networks of workstations are becoming increasingly popular as parallel systems, because of they have the advantage of high scalability and cost-effectiveness. Tuple-based programming is a novel way of building distributed applications that aim at improved simplicity, scalability and reliability.

JavaSpaces(TM) is a modern tuple-based technology from Sun MicroSystems that is claimed to be a simple, expressive, and a powerful tool to ease the burden of creating distributed applications.

#### Problem Statement 2:

• Examine if the JavaSpace technology is a suitable environment for parallelizing the processing of the developed chess engine.

# **Outline of this Thesis**

**Chapter 1** "Background and Related Work" gives an introduction to the field of computer chess programming. First we describe those rules of the game, which are relevant for programming a computer for playing chess. We discuss artificial intelligence and computer chess programming and give a brief history overview. Next, we introduce the different subjects of computer chess programming and point out related work that has been relevant for the development of our chess engine. Finally, we examine the subject of parallel systems and potential sources for parallelism in a chess program.

**Chapter 2** "Data Structures" describes the challenges that are present for internal representations within the program. We examine the core structure in our program: bitboards and present some of our own ideas for move generation. Data Structures is a complex subject and this chapter therefore includes many diagrams to enhance the readability.

**Chapter 3** "Implementing Chess Knowledge" discusses some of the most used approaches to embed general chess knowledge into chess programs. An important element in all chess programs is the *evaluation function* that encapsulates chess knowledge so the chess program can identify good positions, which it is aiming to obtain. The various properties of the evaluation function are discussed in this chapter, as well as other important components that aim at increasing the programs level of play by other means than brute-force searching.

**Chapter 4** "Sequential Search" begins with an explanation of game-tree search. We review the three basic search algorithms: MiniMax, NegaMax and Alpha-Beta Pruning. Next, we examine different methods to enhance the Alpha-Beta search algorithm.

**Chapter 5** "Distributed Search Strategy" examines the area of parallel search. Two popular parallel search algorithms are examined and we take a closer look at the JavaSpace environment. Finally, we present our suggestion for a parallel algorithm in a JavaSpace-based environment.

**Chapter 6** "Tests and Experiments" presents the data obtained from the various tests and experiments that we have performed on our program, to verify the quality of the program as well as measuring performance issues.

**Chapter 7** "Conclusions" draws overall conclusions and we list directions for suggested future research.

Chapter 1 should be read before the other chapters. The chapters from 2 to 5 can be read independently of each other, but the subjects dealt with in these chapters are, in our opinion, ordered in a logical manner. We therefore recommend reading these chapters in the listed order, especially chapter 5 might be difficult to comprehend without the necessary prior knowledge.

This thesis reviews systematically the elements of a chess-playing program, and it presents concrete problems and solutions that we experienced during the development of our chess program.

Aspects of software engineering such as requirements documentation, process models and applied design patterns are not within the scope of this work, and are not examined in this thesis.

# Chapter

# Background and Related Work

This chapter provides an introduction to the subject of computer chess development, and a summary of some of the work that has been carried out in the fields of both sequential and parallel chess programs.

#### 1.1 The Game of Chess

Different cultures throughout history have had different rules and variants for the game of chess. The unified rules that are most common today were proposed in 1853 by Howard Staunton, and have been managed by the world chess federation *"Federation Internationale des Echecs"* since 1924 when the organization was founded.

Chess is a game for two players. It is played on an 8x8 checkered board with two sets of pieces of 8 chessmen. The player who controls the white pieces moves first, and then the players move their pieces alternately.

The goal of a chess game is to manoeuvre the pieces in order to place the opponent's king in a position where no legal move can save the king from being captured in the following move. This state is called a *checkmate* and the player with the attacking pieces is the winner of the game [FIDE 1997].

A game of chess is traditionally divided into three logical phases: the opening, the middle game, and the end game [Nimzowitch 1925]. The *opening* consists of the first five to ten moves when both sides aim to develop their pieces into action, and at the same to get the king into safety. The *middle game* is the most complex stage of the game with all the pieces participating in the action, and is the phase where most games are won or lost for either side. The third and final phase of the game is called the *end game*, and begins when most pieces have left the board. The end game is the stage in which the player with the advantage is to realize his gains resulting in check-mating the opponent.



Figure 1.1: The initial position.

#### 1.1.1 The Initial Position

The board is placed between the players and must be rotated in a position where the nearest corner square to the left of both players is a black square. At the start of the game sixteen pieces are situated on two ranks nearest each player. The pieces on the first rank are: two rooks in the corner squares, two knights in the adjoining squares, two bishops next to the knights, the queen on the remaining square corresponding to her colour, and the king on the other remaining middle square. On the second rank eight pawns are placed immediately in front of the pieces on the first rank [FIDE 1997].

A vertical column of squares running up the board is called a *file*. A horizontal row of squares on a chessboard is called a *rank*. A group of squares running at a 45 degree angle to the files and ranks is called a *diagonal*.

Chess notation gives the possibility to write down and record the moves of a chess game. The rows are assigned the numbers '1' through '8' starting from the white side. The columns are assigned letters 'a' through 'h', from left to right from the white side. A square is identified by the letter of the column and the number of the row that the square intersects.

#### 1.1.2 Rules for Piece Movement

- **The King** can move one square in any direction. If an opponent's piece is attacking the king the king is said to be *in check*, and must get out of this state immediately. The King may never move into check that is, onto a square attacked by an opponent's piece.
- **The queen** can move any number of squares in any direction horizontally, vertically, or diagonally but only if the traversed path is not blocked by any piece.
- **The rook** can move any number of squares horizontally or vertically but only if the traversed path is not blocked by any piece.
- **The bishop** can move any number of squares diagonally but only if the traversed path is not blocked by any piece.
- The movement of **the knight** is special. The knight is capable, as the only piece, to jump over other pieces on its path from an old position to a new. The distance between the old square and the new square is two squares forward in any direction, and one square sideways to either side of that direction. The knight therefore always lands on a square opposite in colour from its old square.
- **Pawns** cannot move backwards or sideways, but must move straight ahead unless they are capturing a piece of the opponent, which is done by capturing to the upper left or upper right square.
  - Pawns move only one step at the time except when located in their initial position, and not blocked by any piece. Then they may advance one or two steps according to the plan of the player.

#### 1.1.3 Special Rules

#### Castling

Castling is a special move, in which it is possible to move the king and the rook simultaneously. Two kinds of castling are possible: short castling where the king moves two squares to the right, and long castling where the king moves two squares to the left. The rook is placed beside the king further to the centre.

Castling is not allowed in four cases. 1) If the king and the rook involved have moved earlier in the game. 2) If there is a piece located on the path between the king and the rook involved. 3) If the king is attacked by an opponent's piece. 4) If the path between the king's move from square to square is attacked by an opponent's piece.

#### • Promoting a pawn

If a pawn travels to the back rank (last rank), it must be replaced by any other piece except a king or another pawn.

#### • En passant

If any pawn's capturing ability is "bypassed" by an opponent pawn that advances two squares on its first move, then it is permitted to capture the opponent's pawn. But this must be done immediately on the following move, after that this option expires.

#### • Resign and draw proposals

A player can resign the game which means that he has lost and his opponent has won the game.

A player can propose a draw after making a move. The opponent can accept the proposal in which case the game ends and is a draw, or refuse the proposal in which case the game continues.

#### Repetition of moves

If the same position with the same player to move occurs three times in the same game then the game is drawn, but only if either player claims the game to be a draw.

#### • 50 moves rule

If, at any time of the game, both players play 50 moves without a pawn move or a capture the game is drawn, but only if either player claims the game to be a draw.

#### Stalemate

If either player on the move has no legal move, and their king is not in check then the situation is called stalemate, and the game is an immediate draw.

#### Chess clocks and time

In tournament play each player has a fixed amount of time to play the game. If a player runs out of time he loses the game, and the opponent wins the game. But if the opponent does not have enough material to win the game the game ends in a draw.

There are other rules, stating what should happen in special occasions when the rules are not abided, such as when a player touches a piece, an illegal move is made, etc. Because of the nature of these rules, they have no relevance within computer chess.

# **1.2 Computer Chess Development**

#### 1.2.1 Computer Chess and Artificial Intelligence

"Chess is the Drosophila of artificial intelligence."

-Alexander Kronrod, Russian mathematician (1965)

- The study of the fruit fly has served as a basis for much knowledge of genetics - When Alexander Kronrod was accused of wasting time and resources on a mere game, he drew an analogy between the importance of the research of the fruit fly for genetics, and the progress that Artificial Intelligence (AI) would experience through computer chess research.

The basic principles of chess programming were presented in Claude Shannon's classical paper "Programming a computer for playing chess" [1950]. In this paper Shannon stated that while programming chess was "perhaps of no practical importance, the question is of theoretical interest."

So why study a game? What is the role of computer chess in science, and does the study of chess programming have a higher goal than just to entertain the chess community?

In the famous publication "The Theory of Games and Economic Behaviour" by Neumann and Morgenstern [1944] methods to solve **two-player zero-sum games of perfect information** are presented [Simon et al. 1992]. A zero-sum game is defined as a game where one player's loss is the other player's gain. The game is of perfect information because both players can see the whole state of the game all the time. Chess belongs to this category of games. So according to Neumann and Morgenstern's game theory, solving the game of chess should be a trivial task. However, the number of legal positions in chess is estimated to be in the order of  $10^{43}$  and the number of potential variations from the initial position is estimated to be  $10^{120}$  [Shannon]. There is a general consensus amongst the scientific community that a solution-space of this magnitude is not achievable to explore neither today nor in the future.

Shannon also notes the existence of a perfect game of chess, but that it is practically impossible to compute [1950].

Chess as many other board games, is faced with the *contingency problem*, which states that a state of a game tree is difficult to quantify, because there is a degree of uncertainty introduced by the presence of an opponent. It is not possible to know in advance the outcome of a move, because you do not know the reply of your opponent, and the consequence may not be visible due to our 'look-ahead horizon'.

The large solution space together with the presence of the contingency problem makes the game of chess a complex computing problem that resembles many real-world problems. Shannon noted that other devices of a similar nature would benefit from the approach developed for the chess machine, and suggested a machine capable of logical deduction, designing filter, equalizers, switching circuits etc. [1950]. The Deep Blue project (1997) by IBM - the largest and most expensive chess machine to date gave as an argument for the investment: that the game of chess was merely a good example of a complex problem that could be better solved using parallel processing to give us insight on its effectiveness. A more recent project, the Hydra project, a 16 way cluster computer assembled in 2004 for the purpose of playing chess, gives the same argument for its existence: "Gaining experience in solving problems that require prodigious amounts of computational resources on specialised hardware components".

In addition, the following aspects of chess can be given as arguments why chess is considered to be an excellent research environment to test a machine's ability to dynamically solve problems<sup>2</sup>:

- Chess has clear rules with concrete and simple representations.
- Chess has a standard rating system which measures the strength of a player with a single numerical value. This makes it easy to measure the progress of computer programs.
- Chess is the most popular board game of all games both amongst scientist and the public.

<sup>&</sup>lt;sup>1</sup> http://www.chessbase.com/newsdetail.asp?newsid=1866

<sup>&</sup>lt;sup>2</sup> http://www.chessbrain.net/story.html

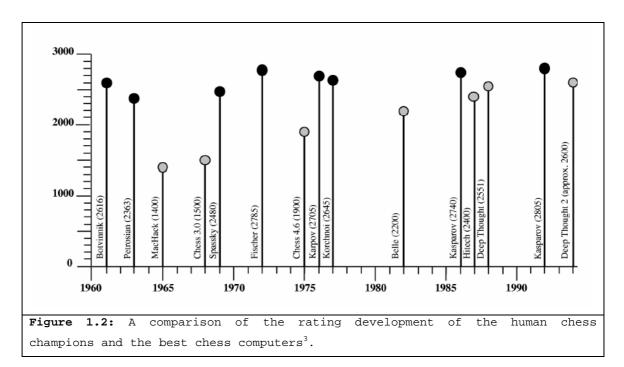
# 1.2.2 A Brief History Overview

The first alleged chess machine was built in 1769 by an Hungarian engineer Baron Wolfgang von Kempelen, and was called "The Turk". The machine won fame for many years, but was finally exposed as a hoax, as there was a dwarf hidden inside the machine, who happened to be an excellent chess player. More than a century later the Spanish inventor, Leonardo Torres y Quevedo, built a mechanical device that could play a King and Rook vs. King endgame. The first great contributions to the subject of computer chess programming were made by two renowned scientists, Alan Turing and Claude Shannon. The first computer chess program was written in 1950 by Turing, who had to simulate the execution of the program with pen and paper [Wall 2004]. The level of play of the program was very low, but it served its main purpose: it showed that computers could play a legal game of chess. In the same year, Claude Shannon wrote the paper "Programming a Computer for Playing Chess" which became the ground-breaking statement of the problem. Shannon distinguished between a "Type A strategy", in which all sequences of possible moves are examined up to a maximum depth level, and a "Type B strategy" in which only a selected subset of possible moves are examined by somehow implementing general chess knowledge to prune away uninteresting branches [Shannon 1950].

The first complete chess program to actually execute on a computer was written by Alex Bernstein et al. in **1957**. The playing strength was very poor though it could win against a complete newcomer. Ten years later Greenblatt et al. developed the first chess program of reasonable strength, MAC HACK SIX. [Marsland 1991:2] The level of play was higher than in all other chess programs and also superior to most casual chess players. This spurred several people to begin developing chess programs and writing proposal [*ibid.*].

The first world computer chess championship was held in Stockholm in **1974** and thirteen computers from eight nations participated. The event was won by a soviet chess program, Kaissa, which was a Shannon type A program. The International Computer Chess Association (ICCA) was founded in **1977** by computer chess programmers with the purpose to organise future

championship events for chess programs. (The organisation was renamed in 2002 to ICGA to broaden the scope to empass the whole community of Computer Games).



During the 1980s the strength of chess computers continued to rise and in **1983** the chess computer BELLE won, as the first computer ever, a U.S. Master title. Five years later a chess computer finally won a human chess grandmaster when DEEP THOUGHT beat the Danish champion Bent Larsen.

The first real commercial interest in computer chess emerged when IBM in 1989 took over the DEEP THOUGHT project and hired the programmers behind the project. The software was further enhanced and specialized hardware was developed, and in 1993 the largest and most expensive chess computer ever was launched: DEEP BLUE. In 1996 the current World Chess Champion Garry Kasparov was challenged to a match. Humankind won the match, but DEEP BLUE surprised many by winning one game against the world champion.

\_

<sup>&</sup>lt;sup>3</sup> Source: http://www.cs.nott.ac.uk/~gxk/papers/cec2001chess.pdf

IBM got a rematch already the following year. DEEP BLUE managed this time to win the match by winning two games while Kasparov only won a single game and three games ending in a draw. "The dream of a world-class chess-playing program, a 50-year quest of the computing science and artificial intelligence communities, was finally realized." [Schaeffer 2000:42]

Kasparov claimed another rematch that was declined and IBM decided to dismantle DEEP BLUE.

The match generated a lot of advertising for IBM as well as for the whole area of computer chess. New chess programs as well as proposals are still being developed, probably more now than ever. Matches between the best chess computers and human world champions have since been arranged, but all ending in a tie.

The history of computer chess development can be divided into eras. Donskoy and Schaeffer suggested the following three eras based on search methods [Donskoy et al. 1989]:

#### • 1950s - mid 1970s, The Pioneering Era

Shannon Type A programs were dominant: focusing on selective search and implementation of as much chess knowledge as possible.

#### • Mid 1970s - late 1980s, The Technology Era

Shannon Type B programs were dominant: focusing on maximizing the speed of search by implementing as simple as possible brute-force algorithms. Specialized hardware was developed followed by parallel systems.

#### • Late 1980s -, The Algorithm Era

The current era went back to focus on developing new search algorithms as well as implementing knowledge into the programs, and at the same using even faster and better specialized hardware.

#### **1.2.3 Sequential Computer Chess**

In order to develop a chess-playing program, the following fundamental problems have to be solved [Marsland 1991:17].

#### • Internal Representations and Move Generation

One of the most important contributions to the development of computer chess was when Professor Robert Hyatt in 1994 released his strong chess engine, *Crafty*, as open source. Crafty's well-documented source-code implements bitboards as the central data structure for internal representations and move generation, and is considered to be the main source for the increasing popularity of the bitboard approach. In a publication Hyatt [1999] gives an explanation of different methods of implementing board representations and introduces the concept of *Rotated Bitmaps* (described in chapter 2.2.2).

A recent master thesis [Rasmussen 2004] sets out to examine how the use of bitboard affects the performance of parallel search. The thesis suggests that parallel search gives a speedup compared to sequential search, but fails to show any connection between this performance change and the use of bitboards.

#### Game-Tee Search

Algorithmic enhancements together with specialized and often parallel hardware are given as the main reasons for the improved playing strength achieved through the last 50 years. One of the greatest advances was laid in 1958 when Newell et al. developed the idea behind the **Alpha-Beta algorithm**, which Knuth et al. in the mid 1970s improved (described later in chapter 4.3).

In his overview of *computer chess and search* which covers developments up to late 1980s Marsland [1991] describes the various enhancements to the Alpha-Beta algorithm (described in more detail in chapter 4.4).

In his work on enhancements of minimax-search Platt [1996] presents a new interesting search algorithm which he calls **MTD(f)**, which pushes the idea of minimal window searches to the extreme.

#### • Position Evaluation

Evaluation of the properties of a chess position is probably the area where the least research has been carried out, and a subject that probably will experience more focus in the future. Recent research has had some attention towards implementing *learning* in to chess programs. One promising technique that has received some attention is *temporal-difference learning*, which aims at deciding the optimal combination of different weights of the evaluation function by continually analyzing all the games played by the chess program [Schaeffer et al. 2002].

Today, a popular area of artificial intelligence is neural networks. Current attempts at implementing learning with neural networks in chess programs have not gained much success. Neural Networks do not work very well in situations they are not trained for, which is a big obstacle because when humans play a game of chess it is common to aim at new unknown positions on the board, which have never been explored before.

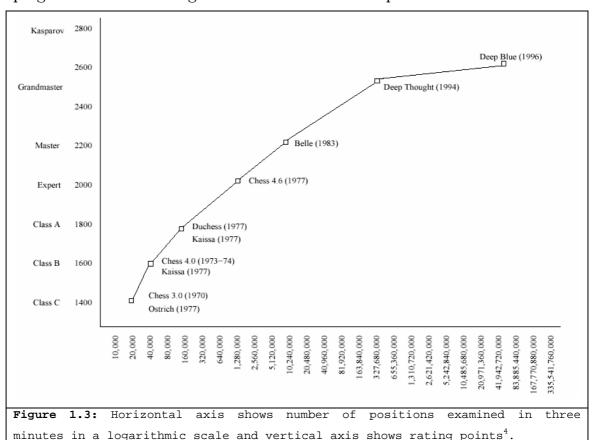
Techniques and advances in implementing chess knowledge are not within the main scope of this thesis, and therefore will only be examined to a limited extent.

#### 1.3 Parallel Search

On a sequential computer there is but one processor. To improve the speed of a program, parallel processing is often employed. Parallel processing is the simultaneous use of more than one processor to solve a problem. In the domain of computer chess programming the problem is searching for the best move in a given position.

# 1.3.1 The Need for Speed

It is said humorously that any dedicated chess programmer is willing to trade any of his kids in return for increased search speed in his chess program. It is considered general chess programming knowledge that there is a strong correlation between the strength of a chess program, and the speed which the program traverses through the search tree of board positions.



<sup>&</sup>lt;sup>4</sup> Source: http://www.cs.man.ac.uk/~schalk/3192/large.pdf

A scientific examination was performed in 1982 by Thompson which clearly indicated that there was a relation between search depth, and the strength of the program. The experiment featured 100 self-play games with matches of 20 games between versions of the chess computer BELLE; each version differing only by the depth of the search. The gain in playing strength for each extra depth increased the playing strength linearly by an average of 246 rating points. Later experiments [Junghans et al. 1997] gave an average increase of strength around 200 rating points for each extra depth searched.

#### 1.3.2 Sources of Parallelism

It is possible to classify the approaches used to achieve parallelism in chessplaying programs into four different strategies [Lu 1993]:

#### • Component Parallelism

The idea of *component parallelism* is to parallelize the computationally expensive part of a chess program. This approach has most often been implemented by utilizing special developed hardware that parallelizes selected tasks, e.g. the Deep Blue project designed chess-specific processor chips that were capable of examining and evaluating two to three thousand positions per second<sup>5</sup>.

#### • System Parallelism

System parallelism is a method that has not gained much attention in game-playing research [ibid.]. The idea is to have different systems each with their own knowledge and search tree that contribute to the overall result. This approach was first tried in the domain of computer chess in 1989 by the program ParaPhoenix [Shaeffer 1989]. Later experiments that focused more on directy distributing chess knowledge were performed in 1994 by [Ciancarini 1994a].

-

<sup>&</sup>lt;sup>5</sup> http://www.research.ibm.com/deepblue/

#### **Parallel Aspiration Search**

In *parallel aspiration search*, the Alpha-Beta search window is divided into non-overlapping segments to examine different search processes. This method was first described by Baudet where he notices a maximum speedup between 5 and 6 independent of the number of processors searching the tree [BAUDET 1978 as cited by Feldmann

#### • Tree Decomposition

1993:461.

Tree decomposition is a strategy for distributing the tree search by splitting the tree amongst many processes that are running in parallel. This method has received a lot of attention and has gained the most success [Lu 1993]. The first algorithm to exploit game trees by decomposing the tree, and searching each segments in parallel was done by Finkel and Fishburn in the early 1980s [Feldmann 1993].

A more recent contribution to parallel search algorithms was described in Brockington's PhD thesis "Asynchronous Parallel Game-Tree Search" [Brockington 1998] that suggests that asynchronous game-tree search algorithms can be as efficient as synchronous methods in determining the best move. Most parallel algorithms follow **synchronous** methods where the search is concentrated within a specific part of the tree or a given search depth, and where the work of each process is somehow related to the other processes. Conversely, an **asynchronous** process will execute its own work without regard for the global state of the search. A major benefit is that a process in an asynchronous search algorithm will, at no time, sit idle while waiting for other processes. [ibid.]

It is important to note that these approaches are **not mutual exclusive**.

Another approach at classifying parallel search algorithms is solely on their algorithmic properties instead of merely focusing on the aspects of the implementation. The interested reader is referred to the excellent paper [Brockington 1996] that classifies all the parallel search algorithms that have been developed during the last thirty years.

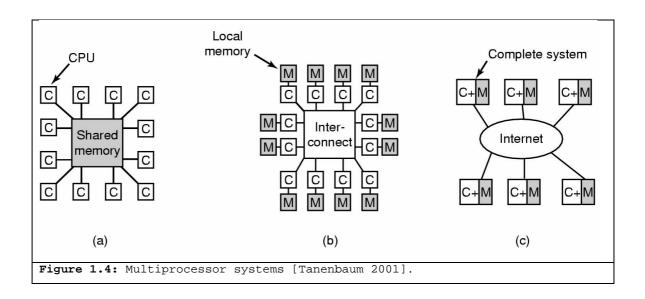
#### 1.3.3 Parallel Architectures

#### What is Computer Architecture?

"The term *architecture* is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behaviour."

-Gene Amdahl (1964)

The parallel architectures of the adopted hardware will often dictate the selection and use of special programming models and techniques to exploit the potential of the hardware. It is very important when designing a parallel gametree search algorithm whether a common memory is shared amongst the processes or not. Tanenbaum (figure 1.4) categorized multiple processor systems into three classes [2001]:



#### Multiprocessors

Multiprocessors is a computer system in which multiple processes share full access to a common memory. In the UMA-architecture all processes share the same memory, while in the NUMA-architecture each processor has its own local memory, but can also access memory owned by other processors. Most top chess programs today support multiprocessor computation. An interesting project is the CILK

language that is built with the purpose of providing a general-purpose parallel programming language for multithreaded parallel programs. "The programmer need not worry about protocols, job coordination, and load balancing, since they are handled automatically by CILK's runtime system". 6 CILK has been used to write at least three strong multiprocessor chess programs.

#### • Multicomputers

*Multicomputers* is a computer system in which tightly-coupled CPUs do not share a common memory. This system is also known as cluster computers or clusters of workstations (COWs) [Tanenbaum 2001].

The probably most famous multicomputer system utilized for chess is the *BeoWulf* cluster where the infrastructure consists of commodity hardware with open source software (Linux). The main deficit with multicomputers compared to multiprocessors is that the communication between processes in multicomputers goes through some interconnect, which is orders of magnitude slower than shared-memory in multiprocessors.

#### • Distributed systems

Distributed Systems gives the option of connecting completely different machines in a joint environment. The connected computers can be located at different locations with different hardware and possibly also different operating systems [ibid.]. A recent implementation of a distributed chess system was the ChessBrain project in 2004. The Danish Grandmaster Peter Heine Nielsen played a draw against more than 2000 computers located all around the world communicating over the internet. ChessBrain consists of a "SuperNode" server that utilizes Hypertext Transfer Protocol (HTTP) and Simple Object Access Protocol (SOAP) to communicate to all the different "PeerNode" clients around the world?

-20-

<sup>&</sup>lt;sup>6</sup> http://supertech.lcs.mit.edu/cilk/home/intro.html

<sup>&</sup>lt;sup>7</sup> http://www.chessbrain.net/

# 1.3.4 The Tuple Space Model

To be able to distribute an application over a heterogenic and often complex network a class of software technologies called *Middleware* is required. A Middleware is defined as the layer of software above the operating system and below the application program, and it provides a common programming abstraction across a distributed system [Tanenbaum 2001].

Many different kinds of middleware are available, e.g. Remote Procedure Calls, Message-Oriented Middleware, CORBA and Remote Method Invocation. A quite different type of middleware was introduced by Gelernter in 1982 in the Linda programming environment which was called the *Tuple-Space Model*. The idea behind the model is the creation of **Tuples** that can easily be accessed by many processes through a **Space**. A tuple can consist of a subroutine, a function, or any ordered set of data. This concept was extended in **JavaSpaces** which introduced **Objects** as the only element in the shared space. JavaSpaces was introduced in 1999 by Sun Microsystems and was released together with the *Jini Service Environment* [Freeman et al 1999]. While Linda had many cross-platform obstacles, JavaSpaces runs on the Java Virtual Machine which can operate on many different platforms.

The only experiments and proposal to implement parallel search of a gametree in a tuple-based environment were, to our knowledge, done by P. Ciancarini from the University of Bologna from 1994-1995.

In his work [1994b] Ciancarini classifies the parallel search algorithms in two classes: *static* and *dynamic* based on the time when the splitting of nodes between different processes occurs:

- **Static algorithm** decides all the splitting nodes before the search starts
- **Dynamic algorithm** decides which nodes to split during the search.

Ciancarini concludes that the static algorithms were much more efficient than the dynamic algorithms [*ibid.*], and suggests that the reason for this is that the inter process communication overhead increases a lot when running on a network of workstations.

# 1.4 Talvmenni ver. 0.1

From the start the aim of this thesis was to undertake the development of a fully-fledged chess playing program. The result is *Talvmenni* version 0.1; a java-based chess engine which also supports parallel computation over a network of workstations.

Our main arguments for deciding to develop and program our own chess engine, as the basis for further examination of JavaSpaces as a parallel search environment, were:

- Almost every serious chess engine today is implemented in C or assembler. JavaSpaces is developed in Java and must therefore be implemented on top of a java-based engine. The chess-engines that we found in our research that were developed in Java did not meet our necessary requirements to object-orientation and modularization.
- As we completely agree with Petur Naur's essay from 1985 "Programming as Theory Building", we believe that programming around a problem builds greater knowledge and theory as basis for understanding the domain of the problem.

Most chess programs consist of two main components: a *user interface* and a *chess engine*. The graphical user interface (GUI) is responsible for what is displayed on the user's screen, while the engine is the part of the program that actually plays chess. The interface and the engine interact with each other through a predefined protocol. Two different protocols are pre-dominant today. The open-source Winboard-protocol that was developed by Tim Mann back in 1980 and the UCI-protocol developed by ChessBase, which is the only large commercial entity in the chess software industry. *Talvmenni* supports, at the moment, the Winboard-protocol and is also able to play in a console-mode that we have developed.

<u>Reference to section 1.2.3</u>: **Talvmenni**'s internal board representation is based on bitboards, which is also used for move-generation and positional evaluation.

**Talvmenni** supports many different search-algorithms. We have developed an object-oriented *strategy-module* that makes it easy to swap dynamically between different search algorithms and evaluations. We have experimented with all the most known algorithms such as Negamax, Alpha-Beta, MTD(f) and Iterative Deepening. **Talvmenni**'s current evaluation function is very simple, mainly focusing on the material balance as this topic is outside the focus of this thesis.

<u>Reference to section 1.3.2</u>: **Talvmenni**'s source of parallelism is a combination of *system parallelism* and *tree decomposition*. We have developed a new algorithm for parallelizing search within the JavaSpace environment that has some resemblances with the Brockington's APHID-algorithm.

<u>Reference to section 1.3.3</u>: **Talvmenni** can execute both as a standalone application as well as in a parallel and distributed architecture over a network of workstations.

<u>Reference to section 1.3.4</u>: **Talvmenni** implements a tuple-based environment: JavaSpaces and Jini using the master/worker-idiom.

# **1.4.1 Development Tools**

We have used the following tools during the development of **Talvmenni**.

#### • SourceForge8

SourceForge is an open-source environment providing the developers with a Concurrent Version System (CVS-repository) and a site for publication of the software<sup>9</sup>. The main task of a CVS-system is to manage the changes in the source-code of a project with many developers working on the project.

#### • ECLIPSE<sup>10</sup>

Eclipse is an open-source Java Integrated Development Environment (IDE) developed by IBM.

#### • JUNIT<sup>11</sup>

JUNIT is a unit-test framework, for testing the correctness of the written source-code.

<sup>&</sup>lt;sup>8</sup> http://www.sourceforge.net

<sup>9</sup> http://www.sourceforge.net/projects/talvmenni

<sup>10</sup> http://www.eclipse.org

<sup>11</sup> http:///www.junit.org

# Chapter

# **Data Structures**

The data structures are the bricks and mortar of a chess program. A good understanding and choice of data structures is important; firstly, for the performance of the program, and secondly, for the ease of design and implementation.

To represent the state of a game of chess we need many different data structures. None of the current alternatives for representing all the tables and data structures necessary to describe a chessboard, are proven to be the optimal solution [Marsland 1991:17].

Structures must be created that state:

- The placement of all the pieces on the board.
- If white and black is allowed to castle kingside and/or queenside.
- If either white or black is allowed to make en passant.
- Numerical position evaluation.
- Who is to move and move number.
- Draw related information.

# 2.1 Board Representations

The representation of the chess board varies quite a lot for different chess programs. Completely different structures have evolved over time, though no radically new structures have been developed during the last 35 years [Laramee 2000].

The first suggestion for an internal board representation came in Claude Shannon's paper [1950]. He suggested using 64 words representing the board.

Each word is a square of the board that can be occupied in 13 different ways. Empty squares are assigned a 0. Each chess piece is given a number: +1 for a white pawn, +2 for a white knight, +3 for a white bishop, etc., and corresponding negative numbers for the black pieces. Thus, the state of a square is specified by giving an integer from -6 to +6.

Today (2004) two different methods are most often implemented: The bitboard method that will be described in section 2.2, and the offset approach that will be examined in the following section.

#### 2.1.1 Offset Representation

The chess board consists of 8x8 chess squares. Therefore it may seem natural to represent the board with an 8x8 matrix of squares. Each matrix holds one value per square: 0 if there the square is empty, 1 if there is a white pawn, 2 if there is a white knight, -1 if there is a black pawn, etc.

References to a square (rank, file) require that rank\*8 + file is calculated, and then this can be used as an offset from the first element of the array (hence the name offset board representation) [Hyatt 1999].

When generating all possible moves for a position it is necessary to loop through all the squares and branch according to the current piece on each square. The pseudo-code in figure 2.1 shows an example of move generation for pawns [Eppstein 1997]:

```
for (i=0;i<8;i++)
1
2
                 for(j=0;j<8;j++)
                      switch (board(i,j)) {
                      case wP: // wP means white pawn
                          if (board(i+1,j) empty) generate move to board(i+1,j)
                          if (i==2 \&\& board(i+1,j) empty \&\& board(i+2,j) empty)
6
7
                              generate move to board(i+2,j)
                                        if (j > 0 && board(i+1,j-1)contains black piece)
                              generate capture of board(i+1,j-1)
                          if (j < 7 && board(i+1,j+1) contains black piece)</pre>
10
                              generate capture of board(i+1,j+1)
11
12
                          break;
13
                      }
```

Figure 2.1: Move generation for pawns.

This structure is easy to implement, but is too slow in the long run because of various boundary checks that must be implemented to ensure that pieces do not move off the board. This problem can be solved by expanding the matrix, and placing sentinels or dummy entries on a two-square border around the actual board.

The variation of the offset representation that is most common in modern chess engines is called **the 0x88 representation**, which is much faster than the first offset version. This method uses a board that is 128 squares. To the right side of the "real" chess board is a "dummy" chess board, so that we have a board of 8 rows of 16 squares each.

The squares are numbered as indicated in figure 2.2: The first eight squares of each row are on the board while the next eight squares are unused.

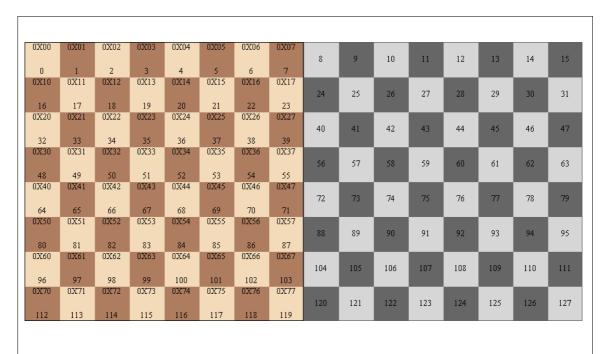


Figure 2.2: The 0x88 board representation.

The format of the *square index* in binary is: **xrrr yfff**.

rrr is the rank number and fff is the file number, and x and y are always 0 for squares that are on the board.

To move one square to the left or right just subtract or add one to the square index. To move one square up or down a file, add or subtract 16 (0x10) to the square index to reach the next rank.

If a bit-wise AND on the resulting square index and 0x88 (binary constant = 10001000) results in a non-zero value, then the piece has reached outside the board. All these calculations are wrapped-around if the result is outside the boundaries 0-127 (0x00-0x7F).

This method for masking can also be used for files and diagonals. A benefit from this method is the ability to easily determine the relation between two squares if they are on the same diagonal, rank or file.

#### 2.2 Bitboards

In the late 1960's developers with the KAISSA team in the Soviet Union apparently were the first to develop the idea of using a bitboard to represent some property in a chess position. The basic idea is to use a 64-bit vector of 0's or 1's, which represents the absence/presence of some property for each square on the board. The chessboard has 64 squares. Thus different information about the complete chessboard can be stored in a 64-bit integer data type, e.g. figure 2.3 indicates a bitboard mapping that registers the placement of all white pawns on the board.

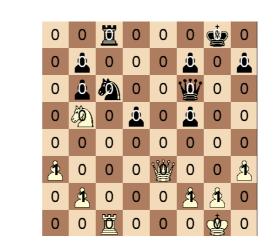


Figure 2.3: A bitboard mapping for all white pawns.

Most mainstream processors currently available are still using a 32-bit architecture, and therefore cannot directly manage 64-bit instructions. Most compilers today, though, support 64-bit variables, but split the variables into two 32-bit instructions before sending them to the processor. The trend is that 64-bit processors become more and more common. The Java primitive data type **long** is a 64-bit integer using the two's complement format. Java, C and C++ all support bitwise operations and all share the same syntax for bitwise operations.

**Talvmenni** is using bitboards as its core data structure. The standard bitwise operations are listed in Appendix C.

#### 2.2.1 Pre-generated Bitboards

The most computationally expensive part in a chess program is the calculation of the list of legal moves for a given position [Laramée 2000]. To minimize the required runtime calculations, some pre-generated structures can be calculated preferably already at compile time and looked up during runtime.

**Talvmenni** implements 64 bitboard constants representing all the squares on the chessboard, which are the building blocks for further bitboard constants. All square constants have only a single bit set and are ordered so that the first bit (leftmost) maps onto the square A8. The second bit that is set maps onto the square B8, and so forth to the last bit (rightmost) that maps onto the square H1. Mapping the squares in this order makes it easy to output the chessboard by slicing the bit-string in 8 rows. Figure 2.4 displays a unit-test for the bitboard constant Square.\_G7, which represent the square g7 and shows the sliced bit string formatted in the source code so that the chessboard is easily read as seen by the player with the white pieces.

Figure 2.4: A unit-test mapping the bit string for G7.

[org.forritan.talvmenni.bitboard.SquareTest]

A bitboard-constant representing a rank is pre-generated as bitwise-OR's of the eight bitboard-constants representing the squares in the corresponding rank. In figure 2.5 the construction of the constant for the first rank is shown.

E.g.: This pre-generation for the constant \_1 representing the first rank can also be shown as a bit-string calculation:

The other seven ranks are constructed in the same way. Files and diagonals are also defined in this manner, e.g.: figure 2.6.

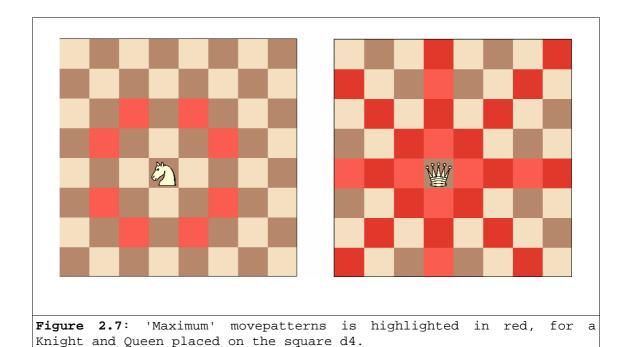
Although this kind of coding may seem somewhat tedious at first, it is a great help when the bitboard patterns get more complicated, because each bitboard constant is coded twice – first, when the constants are defined in the source-code, and secondly, when unit-tests are written for testing these constants. By using a unit-test framework during development, this "double book-keeping" of the source code ensures that it is possible to verify the correctness of the constant definitions at any time.

```
public void testC() {
      Assert.assertEquals("" + //
             "00100000" + // A8-H8
            "00100000" + // A7-H7
            "00100000" + // A6-H6
            "00100000" + // A5-H5
            "00100000" + // A4-H4
             "00100000" + // A3-H3
             "00100000" + // A2-H2
             "00100000", // A1-H1
            (this.zeroPrefix[
Long.toBinaryString(File._C).length()
Long.toBinaryString(File._C)));
   public void testA2G8() {
      Assert.assertEquals("" + //
            "00000010" + // A8-H8
            "00000100" + // A7-H7
            "00001000" + // A6-H6
            "00010000" + // A5-H5
            "00100000" + // A4-H4
             "01000000" + // A3-H3
             "10000000" + // B2-H2
            "00000000" // A1-H1
      , (this.zeroPrefix[
Long.toBinaryString(SlashDiagonal._A2G8).len
gth()
                           11
Long.toBinaryString(SlashDiagonal._A2G8)));
                                                  [org.forritan.talvmenni.bitboard.FileTest]
Figure 2.6: Unit-tests for file C and diagonal a2-g8.
                                                  [org.forritan.talvmenni.bitboard.SlashDiagonal
```

Any given chess piece at any given square has some maximum potential moves as defined by the rules of the game. The rules state a move pattern for the six different types of pieces which are limited by the placement of all other pieces on the board.

**Talvmenni** implements pre-generated bitboard constants and corresponding unit-tests for the static part of the chess rules. These are the "maximum" move patterns for all pieces on any of the 64 squares on the chessboard, i.e. as if the piece stood alone on the chess board.

The purpose of these pre-calculated constants is to avoid the need for calculating the statically known part of the chess rules, and instead create pre-generated maps of these constants, thus making it possible to lookup the move pattern, for a given piece, at a given square as shown in figure 2.7.



The dynamic part of the chess rules, i.e. the limitation of the move-patterns by the placement of other pieces on the chessboard, is not feasible to precalculate - not for more than a few pieces anyway. If it were we would have a complete solution to chess.

Thus in *Talvmenni* the actual possible moves for a given piece at a given position must be calculated at runtime by bitwise manipulating the returned bitboards from the lookups with the bitboards for both own and opponent's pieces. This is described in more details in section 2.2.4 – "Talvmenni bitboard Masks".

#### 2.2.2 Rotated Bitboards

The *sliding pieces* – rooks, bishops, and queen – are more difficult to handle with bitboards because they must coordinate their movements according to the placement of all other pieces. Professor Robert Hyatt introduced in Crafty a new technique to handle this issue known as *rotated bitboards* [Hyatt 1999], and the programmers of the chess program DarkThought did the same.

A normal bitboard has one byte (8-bit) for each rank on the chess board. This allows for an efficient lookup of rook attacks from a lookup table, which is indexed by the attacking square and an occupancy bitboard.

#### • Calculating occupancy for the ranks:

Rank is a number from 1-8 representing the rank to be examined. AllPieces is a 64-bit bitboard for the positioning of all the pieces. Thus occupancy can be calculated as: (AllPieces >> ((Rank-1)\*8)) & 255 (0xFF)

By first shifting the rank to the 8 lowest bits and then masking these 8 bits by a bitwise AND (&) with 255 (0xFF), a 8-bit occupancy bitboard is returned for the rank.

The concept of rotated bitboards is to rotate the "normal" bitboard, and thereby it is possible to re-use the lookup tables for rook attacks.

Files are computed by rotating the occupancy-bitboard 90 degrees, and in the same way as above, the rook attacks on a file can be looked-up. This is implemented by maintaining an additional bitboard, which is updated after each move, because it is too expensive to calculate dynamically.

Diagonals are computed by rotating the occupancy-bitboard 45 degrees left or right. Bishop attacks can be determined by a table lookup for the attacking square together with these rotated bitboards. Queen attacks are determined by bit-wise OR'ing the rook and bishop attacks.

#### 2.2.3 Talymenni Bitboard Iterator

As the bitboards in **Talvmenni** are represented in two complement form it is possible to effectively iterate on <u>only</u> the bits that are set in a bitboard by using the technique described in [Warren2003] where the lowest bit set in a number x can be found with x & -x.

In **Talvmenni** a general BitboardIterator shown in figure 2.8 is defined to encapsulate this iteration. This BitboardIterator also implements the general java.util.Iterator interface.

```
package org.forritan.talvmenni.bitboard;
import java.util.Iterator;
public class BitboardIterator implements Iterator {
   private long bitboard;
   public BitboardIterator(
         long bitboard) {
      this.bitboard= bitboard;
   public boolean hasNext() {
      return this.bitboard != 0L;
    * @deprecated Use nextBitboard() instead...
   public Object next() {
      return new Long(this.nextBitboard());
   public long nextBitboard() {
      long result= this.bitboard & -this.bitboard;
      this.bitboard= this.bitboard
            ^ result;
      return result;
   }
   public void remove() {
      throw new UnsupportedOperationException();
Figure 2.8: The complete code for the bitboard-iterator.
                            [org.forritan.talvmenni.bitboard.BitboardIterator]
```

The BitboardIterator is used all the time during move generation, so in order to optimize performance the nextBitboard() method (which returns a long i.e. a primitive java data type) is used instead of the next() method. The next() method must – as it implements java.util.Iterator - return an instance of the class Object (or the class Long in this case) instead of a primitive java data type as long.

Being able to easily and effectively iterate over bitboards and getting bitboards that represent only one square from the iterators nextBitboard() method, the next challange is to be able to use these square bitboards as keys for lookup in arrays. But in Java these keys must be 32-bit integers, so to map the 64-bit bitboards to a 32-bit integer some effecient method of conversion is needed. One solution to this problem is a technique called the deBruijn algorithm [deBruijn 1998].

Each 64-bit bitboard (long) representing a square returned from the method nextBitboard() is converted into a 32-bit int value which can be used as an index in an array.

```
public interface Square
   public static class Util {
       * de Bruijn sequence 0000 0011 1111 0111 1001 1101 0111 0001 1011 0100
       * 1100 1011 0000 1010 1000 1001 or in hex 3F79D71B4CB0A89
       *@see http://supertech.csail.mit.edu/papers/debruijn.ps
       *@see http://www.cs.princeton.edu/introcs/31datatype/DeBruijn.java.html
      public final static long DEBRUIJN64 = 0x3f79d71b4cb0a89L;
. . .
      /**
       * @param square
       ^{\star} @return a unique index between 0 and 63
      public static int deBruijn64Index(
            long square) {
         square*= Util.DEBRUIJN64;
         return ((int) (square >>> 58));
      }
```

Figure 2.9: deBruijn method in the class Square.Util.

[org.forritan.talvmenni.bitboard.Square]

All the squares have the property that just one single bit is set, and thus they are all powers of 2. Multiplying by a power of 2 is equivalent to a left shift. When multiplying a square and the deBruijn sequence in the method deBruijn64Index(long square) shown in figure 2.9, the result is a long where the most significant six bits are unique for this square. From this a unique 32-bit int value can be created simply by right-shifting with zero-fill the long with 58 bit-positions, so that the most significant six bits are shifted into the six least significant bits. The long value is then finally casted to the type int.

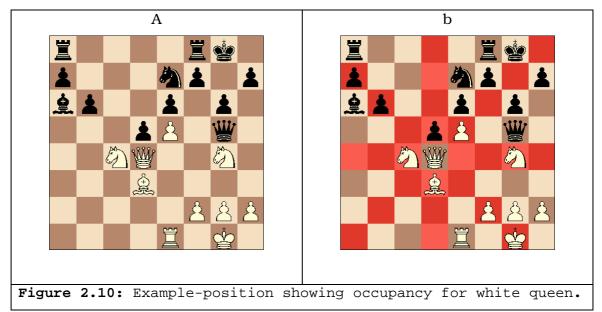
#### 2.2.4 Talymenni Bitboard Masks

Rotated bitboards are not implemented in *Talvmenni* to handle sliding pieces. Instead another approach is used in which the key elements are some pregenerated bitboard masks, which in *Talvmenni* are baptized as "Talvmenni Bitboard Masks", and also the Bitboard Iterator described in the previous section.

The main idea is to use 64-bit bitboards for all operations, and the bitboards are always mapped in the same way to the chessboard. Rotated bitboards, on the other hand, convert 64-bit bitboards into 8-bit bitboards for all operations and therefore require more bitboards with different mappings to the chessboard at all times.

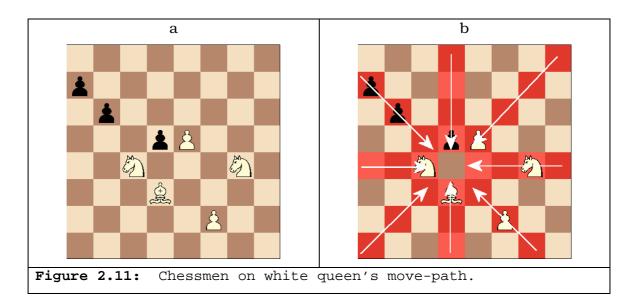
Instead of only looking at a rank, file or diagonal at the time, the calculations in *Talvmenni* are always done for the whole board with regards to a specific piece placed on a specific square.

E.g.: to calculate the occupancy for the white queen in the position in figure 2.10a:

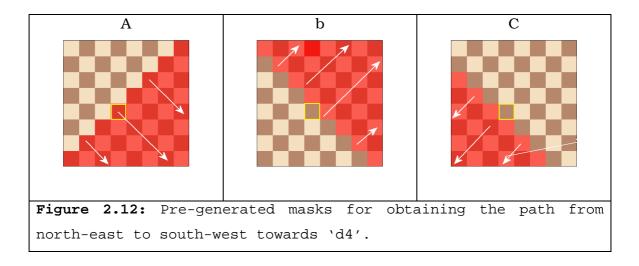


First, a pre-generated 64-bit bitboard-mask is retrieved for the queen's largest possible move-pattern from the square 'd4' as indicated in figure 2.10b.

Secondly, a bitboard for all pieces are bit-wise AND'ed (&) with the retrieved mask resulting in the bitboard indicated in figure 2.11, in which only the pieces that are located on the queen's move-path remain.



The arrows in figure 2.11b indicate the 8 move-directions that must be examined to decide the occupied square closest to the square 'd4'.



For each direction three pre-generated bitboard-masks for the square 'd4' as indicated in figure 2.12 are looked up and bit-wise OR'ed with each other resulting in a new bitboard-mask indicated in figure 2.13a which then is complimented giving a mask for one of the 8 possible directions indicated in figure 2.11b. This mask is then bit-wise AND'ed with the board indicated in figure 2.11a resulting finally in a bitboard where only pieces on the given

direction are set as in figure 2.13c.

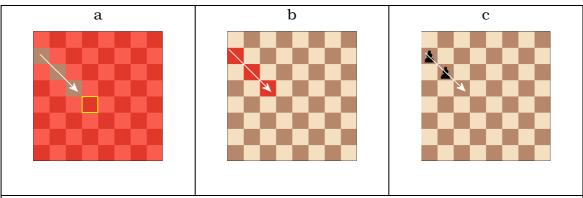
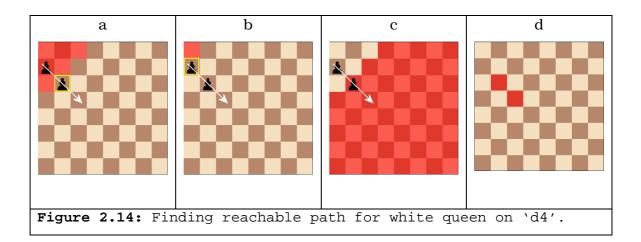


Figure 2.13: Chessmen masked out with north-west to south-east path towards 'd4' mask.

The next task is to mask out those squares which are unreachable for the queen on the given path, in this case the pawn on the square 'a7'. This is done by iterating over each square that is occupied in the bitboard in figure 2.13c and looking up pre-generated bitboard-masks for each of these squares as indicated in figure 2.14a and figure 2.15b.



These bitboard-masks are then bit-wise OR'ed with each other and the result is complimented giving the bitboard-mask indicated in figure figure 2.14c, which then is bit-wise AND'ed with the bitboard indicated in figure 2.13b resulting in the squares which are reachable for the queen from the square 'd4' in the direction examined (figure 2.14d). Note that this is including squares that are potentially occupied by own pieces.

When all the other seven directions are similarly examined the resulting bitboards are bit-wised AND finally giving a bitboard containing only all the squares that are possible for the queen to either move to or to capture. Those squares which are occupied by ones own pieces are thus guarded by the queen (figure 2.15).

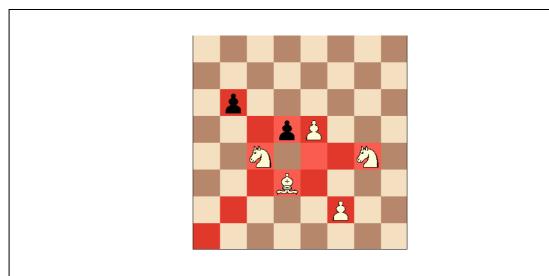


Figure 2.15: All reachable paths for white queen on 'd4'.

#### 2.3 Move Generation

There are three major approaches to solving the task of move generation. *Talvmenni* adopts an approach that can be called "full move generation" which will be described in section 2.3.1. The two other approaches, which will be described below, are not as popular today as the "full move solution":

#### **Selective Move Generation**

The program first evaluates the position of the board and determines a limited list of moves that are expected to be the possible best moves. This approach derives from Shannon's early suggestion how to create a chess-playing algorithm and corresponds to his type B strategy [1950]. This method resembles the way in which humans play chess. An experienced player is able to prune a complete list of possible moves almost intuitively, and grandmasters typically examine only a handful of candidate moves from one position. Almost all early attempts adapted this approach, as did Alan Turing's famous program from 1951, which only generated lines involving captures. This method is almost completely abandoned today, because it has not shown the possibility to create a position evaluator that is accurate enough. Even a position evaluator that selects the list of best possible moves with only a faultrate of 1 % is not accurate enough, because with 30-35 moves as average length of a game the program will make a blunder in every third game [Laramée 2000]. However, the associated search approach, called forward pruning, has produced some techniques that are useful in modern chess engines which will be discussed in Chapter 4.

#### **Incremental Move Generation**

This approach merges the move generation with the search algorithm. The program first generates a limited list of possible moves, and if no adequate move is found within the list additional possible moves will be generated. If a good move is found within the list then no more possible moves will be generated and the search is terminated. The main advantage is that it requires

less memory compared to the complete move generation, which was of great benefit to early chess computers, which had only a limited amount of memory. The method is however, somewhat complicated to program, and it is not often used today. Though, the idea of ordering the search of possible moves is of great importance to modern chess engines and will be discussed in section 4.4.1.

#### 2.3.1 Full Move Generation

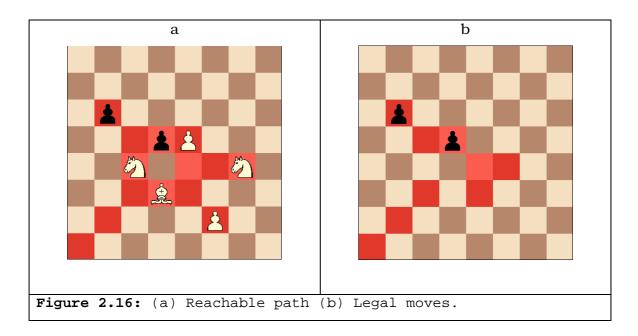
The most straightforward way – and the chosen method used in *Talvmenni* - to implement full-move generation is to generate all the legal moves available for all positions. Many chess engines implement pseudo-legal movegeneration, which allows generation of positions where the king is left in check for many moves. These position are not legal, but are temporarily allowed for optimization causes, as the issue will be dealt with during the search and evaluation process. Pseudo-legal moves are not generated in *Talvmenni*.

In **Talvmenni** a position is represented by the following collection of bitboards for each side:

- King(s)
- Queens
- Rooks
- Bishops
- Knights
- Pawns
- Castling
- En passant
- All pieces

These are used when the list of possible moves is being generated. The bitboards for each type of chessmen is iterated, and for each piece or pawn found a list of legal moves is generated.

E.g. to find the possible moves for the white queen on the square 'd4' in the position in figure 2.10a, first, the last bitboard found in section 2.2.4 which, only contained all the squares that are possible for the queen to either move to or to capture, and those squares occupied by own pieces, which thus are guarded by the queen (figure 2.16a), is calculated.



Then this bitboard is bit-wise AND'ed with the complemented bitboard representing all the white pieces resulting in the bitboard indicated in figure 2.16b.

Finally, the move list for the white queen can be generated by iterating over this bitboard as the bits that are set represent the squares, which the queen can move to from the square 'd4'.

#### 2.3.2 Perft Calculation

Perft calculation was probably first introduced in the popular open source chess engine, Crafty, by Professor Robert Hyatt. The Perft function requires two parameters as input: the start position given in FEN notation and the depth number as an integer. *Forsyth-Edwards Notation* (FEN) is a standard notation for defining all the properties of any board position.

The returned value from the Perft function is the total number of traversed positions starting from the supplied position down to the specified depth.

The Perft function has three main purposes:

- To confirm that the move generator and tree search is working properly.
- To verify that a chess program correctly implements the rules of the game.
- To measure the speed of the move generator.

The idea is to take different, already examined positions where the correct Perft values are known, and match these values with the values that your chess program returns. If the values are not equal, then there is some flaw in the move generation of the chess engine. A Perft-method has been implemented in *Talvmenni*, and the performed Perft-tests will be described in chapter 6.

# 2.4 Chapter Summary

Bitboards have the great advantage that they compute evaluations much faster than the off-set method, because they are able to compute several moves at the same time. Whereas the main advantage of the offset method over bitboards is that it is easier to understand for the programmer. The bitboard approach lacks clarity in the programming methodology [Hyatt 1999]. We claim, however, that this disadvantage in the bitboard approach can be minimized by applying good object-oriented design principles that make it possible to hide the representation details behind classes, and by using explicit and descriptive naming.

As described in section 2.2 rotated bitboards are not implemented in *Talvmenni*, but an experimental approach of our own is used and therefore it would be an interesting test to implement a version of the class Position.Bitboard using rotated bitboards and compare these two approaches.



# Representing Chess Knowledge

This chapter discusses some of the most used approaches to embed general chess knowledge into chess programs.

#### 3.1 Introduction

The world's fastest chess computer in history was Deep Blue. It won fame in 1997, as told in the introduction, when it finally beat the world's strongest chess player, Garry Kasparov.

Deep Blue processed 200 million positions per second, while Kasparov is estimated to evaluate approx. three positions per second<sup>12</sup>. Still - the game was a close race, Deep Blue winning two games, Kasparov winning a single game and three games ending in a draw. This indicated clearly that it was not just a matter of brute-force capacity, but also a question about the purpose of brute-force and how it was exploited. It is generally agreed that Garry Kasparov has a very large amount of chess knowledge while Deep Blue, compared to Kasparov, had a very small amount of chess knowledge. It is vital to any chess program to implement an efficient and good *evaluation function* that encapsulates chess knowledge so the chess program can identify good positions, which it is aiming to obtain [Schaeffer 2000:13.] It would not do any good for a program to process 200 million positions per second if it were not able to distinguish a good position from a bad position.

The concept of an evaluation function was introduced in Shannon's classical paper [1950]. The elements of positional evaluation that he described are still today the basis for modern chess engines, and will be described in section 3.2.

<sup>12</sup> http://www.research.ibm.com/deepblue/meet/html/d.2.html

#### 3.2 Position Evaluation

In order to make your a chess engine play good chess one must establish a method of numerical evaluation for any given chess position. A human chess player looking at a position on the chess board can give an objective estimate as to which side, white or black, has the advantage. Often different players have different judgement of the same position on the board. The reason is that they look at different characteristics of the position, as there does not exist a critical solution to positional evalution. This makes it very difficult to quantify and embed the positional elements of a position into a chess program.

For generations a wealth of chess literature has been written by chess grandmasters describing various positional elements, which are deemed good and others, which are deemed poor. Some of these concepts are relatively simple to program and embed as chess knowledge in to a program, while others are more difficult to define in simple mathematical metrics.

A function must be developed to evaluate all the different positional features and can be given as: E(P) = SUM (F(P) \* W(P)), where F(P) is a feature of the position and W is a weight determined for this feature.

#### 3.2.1 Material Balance

The material balance is an account of which pieces are left on the board for each side [Laramée 2000]. This element is by far the most important concept of the positional evaluation, and also the easiest to implement.

Classical chess litterature states that a queen is worth 9 pawns, a rook is worth 5 pawns and bishop and knight are worth 3 pawns. The value of the king is set high above the value of all other pieces, e.g. 1000 pawns.

As a rough approximation, a position can be evaluated by merely summarizing the total value of the pieces for each side measured in pawn units. In the beginning of the game the material score can be calculated as:

9.0 + 2\*5.0 + 2\*3.0 + 2\*3.0 + 2\*3.0 = 39.0 +the value of the king.

#### 3.2.2 King safety

Since the loss of the King means losing the game, the safety of the king is an important element in positional evaluation. But it is complex to calculate the king's safety, and implementation methods and approaches vary a lot between different chess engines.

There are 2 major aspects that must be considered to ensure the safety of the king:

#### • Pawn structure around the king

It is considered as general chess knowledge that the king is best protected with a "shield" of own pawns right in front of the king [Seirawan 2003]. Therefore it is recommended that the king is castled to either side as the centre pawns usually are advanced to support the development of the bishops. Castling not only gets the king into safety, it also brings one rook into play.

#### • Piece placement

The first basis for an assault on either king is that aduquate pieces participate in the attack. An old chess saying states that the attacker must mobilize one more piece in the attack than the defender if the attack is too become a succes. The evaluation function will therefore award piece movements in the area around the opponents king. The presence of the queen is also taken into account, because if the queen participates in the attack the possible outcome of check-mate increases.

The are many other factors that can be incorporated into the king's safety evaluation function, such as if there are open or half-open files close to the file where the king is placed. *Talvmenni* awards a penalty score if the king is close to the centre of the board in the opening of the game, and a bonus score if the king is close to the centre in the endgame.

#### 3.2.3 Mobility and Development

It is, generally, considered to be an advantage to have a high number of possible moves to choose between in a position, because then it is more likely that you can select at least one good move.

This element is also easy to implement, as the number of possible moves are easily summarized and multiplied with some predefined constant. Shannon suggested that each possible move was worth 0.1 pawn [1950].

At the inital position of the game the mobility score can be calculated as :  $n^* \ w = 20 * 0.1 = 2.0$ .

The first principle a new chess student is taught is the fundemental rule to develop the pieces from the back rank as fast as possible [Seirawan 2003]. The closer the pieces are to the center of the board, the larger is the space of squares that the pieces cover. E.g. a knight located on it's inital position covers an area of maximum 3 squares, while a knight posted on the square 'c3' hits 8 squares.

The positional element of development and board control can be implemented in many different ways. It is possible to either reward or penalize certain selected factors, e.g. award a penalty of 0.15 points to pieces that never have been moved away from it's inital position.

#### 3.2.4 Pawn Structure

The pawn structure is the placement of the pawns on the board ignoring the placement of all other pieces [*ibid.*]. Mostly pawns can move a single square forward and are limited to maximum 5 or 6 moves per pawn in a game. The structure of the pawns therefore changes slowly through out the game, and lay as the foundation for the course of the game. A strong and "healthy" pawn structure supports the pieces both in attack and defence, while weaknesses in the pawn structure become targets for the opponent.

The most important elements of the pawn structure are doubled, isolated and passed pawns.

Doubled pawns are, generally, considered to be an disadvantage, and are therefore often penalized in the evaluation function. Doubled pawns cannot protect each other and are therefore vulnerable when attacked.

Isolated pawns are also, generally, considered to be an disadvantage for the same reason stated above, as it has no neighboring pawns to support safety and protection. In some positions isolated pawns can be utilized as an advantage, because they can enhance the mobility of the pieces of the side that posesses the isolated pawn.

Passed pawns are considered to be an advantage, because they have a large potential of promoting to a major piece [Seirawan 2003]. The phase of the game plays an important role, as the threat of promotion is greater when there are less pieces left on the board.

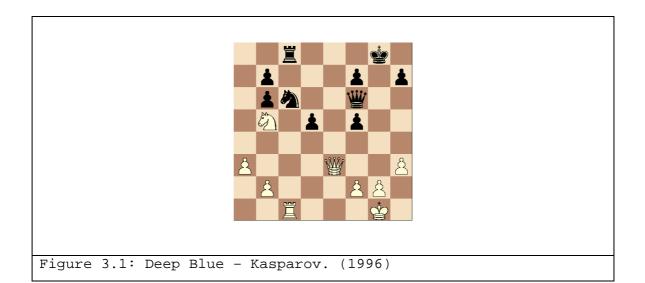
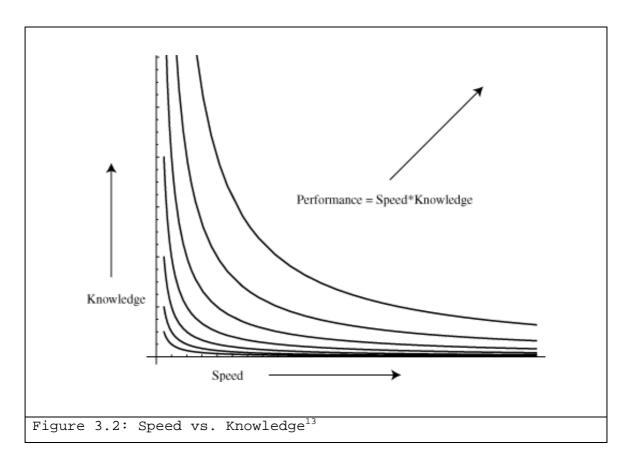


Figure 3.1 shows an interesting position from a game between Deep Blue (white) and Garry Kasparov (black) from their match in 1996. The computer has been able to provoke many weaknesses in blacks pawn structure of the human world champion. Black pawns are doubled both on the b-file and the f-file and all black pawns are isolated as none of them are guarded by own pawns. Deep Blue continued the game by putting pressure onto black's weak pawns and finally won the game in 36 moves.

Other features of the pawn structure can also be implemented and valued in the evaluation function, e.g. pawn chains, hanging pawns, pawns majority, pawn islands, e.t.c. It is possible as well to encode a lot of other chess knowledge into the evaluation function. But the more complicated the evaluation function is, the more computation is required, which is likely to result in slower execution. Typically, the strength of a program has been estimated as behaving like the product of knowledge and speed as seen in figure 3.2:



It is important to carefully select what knowledge to implement in the evaluation function, and find the optimal balance between speed and knowledge. Implementing extra knowledge into an evaluation function that is ineffective, can cause the chess program to perform worse. This is the main cause that a newer version of a chess program actually might perform worse than older versions, and is being refered to as "the program is out of form".

<sup>13</sup> http://www.ics.uci.edu/~eppstein/180a/990114.html

#### 3.3 Opening Book

During the opening phase of a chess game the evaluation function combined with search have proven to be insufficient to find the best moves. Most chess programs therefore implement an "Opening-Book", which is a kind of a database containing set of positions, along with associated recommended moves. Human players have over time gained extensive experience playing the opening phase, and have documented the found results in litterature about openings. By building and implementing opening books into the chess programs this knowledge becomes availabe to the chess programs.

Most chess program build their own propritary opening-book as an effort to achieve an advantage over other competing programs. The main functions of different opening books are essentially the same, but the implementation might vary.

#### **Building an Opening-Book**

Essentially, there are two approaches to building an opening-book. You can either build the opening-book manually or create it automatically.

The most used method is to read and process a large number of grandmaster games from some database and calculate a statistical profile of all the positions occurring in all of the games. Those moves for a given position that frequently result in a win for either side are considered good moves. These analyzed positions are then added to the opening-book together with selected moves. Only positions from an arbitrary depth from the start-position are taken into account, as the large branching factor results in a large space of positions that only occur in a single game. Lines that are only played by a single player in a single game can not be trusted to be good lines.

The second method is to build the opening-book manually. This method is, of course, much slower and is not practical when building a large opening book. In addition, this method requires quite advanced general chess knowledge together with deep insight into how the program plays chess, to be able to select opening-lines that are suitable for the program.

#### 3.4 Hash Tables

Two important functions of hash tables in chess programs are:

- To determine whether a position is being repeated. The rules of chess state that if the same position has been repeated three times then either player may claim a draw.
- To keep track of which positions have been searched before. What evaluation score was calculated for them, and to which depth they were searched at the time. Hash-tables in this context are often referred to as Transposition Tables [Marsland 1986].

Hash tables can also be employed to store results for parts of computations in the evaluation functions, which rarely change in value [Laramée 2000]. If the value instead is looked-up from the hash-table, valuable time can be saved. Examples of such elements are:

**Pawn structure:** The positions of all pawn on the board rarely changes as there are few possible pawn moves.

**Material balance**: The summarized balance of material changes only after a capture or a pawn promotion.

# 3.4.1 Transposition Tables

When a chess program searches through the game-tree to find the best move to play, the same positions often occur repeatedly. This is because the same position on the board can be reached through different move ordering.

An example is the position in figure 3.3 which can be reached by: white can advance the pawn on the first move and on the second move advance the knight or; white can choose to advance the knight on the first move and advance the pawn on the second move.



Figure 3.3: Positions can be reach by different move ordering.

When human players encounter the same position that they have evaluated before, they immediately identify the position as a known position, and therefore do not spend time on the same calculations and evaluations.

The chess program MAC HACK was the first program to implement a **transposition table** [Feldmann 1993] back in 1967. Before evaluating a given position, MAC HACK looked in its transposition table to see whether it had already evaluated an equivalent position. Most, if not all successful programs, have since employed transposition tables. A transposition table is a cache that contains information about positions recently visited by the search-algorithm. [Schaeffer 2000: 7]

A transposition table entry stores the following information:

- the move-list ordered by the search.
- the score of the search returned.
- the depth of the search.
- upper and lower bounds (bounds are described in chapter 4).

**Talvmenni** implements its transposition table as a hash map with an upper bound on the number of entries, with continuous memory allocation. The transposition table in **Talvmenni** implements an aging strategy, so when the upper limit is reached the eldest entry is removed from the transposition table before a new entry is inserted.

# 3.5 Endgame Databases

The third and last phase of a chess game is called the endgame. There is no clear-cut definition of when a middle-game transforms to an endgame, but there are some general characteristics [Seirawan 2003]:

- There are relatively few pieces left on the board, especially the queens have often left the board.
- The role of the kings changes from mainly hiding away from the action to participating actively in action.
- The promotion of pawns often becomes an important goal for both sides.

Chess computers have reputation for performing worse in the endgame than in the opening and middle-game, because this phase requires long-term planning, much further than the look-ahead horizon of implemented search algorithms. Two different approaches are used as a remedy for this problem:

#### • Implementing an Endgame Database

Chess endgame databases, also referred to as tablebases (EGTB), have the function to store all possible positions with a given set of material and the result for the given position, if both sides play all the "correct" moves. Today all possible positions with a set of up to five pieces (king + king + three other pieces) and a subset of positions with 6 pieces on the board are stored in endgame databases.

#### • Adjusting the Evaluation Function

In the endgame, the role of all the different pieces changes as described in the beginning of this section. This observation supports the idea to adjust the weights and values in the endgame by some special endgame heuristic. Another different idea is to strip down the evaluation function to speedup the search so the program searches deeper into the gametree.

### 3.6 Chapter Summary

The weakness of all evaluation function is that they are a heuristic estimate. It is not possible to know the correct value for positional elements, and we can therefore only speculate about the approximate value based on our subjective experience.

The evaluation function implemented in *Talvmenni* is very simple, as we decided that the area of knowledge representation was not within the main scope of this thesis. But as described in the introduction of this chapter, a chess program must be able to distinguish between a good and a bad position to be able to play a proper game of chess. *Talvmenni*, at present, evaluates elements as material balance, simple mobility and a simple king safety function that keeps his highness out of too much trouble.

**Talvmenni** implements an opening book, which reads a book-file that is borrowed from the open-source BeoWulf project<sup>14</sup> and which recognizes 135.730 different positions in the opening phase of the game. **Talvmenni** does not, at present, implement any endgame database.

\_

<sup>14</sup> http://www.chessbrain.net/beowulf/

# Chapter

# **Sequential Search**

In this chapter we discuss the search algorithms that are used for sequential game-tree search and are the basis for our distributed search.

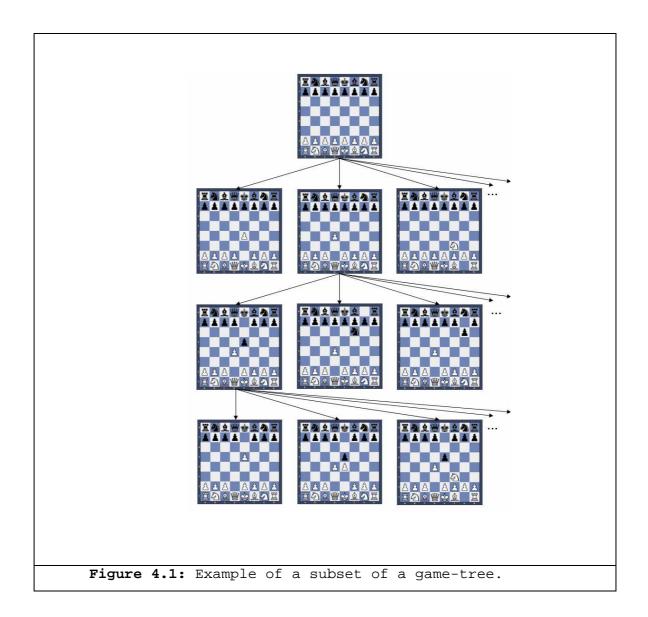
#### **4.1 Game Trees**

Agame of chess can be modelled as a *tree*, each node representing a possible position.

- The root corresponds to a given position of the game from which the search is initiated.
- Edges from a node represent possible moves for that given position.
- The children of each node correspond to those positions that are possible to obtain when moving from the parent position.
- There are two players white and black who alternate making moves.
- The leaves in the tree are essentially end-positions representing either a win, draw or loss for the player at the root node.

A *depth-first* search is commonly used in chess programs. That is, from the root position the first branch (i.e. left-most) is recursively expanded until a leaf node is reached and evaluated. The remaining branches are then traversed from left-to-right in turn as the search algorithm backtracks up to the root position.

To represent the depth in a game tree the term *ply* was introduced in 1959 by Arthur Samuel in his paper on machine learning [Marsland 1991]. A ply is also called a "half move". That is a move by one side, either white or black. Searching to ply depth 2 means that the program is going to search all the legal moves in the current position, and for each of these moves it will also search all possible answers.



The **branching factor** is the number of children at each node meaning how many legal moves are possible to play in a given position. This value is not uniform, so an average branching factor can be calculated. A game of chess has an average branching factor of 35 [DeGroot as citet by Shannon 1950]. In an average game each player makes around 50 moves resulting in the large solution space of approx.  $10^{120}$ , as described in section 1.3.1.

#### 4.2 Minimax

The *Minimax* algorithm is the core algorithm in almost all chess programs and is responsible of selecting the next move that will be played. The Minimax algorithm is often applied in two player games, such as chess, Othello, tic-tactoe and checkers. These games are categorized as **two-player deterministic zero-sum games with perfect information** [Neumann et al. 1944].

The term *two-player* refers to that there are two opposing forces. The game is *deterministic*, because all positions including the initial position are either winning for white, winning for black or drawn. The game is a *zero-sum* game because the score of the same position for one side is the negation of what it is for the other side. The game is of *perfect information* because both players can see all of the chess board and the moves that can be played are deducible by both players.

All nodes in a minimax game-tree are either:

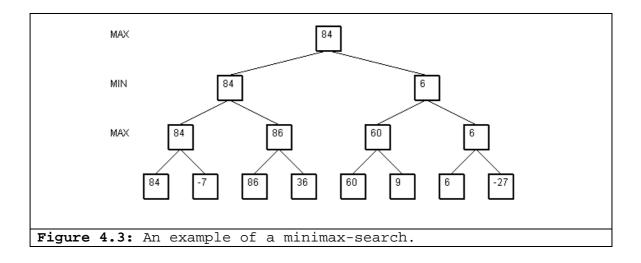
- **Max-node**: A move by a player who is trying to maximize the score and thereby maximizing the quality of his position.
- **Min-node**: A move by the other player trying to minimize the score and thereby maximizing the quality of his position.

The Minimax algorithm is a recursive algorithm for choosing the next move. A score is associated with each possible position that can be obtained. The score value is computed by an *evaluation function* and it indicates how attractive it would be for a player to reach that position.

```
minimax(turn, position)
if (current.node = leaf.node)
  return position.evaluate
children = position.generate_legal_moves
  (turn = max)
  return max_value of minimax(min, children)
else
  return min_value of minimax(max, children)
```

Figure 4.2: Pseudo minimax algorithm.

The algorithm takes as parameters the player who is to move and the current board position. The algorithm starts by examining if the search has reached a leaf-node. If it has reached a leaf node then the leaf-position will be evaluated and a score is returned. If the search has not reached a leaf node then all the children of the current position will be generated. If it is Max's turn then the minimax-algorithm is called recursively for all the legal moves and the highest value if returned. On the other hand, if it is Min's turn then the minimaxalgorithm is called recursively for all the legal moves and the lowest value is returned. The final value returned to the root, is the value that delineates the best possible move for the player at the root.



Minimax is a strictly exponential algorithm. With a branching factor B and search to the depth **D**, the search traverses  $\boldsymbol{B}^{D}$  nodes.

#### 4.2.1 Negamax

The Minimax source-code leaves room for optimization. *Negamax* is an adjustment of Minimax where the score is calculated from the perspective of the active player deferring from Minimax where the score is found from the perspective of a particular player, i.e. either white or black. This simplifies the algorithm because, while, Minimax always has to check if the active node is the side who is trying to maximize or the side who is trying to minimize, Negamax only uses maximization operations. While the search algorithm is simpler the evaluation function must now handle which side has the move and negate the scores when MIN is active.

```
negamax(turn, position)
if (current.node = leafl.node)
   return position.evaluate(turn)
children = position.generate_legal_moves
return -(max_value of negamax(not turn, children))

Figure 4.4: Pseudo negamax algorithm.
```

The Negamax algorithm is used as basis for the different variants of the alphabeta search algorithm that are implemented in *Talvmenni*, which will be described in the following section.

# 4.3 Alpha-Beta Pruning

If the minimax algorithm would examine only the best move at every depth then the minimax value is found from a traversal of only a subset of the game tree which is called the **Minimal Game Tree**. The minimal game tree is the theoretical optimal solution tree and is the lower bound for the game search. The size of the minimal game tree was determined by Knuth & Moore [1975, cited by Schaeffer 1989:3]:

$$\mathbf{B}^{\text{ceil}(D/2)} + \mathbf{B}^{\text{floor}(D/2)} - 1$$

Where  $\bf B$  is the branching factor and  $\bf D$  is the depth of the search.

A closer examination of the minimax algorithm reveals that it is possible to enhance the basic technique of the algorithm, so we get closer to the minimal game tree. We can adapt minimax so it only explores down the branches from nodes where it possibly is fruitful. A technique that solves this problem is called **Branch and bounding** and the idea can be described as: "If you have an idea that is surely bad, don't take the time to see how truly awful it is. -Pat Winston". In other words, branch and bounding relies on the idea that it is possible to partition the search tree into sets using knowledge accumulated from the traversal of the tree, and ignore a set when we can determine that it cannot contain the optimal element.

The most popular refinement to the minimax search is *alpha-beta pruning*, which is a branch and bounding algorithm. The alpha-beta algorithm produces the same result as minimax, but at reduced cost because it traverses only a subset of the nodes of the game tree [Marsland et al 1986:3].

The Alpha-beta algorithm maintains two variables during search;

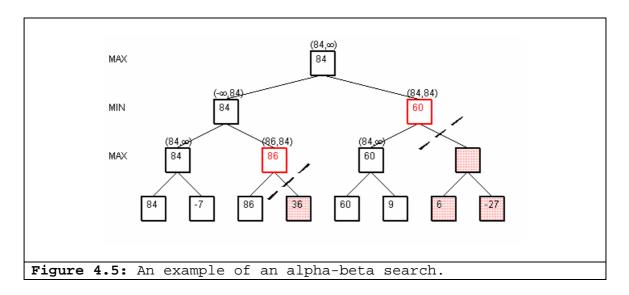
- Alpha as the lower bound for the best possible score that the player to move can achieve.
- Beta as the upper bound for the best score that the **opponent** can achieve.

Alpha has the following properties:

- Alpha never decreases and denotes the value of the best path encountered so far below a Max-node.
- When a child node is found with a higher beta than the value of Max's alpha, then alpha is set to the value of child's beta.
- Discontinue searching below a Max-node when there exists a Min ancestor of the node for which the ancestor's **alpha** >= Max's **beta**.

Beta has the following properties:

- Beta never increases and denotes the value of the best path encountered so far below a Min node.
- When a child node is found with a lower alpha than the value of Min's beta, then beta is set to the value of child's alpha.
- Discontinue searching below a Min node when there exists a Max ancestor for which the ancestor's **alpha** >= Min's **beta**.



The algorithm is heavily dependent on in which order the moves are searched. The best case, with optimal move ordering the alpha-beta algorithm searches the minimal game tree. The worst case is if there are no cut-offs and the algorithm searches all the same nodes as the minimax search.

Hence the performance of a plain Alpha-Beta search is unpredictable and must be further enhanced to be useful.

# 4.4 Alpha-Beta Enhancements

Since the Alpha-Beta algorithm was introduced in 1958 by Newell et al., several additional heuristics have been suggested to achieve further improvement [Reinefeld 1983].

# 4.4.1 Move Ordering

There is an exponential difference in the size of the search-tree between the best and worst case of an alpha-beta search. The deciding factor for the success of the alpha-beta pruning is the order in which the moves are being considered. [Schaeffer 2000] The sooner the best move is examined the larger effect of the alpha-beta cut-offs, hence a larger solution space can be searched in the same amount of time allocated resulting in playing chess at a higher level. Many ordering techniques have been developed for ordering moves in a best-to-worst order. A good move ordering scheme will reduce the branching factor at each node from 30-40 to about 7 or  $8^{15}$ . This gives a potential improvement to search almost twice as deep as a full search.

As described in section 3.4.1 when **Talvmenni** starts searching for a move to play, first, the transposition table is queried for the current position. If the response is negative then the following move ordering scheme is used:

- First the moves that are selected from the *killer heuristics* are searched.
- Second are the moves found in the *history heuristic* searched in decreasing order of their history values.
- Finally are all the remaining possible moves searched in a randommove order.

The ideas behind the killer heuristic and the history heuristic will be discussed next.

<sup>15</sup> http://www.chessbrain.net/beowulf/theory.html

# 4.4.2 History Heuristic and Killer Moves

The simplest method to improve the move ordering is to use a *static ordering* scheme that is based on the observations that certain classes of moves are most often the best moves. A class of "good moves" is *captures* of the opponent's pieces and another class is *checking* the opponent's king. Therefore it is logical to search these moves before all other "quiet" moves.

Two other more extensive approaches that have been proposed are based on the observation that different positions in a game of chess have many common resemblances. For each move that either player plays, only one or two pieces change position. Therefore a good move in one position is also likely to be a good move in many other positions in the same game.

#### • The Killer Move Heuristic

The Killer Move Heuristic maintains a small list of moves (1-3 moves) that during the search causes beta cut-offs. When the next move is searched, the list of "killer" replies is first examined before all other alternatives.

An example would be if white is to move and black threatens to check-mate white's king on the following move, then it does not make sense to search other white alternatives than those that save the king from check-mate. The Killer Move Heuristic is simple to implement and is at the same time claimed to improve the move ordering remarkably [Schaeffer 1989].

#### • The History Heuristic

The History Heuristic is a generalization of the killer move heuristic. Rather than storing specific moves, the History Heuristic stores from- and to-square every time a move causes a cut-off in the search by incrementing corresponding values in a hash table. The heuristic ignores which specific piece is moved and also ignores at what depth the move was found and maintains all the information in one table.

# 4.4.3 Iterative Deepening

The number of positions traversed by the alpha-beta algorithm grows exponentially with increasing search depth. It is difficult to judge in advance how much time a search to a certain depth will require. An interesting and relevant question is to what depth the game tree should be searched?

In competitive play, there is only a limited amount of time allocated for the whole game. Therefore, it is necessary to allocate only a certain amount of time per move searching the game tree. Nevertheless, if a search to a predefined depth is interrupted before finished, there is a great chance than the current best move is not a good move at all.

A solution to the problem was introduced with a technique called *Iterative Deepening*. This technique suggests that instead of searching from the root of the tree directly to a predefined depth it instead searches successively to increasing depths. When the allocated time is up, the search algorithm returns its current best move.

Iterative Deepening provides two main advantages:

- Even if only the result of the search at the last level is being used, the previous iterations have not been without use. *Talvmenni* maintains a list of legal moves for each position that it searches, which is sorted for every iteration in a best-to-worst order. The extra time that is spent searching previous iteration-levels is repaid as the performance of alpha-beta search is increased by the improved move-ordering.
- It is an excellent time control mechanism. Already after searching the tree at the first level, the search algorithm can return a reply as the best current move. This is useful in cases when there is not much time left on the clock.

# 4.5 Minimal Window Search

There is another class of enhancements with the same goal as the move ordering schemes: to limit the number of nodes to traverse, but with a different approach.

The interval enclosed by the two bounds alpha and beta is commonly referred to as a *search window* [Reinefeld 1983]. The basic alpha-beta algorithm is invoked with a maximum window ranging from minus infinite to plus infinite to ensure that the correct minimax value is returned. The family of *Aspirated Search* algorithms exploit the fact that if the alpha-beta search algorithm is invoked with a narrower search window, then it is more likely that more cutoff's will occur and fewer position needs to be examined.

# 4.5.1 Principal Variation Search

Principal Variation Search is one of the most popular refinements to the Alpha-Beta search algorithm. The basic idea is to search with a minimal window in as many nodes as possible and with a wider window only where it is necessary. When the correct minimax value does not lie within the range of the search window, the tree search is said to *fail* and must be started again with a wider search window. Every time that the search fails, a certain amount of search time is wasted. But then again, when the search window is set correctly a lot of search time is saved. This makes the order in which the moves are examined very important, as the sooner the good moves are examined, the faster the minimax value is found.

When the PVS is started, the first search is invoked with a normal window, while all subsequent searches are using an empty window (alpha, alpha+1) to test each successive move against the first move. The best move from previous searches is used first for each new level of depth, as this move is likely to be the best move.

An almost similar algorithm as PVS called <u>Negascout</u> was invented by Prof. Dr. Alexander Reinefeld in 1983. Negascout uses the same approach as PVS, but is a bit simpler implemented. Reinefeld suggests that Negascout traverses between 20 to 30 percent fewer leaf-positions than an alpha-beta search [Reinefeld 1983].

# 4.5.2 MTD(f)

MTD(f) is a relatively new search algorithm that is claimed to be more simple and efficient than previous algorithms [Platt 1996]. While NegaScout and PVS always are invoked with a wide window, MTD(f) is only invoked using a zero window. Therefore MTD(f) may require to search many times, but the returned bounds will enable it to rapidly converge towards the minimax value.

MTD(f) is initiated with 3 different parameters

- **root** is the current board position
- **f** is the first guess of the expected minimax value
- **d** is the search depth

```
function MTDF(root : node_type; f : integer; d : integer) : integer;

g := f;
upperbound := +INFINITY;
lowerbound := -INFINITY;
repeat

if g == lowerbound then
    beta := g + 1
else
    beta := g;

g := AlphaBetaWithMemory(root, beta - 1, beta, d);
if g < beta then
    upperbound := g
else
    lowerbound := g;
until lowerbound >= upperbound;
return g;
Figure 4.6: MTD(f).
```

MTD(f) is initiated around an arbitrary value. The first pass sets either a lower or an upper bound on the expected minimax value. Consecutive passes improves both bounds, until they equal each other which means that the correct minimax value is found.

The name of the algorithm MTD is for "Memory-enhanced Test Driver".

# **4.6 Horizon Effect**

Humans do still today occasionally beat the strongest chess programs. When human chess players are asked how they are able to win, the response is often "By deploying one of human's advantages over computers: Intuition".

The reason that computers lack the so-called intuition is a well-known problem of AI "**The Horizon Effect**", first described by Hans Berliner [Laramée 2000]. The look-ahead horizon for the strongest human chess players varies a lot in depth for the different branches. Some (few) selected variations are examined to a greater depth than even the strongest computers are capable of, while the calculation of some branches may fail at a very low depth, resulting in a blunder for even the strongest human grandmasters.

As described earlier, the program always searches to a predefined depth and thereby has a limited look-ahead horizon. For example, if the price for capturing the opponent's queen on ply 5 is that you are being checkmated on ply 6, and the program only searches to exactly ply 5 then it will decide that the capture of the opponent's queen is a good move.

Two techniques have been suggested as a solution to avoid these often catastrophic moves due to "short-sightedness": *Secondary Search* and *Quiescence Search*.

The basic idea behind the secondary search algorithm is to double-check a move before actually playing it.

#### • Secondary Search Algorithm

- 1. Find the best moves by searching to a predefined depth
- 2. Search the best move of the list some depths further
- 3. Verify that this move still is good, if not repeat step 2 for second best move.

The approach of secondary search was introduced by Greenblatt's famous chess program from 1967, MAC HACK SIX. Today most chess programs instead implement another alternative, which will be described in the following section.

#### 4.6.1 Quiescence Search

The basic idea behind the quiescence search algorithm is as follows [Laramée 2000]:

#### **Quiescence Search Algorithm**

- First, the program searches the tree down to a pre-defined depth, and secondly continues the search only for those branches that are defined as "unstable".
- When the position "quiets down" (hence the name: quiescence search), i.e. no unstable continuations are present, the program stops iterating and returns the pay-off value from the positional evaluation.

Which branches that are considered "unstable" varies a lot between different chess engines:

- Capture moves, especially those of major pieces, are most commonly included in quiescence search.
- Promoting a pawn changes the material balance drastically and is therefore relevant to examine further.
- A check of the opponent king can result in an mating attack, so checkmoves are often also included in quiescence search.

Extending the alpha-beta search algorithm with the quiescence search algorithm can increase the amount of time the programs spends searching drastically. It is important to carefully select which moves are included in quiescence search and in what order the moves are examined. The most used scheme for ordering of capture moves is called "Most valuable victim, least valuable aggressors" (MVV/LVA). The idea is to first examine the captures of the opponent's biggest pieces (Hence: most valuable victim). If many pieces are capable of capturing the opponent's big piece, the scheme suggests that it is best first to try to capture with the smallest piece (Hence: Least valuable aggressors).

# 4.7 The Null-Move Heuristic

Shannon [1950] was confident that it was necessary to find methods to prune away uninteresting branches of the game tree as early as possible, or even before traversal of the search-tree was initiated. This resembles the thinking-process of humans and would reduce the search-tree in an order of magnitude. The idea of *Forward Pruning* has not gained great success though, except for a technique called *The Null-Move Heuristic*. The risk of pruning away good moves is too big and it is very difficult to decide which branches are uninteresting before they have been searched.

#### **Null-move Pruning**

Shannon [1950] philosophized about the possibility that a player could make a pass move, i.e. skip a move and let the other player make two moves in a row. If the rules would allow this, then we could definite state that the player with the white pieces always would be able to hold a draw from the initial position assuming optimal play. The rules do not allow making pass moves, but it is of course, not possible to prevent the search algorithm from making a pass-move during search. The idea is that the search algorithm lets the opponent play two sequential moves, and if the program still has a better position, then it is not necessary to continue to search the branch further down, because the opponent does not seem to be able to improve his position anyway.

Null-Move Pruning can, as all other forwarding pruning techniques, fail badly. The problem especially occurs in *zugzwang* positions (German term meaning that a player is put at a disadvantage because he has to make a move). Null-move pruning does not work in zugzwang positions because the whole idea behind the method does not hold, that is: The position would deteriorate if the computer would make the move that is skipped.

Null-move Pruning is quite easily implemented in search algorithms, but makes a big performance difference, and can save between 20% and 75% of the effort required by a given search [Laramée 2000].

# 4.8 Chapter Summary

The invention of the Alpha-Beta algorithm was a huge advance for computer chess programs, and still today is implemented in almost every chess program. However, the Alpha-Beta algorithm has some drawbacks: Firstly, the program evaluates and quantifies all positions in just a single number. This is an oversimplification, as a board position contains many different properties, which are impossible to in a single numeric value. The have been made some efforts finding alternatives for Alpha-Beta to solve this issue, for instance Hans Berliner's B\* algorithm, which used two values for each position: An optimistic evaluation and a pessimistic evaluation.

In this chapter, we have examined the most popular sequential search algorithms used in modern chess programs. We have in *Talvmenni* developed a modular "strategy framework" which enables us to easily substitute and compare different search algorithms. Talvmenni supports at present the following search-algorithms: A random-move search algorithm, the Minimax algorithm, The Negamax algorithm, The Alpha-Beta algorithm and MTD(f). All these algorithms are optionally enhanced with Iterative Deepening, Quiescence Search, History Heuristic, Transposition Tables and the Null-Move heuristics.

# Chapte

# Distributed Search Strategy

In this chapter, we first look at some relevant performance issues for parallel search. Next, we discuss two different popular parallel algorithms. Thirdly, we examine JavaSpaces and its programming environment. Finally, we describe a new parallel algorithm that we introduce in *Talvmenni*.

# 5.1 Introduction

As described in section 1.3 there is a strong correlation between the strength of a chess program and the speed with which the program searches through the game tree. Increased speed in a chess program may be achieved by many different non-mutual exclusive means; one approach is *Parallelism* that increases the processing capacity available to the chess program, in which he main source of improvement comes from searching different parts of the game tree at the same time most often in an alpha-beta-like manner.

Networks of workstations are increasingly prevalent because of their advantage of high scalability and cost-effectiveness over parallel machines [Marsland 1996]. Scalability can be defined as how well a parallel system (parallel algorithm + parallel architecture) utilizes increased computing resources.

The alpha-beta algorithm has proved to be hard to parallelize efficiently on shared-memory systems [Brockington 1998]. On a distributed-memory system such as JavaSpaces, the alpha-beta algorithm is expected to be even harder to implement efficiently. Both parallel systems suffer from various sources of parallel inefficiencies, which will be the topic of the next sections.

# **5.2 Parallel Performance**

A measurement of how the parallel search algorithms improve the performance of the sequential algorithms is required. When measuring performance, it is important to understand exactly what is measured and how it is measured. The most used measurement for performance gains in parallel processing is *Speedup*, which is defined as the ratio of sequential running time to parallel running time.

S(n) = T(1) / T(n)

Formula 1: Speedup definition

If n processors results in a n-fold speedup, then it is said to be a *linear* speedup. There are always some differences in the implementation of a sequential search algorithm, and the corresponding parallel algorithm. When calculating the speedup it is important to measure the best possible performance of both the sequential and parallel algorithms.

Another related measurement is the notion of *speedup efficiency*, which is given as the ratio of the achieved speedup to the number of processors.

E(n) = S(n) / n

Formula 2: Definition of Speedup Efficiency

Speedup proportional to the number of processors is very difficult to achieve in practice. It is therefore interesting to measure how efficient the parallel system performs by dividing the achieved speedup with the number of processors used. Hence, "Speedup Efficiency" indicates how much performance increase each additional processor gains.

The different causes to an offset from peak performances in parallel search are called *parallel overheads* and will be examined in the following section.

5.2.1 Search Overhead

The main cause for the efficiency of most sequential search algorithms is that

the accumulated information that has been collected while traversing the

game-tree gives basis for pruning parts of the game-tree [Marsland et al.

1986]. Parallel search algorithms do not often have access to this information,

and in some instances will perform worse than sequential algorithms. In

general when extending sequential searching algorithm for use on a parallel

processing system, the reduction in search time is notoriously less than the

number of processors in the system [Marsland et al. 1985].

The alpha-beta pruning depends on fully searching branches in order to

establish an upper and a lower bound for the search of the next branches.

Parallelism comes from searching different parts of the search tree at the same

time. The upper and lower bound for the different branches obtained in

sequential search will therefore not be same for the same branches when

running in parallel search. The search window will in general be wider in

parallel search, and will therefore cause a parallel overhead.

Search overhead arises when the parallel search algorithm visits nodes that

the sequential algorithm would not visit [Feldman et al. 1991].

The search overhead is defined as [Lu 1993]:

SO(n) = N(n)/N(1) - 1

Formula 3: Search Overhead

That is the ratio of visited nodes when running in parallel search to visited

nodes when running in sequential search.

-76-

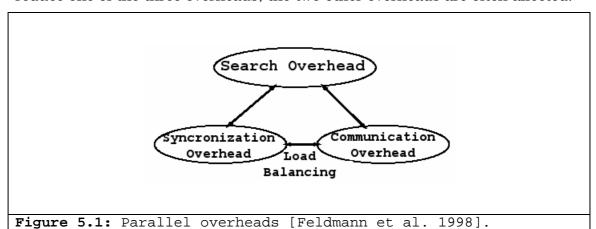
# **5.2.2 Synchronization Overhead**

The size of the different sub-trees that are splitted amongst the different parallel processes is not predictable in advance. This leads to *load balancing* problems, because those processes that examine small sub-trees terminate sooner than processes that traverse large sub-trees.

Work granularity is the arithmetic mean of the sizes of sub-trees explored by a process in relation to the size of the complete search-tree. Smaller work granularity implies that smaller tasks are divided amongst the processes. By lowering the work granularity the synchronization overhead decreases, but unfortunately the communication overhead also increases when a higher number of smaller tasks must be transferred amongst the processes.

#### 5.2.3 Communication Overhead

Communication overhead arises when messages are passed between processes through some medium that will always contain some latency ratio. It includes both the time it takes for the message to travel from one process to another process and the time that it takes to encode and decode the message. Parallel architectures with distributed memory suffer from a higher level of communication overhead. It is therefore important to divide the tasks in optimal sizes finding the best balance between communication and synchronization aspects. When modifying the parallel algorithm in order to reduce one of the three overheads, the two other overheads are often affected.



# **5.3 Parallel Algorithms**

In this chapter, we will examine two completely different parallel algorithms that have gained a lot of focus and research. YBWC is a highly synchronous algorithm while APHID is an asynchronous algorithm, and this gives them different features and properties. Both algorithms have drawbacks and advantages, as do all parallel algorithms because there exist no critical solution to how to parallelize game-tree search.

The study of these two parallel algorithms is the basis for the development of the *Talvmenni Object Space Concurrent Algorithm* (abb. *TOSCA*) described in section 5.4.

# **5.3.1 The Young Brother Wait Concept**

The first branch (furthest to the left) under all nodes is referred to as "the eldest brother" while all other brances are called "younger brothers".

#### The concept of Young Brothers Wait Algorithm:

"The eldest son of any node must be completely evaluated before younger brothers of that node may be transmitted."

[Feldmann 1993]

Section 4.2 described the concept of the minimal game tree. Knuth & Moore in their work also assigned different types to all the nodes of the minimal game tree:

- The root is of type 1.
- The first child of a type-1 node is also of type 1. The other children are of type 2.
- The first child of a type-2 node is of type 3.
- All children of a type-3 node are of type 2.

#### The YBW algorithm [Feldmann 1997]:

- The search below the leftmost child must be completed before parallel search below right children of a type-1 node is allowed.
- The parallel search below right children of a type-2 node is only allowed if the searches below all promising children are complete. Promising children are the leftmost ones, those reached by a move proposed by the transposition table, the killer heuristic, or any positive capturing move.
- The parallel search below a type-3 node may begin immediately after its generation.

#### Advantages and drawbacks of the YBW Algorithm:

#### • Reduces search overhead (advantage)

The YBW algorithm has low search overhead, because of the improved move ordering obtained within the game-tree. YBW performs a search in the same manner as a sequential alpha-beta algorithm, and if the move ordering is perfect then YBW searches exactly the same nodes as a sequential alpha-beta search.

#### Causes synchronization overhead (drawback)

Because the parallel search does not start until the search of the leftmost branch (principal variation) is completed, all other processes must wait idle.

#### • Possible large communication overhead (drawback)

In an environment with a high message latency ratio, e.g. a network of workstations, the high frequency of accesses to a global transposition table would cause a very large communication overhead.

#### **5.3.2 APHID**

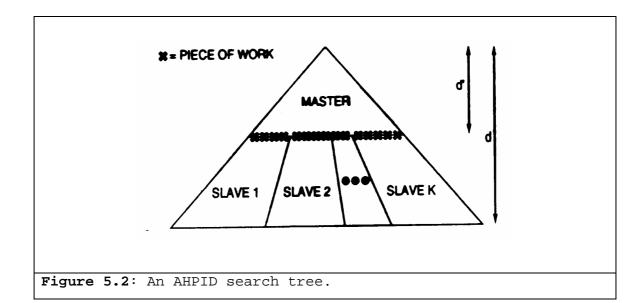
The Asynchronous Parallel Hierarchical Iterative Deepening algorithm is an extension of the ideas of Newborn's UIDPABS algorithm for use on a network of workstation. APHID has been implemented as a game-independent library with the purpose of easily extending existing sequential game-playing programs [Brockington 1996].

#### The concept of APHID:

APHID is an asynchronous algorithm; it does not synchronize at any node at any time during search. It is an hierarchical algorithm with a master controlling the top of the tree and slaves searching the rest of the tree.

#### The AHPID algorithm:

- The Master process searches a truncated game-tree to a pre-defined depth and allocates works for the slave processes.
- The Slave processes search their work with minimal synchronization with the master and determine their own work schedule [ibid.].
- The Slave processes return the discovered information to the master whom generates approximate minimax values, until all of the required score information is available.



#### Advantages and drawbacks of the APHID algorithm:

#### • No synchronization overhead (advantage)

When a slave is idle, it will continue to search the tree to the next level of the iterative depth. Each slave process will at any time have many different branches to choose from to search. The slave chooses to search the branch that it at the current point of time has searched to the lowest depth. This ensures that all processes at all time are kept busy.

#### Causes search overhead (drawback)

When load-balancing problems occur, then the master reassigns a task from one worker to another worker, and the new owner must recalculate the task from scratch, wasting the work done by the initial worker. This problem increases when there are many workers available, causing even more search overhead [Brockington 1998:141].

Additionally, the master participates in both searching the game-tree as well as delegating tasks to the workers. When there is much work to be done, the master easily becomes a bottleneck.

# **5.4 Jini and JavaSpaces**

Modern distributed systems have to cope with a network of heterogeneous members under different dynamic network conditions, and some of the many challenges they are faced with are: computers failing, network latency and members joining the network spontaneously.

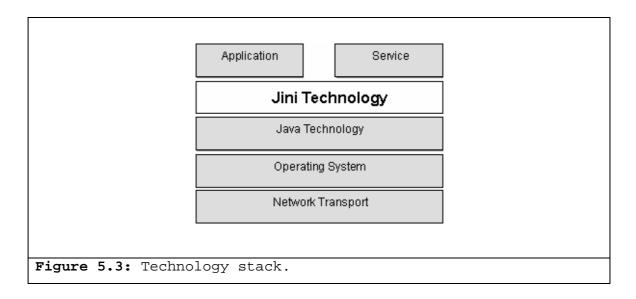
Sun Microsystems in 1999 released the Jini technology framework, which in many ways was the realisation of Sun's mantra – "the network is the computer". Jini supports computers and devices in dynamically sharing and exchanging services across a network. Jini also provides the programmer with a higher layer of abstraction, which masks the complexity of network topologies, dynamic network discovery and distributed network security. As a part of the Jini project, Sun also introduced *JavaSpaces*, which facilitates the creation of reliable and fault-tolerant distributed systems. JavaSpaces technology is a descendant of Dr. Gelernter's and the Yale Linda group's Coordination Language from 1982 [Freeman et al. 1998]. JavaSpaces is an object-oriented tuple space implementation that implements objects as tuples

within the space. The system is simple, which gives the programmer the

opportunity to concentrate and focus on the domain of the problem itself.

#### **5.4.1 The Jini Architecture**

The Jini technology is a set of APIs that define how to build adaptive distributed systems that run on top of the Java platform (figure 5.3).



A *Jini Federation* is a group of devices and software programs that form a single dynamic distributed system. The Jini federation consists of the following main components:

#### • A Client

All members of a Jini federation are called clients.

#### A Service

A service can be a software program or some hardware device that clients of the federation can use. Services export their services in form of java objects for clients to download and execute.

#### • A Lookup-Service

The lookup service, also known as the service locator, keeps track of the services offered in the federation. Clients contact the lookup service to find a particular service.

# 5.4.2 JavaSpaces

JavaSpaces are available in a Jini federation as services. A JavaSpace is a mechanism for loosely coupled processes to coordinate communication of objects in a distributed shared environment.

#### Features of JavaSpaces:

#### • Sharing and concurrency

Multiple processes can access and work concurrently on a JavaSpace.

#### • Persistence

An object remains in the space until it is explicitly removed and can be stored and retrieved from remote systems.

#### • Transactions

JavaSpaces can use the transaction-service provided by Jini. All operations on a JavaSpace are atomic. Multiple operations can be grouped in a single transaction object. If any step cannot be completed successfully a transaction can be "rolled back".

#### Leasing

JavaSpaces can use the lease-service provided by Jini. Entry in a JavaSpaces (tuple) can be given an expiry time after which it will automatically be removed from the JavaSpace. This provides a form of distributed garbage collection.

#### • Associative

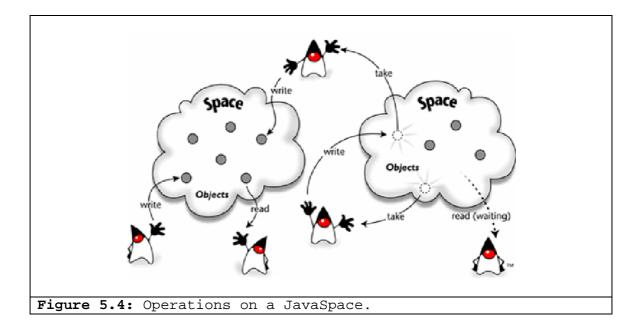
Contents of a JavaSpace are addressed (searched) by their contents using template matching, instead of a specific location identifier.

#### **JavaSpace Operations:**

JavaSpaces implements only four basic operations for all interaction (plus two of the same operations in predicate input form).

- **Write:** Puts an entry (object) into the space.
- **Read:** Return a matching entry from the space.
- **Take:** Remove a matching entry from the space.
- **Notify:** Send an event to a listener if matching entry is found.
- **ReadIfExists:** A non-blocking version of the Read-operation.
- **TakeIfExists:** A non-blocking version of the Take-operation.

Figure 5.4 shows a classical diagram of the Java mascot the Duke who represents clients that perform different operations on two JavaSpaces.



The Java RMI (Remote Method Invocation) protocol or the Jini Extensible Remote Invocation (JERI) provides network support, while the Hypertext Transfer Protocol (HTTP) provides distribution of classes to the clients.

Before *Talvmenni* can use JavaSpaces for parallel search, the following services must be up and running:

- A web (HTTP) server for distributing the classes to the clients.
- A RMI or JERI activation server supporting network communication.
- A Jini lookup service for locating available services.
- A Jini transaction manager handles transactions.
- A JavaSpaces server provides the required object space.

Many command line parameters are required to start the whole environment, which adds some complexity to using JavaSpaces.

# **5.4.3 The Replicated-Worker Pattern**

The replicated-worker pattern is one of the most commonly implemented patterns in applications also often referred to as the "master-worker pattern" [Freeman et al. 1999]. The idea behind the replicated-worker pattern is that computing a task consists of *n* independent identical subtasks. This method is best suited for simple "divide-and-conquer" problems, and is ideal for problems where the number of tasks changes during execution, e.g. search algorithms.

#### Replicated-Worker Algorithm for an Object Space

- 1. The Master creates task entries and writes them into the Object Space.
- 2. The Workers take a task entry, perform the computation, and write the result back into the Object Space.
- 3. The Master reads the Object Space concurrently with the work of the workers in step 2, in order to take possible returned results which are then used for the final result.

To indicate whether the replicated-worker pattern is appropriate a "Computation/Communication Ratio" can be calculated. A too low ratio suggests that a single-processor approach may be more efficient.

# **5.5 TOSCA**

In the following, we describe **TOSCA**, a scheme that we humbly suggest for parallel asynchronous iterative-deepening search in an object space. **TOSCA** is based on a *data partitioning scheme* using the *replicated-worker pattern* where different parts of the search tree are processed asynchronously by any available sequential worker running in the distributed object space environment. The approach has many resemblances to the APHID algorithm.

#### **TOSCA** consists of four phases:

- An *Initial Phase* where the master prepares for the next phase.
- In the *Factory Phase* the master creates chess engine tasks and writes them into the JavaSpace.
- Next in the *Processing Phase* the workers read the JavaSpace and download chess engine tasks which are processed and the result is written back to the JavaSpace.
- Finally, in the *Collect Phase*, the master collects the results from the workers and determines the best move.

The Processing Phase is asynchronous with both the Factory Phase and the Collect Phase, as the workers will process the tasks at the same time as the master either creates new tasks or collects results. Only at the Initial Phase the workers will have to wait briefly for the master while it is preparing for the Factory Phase.

**TOSCA** maintains special "alpha-beta" entries in JavaSpace for each depth of the tree that is to be searched to by iterative deepening. These global alphabeta boundaries are used by all tasks to improve the alpha-beta search performed in parallel by the workers.

# **5.5.1 Parallel Strategies**

**Talvmenni** implements a parallel search strategy as easily as any other sequential search strategy, as these parallel strategies extend the abstract class AbstractParallelStategy which also extends the abstract AbstractStrategy class.

When *Talvmenni* is initiated with a parallel strategy, this concrete parallel strategy (i.e. parallel iterative deepening alpha-beta search strategy) is responsible for creating one instance of the ChessEngineMaster and as many instances of ChessEngineWorker as required. The ChessEngineMaster instance is created in the same thread as the AbstractParallelStrategy while a separate thread is created for constructing ChessEngineWorker(s). *Talvmenni* continues to write new chess engine workers into space for as long as there are available members (generic workers) in the network which take down and execute the chess engine workers. Java supports mobile code, which JavaSpaces utilizes extensively, and thus the deployment of chess engine workers is done dynamically as the generic workers download the executable binaries. This ensures that members joining the network spontaneously during the game will be able to participate.

\_\_\_

**Talvmenni** implements at present a strategy for *Parallel Iterative Deepening Alpha-Beta Search with Transposition Tables*. In this strategy a concrete master is implemented which has search method which works as follows:

- 1. The master completes a small iterative deepening search, e.g. to 3-4 ply, to improve the move ordering.
- 2. The master generates tasks containing positions to be searched, which are put into the JavaSpace.
- 3. The workers take positions from the JavaSpace, which are searched and then put the results into the JavaSpace.
- 4. The master collects the results for each additional ply to the master's search in step 1 that the workers have processed.
- 5. For each ply the master chooses the best move.

The ChessEngineWorker is a specialized worker and extends Task from the generic replicated worker pattern. When a chess engine worker is downloaded and executed on a generic worker, it creates a new local transposition table and local history heuristic which is maintained during all the life-time of the worker and spans the execution of tasks.

The workers execute a blocking-take on the JavaSpace until at task is found and downloaded. If many tasks are present in the JavaSpace, the task with the highest priority is downloaded and executed.

When a task is executed, it first examines the JavaSpace for an alpha-beta entry for the current search depth. If the alpha-beta values in the JavaSpace are better than the alpha-beta value for the task then the task's alpha-beta values are updated before the position is searched. After the search is completed the task again examines the alpha-beta entry in JavaSpace, and if the search has improved the task's alpha-beta values, so they are better than the values in space, then the alpha-beta entry is updated in JavaSpace.

# **5.6 Chapter Summary**

JavaSpaces provide a simple and flexible environment for developing parallel algorithms. The development of **TOSCA** is at an early stage as the current scheme is a rather naive parallel algorithm. Developing against JavaSpaces makes it easy to change and gives an opportunity to experiment with different methods and algorithms. The current version of **TOSCA** should, at least, be improved on the following aspects:

#### • Termination of tasks:

The master should be able to terminate tasks that have been delegated to workers, if the master concludes that it already has received a good enough move (e.g. forced by time constraints). This could be solved by employing the concept of "poison pills" [Freeman et. al 1999].

#### Load Balancing:

"The key to successful large-scale parallelism lies in the dynamic distribution of work" [Kopec et al. 1996]. A good load balancing scheme is vital to any parallel algorithm. The Replicated-Worker pattern provides reasonable load balancing if the execution-time of the tasks are of equal size, and the number of tasks fits to the number of workers. This is not the case in searching game sub-trees, and **TOSCA** should therefore be able to adapt the workload granularity to the number of participating workers.

#### Workers failing:

Distributed systems are inherently unreliable, and **TOSCA** should be able to deal with instances such as high network latency and computers failing. This could be solved elegantly by employing Jini transactions on the tasks with relatively short leases.



# **Tests & Experiments**

This chapter is divided in two parts: In section 6.1 we discuss in short, the methodology used for our tests and experiments. In section 6.2 we present the results for the different tests for the sequential version of *Talvmenni* that we have performed, and in section 6.3 we describe the results from testing *Talvmenni* running in parallel utilizing JavaSpaces.

# **6.1 Methodology**

The quality and correctness of **Talvmenni** has been measured by

#### • Unit Tests

JUnit is a regression testing framework written by Erich Gamma and Kent Beck. We have during the development of *Talvmenni* created 1353 unit tests to verify the correctness of vital parts of the source-code. The accompanying CD-ROM contains all written unit-tests.

#### • Performing Perft Tests

The concept of Perft Tests was described in chapter 2.3.2. The main purposes of Perft Tests are firstly to examine if the Move Generation Component generates a correct list of legal moves for all positions abiding the rules of the game.

The strength of Talvmenni as a chess-playing program has been measured by

#### • Playing against humans on the internet

The explosive growth of the internet during the last decade has made the game of chess more popular than ever. Both humans and chess programs from all over the world meet on different servers that offer the services of hosting game-playing activities. The most popular chess-server is the Internet Chess Club<sup>16</sup> where *Talvmenni* became a member in October 2004.

### • Playing against other chess programs

The easiest method to test a chess programs is to set it up to play a match or tournament against other chess programs. Various freely available programs allow you to run a series of unattended matches between many chess engines.

The <u>potential speedup</u> of parallelizing search with JavaSpaces has been measured by

#### • Timing performance on a selected set of test positions

We selected some different positions and registered the time *Talvmenni* required to find the best move in the position. This test was repeated for a different number of available workers and for different search depths. The accompanying CD-ROM contains the collected data from the tests.

#### 6.1.1 Hardware

Hardware used for all sequential tests and the master process in parallel tests:

Intel Pentium 3 - M CPU 1.00GHz with 512 MB of RAM.
 Operating system: Microsoft Windows 2000 Service Pack 4

Hardware used for all slaves:

A network of workstations ranging from Pentium 3 – CPU 500MHz with
 256 MB of RAM to Pentium 4 – CPU 2.00 GHz with 512 MB of RAM.

All machines were connected through a 100 Mbit/s switched Ethernet.

-

<sup>16</sup> http://www.chessclub.com

# **6.2 Test Results**

In this section we present the results from the Perft Tests and play against humans and other chess programs.

#### **6.2.1 Test 1: Perft Tests**

We performed four Perft tests; first from the initial position and then from various positions that are better suited for Peft testing because they are more complicated and contained more potential elements such as stalemate and promotion<sup>17</sup>.

#### Test position no. 1

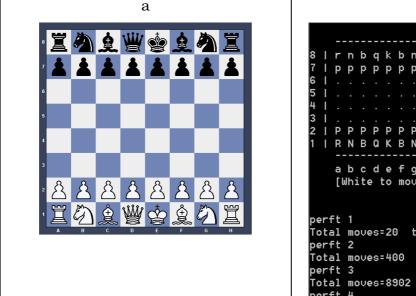


Figure 6.1: (a) Test position no. 1 (b) Screendump of the perft results returned from *Talvmenni*.

\_

<sup>17</sup> http://homepages.caverock.net.nz/~peter/perft.htm

**Talvmenni** calculated the perft values down to 6 plies for test position no. 1 (figure 6.1) and we compared the results with already verified results (figure 6.2).

Depth	1	2	3	4	5	6	7	
Verified values	20	400	8902	197281	4865609	119060324	3195901860	
Talvmenni values	20	400	8902	197281	4865609	119060324	?	
Figure 6.2: A comparison of Perft values for test position no. 1								

Test position no. 2

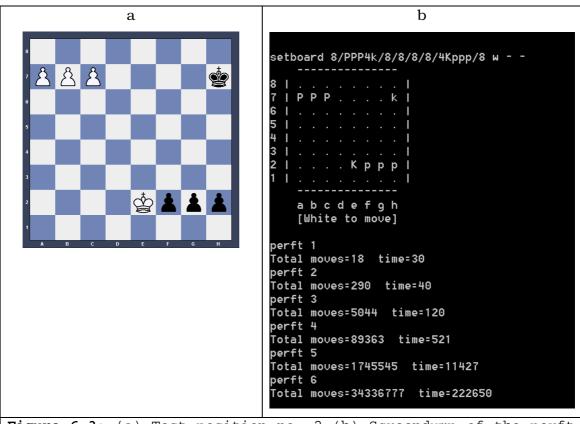


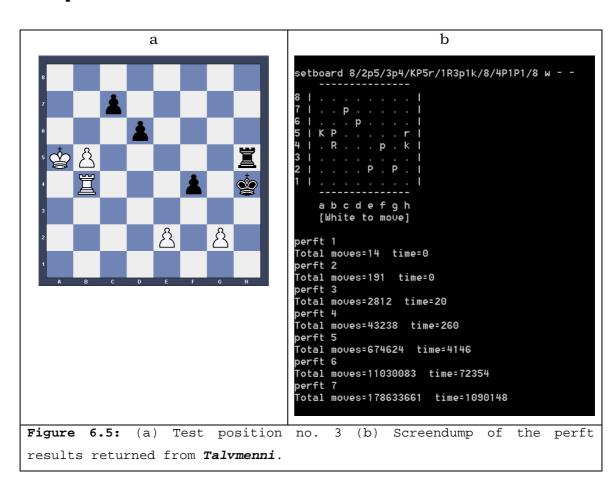
Figure 6.3: (a) Test position no. 2 (b) Screendump of the perft results returned from *Talvmenni*.

**Talvmenni** calculated the perft values down to 6 plies for test position no. 2 (figure 6.3) and we compared the results with already verified results (figure 6.4).

Depth	1	2	3	4	5	6	7
Verified values	18	290	5044	89363	1745545	34336777	749660761
Talvmenni values	18	290	5044	89363	1745545	34336777	?

Figure 6.4: A comparison of Perft values for test position no. 2

#### Test position no. 3



**Talvmenni** calculated the perft values down to 7 plies for test position no. 3 (figure 6.5) and we compared the results with already verified results (figure 6.6).

Depth	1	2	3	4	5	6	7
Verified values	14	191	2812	43238	674624	11030083	178633661
Talvmenni values	14	191	2812	43238	674624	11030083	178633661

Figure 6.6: A comparison of Perft values for test position no. 3

# **6.2.2 Test 2: Play against Humans**

We enrolled *Talvmenni* as a member of the Internet Chess Club in October 2004 and *Talvmenni* has to date played approx. 200 games against human opponents. An advantage of playing against humans is that human opponents vary a lot in style and strength. This ensures that *Talvmenni* is tested in a much wider range of positions than seen in games against other computers.

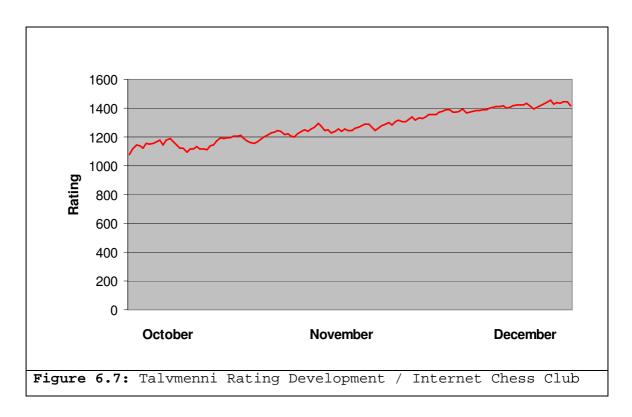


Figure 6.7 shows *Talvmenni*'s rating development in the time period from the beginning of October till midst December. During the month of October *Talvmenni* had a rating that averaged on 1150 rating-points. In the end of October "Quiescence Search" (described in chapter 4.6.1) was implemented which increased the playing strength approx. 100-200 points.

The "Evaluation Function" (described in chapter 3.2) was improved by adjusting some positional elements in the end of November, which gave an additional increase by approx. 100 rating-points.

Today *Talvmenni* averages on 1450 rating-points, which corresponds to the strength of an average human club player.

# **6.2.3 Test 3: Play against Computers**

We selected seven free chess programs of increasing strength to play a tournament with *Talvmenni* participating of 28 games. The software 'Arena' by Martin Blume<sup>18</sup> was used as the interface for handling the tournament.

The participants were:

- Name: LaMoSca ver. 0.10 Author: Pietro Valocchi, Italy Rating: 1055. Programming language: C
- Name: Geko ver. 0.43 Author: Giuseppe Cannella, Italy Rating: 1136. Programming language: Pascal
- Name: MurderHole ver. 1.0.10 Author: Eric Oldre, USA Rating: 1254. Programming language: VB.Net
- Name: Raffaela ver. 0.11 Author: Stefano Gemma, Italy Rating: 1332. Programming language: Assembly
- Name: MinichessAI ver. 1.19 Author: Marcin Gardyjan, Poland Rating: 1422. Programming language: C/C++
- Name: Yawce ver. 0.16 Author: Jakob Sandholm, Denmark Rating: 1511. Programming language: ?
- Name: Chessterfield ver. i5a Author: Matthias Luscher, Denmark Rating: 1618. Programming language: C/C++

The ratings of these chess programs are taken from ChessWar<sup>19</sup> which is an internet site that arranges tournaments for chess programs.

<sup>18</sup> http://www.playwitharena.com/

<sup>19</sup> http://loirechecs.chez.tiscali.fr/chesswar/

Rank	Engine	Score		Mu	Ya	Ra	Mi	Ta	Ge	La	S-B
1	ChessterfieldCL	7,0/7		1	1	1	1	1	1	1	21,00
2	Murderhole	5,0/7	0		1	0	1	1	1	1	12,50
3	Yawce016	5,0/7	0	0		1	1	1	1	1	11,00
4	Raffaela	3,5/7	0	1	0		0	1	=	1	8,25
5	Minichessai	3,5/7	0	0	0	1		=	1	1	6,25
6	Talvmenni	2,5/7	0	0	0	0	=		1	1	3,25
7	Geko_043	1,5/7	0	0	0	=	0	0		1	1,75
8	LaMoSca	0,0/7	0	0	0	0	0	0	0		0,00

28 games played / Tournament finished Tournament start: 2004.09.15, 16:11:18 Latest update: 2004.12.07, 03:07:50 Site/ Country: EYDUN, Faeroe Islands

Level: Blitz 5/5

Hardware: Mobile Intel(R) Pentium(R) 4 - M CPU 2.00GHz

Operating system: Microsoft Windows XP Build 2600 Service Pack 1

PGN-File: <u>Arena tournament 5.pqn</u> Table created with: <u>Arena 1.08</u>

Figure 6.8: Tournament Table of the Test Tournament

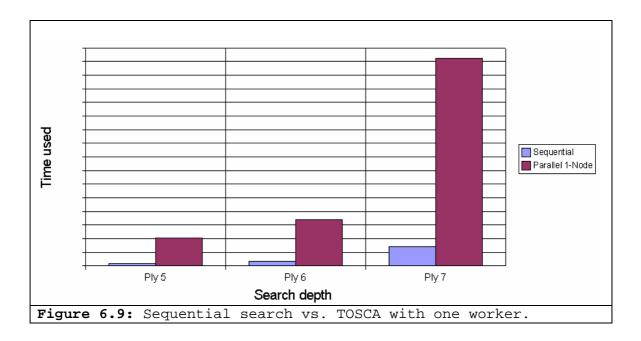
**Talvmenni** finished in 6<sup>th</sup> place winning two games, drawing a single game and losing 5 games. Appendix B contains the games played by *Talvmenni* in the tournament.

#### **6.2.4 Further Remarks**

The results of the performed Perft tests indicate clearly that the move generator implemented in *Talvmenni* generates a correct list of legal moves that abides by the rules of the game. The games that *Talvmenni* has played on the internet and against other computers support this suggestion. *Talvmenni* at present does not implement the 50-move rule and the draw by repetition rule. This is not required to able to play a legal game of chess, as the GUI-interface is responsible for detecting these states. However, the consequence of this deficiency is that *Talvmenni* never claims a draw in repeated positions. Neither does *Talvmenni* try to avoid repeating identical positions, which often results in games ending in a draw even though *Talvmenni* is in a winning position.

# **6.3** Parallel Speedup

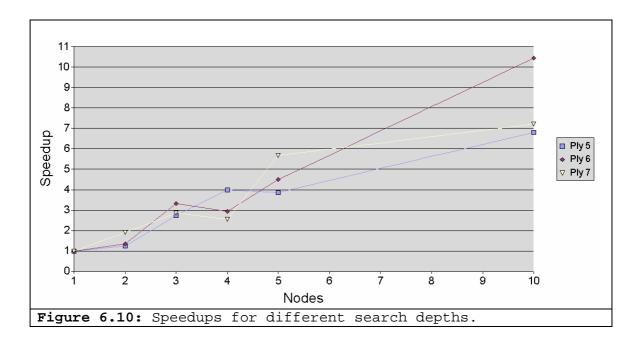
We performed 10 different experiments for *Talvmenni* performing sequential search and calculated the average time used for searching to different depths. The results were compared to similar experiments done for *Talvmenni* performing parallel search (*TOSCA*) with one worker (figure 6.9).



The results show a parallel overhead of a factor of approx. 10 for sequential search compared to parallel search with one worker. The parallel overhead seems to decrease for greater search depths.

We believe that this overhead can be attributed mainly to **communication overhead**, as the two search-algorithms for sequential search and parallel-search are almost similar. The big difference was that the sequential version was executed in a single process, while the parallel version was executed running on a LAN where the master was executed on one machine, while the worker was executed on another machine and all communication was directed through the JavaSpace, which was running on the first machine.

Next, we performed 38 different experiments and calculated the average speedup gained for a variable number of workers searching to different depths. The tests were first performed with 1 to 5 workers (nodes) and finally with 10 workers.



The results fluctuate somewhat, but show a clear speedup for each additional node. The results seem to indicate that the speedup increases when searching to greater depths, which corresponds to results obtained from other similar experiments, e.g. [Feldmann 1993:92].

The fluctuations can be attributed to two main sources:

- The workload granularity in TOSCA does not adapt to the number of workers.
- There are many potential sources for uncertainties when performing tests on network of heterogeneous computers, some computers perform much worse than others, and might become a bottleneck if they receive one of the final tasks to process.

# Chapter

# **Conclusions**

In this chapter, conclusions drawn from this research and some recommendations for future work are presented.

The work presented in this thesis has two main parts: the review and development of a chess-playing program, and the testing of JavaSpaces as an environment for pallelizing the game-tree search of the developed chessengine.

In this thesis we have identified and reviewed the main elements of a
modern chess-playing program and provided solutions to some of the
key-issues that we encountered in the process of developing our chessengine, *Talvmenni*.

The experience gained from developing a chess-playing program has resulted in the following observation: We do not agree with the common perception amongst the computer science community that Java is not suitable as a platform for a chess-playing program, because of its slow nature compared to high-performance languages such as assembler and C. The experience we have gained from this research has provided us with the insight that rather than focusing on the number of CPU-cycles, to focus on developing correct and efficient search-algorithms. Today (2004) Java executes a constant factor of approx. two-three times slower than C. But this factor is insignificant compared to the potential gain from a more efficient and expressive search-algorithm, which can be more easily implemented with an object-oriented and type-safe language.

In the time-period from February 2004 to November 2004, we have built a fully-fledged chess-playing program, which we have decided to release as an Open-Source project at: <a href="www.sourceforge.net/projects/talvmenni">www.sourceforge.net/projects/talvmenni</a>. The aim during the development has not been to optimize the program to maximum playing strength, but instead to use the development-process as a stepping stone for build knowledge around the domain of computer chess programming.

### Some features of **Talvmenni** at present are:

- o Alpha-beta Search
- Iterative Deepening
- Bitboards
- o Transposition Tables
- o Null-move Forward Pruning
- o History and Killer Heuristics
- o Quiescence Search
- o Static Positional Evaluation
- o Opening Book
- o Parallel version for Search (TOSCA)
- o Control with a Console or the Winboard-interface

• *Talvmenni* is probably the first Faroese made chess-program ever, but around the world maybe over a thousand chess-programs exists. However, utilizing JavaSpaces for performing parallel processing of the developed chess-program which is the second aim of this thesis is a novel approach, and has to our knowledge, not been tried before.

We propose in this thesis a parallel algorithm, **TOSCA**, for utilizing JavaSpaces as a distributed processing environment. We perform a number of experiments for testing the suitability of JavaSpaces as responsible for parallelizing the search. The results from the performed experiments are partly encouraging. The tests indicate that JavaSpaces contain a high ratio of communication overhead, but do scale very well with additional computer resources. An interesting future experiment would be to examine **TOSCA**'s ability to scale on a much higher number of workers than we had available.

The main benefits of Jini/JavaSpaces can be summarized as an extremely flexible programming model, and a runtime environment which easily can be deployed onto a network of workstations. The main disadvantage of JavaSpaces running on a network of workstations is the high ratio of communication overhead. Therefore, an idea for future research would be to take advantage of the environment by other means than to parallelize the search-function, and instead parallelize the evaluation-function. This could be done by splitting the evaluation function between many workers, where each worker would only evaluate a specific aspect of the position. This would not require the same level of communication between the space and the worker as parallelizing the search. And since the entire game-tree would be searched by each worker locally, there would be no loss of internal knowledge in the inherently sequential search function.

# **A - List of Abbreviations**

AHPID Asynchronous Parallel Hierarchical Iterative Deepening

**COW's** Cluster of Workstations

**CVS** Concurrent Version System

**EGTB** Endgame Tablebases

FEN Forsyth-Edwards Notation
GUI Graphical User Interface

**HTTP** Hypertext Transfer Protocol

**JERI** Jini Extensible Remote Invocation

MTD Memory-enhanced Test Driver

**MVV/LVA** Most Valuable Victim/Least Valuable Attacker

**NOW's** Network of Workstations

**ICC** Internet Chess Club

**IDE** Integrated Development Environment

PGN Portable Game Notation

PVS Principal Variation Search

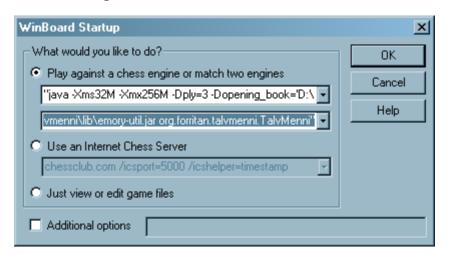
RMI Remote Method Invocation

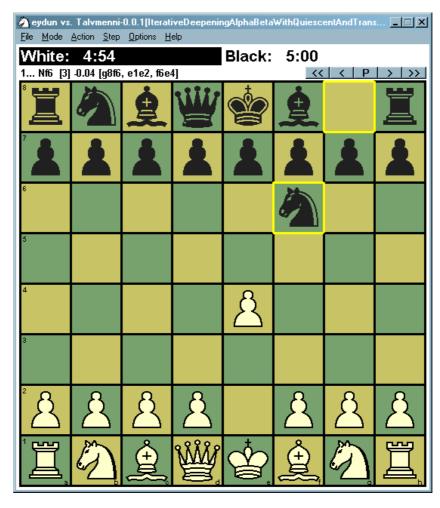
SOAP Simple Object Access Protocol

**TOSCA** Talvmenni Object Space Concurrent Algorithm

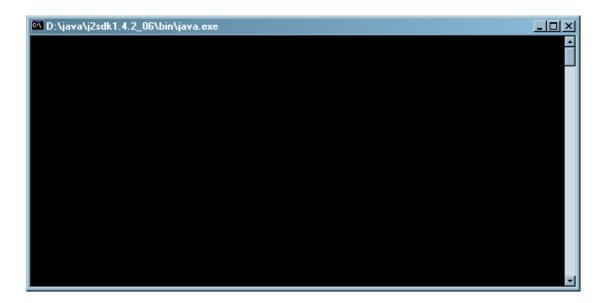
**YBWC** Young Brother Wait Concept

**Talvmenni** running with the Winboard-interface:





**Talvmenni** running in console-mode:



When *Talvmenni* first starts, the console is completely blank:

Next, commands are available that will activate one of the two currently supported UI-protocols: xboard which is the protocol used by Xboard/WinBoard GUI's and cmd for the command line console.

When entering 'cmd' the console starts up and we get a welcome-screen:

```
Total Scholin Scholin
```

To get started we can write 'help' to see which commands are available:

```
D:\java\j2sdk1.4.2_06\bin\java.exe

For help on usage, type in HELP and press enter. -

(Type in quit and press enter to quit -

List of commands

List of commands

HELP: This help screen

POSITION: Display current position

HISTORY: Show the movelist

POSSIBLE: Display list of allowed moves

NEW: Start a new game

MOUE: Make a move. (Format: 'MOUE fftt')

?: Force computer to move now

FEN: Show the position in FEN-notation

SETBOARD: Setup a position. (Format: SETBOARD 'FEN-string'

WHITE: Give white the move

QUIT: Quit and leave the program
```

Entering 'new' starts a new game:

Screenshots for parallel version of Talvmenni with two workers.

First the JavaSpaces and Transaction Manager Service are started:

```
EXEC:\WINNT\system32\cmd.exe - D:\java\jini2_0_1\example\scripts\wrun.bat jmmp-transient
\[ \] \[ \] \] \[ \] \[ \] \] \[ \] \[ \] \[ \] \[ \] \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \] \[ \
```

and also the 'class'-serving http-server:

```
Dec 17, 2004 8:20:44 PM com.sun.jini.tool.ClassServer run
INFO: ClassServer started ID:\java\jini2_0_1\lib\, port 80801
reggie-dl.jar requested from latitude1:1120
reggie-dl.jar probed from latitude1:1121
mahalo-dl.jar requested from latitude1:1123
mahalo-dl.jar probed from latitude1:1123
mahalo-dl.jar requested from latitude1:1128
fiddler-dl.jar requested from latitude1:1129
fiddler-dl.jar requested from latitude1:1131
fiddler-dl.jar requested from latitude1:1133
norm-dl.jar requested from latitude1:1136
norm-dl.jar requested from latitude1:1141
mercury-dl.jar requested from latitude1:1141
mercury-dl.jar requested from latitude1:1142
mercury-dl.jar requested from latitude1:1146
mercury-dl.jar requested from latitude1:1148
outrigger-dl.jar requested from latitude1:1152
outrigger-dl.jar requested from latitude1:1153
-
```

Starting two workers:

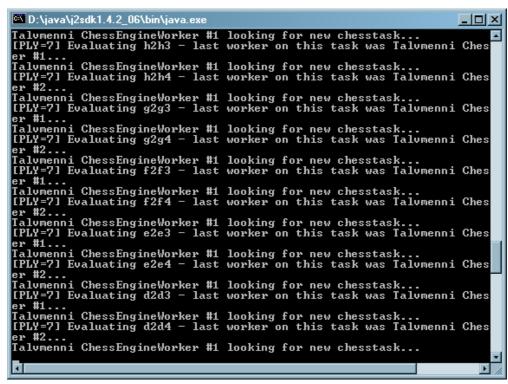
After starting the *Talvmenni* application:

Each worker take down specialized chess engine workers:

Talvmenni generates task, which are written to space:

```
D:\java\j2sdk1.4.2_06\bin\java.exe
                                                                                                                                                                                                                                                                               _ U ×
         Welcome to TalvMenni console -
        For help on usage, type in HELP and press enter. -
        (Type in quit and press enter to quit -
 new
Setting up a new game...
                r n b q
p p p p
                                              k
p
                                                      b
p
                                                               n r
p p
                PPPPPPP
RNBQKBNR
                 a b c d e f g h
[White to move]
Writing task for move: g1h3
Writing task for move: g1f3
Writing task for move: b1c3
Writing task for move: b1a3
Writing task for move: h2h3
Writing task for move: h2h4
Writing task for move: g2g3
Writing task for move: g2g4
Writing task for move: f2f3
Writing task for move: f2f3
Writing task for move: e2e3
Writing task for move: e2e3
Writing task for move: e2e4
Writing task for move: d2d3
Writing task for move: d2d3
Writing task for move: c2c3
Writing task for move: c2c3
Writing task for move: c2c4
Writing task for move: d2d4
Starting to collect 20 results at 5 ply
```

Workers take the tasks, process them and write the results back to space:



```
Swapping task...

[PLY=61 Evaluating bia3 - last worker on this task was Talumenni Cheser #2...

Setting local ply to: 5

[Ply=61 New alpha: 8

Swapping task...

[PLY=61 Evaluating h2h4 - last worker on this task was Talumenni Cheser #2...

Setting local ply to: 5

[Ply=61 New alpha: 8

Swapping task...

[PLY=61 Evaluating g2g4 - last worker on this task was Talumenni Cheser #2...

Setting local ply to: 5

[Ply=61 New alpha: 8

Swapping task...

[PLY=61 Evaluating f2f4 - last worker on this task was Talumenni Cheser #2...

Setting local ply to: 5

[Ply=61 New alpha: 8

Swapping task...

[PLY=61 Evaluating e2e4 - last worker on this task was Talumenni Cheser #2...

Setting local ply to: 5

[Ply=61 New alpha: 8

Swapping task...

[PLY=61 Evaluating d2d4 - last worker on this task was Talumenni Cheser #2...

Setting local ply to: 5

[Ply=61 New alpha: 8

Swapping task...

[PLY=61 Evaluating d2d4 - last worker on this task was Talumenni Cheser #2...

Setting local ply to: 5

[Ply=61 New alpha: 8

Swapping task...

[PLY=61 Evaluating c2c4 - last worker on this task was Talumenni Cheser #2...

Setting local ply to: 5

[Ply=61 New alpha: 8

Swapping task...

[PLY=61 Evaluating c2c4 - last worker on this task was Talumenni Cheser #2...

Setting local ply to: 5

[Ply=61 New alpha: 8

Swapping task...

Setting local ply to: 5

[Ply=61 New alpha: 8

Talumenni ChessEngineWorker #2 looking for new chesstask...
```

**Talvmenni** collects the results from the space and chooses the best move to make:

```
Starting to collect 20 results at 5 ply IPLY=51 got a result: e2e4 is scored with 0 starting to collect 20 results at 5 ply IPLY=51 got a result: d2d3 is scored with 0 starting to collect 20 results at 5 ply IPLY=51 got a result: d2d4 is scored with 0 starting to collect 20 results at 5 ply IPLY=51 got a result: d2d4 is scored with 0 starting to collect 20 results at 5 ply IPLY=51 got a result: c2e3 is scored with 0 starting to collect 20 results at 5 ply IPLY=51 got a result: c2e4 is scored with 0 starting to collect 20 results at 5 ply IPLY=51 got a result: b2b3 is scored with 0 starting to collect 20 results at 5 ply IPLY=51 got a result: b2b3 is scored with 0 starting to collect 20 results at 5 ply IPLY=51 got a result: b2b3 is scored with 0 starting to collect 20 results at 5 ply IPLY=61 got a result: b2b3 is scored with 0 starting to collect 20 results at 6 ply IPLY=61 Got a result: b2b3 is scored with 0 starting to collect 20 results at 6 ply IPLY=61 Got a result: a2a3 is scored with 0 starting to collect 20 results at 6 ply IPLY=61 Got a result: a2a4 is scored with 4 starting to collect 20 results at 6 ply IPLY=61 Got a result: g1b3 is scored with 4 starting to collect 20 results at 6 ply IPLY=61 Got a result: g1b3 is scored with 4 starting to collect 20 results at 6 ply IPLY=61 Got a result: g1b3 is scored with 4 starting to collect 20 results at 6 ply IPLY=61 Got a result: g1b3 is scored with 4 starting to collect 20 results at 6 ply IPLY=61 Got a result: b1c3 is scored with 4 starting to collect 20 results at 6 ply IPLY=61 Got a result: b1c3 is scored with 4 starting to collect 20 results at 6 ply IPLY=61 Got a result: b1c3 is scored with 4 starting to collect 20 results at 6 ply IPLY=61 Got a result: b1c3 is scored with 4 starting to collect 20 results at 6 ply IPLY=61 Got a result: b1c3 is scored with 4 starting to collect 20 results at 6 ply IPLY=61 Got a result: c2c4 is scored with 4 starting to collect 20 results at 6 ply IPLY=61 Got a result: c2c4 is scored with 4 starting to collect 20 results
                  D:\java\j2sdk1.4.2_06\bin\java.exe
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          r n b q k b n r
p p p p p p p
                                                                                 . . . . . . . . N
P P P P P P
R N B Q K B
                                                                                                                                                                                                                                                                                                 Ė
                                                                                      a b c d e f g h
[Black to move]
```

# **C – Games from Test Tournament**

Talvmenni - Murderhole [B23/09] 06.12.2004

 $1.e2-e4\ c7-c5\ 2.Nb1-c3\ d7-d6\ 3.f2-f4\ Nb8-c6\ 4.Ng1-f3\ g7-g6\ 5.d2-d4\ Bf8-g7\ 6.Bc1-e3\ c5xd4\ 7.Be3xd4\ e7-e5\ 8.f4xe5\ d6xe5\ 9.Bd4-e3\ Qd8xd1+\ 10.Ke1xd1\ Ng8-e7\ 11.Kd1-c1\ 0-0\ 12.Nc3-d5\ f7-f5\ 13.Nd5xe7+\ Nc6xe7\ 14.Bf1-c4+\ Kg8-h8\ 15.Be3-c5\ Rf8-e8\ 16.Bc4-f7\ Re8-f8\ 17.Bf7-d5\ Rf8-e8\ 18.Bd5-f7\ Re8-f8\ 19.Bf7-d5\ f5xe4\ 20.Bc5xe7\ Rf8-e8\ 21.Be7-d6\ e4xf3\ 22.Bd5xf3\ e5-e4\ 23.Bf3-e2\ Bc8-f5\ 24.g2-g4\ Bg7-h6+\ 25.Kc1-b1\ Bf5-e6\ 26.Be2-b5\ Re8-d8\ 27.Bd6-e7\ Rd8-d2\ 28.Be7-b4\ Rd2-g2\ 29.Bb5-f1\ Rg2-f2\ 30.Bb4-c5\ e4-e3\ 31.Bf1-h3\ Be6-d5\ 32.Rh1-d1\ Rf2xh2\ 33.a2-a4\ e3-e2\ 34.Bc5-d4+\ Kh8-g8\ 35.Rd1-e1\ Rh2xh3\ 36.Bd4-f6\ Bh6-d2\ 37.c2-c4\ Bd2xe1\ 38.c4xd5\ Be1-a5\ 39.Kb1-c2\ e2-e1R\ 40.Ra1xe1\ Ba5xe1\ 41.g4-g5\ Be1-h4\ 42.a4-a5\ Ra8-c8+\ 43.Kc2-b1\ Re8-c5\ 44.d5-d6\ Bh4xg5\ 45.Bf6xg5\ Rc5xg5\ 46.Kb1-c2\ Rg5xa5\ 47.b2-b4\ Ra5-a2+\ 48.Kc2-b1\ Ra2-f2\ 49.Kb1-c1\ Rh3-h1#\ 0-1$ 

Minichessai - Talvmenni [E20] 06.12.2004

 $1.d2-d4\ Ng8-f6\ 2.c2-c4\ e7-e6\ 3.Nb1-c3\ Bf8-b4\ 4.Ng1-f3\ c7-c5\ 5.g2-g3\ 0-0\ 6.a2-a3\ Bb4xc3+7.b2xc3\ c5xd4\ 8.c3xd4\ Nb8-c6\ 9.h2-h4\ Nf6-e4\ 10.d4-d5\ e6xd5\ 11.c4xd5\ Qd8-a5+12.Bc1-d2\ Qa5xd5\ 13.Bd2-b4\ Nc6xb4\ 14.a3xb4\ Qd5-c4\ 15.Qd1-d4\ b7-b5\ 16.e2-e3\ Qc4-c3+17.Qd4xc3\ Ne4xc3\ 18.Nf3-d4\ Bc8-b7\ 19.Rh1-g1\ Bb7-d5\ 20.Nd4xb5\ Nc3xb5\ 21.Bf1xb5\ Rf8-b8\ 22.Bb5xd7\ Rb8xb4\ 23.f2-f4\ a7-a5\ 24.g3-g4\ a5-a4\ 25.h4-h5\ a4-a3\ 26.g4-g5\ a3-a2\ 27.Bd7-h3\ Ra8-e8\ 28.Ke1-d2\ Rb4-b2+29.Kd2-c1\ Rb2-b4\ 30.Rg1-g3\ Re8-b8\ 31.Rg3-g1\ Rb4-c4+32.Kc1-d2\ Rb8-b2+33.Kd2-d3\ Rc4-e4\ 34.Bh3-g2\ Re4-a4\ 35.Bg2xd5\ Ra4-a3+36.Kd3-d4\ Rb2-d2+37.Kd4-e4\ Ra3-a7\ 38.Ke4-e5\ Ra7-e7+39.Ke5-d6\ Re7-e6+40.Kd6-c5\ Rd2-c2+41.Kc5-b5\ Rc2-b2+42.Kb5-c5\ Rb2-c2+43.Kc5-b5\ Rc2-b2+44.Kb5-c5\ Rb2-c2+\frac{1}{2}-\frac{1}{2}$ 

Talvmenni - LaMoSca [A10/08] 07.12.2004

 $1.c2-c4\ f7-f5\ 2.Nb1-c3\ c7-c5\ 3.e2-e4\ f5xe4\ 4.Qd1-h5+\ g7-g6\ 5.Qh5-e5\ Ng8-f6\ 6.Nc3xe4\ Nb8-c6\ 7.Ne4-d6\#1-0$ 

Geko 043 - Talvmenni [D00] 07.12.2004

 $1.e2-e3\ Ng8-f6\ 2.d2-d4\ d7-d5\ 3.Bf1-b5+Bc8-d7\ 4.a2-a4\ c7-c6\ 5.Bb5-d3\ c6-c5\ 6.d4xc5\ Qd8-a5+7.Nb1-c3\ Ke8-d8\ 8.Bd3-c4\ d5xc4\ 9.b2-b4\ c4xb3\ 10.Bc1-b2\ b3xc2\ 11.Qd1-d3\ Kd8-c7\ 12.Ke1-d2\ Qa5xc5\ 13.Kd2xc2\ Bd7-f5\ 14.e3-e4\ Bf5xe4\ 15.Bb2-a3\ Be4xd3+\ 16.Kc2xd3\ Qc5xf2\ 17.Nc3-b5+Kc7-c6\ 18.Ra1-c1+Kc6-d7\ 19.Nb5-c7\ Qf2xg2\ 20.Nc7xa8\ Qg2xh1\ 21.Ba3-b2\ Qh1-e4+22.Kd3-d2\ Qe4-d5+23.Kd2-e1\ Qd5-e4+24.Ke1-f2\ Qe4-f4+25.Ng1-f3\ Nf6-e4+26.Kf2-e2\ Nb8-c6\ 27.Rc1-c4\ Ne4-g3+28.Ke2-d3\ Qf4xf3+29.Kd3-d2\ Ng3-f1+30.Kd2-e1\ Qf3-e3+31.Ke1xf1\ Qe3-d3+32.Kf1-f2\ Qd3xc4\ 33.Kf2-g1\ Qc4-b4\ 34.Bb2-a1\ Qb4-e1+35.Kg1-g2\ Qe1xa1\ 36.a4-a5\ Qa1xa5\ 37.Kg2-g3\ Qa5-e5+38.Kg3-f2\ Qe5xh2+39.Kf2-e3\ e7-e5\ 40.Ke3-e4\ f7-f5+41.Ke4-f3\ Nc6-d4+42.Kf3-e3\ Qh2-e2\#\ 0-1$ 

Talvmenni - Chessterfield<br/>CL [D21] 07.12.2004

 $1.d2-d4\ d7-d5\ 2.c2-c4\ d5xc4\ 3.Ng1-f3\ b7-b5\ 4.Ke1-d2\ e7-e5\ 5.Kd2-c3\ c7-c5\ 6.Nf3xe5\ c5xd4+\ 7.Kc3-c2\ Bc8-f5+\ 8.Ne5-d3\ c4xd3+\ 9.e2xd3\ Qd8-c8+\ 10.Kc2-d2\ Bf8-b4+\ 11.Kd2-e2\ Qc8-e6+\ 12.Ke2-f3\ Bf5-g4+\ 13.Kf3-g3\ Bg4xd1\ 14.h2-h3\ 0-1$ 

Yawce016 - Talvmenni [A05] 07.12.2004

1. Ng1-f3 Ng8-f6 2. e2-e3 Nf6-e4 3. d2-d3 Ne4-f6 4. Bf1-e2 d7-d5 5.0-0 Nb8-c6 6. Nb1-d2 e7-e5 7. c2-c4 Bc8-g4 8. c4xd5 Nf6xd5 9. Nf3xe5 Bg4xe2 10. Ne5xc6 Be2xd1 11. Nc6xd8 Ke8xd8 12. Rf1xd1 Kd8-c8 13. Nd2-f3 f7-f5 14. e3-e4 Nd5-e7 15. Bc1-g5 f5xe4 16. d3xe4 Ne7-c6 17. h2-h4 Bf8-d6 18. a2-a4 Rh8-e8 19. Nf3-d2 h7-h6 20. Bg5-e3 Bd6-e5 21. Ra1-b1 Be5-f6 22. h4-h5 Re8-e5 23. g2-g4 Re5-a5 24. b2-b3 Ra5-e5 25. b3-b4 b7-b6 26. b4-b5 Nc6-e7 27. f2-f4 Re5-e6 28. f4-f5 Re6-d6 29. Nd2-c4 Rd6xd1+ 30. Rb1xd1 c7-c5 31. Be3-f2 Bf6-c3 32. e4-e5 Kc8-c7 33. e5-e6 Ra8-d8 34. Bf2-g3+ Kc7-c8 35. Nc4-d6+ Rd8xd6 36. Rd1xd6 Bc3-f6 37. Rd6-d7 Bf6-d4+ 38. Kg1-g2 Ne7-g8 39. Rd7-f7 a7-a5 40. b5xa6 Bd4-e5 41. Bg3xe5 Kc8-d8 42. a6-a7 g7-g6 43. a7-a8Q# 1-0

Talvmenni - Raffaela [C40] 07.12.2004

 $1.e2-e4\ e7-e5\ 2.Ng1-f3\ Qd8-f6\ 3.Bf1-d3\ d7-d5\ 4.e4xd5\ e5-e4\ 5.Bd3xe4\ Qf6-d6\ 6.Qd1-e2\ Ke8-d8\ 7.Nf3-g5\ Ng8-h6\ 8.Ng5xh7\ Bf8-e7\ 9.Nb1-a3\ f7-f5\ 10.Na3-c4\ Qd6-a6\ 11.Be4-f3\ Rh8xh7\ 12.0-0\ Nh6-f7\ 13.Rf1-e1\ Qa6-f6\ 14.d2-d4\ Nf7-g5\ 15.Qe2xe7+\ Qf6xe7\ 16.Re1xe7\ Ng5xf3+\ 17.g2xf3\ Kd8xe7\ 18.a2-a4\ c7-c6\ 19.d5-d6+Ke7-e6\ 20.b2-b3\ Ke6-d5\ 21.Bc1-e3\ b7-b6\ 22.f3-f4\ Rh7-h6\ 23.Kg1-f1\ Rh6xh2\ 24.Kf1-g1\ Rh2-h6\ 25.c2-c3\ Rh6-g6+\ 26.Kg1-h1\ Rg6-h6+\ 27.Kh1-g2\ Kd5-e4\ 28.f2-f3+Ke4-d3\ 29.Ra1-h1\ Rh6xh1\ 30.Kg2xh1\ b6-b5\ 31.a4xb5\ c6xb5\ 32.Be3-f2\ b5xc4\ 33.b3xc4\ Kd3xc4\ 34.Bf2-h4\ Nb8-d7\ 35.Kh1-g2\ a7-a5\ 36.Kg2-h3\ a5-a4\ 37.Kh3-g2\ a4-a3\ 38.Bh4-f6\ g7xf6\ 39.d4-d5\ a3-a2\ 40.Kg2-h3\ a2-a1Q\ 41.Kh3-g2\ Ra8-a2+\ 42.Kg2-h3\ Qa1-b2\ 43.Kh3-g3\ Qb2-h2\#\ 0-1$ 

# **D** – **Bit-wise Operations**

Operator	Meaning
&	Bitwise AND
I	Bitwise OR
^	Bitwise XOR
<<	Left shift
>>	Right shift
>>>	Zero fill right shift
~	Bitwise complement
<<=	Left shift assignment $(x = x \le y)$
>>=	Right shift assignment $(x = x >> y)$
>>>=	Zero fill right shift assignment $(x = x >>> y)$
x&=y	AND assignment $(x = x \& y)$
X = y	OR assignment $(x = x \mid y)$
x^=y	XOR assignment $(x = x ^ y)$

# **E - Resources CD**

The attached CD contains:

- This thesis (pdf and ps)
- Source-code and Unit Tests + Javadoc. API
- *Talvmenni* running with Winboard
- **Talvmenni** executable (console mode)
- Test data spreadsheet

# References

### [Brockington 1996]

Brockington, M. G.

A Taxonomy of Parallel Game-Tree Search Algorithms.

ICCA Journal, 19. (3), pp. 162-174. (1996)

### [Brockington 1998]

Brockington, M.G. Asynchronous Parallel Game-Tree Search.

Phd Thesis. University of Alberta, Canada. (1998)

### [Ciancarini 1994a]

Ciancarini, P.

Experiments in Distributing and Coordinating Knowledge.

ICCA Journal, Volume 17: Number 3. (1994)

### [Ciancarini 1994b]

Ciancarini, P. A Comparison of Parallel Search Algorithms based on Tree Splitting.

Technical Report UBLCS 94-14, Dipartimento di Scienze dell'Informazione, Universit`a di Bologna, Italy. (1994)

### [Donskoy et al. 1989]

Donskoy M. & Schaeffer, J. Perspectives on Falling From Grace. New Directions in Game-Tree Search, pp. 85-93. (1989)

### [Eppstein 1997]

http://www1.ics.uci.edu/~eppstein/180a/w99.html (Accessed at 01.07.2004)

### [Feldmann 1991]

Feldmann, R. A Fully Distributed Chess Program.

Advances in Computer Chess 6. (1991)

### [Feldmann 1993]

Feldmann, R. Game Tree Search on Massively Parallel Systems Phd Thesis. University of Paderborn, Germany. (1993)

### [Feldmann 1997]

Feldmann, R.

Algorithms and Heuristics for a Deep Look into the Future

Theory and Practice of Informatics, 1-18 (1997)

### [Feldmann et al. 1998]

Feldmann R. and Monien B..

Selective Game Tree Search on a Cray T3E

University of Paderborn, Germany. (1998)

### [FIDE 1997]

FIDE, Laws Of Chess.

http://www.fide.com/official/handbook.asp?level=EE101. (1997)

### [Hyatt 1999]

Hyatt, R. Rotated Bitmaps.

ICCA Journal 22, 4 (Dec 1999), pp. 213-222. (1999)

### [Kervinck 2002]

van Kervinck, M.N.J. The Design and Implementation of the Rookie 2.0 Chess Playing Program

Master Thesis, University of Eindhoven, Holland. (2002)

### [Kishimoto 2002]

Kishimoto, A.

Transposition Table Driven Scheduling for Two-Player Games

Master Thesis, University of Alberta, Canada. (2002)

### [Kopec et al. 1996]

Kopec, D. & Marsland, T.A.. **Search** Methods in Artificial Intelligens. (1996)

### [Laramée 2000]

Laramée, F.D. Chess Programming. (2000)

http://www.gamedev.net/reference/programming/features/chess1/(Accessed at 01.07.2004)

### [Leiserson et al. 1998]

Leiserson, C.E., Prokop H. and Randall, K.H.

Using de Bruijn Sequences to Index a 1in a Computer Word.

MIT Laboratory for Computer Science, Cambridge. (1998)

### [Lu 1993]

Lu, C.P.P. Parallel Search of Narrow Game Trees.

University of Alberta, Canada. (1993)

### [Marsland et al. 1985]

Marsland, T.A & Popovich, F. Parallel Game Tree Search.

University of Alberta, Canada. (1985)

### [Marsland et al. 1986]

Marsland, T.A & Campell, M.

Parallel Search of Strongly Ordered Game Trees.

University of Alberta, Canada. (1986)

### [Marsland 1986]

Marsland, T.A. A Review of Game-tree Pruning.

University of Alberta. (1986)

### [Marsland 1991]

Marsland, T.A. Computer Chess and Search.

University of Alberta. (1991)

### [Marsland et al. 2000]

Marsland, T.A. & Bjørnsson, Y. From Minimax to Manhattan. University of Alberta. (2000)

### [Neumann et al. 1944]

von Neumann, J. **Theory of Games and Economic Behavior**. Princeton University Press, Princeton. (1944)

### [Nimzowitsch 1925]

Nimzowitsch, A. **My System**. Hernovs forlag. (1925)

### [Platt 1996]

Platt, A. Research Re: search & Re-search.
PhD thesis. Erasmus University, The Netherlands. (1996)

### [Rasmussen 2004]

Rasmussen, D.R. Parallel Chess Searching and Bitboards. Master Thesis, Technical University of Denmark. (2004)

### [Reinefeld 1983]

Reinefeld, A.

An Improvement of the Scout Tree-Search Algorithm. ICCA Journal 6, 4 (Dec 1983), pp. 4-14. (1983)

### [Schaffer 1989]

Schaeffer, J. The History Heuristic and Alpha-Beta Search Enhancements in Practice.
University of Alberta. (1989)

### [Schaffer 2000]

Schaeffer, J. The Games Computers (and People) Play. University of Alberta. (2000)

### [Schaffer et al. 2002]

Hlynka M., Jussila, V., Schaeffer, J.

Temporal Difference Learning Applied to a High-Performance Game-Playing Program.

University of Alberta. (2002)

### [Seirawan 2003]

Yasser Seirawan. Winning Chess Strategies. Everyman Chess (2003)

### [Shannon 1950]

Shannon, C. E. **Programming a Computer for Playing Chess**. Philosophical Magazine 41 (1950)

### [Simon et al. 1992]

Simon, H. and Schaeffer J. **The Game of Chess**. Carnegie-Mellon University. (1992)

### [Tanenbaum 2001]

Tanenbaum, A.S. Modern Operating System.

Prentice Hall, Inc. (2001)

### [Wall 2004]

Wall, B. Computer Chess History.

http://www.geocities.com/SiliconValley/Lab/7378/comphis.htm (Accessed at 18.07.2004)

### [Warren 2003]

 $Warren\ Jr.,\ H.S.\ Hacker's\ Delight.$ 

Addison Wesley (2003)