THE HATFIELD POLYTECHNIC

School of Information Sciences

BACHELOR OF SCIENCE IN COMPUTER
SCIENCE

Project Report

Investigation of Tactical and Positional Evaluation in Computer Chess

Mark J. Uniacke

May 1990

Abstract

This report examines and describes various techniques currently used to program a computer to play chess.

New ways of improving both positional and tactical ability are investigated.

An evaluation function is developed after analysing how strong chess players select moves. The evaluation function criteria are formally defined and cover both positional and tactical aspects.

Various search extension heuristics are investigated and a method of encorporating these selective extensions into an aspiration $\alpha\beta$ search is developed.

The concept of uncertainty is encorporated into the evaluation function and a close connection between static evaluation and search decision making is shown to be effective.

The ideas developed are combined into a chess program encorporating many state of the art techniques. The program is shown to perform at the expert level of ability both tactically and positionally.

The trade off between knowledge and search is examined and conclusions about the effectiveness of reducing search effort by increasing knowledge content are drawn.

Acknowledgements

I would like to thank my project supervisor, Neil Davey, for the many interesting discussions throughout the year and Piet Tutelaers, for designing chess fonts which can be used in LaTeX documents.

Contents

1 Introduction **7**
- 1.1 A Brief History 8
- 1.2 My Background 10
- 1.3 Aims of the Project 10

2 Basic Overview **11**
- 2.1 Machine Representation 11
- 2.2 Move Generation 13
- 2.3 Evaluation Functions 14
- 2.4 Look Ahead 15
- 2.5 Selective Searching 17
- 2.6 Full Width Searching 19
- 2.7 Quiescence 20
- 2.8 Horizon Effect 20

3 Advanced Topics **22**
- 3.1 Tree Searching 22
  - 3.1.1 Alpha Beta Search Algorithm 22
  - 3.1.2 Aspiration $\alpha\beta$ Search 25
  - 3.1.3 Principle Variation $\alpha\beta$ Search 25
  - 3.1.4 Iterative Deepening 28
  - 3.1.5 Move Ordering 28
  - 3.1.6 Null Moves 29
  - 3.1.7 Razoring 31
- 3.2 Evaluation 33
  - 3.2.1 Piece Square Tables 33
  - 3.2.2 Static Evaluation 34

3.2.3 Incremental Evaluation 35
3.2.4 Phases of Play 36
3.2.5 Planning 37
3.3 Learning 37
3.3.1 Rote Learning 38
3.3.2 Advice Taking 38
3.3.3 Induction 39
3.3.4 Learning by Analogy 39
3.3.5 Parameter Adjustment 39
3.4 Time Control 40

4 The Evaluation Function 42
4.1 Definition 1 : Material 44
4.2 Definition 2 : Game Phase 46
4.3 General Positional Evaluation 47
4.3.1 Definition 3 : Mobility 47
4.3.2 Definition 4 : Development 49
4.3.3 Definition 5 : Centre Control 50
4.3.4 Definition 6 : Square Control 52
4.4 Pawn Evaluation 53
4.4.1 Definition 7 : Doubled Pawns 53
4.4.2 Definition 8 : Isolated Pawns 54
4.4.3 Definition 9 : Weak Pawns 54
4.4.4 Definition 10 : Passed Pawns 56
4.4.5 Definition 11 : Pawn Table 58
4.4.6 Definition 12 : Pawn Value 58
4.5 Knight Evaluation 60
4.5.1 Definition 13 : Knight Value 60
4.6 Bishop Evaluation 61
4.6.1 Definition 14 : Bishop Attack 61
4.6.2 Definition 15 : Bishop on Kings Diagonal 61
4.6.3 Definition 16 : Bishop Pair 62
4.6.4 Definition 17 : Horrwitz Bishops 62
4.6.5 Definition 18 : Enemy Pawn Obstruction 63
4.6.6 Definition 19 : Bishop Table 64
4.6.7 Definition 20 : Bishop Value 64
4.7 Rook Evaluation 66
4.7.1 Definition 21 : Rook–King Closeness 66

    - 4.7.2 Definition 22 : Connected Rooks 67
    - 4.7.3 Definition 23 : Rook on Open File 68
    - 4.7.4 Definition 24 : Rook on Seventh Rank 68
    - 4.7.5 Definition 25 : Rook Value 69
  - 4.8 Queen Evaluation 70
    - 4.8.1 Definition 26 : Queen–King Closeness 70
    - 4.8.2 Definition 27 : Queen Power 71
    - 4.8.3 Definition 28 : Queen Value 71
  - 4.9 King Evaluation 72
    - 4.9.1 Definition 29 : King On Open Files 72
    - 4.9.2 Definition 30 : Weak Squares around King . . 73
    - 4.9.3 Definition 31 : Mating Net 75
    - 4.9.4 Definition 32 : Check Threats 76
    - 4.9.5 Definition 33 : King Table 77
    - 4.9.6 Definition 34 : King Value 77
  - 4.10 Static Tactical Evaluation 79
    - 4.10.1 Definition 35 : Pinned Pieces 79
    - 4.10.2 Definition 36 : Attacked Pieces 81
  - 4.11 Planning 82
    - 4.11.1 Definition 37 : Pressure on King 83
    - 4.11.2 Definition 38 : Plan Selection 84
    - 4.11.3 Definition 39 : Pawn Square Table 86
    - 4.11.4 Definition 40 : Knight Square Table 88
    - 4.11.5 Definition 41 : Bishop Square Table 89
    - 4.11.6 Definition 42 : King Square Table 90
    - 4.11.7 Definition 43 : Square Control Table 92
  - 4.12 Definition 44 : Total Evaluation Function Value . . . 94

5 Tree Searching **96**
  - 5.1 Main Search 97
  - 5.2 Search Extensions 99
    - 5.2.1 Threats 100
    - 5.2.2 Threat Execution 103
  - 5.3 Quiescence Decision 106
  - 5.4 Quiescence Search 107

6 HIARCS 5.0 **110**
  - 6.1 Program History and Environment 110

6.2 Program Structure 111
6.3 Data Structures 113
    6.3.1 Position Representation 114
    6.3.2 Game Tree 115
    6.3.3 Transposition Table 115
    6.3.4 Refutation Table 118
    6.3.5 The History Heuristic 119
    6.3.6 Incremental Attack Tables 119
    6.3.7 Game History 121
6.4 The Algorithms 122
    6.4.1 Move Generation 122
    6.4.2 Move Ordering 124
    6.4.3 Tree Searching 127
    6.4.4 Evaluation 131
    6.4.5 Time Allocation and Control 133
6.5 Efficiency Aspects 139
    6.5.1 Algorithms 139
    6.5.2 Data Structures 139
    6.5.3 Code Production 140

7 Program Performance **141**
7.1 Bratko-Kopec Tests 142
7.2 Computer Opposition 146
7.3 Human Opposition 152
7.4 Knowledge versus Search 158
7.5 Internal Statistics 160
7.6 Summary of Performance 162

8 Conclusion **164**
8.1 Review 164
8.2 Achievements 166
8.3 Implications and Future Prospects 168

A Glossary **175**

B Pseudo Functions **176**

C Time Allocation Graphs **202**

D Bratko-Kopec Test Positions 208

E Test Games Played 221
- E.1 HIARCS 5.0 versus Mephisto Exclusive 222
  - E.1.1 Game 1 222
  - E.1.2 Game 2 223
  - E.1.3 Game 3 224
  - E.1.4 Game 4 225
  - E.1.5 Game 5 226
  - E.1.6 Game 6 227
- E.2 HIARCS 5.0 versus Mark Uniacke 228
  - E.2.1 Game 1 228
  - E.2.2 Game 2 229
- E.3 HIARCS 5.0 versus NOKNOW 231
  - E.3.1 Game 1 231
  - E.3.2 Game 2 233
  - E.3.3 Game 3 234
  - E.3.4 Game 4 235

Chapter 1

Introduction

Chess is an extraordinary game. The rules are clearly defined and quite simple to learn, yet the game is so complex it has challenged mankind for centuries and is still as rich and varied as ever. Such an enormous amount of material has been written about the game that there are more books written about chess than all other games put together. The game appeals to all ages and abilities from beginner to expert.

> "Chess is a sea in which a gnat may drink or an elephant may bathe."
>
> *Indian proverb.*

Chess epitomises many of the problems facing the field of artificial intelligence, that of the storage, representation, retrieval and application of knowledge; searching infinite domains and problem solving using inexact information. Chess therefore provides an almost perfect environment in which to experiment with new ideas. Unlike most other problem domains, progress can be accurately measured, see [19]. Ideas can easily be put to the test, both against human and computer opposition in competition; and in specially designed experiments. For these reasons, chess is an excellent research environment in which to investigate human intellectual skill and artificial intelligence. In fact, winning the human world chess championship was defined as one of the fundamental aims of artificial intelligence. It is a testimony to the problem's complexity that after more than forty years of concerted effort this aim is still far from being realised.

1.1 A Brief History

On March 9th 1949, Claude E Shannon, a researcher at Bell Telephone Laboratories, presented a paper titled "Programming a Computer for Playing Chess" [53]. This was the first paper of its kind and even today most computer chess programs owe much to Shannon's original ideas. Essentially, Shannon put forward three different types of program:

1. Type **A**: A program that does a full width search to a given depth and uses a simple terminal evaluation function.
2. Type **B**: A program that does a selective search of the most promising lines.
3. Type **C**: A program that is goal oriented, although little detail about this approach was given.

Alan Turing built further on some of these ideas and in 1950 wrote a paper about his work on computer chess at Manchester University [60]. However, Shannon's ideas were not actually incorporated into a chess playing program until 1956 when a Shannon type A program was developed at the Los Alamos Scientific Laboratory[1]. In 1957, Bernstein and Roberts wrote a type B program for the IBM 704 [9]. Most notably Newell, Simon and Shaw wrote a Shannon type C program in 1958 [42]. This was the first ever practical application of the alpha-beta algorithm. A number of programs followed, but all played very poor chess. Despite this, many researchers in the field of artificial intelligence were very confident about the potential of their machines.

> "Within 10 years a digital computer will be the world's chess champion, unless rules bar it from competition"
>
> *Herbert Simon, 1957.*

It was not until 1966 that real progress was made by Richard Greenblatt at MIT [23]. His program MacHack-6, was the first program which could be considered to play "real chess". It played in various human tournaments and established performance ratings in the 1250 to 1640 USCF range. This is quite an achievement, particularly when one considers it was in 1966.

In 1968, various AI researchers, encouraged by Greenblatt's progress made a 2500 dollar bet with David Levy, an International Master, that a computer would be able to beat him in a match within 10 years. Encouraged by such activities various scientists from the artificial intelligence field got involved and in 1970 the first United States Computer Chess Championship was held. This event has continued to be held annually to the present day.

---

[1] only developed for a 6x6 board with no bishops.

In 1974, the first World Computer Chess Championship was held in Stockholm. This event was won by a Russian program called Kaissa, essentially a Shannon type A program with numerous enhancements. Kaissa's rating was somewhere in the 1500 to 1700 ELO range; impressive for a program, but very low in world chess terms. This compares with a typical 2500 ELO Grandmaster rating, even David Levy was rated at about 2350 ELO.

In 1977, real progress was made when Chess 4.5 [4], an American program written at the Northwestern University, competed in the Minnesota Open Chess Championship and won the event with a 5-1 score. This success was very significant because several Experts and Class "A" players competed. It was not achieved as one might expect by a breakthrough in chess algorithms, but rather through the use of one of the latest "super-computers" - a CD Cyber 170. Inevitably, Chess 4.6 later won the triennial World Computer Chess Championship with a perfect 4-0 score.

In September 1978, Chess 4.7 challenged David Levy for the bet made 10 years earlier, the program did well, but lost by a score of $3\frac{1}{2}$-$1\frac{1}{2}$.

Programs based on a Shannon type A approach with little knowledge, but very fast searching had been dominating "intelligent" selective searchers for some 5 years. In 1978, using this principle of fast speed with little knowledge, special chess hardware was designed and built by Ken Thompson and Joe Condon [14]. The program, Belle, from Bell Laboratories, running on this special hardware ended up winning the 1980 World Computer Chess Championship.

By 1983, the strength of deep brute force searchers was further emphasised when the program Cray Blitz [31], running on a 2 processor Cray XMP/48 computer, won the 4th World Computer Chess Championship. The following year an exhibition match was arranged between Cray Blitz and David Levy. In the match, David Levy crushed the machine 4-0. This clearly indicated that although the machine was of master strength in the realms of tactics, its positional and strategic play was weak. Cray Blitz retained its world title in 1986.

Currently there are 2 programs of international master strength[1], Hitech and Deep Thought. Both Hitech and Deep Thought run on special purpose hardware and essentially perform sophisticated brute force searches. Deep Thought is clearly the strongest of these programs with an official rating of 2551 USCF which equates to approximately 2450 ELO. During May 1989, Deep Thought demonstrated this strength by winning the 6th World Computer Chess Championship.

---

[1] around 2400 ELO, not to be confused with a US master rating of 2200 USCF

In October 1989, Deep Thought challenged Gary Kasparov, the world chess champion, to a two game exhibition match. Kasparov outplayed the machine not only positionally, but more surprisingly tactically, to win the match 2-0. During December 1989, Deep Thought played David Levy in a four game match. Deep Thought illustrated its impressive ability by winning the match 4-0.

1.2 My Background

I have been very interested in computer chess for some considerable time. I am a member of the International Computer Chess Association and have written a number of programs using both selective and brute force searching.

In August 1989, I entered my latest program HIARCS 4.1 in the chess tournament of the first ever *Computer Olympiad*. HIARCS came a clear 6th with 3 out of 8 points. It was running on a 7Mhz 68000 processor, slower hardware than all of the opposition, for example a *Fidelity Electronics* entry from Miami ran on a 68030 at 44Mhz with 25ns 1 Mb DRAM.

1.3 Aims of the Project

It is clear to me that current programs suffer from a major weakness in the positional and strategic area of chess. Below 2000 ELO, tactics play a major role, however, above 2000 ELO, tactical ability becomes less important and good positional sense is vital. Currently, the best programs are much closer to world championship strength tactically than they are positionally speaking.

In my project, I will give an overview of the subject and explain various techniques used in current programs. I will look into ways of developing an evaluation function that displays the positional ability of an expert level chess player (approximately ELO 2000). I will also look into fresh ways of improving the tactical strength of a chess program. This is particularly important for programs running on standard microprocessors which obviously cannot compete with the raw speed of special purpose hardware. I then intend to implement certain aspects of my findings in a chess program and evaluate them against both human and computer opposition. I also intend to use the Bratko-Kopec experiment [12] to test the relative tactical and positional ability of the program.

The reader is assumed to have a reasonable knowledge of chess and algebraic chess notation in order that chess concepts can be explained to a high enough level.

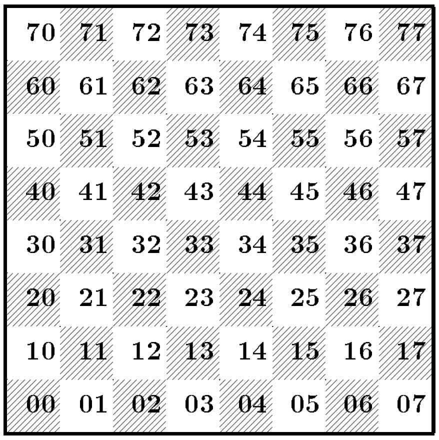
Chapter 2

Basic Overview

In later chapters of this report I will explain and put forward complex ideas concerning various aspects of computer chess. In order to help the reader cope with these ideas, this chapter will give a basic overview of various techniques used in a conventional chess program. I would also advise the reader to follow up the references given for a more complete explanation of the techniques mentioned, as this chapter is by no means intended to cover all areas in detail.

2.1 Machine Representation

A vital element of any chess program is how the chess board and pieces are represented in the machine.



70	71	72	73	74	75	76	77
60	61	62	63	64	65	66	67
50	51	52	53	54	55	56	57
40	41	42	43	44	45	46	47
30	31	32	33	34	35	36	37
20	21	22	23	24	25	26	27
10	11	12	13	14	15	16	17
00	01	02	03	04	05	06	07

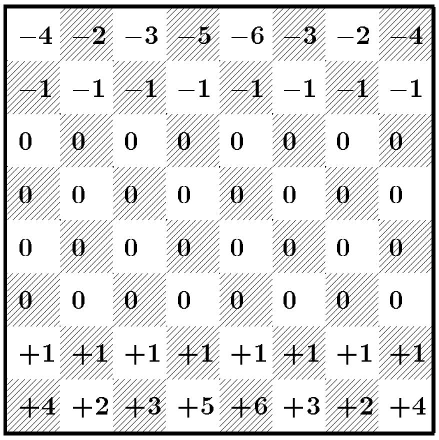
Figure 2.1: A Chess Board indexed by Co-ordinates.

Claude Shannon suggested in his famous 1949 paper [53], that a chess board could be represented in 64 bytes, one for each square on the board. It was proposed that each square could be addressed by the co-ordinate scheme shown in figure 2.1. Each square also had an associated code which identifies the type of piece (if any) occupying that square. The suggested piece codes can be seen in figure 2.2.

Occupied by	*White*	*Black*
Empty Sq	0	0
Pawn	1	-1
Knight	2	-2
Bishop	3	-3
Rook	4	-4
Queen	5	-5
King	6	-6

Figure 2.2: A table showing piece codes

This method is known as the "mailbox" representation. By storing one of these codes in each square's address, any chess position can be represented successfully. The opening chess position, encoded using this format can be seen in figure 2.3.



−4	−2	−3	−5	−6	−3	−2	−4
−1	−1	−1	−1	−1	−1	−1	−1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
+1	+1	+1	+1	+1	+1	+1	+1
+4	+2	+3	+5	+6	+3	+2	+4

Figure 2.3: Mailbox Representation of the Opening Position.

Most implementations of the mailbox idea actually use a 10 x 12 board. This 10 x 12 scheme gives a 2 square edge around the board, each 'off board' square containing a code which identifies it as not being on the conventional board. This is done in order to detect moves off the edge of the board simply by testing the code of the square being moved to. The 2 square edge is necessary to allow for knight moves.

A generally more efficient method of representing a chess board was first suggested by a Russian group in 1970 [2]. It involves using 64 bits, 1 bit per square, to represent a single piece of information about all squares. Each 64 bit representation is known as a bitmap. A bitmap could, for example, represent all squares occupied by a white knight. Bit(s) being set corresponding to whatever squares are occupied by white knights. Using 12 such bitmaps, 6 for white pieces and 6 for black pieces, any position can be represented.

2.2 Move Generation

Using a 10 x 12 mailbox representation, moves are generated by doing a series of adds and tests. For example, to generate all white knight moves from a given square the offsets: +8, +19, +21, +12, −8, −19, −21 and −12 must be added to the knight's original location, see figure 2.4.

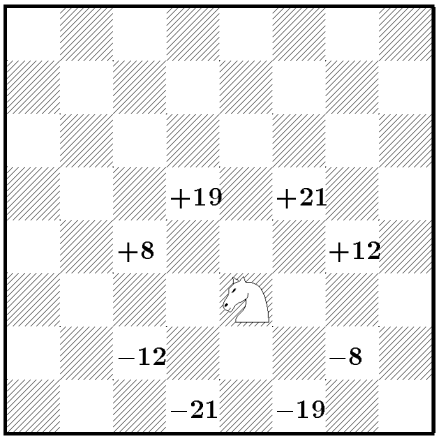


Figure 2.4: Move generation of knight moves using offsets.

Each of these potential "to-squares" must be tested for 2 possibilities:

- Is the to-square on the board ?
- Is the to-square unoccupied by a friendly piece ?

If both these tests prove to be true, then a pseudo legal move has been generated. Note that this does not determine whether the move actually leaves the king in check, hence it is called pseudo legal. Nearly all programs leave this calculation until a more convenient time. As you can see this method of move generation is quiet simple.

Using a bitmap representation, move generation is much more efficient with respect to time, than with the previous method. The reason for this advantage lies in the fact that moves are generated by using bitwise logical operations such as "and", "or" and "not".

In order for this method to generate all white knight moves from a given square, a bitmap needs to be fetched which represents all moves a knight can make from the given square, call this bitmap 1. Next, a bitmap is fetched which represents the location of all friendly pieces. This bitmap is negated and the result is logically and-ed with bitmap 1. The result is a bit, set in the location of every pseudo-legal move possible by a white knight from the given original square.

As can be seen this method is extremely fast. It does however, use much more memory than the mailbox based method because the moves of every piece from any given square must be pre-stored. The method also relies on a large word size, such as 64 bits. Although it is still feasible with a 16 or 32 bit word, it is significantly less efficient. Also it is very important for the computer to have a machine instruction which can give the location of the first bit set in any word; without this, the method may be too inefficient. For a more complete explanation of this technique see [15].

In both methods the moves of vector pieces such as bishops, rooks and queens are more complex to generate. Pawn moves and special moves such as castling and en-passant are also more complicated, but can still be generated using either representation.

2.3 Evaluation Functions

Given that moves can be generated how can the relative merits of each move be determined ? That is the purpose of the *Evaluation Function.*

Shannon proposed that an evaluation function should take account of such features as material, mobility, position of pieces and pawn structure. Each of these factors required an importance weighting to be associated with them. Each factor would then be valued depending on its presence or absence in a given position, and the associated weightings applied. All the values could then be added together to give a single discrete value for the position. The move or series of moves which produce the most desirable position from the evaluation function's point of view can then be chosen.

Almost all evaluation functions use material as an overriding factor, it is very rare for even the sum of all other factors to exceed the value of one pawn. The reason for this is that because the positional values are so crude, if material were to be given away in exchange for positional compensation, the machine would more than likely lose because its evaluation of the compensation was inaccurate.

In the forty years since Shannon's paper, many additional evaluation function factors have been suggested and built upon. They include king safety, occupation of weak squares, development, piece exchange term etc. However, even today very few evaluation functions would even consider exchanging material for other factors such as active play.

In programs which search only a few positions, the evaluation function can be quite complex and time consuming. However, all successful programs search a large number of positions and therefore can only spend a small amount of time evaluating each position. For this reason many of the originally simple ideas have remained to the present day in most programs. Some evaluation functions do not even evaluate the whole position, but rather the value of a move itself. This does not, however, consider other interactions that a move may have on a position. For example, a knight that moves revealing a bishop pin of a queen to a king. The Technology program [22], is an extreme example which only evaluates material during its search. A one ply positional value is combined at the end of the search to distinguish between moves of equal material outcome.

A good explanation of a conventional chess evaluation function can be found in [4]. Evaluation functions will be looked at in much greater detail in future chapters.

2.4 Look Ahead

In order to look ahead into possibilities which may develop in the game, it is necessary generate what is called a "Game tree". A game tree is a diagrammatic representation of possible move sequences. This is achieved by generating moves for the side to play, then replies for the opponent, then counter replies and so on, depending on the depth of look ahead required. Figure 2.5 shows a possible look ahead game tree from the opening chess position.

Figure 2.5: A Possible look ahead tree.

Strictly speaking the game tree is not a tree, but rather a directed graph. This is because some nodes can be reached through different paths, known as a transposition of moves. For the purpose of consistency with other material on the subject I will also use the term search or game tree.

There are a number of ways the game tree can be expanded and searched:

- **Depth first search**: The first branch is explored to the required depth before the machine starts on the next unexplored branch one level up. This continues until all nodes have been explored to the required depth.
- **Breadth first search**: All nodes at one level are examined before progressing one level down the tree and examining all nodes at that level. This continues until all nodes have been explored to the required depth.
- **Best first search**: At each step the most promising node is selected from all the nodes generated so far in the tree. New nodes are then generated from the selected node, and again, the most promising node is selected. A node can be selected as most promising no matter where in the tree it

occurs. Also nodes previously searched, but rejected as unpromising, may again become promising and be re-searched.

Note that the required depth will be the limiting factor for the breadth first and depth first searches. The best first search must have some way of terminating, although depth may not be the limiting factor.

I know of no active programs which uses the breadth first search. Most chess programs use a modified version of the depth first search. These modifications will be explained in chapter 3. The best first procedure is used in some selective search programs. Of particular interest is the B* tree search algorithm [6], a development of the best first search. This algorithm attempts to allow for uncertainty to be expressed in the search tree.

The search of a two-player game tree can be achieved by the use of the minimax procedure. This is based on the idea that two players exist, Max and Min. Max chooses moves leading to high scores, where as Min chooses moves leading to low scores. Max and Min choose moves at alternate levels of the tree, odd and even respectively.

2.5 Selective Searching

It has been shown by de Groot [16, 17] that there are on average 38 moves in a given position, yet only an average of 1.76 good moves[1] exist in each position. A human chess master's search tree typically involves a total of around 35 moves; variations are explored to an average depth of 7 plies. It is clear from the research of de Groot that the human chess master is very efficient at selecting moves to explore further. It therefore seems logical to program a computer which selects the most plausible moves to be explored next from those available. If such a program had a selection mechanism as sophisticated as a master's, it would almost certainly play at master level or above. This method is know as *Selective Searching* or a Shannon type B/C strategy.

The Greenblatt program was the first successful 'selective' searching chess program which could play at a reasonable level. The program placed limits on the number of moves which could be searched at different depths. The number of branches explored in the first 5 plies were as follows: 15, 15, 9, 9, 7. This tapering of the search meant that deeper searches were possible when compared with the full width approach at the time. In the early 1970's, Hans Berliner considered very selective searching as the way forward and expected the search tree would contain less than 500 nodes. It is ironic that Hans is now a major

---

[1] Grandmaster standard moves

force behind Hitech, a very strong brute force searching program from Carnegie-Mellon University.

The best attempt at selective searching was made by Wilkins [61], who produced a knowledge based program, PARADISE. The program performed at the expert level in the relatively narrow domain of tactical problems.

All the attempts to create a good selective search program to play the whole game have hit problems. This is because it is vital to include all important moves in the selective search, as any moves not selected can never be played over the board. This can cause serious tactical threats to be overlooked.

Some moves may not appear immediately plausible, although they may turn out to be very strong some moves later. For these reasons many good moves are overlooked and not selected by a plausible move generator. These problems are best illustrated by some examples.

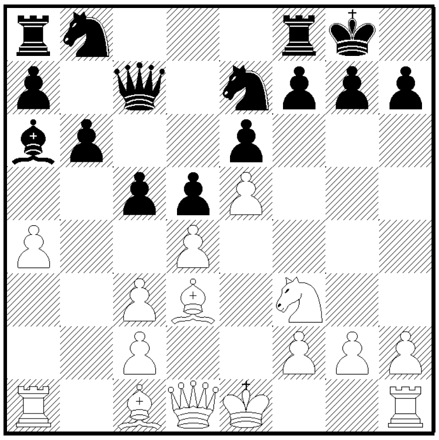


Figure 2.6: White to play and win.

In figure 2.6, white can play 1. Bxh7+ Kxh7 2. Ng5+ Kg8 3. Qh5 leading to mate. The initial move may however, not appear plausible to a selective search program because it appears to lose the bishop for a pawn. Any plausible move generator which did consider Bxh7+ plausible would need to 'recognise' in figure 2.7, an almost identical position, that 1. Bxh7+ Kxh7 2. Ng5+ Kg8 3. Qh5 fails to 3. ... Bd3.

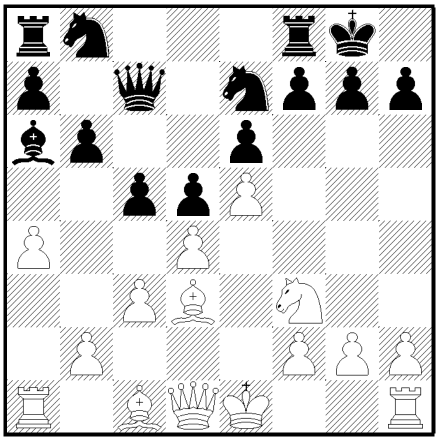


Figure 2.7: White to play, no forced win.

These difficulties illustrate that to keep the branching factor down the plausible move generator needs to be very selective, while not excluding key moves. At the present time we do not have the capability to instill enough human chess knowledge into a program to make this method effective for the whole game.

2.6 Full Width Searching

Instead of selecting promising lines to investigate further, it is possible to explore all moves to a given depth. This Shannon type A method is known as a *Full Width Search* or a *Brute Force Search.*

This method is much easier to implement than a selective search. However, in order for play to be of a reasonable standard a relatively deep search is necessary. The problem here is that the number of nodes involved in a full width search grows exponentially with depth. In a tree of uniform width 38, the search tree will contain over 3 billion nodes after just 6 plies. To cut down this exponential explosion, various, often complicated techniques need to be used. In fact, a good full width search requires many support routines to enable a deep search to be completed in a reasonable time (3 minutes). Using the techniques currently available it is possible to reduce the average branching factor of a full width search from 38 to around 6 to 8 branches. This is achieved through the use of backward pruning techniques which will be explained in detail in a later chapter. Some selective search programs actually have a branching factor higher than 8 e.g. Greenblatt's MacHack. In such cases a full width search can reach at least as deep as the selective search, without any of the possible drawbacks which were outlined in the previous section.

2.7 Quiescence

A position is considered *Quiescent* if it is quiet (inactive). A quiet position could be one where no immediate threats exist. It is very important in any chess program to determine whether a position is *Quiescent* or not, before applying the evaluation function. For example, it would be wrong to apply an evaluation function to a position where a capture had just taken place without checking if a piece can be recaptured. Positions can also be un-quiescent on positional grounds, although the detection of such positions is far more complicated than for quiescence involving tactics.

An evaluation function should only be applied in quiescent positions. Most current programs resolve this problem of quiescence by doing what is known as a quiescence search. That is, a limited search is performed until a quiescent position is reached. Only at that stage are factors such as material balance evaluated. In principle this idea is an excellent one. However, in practice most programs only perform what is known as a capture search at horizon nodes of their main search in order to resolve the quiescence problem. In such a capture search the side to move can either stop the search along a branch and accept the current evaluation score or make more capture moves. When no captures exist for the side to move, the position is considered quiescent. This "quiescence search" significantly improves the ability of the chess programs that use it. However, many threats can be overlooked by such a search, for example, a move forking a king and queen.

2.8 Horizon Effect

Chess programs typically suffer from what is known as the *Horizon Effect*. This effect is best described using an example. In figure 2.8, it can be seen that black is faced with the certain loss of its queen for at most a white rook. However, if a program searching 2 plies were playing black it would think it could save the queen by playing Bxh2+. This appears to the program to save the queen because its loss is now pushed over the program's search horizon.

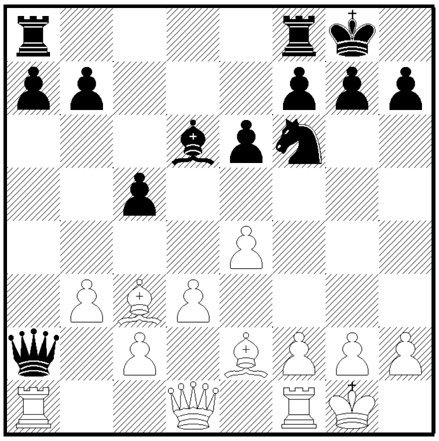


Figure 2.8: Illustration of the Horizon Effect, Black to Play.

This horizon effect is particularly prevalent in programs which explore variations to a fixed depth. This is because moves which delay the loss beyond the depth limit are considered good. A program which can extend its search depth along variations which lead to non-quiescent positions is better able to avoid problems caused by the horizon effect. A program's ability to minimise the horizon effect is therefore largely dependent on its definition of quiescence and how non-quiescent positions are resolved.

Chapter 3

Advanced Topics

This chapter contains both an explanation and review of some of the techniques I have researched during my investigation into tactical and positional evaluation. Some of the techniques represent the state of the art, others have been rejected by the computer chess community as no longer being suitable. These older techniques are investigated because they may contain ideas relevant to my project, and may in fact still be useful when combined with some other ideas of my own.

3.1 Tree Searching

This section covers techniques for searching the game tree. The basic alpha–beta search algorithm is explained and various enhancements over the basic algorithm are presented. These backward pruning algorithms are most applicable to full width searching. This section also contains a brief description of two forward pruning techniques more applicable to an algorithmic approach to selective searching.

3.1.1 Alpha Beta Search Algorithm

In 1958, it was discovered [42] that large savings could be made over the normal minimax procedure by the use of what is now known as the Alpha-Beta ($\alpha\beta$) search algorithm. The $\alpha\beta$ algorithm uses two bounds, a lower bound called alpha and an upper bound called beta. These bounds form a search window within which the true search tree value is known to lie. Knuth and Moore give an in depth analysis of this backward pruning algorithm in [33]. Figure 3.1 illustrates the application of the $\alpha\beta$ bounds. The figure uses the negamax convention where each subtree returns its value negated. This allows each side to maximise its score at each level of the tree, simplifying the implementation of the algorithm. α and β are initially set to $-\infty$ and $+\infty$ respectively.

Figure 3.1: An example Negamax Search tree.

In the figure, nodes C and D are searched and evaluated, resulting in scores of -5 and -2 respectively. Using negamax a score of $+2$ is backed up to node B. This value then replaces the α value at ply 1 because it is larger than α. Node F is then searched, which produces a value of $+1$. This value is not large enough to affect the α value at ply 1 above. Hence, there is no reason to search node G since the resulting value of node E can only be ≤ -1 (negamaxed). Therefore, node G is said to be cut off or pruned from the search tree. The negamax approach has the effect of making all cut offs, β cut offs. The pseudo code for the $\alpha\beta$ algorithm can be found in figure 3.2.

The normal full width minimax procedure, applied to a tree of uniform width W and depth d, would have to search W^d terminal nodes. The $\alpha\beta$ algorithm under optimal conditions can prune a tree to the extent that only the following terminal nodes need to be examined [13]:

- $W^{(d+1)/2} + W^{(d-1)/2} - 1$ at odd depths.
- $2 \cdot W^{d/2} - 1$ at even depths.

This is known as the minimal search tree. To achieve this result, most moves must be searched in a descending order of merit. Of course the true merit of any

```
Function AlphaBeta(P : position; $\alpha$, $\beta$ : integer) returns integer

var
    m       : move
    score,
    best    : integer

begin

    if (Terminal(P))                       /* Is position P terminal ? */
        return (Evaluate(P, $\alpha$, $\beta$))          /* Return score at leaf nodes */
    endif

    GenerateMoves(P)                       /* Generate all moves in position P */
    best := $-\infty$                      /* best move score not yet known */
    for each move m do
        /* Search move m using negamax convention (parameters reversed) */
        score := $-$AlphaBeta($P_m$, $-\beta$, $-$max($\alpha$, best))
        if (score > best)
            best := score                  /* Store best score found */
            if (best $\geq \beta$)
                return (best)              /* $\beta$ cut off : return best score */
            endif
        endif
    endfor

    return (best)                          /* Return score of best move found */

end AlphaBeta
```

The **max**(α, best) operation represents the 'fail soft' condition which allows the search at each level to return the best value found, even if this is less than α.

Figure 3.2: Pseudo code for the $\alpha\beta$ Algorithm.

move is not known before it is searched. Therefore, heuristics must be used to achieve this ordering. In practice the $\alpha\beta$ algorithm searches a tree 3 to 6 times larger than the minimal tree because of sub–optimal ordering. However, the enormous savings produced by using the $\alpha\beta$ algorithm are such that it begins to make full width searching a plausible alternative to selective searching.

3.1.2 Aspiration $\alpha\beta$ Search

The basic $\alpha\beta$ search is carried out with initial $\alpha\beta$ values set to $-\infty$ and $+\infty$ respectively. The aspiration $\alpha\beta$ search makes assumptions about the expected value of the search tree. The $\alpha\beta$ bounds are therefore initially set to a narrow window centered around the expected search tree value, ± a pawn, for example.

Unlike the basic $\alpha\beta$ search which always returns the true score of the tree, an aspiration $\alpha\beta$ search can produce 3 possible outcomes:

1. **A true score**: The true value of the tree lies inside the search window and was therefore found by the search.
2. **A fail low**: The true value of the tree is smaller than the α bound.
3. **A fail high**: The true value of the tree is larger than, or equal to, the β bound.

In case 1, the search has been successful because the best move and its true value has been found. In cases 2 and 3 the search has failed to find a true value for the tree because of $\alpha\beta$ cut offs. However, the direction of the tree's true value relative to the $\alpha\beta$ bounds is now known. When the search fails low the true value is known to lie in the range $(-\infty,\alpha)$, and when it fails high, the true value will lie in the range $(\beta,+\infty)$. Using the appropriate new search window the tree must be re-searched to find the true score.

This approach can produce a noticeably faster search than the basic $\alpha\beta$ algorithm if the initial window is well chosen. It can, however, also produce a slower search if the value of the tree lies outside the initial window, as a re-search will be necessary. In chess, material is rarely won or lost. Therefore, a window of a couple of pawns width will only fail occasionally to find a true score at the first attempt, making this technique generally faster than the basic $\alpha\beta$ algorithm.

3.1.3 Principle Variation $\alpha\beta$ Search

The aspiration $\alpha\beta$ search is effective because the reduced $\alpha\beta$ window cuts off numerous subtrees on the assumption that something better exists. If we took this reduced window idea to its limits we would have a minimal window.

A minimal window uses α and β values set so close together that no tree value can be enclosed within the window. Therefore, the $\alpha\beta$ search will definitely fail low or high depending on where the true score lies relative to the minimal window, thereby identifying in which direction the true score lies. The **P**rinciple **V**ariation $\alpha\beta$ **S**earch (**PVS**) uses this idea by first doing a basic $\alpha\beta$ search to find the true value of the principle variation. All subsequent searches are then done with a minimal window centered on the current value of the principle line. This works on the idea that it is much easier to show that a subtree is inferior than determine its true value. Therefore, when a minimal window search fails low, the move is discarded as being worse than the move already found. If however, it fails high, it must be re-searched with a window wide enough to encompass the true score.

Figure 3.3 shows the pseudo code for the PVS algorithm. The pseudo code includes Reinefeld's depth ≤ 2 enhancement [46]. This enhancement uses the fact that the 'fail-soft' max operation allows true scores rather than β values to be returned from the last 2 plies of the tree. Hence, at the bottom 2 plies of the tree, re-searches are unnecessary.

```
Function PVS(P : position; α, β, depth : integer) returns integer

var
    m       : move
    score,
    best    : integer

begin

    if (depth = 0)
        return (Evaluate(P, α, β))              /* Evaluate leaf node */
    endif

    GenerateMoves(P)                            /* Generate all moves in position P */
    /* Search principle variation to next depth using negamax convention */
    best := $-$PVS($P_1$, $-\beta$, $-\alpha$, depth$-$1)
    if (best $\geq \beta$)
        return (best)                           /* β cut off : return best score */
    endif
    for each move m do
        $\alpha$ := max(best, $\alpha$)                   /* Allow fail soft */
        /* Search move m to next depth using negamax convention */
        score := $-$PVS($P_m$, $-\alpha - 1$, $-\alpha$, depth$-$1)
        if (score > best)
            /* Re-search only if score is within αβ window and not a true */
            /* score returned from deepest 2 levels (Reinefeld's enhancement) */
            if (($\alpha$ < score) and (score < $\beta$) and (depth > 2))
                best := $-$PVS($P_m$, $-\beta$, $-$score, depth$-$1)
            else
                best := score
            endif
            if (best $\geq \beta$)
                return (best)                   /* β cut off : return best score */
            endif
        endif
    endfor

    return (best)                               /* Return score of best move found */

end PVS
```

Figure 3.3: Pseudo code for the PVS Algorithm.

3.1.4 Iterative Deepening

Iterative deepening [4] is a method of improving move ordering, thereby increasing $\alpha\beta$ cut offs. It also enables the expected principle variation to be primed.

Iterative deepening is achieved by performing a series of searches to progressively greater depths e.g. 1 ply, 2 ply, 3 ply etc. After each search the 1 ply move list can be re-ordered to increase the efficiency of the subsequent iterations.

It might be thought that doing an iterative search to 4 ply would be less efficient than a direct 4 ply search, which after all will be done anyway as part of an iterative search. However, there exists numerous techniques for improving the speed of each subsequent iteration by making best use of the information gained so far. These techniques include refutation tables, transposition tables, the history heuristic etc. These techniques will be explained in detail later. An iterative PVS combined with some of the techniques mentioned can produce trees very close to the minimal tree [40].

An iterative deepening $\alpha\beta$ search also has another important benefit; the best move found so far is always known. As all moves are searched at each iteration, this move is likely to be reasonable. Therefore, if the available time expires, the best move found so far could be played with confidence. In a non–iterative search, however, if the time expires, the move selected as best would depend on what moves had been searched so far. If only poor moves had been searched, only a poor move could be chosen. This is an important benefit for competitive chess programs because they must control their own time usage.

3.1.5 Move Ordering

As stressed earlier, the order in which moves are examined is critical to the efficiency of $\alpha\beta$. The moves from each node can be ordered using a variety of heuristics:

- An iterative search is a good basis for improving move ordering as the moves at the root of the tree are directly ordered by a search to a depth 1 ply less than the current search depth. As the subtree below moves at the first ply is usually larger than subtrees at greater depths, move ordering at ply 1 is likely to have the greatest effect on overall tree size.
- Storing moves in a transposition table represents perhaps the best method of ordering moves in an iterative $\alpha\beta$ search [18]. This is because moves found to be best at iteration n are very likely to remain best at iteration n+1. Therefore moves stored in the transposition table should be tried first if the node is re-visited.

- Capture moves are worth searching before other less dynamic moves because they are more likely to refute the current line of play and therefore cause a cut off. Capturing the piece just moved is generally very effective at refuting a given line of play [4].
- A move that causes a cut off at a given depth should be tried early as part of another variation at the same depth. This technique is known as the *Killer* heuristic [4].
- The *History* heuristic provides a method of "recording each moves ability to cause a refutation" [50]. This information can then be used to order moves according to the likelihood of them causing a refutation.
- Methods which use the evaluation function to score and order moves are quite accurate, but expensive; prohibitively so at the deepest levels of the tree.

The objective of move ordering is therefore to search moves in a best first order. Therefore finding good α values quickly and causing early β cut offs. The time spent ordering moves must be small or the process will be self–defeating. Therefore the first five methods are most common in contemporary brute force chess programs.

3.1.6 Null Moves

In chess, unlike some other games, the option of not playing a move does not exist. However, this option can be used to good effect as part of the search tree in order to detect tactical moves.

The idea is that by giving up the right to move, it allows the opponent a second consecutive move. If this second move does not accomplish anything beneficial, then it can be assumed that the first move did not represent any threat. In this way moves which represent a threat can be identified.

This method assumes that it is always better to make a move, than no move at all (a *null move*). In fact in chess, a situation known as "zugzwang" can exist where all available moves lead to a loss of some sort; whereas if no move were made, the loss could be avoided. An algorithm using the null move to determine threats could therefore overlook moves which leave the opponent in zugzwang. However, zugzwang is very rare and does not cause a significant problem for the technique.

Don Beal has done a lot of experimentation into the use of null moves in a normal $\alpha\beta$ minimax search. In his paper "Experiments with the Null Move" [5], he proposes an algorithmic approach to selective searching based on null

```
Function Quiescence(P : position; $\alpha$, $\beta$ : integer) returns integer

var
    m       : move
    score,
    best    : integer

begin

    /* Calculate null move score. Note : Evaluate may itself be a search */
    best := $-$Evaluate(P, $-\beta$, $-\alpha$)

    GenerateMoves(P)
    for each move m do
        if (best $\geq \beta$)
            return (best)                  /* $\beta$ cut off : return best score */
        endif
        /* Search move m to next depth using negamax convention */
        score := $-$Quiescence($P_m$, $-\beta$, $-$best)
        if (score $>$ best)
            best := score                  /* Store best score found */
        endif
    endfor

    return (best)                          /* Return score of best move found */

end Quiescence
```

Figure 3.4: Pseudo code for a Quiescence search based on Null Moves.

moves. In particular he defines a null move quiescence search which is capable of "selecting moves such as checks, moves out of check, attacks on pieces, defences, blocks and other types of sharp tactical moves that chess players concentrate on". This is achieved with no more chess knowledge than material balance.

Beal uses the score returned from a quiescence search following a null move as a bound to guide the search and enable forward pruning of inadequate moves. Beal shows that such a quiescence search is capable of a performance on Reinfield's 300 tactical positions [47], equivalent to that of the chess machine Belle, while using only 1/15th of the effort !

Very recently, Hamlen [25] extended this idea by applying a full width 2 ply search, in addition to the normal quiescence search, after a null move. This allowed the algorithm to additionally detect tactical ploys such as forks and skewers. Other programs, notably Kaissa [1], have used the null move idea to provide faster cut offs, without using it algorithmically to select moves as Beal and Hamlen have.

Beal's null move quiescence search can be seen in figure 3.4. It is important to realise that Evaluate may itself be a search of some kind. In fact Beal advocates the use of a 2nd level null move quiescence search supported by a 1st level quiescence search, in place of Evaluate, using material balance as its evaluation.

3.1.7 Razoring

Razoring is a technique devised by Birmingham and Kent [11]. The technique uses the assumption that a player who has the move can always maintain or improve his position by playing a move.

The technique works by using a static evaluation value at interior as well as terminal nodes. This value is compared to the current α value with the assumption that the opponent can always maintain or improve his position with a move. Therefore, the static score is treated as an upper bound on the possible scores resulting from the opponents available moves. Hence, if this static value is no greater than the α value, it is assumed that the subtree cannot affect the α value at this level and can therefore be cut off. Note that this technique is only applied from the point of view of the player who has just moved.

Figure 3.5 illustrates the razoring technique in action. The numbers to the left of each node represent the static values, while the numbers to the right represent backed up negamax values. Node B is examined first giving a static value of $+7$. Nodes C and D are then searched, resulting in values of -15 and -4 respectively. A value of $+4$ using negamax is subsequently backed up to

Figure 3.5: Razoring in action.

node B. The α value at this level is then made equal to +4. The move leading to node E is now examined and given a static value of +1. Using the razoring technique, this value represents an upper bound, because the opponents moves leading to nodes F and G are assumed to maintain or improve the opponents score thereby decreasing the score at this level. As the upper bound of +1 is already less than the current α value at ply 1, nodes F and G cannot affect the search result and can therefore be cut off by razoring.

The razoring technique can be extended by using the assumption that a player can improve his current position by playing his move, your move, his move. This extension of the idea is known as deep razoring. It causes even more razor cut offs, but at a much greater risk of cutting out a vital move.

Birmingham and Kent also proposed a further enhancement where by the assumption is made that there always exists a move which not only improves a players position, but improves it by at least 3/10ths of a pawn, for example. This 'enhancement' further increases razored cut offs, but at the cost of missing possibly more vital moves.

I believe the technique would be better enhanced by taking an opposite view to the enhancement above. For example, assume that there always exists a move that will not deteriorate the position by more than a threshold, of say half a pawn. Different thresholds could be applied at each level of the tree, perhaps weighted so that as the search progresses more deeply down the search tree, more cut offs are made. This enhancement would decrease the number of cut offs, but improve the likelihood of including good moves in the search tree, particularly at shallower depths where missing such moves can be fatal.

I believe the assumption made by the Razoring technique is much weaker than that of the Null move technique, which only assumes that it is better to make a move, than no move at all.

3.2 Evaluation

This section describes a number of techniques for heuristic positional evaluation of chess positions. Most of these techniques are best used in combination with each other.

The importance of the evaluation function cannot be too highly stressed as it is this that gives a program its distinctive style. Achieving a good position is often a prelude to a winning tactical combination.

> "Tactics flow from a good position"
>
> *Robert J. Fischer.*

3.2.1 Piece Square Tables

Piece Square (Pc/Sq) tables are used to hold values for pieces occupying squares on the board. For example, a knight square table would contain a score for each square indicating the merits of having a knight on that square. Tables can be used for each type of piece and can be pre-loaded before the search begins, by doing some preliminary positional analysis. Using such tables, it becomes a simple matter to perform a positional evaluation at a given node. Given the location of each piece on the board, a value can be looked up in the relevant table and added to the positional total for the appropriate side, white or black. Finally the difference between the two totals is calculated giving an overall value for the position.

The main advantage of using these tables is speed. The worlds fastest chess machines, Hitech and Deep Thought, make extensive use of Pc/Sq tables. However, because these tables are generally pre-loaded before the search begins and applied n ply further down the tree at leaf nodes, many positional aspects are no longer applicable. The tables also do not reflect up to date piece interactions.

These problems are exasperated by deep searching. It is therefore ironic that the tables are used for the express purpose of deep searching due to their efficiency. It is interesting that the Hitech team at CMU are now finding that as their program improves, more knowledge must be migrated from the Pc/Sq tables to the leaf node static evaluation process [7].

3.2.2 Static Evaluation

In most programs, a static evaluation is performed at the leaf nodes of the main search. In full width searches, a quiescence search is usually also applied at these leaf nodes, thus creating new leaf nodes at a deeper level which are not usually positionally evaluated for efficiency reasons.

The static evaluation process involves computing the existence of various positional factors. Depending on the degree to which these factors exist, appropriate values are calculated. Finally the values are weighted according to their relative importance.

In chess, determining the factors which contribute to a good position is very difficult. Standard criteria have been developed over the years by which a program can judge a position. Using such criteria current programs play a reasonable game. However, it is still clear from their positional play, that some subtle factors are still not being included. I will try to uncover some of these in the next chapter. The most difficult task remains determining the relative importance of each factor or criterion. An attempt was made to determine the relative importance of a number of evaluation criteria in [51]. I think the basic idea of this research is an excellent one, however, I feel the specific order in which the evaluation criteria were tested, influenced the result of the experiments.

Several attempts have been made to generate a 'best fit' evaluation function [27, 39]. Most of these studies are based on an analysis of grandmaster moves. The objective is to find a function which selects grandmaster moves as often as possible. Such attempts have hit serious problems because, as I see it, 5 problem areas exist:

1. A best fit can only be attempted for the known functions. If a program does not know about a particular aspect of chess, for example weak pawns, it cannot include such considerations in its evaluation process. Therefore, possibly introducing side effects into existing functions.
2. The criteria by which a human chess player evaluates a position are not mutually exclusive, but subtly linked. Identifying these links is very difficult.
3. So much depends on the phase of the game e.g. middle game or end game, that I believe specific functions are required for each game phase.
4. Positionally moves vary with player style and mood.

5. Some moves are tactically conditioned. Such moves must be identified because they affect the positional choice available.

Obviously this area would benefit from more involvement by strong chess players.

In practice, static evaluation functions tend to be primitive, particularly in brute force programs, to enable deep searching. Pc/Sq tables are used where possible to improve efficiency. Information which cannot be stored in the tables, such as pinned piece information, must be calculated. The evaluation process can be supported by specific hash tables e.g. pawn structure hashing. This enables much faster evaluation because calculations do not have to be re-done when positions with identical pawn structure, for example, are encountered. In selective search programs the evaluation process can be far more complex as the reduced number of nodes examined allows more time to be spent at each node. The difficulty here is that knowledge deficiency cannot be easily made up by searching. Therefore, chess knowledge acquisition from grandmasters must play a vital role in such programs. At the present time, techniques do not exist which would allow grandmaster thought processes to be captured and understood in detail, let alone, allow such knowledge to be transferred to the machine.

It should be realised that static evaluation is not adequate in dynamic tactical situations. Such tactical problems are better resolved by some kind of quiescence search. I believe it is the job of the static evaluation function to realise when a position is unsuitable for static evaluation and inform the search process, so it can attempt to resolve the tactical uncertainty.

3.2.3 Incremental Evaluation

Static evaluation at leaf nodes is expensive in terms of time; and Pc/Sq tables do not reflect the true merits of a position, especially as the search deepens. A third option is available, that of incrementally updating the positional totals as moves are made. The positional totals could be updated as follows:

1. Calculate the positional value, for the piece just moved, on its from–square.
2. Decrement this value from the positional total for the relevant side.
3. Calculate the positional value, for the piece just moved, on its to–square.
4. Increment this new value to the positional total for the relevant side.

This process is quite simple and therefore fast, but does not take account of the effects moving a piece may have on the positional value of the pieces around it. For example, moving a piece may reveal or block a number of possible attacks.

Therefore, if the process is to be truly incremental it must take account of such factors. However, such a detailed incremental evaluation will be costly in terms of time, perhaps prohibitively so.

In practice incremental evaluation is used in conjunction with a static evaluation at the horizon nodes of the main search. Incremental evaluations can also be useful during a quiescence search where there is not enough time for a full static evaluation. This is because a quiescence search also, usually occurs at the horizon nodes of the main search.

The Pc/Sq tables explained earlier can also benefit from an incremental approach, to close the gap between knowledge storage and application.

3.2.4 Phases of Play

The game of chess consists of 3 main game phases:

- The Opening.
- The Middle game.
- The Endgame.

Different evaluation criteria are applicable at different phases of the game. Therefore, it is important for a chess program to recognise what phase the game is in, so the correct evaluation criteria or weightings can be applied.

In practice the game phases need to be broken down into more detail, so that the transition between phases is not large. If this is not done, there will be cliffs in the evaluation surface where a program suddenly becomes aware of some aspect of the position. For example, early in the game the king needs to stay near the edge of the board protected by its pieces. As the game progresses however, the king should begin to be moved to an active position ready for the endgame. If this process is not performed gradually, the program may find itself in an endgame with its king misplaced, and a sudden realisation that things are not as good as predicted by the evaluation function at the previous game phase.

Assume, for example, we have 8 game phases, then new evaluation criteria could be introduced gently, or existing criteria could be scaled according to the phase.

The game phase is a factor of the material remaining on the board and can therefore be easily calculated, although deciding when one phase should end and another should start is more difficult. This recognition of the game phase must also include realising when positional factors can be ignored and the opponents king can be driven to checkmate.

The game phase can also be used to encourage the program to trade pieces when ahead in material.

3.2.5 Planning

Planning plays an important role in human chess. It could be said that a program is continually planning to maximise its evaluation function's score. However, this often leads to aimless play, waiting for the opponent to make a mistake.

A plan could be conceived at the root of the tree by an analysis of the position. Specific goals can be identified, such as a king side pawn rush, for example. The relevant Pc/Sq tables and static evaluation criteria can be given weightings to encourage achievement of these goals. For example, increasing the bonus for pushing pawns on the king side. The subsequent search would then be biased towards selecting moves which achieve these 'sub–goals'. To further encourage the use of a consistent plan, the principle variation from the last search can be used to direct the current search. Deviations only being allowed when they are clearly superior.

Chess programs generally have no problem in finding the correct moves of combinations leading to a gain in material. However, they find it far more difficult to create positions where such combinations exist.

3.3 Learning

The Learning process can be split into 5 main categories:

- Rote Learning.
- Advice Taking.
- Induction.
- Learning by Analogy.
- Parameter Adjustment.

The first four categories are directly related to the way humans learn. The final category is derived more from the way programs make decisions than the human learning process.

3.3.1 Rote Learning

This is the simplest form of machine learning. It involves the recording of data, and use of that same data when faced with exactly the same situation again.

Most chess programs use what is known as an "Opening Book". This is a database of pre-stored opening lines of play which are looked up and played automatically while the two players stick to the stored opening line of moves. In more sophisticated chess programs, positions and moves are stored, rather than lines of moves. This enables a transposition of moves to be detected and allows moves to be played whenever a 'known' position is reached. The opening book is very useful for avoiding opening traps, saving time and making moves of a very high standard which have years of theory and practice behind them. Obviously this technique can only cover the first few moves of a game, any deviation causing the program to 'think' for itself.

Various endgame databases have also been created for elementary endings such as king and pawn against king. When database positions are reached the program can simply look up the correct move and play it. I believe for such simple endings, heuristics can be developed which will allow correct play using search rather than massive storage.

A more versatile technique has been suggested by David Slate [58] of Chess 4.5 fame. Slate suggests a method by which a transposition table can be used to enable a chess program to learn from its experience. The idea being to store positions where the program's evaluation changes dramatically. This allows the program to avoid falling into the same trap again. This is of most use in non–book opening positions which may be encountered in another game. Fidelity Electronics are using a similar technique in one of their latest commercial chess machines [21].

3.3.2 Advice Taking

Advice taking chess programs require the gap between the programming level and a chess masters knowledge to be bridged by some high level chess language.

Most notably Zobrist and Carlson [64] developed such a type of advice taking program. The program was tutored by Charles Kalme, a US Senior Chess Master. The 'Chess Language' used was relatively sophisticated, but still required mathematical weights to be assigned to patterns. However, chess masters do not consciously conceptualise positions in this way. Therefore, the advice that can be given is somewhat difficult to assign discrete weightings to.

The program only contained 50 patterns. Research done by Simon and Gilmartin [55] indicates that a grandmaster has a vocabulary of about 50,000 chess patterns. Therefore, it is not surprising that Zobrist and Carlson's program only performed at the novice level.

3.3.3 Induction

A number of attempts have been made to use induction in chess programs. Most involved the narrow domain of simple endings such as king and pawn against king [54]. Given a relatively large number of training events, such programs are able to use induction on a set of known predicates in order to derive a set of formulas. The programs are then able to deal with such endings with 80 to 90% success. However, a 10 to 20% level of incorrect play, especially in such simple endings, is not adequate in tournament programs.

Steven Skiena states in [57], "When more complex endgames are considered, increasingly larger number of predicates will be needed, until forms of constructive induction or feature formation are necessary. Systems for significant constructive induction simply do not exist.".

3.3.4 Learning by Analogy

Learning by analogy is probably the most powerful method applicable to playing chess. Unfortunately it is also very difficult to achieve on a machine.

It has been found by de Groot [16] that grandmasters notice similarities in piece placement, square control and pressure. These concepts are known to strong players in the form of patterns, moves and goals. No major work seems to have been done in this important area. This seems to me to be a major omission, if knowledge based programs are to succeed in reaching master level.

3.3.5 Parameter Adjustment

This type of learning involves adjusting the weights applied to evaluation criteria in the light of experience. Those criteria which lead to improved performance are increased, whereas those criteria which deteriorate performance are decreased or discarded altogether.

The most notable example of this type of machine learning is Samuel's draughts program [48, 49]. However, chess is a far more complex game than draughts. In 1961, Max Euwe, ex–world chess champion, set up a committee

[20] to explore the possibilities of applying Samuel's techniques to chess. The committee concluded that for chess Samuel's techniques "would have to deal not with 31 component numbers but with 31 million or even more" [28]. In other words the techniques are impractical for chess.

More recently Thomas Nitsche [45] has experimented with a learning chess program which adjusts its own parameters using linear regression. The program is capable of learning from three different sources:

- Analysing Master Games.
- Playing against Human Opposition.
- Playing against itself.

Such techniques have not been generally successful in competitive chess programs. I believe the reason for this is that most evaluation functions are not made up of simple linear features as used by Nitsche. Therefore making this type of learning much more difficult to apply.

In conclusion, I think the amount of learning necessary for a program to reach a high level of ability has been severely under–estimated. Simon and Chase [56] found that it takes a player about a decade of intense pre–occupation with playing chess to become a grandmaster. Humans are superb learners, machines are clearly not. It therefore follows that it will take success in most of the categories outlined in this section together with at least a decade of full time work, advice and training to produce a learning, knowledge–based chess program anywhere near the grandmaster level.

3.4 Time Control

Chess clocks are used in competitive chess to allow fair allocation of time to each player for the game. Both players must then conform to a set of time controls. In tournament chess the general time control is 40 moves in 2 hours followed by 20 moves in every subsequent hour. Some time controls can be "sudden death", e.g. all moves in 30 minutes.

Any chess program written to compete with humans or other programs in tournament games must therefore have an algorithm to control its time usage. The time spent on each move does not have to be constant, in fact it is preferable that it is not. It is wise to play obvious or forced moves quickly and take longer on more difficult moves. Therefore, a sophisticated time control algorithm must detect these cases and allocate its time appropriately. I believe it is also part of such an algorithms responsibility to detect when the horizon effect may be occurring and allocate enough time where possible to see over the horizon. To achieve these aims, the algorithm must know what scores the search is calculating at each iteration and the state of the search at a given point in time.

Time control problems are eased by the use of iterative deepening because of the continuous availability of the best move found so far. The time used up thinking about the current move must be checked regularly against the time allocation to avoid the possibilities of over using time or even losing on time. Unfortunately, it is not enough to check the time after each iteration or even after each first level branch. This is because some branches may take a large amount of time to search. Therefore, a successful time control algorithm must be closely linked to the actual tree search mechanism at all but the very deepest levels of the tree.

During my research I have noticed a distinct lack of published material in this area. The articles by R. Hyatt [30] and H. Berliner [7] are the only articles of any substance in this area. I find this rather surprising because as I see it, this is an important element of any strong chess program.

Chapter 4

The Evaluation Function

In this chapter I will define and explain the heuristics I consider to be important in assessing the merits of a chess position. I intend to implement these ideas in a new chess program called "HIARCS 5.0".

I hope to produce an evaluation function which has the positional knowledge at or near the expert level (2000 ELO). It would normally take a human player some 5 years of intensive chess experience and learning to achieve this level. Therefore, my aim to instill enough knowledge in a program within the short period of time available for this project is daunting. However, I can draw on my own experience of tournament chess and the experiences of some great master players who have written about chess strategy. In particular, the famous books "My System" and "Chess Praxis" by Aron Nimzowitsch [43, 44] contain some very deep thoughts on positional and strategic play. Therefore, I will use such ideas to help develop heuristics to evaluate positions.

As I see it, there are three main problems facing the development of a 'good' evaluation function:

- Determining **which** criteria are important.
- Deciding **when** the criteria are important.
- Deciding **what** the relative merits of each criterion should be.

It is vital to get all three of these aspects 'right'. For example, a program may know that bishops are strong in some positions. However, if it does not know with accuracy when those positions occur, it cannot apply its evaluation criteria effectively. There are no perfect answers, often evaluation criteria may contradict each other because of an inevitable interaction and overlap between each type of chess knowledge represented. Therefore, each criterion is not independent in its own right, but must be considered with all other relevant criteria. Hence, the relative weighting for each aspect of evaluation is important. However, the criteria can only be considered correct in a statistical sense, not an absolute sense.

These three main problem areas must be considered when developing each heuristic. Developing an evaluation function is a very iterative process. In the light of seeing a program play the game, the heuristics will be continually evolved. This fits in well with the equivalent human process where knowledge is continually in a state of evolution, although this is an automatic process in the case of a human player.

The evaluation function will deal with the opening, middle game and general endgame to allow a program to play the entire game. Knowledge for specific endgames, such as bishop versus knight endings, will be omitted because of time constraints. However, their implementation is just a matter of applying a similar approach to that which I am applying to the more general cases.

The evaluation criteria will be defined in a formal manner, using both functional and algorithmic descriptions where appropriate. To raise the level of chess knowledge which can be concisely represented by the definitions, I have written a large number of pseudo code functions. These pre-defined functions will be used extensively in the definition of evaluation criteria. A full description of these functions can be found in appendix B.

The evaluation function must be considered as a whole. The positional features for even a single type of piece are evaluated over a number of criteria. Therefore, remember the criteria are not independent, but closely linked together. The evaluation criteria have an affect on both sides. Therefore, while weaknesses are discouraged from one sides point of view, their creation in the opponents position is actively encouraged.

4.1 Definition 1 : Material

Like all other chess programs, my evaluation function will hold material as the main component. However, unlike most other programs this will not be the overriding factor. In fact, I intend to make the evaluation function sophisticated enough to be willing to give up material for good positional compensation.

Traditional chess books for beginners advocate the following approximate values for each piece :- Pawn, 1 point; Knight, 3 points; Bishop, 3 points; Rook, 5 points; Queen, 9 points and King, ∞. However, these piece values would require positional values having a non integer representation in most cases. For reasons of efficiency, most contemporary chess programs use integers for positional as well as material values. Therefore, I propose to use a basic pawn value of 100. This provides a positional resolution to 1/100 th of a pawn. The values of the other pieces will be assigned relative to this basic pawn value.

```
$MT_{col} = \sum_{sq=1}^{64}$    if (Occupy([Any],col,sq))
                                     case On(sq) of
                                         Pawn      : PawnValue
                                         Knight    : KnightValue
                                         Bishop    : BishopValue
                                         Rook      : RookValue
                                         Queen     : QueenValue
                                         King      : KingValue
                                     endcase
                                 endif

where
  PawnValue        = 100
  KnightValue      = 325
  BishopValue      = 340
  RookValue        = 520
  QueenValue       = 960
  KingValue        = 10000
```

Few players agree on the relative merits of a knight and bishop. Both are suited to different types of position. My own view is that programs generally use bishops more effectively than knights. This is because programs play open positions better than closed ones. Bishops are usually stronger than knights in open positions and inferior in closed positions. Therefore I propose to set the knight and bishop values to 325 and 340 points respectively.

Rooks are particularly strong pieces when given scope e.g. on open files.

They are generally worth about a knight and two pawns. I will use a rook value of 520 points.

The queen is the most powerful piece on the board. Its scope and contribution to tactical chess makes it ideal for chess programs. To reflect this, I intend to use a queen value of 960 points.

The king is obviously the most important piece. It's capture will never occur in an actual game. However, it remains useful to assign a value to reflect its ultimate importance. I will use a value of 10000 points.

It must be remembered these values only represent the basic piece values. When positions particularly suited to certain types of pieces occur, the positional value of the relevant piece will increase to reflect its added importance in the current position.

A total material value will be maintained incrementally throughout the search, for reasons of efficiency. This allows the current material value to be almost instantly available at all nodes of the tree, interior and leaf.

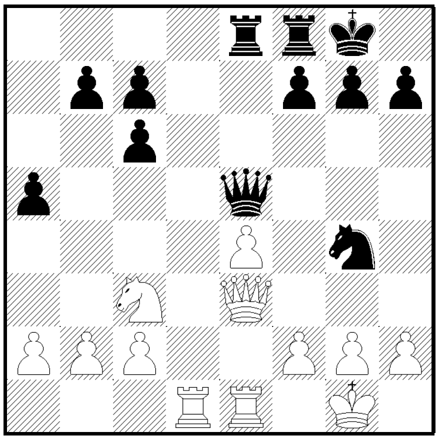


Figure 4.1: **Fidelity 68030 v HIARCS 4.1**, 1989 Olympiad, White to Play.

In figure 4.1, the Material totals for white and black are as follows:

$MT_{White} = 13025$
$MT_{Black} = 13025$

4.2 Definition 2 : Game Phase

As already explained, it is important to split the game into phases, so that the evaluation criteria can be applied smoothly, thus avoiding notorious cliffs in the evaluation surface.

To achieve this smooth transition between different evaluation criteria, I will split the game up into 16 separate phases.

```
phase = PS
revPhase = 17 − phase

where
  PS :          totalMtl = $Mtl_{White} + Mtl_{Black}$
                if (totalMtl $\leq$ 21870)
                    1
                else
                    (totalMtl − 21570) / 300
                endif

  $Mtl_{White}$ :  Total non–pawn white material.

  $Mtl_{Black}$ :  Total non–pawn black material.
```

In the opening position the phase will be 16. The phase then decreases linearly as material is captured, except when the total non-pawn material value falls below 21870 units. At this point the phase remains at 1 for the rest of the game, unless the material value rises again[1]. As an example, in figure 4.1 the Game Phase is calculated by: phase = (24650 − 21570) / 300 = 10.

This phase concept will not be used to decide when rapid development of pieces should take place. This will be left to special development criteria. The concept will however, be used to progressively modify those evaluation criteria which are affected by material status. For example, passed pawns become more important as the material on the board diminishes.

The game phase is based on non–pawn material because in chess it is generally important to trade pieces not pawns when ahead in material. Therefore, the phase provides a method of encouraging the program to trade pieces when ahead in order to increase some favourable positional criteria such as passed pawn values. Trading pawns will not be encouraged because the phase will remain unaffected.

---

[1] because of pawn promotions

4.3 General Positional Evaluation

4.3.1 Definition 3 : Mobility

Mobility is the degree of piece freedom in a given position. Study has shown that there is a clear difference between the mobility of the winning and losing sides as the game progresses. This is to be expected when one side has a material advantage. However, it has also been found that the side which can maintain the highest mobility is the side most likely to win the game [27].

This shows that maximising mobility is an excellent heuristic. In fact, Slater [59] is of the opinion that mobility in itself is a sufficient measure of all the factors suggested by Shannon [53]. I also consider mobility to be an important factor, although I do think many other heuristics are also important.

In 1985, Johnathon Schaeffer [51] did some research to find the relative importance of various evaluation criteria. In this research Schaeffer found that the mobility heuristic contributed more than any other term to program strength. In fact, when a mobility criterion was added to a program without any positional knowledge, a rating increase of 310 points occurred. This was the first evaluation criterion added. It could therefore be expected to make a large contribution to program performance. However, such a large increase is exceptional and clearly shows mobility should be one of the major terms in any evaluation function.

Normally, mobility is taken as the number of legal or pseudo–legal moves. I will take a slightly different view and use the following definition of mobility:

$$MB_{col} = \sum_{sq=1}^{64}(A_{col,sq} + P_{col,sq})$$

where

$A_{col,sq}$: ♯Attacks([Pawn,Knight,Bishop,Rook],col,sq)

$P_{col,sq}$:

```
if (Occupy([Pawn],col,sq))
    ♯Moves([Pawn],col,sq)
else
    0
endif
```

The mobility of the king is not included in this definition because I do not consider a kings mobility makes any meaningful contribution to a successful strategy. The king must be protected in the opening and middle game. In the ending it should be used actively. However, its mobility should neither be minimised nor maximised. The queens mobility is also not included although a specialised queen mobility function is included in definition 27, Queen Power. Other non-pawn pieces are given a bonus for each square they attack, including friendly pieces defended.

Unlike other pieces, pawns cannot move to squares they attack, therefore I have given pawns a bonus for squares they can move to. This in effect encourages the blockade of enemy pawns by pieces.

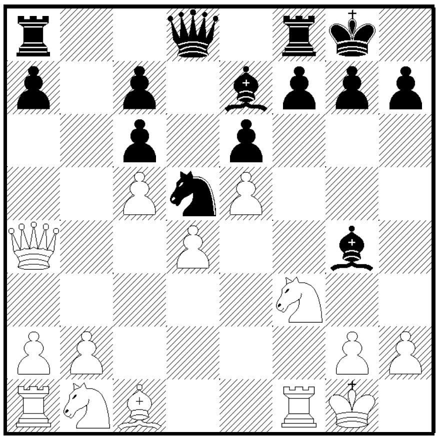


Figure 4.2: **HIARCS 4.1 v Mephisto**, 1989 Olympiad, White to Play.

Applying these Mobility criteria to the position in figure 4.2 gives:

$MB_{White} = 43$
$MB_{Black} = 47$

4.3.2 Definition 4 : Development

The main objective of the opening is to develop pieces onto their 'best' available squares. Development, therefore, obviously plays an important role in chess. Many openings offer material, generally pawns, in exchange for rapid development. A developed side is able to combine pieces in attack, often catching an unprepared and undeveloped opponent by surprise. I believe it is important for chess programs not to place too much faith in their opening books. A program should know how and when to develop its pieces.

$$DV_{col} = K_{col} - B_{col} \cdot phase/3 - N_{col} \cdot phase/2 + Q_{col}$$

```
where
  $K_{col}$ :  if (Castled(col))
                 phase
               else
                 if (not (KCRights(col) or QCRights(col))
                   $-2 \cdot phase$
                 else
                   if (not KCRights(col))
                     $-phase/2$
                   else
                     if (not QCRights(col))
                       $-phase/4$
                     else
                       0
                     endif
                   endif
                 endif
               endif

  $B_{col}$ :  ♯UnMoved(col,Bishop)

  $N_{col}$ :  ♯UnMoved(col,Knight)

  $Q_{col}$ :  if ($B_{col} + N_{col} \geq 2$ and ♯UnMoved(col,Queen) = 0)
                 $-phase \cdot (B_{col} + N_{col})/4$
               else
                 0
               endif
```

These development criteria encourage castling and penalise the loss of castling rights without having castled. Knights and bishops are penalised for not having moved. This has the effect of discouraging the program from moving the same piece twice in the opening before development is complete. Where the undeveloped pieces should be developed to is defined by other evaluation criteria.

The queen is discouraged from moving before at least 3 minor pieces (knights and bishops) are developed. This is an attempt to avoid positions where the program's queen is developed too early, allowing the opponent to gain a tempo by playing a developing move attacking the queen. Applying these Development criteria to the position in figure 4.2 gives:

$$\begin{aligned} DV_{White} &= 14 - 4 - 7 - 7 = -4 \\ DV_{Black} &= 14 - 0 - 0 - 0 = 14 \end{aligned}$$

4.3.3 Definition 5 : Centre Control

The importance of controlling the centre of the board cannot be too highly stressed. A clear connection between winning the game and controlling the centre has been established in [27]. However, this will come as no surprise to any strong chess players.

The centre is controlled by both occupation and attack of central squares. Therefore my centre control evaluation criteria must detect such factors and encourage their achievement.

1	1	1	1	1	1	1	1
1	2	2	2	2	2	2	1
1	2	4	4	4	4	2	1
1	2	4	8	8	4	2	1
1	2	4	8	8	4	2	1
1	2	4	4	4	4	2	1
1	2	2	2	2	2	2	1
1	1	1	1	1	1	1	1

Figure 4.3: Centre Control bonus table.

$$CC_{col} = \sum_{sq=1}^{64} V_{sq} \cdot (A_{col,sq} + O_{col,sq} + P_{col,sq})$$

where

V_{sq} : Centre Control bonus table, indexed by sq. See figure 4.3.

$A_{col,sq}$: ♯Attacks([Any],col,sq)

$O_{col,sq}$:
```
if (Occupy([Knight],col,sq))
    1
else
    0
endif
```

$P_{col,sq}$:
```
if (Attacked([Pawn],col,sq))
    1
else
    0
endif
```

Pieces are given bonuses for attacking central squares. The bonus is dependent on how close the square is to the centre. Pawn attacks receive an additional bonus to reflect the importance of such attacks. Occupation of the centre by knights is of particular importance. A knight has a relatively short range, more than any other piece it needs to be close to the field of action. A centrally placed knight is best able to exert pressure over most of the board. Therefore, central occupation by knights receives a special bonus.

I believe that centre control is important throughout the game, although its importance should diminish as the game progresses. This function will naturally play a less prominent role as pieces are exchanged because of the reduced number of attacks and possible squares occupied. Therefore, no phase adjustment was felt necessary.

Applied to the position in figure 4.2, the Centre Control criteria will produce the following values:

$CC_{White} = 169$
$CC_{Black} = 183$

4.3.4 Definition 6 : Square Control

During the game it is important to achieve a balance between attack and defence. As the game progresses this balance may shift a number of times from defence to attack and visa–versa. Square control can provide a method of measuring the extent to which the program is attacking or defending.

We can define attack as the number of squares attacked in the opponents half of the board. Defence can be defined as the number of squares attacked in ones own half of the board.

A comparison between such attack and defence for both eventual winners and losers is made in [27]. Analysing the results obtained it can be seen that throughout the game the winners and losers maintain approximately the same number of squares defended. However, there is a clear difference in the attacked squares between winners and losers as the game progresses. The best strategy, therefore, seems to be to maximise the attack while maintaining an adequate defence.

$$SC_{col} = \sum_{sq=1}^{64}(S_{col,sq} \cdot A_{col,sq})$$

where

$S_{col,sq}$: 2 tables of square values indexed by colour col and square sq. See planning section 4.11.7 for details.

$A_{col,sq}$: ♯Attacks([Any],col,sq)

I will use a table driven function. This will enable pre-loaded values derived from an initial planning phase to be used. A planning phase could use this function to good effect by identifying a basic plan, like a king side attack. The table could then be pre-loaded with values which would encourage the program to attack certain squares on the king side, for example.

Applying this Square Control criterion to the position in figure 4.2 gives:

$SC_{White} = 74$

$SC_{Black} = 62$

4.4 Pawn Evaluation

Pawns provide the basic framework around which the game will develop. The correct handling of the pawn structure is vital in high level chess.

> ”Pawns are the Soul of Chess”
>
> *Philidor*

I believe this is the one area which above all others is most neglected in chess programs. The problem is that weak pawns do not usually cause any immediate deterioration in the overall position. However, over a longer term, perhaps 30 or 40 plies, the existence of weak pawns causes numerous positional weaknesses e.g. weak squares, space to blockade, isolation etc. Often a game's result hinges on the strength or weakness of the underlying pawn structure.

It is notable in a recent [27], relatively large study of evaluation criteria, that there was no specific attention given to pawn evaluation. I feel this is a serious omission.

My pawn evaluation criteria will be amongst the most complex and comprehensive of the whole evaluation function. I feel this is critical to overall success because many of the effects caused by bad pawn play cannot be determined by search. Therefore, heuristics must fill this gap.

I will split the pawn evaluation criteria up so they can be dealt with in sufficient detail.

4.4.1 Definition 7 : Doubled Pawns

Pawns are doubled if there are at least two pawns of the same colour on the same file. Doubled pawns are generally weak. They are unable to support each other and are vulnerable to attack and blockade.

$$DP_{col} = \sum_{sq=1}^{64} \begin{array}{l} \textbf{if } (\text{Occupy}([\text{Pawn}],\text{col},\text{sq}) \textbf{ and } \text{Doubled}(\text{col},\text{OnFile}(\text{sq}))) \\ \quad D_{OnFile(sq)} \\ \textbf{else} \\ \quad 0 \\ \textbf{endif} \end{array}$$

where

D_f : $[-6, -8, -10, -14, -14, -10, -8, -6]$
Indexed by file f (1 to 8).

Doubled pawns are penalised, the size of the penalty depends on the file. Doubled pawns are worse on a centre file than on the flank, not because they are considered any weaker, but because they hinder inter–piece communication. Often there are compensations for doubled pawns, such as open files, extra centre control and square protection, for example. However, these aspects are dealt with by other evaluation criteria.

This criterion takes a simplistic view of the penalties associated with doubled pawns by applying a basic penalty. A more extensive analysis of the associated weaknesses such as vulnerability to attack, blockade etc are addressed in the general pawn weakness criteria (definition 9).

4.4.2 Definition 8 : Isolated Pawns

A pawn is isolated if there does not exist a friendly pawn on either adjacent file. Isolated pawns are weak because they cannot be supported by friendly pawns and like doubled pawns, are vulnerable to attack and blockade.

```
$IP_{col} = \sum_{sq=1}^{64}$  if (Occupy([Pawn],col,sq) and Isolated(col,OnFile(sq)))
                                   $I_{OnFile(sq)}$
                               else
                                   0
                               endif
```

where
I_f : $[-6, -10, -14, -20, -20, -14, -10, -6]$
Indexed by file f (1 to 8).

Isolated pawns are penalised depending on their centralisation. An isolated pawn on the **d** or **e** files is considered the weakest. However, an isolated centre pawn has a dynamic strength which Nimzowitsch so eloquently said was its "lust to expand". This aspect is covered by other evaluation criteria, for example, centre control. Also aspects concerning the weaknesses of isolated pawns are specifically addressed in the pawn weakness criteria below.

4.4.3 Definition 9 : Weak Pawns

Pawns are weak when they are not protected by friendly pawns and a single pawn move cannot be made which achieves this protection. Weak pawns are therefore vulnerable to attack. They constitute targets which the opponent can aim for. A backward pawn is therefore, a classic example of a weak pawn under my definition. The idea of this definition is to pick up as many cases of weak pawns as possible in one general set of criteria. Therefore isolated, doubled, backward and pawn chain bases may all be penalised for weakness.

In "My System" [43], Aron Nimzowitsch gives the following chess strategy for handling pawn chains, "We first direct our operations against the base, which we attack with a pawn, and by threats or otherwise, seek to cut off the base from its associates in the chain. This done, we turn our attention to the next opponent, namely the link which has now become the new base". This method of handling pawn chains will be encouraged by this definition because weak pawn bases are identified and progressively penalised, therefore encouraging the attacking side to build up pressure on the weak pawn.

$$WP_{col} = \sum_{sq=1}^{64} WK_{col,sq}$$

```
where
  $WK_{col,sq}$ : if (Occupy([Pawn],col,sq) and Weak(col,sq))
                    $WK_{col,sq} = W_{col,sq} - 10 \cdot \sharp$Attacks([Pawn],Opp(col),$sq_{infront}$)
                    if (SemiOpen(Opp(col),OnFile(sq))
                    and Has([Rook,Queen],Opp(col)))
                        $WK_{col,sq} = WK_{col,sq} -$ phase/2
                    endif
                    if (Attacked([Rook],Opp(col),sq))
                        $WK_{col,sq} = WK_{col,sq} - 4$
                    endif
                    if (Occupy([Any],Opp(col),$sq_{infront}$))
                        $WK_{col,sq} = WK_{col,sq} - 8$
                    endif
                  else
                    $WK_{col,sq} = 0$
                  endif

  $W_{col,sq}$ :  diff = $\sharp$DiffAttacks(Opp(col),sq)
                  if (diff $> 0$)
                    if (diff $< 5$)
                        $W_{col,sq} = [-10, -18, -26, -32]$
                            Indexed by diff (1 to 4).
                    else
                        $W_{col,sq} = -35$
                    endif
                  else
                    $W_{col,sq} = -4$
                  endif
```

Weak pawns are penalised for the severity of the attack upon them. A penalty is levied when a weak pawn is on a semi-open file and the opponent has the pieces to take advantage of this weakness. A weak pawn whose advance is restricted by enemy pawn attacks is further penalised.

The blockade of weak points form an important part of positional chess. Therefore weak pawns are penalised for being blockaded by the opponent.

4.4.4 Definition 10 : Passed Pawns

A pawn is passed when no enemy pawns on the same or adjacent files hinder the pawns advance to the promotion square. A passed pawn is of particular value, especially as the game progresses into the endgame. It not only carries the potential threat of becoming a queen, but also distracts the enemy pieces from other duties in order to restrain the passed pawn.

$PP_{col} = \sum_{sq=1}^{64} PD_{col,sq}$

```
where
$PD_{col,sq}$ :   if (Occupy([Pawn],col,sq) and Passed(col,sq))
                      if (phase > 9)
                          $PD_{col,sq} = P4_{OnRank(col,sq)}$
                      else
                          if ($Mtl_{Opp(col)}$ = KingValue)
                              if (not InsideQuadrate(Opp(col),sq))
                                  $PD_{col,sq} = P1_{OnRank(col,sq)}$
                              else
                                  $PD_{col,sq} = P2_{OnRank(col,sq)}$
                              endif
                          else
                              $PD_{col,sq} = P3_{OnRank(col,sq)}$
                          endif
                          if (InsideQuadrate(Opp(col),sq))
                              $PD_{col,sq} = PD_{col,sq} - revPhase/2$
                          endif
                          $PD_{col,sq} = PD_{col,sq} - \sharp$Occupied($sq_{infront}$,PSq(col,sq))·revPhase/4
                          if  (Attacked([Pawn],col,$sq_{infront}$)
                              or Attacked([Pawn],col,sq))
                              $PD_{col,sq} = PD_{col,sq} + UN_{OnRank(col,sq)}$
                          endif
                          if (Occupy([Any],Opp(col),$sq_{infront}$))
                              $PD_{col,sq} = PD_{col,sq} - revPhase$
                          endif
```

```
      for each square sq2 between sq and PSq(col,sq) do
         $PD_{col,sq} = PD_{col,sq} + 4 \cdot \sharp$DiffAttacks(col,sq2)
      endfor
   endif
   $PD_{col,sq} = PD_{col,sq} \cdot$ revPhase/16
else
   $PD_{col,sq} = 0$
endif
```

$P1_r$: $[0, 400, 400, 450, 515, 600, 700, 0]$
Indexed by rank r (1 to 8).

$P2_r$: $[0, 40, 40, 55, 80, 115, 155, 0]$
Indexed by rank r (1 to 8).

$P3_r$: $[0, 22, 22, 40, 65, 95, 135, 0]$
Indexed by rank r (1 to 8).

$P4_r$: $[0, 16, 16, 32, 44, 56, 68, 0]$
Indexed by rank r (1 to 8).

UN_r : $[0, 6, 6, 12, 28, 86, 120, 0]$
Indexed by rank r (1 to 8).

Using this definition, passed pawns are classified under 4 categories. This allows unstoppable passed pawns to be identified and given increased bonuses. The value of a passed pawn is dependent on its category and the obstructions between its current square and the queening square.

The passed pawn is penalised for being 'within range' of the enemy king. The blockade of a passed pawn is very effective as it is a method of mechanically stopping the pawns progress.

> "The passed pawn is a criminal who should be kept under lock and key. Mild measures, such as police surveillance are not sufficient."
>
> *Aron Nimzowitsch*

Therefore, a passed pawn is penalised for both immediate and future blockaders. This has the benefit of encouraging the opposing side to blockade along the complete path of the passed pawn. Connected or united passed pawns receive a bonus for their added strength. This bonus increases dramatically with rank. Control is encouraged over all squares between the pawn and the promotion square. The bonuses and penalties are adjusted depending on the game phase.

4.4.5 Definition 11 : Pawn Table

```
$PT_{col} = \sum_{sq=1}^{64}$   if (Occupy([Pawn],col,sq))
                                    $PST_{col,sq}$
                                else
                                    0
                                endif
```

where
$PST_{col,sq}$: Pawn Square Table. Initialised before each search.
See planning section 4.11.3 for details.

A static pawn square table will be used to store the values of each square for having a pawn occupy that square. This enables knowledge calculated at the root to be efficiently applied at leaf nodes. The table will be calculated before commencement of the search in an initial planning phase.

4.4.6 Definition 12 : Pawn Value

$$PV_{col} = DP_{col} + IP_{col} + WP_{col} + PP_{col} + PT_{col}$$

where
DP_{col} : **Doubled Pawns (Definition 7).**

IP_{col} : **Isolated Pawns (Definition 8).**

WP_{col}: **Weak Pawns (Definition 9).**

PP_{col} : **Passed Pawns (Definition 10).**

PT_{col} : **Pawn Table (Definition 11).**

All the pawn evaluation criteria used in this section can be summed to calculate a total pawn value for every pawn of a given colour.

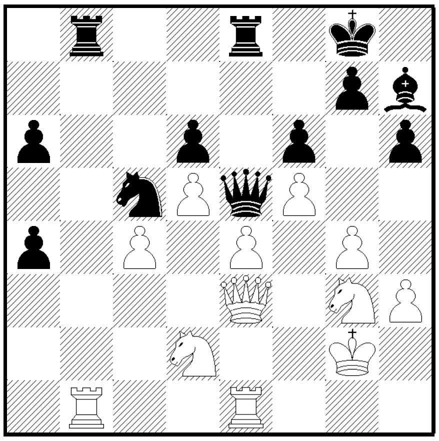


Figure 4.4: **Capablanca v Bogljuboff**, London 1922, White to Play.

Applying these pawn evaluation criteria to the position shown in figure 4.4, will produce the following results:

$$
\begin{aligned}
PV_{White} &= 0 + 0 - 74 + 0 + 118 = 44 \\
PV_{Black} &= -12 - 32 - 70 + 18 + 84 = -12
\end{aligned}
$$

4.5 Knight Evaluation

Knights are very useful pieces, particularly in the opening and middle game. Knights are especially effective in closed positions where they can 'jump' over an obstructing pawn chain. A knight becomes very strong if it reaches an advanced central post which cannot be attacked by enemy pawns. In the ending, knights are generally less effective because they only have a short range. Knights also have the ability to fork opposing forces.

4.5.1 Definition 13 : Knight Value

```
$NV_{col} = \sum_{sq=1}^{64}$  if (Occupy([Knight],col,sq))
                                   $NST_{col,sq}$
                               else
                                   0
                               endif
```

where

$NST_{col,sq}$: Knight Square Table. Initialised before each search.
See planning section 4.11.4 for details.

Knight evaluation criteria are combined in a knight square table. Concepts such as pawn holes, strong squares, centralisation and king closeness will be calculated in an initial planning phase.

It must be remembered that all evaluation criteria are intrinsically linked. Therefore, knights will be additionally evaluated by other criteria for such features as the pawn blockade, centre control, square control, ... etc,etc. Applying these knight evaluation criteria to the position in figure 4.4 gives:

$NV_{White} = 6$
$NV_{Black} = 6$

4.6 Bishop Evaluation

Bishops are effective pieces throughout the game, but especially in open positions where their scope is less hindered. A bishop is often used to pin opposing forces and attack weak points. A single bishop can only exert pressure (attack) on squares of the colour it is on. Therefore, bishops are particularly effective in pairs where they can attack both coloured squares. In the ending, bishops are as a general rule better than knights. However, there are strategies which diminish a bishops power e.g. pawn obstructions.

4.6.1 Definition 14 : Bishop Attack

```
$BA_{col} = \sum_{sq=1}^{64}$ if (Occupy([Knight,Bishop,Rook,Queen],Opp(col),sq)
                   and Attacked([Bishop],col,sq))
                       Value(sq)
                   else
                       0
                   endif
```

Bishops are given a bonus for attacking enemy pieces, the value of which depends on the type of piece attacked. This criterion will encourage the program to harass the enemy forces particularly those of high value, e.g. rooks and queens.

Note that pawns are excluded as they are more likely to obstruct the attacking bishop than be harassed themselves. See definition 18, Enemy Pawn Obstruction, for more details.

4.6.2 Definition 15 : Bishop on Kings Diagonal

```
$BK_{col} = \sum_{sq=1}^{64}$ if (Occupy([Bishop],col,sq)
                   and SameDiagonal(sq,KingSq(Opp(col))))
                       phase/2
                   else
                       0
                   endif
```

Bishops are encouraged to remain on the same diagonal as the enemy king. The bonus received decreases as the game progresses. This is an attempt to correct a weakness I feel most chess programs have. That of leaving the king on the same diagonal as the enemy bishop, particularly when it is on a long diagonal and a number of pieces are interposed.

The ideas behind such a strategy are normally more long term and therefore cannot be clearly identified by search. Against a human opponent this type of bishop placement will hopefully provide an attacking potential which disturbs the opponents liking for the position.

4.6.3 Definition 16 : Bishop Pair

```
$BP_{col} =$    if (Has([Bishop],col) $> 1$)
                    8
                else
                    0
                endif
```

Bishops work most effectively in pairs. This is because they can control both coloured squares and hence complement each other. Therefore, this heuristic rewards the player who maintains a pair of bishops.

4.6.4 Definition 17 : Horrwitz Bishops

```
$HB_{col} = \sum_{sq=1}^{64}$    if (Attacked([Bishop],col,sq) and Attacked([Bishop],col,$sq_{next}$))
                                     2
                                 else
                                     0
                                 endif
```

This criterion essentially rewards a bishop pair for working together. When two bishops work on adjacent diagonals their scope and power is greatly increased. This barrier of 'bishop power' can completely restrict an enemy king, particularly in an ending where the position is open. Therefore, a bonus is given for every pair of adjacent horizontal squares attacked by bishops.

4.6.5 Definition 18 : Enemy Pawn Obstruction

$EO_{col} = \sum_{sq=1}^{64} OB_{col,sq}$

```
where
  $OB_{col,sq}$ :   if (Occupy([Pawn],Opp(col),sq) and Attacked([Bishop],col,sq))
                       case ♯Attacks([Pawn],Opp(col),sq) of
                          0 : $OB_{col,sq} = 2$

                          1 : for each square sq1 do
                                 for each square sq2 do
                                    if (Attacking([Bishop],col,sq1,sq)
                                    and Attacking([Pawn],Opp(col),sq2,sq))
                                       if (SameDiagonal(sq1,sq2))
                                          $OB_{col,sq} = -10$
                                       else
                                          $OB_{col,sq} = -6$
                                       endif
                                    endif
                                 endfor
                              endfor

                          2 : $OB_{col,sq} = -16$
                       endcase
                    endif
```

It is important to penalise a bishop which is attacking an almost impenetrable enemy pawn wall. The degree of pawn protection provided by the opponent to the attacked enemy pawn is determined and an appropriate penalty applied. A bishop attacking an unprotected (by pawns) enemy pawn actually receives a small bonus to indicate that it may be an effective attack.

This criterion helps encourage the program to place its bishops on diagonals where they can be most effective. Hence, this in effect, complements definition 14 (Bishop Attack) which is also concerned with the effectiveness of attacks on enemy pieces by bishops.

4.6.6 Definition 19 : Bishop Table

```
$BT_{col} = \sum_{sq=1}^{64}$  if (Occupy([Bishop],col,sq))
                     $BST_{col,sq}$
                 else
                     0
                 endif
```

where

$BST_{col,sq}$: Bishop Square Table. Initialised before each search. See planning section 4.11.5 for details.

A static bishop square table will be used to store the values of each square for having a bishop occupy that square. This enables knowledge calculated at the root to be efficiently applied at leaf nodes. The table will be calculated before commencement of the search in an initial planning phase.

4.6.7 Definition 20 : Bishop Value

$$BV_{col} = BA_{col} + BK_{col} + BP_{col} + HB_{col} + EO_{col} + BT_{col}$$

where

BA_{col} : **Bishop Attack (Definition 14).**

BK_{col} : **Bishop On Kings Diagonal (Definition 15).**

BP_{col} : **Bishop Pair (Definition 16).**

HB_{col} : **Horrwitz Bishops (Definition 17).**

EO_{col} : **Enemy Pawn Obstruction (Definition 18).**

BT_{col} : **Bishop Table (Definition 19).**

All the bishop evaluation criteria defined in this section are summed to calculate the total bishop value for a given colour.

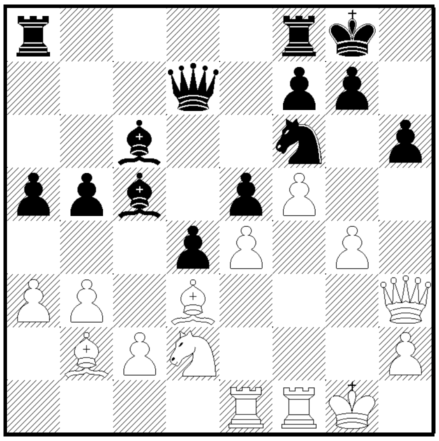


Figure 4.5: **Van Vliet v Nimzowitsch**, Ostend, 1907, Black to play.

Applying these bishop evaluation criteria to the position in figure 4.5 gives:

$$\begin{aligned} BV_{White} &= 0+0+8+4-8+18 = 22 \\ BV_{Black} &= 0+7+8+6+4+20 = 45 \end{aligned}$$

4.7 Rook Evaluation

The rook is a very strong piece. However, it requires 'room' to work most effectively.

In the opening the main objective of the rooks is to unite with one another and support more advanced pieces. This provides extra protection and diversifies the available plans for each rook.

In the middle game they should be placed on the file where a break through is likely to occur or to protect a weak or strong point. Rooks should not usually leave the confines of their own forces unless they can gain material or penetrate to the 7th rank successfully. A rook can make a heavy contribution to an attack on the enemy king provided it can be manoeuvred into an attacking position.

The endgame is where the rook truly excels. Its range allows it to exert pressure over most of the board. It can threaten enemy pawns and cause untimely pawn advances. It can blockade and obstruct enemy pawns and of course capture them. A rook is excellent at restricting the enemy kings movement thereby keeping it away from the field of action. It also has the ability to support the advance of friendly passed pawns and obstruct enemy ones.

4.7.1 Definition 21 : Rook–King Closeness

```
$RC_{col} = \sum_{sq=1}^{64}$   if (Occupy([Rook],col,sq))
                                    $-2 \cdot$KingDistance(sq,KingSq(Opp(col)))
                                else
                                    0
                                endif
```

Each rook is penalised for its distance from the enemy king. A rook on the same file (or rank) as the enemy king would not receive any penalty no matter what the corresponding rank (or file) distance is. Any penalties given will be directly related to the minimum rank or file distance. This avoids the pitfall of encouraging a rook to make weak moves in order to attain the best rank *and* file distance. For example, in figure 4.6, this criterion would **not** encourage the moves Rh4-h5 or Rg3-g4, however, it **would** encourage the move Rg3-h3.

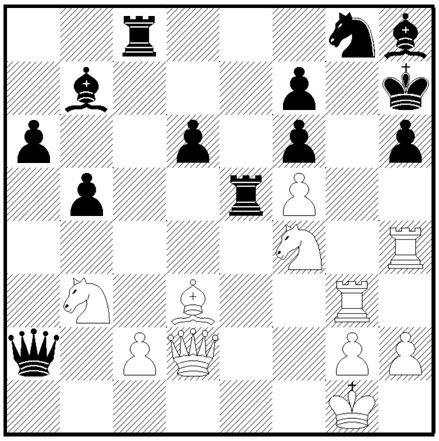


Figure 4.6: Illustration of the Rook–King Closeness Heuristic, White to Play.

4.7.2 Definition 22 : Connected Rooks

```
$CR_{col} = \sum_{sq=1}^{64}$ if (Occupy([Rook],col,sq) and Attacked([Rook],col,sq))
                                  3
                              else
                                  0
                              endif
```

Rooks are said to be connected if they defend each other. Rooks are given a bonus for being connected. This encourages rooks to double up on open files and weak points. Connected rooks are also self supporting which diversifies the options available.

4.7.3 Definition 23 : Rook on Open File

```
$RO_{col} = \sum_{sq=1}^{64}$ if (Occupy([Rook],col,sq))
    if (Open(OnFile(sq)))
        14
    else
        if (SemiOpen(col,OnFile(sq)))
            6
        else
            0
        endif
    endif
else
    0
endif
```

A file is open if no pawns of any colour exist on the file. A file is semi–open if only enemy pawns exist on the file. Rooks receive a bonus for being on an open file. Control of the open file is vital in order to be able to eventually penetrate to the 7th and 8th ranks. Even if the opponent has control of the file it is important to reward attempts which challenge this control.

This heuristic has the added benefit of encouraging the program to open files which it can control and keep closed those which would benefit the opponent.

A rook is also rewarded for being on a semi–open file. The enemy pawn which makes the file semi–open is an obvious weak point to challenge with ones own pawns. In this way the opponent can be put on the defensive and attempts can be made to build up pressure or fully open the file at an opportune moment.

4.7.4 Definition 24 : Rook on Seventh Rank

```
$RS_{col} = \sum_{sq=1}^{64}$ if (Occupy([Rook],col,sq) and OnRank(col,sq) = 7)
    if (OnRank(col,KingSq(Opp(col))) $\geq$ 7)
        14
    else
        6
    endif
else
    0
endif
```

Penetration to the 7th rank is the natural consequence and goal of rook play on open files. A rook on the 7th rank can wreak havoc on the enemy king and pawns. Therefore occupation of the 7th rank by a rook receives a bonus. This bonus is especially large if the enemy king is on the 7th or 8th ranks.

4.7.5 Definition 25 : Rook Value

$$RV_{col} = RC_{col} + CR_{col} + RO_{col} + RS_{col}$$

where

RC_{col} : **Rook–King Closeness (Definition 21).**

CR_{col} : **Connected Rooks (Definition 22).**

RO_{col} : **Rook on Open File (Definition 23).**

RS_{col} : **Rook On Seventh Rank (Definition 24).**

All the rook evaluation criteria defined in this section are summed to calculate the total rook value for a given colour.

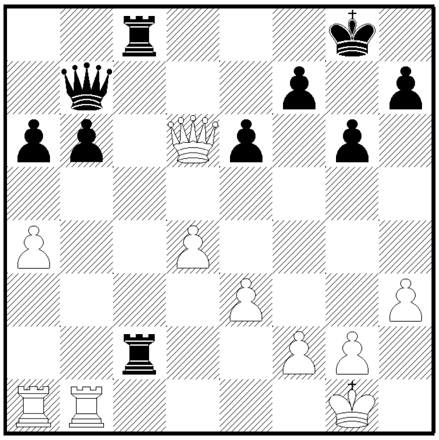


Figure 4.7: **Alatorzeff v Capablanca**, Moscow, 1935, Black to play.

Applying these rook evaluation criteria to the position in figure 4.7 gives:

$$\begin{aligned} RV_{White} &= -22 + 6 + 6 + 0 = -10 \\ RV_{Black} &= -10 + 6 + 28 + 14 = 38 \end{aligned}$$

4.8 Queen Evaluation

The queen is the most powerful and also the most valuable piece apart from the king. The queen has great range which allows it to make simultaneous multiple attacks on enemy pieces.

In the opening a queen should not be exposed to hostile attacks where a loss of tempo will occur. Therefore, like rooks it should be kept back during the opening.

In the middle game the queen can truly exert its powers. The queen can harass enemy pieces, help build up attacking formations and launch attacks on the enemy king. Once pieces begin to be exchanged the position invariably becomes clearer and allows the queen to move with more freedom.

In the ending a queen is ruthless, often picking up stray pawns and holding back enemy advances. Even in inferior endgames, the queen can offer counter chances and can sometimes be used to draw otherwise lost games by a repetition of moves.

4.8.1 Definition 26 : Queen–King Closeness

```
$QC_{col} = \sum_{sq=1}^{64}$  if (Occupy([Queen],col,sq))
                    $-2\cdot$KingDistance(sq,KingSq(Opp(col)))
                else
                    0
                endif
```

The queen is penalised for its distance from the enemy king. A queen on the same rank (or file) as the enemy king will receive no penalty regardless of the corresponding file (or rank) distance. Any penalties given will be directly proportional to the minimum rank or file distance.

This criterion therefore encourages the queen to keep close to the enemy king with respect to either the rank or file distance, not both. If a more dominating queen/king position is warranted, the king safety criteria will highlight this and encourage its achievement.

4.8.2 Definition 27 : Queen Power

```
$QP_{col} = \sum_{sq=1}^{64}$  if (Attacked([Queen],col,sq)
                and not Attacked([Pawn,Knight,Bishop,Rook],Opp(col),sq))
                    1
                else
                    0
                endif
```

As explained in the mobility definition, a straight forward measure of a queens mobility is not adequate. It is often the queens placement rather than its mobility which is important. I feel a queens real mobility is the number of squares it can actually move to without being captured by lower valued enemy pieces. Therefore, the queen is encouraged to maximise its mobility to accessible squares only. This means attacks on squares which are undefended or only defended by the enemy queen or king are worthwhile.

I feel this criterion provides a much better evaluation of a queen's degree of freedom in a given position than the more traditional mobility functions.

4.8.3 Definition 28 : Queen Value

$$QV_{col} = QC_{col} + QP_{col}$$

where

QC_{col} : **Queen–King Closeness (Definition 26).**

QP_{col} : **Queen Power (Definition 27).**

All the queen evaluation criteria defined in this section are summed in order to calculate the total queen value for a given colour.

Applying these queen evaluation criteria to the position in figure 4.7 gives:

$QV_{White} = -4 + 10 = 6$
$QV_{Black} = -10 + 13 = 3$

4.9 King Evaluation

The king is the most important piece on the board. Its loss means the end of the game.

In the opening, the king must seek a save haven protected by its own pawns and pieces. This is generally best accomplished by castling.

In the middle game, the king must still be protected against attack. This protection will involve a strong pawn structure in front of the king with piece cover for any weak points.

In the endgame, unlike other stages of the game, the king is a strong attacking piece. A king can support its own pieces and pawns or penetrate a weak enemy pawn structure to win material. Late activation of a king in the endgame could be disastrous. Therefore, the king must be suitably positioned to take advantage of any weaknesses in the opposing forces.

4.9.1 Definition 29 : King On Open Files

$KO_{col} = \sum_{sq=1}^{64} KF_{col,sq}$

```
where
  $KF_{col,sq}$ : if (Occupy([King],col,sq) and Has([Rook,Queen],Opp(col)))
      if (Open(OnFile(sq)))
          $KF_{col,sq} = -2 \cdot phase$
      else
          if (SemiOpen(Opp(col),OnFile(sq))
          or SemiOpen(col,OnFile(sq)))
              $KF_{col,sq} = -phase$
          else
              $KF_{col,sq} = 0$
          endif
      endif
      if (OnFile(sq) = 2 or 3 or 4 or 8)
          if (Open(OnFile(sq)-1))
              $KF_{col,sq} = -phase$
          else
              if (SemiOpen(Opp(col),OnFile(sq)-1))
                  $KF_{col,sq} = -phase/2$
              endif
          endif
```

```
        else
            if (Open(OnFile(sq)+1))
                $KF_{col,sq} = -phase$
            else
                if (SemiOpen(Opp(col),OnFile(sq)+1))
                    $KF_{col,sq} = -phase/2$
                endif
            endif
        endif
    else
        $KF_{col,sq} = 0$
    endif
```

The king is penalised for being on either an open or semi–open file. It is also penalised for being adjacent to an open or semi–open file under two conditions:

1. The king is **not** on either the 'A' or 'H' files and the open file is on the adjacent file nearest the edge of the board.
2. The king is on either the 'A' or 'H' files and the open file is adjacent.

Note these penalties are only imposed if the opponent has the necessary pieces to take advantage of the weakness.

4.9.2 Definition 30 : Weak Squares around King

Control must be held over the weak squares around the king. By weak squares I mean those squares not covered by friendly pawns which are therefore, possible pawn holes. These pawn holes are ideal weak points for the opponent to attack and occupy.

```
$WS_{col} = WK_{col,CentralPoint(KingSq(col))}$

where
  $WK_{col,csq}$ : $\sum_{sq=1}^{64}$ if (Distance(sq,csq) $\leq$ 1 and
                   not (Attacked([Pawn],col,sq) or Occupy([Pawn],col,sq)))
                       $CO_{col,sq} + OC_{col,sq} + FB_{col,sq} + EB_{col,sq}$
                   else
                       0
                   endif
```

```
$CO_{col,sq}$ :   if (Control(sq) = col)
                      $-1$
                  else
                      $-4$
                  endif

$OC_{col,sq}$ :   if (Occupy([Any],Opp(col),sq))
                      $-8$
                  else
                      0
                  endif

$FB_{col,sq}$ :   if (Attacked([Bishop],col,sq))
                      6
                  else
                      if (Occupy([Bishop],col,sq))
                          16
                      else
                          0
                      endif
                  endif

$EB_{col,sq}$ :   if (Attacked([Bishop,Queen],Opp(col),sq))
                      $-6$
                  else
                      0
                  endif
```

Squares on or adjacent to the central point of the king's defensive position are penalised for being weak if they are not occupied or attacked by friendly pawns. Squares are quite heavily penalised for not being controlled by friendly pieces. Additionally, an occupied weak square is heavily penalised. A bonus is received if a friendly bishop attacks the weak square or occupies the actual square itself. A penalty is levied if the opponent has a bishop or queen which can infiltrate the weak square.

These criteria will encourage the program to force weak squares near the opponents king and realise which of its pieces are therefore made more valuable. It will also encourage vital occupation and blockade of these weak points. Of course, the program will be discouraged from allowing such weak squares to develop near its own king. Even if it is forced to accept some weak squares, it will actively seek to exchange the opponents most dangerous pieces with respect to those weaknesses.

4.9.3 Definition 31 : Mating Net

The kings position should always be checked to see if a mating net is being formed around it. The building of such a mating net around the opponents king is to be encouraged. It will, however, be discouraged from allowing the opponent to achieve similar aims.

$MN_{col} = NT_{col}$

```
where
  $NT_{col}$ :  WeakSqs = 0
               $NT_{col} = 0$
               for each square sq do
                   if (Distance(KingSq(col),sq) = 1)
                       diff = ♯DiffAttacks(Opp(col),sq)
                       if (Control(sq) = Opp(col) or diff > 0)
                           WeakSqs = WeakSqs + 1
                           if (♯Attacks([Any],Opp(col),sq) > 1)
                               $N_{col} = N_{col} - 4 \cdot diff - 2$
                           endif
                       endif
                   endif
               endfor
               if (KingSq(col) = a8 or h8 or a1 or h1)
                   WeakSqs = WeakSqs + 1
               endif
               $N_{col} = N_{col} - T_{WeakSqs}$

  $T_c$ :      [0, 8, 20, 36, 56, 80, 108, 140, 180]
                   Indexed by weak square count c (0 to 8)
```

Any squares adjacent to the king are penalised for lack of control. Particularly if the opponent has two or more pieces interworking against the square.

Notice the penalty for the number of weak squares which may form a mating net rises dramatically, the larger the number of such squares. The king is additionally penalised for being on a corner square. This is because the potential number of escape squares for the king from a corner square are severely reduced. This would not be picked up by the general weak square count previously calculated. Therefore an adjustment is made to allow for this possibility.

4.9.4 Definition 32 : Check Threats

The kings position must be monitored and the potential threats of any checks assessed. Check moves can be very effective; the value of which can be anything from a gain in tempo to checkmate itself. Of course some check moves may also be inferior for the side giving the check, possibly resulting in the loss of a tempo or worse still the checking piece. A well known chess saying states:

> "Patzer sees check; Patzer gives check."

Indicating that it is not always wise to put your opponent in check. In fact, often the threat is stronger than the execution. This criterion must therefore determine the relative merits of each check threat and decide what penalties (if any) to assign.

```
$CT_{col} = TS_{col,KingSq(col)}$

where
  $TS_{col,ksq}$ :TCheck = 0
        $TS_{col} = 0$
        for each square sq do
            for each square sq1 do
                if (PotentialCheck(Opp(col),sq,sq1))
                    if (♯DiffAttacks(Opp(col),sq1) > 0)
                        TCheck = TCheck + 1
                        if (♯SafeBlocks(col,ksq,sq1) = 0)
                            if (not Escape(col,sq,sq1))
                                $MateThreat_{col} = \text{true}$
                                $TS_{col} = TS_{col} - 128$
                            else
                                $TS_{col} = TS_{col} - 16$
                            endif
                        else
                            $TS_{col} = TS_{col} - (6 - \sharp SafeBlocks(col, ksq, sq1))$
                        endif
                    else
                        $TS_{col} = TS_{col} - 2$
                    endif
                endif
            endfor
        endfor
        $TS_{col} = TS_{col} - KT_{TCheck}$
```

KT_n : [0,14,30,48,70,86,100,110,120,120,120,120,120,120,120,120]
Indexed by number of likely checks n (0 to 15).

Penalties are given for potential checks if the side giving check has enough control over the checking square. Checks are considered more serious if there are no safe blocking moves available for the defender. In fact, if there is also no escape square for the checked king, the situation is considered very serious indeed. The *MateThreat* predicate is set to indicate the severity of the situation. Hence, increasing the number of safe blocks is rewarded as is creating escape squares for the threatened king.

These criteria, therefore, encourage the program to maintain a king position free from checks and to provide enough potential piece cover to parry any checks received.

4.9.5 Definition 33 : King Table

```
$KT_{col} = \sum_{sq=1}^{64}$  if (Occupy([King],col,sq))
                                   $KST_{col,sq}$
                               else
                                   0
                               endif
```

where

$KST_{col,sq}$: King Square Table. Initialised before each search. See planning section 4.11.6 for details.

A static king square table will be used to store the values of each square for having a king occupy that square. This enables knowledge calculated at the root to be efficiently applied at leaf nodes. The table will be calculated before commencement of the search in an initial planning phase.

4.9.6 Definition 34 : King Value

$$KV_{col} = (KO_{col} + WS_{col} + MN_{col} + CT_{col}) \cdot phase/16 + KT_{col}$$

where

KO_{col} : **King On Open Files (Definition 29).**

WS_{col} : **Weak Squares around King (Definition 30).**

MN_{col} : **Mating Net (Definition 31).**

CT_{col} : **Check Threats (Definition 32).**

KT_{col} : **King Table (Definition 33).**

All the king evaluation criteria defined in this section are summed in order to calculate the total king value for a given colour. The values are further adjusted by the game phase to take account of a kings more active role as the endgame approaches. Therefore, the first four features, which are essentially concerned with king safety, are reduced in importance as the game progresses. The king table, which already has information relative to the kings latest role, will not be adjusted in this way.

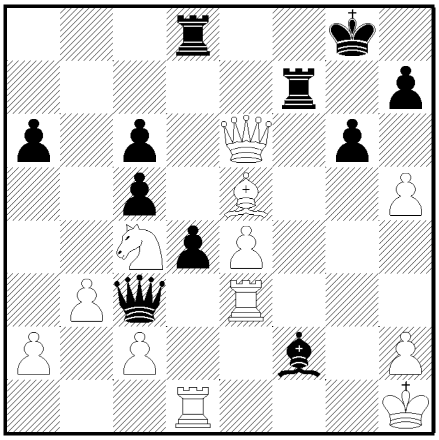


Figure 4.8: **Capablance v Marshall**, New York, 1909, Black to play.

Applying these king evaluation criteria to the position in figure 4.8 gives:

$$KV_{White} = (-11 - 27 - 20 - 2) \cdot 11/16 - 1 = -42$$
$$KV_{Black} = (-5 - 43 - 36 - 34) \cdot 11/16 + 0 = -81$$

4.10 Static Tactical Evaluation

The following criteria attempt to evaluate tactical aspects statically from within the evaluation function itself. The objective of these criteria is to derive an accurate value where possible, and where uncertainty exists to give an estimate of the likely value and recommend a deeper search to resolve the tactical uncertainty. This requires the criteria to detect when a position is tactically active.

4.10.1 Definition 35 : Pinned Pieces

A pin exists if a piece exerts a hidden (X-ray) attack through an enemy piece onto another enemy piece which is either undefended or of greater value than the pinning piece.

A pin severely restricts the movement of the enemy pieces involved in the pin. A pin against the king is the most restrictive type of pin because the movement of the pinned piece is illegal in all but the pin direction and anti–direction. Therefore, a pinned piece's defensive and attacking capabilities are either severely diminished or completely lost.

$$PN_{col} = PS_{col}$$

```
where
  $PS_{col}$ : $PS_{col} = 0$
          for each square sq do
             for each square sq1 do
                for each square sq2 do
                   if (Pin(col,sq,sq1,sq2))
                      if (Value(sq2) > Value(sq)
                      or not Attacked([Any],Opp(col),sq2))
                         $PS_{col}$ = Value(sq2) + Value(sq1) − Value(sq) + 10
                         if (Attacked([Any < On(sq1)],col,sq1)
                         or not Attacked([Any],Opp(col),sq1)
                         or (♯DiffAttacks(col,sq1) ≥ 0 and Control(sq1) = col))
                            $Threat_{col} = Threat_{col} + 1$
                            $PS_{col} = PS_{col} + revPhase$
                         endif
                      endif
                   endif
                endfor
             endfor
          endfor
```

It is important to be able to identify and assess the value of pins. This definition gives a bonus for pinning an opponents piece. Pins are tactically complex. Therefore, this criterion attempts to assess those aspects which can be statically valued. If the criterion assesses the pin as a tactically dynamic threat, then a tactical count indicating the uncertainty of the static evaluation is incremented. Basically, this indicates that although a static value is calculated, the tactical uncertainty should be resolved by extending the search until a more quiescent position is reached. See chapter 5 for more details.

The evaluation function does not know whether a deeper search will be successful in finding a quiescent position or even has enough time to undertake such a task. Therefore, a value indicating its liking for the tactical uncertainty is included in the overall static value.

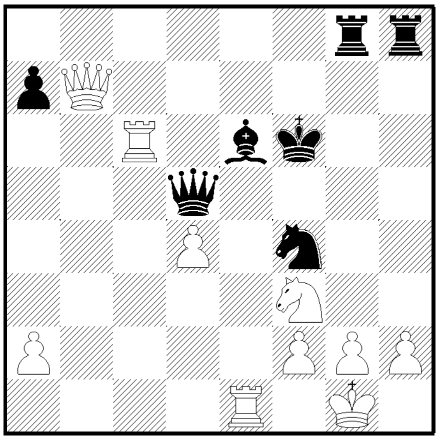


Figure 4.9: **Dyatlov v Shashin**, Leningrad, 1962, Black to play.

Applying the Pinned Pieces evaluation criteria to the above position gives:

$PN_{White} = 22$
$PN_{Black} = 15$

4.10.2 Definition 36 : Attacked Pieces

$AP_{col} = \sum_{sq=1}^{64} AT_{col,sq}$

```
where
  $AT_{col,sq}$ :  if (Occupy([Pawn,Knight,Bishop,Rook,Queen],col,sq)
                 and Attacked([Any],Opp(col),sq))
                     if (not Attacked([Any],col,sq)
                     or Attacked([Any<On(sq)],Opp(col),sq)
                     or (Control(sq) = Opp(col) and ♯DiffAttacks(Opp(col),sq) > 0))
                         $AT_{col,sq} = AT_{col,sq} - 8 -$ Value(sq)
                         $Threat_{col} = Threat_{col} + 1$
                         if (Trapped(col,sq))
                             $AT_{col,sq} = AT_{col,sq} - Value(sq) \cdot$ revPhase
                             $Threat_{col} = Threat_{col} + 1$
                         endif
                     else
                         if (Control(sq) = Opp(col) or ♯DiffAttacks(Opp(col),sq) > 0)
                             $AT_{col,sq} = AT_{col,sq} -$Value(sq)
                         endif
                     endif
                 endif
```

Pieces which are threatened with profitable capture (en-prise) by the opposing side are penalised. This penalty is levied not because the loss of the piece is considered likely, but because it is very likely that a tempo will be wasted in protecting or moving the threatened piece. The threat count is incremented for every piece en-prise.

Trapped pieces which are also en-prise are additionally penalised and the threat count is further incremented. Pieces which are under pressure, but not en-prise are also penalised although to a lesser extent. The threat count is used to extend the search in positions where an accurate static evaluation value cannot be calculated.

Applying the Attacked Pieces criteria to the position in figure 4.9 gives:

$AP_{White} = -18$
$AP_{Black} = -9$
$Threat_{White} = 2$
$Threat_{Black} = 1$

The number of threats against white (2), represents a degree of uncertainty about the tactical possibilities threatened against the white pieces.

4.11 Planning

Before each search begins an analysis of the position will be done. This will enable various piece square tables to be loaded with values which can be applied at leaf nodes using look up tables.

Obviously this is much faster than repeatedly calculating information at leaf nodes. Hence, in an implementation, all leaf node evaluation criteria which are essentially static should be incorporated into these tables and removed from the leaf evaluation process. However, this chapter represents a definition of my proposed evaluation function regardless of implementation detail. Therefore, some criteria which are suited to piece square tables are defined separately because of their importance.

To combat the typically aimless play of chess programs I will include some rudimentary planning in the evaluation function. Part of the initial analysis done before each search commences will include a selection of an appropriate plan for the current position. The plans which will be recognised are:

1. **King side attack with pawns**.
2. **King side attack with pieces**.
3. **King side attack with pieces and pawns**.
4. **Queen side attack with pawns**.
5. **Queen side attack with pieces**.
6. **Queen side attack with pieces and pawns**.
7. **Using pawns as levers**.
8. **Endgame, king and pawn co-ordination**.

These plans are relatively general, but should provide the program with enough variety to choose a plan relevant to the position and attempt to co-ordinate its moves to achieve common goals. It is not the object of these plans to suggest specific moves, but rather to allow the search to be guided toward the goals identified.

To enable the search to be guided, the appropriate piece square tables will be updated in such a way that moves which contribute to the achievement of the chosen goals will be encouraged. This will hopefully have the effect of providing a common purpose to the programs moves rather than allowing it to play aimless, although solid moves.

There are no specific opening plans, however, the evaluation function does provide ample criteria such as Development, Centre Control etc, to encourage good opening play.

The plans available in the middle game are attack oriented. However, because appropriate plans are selected for both sides this will have the effect of making each side attempt to obstruct the other sides plan. This will obviously include both defensive and aggressive measures.

In the endgame, only a general approach to king–pawn co-ordination will be taken. The endgame is a stage of the game where specific knowledge and plans would greatly help in the handling of certain types of endings which have more discernible goals. However, such a comprehensive approach could form the basis for an entire project in itself.

4.11.1 Definition 37 : Pressure on King

$$PK_{col} = (\sum_{sq=1}^{64} CL_{col,sq}) + AD_{col,OnFile(KingSq(col))}$$

where

```
$CL_{col,sq}$ :  if (Taxicab(KingSq(col),sq) $\leq 4$
                 and | OnFile(KingSq(col)) $-$ OnFile(sq) | $\leq 2$)
                     $AT_{Opp(col),sq} + PM_{Opp(col),sq} - DF_{col,sq}$
                 else
                     0
                 endif

$AT_{col,sq}$ :  $\sharp$Attacks([Rook],col,sq) +
                 $2 \cdot \sharp$Attacks([Pawn,Knight,Bishop],col,sq) +
                 $3 \cdot \sharp$Attacks([Queen],col,sq)

$PM_{col,sq}$ :  $\sum_{sq2=1}^{64}$  if (Attacking([Pawn],col,sq2,sq)
                            and $\sharp$Moves([Pawn],col,sq2) $> 0$)
                                2
                            else
                                0
                            endif

$DF_{col,sq}$ :  $\sharp$Attacks([Rook],col,sq) +
                 $2 \cdot \sharp$Attacks([Knight,Bishop,Queen],col,sq) +
                 $3 \cdot \sharp$Attacks([Pawn],col,sq)
```

```
$AD_{col,kf}$ : $\sum_{f=kf-1}^{kf+1}$ if (Open(f) or SemiOpen(Opp(col),f))
                    3
                else
                    0
                endif
```

This definition attempts to calculate how much pressure in terms of the attack/defence balance is currently exerted on the king in question. The higher the value of PK, the more pressure is thought to exist. This definition will form the basis of a decision as to whether an assault against the king is likely to be worthwhile.

This definition takes into account attacks by both sides on squares within a 4 square, 2 file boundary of each king. The value of the attack depends on the type of pieces involved. The defensive value is calculated in a similar way, except the relative value of the attacks by pieces are different. For example, a pawn is considered to perform a better defensive role than an attacking one.

Mobile pawns in attack are considered superior to blocked pawns. Also, an allowance is made for open or semi–open files available to the attacking side because this obviously improves the likelihood of an attack on the king succeeding.

4.11.2 Definition 38 : Plan Selection

```
$PS_{col}$ =    if (phase $<$ 8)
                    if ($\sharp$Pawns $>$ 0)
                        $PlanType_{col}$ = EndGameKP
                        $PlanWing_{col}$ = QueenSide $\cup$ KingSide
                    endif
                else
                    if ($PK_{Opp(col)}$ $>$ phase)
                        if ($PK_{Opp(col)}$ $>$ phase+3 or OppositeFlanks)
                            $PlanType_{col}$ = PawnAttack $\cup$ PieceAttack
                        else
                            $PlanType_{col}$ = PieceAttack
                        endif
                        $PlanWing_{col}$ = Wing(KingSq(Opp(col)))
                    else
                        if (OppositeFlanks)
                            $PlanType_{col}$ = PawnAttack
                            $PlanWing_{col}$ = Wing(KingSq(Opp(col)))
                        else
```

```
            if (phase $> 10$)
                $PlanType_{col}$ = LeverPawns
                $PlanWing_{col}$ = QueenSide $\cup$ KingSide
            else
                $PlanType_{col}$ = None
            endif
        endif
    endif
endif
```

This definition attempts to select the best plan at its disposal based on the known information. It may be the case that no plan is clear, in this case no plan will be selected.

In the endgame only one plan is known, that of king–pawn co-ordination. In the opening or middle game, if the attack–defence balance (pressure) on the opponents king appears to offer some likelihood of success, then a piece attack plan is considered suitable. If the pressure is even greater, a pawn assault plan is also selected. The idea here is to only use pawns when there is enough piece coverage to make their use effective.

Pressure is in fact a measure of the balance between attack and defence, therefore a king assault plan may even be considered if no relevant attacks currently exist. This of course depends on the enemy defence being weak. Hence, once a weakness is detected by the program it may switch its attention to a king assault. Hopefully this will help the program to exhibit a more human style of play.

Additionally, if the kings have castled on opposite flanks, a plan to encourage an assault by pawns will be selected.

If none of these plans are selected, a general goal of using pawn levers may be used. The idea here is to use pawns as levers against the enemy pawn structure, therefore, creating dynamic type positions which are best suited to a programs tactical ability.

4.11.3 Definition 39 : Pawn Square Table

0	0	0	0	0	0	0	0
20	20	24	28	28	24	20	20
16	16	20	24	24	20	16	16
12	12	16	20	20	16	12	12
8	8	12	16	16	12	8	8
6	6	8	12	12	8	6	6
4	4	6	8	8	6	4	4
0	0	0	0	0	0	0	0

Figure 4.10: Initial Weightings for Pawn Square Table.

```
$PST_{col,sq}$ :   for each square sq do
                       $PST_{col,sq} = ST_{col,sq} + KS_{col,sq} + PA_{col,sq} + PL_{col,sq}$
                   endfor

where
  $ST_{col,sq}$ :  Static weightings as defined in figure 4.10.

  $KS_{col,sq}$ :  if (♯UnMoved(col,King) = 0)
                       if (KingDistance(KingSq(col),sq) < 2)
                           case Distance(KingSq(col),sq) of
                               1 : phase
                               2 : phase/2
                           else
                               0
                           endcase
                       else
                           0
                       endif
                   else
                       if ((OnFile(sq) = 1 or 2 or 7 or 8) and OnRank(col,sq) $\leq$ 3)
                           phase / OnRank(col,sq)
                       else
                           0
                       endif
```

```
                endif

$PA_{col,sq}$ : if ((PawnAttack $\in PlanType_{col}$ and OnWing($PlanWing_{col}$,sq))
                and PawnLever(OnFile(sq),col))
                    $40 - 8\cdot$Taxicab(LeverSq(OnFile(sq),col),sq)
                else
                    0
                endif

$PL_{col,sq}$ : if (LeverPawns $\in PlanType_{col}$ and PawnLever(OnFile(sq),col))
                    $10 - 2\cdot$Taxicab(LeverSq(OnFile(sq),col),sq)
                else
                    0
                endif
```

A general table of values for pawn occupation is used. Once a king has moved, the pawns near the king are discouraged from moving as this would weaken the pawn shield around the king. Before the king first moves, the pawns on either flank are discouraged from moving as they are likely to form the future pawn shield. As the game progresses these restrictions are reduced, thereby encouraging pawn advances. Centre pawns are free from such restrictions.

A pawn attack plan is implemented by encouraging pawn moves up the board which would result in pawn leverage. The bonus given depends on how close to the lever square the pawn is.

A general pawn lever plan is actioned in a similar manor to the pawn attack plan above, except the bonuses used are significantly lower.

Remember, these values are for possible pawn occupation. Even though no pawns exist on the file these values may still be important because a pawn may capture onto the said file during the search and should therefore incur the relevant bonus or penalty.

4.11.4 Definition 40 : Knight Square Table

2	3	4	4	4	4	3	2
3	4	6	6	6	6	4	3
4	6	8	8	8	8	6	4
4	6	8	10	10	8	6	4
4	6	8	10	10	8	6	4
4	6	8	8	8	8	6	4
3	4	6	6	6	6	4	3
2	3	4	4	4	4	3	2

Figure 4.11: Initial Weightings for Knight Square Table.

```
$NST_{col,sq}$ :  for each square sq do
                      $NST_{col,sq} = ST_{col,sq} + PH_{col,sq} + KC_{col,sq}$
                  endfor

where
  $ST_{col,sq}$ : Static weightings as defined in figure 4.11.

  $PH_{col,sq}$ : if (PawnHole(sq,Opp(col)))
                      phase
                  else
                      0
                  endif

  $KC_{col,sq}$ : $-$Taxicab(KingSq(col),sq) $-$ Taxicab(KingSq(Opp(col)),sq)
```

Knights are encouraged to be on squares where they have as many options as possible. They are also encouraged to occupy pawn holes, the bonus being dependent on the game phase. Knights are penalised for their distance from each king. This encourages a knight to maintain a position from where it is poised to attack the enemy king or protect its own king as the need arises.

4.11.5 Definition 41 : Bishop Square Table

0	2	2	2	2	2	2	0
2	8	8	8	8	8	8	2
2	8	10	10	10	10	8	2
2	8	10	12	12	10	8	2
2	8	10	12	12	10	8	2
2	8	10	10	10	10	8	2
2	8	8	8	8	8	8	2
0	2	2	2	2	2	2	0

Figure 4.12: Initial Weightings for Bishop Square Table.

```
$BST_{col,sq}$ :   for each square sq do
                       $BST_{col,sq} = ST_{col,sq} + PH_{col,sq}$
                   endfor

where
  $ST_{col,sq}$ :  Static weightings as defined in figure 4.12.

  $PH_{col,sq}$ :  if (PawnHole(sq,Opp(col)))
                       phase/2
                   else
                       0
                   endif
```

Bishops are encouraged to keep on long diagonals and have power in as many directions as possible. A static table provides appropriate initial bonuses. Bishops are additionally encouraged to occupy pawn holes, the bonus being dependent on the game phase.

4.11.6 Definition 42 : King Square Table

−2	+0	−4	−8	−8	−4	+0	−2
−2	−2	−6	−8	−8	−6	−2	−2
−16	−16	−18	−20	−20	−18	−16	−16
−18	−18	−22	−26	−26	−22	−18	−18
−18	−18	−22	−26	−26	−22	−18	−18
−16	−16	−18	−20	−20	−18	−16	−16
−2	−2	−6	−8	−8	−6	−2	−2
−2	+0	−4	−8	−8	−4	+0	−2

Figure 4.13: Initial Weightings for King Square Table.

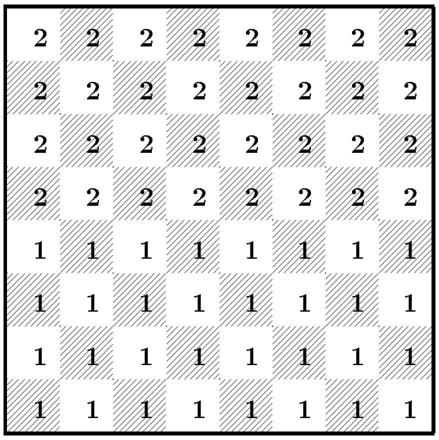
```
$KST_{col,sq}$ :  for each square sq do
                      $KST_{col,sq} = ST_{col,sq} + KE_{col,sq} - PC_{col,sq}$
                  endfor

where
  $ST_{col,sq}$ :  Static weightings as defined in figure 4.13.

  $KE_{col,sq}$ :  $(-ST_{col,sq})\cdot$ revPhase/8

  $PC_{col,sq}$ :  if (EndgameKP $\in$ $PlanType_{col}$)
                      for each square sq2 do
                          if (Occupy([Pawn],col,sq))
                              if (Passed(col,sq2))
                                  PawnMass = PawnMass + 4 $\cdot$ Taxicab(sq,sq2)
                                  PawnMass = PawnMass + Taxicab(sq,PSq(col,sq2))
                              else
                                  PawnMass = PawnMass + Taxicab(sq,sq2)
                              endif
                          else
                              if (Occupy([Pawn],Opp(col),sq))
                                  if (Passed(Opp(col),sq2))
                                      PawnMass = PawnMass + 4 $\cdot$ Taxicab(sq,sq2)
                                      PawnMass = PawnMass + Taxicab(sq,PSq(col,sq2))
                                  else
```

```
                    PawnMass = PawnMass + Taxicab(sq,sq2)
                endif
            endif
        endif
    endfor
    $PC_{col,sq} = (\text{PawnMass}/\sharp\text{Pawns} - 8) \cdot (\text{revPhase} - 8)$
else
    0
endif
```

At the beginning of the game the king is encouraged to seek a safe haven near one of the corner squares. As the game progresses the king is encouraged to play a more active role. Eventually, the king will receive more bonus points for being in the centre than near the edge of the board. However, this change will be gradual as it is related to the game phase.

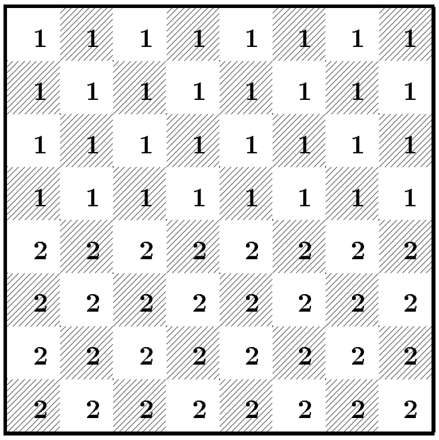
When in the endgame, the king gets bonuses for maintaining an optimum distance from all the pawns with particular emphasis on passed pawns.

4.11.7 Definition 43 : Square Control Table



2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

Figure 4.14: Initial Weightings for White's Square Control Table.



1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2

Figure 4.15: Initial Weightings for Black's Square Control Table.

```
$S_{col,sq}$ :    for each square sq do
                      $S_{col,sq} = ST_{col,sq} + PL_{col,sq}$
                  endfor
```

where

$ST_{col,sq}$: Static weightings indexed by colour and square, see figures

4.14 and 4.15

```
$PL_{col,sq}$ :  if (PieceAttack $\in PlanType_{col}$ and OnWing($PlanWing_{col}$,sq)
                 and Side(sq) = Opp(col))
                     2
                 else
                     0
                 endif
```

This table will be used by the evaluation function criterion, Square Control, to help direct attacks. The table is initially set up to encourage attacks in the enemy half of the board. This is further modified depending on the plan selected to further encourage attacks on the planned attack wing.

4.12 Definition 44 : Total Evaluation Function Value

$$\begin{aligned} TEFV_{col} = \quad & MT_{col} + MB_{col} + DV_{col} + CC_{col} + SC_{col} + PV_{col} + \\ & NV_{col} + BV_{col} + RV_{col} + QV_{col} + KV_{col} + PN_{col} + AP_{col} \end{aligned}$$

where

MT_{col} : **Material (Definition 1).**

MB_{col}: **Mobility (Definition 3).**

DV_{col} : **Development (Definition 4).**

CC_{col} : **Centre Control (Definition 5).**

SC_{col} : **Square Control (Definition 6).**

PV_{col} : **Pawn Value (Definition 12).**

NV_{col} : **Knight Value (Definition 13).**

BV_{col} : **Bishop Value (Definition 20).**

RV_{col} : **Rook Value (Definition 25).**

QV_{col} : **Queen Value (Definition 28).**

KV_{col} : **King Value (Definition 34).**

PN_{col} : **Pinned Pieces (Definition 35).**

AP_{col} : **Attacked Pieces (Definition 36).**

The evaluation function will return a single value for the position. To achieve this, all the evaluation criteria must be summed. The position value from either sides point of view can then be easily calculated, e.g.:

$$V_{White} = TEFV_{White} - TEFV_{Black}$$

A very important decision is when to apply the evaluation function to a position. After a lot of deliberation, I decided the evaluation function will be applied at leaf nodes of the main search in all cases except:

- It is never applied if one side is in-check.
- It is not applied if the deficit between α and the current material value is larger than the greatest positional value so far encountered in the search.

The first condition avoids application to positions which are clearly not quiescent. The second condition attempts to only apply the evaluation function to positions which have a reasonable chance of affecting the search result.

Chapter 5

Tree Searching

The evaluation function contains criteria to assess positional and tactical values. The tactical evaluation is however, far from complete in terms of deriving an accurate tactical value. The tactical part of the evaluation function therefore, only attempts to give a rough estimate of the tactical possibilities. It takes particular account of uncertainties in the position. For example, a position where two pieces are simultaneously en-prise would constitute a degree of uncertainty about the true value of the position.

A swap off algorithm [37] could have been additionally used to improve the accuracy of this tactical value. However, this would still not take account of pinned pieces or exchanges which expose hidden attacks on other squares. The added complexity involved in attempting to correct these faults would be very expensive in terms of processing time. Therefore, I feel such analysis is better left to a dynamic search. Using various additional tree pruning mechanisms, I hope to show that this search can be done efficiently.

Hence, it is important to support the evaluation function by the tree searching of possibilities. This will enable the program to explore possible move sequences and therefore chose moves which lead to favourable positional and tactical outcomes.

In this chapter, I will explain what searching schemes will be used and how I intend to complement the evaluation function by the use of a quiescence search.

The overall search procedure is as follows: The main search will progress iteratively to deeper levels of the search tree. Selected, more interesting lines will be extended. At the leaf nodes of this main search tree the evaluation function will be applied. The evaluation function may then advise the search to take one of two possible actions:

1. The position is suitable for the quiescence search to be applied. A quiescence search will then be used to explore any dynamic tactical possibilities which need resolving. This will allow a more accurate tactical value to be derived at the leaf node than the value calculated purely in the evaluation function. The main search can therefore have higher confidence in the final leaf node value calculated.
2. The position is considered to be too unstable for the quiescence search to be applied accurately. This means the main search should be extended in order to resolve the uncertainty. This 'Quiescence Decision' may also cause the main search to be re–entered from nodes that are already part of the quiescence search.

The results obtained from searching each node is then passed back up the tree where search heuristics can make use of the information in searching other nodes. It is important to realise the search tree explored by such a search will be of variable depth, the deepest lines being those of most interest to the search. This shows some similarities to the way humans approach the task of tree searching.

The following sections explain this search process in more detail.

5.1 Main Search

The main search will use the alpha–beta ($\alpha\beta$) tree pruning algorithm on negamax trees. This algorithm has many derivatives and enhancements, some of which were explained in chapter 3.

At first sight, the principle variation $\alpha\beta$ search (PVS) appears to be the best algorithm to select as it typically visits fewer nodes than other $\alpha\beta$ derivatives [40]. However, I intend to use search extensions which actually use the values of the current $\alpha\beta$ window as part of their search extension decision. This breaks one of the major principles of the $\alpha\beta$ algorithm :- the value of a move is independent of the $\alpha\beta$ window used to search it.

The PV search uses a minimal window as its basis. When the search of a subtree fails high (score $\geq \beta$), it is re-searched using a suitably adjusted window. When the value is not dependent on the window used, everything works fine. However, if the value is dependent, a situation could arise where a move found which fails high, is re-searched with an enlarged window, only to fail low. This violates the PVS's ability to search minimal trees for all moves except those which constitute a new principle variation. This can cause serious problems not only for the efficiency of the PVS, but also for its correctness.

I also wish to use the current search window as part of a decision as to whether to apply the evaluation function at a given node in the tree. After all, if the deficit between the required score ($\geq \alpha$) necessary to effect the search result and the current material value is so large that even the maximum positional value cannot alter α, then it would constitute wasted effort to apply the evaluation function. This approach could also adversely affect the PV search.

These problems lead me to prefer an aspiration $\alpha\beta$ search. Although this search algorithm may still give rise to some of the problems already explained, they will be to a far lesser extent. The important aspect about this algorithm as far as search extensions are concerned, is that the $\alpha\beta$ values of the current window constitute the true best score attained by each side. The only exception being when one (or both) of the initial aspiration $\alpha\beta$ bounds are still being used. Even in this case the initial aspiration window can be saved and taken into account by the search extension heuristics.

In the past, I have observed many commercial chess programs which undoubtedly use an aspiration $\alpha\beta$ search. I have also noticed in some positions, a particular search to depth n takes considerably longer than expected. This is because of a problem which I believe can be solved by an enhancement to the aspiration $\alpha\beta$ algorithm. The problem is as follows: A search to depth $n-1$ derives a value for the best move found. This value then forms the basis by which the new $\alpha\beta$ window is set for the next iteration. A search to the next iteration is then performed. If this search discovers something which refutes its previous analysis (because of the horizon effect), each and every move may fail low in turn. This will take some time to occur at the greater iteration depths. After all this rather wasted effort, a re-search is then necessary with an adjusted $\alpha\beta$ window.

I have found that if the principle move fails low when searched, then it is better to abandon the current $\alpha\beta$ window immediately, adjust it and restart the iteration. I believe this to be a better approach because if the best move is refuted, it is likely that all other moves will be similarly refuted when using the same search window. Of course this may not always be the case, however, I believe the savings will clearly outweigh any time lost when averaged over a number of games. In summary, both fail lows and fail highs are treated similarly at the first ply.

To summarise, my chosen main search will consist of an iterative, depth first, aspiration $\alpha\beta$ search with the enhancements outlined above. The $\alpha\beta$ window will be based around the result of the previous iteration at depths ≥ 2. The initial iteration window will be based on a preliminary analysis.

5.2 Search Extensions

The basic approach when using any of the $\alpha\beta$ derivatives is to search to a depth n, evaluate the leaf nodes and minimax the results. This method exemplifies the full width approach. However, it suffers from a number of weaknesses:

1. The search is terminated at a fixed depth, therefore, the evaluation function may be applied to non quiescent positions leading to grossly inaccurate results.
2. This approach makes inefficient use of the $\alpha\beta$ algorithm because its leaf nodes are inaccurately evaluated.
3. Equal effort is applied throughout the tree with respect to depth. Therefore, stupid uninteresting variations are explored as deeply as interesting or good variations.
4. The horizon effect is severely exasperated by the fixed depth search.

The first two problem areas can be significantly eased by the use of a quiescence search at the leaf (horizon) nodes of a main search. This quiescence search will be a form of selective search, only exploring tactical possibilities. See section 5.4 for more details.

It is surprising how many commercial and tournament chess programs suffer from the third problem area. I believe the solution to this problem lies in a program's ability to select and search interesting or good variations more deeply.

Harris [26] proposed a similar idea of exploring interesting variations, except he suggested doing so without the $\alpha\beta$ algorithm. Harris omitted specific details of how his ideas might be implemented. Levy et al [38] have implemented a number of programs using the idea of interestingness, however, their approach to classifying interesting moves differs from the approach I am taking. In particular, my use of the aspiration $\alpha\beta$ search and its bounds in my search extension heuristics is novel.

I define an interesting variation as one which contains tactical moves that offer some possibility of affecting the search result, e.g. producing a score which is inside the $\alpha\beta$ search window. I define good variations as those which would affect the search result if the search were terminated immediately.

Applying these definitions to create search extension heuristics, I hope to encourage the search to explore variations offering some likelihood of success, and ignore uninteresting variations which offer little or no likelihood of success.

This approach shows some similarities to the way humans play the game. However, because these heuristics are far less sophisticated than a strong human chess players, the program will attempt to bridge this gap by exploring many more variations than its human counter part. These search extension heuristics must therefore be free enough to extend many interesting variations, but also be selective enough to avoid the search blowing sky high. A search which is not depth limited and can explore interesting variations deeply, is much less likely to suffer from the horizon effect (problem area 4).

In order to determine what is an interesting or good variation, moves are classified and receive 'points' depending on how dynamic each move is. Interest points are passed down the search tree from node to node. Each ply of search deducts 16 points worth of interest from the variation. When the interest falls below zero, the node will be considered as a leaf or horizon node of the main search. At this point a quiescence decision is made, leading to a quiescence search or continuation of the main search.

All interesting moves are classified under two categories, Threats and Threat Execution.

5.2.1 Threats

For threatening moves to be considered interesting they must indicate some likelihood of affecting the search result and therefore becoming the main line. Assuming a negamax approach, the node resulting from a threatening move must meet the following criteria:

$$\text{Est} + \text{Thres} > \alpha \textbf{ or } \alpha = \text{Aspir}\alpha$$

where

1. **Est**: This is an estimate of the current node's value, mostly dominated by material.
2. **Thres**: Represents the value of having the move (a tempo) and what that move might achieve.
3. α: This is the current value of alpha at this node, which therefore represents either the value of the best variation found for this side elsewhere in the search tree or an initial aspiration value.
4. **Aspir**α: This is the initial aspiration value used for the best value achievable by the current side to move. At even depths below the root, this is the initial α value; at odd depths, it is the initial β value.

The idea of this heuristic is to exclude moves which may otherwise be interesting if they are unlikely to change the current α value. This value after all, represents the score of the best variation that the side to move can achieve. Of course if this score happens to be an aspiration value, then the score of the best variation is not yet known, it would therefore be unwise to assume that the current variation cannot affect the search result.

The effect of this heuristic is to avoid extending when the line appears fruitless relative to what can be achieved elsewhere in the tree. For example, it is likely to be worthless exploring a variation which wins a pawn after throwing away a queen, when another, better variation is already known to be achievable.

Given that the above criteria are met, the following threatening moves are classified and the appropriate interest points added to the current interest value at this node.

1. **In-check with one legal reply**

   This heuristic provides a search extension along variations which are highly forced. Often mating sequences involve forced checks where there is only one legal response. *Receives 24 points of interest.*

2. **Double Check**

   This heuristic provides a search extension when a side is in-check from more than one piece. By definition this must additionally be a discovered check. Extensions along variations involving double check are important as it is implicit that at least two pieces are co–operating in the attack on the king. *Receives 24 points of interest.*

3. **Discovered Single Check**

   This heuristic provides a search extension when a side is in-check from a discovered attack. It is implicit that the piece moved is not the piece giving check. As this effectively gives the moving piece a free move, it is very important to explore such moves throughly. *Receives 20 points of interest.*

4. **In-check with no legal king reply**

   This heuristic provides a search extension when a side is in-check and the king has no legal escape moves. The idea being to identify possible mating nets. *Receives 20 points of interest.*

5. **In-check with more than one legal reply**

   This heuristic is common to a number of top programs, Hitech included [7]. However, my program is more selective because this heuristic is only applied where the search result is likely to be affected. I feel this is a significant advantage over its otherwise unrestricted application. It is important to explore in check variations throughly because other tactics may flow from a check. *Receives 16 points of interest.*

6. **Pawn moves to the 7th Rank**

   This heuristic encourages exploration of variations where the pawn is pushed near to promotion. *Receives 16 points of interest.*

7. **Pawn moves to the 6th Rank**

   This is a subset of the above heuristic and serves to increase exploration of positions where passed pawns can be pushed, especially in or near the endgame. *Receives revPhase points of interest.*

8. **Piece Threats**

   The following search heuristics are concerned with threats on pieces of the side currently to move:

   (a) **More than one piece is en-prise**
   Receives 16 points of interest.

   (b) **A pinned piece which is en-prise**
   Receives 16 points of interest.

   (c) **A trapped piece which is en-prise**
   Receives 16 points of interest.

   These heuristics provide search extensions along variations of complex dynamic possibilities. The quiescence search will resolve such positions when the threatened player is **not** to move. Hence these heuristics are only applied in positions where the threatened player has the move. In this way the player must show the existence of an adequate defensive move or the ensuing quiescence search will refute the variation.

9. **Mate threat against the side to move**

   This heuristic provides a search extension when mate is directly threatened against the side to move. This will force exploration of variations to show whether the mate threat can be defended against. *Receives 24 points of interest.*

Search heuristics 8a, 8b, 8c and 9 are only applied during the quiescence search after a full positional evaluation. Any of the heuristics 1 through 6 can be applied in either the main or quiescence searches. More details about this can be found in sections 5.3 and 5.4. Heuristics 1 to 5 are mutually exclusive.

Each ply of search consumes 16 interest points. It can therefore be seen that some heuristics which equal or exceed an interest value of 16 may in theory cause the search to be unbounded. However, in practice this should not be the case as:

- It is rare to get out of check with a check move itself.
- Heuristics 8a, 8b, 8c and 9 are only applied once or twice down any given variation.
- Heuristics 6 and 7 are self limiting as a pawn can only move forward.

For added protection against an unbounded search, all nodes visited in the main search which are deeper than twice the iteration depth, have their interest value decreased by the amount which the current ply exceeds twice the iteration depth.

5.2.2 Threat Execution

The execution of a previous threat is only interesting if at the current node it looks likely to affect the search result.

For any move which constitutes the execution of a threat, the following criteria must be meet for the move to be further considered as interesting:

$\text{Est} - \text{Thres} < \beta$ **or** $\beta = \text{Aspir}\beta$

where

1. **Est**: This is an estimate of the current node's value, mostly dominated by material values.
2. **Thres**: Represents the value of having the move (a tempo) and what that move might achieve.
3. β: This is the current value of beta at this node, which therefore represents either the value of the best variation found for this side elsewhere in the search tree or an initial aspiration value.
4. **Aspir**β: This is the initial aspiration value used for the best value achievable by the side **not** to move. At even depths below the root, this is the initial β value; at odd depths, it is the initial α value.

This heuristic is almost an exact mirror image of the one used for threats. Given that the above criteria are met, the following threat execution moves are classified and their appropriate interest points added to the current interest value at this node.

1. **Capture**

   This heuristic encourages exploration of interesting capture sequences. Particularly where greater valued pieces are involved. *Receives interest points depending on the type of piece captured, where pawn = 1; knight = 2; bishop = 3; rook = 4 and queen = 5.*

2. **Pawn Promotion**

   This heuristic also encourages exploration of variations which are interesting. Assuming the promoted pawn cannot be captured, the move is very likely to be good and therefore interesting. If, however, the promoted pawn can be captured, the opponents options have a degree of forcedness about them which therefore, also makes this move interesting. *Receives interest points depending on the type of piece promoted to, where knight = 1; bishop = 2; rook = 3 and queen = 4.*

3. **Recapture**

   This heuristic is used in Hitech [7]. I feel it is of great benefit in that it encourages not only exploration of simple exchanges, but also captures with positional content e.g. a rook exchanged for knight and positional compensation. This heuristic is only applied if: $|LastSearchValue - Est| \leq 25$ and the last 2 moves played were captures. *Receives 16 points of interest.*

4. **Mate Threat against the side not to move**

   This heuristic is only applied during the quiescence search after a full position evaluation. It is extremely useful for avoiding early termination of the search in possible mating sequences. *Receives 16 points of interest.*

It should be noticed that some of my definitions of interestingness attempt to measure the forcedness of moves and replies. In this way I hope to achieve some of the benefits found by Feng-hsiung Hsu and his team in their recent paper "Singular Extensions: Adding Selectivity to Brute-Force Searching" [3].

Unlike singular extensions my approach attempts to classify forcedness in a more static way in order to avoid expensive test depth searches. Therefore, this approach does suffer the drawback that dynamic non static forced variations are not extended. This is essentially the trade off between static and dynamic search extensions. However, I have managed to incorporate dynamic search information into the search heuristics without added cost. This allows the search heuristics to be quite selective about which variations are worth extending. Additionally, some of my search extension heuristics can extend non–forced but still interesting variations. I see this as an advantage over singular extensions.

To illustrate how these extension heuristics are able to identify deep combinations, I will apply them to the position shown in figure 5.1.

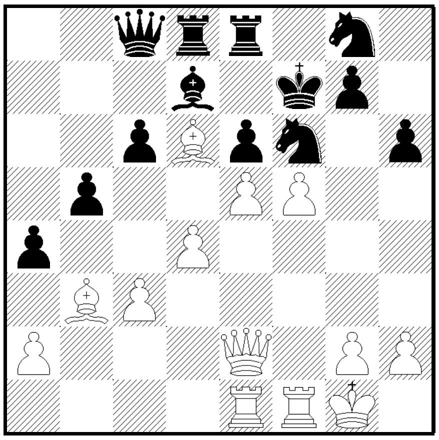


Figure 5.1: White to play and mate in 10.

Ply	*Interest*	*Move*	*Adjust Ply*	*Extension type*	*Interest Bonus*
1	96	Qh5+	−16	T4	+20
2	100	... Nxh5	−16	TE1	+5
3	89	fxe6+	−16	T2+TE1	+24+1
4	98	... Kg6	−16	-	0
5	82	Bc2+	−16	T1	+24
6	90	... Kg5	−16	-	0
7	74	Rf5+	−16	T5	+16
8	74	... Kg6	−16	-	0
9	58	Rf6+	−16	T2	+24
10	66	... Kg5	−16	-	0
11	50	Rg6+	−16	T5	+16
12	50	... Kh4	−16	-	0
13	34	Re4+	−17	T1	+24
14	41	... Nf4	−18	-	0
15	23	Rxf4+	−19	T1+TE1	+24+2
16	30	... Kh5	−20	-	0
17	10	g3	−21	T9	+24
18	13	... any	−22	TE4	+16
19	7	Rh4++	-	-	-

Figure 5.2: A table showing interest values for a 19 ply mating sequence.

Note: Tn represents threat heuristic n, and TEm represents threat execution heuristic m.

In the position in figure 5.1, white can force checkmate in 10 moves (19 ply). The table shown in figure 5.2 shows how on a 6 ply iteration, these heuristics are able to identify the correct 19 ply move sequence. Assume for simplicity that aspiration values of α and β are present when the given variation is explored.

5.3 Quiescence Decision

Unlike most other chess programs, HIARCS will not leap into a quiescence search when the depth of the full width search reaches zero. The reason behind this lies in understanding what the quiescence search is capable of.

Were it truly capable of searching all the important possibilities in a position until true quiescence is achieved, there would be no point in having a main search at all. The quiescence search could be directly applied to all moves at ply 1. Sadly, however, we do not yet know how to build such a near-perfect quiescence search. Therefore, by knowing what my quiescence search is capable of, I am able to select when the position is suitable for such a search.

The quiescence search is capable of exploring lines involving captures, pawn promotions and some checking moves. The current side to move is always given the option of standing pat i.e. not making any move and accepting the current evaluation score. Therefore, 'Threat' search extension heuristics 1 to 6 and 8, and 'Threat Execution' search extension heuristic 4 are applied in the quiescence search possibly causing a rejoining of the main search. Inside the quiescence search, if any of these extension heuristics are applied successfully, the main search is rejoined regardless of the resulting interest value. This avoids the premature termination of the search when aspects of interest still exist.

This approach not only makes a quiescence decision at the horizon of the main search, but also more importantly, at each and every ply during the quiescence search. This should enable the program to find deep combinations even after the quiescence search is underway. For example, a 4 ply iteration may progress down 5 plies of main search (1 ply extension), join the quiescence search for 1 more ply, then rejoin the main search when something interesting is found and proceed for 3 more plies in the restarted main search before finding a 9 ply combination which would otherwise be beyond the program's capability.

5.4 Quiescence Search

A quiescence search is performed to check out purely tactical possibilities which are very difficult to assess accurately at the horizon of a main search by using a static evaluation function.

This search can therefore be seen as a selective tactical search complementing the static tactical parts of the evaluation function. Hence, improving the accuracy of the results backed up from the horizon nodes.

The search explores captures, pawn promotions and some checking moves. Ideally all dynamic moves should be explored, these include checks, check evasion, threats, defences, blocks, X-ray attacks, captures, pawn pushes, promotion etc. A combination of a full null move algorithm and my proposed quiescence search would offer some hope on this direction. However, two reasons have made me avoid such a combination in this project:

1. The search depth is very difficult to control and without strict controls the search can easily explode.
2. This type of search would explore many thousands of variations and hence, require very fast hardware to support this additional load.

Pseudo code illustrating my quiescence search using the negamax convention can be found in figure 5.3.

This search explores moves proposed by the GenerateQMoves function. Each move is searched only if it can change the search result i.e. eval + $Value_m > \alpha$. Each side to move also has an extra option of standing pat thereby in effect playing a null move, see section 3.1.6.

A number of additional tree pruning techniques are used to prune the tree:

- **Note 1**: As the side to move always has the option of playing a null move and accepting the current static score, a node with a static score already above the search window is pruned. Implicit in this is the fact that any subtree below this node is also pruned before it is searched.
- **Note 2**: The null move option gives rise to the possibility of further reducing the width of the search window. This is because the worst the side to play can do is 'stand pat' and accept the current evaluation score.
- **Note 3**: Each move proposed by GenerateQMoves can be assigned an optimistic value ($Value_m$), e.g. captures receive the value of the captured

> piece plus an optimistic positional score. As the opponent always has the option of making a null move[1], this optimistic value represents an upper bound on the score derivable from the subtree resulting from this move. Therefore, if even this optimistic value cannot affect the search result, the potential node and its subtree can be pruned.

These 3 forward pruning mechanisms can drastically reduce the size of the quiescence search.

In conclusion, I feel this combination of main search, quiescence search and selective extensions should significantly improve the program's tactical ability. Without such enhancements, it is likely to compare unfavourably with other strong programs tactically due to the significant amount of positional knowledge elsewhere in the program consuming large chunks of time.

---

[1] this may not be the case if the main search if rejoined, however, the affects are minimal and can be ignored.

```
Function Quiescence(P : position; $\alpha$, $\beta$ : integer) returns integer

var
    m      : move
    best,
    eval,
    score  : integer

begin

    best := Evaluate(P, $-\beta$, $-\alpha$)
    if (best $\geq \beta$)
        return (best)                              /* See note 1 */
    endif
    $\alpha$ := max($\alpha$, best)                       /* See note 2 */

    GenerateQMoves(P)                              /* Generate moves */
    eval := best                                   /* Can always 'Stand Pat' */
    for each move m do
        if (eval + $Value_m > \alpha$)                  /* See note 3 */
            /* Search move m to next depth using negamax convention */
            score := $-$Quiescence($P_m$, $-\beta$, $-\alpha$)
            if (score > best)
                best := score                      /* Store best score found */
                if (best $\geq \beta$)
                    return (best)                  /* $\beta$ cut off : return best score */
                endif
                $\alpha$ := max($\alpha$, best)           /* Update fail soft */
            else
                best := max(best, eval + $Value_m$)
            endif
        endif
    endfor

    return (best)                                  /* Return score of best move found */

end Quiescence
```

Figure 5.3: Pseudo code for my Quiescence Search.

Chapter 6

HIARCS 5.0

In this chapter, I will show how the various techniques I have investigated in this project can be brought together and implemented in a chess program. There is neither the time nor space to fully explain all the design issues, some of which have evolved over the last couple of years. However, I will give an overview of the program's design and structure. I will also discuss some of the main decisions which have been taken relating to the algorithms and techniques used.

This program contains around a 50-50 split of old code and totally new code resulting from my investigations since beginning this project.

6.1 Program History and Environment

I began writing chess programs as long ago as 1979. My first chess program, UCT, only played the ending of king and rook versus king. However, this only served to wet my appetite for the subject. In late 1980, I developed my first full chess program, HIARCS. However, it was only capable of playing random chess. Later, in 1981, the chess program was improved to search 1 ply with material and mobility evaluation. By the end of 1983, HIARCS was enhanced to a 4 ply selective search program applying rudimentary positional and swap off evaluations at its leaf nodes. The program was written in compiled Basic with some Z80 assembly routines and was capable of playing at weak club level, ELO 1300, on a Z80 processor.

At this point activity stopped until early 1987, when I began work developing a totally new full width chess program written in Modula-2 for an Amiga. In mid 1988, the program was rewritten in the C language for two main reasons:

- Code efficiency
- More low level freedom

At this stage the program was playing at around ELO 1550 while analysing 40 - 60 nodes per second.

The program was further enhanced during 1988/89 until by August 1989, when it was entered in the 1st Computer Olympiad, HIARCS 4.1 as it was then known, was analysing at the rate of 80 - 180 nodes per second and playing at about ELO 1750.

Early on during this project I decided to up load my program onto our VAX 785 machine at work. Three reasons lay behind this decision:

1. **Security.** The possibility of my Amiga malfunctioning or even having a virus posed too great a risk. I therefore felt it was safer to develop the program on a machine where regular backup security is already provided.
2. **Speed.** Even though a VAX 785 machine is slow by todays standards, it is still faster than an Amiga. I found that the H/W change provided around a 2-fold speed improvement, provided the machine was otherwise unused.
3. **Environment.** I felt the UNIX[1] operating system provided a better environment in which to develop the program.

6.2 Program Structure

HIARCS 5.0 is a large and complex program. It is split up into 12 modules which each perform a particular type of job. In figure 6.1, a diagram is provided which illustrates how the modules interact with each other. The diagram is at a high level and much detail has been omitted for reasons of clarity. The lines represent the call interaction and the accompanying description gives the reason for the module interaction.

The main functions performed by each module are explained below:

---

[1] UNIX is a Trademark of Bell Laboratories.

Figure 6.1: Module interaction diagram.

1. **Hiarcs**: This is the main program module. It controls initialisation and oversees the interaction between the user, via the MMI module, and the chess decision algorithms.
2. **HouseKeeping**: This module deals with initialisation for a new game.
3. **HashDriver**: This module controls transposition table creation, deletion and entry storage and retrieval. It also updates hash indexes and modifies the transposition table between searches. An opening book transposition table is also loaded and accessed by this module.
4. **MMI**: This module deals with all human/computer interaction. It also checks entered user moves for legality and processes various commands.
5. **TreeSearch**: This module performs the tree searching necessary to select the best available move. To achieve this it coordinates the activities of the program and makes extensive use of most other modules.
6. **Generate**: This module contains all the move generation routines required by the program.
7. **TimeControl**: This module allocates, adjusts and controls the time used by the program.
8. **PosEval**: This module performs a positional analysis on a given position. It contains most horizon node evaluation criteria.
9. **Planner**: This module considers and selects plans to direct the search and updates various weightings associated with Pc/Sq tables and the game phase.
10. **SqAnalysis**: This module incrementally updates the attack tables and analyses various attack interactions for positional evaluation purposes.
11. **Mover**: This module makes and unmakes generated moves. It also controls the incremental updating of the attack tables.
12. **Utils**: Provides various utility functions which most of the modules require.

6.3 Data Structures

The data structures used in HIARCS are of central importance to the overall design of the program. They define to a large extent how intelligent the internal decisions made by the program can be. This in turn largely controls the knowledge based capabilities of the program.

This section explains the main data structures used, the reasons for their given composition and their application to the program.

6.3.1 Position Representation

The position is represented by two structures, Position and Colour.

Position = 64{PieceType}64
PieceType = NoPiece | Pawn | Knight | Bishop | Rook | Queen | King
NoPiece = 0
Pawn = 1
Knight = 2
Bishop = 3
Rook = 4
Queen = 5
King = 6

Colour = 64{colour}64
colour = White | Black | None
White = 0
Black = 1
None = 2

These two structures are similar to the ones Shannon originally suggested. The differences include the splitting of piece type and colour to enable generic move generation and square control functions. Both structures are linear and one dimensional. Note that no off–board edge is used on this board, as an alternate move generation process is used. This enables all piece square tables to be of a significantly reduced size when compared to a 10x12 board approach, for example. Although the structures are one dimensional for efficiency, conceptually they remain two dimensional.

The most important reason for such a 64 element structure is that any move can be represented in 12 bits, 6 bits for the from-square and 6 bits for the to-square. This enables specialised search tables to be used which are directly accessed by the 12 bit move value. Additionally, it allows a complete move description including move type, such as castling, pawn promotion etc, to be stored in 16 bits. Thus move comparisons and copies can be done in one operation.

6.3.2 Game Tree

```
Tree        = 0{Node}2000
Node        = Move + Score           /* 4 bytes */
Move        = type + move            /* 4 bit type, 12 bit move */
move        = 0..4095                /* 12 bits */
type        = Capture | Promotion | Castle | EnPassant
Promotion   = PQueen | PKnight | PRook | PBishop
Capture     = 8                      /* bit 15 */
PQueen      = 4                      /* bit 14 */
PKnight     = 5                      /* bit 14,12 */
PRook       = 6                      /* bit 14,13 */
PBishop     = 7                      /* bit 14,13,12 */
Castle      = 2                      /* bit 13 */
EnPassant   = 1                      /* bit 12 */
Score       = −4096..4095
```

At least part of the game tree must be stored so it can be traversed efficiently. This structure allows the storage of nodes generated, but not yet visited at the current search iteration. Only the nodes generated down the current line of exploration need to be stored as these are the ones which have not been completely searched. Each node is only generated once at each iteration except when a fail high or low occurs. While storage of the whole tree would slightly improve performance, it would be extremely costly in terms of memory usage, particularly in the endgame.

Together with the move stored at each node, a score is used to enable move ordering at internal nodes of the search tree.

6.3.3 Transposition Table

In chess, the game tree is really a directed graph of moves. As the depth of search increases, the number of potential transpositions increases exponentially. Additionally, when using an iterative deepening search, the nodes visited at iteration $n-1$ are also usually visited at iteration n. Therefore, the transposition table provides a useful way of storing and retrieving previously calculated moves and scores. For deeper searches (depth > 5) a transposition table can significantly improve the performance of a program. This is particularly evident in the endgame where much deeper searches are possible.

```
TransTable  = 1{TableEntry}32768
TableEntry  = HashLock + HMove + TScore
HashLock    = long integer                  /* 32 bits */
HMove       = Depth + move
TScore      = Type + Score                  /* 3 bit type, 13 bit score */
Depth       = 0..15                         /* 4 bits */
Type        = Locked + TrueScore + Bound    /* MS 3 bits of TScore */
Locked      = 8                             /* bit 15 */
TrueScore   = 4                             /* bit 14 */
Bound       = 2                             /* bit 13, set if UpperBound */
                                            /* unset if LowerBound */
```

This data structure represents the transposition table used by HIARCS 5.0. Each entry stores 5 distinct types of information:

1. Position lock.
2. Value of the position.
3. Best move known in the position.
4. Depth of the subtree from which the value is calculated.
5. Flags indicating the score type relative to the $\alpha\beta$ window.

Table Access

The access method used is similar to the one proposed by Zobrist [63]. A set S of 2x2x6x64 randomly generated 32 bit integers are used to represent all possible piece/square combinations (2 integers per square; 2 colours; 6 piece types; 64 squares).

A single hash index can then be calculated for any position by applying:

$\forall col, pc, sq \quad H_i = H_i \ xor \ S_{col,pc,sq}$
where
Position[sq] = pc **and** Colour[sq] = col

The hash indexes are actually incrementally updated in HIARCS. For example, to move from square *fsq* to square *tsq*, the following operation is required:

$$H_{i2} = (H_{i1} \ xor \ S_{col,pc,fsq}) \ xor \ \ S_{col,pc,tsq}$$

Captures, pawn promotion, en-passant and castling all use a similar, if slightly more complex approach.

Using two 32 bit hash indexes, direct transposition table access is possible. The first index is used to access the table and the second is used as a position lock to check that the correct entry has been accessed and reduce the possibility of an undetected collision.

Unlike normal hashing, only one table access is tried. This makes the process both simpler and more efficient. Additionally, the table is split into two halfs, one half for white to play positions and the other half for black to play positions.

Entry Storage

Node attributes are stored in the transposition table as the node is about to back up its results to its parent node. The entry is **not** stored if it will overwrite an existing entry whose information is the result of a subtree search of greater depth. Therefore, each entry stored is privileged depending on the subtree depth. As values derived from a deep search can constitute a great deal of effort, this is a significant improvement over a simple replacement algorithm.

The position score stored in an entry may be a true score or a score outside the current $\alpha\beta$ window, i.e. a cut off. In the latter case, the score represents a search bound which may be used to narrow an $\alpha\beta$ window if retrieved. Hence, the score type is also stored in the entry.

The best known move from the current node is also stored to improve move ordering on retrieval.

Entry Retrieval

When a node in the main search is visited, the transposition table is checked to see if an entry for the current position exists. If an entry exists, it is checked to see if it is storing information with a search depth equal or greater to the depth required for this search. If so, the entry can be retrieved.

Once retrieved, the entry's position score can be used. Depending on the score's type, 3 outcomes are possible.

1. **True Score**: The value stored is an exactly calculated value. Therefore, the value of this node and its subtree is known (for the required depth of search) and can be returned to its parent node without searching.
2. **Upper Bound**: The value stored is not exact, it was the result of a cut off in the subtree below the node which stored it. Therefore, the value of this node is not known, however, the value stored can be used to narrow the

current $\alpha\beta$ window by possibly reducing the β value and perhaps thereby causing a cut off.

3. **Lower Bound**: The value stored is not an exact value, only a bound. The value can, however, be used to narrow the $\alpha\beta$ window by possibly increasing the α value and perhaps thereby causing a cut off.

Assuming the node is not terminated by a true score or cut off, the stored move can be retrieved and searched first from this node. This should significantly improve move ordering as this is very likely to be the best move available.

Opening Book

This transposition table is initially loaded with an 'Opening Book' of moves before the game begins. This allows the program to 'know' about particular opening sequences commonly played by strong chess players. Instead of just move sequences being stored, positions and associated moves are stored enabling the program to detect any transposition of moves. This avoids cases where the opponent can fool the program by playing moves in an unusual sequence. In fact, sometimes in the opening the program will transpose into another opening itself, e.g. transposing from an English opening into a Queen's Gambit Declined.

6.3.4 Refutation Table

RefutationTable = 1{Variation}128
Variation = 1{move}32

The refutation table is used to store the expected continuation following each 1 ply move. Some of the variations will represent the best line of play, others will represent the variation which refuted the 1 ply move (hence the name).

Every move searched at iteration n has its expected continuation saved. During iteration n+1, or a re-search, the expected continuation is searched first. This helps to drastically improve $\alpha\beta$ cut offs by guiding the search down lines which are likely to refute the 1 ply move or at least prime $\alpha\beta$ with reasonable values.

6.3.5 The History Heuristic

HistoryT = 8192{Value}8192
Value = 0..65535

The History Heuristic [50] maintains a record of every move's ability to cause a refutation. The table is split into two halfs; one half for white moves and one half for black moves. It is indexed by the 12 bit move value which uniquely identifies any move[2]. Therefore, a total of 8192 table entries are required.

At every interior node of the main search, the best move found is recorded in the history table by modifying the appropriate value. The value is actually modified by 2^d, where d is the minimum depth of the subtree below the derived best move.

At the beginning of the game the history table has all its entries set to zero. Schaeffer [50] suggests that the table is reset in this way at the start of each new search. However, in HIARCS, the table is not reset between searches, but adjusted. Each move's value is adjusted by dividing it by 4. This allows the table to have some influence on the next search.

The history heuristic is applied at each interior node of the main search where move generation is required. As moves are generated, their history value is looked up in the history table and encorporated into a special move ordering score. See section 6.4.2 for more details.

6.3.6 Incremental Attack Tables

AttackT = 64{AttackEntry}64
/* 16 byte entry */
AttackEntry = AttackCode + Count + PieceList
/* 16 bit attack code */
AttackCode = Pad + AttP + AttN + AttB + AttR + AttQ + AttK
Pad = 0
AttP = 0..3 /* Number of pawn attacks, 2 bits */
AttN = 0..7 /* Number of knight attacks, 3 bits */
AttB = 0..7 /* Number of bishop attacks, 3 bits */
AttR = 0..7 /* Number of rook attacks, 3 bits */
AttQ = 0..7 /* Number of queen attacks, 3 bits */
AttK = 0..1 /* Number of king attacks, 1 bit */
Count = 0..255 /* Count of attacks */

---

[2] although not the move's type

```
PieceList        = 13{AttSq}13    /* List of attacking pieces */
AttSq            = square         /* Square of piece attacking this sq */
square           = 0..63
```

Two attack tables are maintained, one for white attacks and one for black attacks. These tables represent perhaps the most important data structure in the program. Each table records all the attacks by a given colour on each and every square. Additionally, a list of each piece attacking each square is also maintained.

This table, allows complex inter–piece calculations which would otherwise be very time consuming. For example, it is a simple matter to find the square occupied by a checking piece; in turn it is simple to calculate whether this checking piece is itself threatened with capture.

The evaluation function makes extensive use of the attack/defence balance of each square. Therefore, these tables are of central importance to the evaluation function as they allow it to make various positional evaluations which would otherwise be far too time consuming.

These tables also contribute to the move generation and in particular the legal check evasion and capture move generation routines of the program. They are also important in the move ordering process because information about attacked squares is used to order moves according to how reasonable they appear. For example, moving an en-prise piece or capturing an undefended piece is ordered higher than moving a piece to a square where it will be en-prise. See section 6.4.2 for more details.

All this attack oriented information is far too time consuming to regenerate at each new node. In fact, each move only changes some of the attack properties of a position. For example, in the opening position, the move e4 would require only the attacks of the white queen, king's bishop and king's pawn to be adjusted. Therefore, an incremental approach seems to be warranted. Although there are moves which make this incremental approach very complex, e.g. an en-passant move; and the programming of such an approach is very tricky, I have found it to be very cost effective.

The added information available at all nodes is extremely useful and has many applications, some of which have been shown. In fact, I feel the full potential of having such a wealth of information has not as yet been fully exploited. I do not know of any other chess programs which incrementally maintain such a comprehensive attack table throughout the search.

6.3.7 Game History

GameHistory = 0{MoveRecord}512
MoveRecord = GMove + GScore + CPiece + CColour + EPsq + Drawmn + Depth + UTime + Nodes + Eval + Hash1 + Hash2
GMove = Move
GScore = Score
CPiece = PieceType
CColour = colour
EPsq = square
Drawmn = 0..512
Depth = 1..32
UTime = long integer
Nodes = long integer
Eval = long integer
Hash1 = long integer
Hash2 = long integer

The game history data structure records the important elements of each move so it can be accurately reversed, if necessary. This structure is used for both actual moves played in the game and moves made while searching.

Drawmn represents the move number of the last irreversible move. This is used to ease detection of the draw by 3-fold repetition or the 50 move rule. EPsq is stored so the possible en-passant capture square can be restored when the move is reversed. GScore stores the positional value of any piece captured to enable incremental evaluation and restoration while searching and stores the move score for moves actually played.

UTime, Nodes, Eval and Depth are all measurements stored after actual moves are played. The other fields, GMove, CPiece, CColour, Hash1 and Hash2 are used to allow easy reversal of any move.

6.4 The Algorithms

This section explains some of the more critical algorithms used in HIARCS. Many other algorithms are used and I would refer the reader to the actual code listing if more details about these algorithms are required.

6.4.1 Move Generation

The move generation routine used depends on the type of search being employed at a given node and the state of that node. Three different types of move generation can be performed:

1. **Full Pseudo Legal Move Generation**: Performed at all nodes in the main search tree where the player to move is not in-check.
2. **Full Legal Move Generation**: Performed at all nodes in the search tree where the player to move is in-check.
3. **Selective Tactical Move Generation**: Performed at all nodes in the quiescence search tree where the player to move is not in-check.

Full Pseudo Legal Move Generation

This routine generates all pseudo legal moves in the current position. The legality of each move is not checked at this stage for 2 reasons:

- Most moves may not need searching at all because of $\alpha\beta$ cut offs.
- Most moves are legal in positions where a king is not in check. Therefore it would be wasteful to test for the exceptional case at this stage.

All moves are generated using an offset method via a set of move generation tables. For example, to move in direction 3 (-8 offset) the current square, *sq1* say, is used as an index into a table which gives the resulting 'to square' (*sq1*-8). If the 'to square' is off the board, the number 128 is returned (most significant bit set). This method avoids the need to have a boundary around the board, therefore allowing a 64 square board. As explained earlier, this allows convenient 12 bit move storage.

All non capture/special moves are generated using these move tables. Captures are generated directly from the appropriate attack table. Pawn promotion to minor pieces is catered for, each move having the same 12 bit move code, but a different move type. Castling moves are generated only if the following 5 conditions are met:

1. The king has never moved.
2. The rook involved has never moved.
3. The squares between the king and rook are empty.
4. The king is not in check.
5. The king will not pass through or into check as part of the castling manoeuvre.

The incremental attack table is very useful for checking conditions 4 and 5.

En-passant moves are tested and generated by checking the relevant attack table for pawn attacks on the current en-passant square (if any).

Full Legal Move Generation

When the king is in check, most pseudo legal moves are in fact illegal. Therefore, it makes sense to generate only legal moves when the king is in check.

It is surprising how many chess programs continue to generate pseudo legal moves even when the king is in check. They then waste time proving that most of the moves generated are in fact illegal. Admittedly, it does take some sophisticated data structures to enable such legal move generation to be accomplished efficiently. My incremental attack tables do provide enough information to allow such move generation to be done very efficiently. In fact, in tests, I have found legal move generation is approximately 10 times faster than pseudo legal move generation and validation, assuming no $\alpha\beta$ cut offs occur.

Legal moves when in check fall into one of the following 3 categories:

1. King moves to unattacked squares.
2. Captures of the checking piece by pieces not pinned to the king.
3. Interposition between the king and the checking piece, by pieces not pinned to the king.

In category 1, it is important to take account of checks by vector type pieces (bishop, rook and queen) as these effectively attack squares through the king. These attacks are not obvious from the attack tables themselves.

Note when in double check, only king moves need to be generated as it is impossible to get out of check by playing moves in categories 2 or 3.

Generating only legal moves gives an added benefit to the program's search extensions: It is known before searching any moves, how many legal moves exist, how many legal king moves exist and whether the check is doubled or discovered.

Selective Tactical Move Generation

Moves generated in the quiescence search when not in check are rather limited to obvious tactical moves. Only two tactical move categories are generated, captures and pawn promotions.

Only pawn promotions to a queen are considered. All capture moves are generated with the exception of en-passant. These special case moves are not considered cost effective enough to warrant inclusion in this move generation process. All moves generated by this phase are only pseudo-legal.

6.4.2 Move Ordering

As explained in chapter 3, move ordering is vital to the efficiency of the $\alpha\beta$ search algorithm. To achieve the optimal $\alpha\beta$ search tree, it is not necessary to examine moves in a strictly best to worst order at each node.

The optimal $\alpha\beta$ search tree can be achieved by:

- Along a variation which will become the new principle variation, the best move must be examined first at each ply.
- The first move explored at even ply depths must be a refutation move, e.g. one that causes a β cut off.
- At odd ply depths which are not part of a new principle variation, ordering is unimportant provided a refutation move is examined first at the next depth.

It can be seen from these points that it is vital to examine refutation moves as early as possible. In HIARCS, the move ordering algorithm's basic aim is to always examine the best move first. However, emphasis is placed on examining moves likely to cause β cut off's, e.g. capturing the piece just moved.

The main move ordering algorithm is performed while generating moves. However, a significant proportion of the time, HIARCS does not need to generate any moves. This is because at each node the refutation and transposition tables are probed, possibly resulting in a move being proposed. Any move proposed by these tables has a high probability of causing a refutation. It is therefore explored first without generating any moves. If a cut off is achieved, the costly exercise of move generation and ordering is avoided altogether.

Two move ordering algorithms are used, one during both the pseudo legal and legal move generation routines and one during quiescence move generation.

Full Move Ordering

As moves are generated in the full pseudo or legal move generation routines a score is assigned. This score allows the moves generated to be ordered on merit. While the score does represent an estimate of the move's value, it also attempts to estimate the likelihood of the move causing a cut off.

Each move is scored by the following procedure:

1. The move is looked up in the history table and the value found is assigned as the move's initial score.
2. When a move is found to be best at a given node in the tree it is stored as a killer move and given an appropriate value. In HIARCS, 3 killer moves are stored at each ply, 2 for moves causing cut off's and 1 for non cut off moves.

   The move ordering algorithm checks the move against the following stored killer moves:

   (a) Refutation killer's 1 and 2 at this ply.
   (b) Normal killer 3 at this ply.
   (c) Refutation killer 1 at ply-2.

   If the move is a killer move at this depth of the tree, the score is incremented by the respective killer value.

   This heuristic supports the history heuristic by suggesting recent strong moves at the current depth. It is often the case that the last move to cause a refutation at a given depth will repeatedly achieve the same result in other positions at the same depth.

3. If the move captures the piece just moved by the opponent, it receives a bonus score of 400 points. This indicates the importance of such moves in causing refutations.

4. The estimate is then modified by:

   $CPT - SWP + ENP$

   where

   CPT : is the value of the captured piece, if any, otherwise 0.

   SWP : is the estimated value of the potential exchanges on the to square. Defined by:

```
SWP = if (not Attacked([Any],Opp(col),to))
          0
      else
          if (not Attacked([Any],col,to))
              $MT_{col,from}$
          else
              if (Attacked([Any $<$ On(from)],Opp(col),to)
              or $\sharp$DiffAttacks(Opp(col),to) $> 0$)
                  if (Control(to) = col)
                      $MT_{col,from}/2$
                  else
                      $MT_{col,from}$
                  endif
              else
                  if (Control(to) = col)
                      0
                  else
                      $MT_{col,from}/2$
                  endif
              endif
          endif
      endif
```

   $MT_{col,sq}$: See evaluation function definition 1, Material.

   ENP : Represents an estimate of the threat to the moving piece on its original square, e.g. its urgency to move. Defined by:

```
ENP = if (Attacked([Any $<$ On(from)],Opp(col),from))
          $MT_{col,from}/16$
      endif
```

Note: 'from' and 'to' are the from and to squares of the generated move.

SWP and ENP are only calculated if the History and Killer values summed, are less than 32. This avoids good refutation moves being badly ordered because they look inferior to this score estimator, when in fact previous searching has shown them to be good in the past.

Quiescence Move Ordering

In the selective tactical move generator, moves are scored by their optimistic possibilities. This is to enable the forward pruning mechanism explained in section 5.4, note 3.

All capture moves are scored by the value of the piece captured minus the capturing piece's representation (1-6), e.g. 3 for a bishop. This results in an ordering where captures of higher valued pieces are examined first. Where more than one way to capture a piece exists, the moves are further ordered by considering captures by the lower valued pieces first.

This is known as Minor to MajorC ordering and has been shown to be the most effective ordering of capture moves [10].

Pawn promotions receive an estimated score of QueenValue − PawnValue. Where the promotion is also a capture, the score is further increased by the capture score explained above.

Although all the ordering algorithms score moves as they are generated, the moves are not fully sorted into order. Only the best move not yet searched is required to be sorted as it may itself cause a cut off, avoiding the need for any more sorting.

6.4.3 Tree Searching

The actual search techniques employed in HIARCS were explained and illustrated in chapter 5, Tree Searching. However, some of the work 'behind the scenes' of the search procedures was not explained. This section will therefore highlight these more implementational tasks which the search must perform.

Three search routines are used in HIARCS:

1. **Root**: This recursive function oversees and controls the search process. It performs the following key tasks:

  - Selects the search required for the next depth.
  - Checks moves for known results, e.g. checkmate in 2 moves.
  - Sorts 1 ply moves between iterations.
  - Allocates and assigns refutation lines to and from the refutation table.
  - Adjusts time control information in the light of search results.
  - Maintains search status indicators which aid the time control decision process.
  - Updates the history table with the best move found.

2. **AlphaBeta**: This recursive search function explores all nodes in the main search tree below ply 1. It performs the following key tasks:
   - Calls for evaluation of nodes.
   - Makes search extension decisions.
   - Probes refutation and transposition tables for scores and moves. Often this is enough to cause termination of the node.
   - Calls for generation of moves when necessary.
   - Calls for a search to the next depth.
   - Maintains the best variation of moves from this node and returns such information to its parent node.
   - Saves information about the node in the transposition table.
   - Updates killer moves and scores for the current depth.
   - Updates the history table with the best move found.

3. **Quies**: This recursive search function handles nodes beyond the horizon of the main search. It performs the following key tasks:
   - Calls for evaluation of nodes.
   - Makes search extension, main search continuation and quiescence decisions.
   - Calls for re-entry of the main search.
   - Calls for generation of tactical moves when necessary.
   - Calls for a search to the next depth.
   - Maintains the best variation of moves from this node and returns such information to its parent node.

Figure 6.2 shows a state transition diagram for the search. It illustrates how each search state interacts. An explanation of each state is given below:

Figure 6.2: State Transition Diagram for the Search.

IDLE

When no searching of any kind is being performed, the program is in the *IDLE* state. At program start up, this is also the initial state.

If the program is expecting a move, it will start to search on the opponents time for its own best response to the expected move. This is known as "thinking on the opponents time". To achieve this the *SEARCHING ON OPP* state is entered. If the opponent actually makes a move while in the *IDLE* state, the entered user move is made and the program enters the *SEARCHING* state.

SEARCHING

This state carries out the tree searching on the computer's time. It uses the aforementioned three search routines. The search may start from the root of the tree or continue a previously interrupted search.

If the search derives a known value for the game tree, like checkmate in 3 moves for example, no more searching is required and the *SEARCH COMPLETE* state is entered. However, if the game tree value is not known, the program continues 'thinking' until either, a user interrupt (↑C) occurs or an alarm timer expires, where upon the *SEARCH INTERRUPTED* state is entered.

SEARCHING ON OPP

Like the *SEARCHING* state, this state carries out the tree searching, except in this case on the opponents thinking time. Again, if the search tree value is known, the search terminates and the *SEARCHED ON OPP* state is entered. Assuming the game tree value is not known, the search continues indefinitely until the user enters his/her/it's move, when one of two possible actions can occur. If the move entered is the expected move, then the program's efforts at thinking on opponents time have not been wasted. In which case, the *SEARCHING* state is entered to continue the search on the program's time. However, if the entered move is unexpected, this search is terminated by entering the *SEARCH INTERRUPTED* state.

SEARCH INTERRUPTED

The objective of this state is to allow a clean backing up and termination of the search tree. Once the search has returned to the root position, either the

SEARCH COMPLETE or *SEARCHED ON OPP* state is entered. The actual state depends on whether the program was previously thinking on its own or the opponents time.

SEARCH COMPLETE

In this state the user is informed of the search result by way of a move or an announcement. Some search values are then reset and finally the *IDLE* state is entered.

SEARCHED ON OPP

This state records the search result and enters the *IDLE* state.

Please note, an explanation of the handling of user commands such as set up position for example, entered while searching on opponents time have been omitted for reasons of clarity. They are only minor side issues which do not affect the underlying design.

6.4.4 Evaluation

The evaluation criteria used in HIARCS were fully explained and illustrated in chapter 4. However, during the implementation and testing, a number of problem areas were observed. In this section I will explain some of the problems encountered and the resulting solutions.

The first major problem noticed was associated with the application of the evaluation function. The evaluation function is usually applied at the leaf or horizon nodes of the main search[3], after which a quiescence search is applied. The positional value calculated at the horizon nodes is carried forward into the quiescence search to enable the final values calculated at the leaf nodes of this search to have reasonably up to date positional values to go with the calculated material value. When pieces are captured during the quiescence search, both the material and positional value of the captured piece should be lost. However, this was not reflected in the positional value carried forward in the quiescence search. In this way the positional value started to become 'out of date'.

When the positional values of pieces are relatively low, this was not a major problem. However, unlike most chess programs, my evaluation function can

---

[3] except where $\alpha\beta$ values indicate that the position cannot affect the search result.

produce very high positional scores. Therefore, when highly positionally valued pieces are captured, the carried forward positional value was drastically out of date, causing serious problems to the search, both in efficiency and accuracy. For example, if a passed pawn positionally valued at 200 points (2 pawns), was captured during the quiescence search, the positional total should have been decreased by around 200 points.

The solution to this problem lay in incrementally updating the positional value during the quiescence search. To achieve this, whenever the full evaluation function is applied, all piece positional scores are saved in a general Pc/Sq table. As moves are made during the subsequent search this table is maintained and the positional value of any captured piece is recorded in the 'Game History' structure and used to incrementally update the positional score. It was also found necessary to include promotion moves into this process by removing the promoted pawns positional score. These enhancements noticeably improved the standard of play by the program.

Another area which caused some problems was the concept of tempo. The player to move should receive some positional bonus for having the move as this is advantageous[4].

I tried out various tempo values based on a direct ratio to the best positional increase caused by any single move from each players point of view. However, this caused problems because the maximum positional change over even a single move can be dramatic over a whole search. I finally used a constant value of 20 points for having the move. I still feel the tempo value should be related to positional values in some way, however, the value chosen seems to behave reasonably in the test games played.

In elementary endings, where a checkmate drive is required, e.g. king and rook versus king, a special routine was found necessary to force checkmate in an efficient manor. The routine used is very similar to the mop up routine used in Chess 4.5 [4].

Cases of insufficient mating material, e.g. king and knight v king, are also detected by the evaluation function and the draw value returned. This enhancement is very effective in the late endgame because it allows early termination of many variations, thereby reducing unnecessary search effort.

In order to ease the debugging of the evaluation function, all values were saved and displayed on request. This allowed numerous small bugs to be identified, which would have otherwise been very difficult to spot as they only slightly affected the final evaluation function score.

---

[4] except in zugzwang positions

Even in the final performance tested version, some of these positional diagnostics have been retained as they allow a valuable insight into why a particular move was chosen.

6.4.5 Time Allocation and Control

As I explained in chapter 3, I believe time allocation and control is an important aspect of a chess program's make up and one which does not seem to have been addressed properly in most chess programs.

At the *Computer Olympiad*, I often noticed how even the strongest manufacturer's programs took approximately the same time over all the moves, even obvious ones. The only time moves were played quickly was when either the machines were thinking on opponents time and an expected reply was made, or when checkmate was seen or when only one legal move existed.

The first thing to do is to analyse how very strong human chess players tackle the problem of time allocation. I did not find any previous research work which provided the detailed move/time analysis I required, so I decided to collate the information myself.

I decided to use the 1986 world championship match between Gary Kasparov and Anatoly Karpov [62] as my case study. This match was chosen for three main reasons:

1. Likely high quality of time allocation decisions.
2. Same players were involved in all games allowing player style to be analysed.
3. Detailed statistics giving time used for each move was available in chess magazine.

I analysed all 24 games of the match. For each game, moves by both sides up to the time control at move 40 were collated. This involved analysing approximately 2000 move times. As a result I was able to produce 5 different graphs detailing the average time spent on each move for:

- Kasparov playing white.
- Kasparov playing black.
- Karpov playing white.

- Karpov playing black.
- Combined time allocation.

The graphs produced can be found in appendix C.

Analysing these graphs it can be seen that both players have distinctly different styles on time control. Kasparov allocates a significant amount of time to the middle game, moves 12-28 in particular. Kasparov's time allocation usually has one very high peak around moves 16 to 20 and then tails off dramatically. Karpov on the other hand has a more uniform usage of time, although his time allocation also usually tails off steadily after move 26. Two reasons are likely for this tailing off:

- Bad time allocation during the middle game leading to a rush near the time control.
- Good planning during the late middle game allowing planned move sequences to be played quickly.

As this tailing off occurs in nearly all the games and the very high calibre of the players involved, I tend to think good planning is responsible.

Another interesting observation occurs when you super-impose the graphs Kasparov as white over Karpov as black and Karpov as white over Kasparov as black. At each peak on the white player's graph there is usually a trough at the corresponding move on the opposing black player's graph. This also occurs when the black player's peaks are compared with the graph for the opposing white player's moves following such peaks.

I think this clearly illustrates that each player makes significant use of their opponents thinking time.

Examining the combined graph for both Kasparov and Karpov, a relationship to a normal distribution can be seen. Overall peak remains at move 20.

¿From these graphs the following points about time allocation and control can be deduced:

- Thinking on your opponents time is important.
- During the opening, prepared moves are typically played quickly.
- A significant amount of time should be allocated to 'unknown' positions occurring in the opening after a prepared sequence of moves.

- Time should be allocated depending on the complexity and understanding of the current position.
- Time allocation should be dynamically altered as a search progresses.
- 'Obvious' moves should be made quickly.

Some of these points are obvious to any experienced tournament player, others are less so. Nevertheless it remains important to have one's suspicions re-enforced by research.

I will now explain how these points are addressed in HIARCS 5.0.

Thinking on Opponents Time

HIARCS thinks about its next move durings its opponent's thinking time. It does this by thinking about the position as if the opponent had made the expected reply. If the opponent actually makes the expected reply, HIARCS has gained time and can continue 'thinking' or reply immediately. If however, the opponent makes a different reply, HIARCS aborts its analysis and starts 'thinking' from the new position. In practice, the correct move seems to be predicted around 50% of the time.

The time spent thinking on the opponents time is taken into account when the expected reply is made.

Initial Time Allocation

Initially the response time (respTime) allocated to a move is calculated from the average time per move remaining to the next time control:

$$\text{respTime} = \text{timeLeft} \,/\, \text{movesLeft}$$

A maximum time (maxTime) is also initially calculated as follows:

$$\text{maxTime} = \text{timeLeft}$$

This is a special case to stop the program from losing on time.

Using only this approach, a problem may occur where the program having half an hour left for its last move will want to use its full time allocation. However, it may be better not to spend all the available time and save some for use during the next time control period. To avoid this problem; during the last 10 moves before a time control, respTime is modified as follows:

respTime = min(respTime, timeLeft2/movesLeft2)

where

respTime : is the initial response time set.

timeLeft2 : is the time left before the next but one time control.

movesLeft2 : is the number of moves left before the next but one time control.

After the program leaves its opening book it is faced with a position whose concepts must be grasped to enable the program to continue the thread of the opening. To help the program at this difficult stage, the response time is increased by between 0 and respTime depending on the number of moves since leaving the opening book, e.g. the first non book move is respTime, and the forth is 0.

Time Allocation Adjustment

The actual time which can be spent on a move can vary between respTime/64 and maxTime. Therefore, for a time control of 40 moves in 2 hours the minimum and maximum times could range from 3 seconds to 2 hours. Of course, if the search tree is terminal or thinking on opponents time has been used an immediate response may still occur.

The time spent thinking on opponents time is taken into account by this 'time window'. For example, if the program had spent 60 seconds thinking on opponents time and the expected reply was made, a new respTime of 62 seconds would only require 2 more seconds thought by the program, all other things being equal.

The actual time taken within this 'time window' will depend on various factors measuring how complex the move choice is. Depending on the search status, the time allocation is adjusted under 5 categories. Note, to facilitate the varying time allocation changes, the concept of extra time is introduced.

1. Best Move leads to a Loss: While no moves are found to avoid losing the game, the allocated time is set to the maxTime. This allows the program to continue searching until either mate is found or almost all the program's time expires. After all, the program is very likely to lose if it plays a move it knows will lead to a loss, therefore it might as well spend most of its time attempting to find an improvement. Of course, if the loss is found to be unavoidable the move which lasts out the longest will be played immediately.

2. New Iteration Begun: Before each new iteration begins the program calculates how long the iteration is expected to take based on the time taken over previous iterations. For example, in the middle game an iteration typically takes about 6 times the length of all previous iterations[5]. Additionally, an iteration commitment time is calculated. If the current response time will expire before the iteration commitment time, the search is aborted. However, if the program is committed to the iteration, it allows the extra time reasonably expected to achieve some results from the new iteration commitment. Note that the time previously taken thinking on opponents time is also taken into account.

$$\mathrm{extraTime} = \mathrm{iterationTime} - \mathrm{respTime} + \mathrm{oppTime}$$

3. Search Failed Low: When the search has failed low, the true value of the best move found is not known, it is only known that it is worse than the α value of the initial search window. In this case, a large amount of time is allocated to actually find a true score. Otherwise, if the best move found were played, it is likely to be a blunder.

$$\mathrm{extraTime} = 8 \cdot \mathrm{respTime}$$

4. Search Failed High: Like a fail low, when the search fails high the true score is not known. However, things are not so severe as with the fail low condition as it is known that the best move found is better than β. Therefore, some extra time is allocated to allow the true score to be determined, however, this time is not excessive, being set at:

$$\mathrm{extraTime} = 2 \cdot \mathrm{respTime}$$

5. Search Value in Aspiration Window: The actual best move value is known, therefore, time can be allocated directly from the value itself. The judgement is based on the difference between the best move's value and the score at the end of the last iteration. If this difference is less than or equal to -16, extra time is allocated as follows:

$$\mathrm{extraTime} = (\mid \mathrm{diff} \mid / \ 16 \cdot \mathrm{respTime}) \ / \ 8$$

---

[5] the fact that odd ply iterations take longer than even ones is also taken into account

For example, if the best move at this iteration appears to lose a pawn relative to the previous iteration (diff = −100), the extra time allocated, given respTime = 180 seconds and integer arithmetic, would be 135 seconds, making a total time of 5 minutes 15 seconds. For the loss of a bishop, the extra time would be about 8 minutes (total time 11 minutes).

This method allows extra time to be allocated as and when necessary. The loss of a piece is very likely to mean the loss of the game, so it is important to allocate a significant amount of time in order to search for a way of avoiding the loss. As can be seen the actual extra time allocated is directly proportional to the threatened loss. This approach also helps the program avoid the horizon effect by detecting when scores drop between iterations, and then allocating more time to see over the search horizon.

When the difference is greater than −16, no extra time is allocated. In fact, the time is reduced for numerous obvious moves. The following attributes are taken into account for time reduction:

- Whether the best move has maintained top position throughout all iterations.
- Whether a fail low or high has occurred during this search.
- Whether the best move is clearly better than all its siblings.
- Whether the move is a capture or recapture.
- Whether the move was part of the main line predicted from the previous search.
- Whether a move's value falls within a score range predicted by the previous search.

Given this information, the time allocation function can make a reasonable decision about what can safely be considered obvious.

Please note, this time allocation adjustment process is continually performed every time a new best move is found or an iteration completes.

Checking Time

Whenever the allocated time is adjusted, a new search expiry time exists. This expiry time is calculated from the response time, extra time, time spent thinking on opponents time and the max time. Using this expiry time, an alarm call is set up to signal the program when its search time has matured. Additionally, before setting a new alarm call time, the old 'alarm call' must be cancelled. This approach makes use of the UNIX system call *alarm*.

6.5 Efficiency Aspects

Efficiency is very important to any chess program. In fact, because of the unusually dominant factor of efficiency, chess programs are usually quite hard to follow. I have tried to make HIARCS as clear and easy to understand as possible, while keeping to the overall aims of efficiency.

The following sections explain program efficiency from three different angles:

6.5.1 Algorithms

Tree searching efficiency is of primary importance to the overall speed of the program. For example, the difference between applying $\alpha\beta$ to a well ordered tree and a badly ordered tree is many orders of magnitude. In HIARCS, particular attention has been paid to improving move ordering to therefore avoid exploration of many unnecessary nodes.

Other areas like move generation, for example, have been made more efficient by the use of suitable data structures.

6.5.2 Data Structures

Data structure composition makes a large contribution to overall program efficiency by allowing fast storage and retrieval of important information. The incremental attack tables and transposition table are obvious examples of this. The rather complex evaluation function would be just too expensive to compute were it not for the attack tables.

Without the use of a large easily accessible transposition table, much wasted effort would result from exploring duplicate subtrees. The improvements are particularly evident in the endgame, where a 5-fold speed improvement is not uncommon.

6.5.3 Code Production

Consideration must also be given to the code produced by the C compiler. It may have been noted how all my main data structure records are of a size divisible by 2. This allows the compiler to use a less time consuming shift operation, instead of multiplication, to access individual records.

Unsigned data types are used where possible to improve the efficiency of the addressing code produced. For example, when using a signed 16 bit integer as an index into an array, the compiler must produce an extra instruction to perform sign extensions (EXT.L on a MC68000) in a 32 bit addressing mode. With unsigned integers, none of this is necessary.

All constructs and expressions which produce time expensive machine code are avoided where possible. For example, few division operations are used.

Most functions use register variables for the most commonly used data objects. Of course, how often these variables are register resident depends on UNIX and the C Compiler.

Unfolding loops and producing linear code is another trick to extract more speed. However, I have avoided such practice on the whole to improve readability and ease development testing.

I have made a number of critical variables global to avoid using up time, passing large amounts of data on the stack. However, I have also employed some data hiding techniques where suitable to ease development and testing. The HashDriver module is an obvious example of this.

Individually, all these improvements are very small. However, when you consider that a number of code units are executed many millions of times, the overall effect can be quite impressive.

Chapter 7

Program Performance

In order to be able to evaluate the techniques employed in HIARCS 5.0, the program's strengths and weaknesses need to be assessed. Fortunately, chess provides many ways of testing a 'players' ability.

The ELO chess rating system [19] has been used world wide for over twenty years to estimate a player's ability in terms of rating points. Chess ratings are achieved by playing a number of games against already rated players. Using a rating/probability table, a rating can be calculated based on ones performance against the rated players. I will use a similar method to provide a rough estimate of HIARCS's chess rating.

Chess ratings can also be estimated from a set of test positions. Although this method is not used officially to rate players, it can be used to give an estimate of the tested players rating. Also, these tests allow a player's ability in various areas of chess, e.g. tactical play, to be measured. Therefore, such test positions will be useful to me in identifying the relative strengths and weaknesses of HIARCS.

The following tests will be performed to assess the program's overall chess rating, style, strength and weakness in the various areas of the game:

1. **Bratko-Kopec Tests**: These test positions will be used to estimate a rating and assess both the tactical and positional strengths of the program.
2. **Computer Opposition**: A match will be played between HIARCS and a rated computer opponent to illustrate its playing style and estimate its rating.

3. **Human Opposition**: A match will be played between HIARCS and a rated human opponent to find out how it copes with a human's style of play and to further estimate its rating.

4. **Knowledge versus Search**: A match will be played between HIARCS and a version of itself with dramatically reduced leaf node knowledge and no search extensions. This 'knowledge-less' version of HIARCS will be allowed to search 3 ply deeper than the more intelligent version. This should enable me to estimate what the knowledge is worth in terms of search depth.

All tests are performed with HIARCS 5.0 running on a SUN 3/80, which provides a rate of approximately 300 to 1200 (legal) nodes per second.

7.1 Bratko-Kopec Tests

A series of 24 test positions was devised by I. Bratko and D. Kopec to investigate and assess chess performance under two categories:

- Tactical play
- Positional play

The positional tests concentrate in particular on lever moves, which Bratko and Kopec believed, play an important role in a strong players ability.

This test set was chosen in particular because it has been applied to almost all the strongest chess programs in the world [41], therefore allowing useful comparisons to be made. In the original test, players were asked to give the best four moves in order of merit for each position. A correct move was then scored as follows: 1 point for a correct 1st choice, $\frac{1}{2}$ point for a correct 2nd choice, $\frac{1}{3}$ point for a correct 3rd choice and $\frac{1}{4}$ point for a correct 4th choice.

The problem with computer test subjects is that the $\alpha\beta$ algorithm does not provide an accurate best n choices, only the best move found so far. Therefore, Bratko and Kopec suggested the computers choice after 120, 180, 30 and 60 seconds should be used as the 1st, 2nd, 3rd and 4th choices respectively. The computer is therefore at a slight disadvantage as its 4 choices may all be the same.

The actual test positions can be found in appendix D. The results produced by HIARCS 5.0 are shown in figure 7.1.

No.	*Best move*	*1st*	*2nd*	*3rd*	*4th*	*Points*
1(T)	... Qd1+	Qd1+	forced mate			1
2(L)	d5	Re3	Re3	Re3	Re3	0
3(L)	... f5	f5	f5	Qd8	Qd8	1
4(L)	e6	e6	e6	e6	e6	1
5(T)	Nd5	Bf4	Bf4	Nd5	Nd5	$\frac{1}{3}$
6(L)	g6	g6	g6	c4	c4	1
7(T)	Nf6	Bb4	Bb4	Rg3	Rg3	0
8(L)	f5	f5	f5	f5	f5	1
9(L)	f5	Re1	Re1	Re1	Re1	0
10(T)	... Ne5	Qc5	Qc5	Qc5	Qc5	0
11(L)	f4	Nf5	Nf5	Nf5	Nf5	0
12(T)	... Bf5	Bf5	Bf5	Bf5	Bf5	1
13(L)	b4	Rc1	Rc1	Rc1	Rc1	0
14(T)	Qd2	Qd2	Qd2	Qd2	Qd2	1
15(T)	Qxg7+	Qxg7+	Qxg7+	Qxg7+	Qxg7+	1
16(T)	Ne4	Ne4	Ne4	Ne4	Ne4	1
17(L)	... h5	h5	h5	Nc5	h5	1
18(T)	... Nb3	f5	f5	f5	f5	0
19(T)	... Rxe4	Rxe4	Rxe4	c5	c5	1
20(L)	g4	Qh5	Qh5	Kb1	Kb1	0
21(T)	Nh6	Nh6	Nh6	Nh6	Nh6	1
22(T)	... Bxe4	Rd8	Rd8	Nc5	Nc5	0
23(L)	... f6	Bf5	Bf5	Bf5	Bf5	0
24(L)	f4	f4	f4	f4	f4	1
Total Tactical Score (T)						$7\frac{1}{3}$
Total Lever Score (L)						6
Total Score (S)						$13\frac{1}{3}$

Figure 7.1: HIARCS 5.0 results on the Bratko-Kopec Test Positions.

To determine the relative bias of the tested players, Bratko and Kopec devised the following equation: bias = $\frac{12 \cdot (T-L)}{S}$, where T is the tactical score, L is the lever score, and S is the total score. This equation will produce a positive bias value for tactically oriented results and a negative bias value for lever oriented results.

Using the results produced by HIARCS, this equation produces a value of +1.2. This indicates that the program has a slight tactical bias. However, when Bratko and Kopec produced a histogram of the bias values produced by the chess programs in their experiment, programs were found to have results in the +2.4 to +8 range[1]. The most frequent values were found in the +3 to +5 range. Therefore, when HIARCS's bias value is compared to these results, it can be seen that HIARCS has a well balanced tactical and positional ability.

It is interesting to note that of the humans tested, by far the most common values lay in the −5 to +3 range. Thus, indicating that most human chess players have a balanced ability with a slight emphasis on positional play.

Bratko and Kopec also gave a table giving rating ranges and corresponding mean values. The values for human players in the ranges 2000-2199 and 2200-2399 are given below:

Rating range	*Mean T*	*Mean L*	*12(T−L)/S*
2000-2199	6.67	4.84	1.73
2200-2399	7.81	8.07	0.2

Comparing these results with those of HIARCS, it can be seen that HIARCS has a slightly better tactical score (T) and a noticeably better lever score (L) than players in the 2000-2199 rating range. Perhaps more surprisingly, HIARCS displayed a better balance between tactical and positional ability than the mean results in that rating range.

For the higher rating range, HIARCS has a lower score, both tactically and positionally. It is interesting, however, that at this higher range the lever score is greater than the corresponding tactical score, re-enforcing my belief that positional ability plays a much greater role in high level chess than it does at the lower levels of ability.

Examining the Bratko-Kopec graph of rating vs score and the results above, an estimated rating can be assigned to HIARCS. As the test sample is small (24 positions), I will only say HIARCS's rating is likely to lie in the range 2030

---

[1] ignoring weak programs which had scores so low that confidence in a realistic bias value is impossible

to 2170. My past experience of chess programs being over-rated, leads me to expect that this rating is more likely to lie at the lower end of this range.

At the recent (1989) World Computer Chess Championship all the strongest programs were also tested on these Bratko-Kopec test positions [41]. I will examine the reasons why HIARCS could not solve some of the test positions and using the above paper, will compare its successes and failures with the results produced by the strongest chess machines in the world.

Looking at the actual moves selected in the tests, a number of points are worth highlighting. In test 1, HIARCS found the forced mate in 3 very quickly (2 seconds). In test 5, the correct move was identified quickly, although the program was not fully aware of all the important variations. After 1 minute 50 seconds its choice changed to Bf4 until at the next iteration, it reverted back to Nd5 once having examined the move to enough depth to fully 'understand' it. However, this took a total of about 4 minutes and could therefore not be included as a correct answer.

Of the tests not solved, the following reasons can be identified. Test 2 required the sacrifice of a pawn for positional compensation which HIARCS could not fully appreciate. However, no other program was able to solve this problem either.

In test 7, HIARCS was able to see the correct tactical move, however, it disdained the move in favour of the strong intermediary move Bb4. The top Mephisto machine also refused to play Nf6 and instead chose Qc1. I cannot believe this was because it could not see the tactical possibilities.

In tests 9, 11 and 13, HIARCS played strong positional moves, although not the best move. This highlights one of the major deficiencies of the Bratko-Kopec tests: no merit is given for chosen moves which, although not correct, are very strong alternatives. After all, positional moves are sometimes a matter of style and 'personal' preference.

Tests 10 and 22 were just too deep, tactically speaking, for HIARCS to see in the allotted two minutes. Tests 20 and 23 are very difficult positional tests which HIARCS, like almost all other strong programs was unable to solve.

I was particularly impressed by HIARCS's correct choice in tests 3 and 17. None of the strongest programs were able to solve test 17, indicating its difficulty. The speed at which HIARCS identified the correct move ... h5 (less than a minute), was encouraging.

HIARCS did achieve a lever score *equal* to Deep Thought, 1 point *ahead* of Hitech and only 1 point *less* than the highly positional Mephisto machine ! This result was very pleasing.

7.2 Computer Opposition

A match will be played between HIARCS 5.0 and the commercial chess machine, Mephisto Exclusive. This will allow a further rating to be estimated for HIARCS and will provide numerous examples of its actual playing style, strengths and weaknesses.

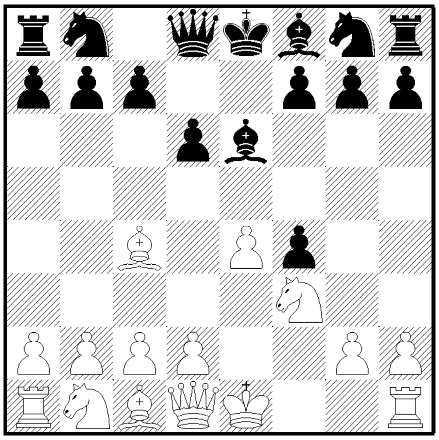
The Mephisto machine used was rated at ELO 1733 [24]. This machine is several years old and should not be confused with the latest commercial Mephistos which have significantly higher ratings (ELO 2100+).

I intend to play a 6 game match between the machines using a time control of 60 moves in 30 minutes followed by 20 moves in every subsequent 10 minutes.

Below is an annotated game from the match. It is one of the better games and illustrates many attributes of HIARCS's play. HIARCS's scores are shown periodically in brackets.

Game 1
White : **Mephisto Exclusive**
Black : **HIARCS 5.0**
Opening : Kings Gambit Accepted

1.	e4	e5
2.	f4	ef
3.	Nf3	d6
4.	Bc4	Be6 (+59)



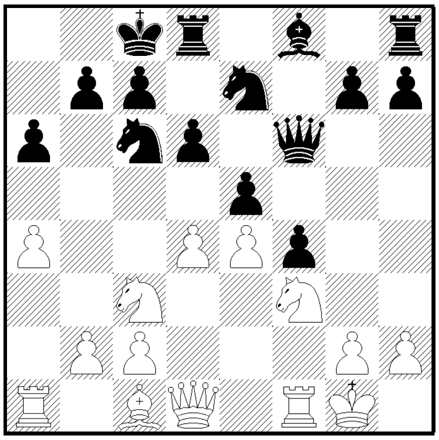
Position after 4. ... Be6

HIARCS was out of its opening book after 4. Bc4. BCO2 recommends 4. ... h6, however, HIARCS's choice of 4. ... Be6 may be better. Mephisto is also now out of its opening book.

5.	Bxe6	fe
6.	d4	Qf6 (+51)

With black's pawn structure in a heathly state, HIARCS aims to hang on to its advanced f-pawn. An alternative move was 6. ... g5, although this would limit HIARCS's castling options.

7.	o-o	Nc6
8.	Nc3	Nge7
9.	Nb5	o-o-o
10.	a4	a6
11.	Nc3	e5! (+148)



Position after 11. ... e5!

The stage is set. With the kings castled on opposite flanks, black's activity will concentrate on the king side while white will concentrate on the queen side. After move 9, HIARCS chose the plan of king side attack with pawns. Having gained a tempo by whites premature Nb5, HIARCS sets out to close the centre. It is important to have a closed central position *before* starting any flank attack.

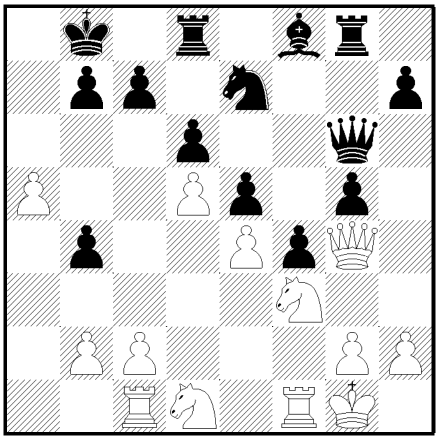
12. d5 Nb4

12. ... Nb8 followed by 13. ... Nd7 looks a better option. However, HIARCS considered whites best plan was a queen side pawn attack, hence it preferred to obstruct this by playing Nb4 stopping the early advance of the b pawn to b4.

13. Bd2 Qg6?! (+82)

HIARCS was intending to follow this up with h5, however an immediate g5! is much stronger.

14. Nh4 Qh6
15. Qg4+ Kb8
16. Rac1 g5
17. Nf3 Rg8
18. Nd1 a5
19. Bxb4 ab
20. a5 Qg6 (+179)



Position after 20. ... Qg6

HIARCS is now preparing a full scale pawn assault on the enemy king. The precarious position of the white queen means blacks advance is likely to proceed very quickly.

21. Nd2

Giving the white queen free space for her impending retreat.

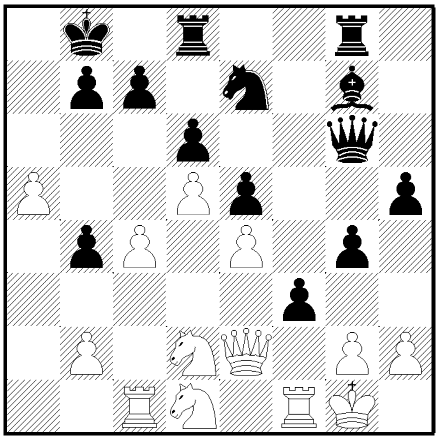
21. ... Bg7?!

Unnecessary, an immediate 21 ... h5 is very strong.

22. c4? h5
23. Qf3? g4
24. Qe2?!

White seems unaware of the impending danger. Blacks advantage has quickly become overwhelming.

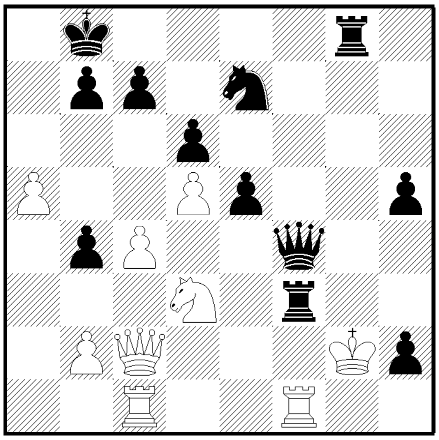
24. ... f3! (+287)



Position after 24. ... f3!

The f-pawn is immune to attack, if 25. gf gf+ wins.

25.	Qf2	Bh6
26.	Ne3	fg
27.	Kxg2	Rdf8
28.	Qe2	Rf3! (+423)
29.	Nd1	Bxd2
30.	Qxd2	Qxe4
31.	Qc2	Qf4
32.	Nf2	g3 (+793)
33.	Nd3	gh+



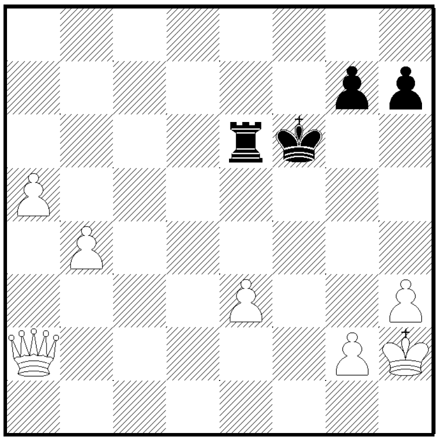
Position after 33. ... gh+

With this move, HIARCS announced mate in 5 moves after a 2 ply search taking just 1 second!

	34.	Kh1	Rxf1+
	35.	Rxf1	Qxf1+
	36.	Kxh2	Qg1+
	37.	Kh3	Qg3+ mate
		0 –	1
Times:		21:23	11:25

Game 2 was an interesting game. HIARCS had some trouble breaking through Mephisto's defence. HIARCS achieved success by 'sacrificing' the exchange of rook for knight and pawn on move 36. Another pawn followed shortly afterwards and HIARCS was left with 2 connected passed pawns on the queen side.

The position below was reached after 56 moves of game 2. HIARCS now played Qxe6+! simplifying into an even more easily won endgame. HIARCS's score increased from 867 to 1116 points after this move.



Position after 56 moves of game 2.

Game 3 was a strange game in which both programs made tactical blunders. However, HIARCS consolidated and recovered from its error, where as the Mephisto machine did not.

Game 4 produced a nice solid performance from HIARCS, announcing mate in 3 after just 37 moves.

Game 5 illustrated reasonably good endgame play, particularly the handling of the king and pawn ending.

Game 6 proved difficult for HIARCS. It tried a number of 'too clever' tactical moves which had longer term adverse effects. The black pawn on b2 allowed black to pose a number of problems. HIARCS managed to see through the complexity and had prepared a nice tactical way of saving its bishop on d7, something which caught Mephisto by surprise. By the endgame HIARCS was a pawn up, however, because of opposite coloured bishops was only able to draw. This is an example where specific endgame knowledge could have made a difference.

All the game records for this match can be found in appendix E. The final result table for this match can be seen below:

Players	*1*	*2*	*3*	*4*	*5*	*6*	*Total*
HIARCS 5.0	1	1	1	1	1	$\frac{1}{2}$	$5\frac{1}{2}$
Mephisto Exclusive	0	0	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$

Before I calculate HIARCS's performance rating, it is important to stress that ratings calculated at one time control are not easily transferred into ratings at other time controls. Therefore, some risk is associated with performing a match at a faster time control in order to extrapolate a rating for HIARCS at tournament times.

HIARCS won the match by the score $5\frac{1}{2}$ - $\frac{1}{2}$. Using the win probability tables suggested by Arpad Elo and found in [32], an estimated rating difference between HIARCS and Mephisto Exclusive can be looked up. The match score represents a win ratio of 92%. This converts into a rating difference of 401 points between the players. Therefore, given Mephisto Exclusive's rating of ELO 1733, HIARCS's performance rating in the test match is ELO 2134 (1733+401).

Taking into account the small sample of the test match and the time control at which it was conducted, this rating only represents a very rough estimate. However, the impressive score by which HIARCS won the match, clearly illustrates its superiority over the Mephisto machine.

7.3 Human Opposition

To determine how HIARCS performs against human opposition, I played a short two game match against HIARCS. When I last played active chess (1981/82) my rating was 1920 on the ELO scale. Since I have not played competitively for a long time, I am likely to be a bit 'rusty'. However, my interest in computer chess has allowed me to play 'friendly' games regularly, so I would expect my rating to be in the 1900-2000 ELO range. The particular advantage I have against computer opponents is a good understanding of their relative strengths and weaknesses.

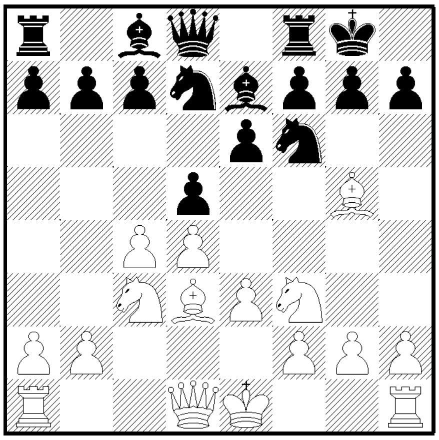
The match games were played at a full tournament time control of 40 moves in 2 hours, followed by 20 moves in each subsequent hour. Both game records can be found in appendix E. The final match score can be seen below:

Players	*1*	*2*	*Total*
HIARCS 5.0	1	0	1
Mark Uniacke	0	1	1

The following game is the first from the match. HIARCS's scores are shown periodically in brackets.

Game 1
White : **Mark Uniacke**
Black : **HIARCS 5.0**
Opening : Queens Gambit Declined

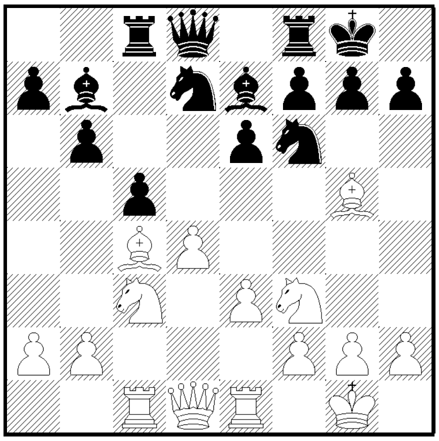
1.	d4	d5
2.	c4	e6
3.	Nc3	Nf6
4.	Bg5	Be7
5.	e3	o-o
6.	Nf3	Nbd7
7.	Bd3	



Position after 7. Bd3.

This move took HIARCS out of its opening book. 7. Rc1 would have kept HIARCS in book.

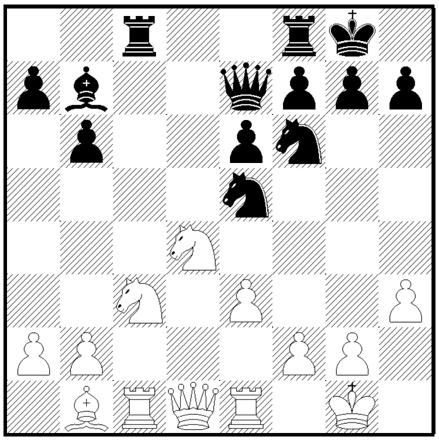
7.	...	dc (−69)
8.	Bxc4	b6 (!)
9.	o-o	Bb7
10.	Rc1	c5
11.	Re1	Rc8 (−9)



Position after 11. ... Rc8.

HIARCS's last 5 moves have been excellent positional moves. I was delighted that it decided to fianchetto black's normally bad queens bishop on the long diagonal. HIARCS seems to be following the correct concepts for the black side of a queens gambit.

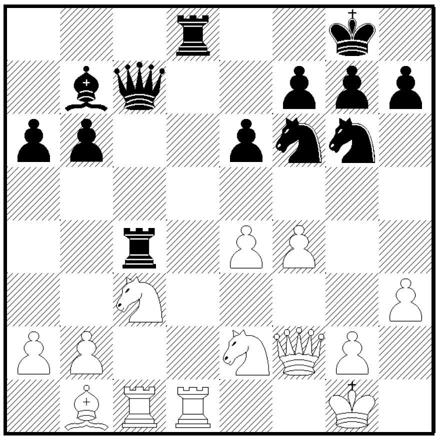
12.	Bd3	cd
13.	Nxd4	Ne5
14.	Bb1	Nfg4
15.	Bxe7	Qxe7
16.	h3	Nf6 (+64)



Position after 16. ... Nf6.

HIARCS has achieved at least equality from the opening. However, I was more afraid of 16. ... Qh4! to which I would have played 17. Qe2; not 17 hg Nxg4 which looks very promising e.g. 18. Nf3 Qxf2+ 19. Kh1 Qg3!

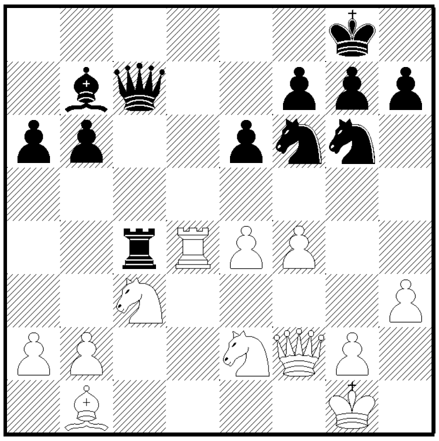
17.	Qe2	Rfd8
18.	Red1	Rc5!?
19.	e4	a6
20.	f4	Ng6
21.	Qf2	Qc7
22.	Nde2	Rc4! (+48)



Position after 22. ... Rc4!

I wanted to form a strong pawn center, however, in hindsight I feel I overstretched myself. The pawns on e4 and f4 are under extreme pressure.

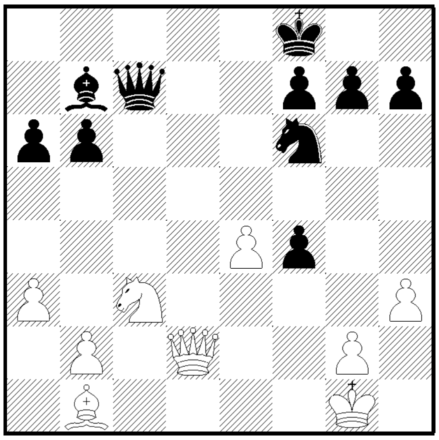
23.	Rxd8+	Qxd8
24.	Rd1	Qc7
25.	Rd4?	



Position after 25. Rd4?

I thought Rd4 would relieve the pressure on the white center pawns. After 25. ... Rxd4 26. Qxd4, I thought my f-pawn was secure because of the mate threat on d8, e.g. 26. ... Nxf4 27. Nxf4 Qxf4?? 28. Qd8+ Ne8 29. Qxe8+ mate. However, I missed 27. ... e5 which wins the pawn.

25.	...	Rxd4
26.	Qxd4	Nxf4!
27.	Nxf4	e5
28.	Qd2	ef
29.	a3	Kf8 (+133)



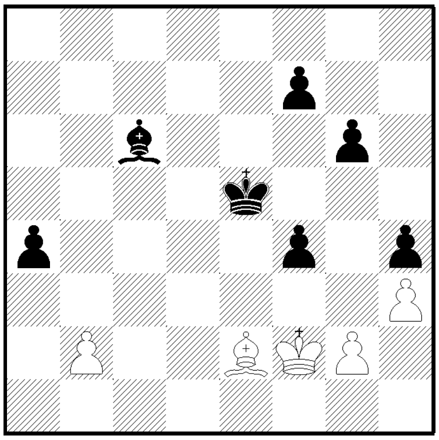
Position after 29. ... Kf8.

HIARCS is preparing to bring its king into play. I played 29. a3 to allow Ba2 at an opportune moment. In fact, I was hoping to play Nd5; swap off knights and obtain a supported passed d pawn. However, I missed 31. Qc5+ and the mate threat on g2 saving blacks b and f pawns. After which the game is lost.

30.	Nd5?	Nxd5
31.	ed	Qc5+
32.	Kh2?	Qxd5 (+201)
33.	Qxd5	Bxd5 (+248)

Blacks h-pawn is safe as Bxh7 is met by g6! The move 33. Qxd5 was bad, but I did not feel like a long hard queen and bishop ending which was lost anyway.

34.	Bd3	b5
35.	Kg1	h5
36.	Be2	g6
37.	a4	ba
38.	Bxa6	Bc6
39.	Kf2	Ke7
40.	Be2	Kf6
41.	Bc4	h4
42.	Be2	Ke5 (+292)



Position after 42. ... Ke5.

I cannot defend the b-pawn, g-pawn and hold back the eventual f5, g5 and g4, therefore, I decided to simplify and attempt to pick off blacks king side pawns while HIARCS attempts to promote its a-pawn. However, HIARCS did not allow me an inkling of counter play.

	43.	Bf3	Bxf3 (+315)
	44.	Kxf3	Kf5
	45.	Kf2	Ke4
	46.	Ke2	f5 (+401)
	47.	Kf2	Kd3 (+670)
	48.	Kf3	g5 (+973)
		resigns	
		0 –	1
Times:		1:46:33	1:59:47

HIARCS played this game very solidly, punishing my mistakes efficiently. I think it would be hard to tell whether a computer of human was black in this game.

In game 2, I was also under some pressure, however, I slowly manoeuvred my 2 knights into active positions on the queen side. I was then able to break the position open to my advantage. I eventually reached the ending, a queen for rook and bishop up. I played well and was able to simplify into an easily won king and pawn ending.

Although HIARCS chose a king side attack plan during this game, it did not execute it efficiently. By the time HIARCS had achieved an open h-file for its rooks, my own plan was already well developed. This illustrates that the planning mechanism used in HIARCS still requires a lot of tuning and enhancement.

I will not use this match to calculate a chess rating as the sample is too small. However, it was clear from playing HIARCS, that it has a distinctive, uncomputer like, playing style and would make a worthy opponent for any player rated at around 2000 ELO.

7.4 Knowledge versus Search

HIARCS 5.0 contains a significant amount of chess knowledge compared with the typical Shannon type A program. Much of the chess knowledge is applied at the horizon nodes of the main search. Although this approach allows application of up to date positional knowledge, it is quite time consuming, and therefore results in slightly shallower searches being performed.

It would therefore be interesting to classify what the knowledge is 'worth' in terms of search depth. This 'worth' could then be compared to its cost to determine whether the knowledge is cost effective.

I have compared knowledge and search by performing a 4 game match between HIARCS 5.0 and a version of itself containing only piece square tables for its horizon node evaluations and no search extension heuristics. This special version of HIARCS will be referred to as *NOKNOW*.

In the match, NOKNOW was allowed to perform a full width 6 ply search and HIARCS 5.0 was restricted to a fixed depth 3 ply search. NOKNOW took between 30 seconds and 20 minutes over each move, whereas HIARCS took around 1 to 10 seconds per move. Thinking on opponents time was disabled during the match. In the first two games both program were allowed to consult their opening book; in last two games, NOKNOW was not allowed to consult its opening book. All game records for this match can be found in appendix E. The final match score can be seen below:

Players	*1*	*2*	*3*	*4*	*Total*
HIARCS 5.0, 3 ply	$\frac{1}{2}$	0	0	1	$1\frac{1}{2}$
NOKNOW, 6 ply	$\frac{1}{2}$	1	1	0	$2\frac{1}{2}$

When considering these results, a number of factors must be taken into account:

- NOKNOW is not a tuned brute force searcher, hence its speed could be significantly improved.
- NOKNOW did make use of HIARCS's extensive move ordering algorithm and incremental evaluation table. This, to some extent, compensated for its lack of tuning.
- HIARCS was unable to use its superior time control algorithm for the match since it was limited to a fixed depth search. Hence, HIARCS 5.0 was unable to expend extra time and depth searching moves it felt were critical.

The match score seems to indicate that HIARCS's positional and tactical knowledge is almost worth 3 extra plies of search. Each additional ply would normally require a 6 fold increase in time[2], therefore, this result is very significant.

---

[2] odd ply's usually require more search effort than even ply's

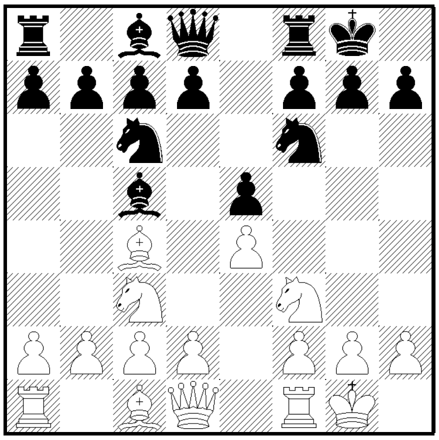
The chess knowledge contained in HIARCS 5.0 seems to cost around a 2-3 fold increase in time. This additional time would not be enough for even 1 extra ply of full width search. Therefore, HIARCS's knowledge appears to be very cost effective when compared with simply investing more search effort.

7.5 Internal Statistics

Various statistics about how the program spends its time have been calculated.

HIARCS 5.0 searches between 300 and 1200 nodes per second. The higher speeds are achieved in the endgame. Unlike most other programs, only legal nodes are counted in this node count.

When playing at tournament rates, HIARCS typically searches to 5 or 6 plies in the opening, 5 plies in the middle game and anything between 6 and 13 plies in the endgame.



ABF calculation: White to move.

The Average Branching Factor **(ABF)** is an extremely important measure as it defines how efficient the tree search is and hence what depths are achievable. It can be estimated by counting the total nodes, tn, explored in a full width search at depths $\leq n$, and using ABF $= tn^{1/n}$.

In the test position above, HIARCS 5.0 explored 168992 nodes within a 6 ply full width search. This equates to an ABF of 7.43. For a selective search to visit fewer nodes, it would have to expand less than 7.43 nodes from each position. Of course, such a selective search could still miss many critical variations within the 6 plies; a problem HIARCS's full width search cannot suffer from. This is the major advantage of a backward pruning algorithm such as $\alpha\beta$ when compared with forward pruning.

This ABF includes the totals for each iteration. The ABF for the final iteration, which *only* explored 125102 nodes, is 7.07. The minimal $\alpha\beta$ search tree for the position would be of the order $2 \cdot 38^{6/2} - 1 = 109743$. The fact that HIARCS's final iteration search was within 14% of this minimal tree illustrates the effectiveness of using the combination of iterative deepening, search tables and interior node move ordering.

Occasionally, searches have been found to explore as few or fewer nodes than the minimal $\alpha\beta$ tree would suggest. This is achieved because the transposition table allows numerous, sometimes large, subtrees to be pruned which would otherwise require searching.

It has also been found that it is not uncommon for HIARCS 5.0 to explore variations to a depth of 15 to 20 ply. Very occasionally, the search may even reach the maximum depth of 32 ply. These extremely deep searches are infrequent and therefore do not usually cause a large search 'explosion'. However, they occasionally allow HIARCS to discover deep combinations which would otherwise be impossible with a more depth restricted search.

Below is given a breakdown of how HIARCS spends its time:

Activity	*% of total time*
Positional Analysis	44.4 %
Making/Unmaking Moves	19.4 %
Updating Attack Tables	14.9 %
Move Generation	9.3 %
Tree Search Control	6.5 %
Everything else	5.5 %

As can be seen from these figures, HIARCS spends an extraordinary amount of time evaluating positions. Compared to more conventional programs, which spend most of their time generating and searching moves, the difference in program philosophy is obvious.

7.6 Summary of Performance

HIARCS 5.0 performs at or near the expert level of 2000 ELO. It displays a good balance between positional and tactical play which makes it an interesting, more human like, opponent. However, HIARCS 5.0 has a number of weaknesses both positionally and tactically. Some of these weaknesses can be corrected by extensive tuning of the evaluation function and search extensions, others require complex higher level knowledge to be included in the program.

HIARCS plays a competent tactical game and excels in identifying critical variations involving checks and captures. This is not surprising considering the type of search extensions included.

Unlike most other chess programs, HIARCS's improvement as time increases is more pronounced than would normally be expected. I believe this is because its search extensions have more scope to expand lines and its positional knowledge has more effect as the search depth increases. For example, a 1 ply increase in full width depth may allow HIARCS to extend a critical variation 2 or 3 plies.

The use of general plans to guide the search by increasing the desirability of certain aspects of a position has on the whole improved performance and given some direction to the play. However, there is much room for improvement in this area.

I have found that the time control algorithm has played an important role in helping HIARCS identify difficult moves and allowed extra time to be spent where it is most needed. On several occasions this has allowed HIARCS to improve its move selection.

The combination of the transposition table and search extensions has caused serious problems. Occasionally, the transposition table causes early termination of interesting variations. The reason for this lies in the fact that at each level in the tree, each node has no way of knowing whether variations will be extended at nodes in their subtree. Hence, when the transposition table is probed, the required search depth may be incorrect because it does not take account of possible extensions n ply down the subtree. Therefore, the transposition table may retrieve an entry which satisfies the depth requirement at this node, but would not satisfy the 'true' depth requirement if search extensions were to occur in the subtree.

In tactical positions, this often causes HIARCS to require 1 or 2 more search iterations than would normally be expected. This is most noticeable when HIARCS's search fails high, indicating it has found a clearly better move, only for the re-search to terminate early because of a premature entry match.

The solution to this problem lies in dramatically expanding each transposition table entry to store information about extensions in the subtree below the node responsible for storing the entry. On retrieval, this information can then be used to determine more accurately whether an entry should cause termination.

The cost effectiveness of each evaluation definition is yet to be determined. I suspect, most criteria will need modification of some sort, while others may even be unrequired. Certainly new criteria are required for significant performance improvement.

Chapter 8

Conclusion

This chapter concludes the report by reviewing what has been investigated, discussing the major achievements and exploring the future implications of my findings. Before I review the project and examine its achievements, let me re-iterate the project aims:

1. Overview the techniques used in current chess programs.
2. Develop an evaluation function which exhibits the positional ability of an expert level chess player (2000 ELO).
3. Explore fresh ways of improving the tactical ability of a chess program.
4. Implement the findings in a chess program (HIARCS 5.0).
5. Apply the Bratko-Kopec experiment to test the relative tactical and positional ability of the program.
6. Evaluate the program against both human and computer opposition.

8.1 Review

Before the project began, the aims of the project were set out and an overview of the required tasks necessary to meet the overall aims was devised. An overall time scale plan was produced which set time scales for each major activity. Regularly, during the project, progress was measured and reviewed against set objectives and time scales. This allowed me to identify and set sub-goals, and kept me aware of the implications any problems would have on the rest of my project. This in turn enabled me to report progress at my regular project meetings and allowed the project to develop smoothly.

In chapters 1 and 2 I introduced the subject of computer chess. This gave a good foundation for the more complex issues explored in future chapters.

Chapter 3 explored various techniques in detail. I introduced the $\alpha\beta$ tree pruning algorithm and a number of its major derivatives. Various techniques were discussed for improving tree search efficiency, e.g. move ordering, iterative deepening etc. Two forward pruning techniques were also examined. The subject of positional evaluation was explored and various methods of evaluation were examined including piece square tables, static evaluation and incremental evaluation. The importance of planning and the game phase was also discussed. The subject of learning in chess was investigated and some conclusions were drawn. The neglected area of time control was also explored.

In chapter 4, many aspects of my investigation were put to use in designing an evaluation function. I drew extensively from my own experience of playing chess to capture some of the heuristics I felt were important in move selection. This was very difficult because much of a chess player's ability seems to be subconscious. The writings of many great players and in particular, Aron Nimzowitsch, helped enormously at this difficult stage. I also felt it was important to develop a set of pseudo code functions for use in defining the evaluation criteria. These functions were broad enough to allow me to express a significant amount of chess knowledge, while being precise enough to be implemented.

In chapter 5, I attempted to define a method of tree searching which would enable most search effort to be expended in pursuit of good or interesting variations. My hope was to simulate the greatly variable depth of a strong human player's search tree, while compensating for weak selectivity by the exploration of many more variations. The search extensions produced, concentrate on forced or highly tactical elements of play which have some likelihood of affecting the search result. Such extra search intelligence was required to compensate for the inevitably shallower searches resulting from more time being spent applying my evaluation function.

Chapter 6 explains some of the implementation issues which arose. In particular it describes the important data structures and algorithms necessary in the program. The importance of program efficiency was also made clear. The results of an extensive analysis of time allocation and control are also explained in this chapter.

Chapter 7 describes how the performance of HIARCS 5.0 was measured. Both tactical and positional ability was quantified and an estimate of its chess rating was made after the application of many tests.

8.2 Achievements

Having investigated both tactical and positional evaluation; together with the development and implementation of ideas in a program, it is time to reflect on what has been achieved.

The ideas investigated have been implemented successfully in a chess program called HIARCS 5.0.

It has been clearly shown that tree searching significantly improves the tactical ability of a program. For example, a 2 ply search with rudimentary capture search is able to 'see' tactical concepts, such as a knight fork, which would otherwise go unseen by a mere 1 ply search. There appears no viable substitute for tree searching of some kind. Static tactical evaluation can be useful in some positions, but cannot be thought of as a search replacement.

More surprisingly, tree searching was seen to significantly improve positional ability. This was clearly illustrated when I developed an almost 'knowledge less' version of HIARCS, which contained only Pc/Sq tables. When this special version was searching 6 plies in the match against HIARCS 5.0, its positional play was found to be of a higher calibre than could be expected considering its rudimentary chess knowledge. The search was actually bridging some of the program's knowledge deficiency gap by allowing higher level concepts, such as multimove piece pathways, to be considered. This in part explains why Deep Thought is able to display a positional knowledge much higher than its evaluation function would suggest.

The amount of knowledge which can be bridged by tree searching depends on the depth of search. However, some chess concepts are so deep, 40 or 50 plies, that it will be almost impossible for searching alone to bridge such a gap. This explains why both the Hitech and Deep Thought teams are encountering diminishing returns from each additional ply of search. In order to avoid similar problems and reduce HIARCS's dependency on tree searching, I have instilled a significant amount of static knowledge in the way of positional criteria and dynamic tactical knowledge in the way of selective search extensions.

HIARCS's performance has clearly illustrated that extra knowledge can more than compensate for reduced search capacity. A pawn outside the quadrate of the opposing king is a clear example of this: without direct knowledge about king-pawn quadrates, a full width search program may require up to an 11 ply search to realise the pawn can queen. HIARCS 5.0 is capable of realising this after a mere 1 ply search because of its chess knowledge.

How far chess knowledge can be increased at the expense of searching, without degrading overall performance is still open to question. However, I believe full width searching can be reduced in favour of sophisticated search extensions. The level to which the full width portion of the search can be reduced greatly depends on how comprehensive and effective the search extensions are. Taking this approach to its extreme would result in a totally selective search of a narrow, deep game tree.

When long term chess knowledge is contained in the evaluation function, such as pawn structure information for example, each additional ply does not seem to suffer such diminishing returns as seen when the knowledge content is lower. This was made clear by the unusually large improvement in HIARCS's play between 30 seconds and 3 minutes a move.

The achievement of identifying and combining tactical uncertainty in the evaluation function has been beneficial. Allowing the search heuristics to make use of this information has helped reduce the programs susceptibility to the horizon effect.

Not only is HIARCS's positional play significantly improved, but the tactical standard produced from even shallow search depths is impressive. For example, on one occasion, HIARCS solved a "mate in 5 problem" while only at the first ply of full width search. The tactical improvement is only evident, however, in variations containing moves that can be identified by the extension heuristics. However, as the tactical coverage is quite large, the overall affect is beneficial.

A lot of progress has been made on the subject of time allocation and control. The resulting algorithm has certainly improved HIARCS's overall standard of play by allocating and adjusting time relative to the needs of the position.

The results produced on the Bratko-Kopec test positions clearly illustrate that HIARCS does display the tactical and positional ability of an expert level player, hence meeting the aims set out at the beginning of the project. What was most surprising about HIARCS's results on the test positions was its balanced positional and tactical ability. These results are particularly impressive when compared to the results obtained from other chess programs on the same test positions.

The results achieved in the matches played by HIARCS against both human and computer opposition re-enforce the results achieved in the Bratko-Kopec experiment.

8.3 Implications and Future Prospects

Over the last 10 years, full width Shannon type A programs have made great advances, partly due to advanced special purpose hardware and partly to sophisticated search algorithms such as PVS with singular extensions, for example. Very little or no progress has been made over the same period in developing purely knowledge based programs capable of high level play throughout the game. It seems that the problems of knowledge acquisition, representation and finally implementation are at present too great to make such knowledge based programs successful. Certainly, the key to these types of program remains in the better understanding of how strong players think.

There seems to be two main approaches to developing an intelligent selective search program.

1. Creating an intelligent, knowledge based program from scratch.
2. Gradual change to a full width Shannon type A program.

An approach of gradual change to existing programs seems to me to offer a better hope of eventual success. I feel a program like HIARCS 5.0 could represent the first step in a gradual transition from a sophisticated Shannon type A program to an intelligent Shannon type B/C program. The major advantage of such an approach is that the program can serve as a vehicle for experimenting with ideas. Hopefully, this will in turn enable the researcher to acquire a better understanding of the problems and issues involved.

The 3 major difficulties affecting the development of a good evaluation function remain knowing:

- **which** evaluation criteria are important.
- **when** evaluation criteria are important.
- **what** the relative merits of each evaluation criterion should be.

The solution to these problems lie in a significant contribution from high level players and the continual evolution of evaluation criteria. The problems associated with assigning discrete values to chess concepts is a major difficulty for the expert chess player. Applying value ranges instead of single values may be a possibility. This combined with a range and probability based search algorithm would be an interesting possibility. Learning may play an important role here, however, many of the problems outlined in chapter 3 must be solved if this is to be the case.

The combination of the transposition table and search extensions has proved problematical. The expansion of each table entry to include move sequences and extension information is required. As well as easing the problems outlined in section 7.6, this improvement could allow a human player to ask "what if ?" questions. The transposition table could be probed after a search to answer such questions with move sequences and scores. The $\alpha\beta$ algorithm would limit such a method because of possible cut offs, but still a step in the direction of understanding move selection would be made. Another possibility is the use of the evaluation function to answer questions about the position e.g. "which squares are weak around the white king ?". Such approaches would help strong chess players to identify program weaknesses and allow the program designer to work more closely with chess players having little programming knowledge.

HIARCS 5.0 contains many search extension heuristics which make a positive contribution to its tactical ability. The next step would not only be to enhance and tune these existing heuristics, but to introduce new heuristics. In particular, there seems no reason why positional type search extensions cannot be introduced. Although the associated difficulties are many, the rewards are great.

Over the coming months, I will endevour to correct many of the weaknesses identified in HIARCS 5.0's play. HIARCS 5.n will be entered in the chess tournament of the 2nd Computer Olympiad later this year. Hopefully, HIARCS will be using a 25Mhz 68030 for this event. The result this summer will be interesting.

Many of the techniques used in computer chess programs may be useful in other problem domains. I contacted Hans Berliner at CMU about the possibilities of applying some of the latest search techniques to Theorem Proving [8]. Although no one has built special purpose hardware, the 'chess approach' is apparently being looked upon favourably.

I hope this project has shown that both chess knowledge and searching can complement each other and that a good balance between the two can achieve results.

Bibliography

[1] G. M. Adelson-Velskiy and M. V. Donskoy (1975) : *Some Methods of Controlling the Tree Search in Chess Programs.* Artificial Intelligence, Vol 6 No 4, pp 361-371.

[2] G. M. Adelson-Velskiy, V. L. Arlazarov, A. R. Bitman, A. A. Zhivotovskiy and A. V. Uskov (1970) : *Programming a Computer to Play Chess.* Russian Math. Surveys, pp 221-262

[3] T. Anantharaman, M. Campbell, F. Hsu (1988) : *Singular Extensions : Adding Selectivity to Brute–Force Searching.* ICCA Journal, Vol 11 No 4, pp 135-143.

[4] L. R. Atkin and D. J. Slate (1977) : *CHESS 4.5 - The Northwestern University Chess Program.* Chess Skill in Man and Machine, P. W. Frey (ed), pp 82-118, Springer–Verlag, New York.

[5] D. F. Beal (1989) : *Experiments with the Null Move.* Advances in Computer Chess 5, D. F. Beal (ed), pp 65-79, Elsevier Science Publishers B.V., North Holland.

[6] H. J. Berliner (1979) : *The B* Tree Search Algorithm : A Best First Proof Procedure.* Artificial Intelligence, Vol 12, No 1, pp 23-40.

[7] H. J. Berliner (1989) : *Some Innovations Introduced by Hitech.* Advances in Computer Chess 5, D. F. Beal (ed), pp 283-293, Elsevier Science Publishers B.V., North Holland.

[8] H. J. Berliner : *Personal Communication, Feb 1990.*

[9] A. Bernstein and M. de V. Roberts (1958) : *Computer vs Chess Player.* Scientific American, 198, pp 96-105.

[10] P. Bettadapur (1986) : *Influence of Ordering on Capture Search.* ICCA Journal, Vol 9, No 4, pp 180-188.

[11] J. A. Birmingham and P. Kent (1977) : *Tree-searching and Tree-pruning Techniques.* Advances in Computer Chess 1, M. Clarke (ed), pp 89-107. Edinburgh University Press.

[12] I. Bratko and D. Kopec (1982) : *The Bratko–Kopec Experiment : A Comparison of Human and Computer Performance in Chess.* Advances in Computer Chess 3, Clarke M. (ed), pp 51-72. Pergamon Press, Oxford.

[13] M. S. Campbell and T. A. Marsland (1983) : *A Comparison of Minimax Tree Search Algorithms.* Artificial Intelligence, Vol 20 No 4, pp 347-367.

[14] J. H. Condon and K. Thompson (1982) : *Belle Chess Hardware.* Advances in Computer Chess 3, M. Clarke (ed), pp 45-54, Pergamon Press, Oxford.

[15] S. M. Cracraft (1984) : *Bitmap Move Generation in Chess.* ICCA Journal, Vol 7, No 3, pp 146-153.

[16] A. D. de Groot (1978) : *Thought and Choice in Chess.* Mouton-DeGruyter, 2nd edition.

[17] A. D. de Groot and R. W. Jongman (1966) : *Heuristics in perceptual processes.* International Congress of Psychology, Moscow, August 1966.

[18] C. Ebeling (1986) : *All the Right Moves – A VLSI Architecture for Chess.* PhD thesis, Carnegie–Mellon University.

[19] A. E. Elo (1978) : *The Ratings of Chess Players Past and Present.* Batsford Books, London.

[20] M. Euwe (1970) : *Computers and Chess.* The Encyclopaedia of Chess, A. Sunnucks (ed), St Martin's Press, New York.

[21] *Discussion with Tom Furstemburg, Fidelity Electronics, August 1989.*

[22] J. J. Gillogly (1972) : *The Technology Chess Program.* Artificial Intelligence, Vol 3, pp 145-163.

[23] R. D. Greenblatt, D. E. Eastlake III and S. D. Crocker (1967) : *The Greenblatt Chess Program.* Proc. FJCC,Vol 31, pp 801-810

[24] G. Grottling (1986) : *The Swedish rating list.* ICCA Journal, Vol 9, No 1, pp 56.

[25] J. Hamlen (1989) : *Tree Searching Algorithms and Computer Chess.* M.Sc. Thesis, University of London.

[26] L. R. Harris (1977) : *The Heuristic Search : An alternative to the alpha-beta minimax procedure.* Chess Skill in Man and Machine. P. W. Frey (ed), pp 157-166, Springer–Verlag, New York.

[27] D. Hartmann (1989) : *Notions of Evaluation Functions Tested against Grandmaster Games.* Advances in Computer Chess 5, D. F. Beal (ed), pp 91-141, Elsevier Science Publishers B.V., North Holland.

[28] E. Hearst (1977) : *Man and Machine : Chess Achievements and Chess Thinking.* Chess Skill in Man and Machine. P. W. Frey (ed), Springer–Verlag, New York.

[29] R. M. Hyatt (1983) : *Cray Blitz - A Computer Chess Playing Program.* M.Sc. Thesis.

[30] R. M. Hyatt (1984) : *Using Time Wisely.* ICCA Journal, Vol 7 No 1, pp 4-9.

[31] R. M. Hyatt, A. E. Gower and H. L. Nelson (1984) : *Cray Blitz.* Advances in Computer Chess 4, D. F. Beal (ed), pp 8-18, Pergamon Press, Oxford.

[32] B. M. Kazic (1980) : *The Chess Competitor's Handbook.* Batsford Books, London.

[33] D. E. Knuth and R. W. Moore (1975) : *An analysis of alpha-beta pruning.* Artificial Intelligence, Vol 6 No 4, pp 293-326.

[34] D. Kopec, M. Newborn and W. Yu (1986) : *Experiments in Chess Cognition.* Advances in Computer Chess 4, D. F. Beal (ed), pp 59-79. Pergamon Press, Oxford.

[35] A. Kotok (1962) : *A Chess Playing Program for the IBM 7090.* B.Sc. Thesis, MIT.

[36] D. Levy (1976) : *Chess and Computers.* Batsford Books, London.

[37] D. Levy (1979) : *Chess Programming.* Personal Computer World, May 1979, pp 25-27.

[38] D. Levy, D. Broughton, M. Taylor (1989) : *The SEX Algorithm in Computer Chess.* ICCA Journal, Vol 12 No 1, pp 10-21.

[39] T. A. Marsland (1985) : *Evaluation Function Factors.* ICCA Journal, Vol 8 No 2, pp 47-57.

[40] T. A. Marsland (1986) : *A Review of Game-Tree Pruning.* ICCA Journal, Vol 9 No 1, pp 3-19.

[41] T. A. Marsland (1990) : *The Bratko-Kopec Test Revisited.* ICCA Journal, Vol 13, No 1, pp 15-19.

[42] A. Newell, J. C. Shaw and H. A. Simon (1958) : *Chess Playing Programs and the Problem of Complexity.* IBM Journal of Research and Development, Oct 1958, pp 320-335.

[43] A. Nimzowitsch (1929) : *My System.* Modern Chess Masterpieces, Batsford Books, London.

[44] A. Nimzowitsch (1936) : *Chess Praxis.* Batsford Chess Classics, Batsford Books, London.

[45] T. Nitsche (1982) : *A Learning Chess Program.* Advances in Computer Chess 3, M. Clarke (ed), pp 113-120. Pergamon Press, Oxford.

[46] A. Reinefeld (1983) : *An Improvement of the Scout Tree-Search Algorithm.* ICCA Journal, Vol 6 No 4, pp 4-14.

[47] F. Reinfield (1945) : *Win At Chess.* Dover Books.

[48] A. L. Samuels (1959) : *Some Studies in Machine Learning using the Game of Checkers.* IBM Journal of Research and Development, 3, pp 211-229.

[49] A. L. Samuels (1967) : *Some Studies in Machine Learning using the Game of Checkers, II - Recent Progress.* IBM Journal of Research and Development, Vol 2 No 6, pp 601-617.

[50] J. Schaeffer (1983) : *The History Heuristic.* ICCA Journal, Vol 6, No 3, pp 16-19.

[51] J. Schaeffer (1984) : *The Relative Importance of Knowledge.* ICCA Journal, Vol 7 No 3, pp 138-145.

[52] G. Schrufer (1989) : *A Strategic Quiescence Search.* ICCA Journal, Vol 12 No 1, pp 3-9.

[53] C. E. Shannon (1950) : *Programming a Computer for Playing Chess.* Philosophical Mag, Vol 41, pp 256-275.

[54] A. Shapiro and T. Niblett (1982) : *Automatic Induction of Classification Rules for a Chess Endgame.* Advances in Computer Chess 3, M. Clarke (ed), pp 73-91. Pergamon Press, Oxford.

[55] H. A. Simon and K. A. Gilmartin (1973) : *A Simulation of Memory for Chess Positions.* Cognitive Psychology, 5, pp 29-46.

[56] H. A. Simon and W. G. Chase (1973) : *Skill in Chess.* American Scientist, 61, pp 394-403.

[57] S. S. Skiena (1986) : *An Overview of Machine Learning in Computer Chess.* ICCA Journal, Vol 9 No 1, pp 20-27.

[58] D. J. Slate (1987) : *A Chess Program that uses its Transposition Table to Learn from Experience.* ICCA Journal, Vol 10 No 2, pp 59-71.

[59] E. Slater (1950) : *Statistics for the Chess Computer and the Factor of Mobility.* Symposium on Information Theory, London : Ministry of Supply, pp 150-152.

[60] A. M. Turing (1950) : *Computing Machinery and Intelligence.* Mind, Vol 59, pp 433-460.

[61] D. E. Wilkins (1980) : *Using Patterns and Plans in Chess.* Artificial Intelligence, Vol 14, No 1, pp 165-203.

[62] B. H. Wood (1986) : *The World Chess Championship.* Chess Magazine, B. H. Wood (ed), August - November 86, Vol 51, Nos 987-994.

[63] A. L. Zobrist (1970) : *A Hashing Method with Applications for Game Playing.* Technical report 88, Computer Sciences Dept, Univ of Wisconsin, Madison WI, April 1970.

[64] A.L. Zobrist and F. R. Carlson, Jr (1973) : *An Advice-Taking Chess Computer.* Scientific American, 228, pp 92-105.

Appendix A

Glossary

ABF	**A**verage **B**ranching **F**actor.
AI	Artificial Intelligence.
ELO	A chess rating system used by the world chess federation.
en-prise	A piece is *en-prise* if it can be profitably captured.
HIARCS	**H**igher **I**ntelligence **A**uto **R**esponse **C**hess **S**ystem.
ply	One half move e.g. a move for white.
Positional Play	The manoeuvering of pieces to better squares.
Pc/Sq table	Piece Square Table.
PVS	**P**rinciple **V**ariation **S**earch.
Tactical Play	Sequence of threat and capture type moves.
Tempo	A single move as a unit of time in chess, e.g. the loss of a tempo would be caused by taking two moves to accomplish something which could be achieved in one move.
USCF	**U**nited **S**tates **C**hess **F**ederation rating system. Ratings on this scale are typically 100-150 points higher than the equivalent ELO rating, e.g. 2100 USCF $\approx$ 2000 ELO.

Appendix B

Pseudo Functions

This appendix contains a description of the pseudo functions used in the definition of my evaluation function. The functions are given in alphabetical order.

FUNCTION

Attacked : PieceList x Colour x Square $\Longrightarrow$ Boolean

Example

Attacked(pl,col,sq)

Description

Returns TRUE if square *sq* is attacked by any of the pieces in piecelist *pl* of colour *col*.
Otherwise FALSE.

FUNCTION

Attacking : Piece x Colour x Square x Square $\Longrightarrow$ Boolean

Example

Attacking(pc,col,sq1,sq2)

Description

Returns TRUE if piece *pc* of colour *col* on square *sq1* is attacking square *sq2*.
Otherwise FALSE.

FUNCTION

♯Attacks : PieceList x Colour x Square $\Longrightarrow$ Integer

Example

♯Attacks(pl,col,sq)

Description

Returns the number of attacks on square *sq* by any of the pieces in piecelist *pl* of colour *col*.

FUNCTION

Bishop : Colour x Colour $\Longrightarrow$ Boolean

Example

Bishop(col1,col2)

Description

Returns TRUE if colour *col1* still has a bishop which operates on squares of colour *col2*.

FUNCTION

Castled : Colour ⟹ Boolean

Example

Castled(col)

Description

Returns TRUE if colour *col* has castled. Otherwise FALSE.

FUNCTION

CentralPoint : Square ⟹ Square

Example

CentralPoint(sq)

Description

Returns an offset square, *csq* which represents the central point of the defensive kings position. Returns *csq* where

```
csq = sq
if (OnRank(sq) = 1)
    csq = csq + 8
else
    if (OnRank(sq) = 8)
        csq = csq - 8
    endif
endif
if (OnFile(sq) = 1)
    csq = csq + 1
else
    if (OnFile(sq) = 8)
        csq = csq - 1
    endif
endif
```

FUNCTION

Control : Square $\Longrightarrow$ Controller

Example

Control(sq)

Description

Returns the controller of square *sq*. The controller is defined to be the colour who attacks the square *sq* with more pieces of a specific type than the opponent and the number of all other attacks by lower valued pieces are equal.
Attacks are compared in the order : P,N,B,R,Q,K. If all attacks by each piece are equal in number then the controller is 'None'.

FUNCTION

♯DiffAttacks : Colour x Square $\Longrightarrow$ Integer

Example

♯DiffAttacks(col,sq)

Description

Returns : ♯Attacks([Any],col,sq) $-$♯Attacks([Any],Opp(col),sq)

FUNCTION

Distance : Square x Square ⟹ Integer

Example

Distance(sq1,sq2)

Description

Returns :
max(| OnRank(white,*sq1*) - OnRank(white,*sq2*) | , | OnFile(*sq1*) - OnFile(*sq2*) |)

FUNCTION

Doubled : Colour x File ⟹ Boolean

Example

Doubled(col,f)

Description

Returns TRUE if more than one pawn of colour *col* is on file *f*.
Otherwise FALSE.

FUNCTION

Escape : Colour x Square x Square $\Longrightarrow$ Boolean

Example

Escape(col,sq1,sq2)

Description

Returns TRUE if the king of colour *col* can escape a check from square *sq2* by a piece moved from square *sq1*.

FUNCTION

Has : PieceList x Colour $\Longrightarrow$ Integer

Example

Has(pl,col)

Description

Returns the number of pieces in piecelist *pl* of colour *col* which are still on the board.

FUNCTION

InsideQuadrate : Colour x Square $\Longrightarrow$ Boolean

Example

InsideQuadrate(col,sq)

Description

Returns TRUE if the king of colour *col* is inside the "queening square" or "quadrate" with respect to square *sq*, taking into account who has the first move.

FUNCTION

Isolated : Colour x File $\Longrightarrow$ Boolean

Example

Isolated(col,f)

Description

Returns TRUE if there does not exist a pawn of colour *col* on either file adjacent to file *f*.
Otherwise FALSE.

FUNCTION

KCRights : Colour $\Longrightarrow$ Boolean

Example

KCRights(col)

Description

Returns TRUE if king side castling rights for colour *col* are intact.
Otherwise FALSE.

FUNCTION

KingDistance : Square x Square $\Longrightarrow$ Integer

Example

KingDistance(sq1,sq2)

Description

Returns :
min(| OnRank(white,*sq1*) - OnRank(white,*sq2*) | , | OnFile(*sq1*) - OnFile(*sq2*) |)

FUNCTION

KingSq : Colour $\Longrightarrow$ square

Example

KingSq(col)

Description

Returns the square the king of colour *col* is currently on.

FUNCTION

LeverSq : File x Colour $\Longrightarrow$ Square

Example

LeverSq(f,col)

Description

Returns the first square where a pawn of colour *col* on file *f* would lever an enemy pawn on an adjacent file.

FUNCTION

♯Moves : Piece x Colour x Square ⟹ Integer

Example

♯Moves(pc,col,sq)

Description

Returns the number of moves possible by piece *pc* of colour *col* from square *sq*.

FUNCTION

Occupied : Square x Square ⟹ Integer

Example

Occupied(sq1,sq2)

Description

Returns the number of squares which are occupied between square *sq1* and *sq2* inclusive in one linear direction.

FUNCTION

Occupy : PieceList x Colour x Square $\Longrightarrow$ Boolean

Example

Occupy(pl,col,sq)

Description

Returns TRUE if one of the pieces in piecelist *pl* of colour *col* is on square *sq*. Otherwise FALSE.

FUNCTION

On : Square $\Longrightarrow$ Piece

Example

On(sq)

Description

Returns the piece type occupying square *sq* regardless of colour.

FUNCTION

OnFile : Square $\Longrightarrow$ File

Example

OnFile(sq)

Description

Returns the file of square *sq*.

FUNCTION

OnRank : Colour x Square $\Longrightarrow$ Rank

Example

OnRank(col,sq)

Description

Returns the rank of square *sq* with respect to colour *col*.

FUNCTION

OnWing : Flank x Square $\Longrightarrow$ Boolean

Example

OnWing(flk,sq)

Description

Returns TRUE if square *sq* is on flank *flk*.
Otherwise FALSE.

FUNCTION

Open : File $\Longrightarrow$ Boolean

Example

Open(f)

Description

Returns TRUE if file *f* has no pawns of any colour on it.
Otherwise FALSE.

FUNCTION

Opp : Colour $\Longrightarrow$ Colour

Example

Opp(col)

Description

Returns the opposite colour of colour *col*.

FUNCTION

OppositeFlanks : $\Longrightarrow$ Boolean

Example

OppositeFlanks

Description

Returns :
((OnFile(KingSq(White)) $\leq$ 3 **and** OnFile(KingSq(Black)) $\geq$ 5) **or**
(OnFile(KingSq(Black)) $\leq$ 3 **and** OnFile(KingSq(White)) $\geq$ 5))

FUNCTION

PSq : Colour x Square $\Longrightarrow$ Square

Example

PSq(col,sq)

Description

Returns the promotion square for a pawn on square *sq* of colour *col*.

FUNCTION

Passed : Colour x Square $\Longrightarrow$ Boolean

Example

Passed(col,sq)

Description

Returns TRUE if there does not exist an enemy pawn on the same or adjacent file, which can hinder the pawns advance to the promotion square.

FUNCTION

PawnHole : Colour x Square $\Longrightarrow$ Boolean

Example

PawnHole(col,sq)

Description

Returns TRUE if square *sq* cannot ever be attacked by pawns of colour *col*. Otherwise FALSE.

FUNCTION

PawnLever : File x Colour $\Longrightarrow$ Boolean

Example

PawnLever(f,col)

Description

Returns TRUE if a pawn of colour *col* can be a pawn lever on file *f* and incur no enemy pawn blockade before the pawn leverage square. Otherwise FALSE.

FUNCTION

♯Pawns : $\Longrightarrow$ Integer

Example

♯Pawns

Description

Returns the number of pawns currently on the board.

FUNCTION

Pin : Colour x Square x Square x Square $\Longrightarrow$ Boolean

Example

Pin(col,sq1,sq2,sq3)

Description

Returns TRUE if a piece of colour *col* on square *sq1* pins a piece of colour OPP(col) on square *sq2* against a piece also of colour Opp(col) on square *sq3*.

where On(sq1) $\in$ [Bishop,Rook,Queen] **and**
On(sq2) $\in$ [Knight,Bishop,Rook,Queen] **and**
On(sq3) $\in$ [Knight,Bishop,Rook,Queen,King].

FUNCTION

PotentialCheck : Colour x Square x Square $\Longrightarrow$ Boolean

Example

PotentialCheck(col,sq1,sq2)

Description

Returns TRUE if a piece of colour *col* on square *sq1* can move to square *sq2* and give check.

FUNCTION

QCRights : Colour $\Longrightarrow$ Boolean

Example

QCRights(col)

Description

Returns TRUE if queen side castling rights for colour *col* are intact.
Otherwise FALSE.

FUNCTION

$\sharp$SafeBlocks : Colour x Square x Square $\Longrightarrow$ Integer

Example

$\sharp$SafeBlocks(col,sq1,sq2)

Description

Returns the sum of :
$\sharp$DiffAttacks(col,sq_n)
if and only if ($\sharp$DiffAttacks(col,sq_n) > 1) and where sq_n is every square on a piece movement direction between *sq1* and *sq2* inclusive.

FUNCTION

SameDiagonal : Square x Square $\Longrightarrow$ Boolean

Example

SameDiagonal(sq1,sq2)

Description

Returns TRUE if squares *sq1* and *sq2* are on the same diagonal.
Otherwise FALSE.

FUNCTION

SemiOpen : Colour x File ⟹ Boolean

Example

SemiOpen(col,f)

Description

Returns TRUE if there are no pawns of colour *col* on file *f*.
Otherwise FALSE.

FUNCTION

Side : Square ⟹ Colour

Example

Side(sq)

Description

Returns the colour whose territory square *sq* is in.
Where each colour's territory is defined to be within its first 4 ranks.

FUNCTION

SqColour : Square $\Longrightarrow$ Colour

Example

SqColour(sq)

Description

Returns the colour of the square *sq*.

FUNCTION

Taxicab : Square x Square $\Longrightarrow$ Integer

Example

Taxicab(sq1,sq2)

Description

Returns :
(| OnRank(white,*sq1*) - OnRank(white,*sq2*) | + | OnFile(*sq1*) - OnFile(*sq2*) |)

FUNCTION

Trapped : Colour x Square $\Longrightarrow$ Boolean

Example

Trapped(col,sq)

Description

Returns TRUE if every square, *sq2*, a piece on square *sq* can move to, meets the following condition :
Control(sq2) = Opp(col)
Otherwise FALSE.

FUNCTION

$\sharp$UnMoved : Colour x Piece $\Longrightarrow$ Integer

Example

$\sharp$UnMoved(col,pc)

Description

Returns the number of pieces of colour *col* and type *pc* which have never moved.

FUNCTION

Value : Square $\Longrightarrow$ Integer

Example

Value(sq)

Description

Returns a value for the piece on square sq regardless of colour.
Defined by :

Pawn	= 1
Knight	= 3
Bishop	= 3
Rook	= 5
Queen	= 9
King	= 14

FUNCTION

Weak : Colour x Square $\Longrightarrow$ Boolean

Example

Weak(col,sq)

Description

Returns :

not (Attacked([Pawn],col,sq) **or**
Attacked([Pawn],col,$sq_{infront}$) **or**
Attacked([Pawn],col,sq_{behind})).

FUNCTION

Wing : Square $\Longrightarrow$ Flank

Example

Wing(sq)

Description

Returns the flank the square *sq* is on.

Pre–Defined types are as follows :

Boolean : TRUE | FALSE
Colour : White | Black
Controller : White | Black | None
File : Cardinal Range 1 to 8
Flank : QueenSide | KingSide
Integer : Integer Range
Piece : Pawn | Knight | Bishop | Rook | Queen | King | Any
PieceList : Set of Piece
Rank : Cardinal Range 1 to 8
Square : Cardinal Range 1 to 64

Note: Any is a 'wild card' piece.

Appendix C

Time Allocation Graphs

This appendix contains the time allocation graphs for Kasparov and Karpov in their 1986 world championship match.

Figure C.1: Time Allocation, Kasparov as White.

Figure C.2: Time Allocation, Kasparov as Black.

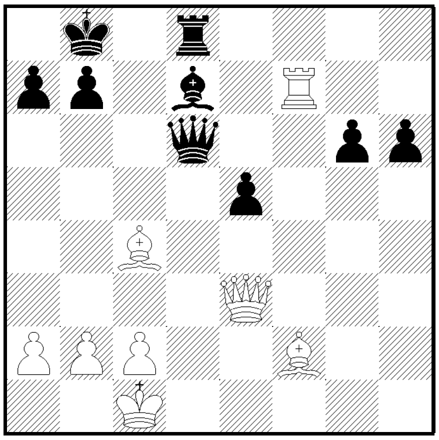
Figure C.3: Time Allocation, Karpov as White.

Figure C.4: Time Allocation, Karpov as Black.

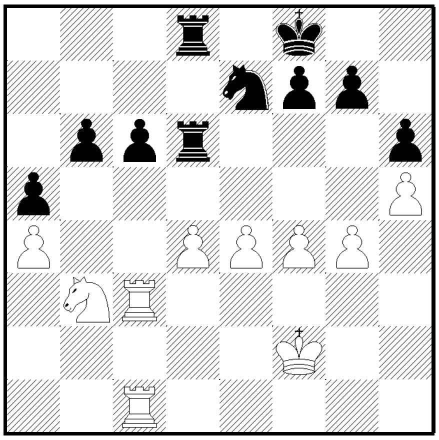
Figure C.5: Time Allocation, Kasparov and Karpov Combined.

Appendix D

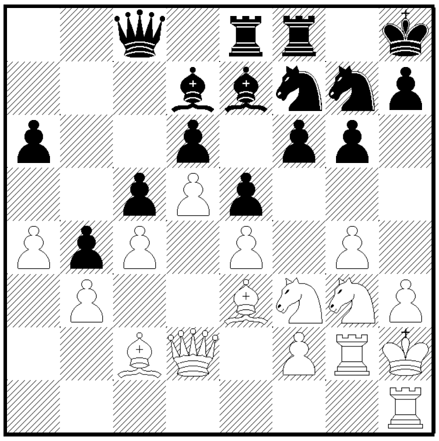
Bratko-Kopec Test Positions



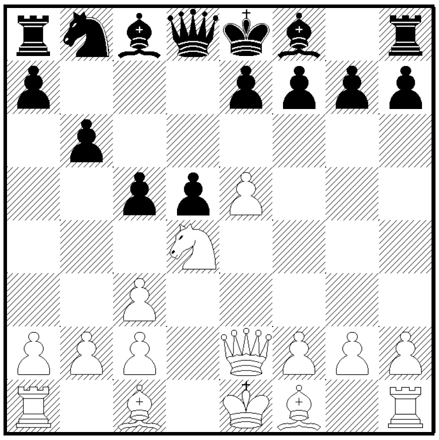
Test 1 (T): *Andersson v Knuttson*, Black to Play, **... Qd1+**.



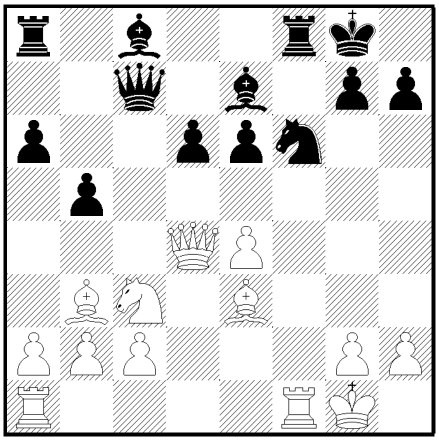
Test 2 (L): *Bogolyubow v Spielmann*, White to Play, **d5**.



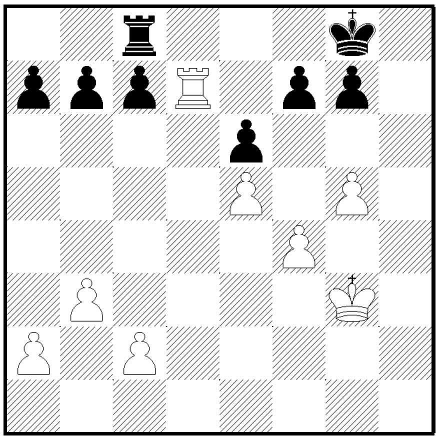
Test 3 (L): *Evans v Rossolimo*, Black to Play, **... f5**.



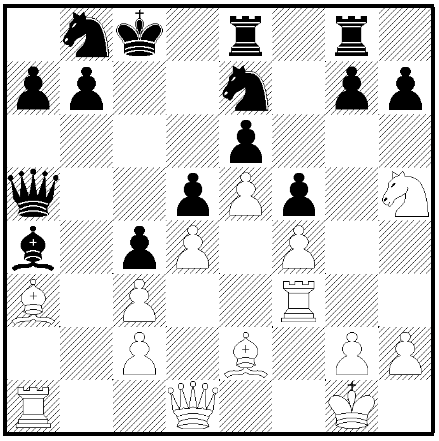
Test 4 (L): *Spielmann v Walter*, White to Play, **e6**.



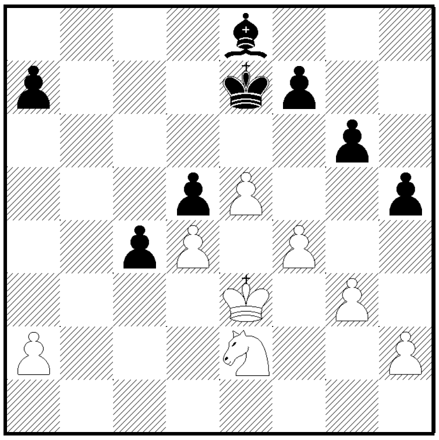
Test 5 (T): *Rogolewicz v Jarecz*, White to Play, **Nd5**.



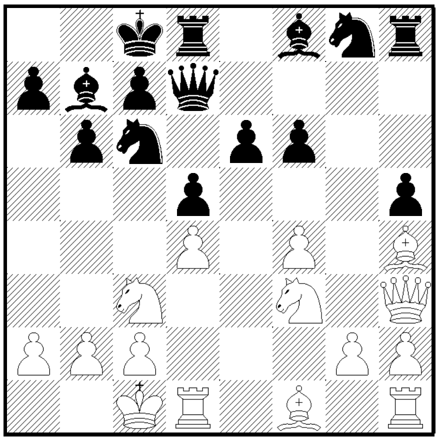
Test 6 (L): *Pachman D222*, White to Play, **g6**.



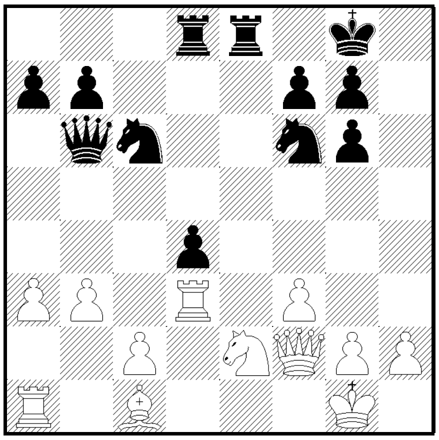
Test 7 (T): *Golyak v Gaiduk*, White to Play, **Nf6**.



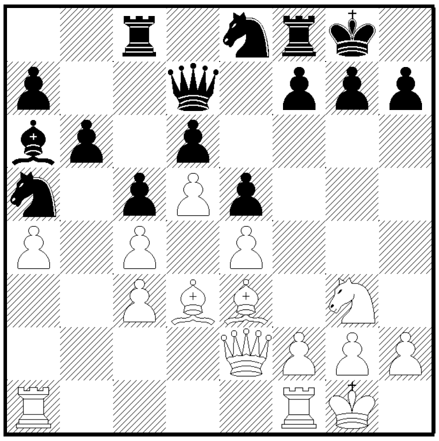
Test 8 (L): *Alekhine v Yates*, White to Play, **f5**.



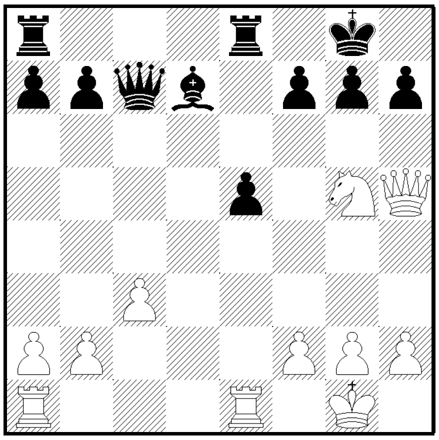
Test 9 (L): *Jansa v Ornstein*, White to Play, **f5**.



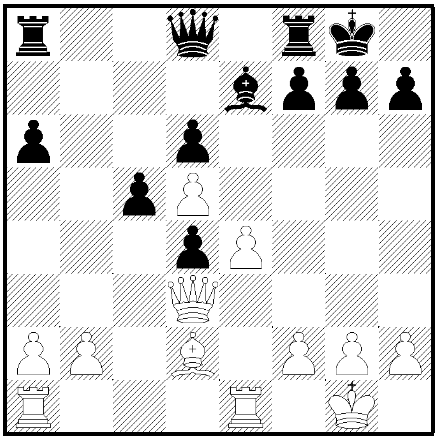
Test 10 (T): *Kabadzjan v Cibelasvili*, Black to Play, **... Ne5**.



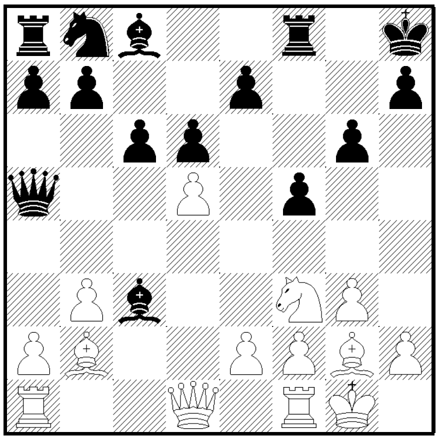
Test 11 (L): *Byrne v Kotov*, White to Play, **f4**.



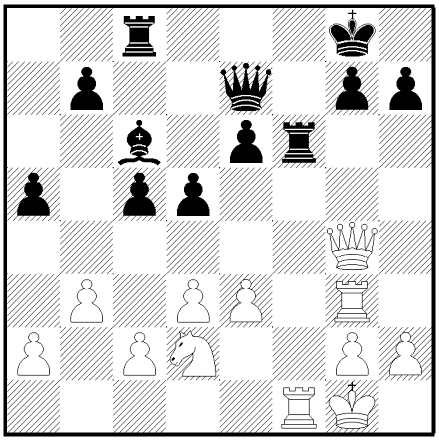
Test 12 (T): *Pachman D3*, Black to Play, **... Bf5**.



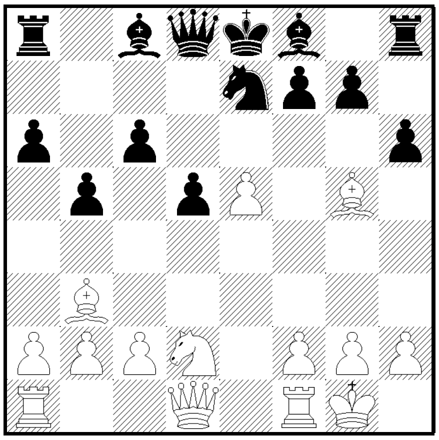
Test 13 (L): *Pfeiffer v Trifunovich*, White to Play, **b4**.



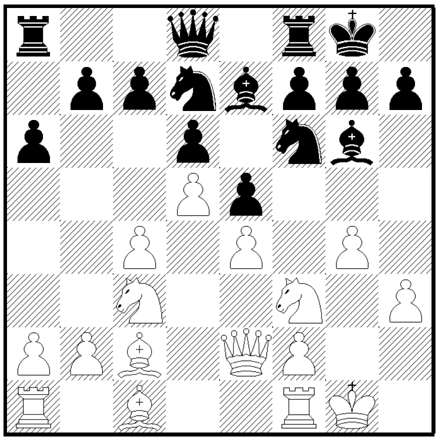
Test 14 (T): *Robatsch v Jansa*, White to Play, **Qd2 or Qe1**.



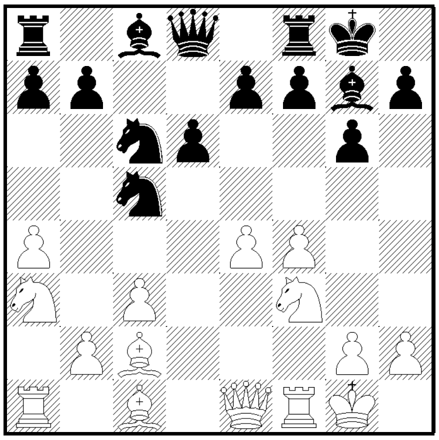
Test 15 (T): *Fischer v Mecking*, White to Play, **Qxg7+**.



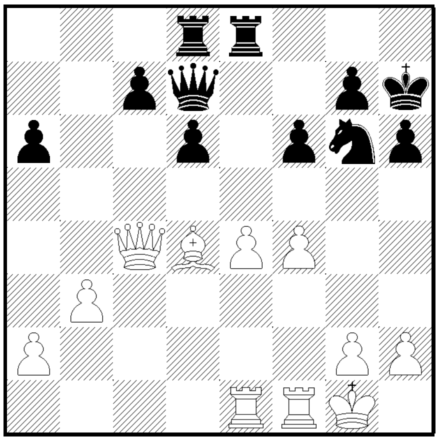
Test 16 (T): *Vasilchuk v Bobolovitch*, White to Play, **Ne4**.



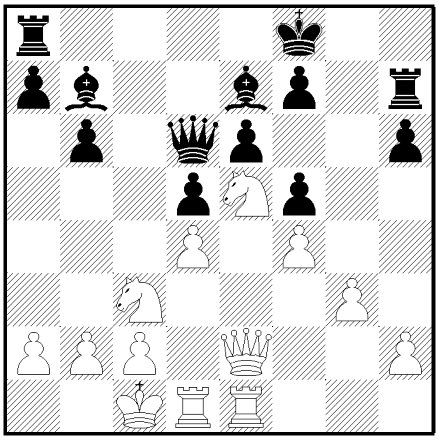
Test 17 (L): *Bisch v Kmoch*, Black to Play, **... h5**.



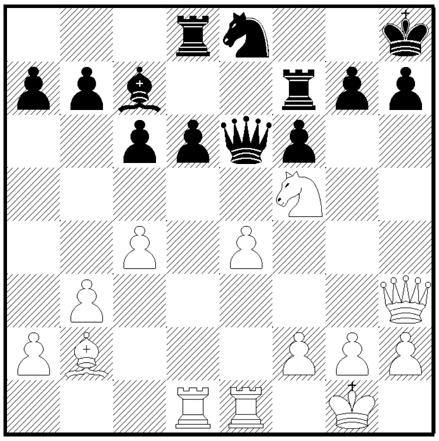
Test 18 (T): *Maric v Fischer*, Black to Play, **... Nb3**.



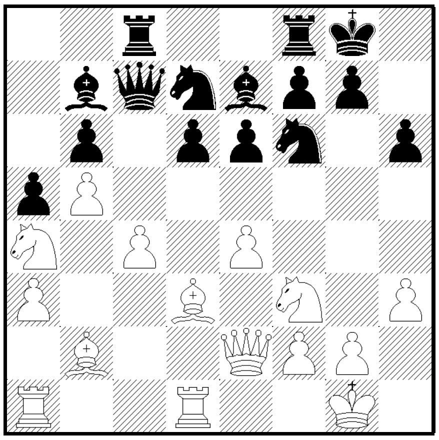
Test 19 (T): *Euwe v Keres*, Black to Play, **... Rxe4**.



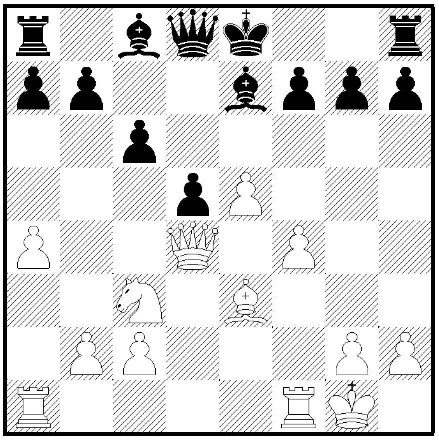
Test 20 (L): *Euwe v Flohr*, White to Play, **g4**.



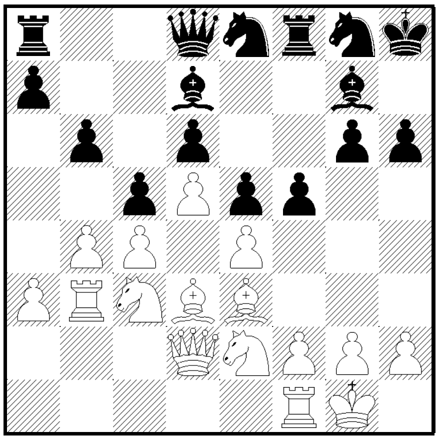
Test 21 (T): *Tarrasch v Blackburne*, White to Play, **Nh6**.



Test 22 (T): *Najdorf v Reshevsky*, Black to Play, **... Bxe4**.



Test 23 (L): *Jansa v Kavalek*, Black to Play, **... f6**.



Test 24 (L): *Szabo v Ivkov*, White to Play, **f4**.

Appendix E

Test Games Played

This appendix contains the test games played by HIARCS 5.0. The match scores were as follows:

Players	*1*	*2*	*3*	*4*	*5*	*6*	*Total*
HIARCS 5.0	1	1	1	1	1	$\frac{1}{2}$	$5\frac{1}{2}$
Mephisto Exclusive	0	0	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$

Players	*1*	*2*	*Total*
HIARCS 5.0	1	0	1
Mark Uniacke	0	1	1

Players	*1*	*2*	*3*	*4*	*Total*
HIARCS 5.0, 3 ply	$\frac{1}{2}$	0	0	1	$1\frac{1}{2}$
NOKNOW, 6 ply	$\frac{1}{2}$	1	1	0	$2\frac{1}{2}$

E.1 HIARCS 5.0 versus Mephisto Exclusive

E.1.1 Game 1

White : **Mephisto Exclusive**
Black : **HIARCS 5.0**
Opening : Kings Gambit Accepted

1.	e4	e5
2.	f4	ef
3.	Nf3	d6
4.	Bc4	Be6
5.	Bxe6	fe
6.	d4	Qf6
7.	o-o	Nc6
8.	Nc3	Nge7
9.	Nb5	o-o-o
10.	a4	a6
11.	Nc3	e5
12.	d5	Nb4
13.	Bd2	Qg6
14.	Nh4	Qh6
15.	Qg4+	Kb8
16.	Rac1	g5
17.	Nf3	Rg8
18.	Nd1	a5
19.	Bxb4	ab
20.	a5	Qg6
21.	Nd2	Bg7
22.	c4	h5
23.	Qf3	g4
24.	Qe2	f3
25.	Qf2	Bh6
26.	Ne3	fg
27.	Kxg2	Rdf8
28.	Qe2	Rf3
29.	Nd1	Bxd2
30.	Qxd2	Qxe4
31.	Qc2	Qf4
32.	Nf2	g3
33.	Nd3	gh+ (mate in 5)
34.	Kh1	Rxf1+
35.	Rxf1	Qxf1+
36.	Kxh2	Qg1+
37.	Kh3	Qg3+ mate
	0 –	1

E.1.2 Game 2

White : **HIARCS 5.0**
Black : **Mephisto Exclusive**
Opening : Bogo–Indian

1.	d4	Nf6
2.	c4	e6
3.	Nf3	Bb4
4.	Bd2	Bxd2
5.	Qxd2	o-o
6.	Nc3	d5
7.	e3	Ne4
8.	Nxe4	de
9.	Ng1	Nc6
10.	Ne2	Rb8
11.	Ng3	f5
12.	Be2	b6
13.	o-o	Bb7
14.	Rac1	Ne5
15.	Qc2	Nc6
16.	c5	Nb4
17.	Qb3	Nd5
18.	c6	Ba8
19.	Qa4	a5
20.	Qb3	Qh4
21.	Rfd1	Rbd8
22.	a4	Rd6
23.	Bb5	Nb4
24.	Rd2	Bxc6
25.	Bxc6	Nxc6
26.	Rdc2	Ne7
27.	Rxc7	Nd5
28.	Ra7	Qd8
29.	Qb5	Qg5
30.	Qa6	Qh4
31.	Rc8	Rdd8
32.	Rc6	Qh6
33.	Qe2	Rf6
34.	h3	Qg5
35.	Qb5	Qh4
36.	Rxb6	Nxb6
37.	Qxb6	Rc8
38.	Rxa5	Rc1+
39.	Kh2	Rc2
40.	Ra8	Kf7
41.	Ra7	Kg6
42.	Re7	Rxf2
43.	Rxe6	Kf7
44.	d5	Qh6
45.	Kg1	Rd2
46.	Nxf5	Rd1+
47.	Kh2	Rxe6
48.	Qb7+	Kf8
49.	Qb4+	Kf7
50.	Nxh6+	Rxh6
51.	Qxe4	Rd2
52.	b4	Rg6
53.	a5	Ra2
54.	d6	Rxd6
55.	Qc4+	Re6
56.	Qxa2	Kf6
57.	Qxe6+	Kxe6
58.	a6	Resigns

1 – 0

E.1.3 Game 3

White : **Mephisto Exclusive**
Black : **HIARCS 5.0**
Opening : Ruy Lopez, exchange variation

1.	e4	e5
2.	Nf3	Nc6
3.	Bb5	a6
4.	Bxc6	dc
5.	d3	Bg4
6.	h3	Bxf3
7.	Qxf3	Nf6
8.	o-o	Qd4
9.	Nc3	Qb4
10.	Qf5	Bd6
11.	Qg5	o-o
12.	Rb1	Rfe8
13.	a3	Qc5
14.	Be3	Qa5
15.	b4	h6
16.	Qg3	Qxa3
17.	Rb3	Qxb3
18.	cb	Bxb4
19.	Na2	Bf8
20.	f4	ef
21.	Bxf4	Bc5+
22.	Kh1	Nh5
23.	Qg4	Nxf4
24.	Rxf4	Bd6
25.	Rf6	h5
26.	Qxh5	gf
27.	h4	Rad8
28.	Nc3	Be5
29.	Qg4	Kf8
30.	d4	Rxd4
31.	Qf3	Red8
32.	Qg4	Rd3
33.	Ne2	Rb3
34.	g3	Rd2
35.	h5	Ke7
36.	h6	Rbb2
37.	Qf3	Rxe2
38.	Qa3	Ke6
39.	Qxb2	Bxb2
40.	h7	f5
41.	Resigns	

0 – 1

E.1.4 Game 4

White : **HIARCS 5.0**
Black : **Mephisto Exclusive**
Opening : Catalan

1.	d4	Nf6
2.	c4	e6
3.	g3	d5
4.	Bg2	dc
5.	Nf3	Nbd7
6.	Nc3	Bb4
7.	Bd2	o-o
8.	o-o	Be7
9.	Bf4	c5
10.	e3	cd
11.	Qxd4	Qb6
12.	Rb1	Qxd4
13.	Nxd4	Nc5
14.	Bc7	Bd7
15.	Rfd1	Ng4
16.	h3	Rfc8
17.	Ndb5	Bxb5
18.	Nxb5	Nh6
19.	Bd6	Bxd6
20.	Nxd6	Rcb8
21.	Nxc4	a5
22.	Rbc1	Nf5
23.	e4	Ne7
24.	Nb6	Ra6
25.	Rxc5	Rxb6
26.	Rd7	Nc6
27.	b3	e5
28.	Bf1	f6
29.	Bb5	Ra8
30.	Bc4+	Kh8
31.	Kg2	Nb4
32.	Rxa5	Na6
33.	Rad5	g6
34.	a4	Nb8
35.	Rf7	Na6
36.	Rd7	Re6
37.	Rxh7+	Resigns

(mate in 3)
1 – 0

E.1.5 Game 5

White : **Mephisto Exclusive**
Black : **HIARCS 5.0**
Opening : Sicilian Dragon, early white divergence

1.	e4	c5
2.	Nf3	d6
3.	Bc4	Nc6
4.	d4	Nf6
5.	d5	Na5
6.	Bb5+	Bd7
7.	Bxd7+	Qxd7
8.	Qd2	Nc4
9.	Qc3	Nb6
10.	Nfd2	e6
11.	de	fe
12.	o-o	d5
13.	ed	Nbxd5
14.	Qh3	Nf4
15.	Qe3	Nf6d5
16.	Qe4	Bd6
17.	Nc4	o-o
18.	Nxd6	Qxd6
19.	Na3	Qc6
20.	Re1	g5
21.	Bxf4	Rxf4
22.	Qxe6+	Qxe6
23.	Rxe6	Raf8
24.	c4	Nb6
25.	Re5	h6
26.	f3	Nxc4
27.	Rxc5	Nxa3
28.	ba	Ra4
29.	Rb1	b6
30.	Rc3	Rd8
31.	Rbc1	Rd2
32.	R1c2	Rxc2
33.	Rxc2	Rxa3
34.	Kf2	Kf7
35.	Re2	Kf6
36.	h3	Kf5
37.	Ke1	h5
38.	Kf2	Ra4
39.	Rd2	Ke5
40.	Re2	Kd5
41.	Rd2	Rd4
42.	Ke3	Rxd2
43.	Kxd2	h4
44.	Ke3	Ke5
45.	Kd3	Kf4
46.	Ke2	b5
47.	Kf2	a5
48.	Ke2	b4
49.	Kf2	a4
50.	g3+	hg+
51.	Ke2	a3
52.	h4	b3
53.	h5	ba
54.	h6	a1(Q)
55.	h7	a2 (mate in 7)
56.	h8(Q)	Qxh8
57.	Resigns	

0 – 1

E.1.6 Game 6

White : **HIARCS 5.0**
Black : **Mephisto Exclusive**
Opening : Petroff Defence

1.	e4	e5
2.	Nf3	Nf6
3.	Nxe5	d6
4.	Nf3	Nxe4
5.	d4	d5
6.	Bd3	Be7
7.	o-o	Nc6
8.	Re1	Bf5
9.	Nbd2	Nd6
10.	Nb3	o-o
11.	Nc5	b6
12.	Bf4	bc
13.	Bxd6	Qxd6
14.	Bxf5	Qf6
15.	Qd3	g6
16.	Bd7	Nxd4
17.	Nxd4	cd
18.	Rad1	Bb4
19.	Re2	c5
20.	c3	dc
21.	a3	Ba5
22.	Qxd5	cb
23.	Qxc5	Qc3
24.	Re5	f6
25.	Qd5+	Kh8
26.	Re3	Qc7
27.	h4	Rab8
28.	Rb3	Rxb3
29.	Qxb3	Rd8
30.	Qxb2	Qf4
31.	Qd4	Qxd4
32.	Rxd4	Kg7
33.	Kf1	Bb6
34.	Rd5	f5
35.	Ke2	h6
36.	h5	Kf6
37.	hg	f4
38.	Kf3	Bc7
39.	Kg4	Kxg6
40.	Bf5+	Kf6
41.	Rxd8	Bxd8
42.	Kxf4	Be7
43.	a4	Bc5
44.	f3	a5
45.	Ke4	h5
46.	f4	h4
47.	Bc8	Bf2
48.	Kf3	Bg3
49.	Kg4	Ke7
50.	Kg5	Kd6
51.	f5	Ke7
52.	f6+	Kf7
53.	Kf5	Kf8
54.	Be6	Ke8
55.	Bd5	Kf8
56.	f7	Ke7
57.	Kg6	Kf8
58.	Drawn	
	$\frac{1}{2}$ –	$\frac{1}{2}$

E.2 HIARCS 5.0 versus Mark Uniacke

E.2.1 Game 1

White : **Mark Uniacke**
Black : **HIARCS 5.0**
Opening : Queens Gambit Declined

1.	d4	d5
2.	c4	e6
3.	Nc3	Nf6
4.	Bg5	Be7
5.	e3	o-o
6.	Nf3	Nbd7
7.	Bd3	dc
8.	Bxc4	b6
9.	o-o	Bb7
10.	Rc1	c5
11.	Re1	Rc8
12.	Bd3	cd
13.	Nxd4	Ne5
14.	Bb1	Nfg4
15.	Bxe7	Qxe7
16.	h3	Nf6
17.	Qe2	Rfd8
18.	Red1	Rc5
19.	e4	a6
20.	f4	Ng6
21.	Qf2	Qc7
22.	Nde2	Rc4
23.	Rxd8	Qxd8
24.	Rd1	Qc7
25.	Rd4	Rxd4
26.	Qxd4	Nxf4
27.	Nxf4	e5
28.	Qd2	ef
29.	a3	Kf8
30.	Nd5	Nxd5
31.	ed	Qc5
32.	Kh2	Qxd5
33.	Qxd5	Bxd5
34.	Bd3	b5
35.	Kg1	h5
36.	Be2	g6
37.	a4	ba
38.	Bxa6	Bc6
39.	Kf2	Ke7
40.	Be2	Kf6
41.	Bc4	h4
42.	Be2	Ke5
43.	Bf3	Bxf3
44.	Kxf3	Kf5
45.	Kf2	Ke4
46.	Ke2	f5
47.	Kf2	Kd3
48.	Kf3	g5
49.	Resigns	
	0 –	1

E.2.2 Game 2

White : **HIARCS 5.0**
Black : **Mark Uniacke**
Opening : Pirc Defence

1.	e4	d6
2.	d4	Nf6
3.	Nc3	g6
4.	f4	Bg7
5.	Nf3	o-o
6.	Be3	Ng4
7.	Bd2	c5
8.	dc	dc
9.	e5	a6
10.	h3	Nh6
11.	Be3	Qa5
12.	Qd5	Nd7
13.	g4	Kh8
14.	o-o-o	e6
15.	Qd6	Re8
16.	Ng5	Bf8
17.	Qd2	Be7
18.	Nge4	b5
19.	Nd6	Bxd6
20.	Qxd6	Bb7
21.	Rh2	Rad8
22.	Bg2	Bxg2
23.	Rxg2	b4
24.	Nb1	Ng8
25.	a3	Ne7
26.	c4	Nc8
27.	Qc6	Ncb6
28.	Qe4	Nb8
29.	Rgg1	Na4
30.	Kc2	Qc7
31.	b3	Nb6
32.	h4	a5
33.	h5	Na6
34.	hg	hg
35.	Rh1+	Kg7
36.	Kc1	a4
37.	ba	Nxa4
38.	Qc2	Qc6
39.	Rh3	ba
40.	Nxa3	Nb4
41.	Rxd8	Nxc2
42.	Rxe8	Qxe8
43.	Kxc2	Qc6
44.	Bd2	Qe4+
45.	Kb3	Qe2
46.	Kc2	Qxg4
47.	Rb3	Qf5+
48.	Kc1	Qe4
49.	Kd1	Qd4
50.	Kc2	g5
51.	Rg3	Kf8
52.	fg	Qxe5
53.	Rd3	Qe4
54.	Be3	Nb2
55.	Kxb2	Qxd3
56.	Bxc5+	Kg7
57.	Be7	Kg6
58.	Bf6	e5
59.	Be7	e4
60.	Nc2	Qxc4

61. Bf6 Qd3
62. Nb4 Qd2+
63. Nc2 e3
64. Bc3 Qd3
65. Bd4 e2
66. Bf6 Qd2
67. Kb1 e1(Q)+
68. Nxe1 Qxe1+
69. Kc2 Qf2+
70. Kd3 Qf4
71. Ke2 Qxf6
72. gf Kxf6
73. Kf2 Kf5
74. Kf3 f6
75. Kg3 Ke4
76. Kf2 Kf4
77. Kg2 Ke3
78. Kg3 f5
79. Kg2 f4
80. Kf1 Kf3
81. Ke1 Kg2
82. Resigns
0 – 1

E.3 HIARCS 5.0 versus NOKNOW

E.3.1 Game 1

White : **NOKNOW, 6 ply**
Black : **HIARCS 5.0, 3 ply**
Opening : Queens Gambit Declined

1.	d4	d5
2.	c4	e6
3.	Nc3	Nf6
4.	Bg5	Be7
5.	e3	o-o
6.	Nf3	h6
7.	Bxf6	Bxf6
8.	Qb3	c6
9.	Rd1	Nd7
10.	Bd3	g6
11.	cd	ed
12.	e4	de
13.	Nxe4	Nb6
14.	Nxf6	Qxf6
15.	Be4	Bf5
16.	Bxf5	Qxf5
17.	o-o	Rde8
18.	Ne5	Kg7
19.	Qa3	f6
20.	Qxa7	fe
21.	Qxb6	Rf7
22.	Qc5	Qf4
23.	a4	Ree7
24.	b4	Ree8
25.	b5	Rf6
26.	Qa7	Re7
27.	bc	Rxc6
28.	Qb8	Rc4
29.	g3	Qf5
30.	de	Rc8
31.	Qa7	Qxe5
32.	Rb1	Qe4
33.	a5	Rc2
34.	a6	ba
35.	Qxa6	Kh7
36.	Qd6	Rc6
37.	Qd1	Rcc7
38.	Rb6	Rcd7
39.	Rd6	Qf5
40.	Rxd7	Rxd7
41.	Qe2	Qc5
42.	Kg2	Qd5+
43.	f3	Qd4
44.	f4	Qc3
45.	Kh3	Qd2
46.	Qxd2	Rxd2
47.	Rh1	Kg7
48.	Kg4	Kf6
49.	Kf3	Rc2
50.	Ke4	Re2+
51.	Kd3	Rb2
52.	Kc3	Rf2
53.	Kd4	Rd2+
54.	Kc4	Rc2+
55.	Kd5	Rd2+
56.	Kc6	g5
57.	fg+	Kxg5
58.	h3	Rc2+
59.	Kd5	Rg2
60.	g4	Kh4

61.	Ke4	Rg3
62.	Kf4	Rxh3
63.	Rg1	Rd3
64.	Kf5	Rd5+
65.	Kf6	Rd2
66.	Ke5	Kg5
67.	Rh1	Re2+
68.	Kd5	Re3
69.	Rh5+	Kxg4
70.	Rxh6	

Drawn

$\frac{1}{2} - \frac{1}{2}$

E.3.2 Game 2

White : **HIARCS 5.0, 3 ply**
Black : **NOKNOW, 6 ply**
Opening : Ruy Lopez

1.	e4	e5
2.	Nf3	Nc6
3.	Bb5	a6
4.	Ba4	Nf6
5.	o-o	Be7
6.	Re1	b5
7.	Bb3	d6
8.	c3	o-o
9.	h3	Bb7
10.	d4	Re8
11.	Ng5	Rf8
12.	f4	ed
13.	cd	h6
14.	Nf3	d5
15.	ed	Na5
16.	Bc2	Bxd5
17.	Nc3	Bxf3
18.	gf	c5
19.	Be3	Nc4
20.	dc	Nxe3
21.	Rxe3	Bxc5
22.	Qe1	Qd4
23.	Nd1	Nd5
24.	Resigns	
	0 –	1

E.3.3 Game 3

White : **NOKNOW, 6 ply**
Black : **HIARCS 5.0, 3 ply**
Opening : Reti (irregular)

1.	Nf3	d5
2.	Na3	Nc6
3.	c4	e5
4.	cd	Qxd5
5.	d3	Bb4+
6.	Bd2	Bg4
7.	Bxb4	Bxf3
8.	ef	Nxb4
9.	Nc4	Nf6
10.	Qe2	o-o
11.	Nxe5	Nxd3+
12.	Qxd3	Qxe5+
13.	Qe2	Qc5
14.	Qd2	Rfe8+
15.	Be2	Rad8
16.	Qc1	Qb5
17.	Qc2	Rxe2+
18.	Qxe2	Re8
19.	Qxe8+	Qxe8+
20.	Kd2	Qe5
21.	b3	Qg5+
22.	Ke2	Qxg2
23.	h4	Nd5
24.	Rag1	Nc3+
25.	Ke3	Qxh1
26.	Rxh1	c6
27.	Kd4	Nb5+
28.	Kc5	Nc3
29.	Re1	Kf8
30.	Kd4	Nxa2
31.	Ra1	Nb4
32.	Rxa7	b6
33.	Rc7	c5+
34.	Kc4	Nc2
35.	Rb7	Nd4
36.	f4	Ne6
37.	Rxb6	g6
38.	f3	Ke7
39.	Kd5	h5
40.	Ke4	Ng7
41.	Ke5	Nf5
42.	Rb7	Ke8
43.	Kf6	Nxh4
44.	Re7+	Kd8
45.	Rxf7	Nxf3
46.	Kxg6	Nd4
47.	Rb7	h4
48.	Kg5	Kc8
49.	Rh7	Nxb3
50.	Kxh4	c4
51.	Kg5	Nd4
52.	Kf6	c3
53.	Rh5	Kb7
54.	Rc5	c2
55.	Ke5	Nc6+
56.	Kd6	Nb4
57.	f5	c1(Q)
58.	Rxc1	Resigns

1 – 0

E.3.4 Game 4

White : **HIARCS 5.0, 3 ply**
Black : **NOKNOW, 6 ply**
Opening : Alekhines Defence

1.	e4	Nf6
2.	e5	Nd5
3.	d4	e6
4.	c4	Bb4+
5.	Bd2	Bxd2+
6.	Qxd2	Ne7
7.	Nc3	Nbc6
8.	Nf3	d6
9.	o-o-o	o-o
10.	Bd3	Nf5
11.	Bxf5	ef
12.	ed	cd
13.	Rde1	b6
14.	Nd5	Be6
15.	Ne3	Bd7
16.	h4	a5
17.	Nd5	Be6
18.	Nxb6	Qxb6
19.	d5	Qc5
20.	b3	Bd7
21.	dc	Bxc6
22.	Re7	d5
23.	Re5	Qa3+
24.	Qb2	Qxb2+
25.	Kxb2	dc
26.	bc	Be4
27.	Rd1	f6
28.	Re7	Rf7
29.	Re6	Rc7
30.	Kc3	Kf7
31.	Rb6	Rac8
32.	Rd4	Bc6
33.	Ra6	a4
34.	Ra5	g6
35.	h5	Ke6
36.	hg	hg
37.	Nh4	Rg7
38.	f4	Ra8
39.	Rc5	Ra6
40.	Rd2	Be4
41.	Nf3	Rg8
42.	Nd4+	Ke7
43.	Rc7+	Ke8
44.	Nb5	Ra8
45.	Nd6+	Kd8
46.	Rf7	Bc6
47.	Nxf5+	Ke8
48.	Nd6+	Kd8
49.	Nb5+	Ke8
50.	Rxf6	Bxb5
51.	cb	Ke7
52.	Rfd6	Rgf8
53.	g3	Rf6
54.	Rd7+	Ke8
55.	Kb4	Re6
56.	R7d6	Ke7
57.	Rxe6+	Kxe6
58.	b6	Rb8
59.	Kc5	Rc8+
60.	Kb5	a3

61.	Rd3	Ra8
62.	b7	Rb8
63.	Kb6	Rg8
64.	Rxa3	Rd8
65.	Re3+	Kf6
66.	Kc7	Rf8
67.	b8(Q)	Rxb8
68.	Kxb8	g5
69.	fg+	Kxg5
70.	a4	Resigns

1 – 0