UNIVERSITY OF ALBERTA

EXPERIMENTS IN SELECTIVE SEARCH EXTENSIONS

by

Chun Ye ©

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirement for the degree
of Master of Science

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA
FALL 1992

ISBN 0-315-77305-7

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Chun Ye

TITLE OF THESIS: Experiments in Selective Search Extensions

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1992

(Signed) [illegible]

Permanent Address:
Room 303, No. 1, Long 386
Shui Cheng Road
Shanghai, Shanghai 200335
People's Republic of China

May 6th, 1992

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Experiments in Selective Search Extensions** submitted by **Chun Ye** in partial fulfillment of the requirements for the degree of **Master of Science.**

T.A. Marsland

Prof. Tony Marsland, Supervisor

[illegible]

Dr. Jonathan Schaeffer

[illegible]

Prof. Douglas Hube (Physics)

April 16th, 1992

Abstract

Selective extensions, also known as selective deepening, is a means of enhancing the normal $\alpha\beta$ search algorithm where a fixed search depth is adopted. There are many ways for making the search horizon variable, and so make the search effort more worthwhile. In this thesis, we experiment with some of the most promising selective extension heuristics in the domain of Chinese Chess. The heuristics being explored include knowledge extensions (check evasions, recaptures, king-threats, piece-evading moves and strictly forced moves), singular extensions (PV-singular moves and fail-high singular moves), null move search (full-width null move search and null move quiescence search) and futility cutoffs (in all search phases wherever applicable). The *Abyss* Chinese Chess playing program was developed to carry out a series of experiments. The results of these experiments are presented and discussed in this thesis. The best combination of these heuristics for micro-ranged computers is also proposed. We conclude that these search extension heuristics successfully carry over from chess to a game like Chinese Chess. However, some of them certainly perform better than the others and are thus preferred.

Acknowledgement

First and foremost, I would like to thank my supervisor, Prof. Tony Marsland, for his endless support and academic inspirations for the research work reported here. I am grateful to Dr. Nick Jacobs for using *Xian*'s source code for comparisons; to Bin Zhang and Alan Sharpe for helping test the *Abyss* Chinese Chess program; to Haiying Wang for developing the X-Window interface. Also, I appreciate the time of Don Beal (Visiting Professor from Queen Mary and Westfield College, England) for discussion about the null move quiescence search heuristic. My other thanks go to the thesis committee members, Dr. Jonathan Schaeffer and Prof. Douglas Hube, for their valuable comments on improving the thesis. Finally, I wish to thank my wife Helen Huan Yang, whose understanding, help and love made my life meaningful.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Selective Extensions

For most of today's chess playing programs, a brute-force $\alpha\beta$ search algorithm is still adopted for move selections, yet almost all of them apply some criteria to extend particular moves during the search, and so make the effort spent more worthwhile. The reason is obvious: no matter how deep the search goes, moves beyond the game tree horizon may still be neglected. Selective extensions provide a means to alleviate this problem.

Generally speaking, such extensions can be divided into two categories: those which use domain specific knowledge and those which are based on information gathered from the search itself. For instance, chess playing programs usually extend the search by an extra ply when the side to move is in check, since checking usually forms a serious threat. The safety of a deeper search is worth the extra-cost, which isn't high since the number of replies to a checking move is small. This is one example of using domain-specific knowledge to extend the search depth. Other approaches include extending on recaptures [Ebeling 1987, pp. 101-102], pawn moves to the 6th and 7th rank in chess [Kaindl 1982, Scherzer *et al.* 1990], moves near the territory of the opponent's king [Anantharaman 1991], strictly forced moves (say if one side has only one legal move) [Uiterwijk 1991] and certain piece evading moves to bring a piece out of the opponent's attack (an *ad hoc* heuristic tried in *Abyss*). The latter two have not yet been adequately explored in the computer chess literature, but both have been exploited in some of the chess and Chinese Chess playing programs. All these can be categorized as knowledge extensions.

However, using static knowledge for extensions may not be enough to cover all the interesting cases and such extensions can sometimes lead you astray. A more powerful search extension heuristic called *Singular Extensions* was thus introduced and proved to be a great success† in the chess playing program *ChipTest* (predecessor of *Deep Thought* [Hsu *et al.* 1990]) [Anantharaman *et al.* 1988]. The idea of singular extensions is to use information gathered from the search itself to extend the search whenever one move is significantly better than the sibling moves. There are other heuristics that also fall in this category, e.g., the *Null Move Quiescence Search* [Palay 1983, Beal 1989]. Null moves provide a cheap means to detect threats. It is possible that by using the null move, one can control and shape the search toward those lines which are more worthy of consideration.

All the above mentioned extension heuristics have been implemented in the existing chess (Chinese Chess) playing programs and some of them have proved to be successful in practice. However, it is not clear whether the relative importance of these heuristics and the best combination have been revealed. Since the rationale for most of the heuristics is domain specific, such a conclusion can hardly be drawn on the pure basis of existing theories; therefore it can only be obtained from experiments through practice. This thesis attempts to carry out a series of experiments and determine the relative importance and best combination of these heuristics.

1.2 Chinese Chess and Computer Chinese Chess

Chinese Chess is an ancient game, yet the history of computer Chinese Chess is still short. Initial research on Computer Chinese Chess started in the early 80's [Zhang 1981, Huang 1986], however, a Chinese Chess playing program didn't appear until 1986, as reported by Tsao [1988]. Further, although the play of the best chess playing program has already achieved the grandmaster level, the best Chinese Chess playing programs are still a long way from such a goal.

† However, later experiments show that such a success can hardly be attributed to *singular extensions* alone [Anantharaman 1991].

There are several reasons for this problem. First is hardware speed. Most of today's best chess playing programs run on mainframes or exploit specially-built hardware for move generation and even tree searches. In contrast, the performance of most of today's Chinese Chess playing programs is based on microcomputers. Speed, if not dominant, is the major hindrance to improving the performance of the current Chinese Chess playing programs.

Second is the complexity of the game itself; such as the repetition rules and the greater tactical nature. Compared with chess, the repetition rule of Chinese Chess is more complicated and none of today's Chinese Chess playing programs can claim that they implement every rule correctly. Conceptually, such a problem can be solved if efficiency is not taken into account. In this thesis, we propose an approximate solution to the treatment for repetition rules. Yet we still have to face a more serious problem: writing an accurate static evaluation function. Because the king in Chinese Chess is confined to only nine squares (therefore more vulnerable than in chess) and because there is no way to block pawns, Chinese Chess is like an open chess game and so it is extremely difficult to take every aspect of the game into account precisely. Also, because there are no pawn promotions, and therefore all games have to be won by a direct attack on the opponent's king, the endgame knowledge (endgame databases) plays a more important role than in chess.

Third is enhancements to selective search extensions. It is possible that by employing the right set of selective extensions, we will shape the search tree in such a way that more time is spent on interesting lines than those less likely to be relevant. So better performance is possible when better selective extension algorithms are adopted.

To develop special hardware for Chinese Chess move generation, to build endgame databases and to write a knowledgeable static evaluation function is not a part of this thesis. Some experiments on evaluation function have already been done by Jacobs

[1989] and Tsao *et al.* [1990]. The thesis concentrates on an alternate way to make a program play stronger, enhanced selective extensions.

1.3 The Abyss Chinese Chess Program

To carry out a series of experiments on search extension heuristics, a Chinese Chess playing program *Abyss* was developed as a test-bed. *Abyss* started as a graduate course project ("Heuristic Search" by Prof. Tony Marsland) in March 1990. It adopts most of the structure from its predecessor *ParaBelle* [Marsland and Popowich 1985], a chess playing program used for testing parallel search algorithms and later revised by Breitkreutz to run under NMP [Marsland *et al.* 1991]. However, the program has been modified dramatically since then. *Abyss* participated in the Chinese Chess tournament of the 3rd Computer Olympiad in Maastricht, The Netherlands, in August 1991 and tied for the first place with one other program.

Abyss was developed using C (C++ and X11 for the X-Window interface) under the UNIX† operating system (particularly SunOS). The program is about 10,000 lines of source code and the compiled running file occupies about 150K memory space (both excluding the X-Window interface). It includes most of the search extension heuristics discussed in this thesis and the common features like opening book and time-control mechanisms for such a program (c.f. Chapter 2). The program (without the X-Window interface) has been compiled on different UNIX operating systems such as SunOS, System V, Ultrix and UMIPS, so it is also portable.

1.4 Notations, Test Data and Measurement

For convenience, throughout this thesis, the Chinese Chess board will be represented by 9 vertical lines (files) from *a* to *i* and 10 horizontal lines (ranks) from *0* to *9*. The two players of the game are given the name Red (the one to move first) and

† UNIX is a trademark of Bell Laboratories.

Black. There are altogether 16 pieces for each side. The name and the number for each piece on the initial board (in brackets) are listed below:

King (1): or General or Shui for Red and Jiang for Black;

Rook (2): or Warrior or Ju;

Knight (2): or Horse or Ma;

Cannon (2): or Gunner or Pao;

Guard (2): or Assistant or Shi;

Bishop (2): or Elephant or Xiang;

Pawn (5): or Soldier or Bin.

The first name for each piece will be used in the thesis, though other names can be seen elsewhere. To represent a move, we adopt a method similar to the algebraic notation from chess (as used in *Xian* [Jacobs 1989]), i.e., a move is represented by a *from-to* square pair. The rules of Chinese Chess can be found in a variety of books (e.g., Lau [1985] and Jacobs [1989]) and are also briefly summarized in Appendix 1 of this thesis.

To carry out our experiments, we use 50 Chinese Chess middle-game positions extracted from a standard work [Tu 1985]. Most of these positions are tactical problems and only a few of them are purely strategic. In most of the positions, the side to move has a chance to launch a tactical blow on the opponent and win material eventually. In some other cases, the side to move has to find a combination to equalize. All the 50 positions and their solutions can be found in Appendix 2 of the thesis.

There are two experimental measures that are of interest. First is the total nodes searched for a given search depth against a particular position. This is used as a basis for comparison with the efficiencies of different extension heuristics. Second is the success rate, i.e., the proportion of the correct moves found in the 50 middle-game positions.

However, there is one problem associated with the calculation of the correct move for each position. It has been noticed by Ye and Marsland [1992] that using only the move returned by the search is not sufficient; the move can be suggested for a wrong reason. Since it is too time-consuming to manually examine the correctness of every move (to see whether a correct principle variation is also provided), we decide to use an approximate method to automate this examination. For each position, we manually calculate the expected score of the position based on the lines provided in the annotation for each position. Only if the move returned by the search is the same as the one suggested, and if the score returned by the search falls into a window (defined as three pawns' value) around the expected score, do we say that the move found is correct. Throughout this thesis, the success rate is calculated based on such a definition of correct moves.

The hardware used for running the tests is a Silicon Graphics machine (IRIX Release 4.0.1 System V; named *innisfree*) in the Department of Computing Science at University of Alberta. It has 4 CPUs, each of which is roughly equivalent to that of a SPARCstation IPC. The time referred to will be CPU time, computed using the time measuring facilities provided in UNIX.

1.5 The Organization of the Thesis

The thesis opens with a brief description of the Chinese Chess playing program *Abyss*; its move generator, evaluation function and search algorithms. Some space is also devoted to the discussion of the implementation of the repetition rule in *Abyss* as well as features such as opening book and time utilizations (c.f. Chapter 2). This is followed by some experimental results for basic enhancements to the $\alpha\beta$ algorithm. These enhancements include heuristics for interior move orderings, transposition table management, null move search and futility cutoffs. The purpose of this suite of experiments, as in Chapter 3, is to give a general feeling for the framework of Chinese Chess before we go further to the more complicated selective search extension heuristics.

A discussion is then initiated in Chapter 4 about the knowledge extension heuristics like check evasion, recaptures, piece-evading moves, king-threat moves and strictly forced moves. Some of the implementation details are shown and the experimental results on these heuristics are presented and compared. A method to compare the relative importance of these extension heuristics is introduced and the test results are presented.

The discussion is carried over on two other extension heuristics which use information gathered from the search to decide whether to search deeper. They are singular extensions and null move quiescence search. The basic algorithms and the experimental results are presented in Chapter 5, followed by an analysis based on the selected test results. Then a series of experiments is carried out to decide their relative performance when combining all the heuristics mentioned in the thesis. The same method for such a comparison as introduced in Chapter 4 will again be adopted.

The thesis closes with our conclusions on selective search extensions in the domain of Chinese Chess and some potential areas for further work.

Chapter 2

The Abyss Chinese Chess Playing Program

2.1 Introduction

In this chapter, we give an overview of the *Abyss* Chinese Chess playing program; its move generator, evaluation function, search framework and other miscellaneous features like opening book and time utilization. From the algorithmic point of view, since Chinese Chess differs most from chess in its repetition rules, some discussion is also devoted to that matter.

We start by giving a description of the Chinese Chess playing program *Abyss*. As we have mentioned earlier, *Abyss* inherits most of the data structure from its predecessor *ParaBelle* [Marsland and Popowich 1985] but has been changed to suit the special needs for a Chinese Chess playing program. The major difference between *Abyss* and *ParaBelle* lies in the evaluation function and the search algorithms. Other modifications to *Abyss* are necessary (for example, the move generator), but remain the same in structure as *ParaBelle*. To prepare for the Chinese Chess tournament of the 3rd Computer Olympiad, two new features (opening book and time control) were added to meet with the tournament requirement.

2.2 The Move Generator†

The move generator is responsible for providing a set of *pseudo-legal moves* [Hartmann 1989] available in a given board configuration for a particular side to move. The same method as *ParaBelle* is used for the move generation in *Abyss*, i.e., *Shannon's Method* [Frey 1983]. There are still other methods (excluding hardware move generators) like *Array Generator* [Zielinski 1976], *Bitmap Method* [Frey 1983, Cracraft 1984]

† This section is based on the CMPUT 507 ("Heuristic Search" by Prof. Tony Marsland) course project paper by Chun Ye, May-Yew Wee and Steve Dai.

and *Table Method* [Bell 1984]. To speedup the development of the program, a few changes have been made to the existing move generator of *ParaBelle*, though it is possible that other methods (such as the table method) may be more efficient in the domain of Chinese Chess.

We use 90 computer words to define the board with each word representing one square of the board. The squares are numbered from 0 to 89 starting from the upper-left corner. Each piece is given a number (+1 for red pawn, -1 for black pawn, ..., -7 for black king) and these numbers will be stored in the word representing the square on which the piece resides. An empty square is represented by 0.

To detect the edges of the board, a 16-bit (12 bits are actually used) direction mask is assigned to each square. The mask is organized into 4 groups with 3 bits each, starting from the least significant bit. The 4 groups represent the directions *Up*, *Down*, *Left* and *Right*. Before generating a move, the direction of the move is *ANDed* with the mask of the current square. If the result is 0, then the move is valid. A similar mask is required to detect the edges of the palace.

Legal moves from any position can be determined by simply noting the mathematical relationship among the squares. For pieces with *fixed moves* (i.e., king, knight, bishop, guard and pawn), the move generation can be done by adding the offsets of the destination squares to the current square. A move is pseudo-legal if there is no friendly piece on the destination square and the direction mask is 0. The knight and bishop would require extra checks to see if they are blocked. Also, it is necessary to see if a pawn has passed the river (the center two ranks in the Chinese Chess board) so that sideways moves can be generated.

For pieces with *sliding moves* (i.e., rook and cannon), the moves for each direction are generated one square at a time starting from the closest square. At each square, a move is generated when the direction mask is 0 and a friendly piece is not occupying that square. If the square is empty, the same process goes on to the next square. For cannon however, after encountering an occupied square (by any side), the sliding process has to be carried on until an enemy piece is reached for capture moves.

A pseudo-legal move doesn't guarantee that it is legal. It could expose the king to the enemy's check, and in Chinese Chess, it could oppose its own king to the enemy's king with no pieces intervening (which is illegal in Chinese Chess). However, this detection of legality is delayed until a move is actually made on the board during the search. Such a treatment will be more efficient since not all the pseudo moves generated will be actually examined during the search.

2.3 The Evaluation Function

Since the quality of the evaluation function directly affects the performance of the program, the evaluation function is a major component in a (Chinese) chess playing program. However, compared with the move generator, the evaluation function can be highly empirical, although some method for automatic tuning is possible [Marsland 1985, Hartmann 1989, Tunstall-Pedoe 1991]. Because we lack a large database of Chinese Chess games for such an optimization, and we are more interested in the search extension heuristics in this thesis, the values for the different notions that are used in *Abyss's* evaluation function were hand-picked based on the existing work and the author's Chinese Chess knowledge.

In this section, we discuss the notions that are used in *Abyss's* evaluation function. Apart from the description for the static positional evaluation, we also discuss some improvements to *Abyss's* capture search from *ParaBelle's*.

2.3.1 Positional Evaluation

The game phase plays an important role in deciding the relative importance of some components of the evaluation function, so in *Abyss* we separate the game into four stages as *opening game*, *early middle game*, *middle game* and *end game* before initiating a search†. The game stage is kept in a special variable and is used to decide whether some notions of the evaluation should be included and what the weight of a particular notion should be.

The following Chinese Chess knowledge has been included in the current version of *Abyss*:

Material Balance

The material balance is the difference between total piece values for the two sides on the board. Two special evaluations are considered, for pawns beyond the *river* and the increment/decrement of the values for knights/cannons. Since pawns that pass the river gain the extra sideways moves, the value of a passed (the river) pawn is doubled in *Abyss*. Cannons need supporters to jump over for capture moves and the values for cannons decrease as the game goes on since fewer pieces will be left on the board to serve as such supporters. Contrary to cannons, knights can be blocked but have more mobility in the endgame when fewer pieces are left on the board. In *Abyss*, the values for cannons and knights are decreased and increased respecting game phase, until by the endgame, the values for the two pieces are reversed. Table 2.1 gives the ranking for each piece and their values as used in *Abyss's* evaluation function.

Table 2.1: Initial Piece Values Used in Abyss

Pieces	King	Rook	Cannon	Knight	Bishop	Guard	Pawn
Values	7000	1800	900	800	300	300	100

Board Control

Abyss doesn't have a mobility term in its evaluation function. Instead, an approximate method is used for the board control. A piece placement table is predefined for

† It may be better to have more game phases as well as to allow the game phase to be changed during the search for a more accurate evaluation.

each type of piece and the board control is the difference between the total piece placement values for two sides. *Abyss* inherits its same piece placement tables from those presented by Tsao *et al.* [1990], but with some modifications: the placement for guard and bishop are also included, and the values for the cannon are updated in the endgame to bring the cannon back near its own king (where the guards and bishops can be used as supporters to attack the opponent's king).

King Safety

Two approximate terms are considered in *Abyss's* evaluation of king safety. First, we introduced the term of head-cannon and side-cannon which refers to such a pattern where a cannon is on the same line (file or rank) as the opponent's king and there is no piece in between. Such a setup is advantageous for the cannon side during the early stages of the game and occurs frequently in Chinese Chess†. To calculate accurately such a term, we need not only to assign a bonus for the head-cannon or side-cannon side, but also to give a penalty when the king moves out of such a situation, since otherwise the king move will be considered as an easy defense to building a head-cannon or side-cannon position. The other term for king safety is the number of squares near the king attacked by opponent's pieces. *Abyss* adopts an approximate way in which this attack term is calculated incrementally when a piece moves.

Special Scoring for Openings

Some special scorings are considered in *Abyss's* evaluation function during the opening stage. The term *tempo* is defined as the number of pieces that have left their original squares and is computed for each side in the evaluation function. *Abyss* also gives a penalty to a side who moves the same piece consecutively as well as for

† To justify such a statement, please notice that the side-cannon position appeared in both of the games between *Abyss* and *Surprise* during the 3rd Computer Olympiad.

capturing a pawn by a cannon during the opening stage if no check results from such a capture.

Compared to what has been done in the evaluation function for some existing chess and Chinese Chess programs, the evaluation in *Abyss* is perhaps crude and less knowledgeable. However, knowledge and search have always come to a compromise in the design of a chess (Chinese Chess) program. Here, we believe the mechanisms used in *Abyss's* search region should somehow compensate for the lack of knowledge in its evaluation.

2.3.2 Capture Search

Capture search [Gillogly 1978, Bettadapur and Marsland 1988] serves as a means to reduce the errors in the static position evaluation. *Abyss's* capture search has been improved from its predecessor *ParaBelle* in the following ways: first, there is no depth limit for the capture search in *Abyss* since the savings are marginal when the depth for capture search goes beyond 8 [Bettadapur and Marsland 1988]; second, the static value of the board is used to raise the lower bound α to yield more possible savings [Schrüfer 1989]; and third, the search only extends if the new material balance after a capture is better than the current best score [Slate and Atkin 1977, Schrüfer 1989].

2.4 Search Algorithms

Abyss uses minimax enhancement like $\alpha\beta$ *pruning* [Knuth and Moore 1975], *iterative deepening* [Gillogly 1978, Slate and Atkin 1977], *minimal window search* [Fishburn 1984, Pearl 1980, Marsland and Campbell 1982, Reinefeld 1983] and *aspiration search* [Baudet 1978, Finkel *et al.* 1980] for its search algorithm; as well as heuristics like *transposition table* [Greenblatt *et al.* 1967, Slate 1977], *refutation table* [Akl and Newborn 1977, Marsland 1983], *history heuristic* [Schaeffer 1986, Schaeffer 1989] for its interior move orderings. In addition, *Abyss* also tries other two heuristics that involve some risk, *null move heuristic* [Goetsch and Campbell 1990] and *futility cutoffs* [Schaeffer 1986], for better search efficiency. All these heuristics are aimed to reduce the search effort with little or no chance of deteriorating the performance. We shall see the results of some experiments on these basic minimax search enhancements in Chapter 3.

There are other heuristics used to enhance the performance. These are mainly selective search extensions like *check evasions*, *recaptures*, *king-threats*, *piece-evading moves*, *strictly forced moves*, *singular extensions* and *null move quiescence search* (also named as *second order quiescence search*). These selective extensions provide a way to control the shape of the search tree towards those lines that need more consideration and therefore make the time spent on search more worthwhile. Please refer to Chapter 4 and Chapter 5 for a more detailed discussion on these heuristics.

2.5 The Impact of the Chinese Chess Repetition Rule

Although similar to chess, Chinese Chess differs significantly in its repetition rule. For example, repetition check is considered a draw in chess, but such a repetition is not allowed in Chinese Chess. In general, the rules of Chinese Chess disallow the use of certain repetitions after a threat move (even so there are exceptions). Three types of moves are considered as threats; checking moves, moves that threaten to win material, moves that threaten to mate. However, the rule allows certain repetitions via a threat, provided the current position is a repetition and is reached by a threat move as well (again there are exceptions to this).

Since the repetition rule of Chinese Chess is so complicated, none of the current Chinese Chess programs can claim that they handle all situations correctly. Tsao *et al.* [1990] proposed a means for their program *Chess Master* to handle most of the commonly occurring situations. The commercial Chinese Chess program *Xian* [Jacobs 1989] guarantees never to make an illegal repetitive move, but still lacks the knowledge to handle cases when a repetition would be legal and in some cases it allows the opponent to make an illegal repetitive move. Another program, *Surprise* [Wu 1991], a participant at the 3rd Computer Olympiad, allows certain illegal repetitions when it finds that all alternatives are significantly worse (as evidenced by some pre-tournament testings and casual plays against *Surprise* during the 3rd Computer Olympiad).

2.5.1 The Repetition Check Algorithm in Abyss

In *Abyss*, we tried a more general repetition detection algorithm, differentiating between detection in the root and during interior nodes. The scenario behind this is to use a more strict rule for the root node but be generous to internal nodes.

For internal nodes, some approximation is made and only check repetition and some simple piece-winning threats are considered. Two positions are considered identical if their transposition table *locks* [Zobrist 1970, Marsland 1987] are the same. A stack (sequential table) is used to store all such *locks* from the first move in each game, but no count of the number of repetitions is kept. If a repetition under such a definition is detected, we determine not only whether the move reaching this position is a check, or a threat to win a lone piece, but also that the previous move is not such a simple threat. If the preceding move was not a threat, then a threatening move leading to a repeated position is assigned an *illegal* score (almost as poor as a *mate* score). Any other combination of moves to a repeated position is given a *draw* score and in both cases there is no further search. The reason for using an *illegal* score is because the definition of repetition here is *approximated*, and so we might miss the only possible defense if an *illegal* score is no better than a *mate* score.

Nevertheless, at the root node an illegal move will be disallowed, so a more strict repetition check algorithm is adopted. For each move being considered at the root, we check backwards to see if this position is being repeated for the third time. If so, and the move that reaches this position falls in the category of *threat* (as defined in Section 2.5.2), we backup to see if the previous position is also a repetition (not necessarily for the third time) and whether the move reaching that position is also a *threat*. If the test of the second position fails (the position is not repeated or the move to it is not a threat), we assign the value of the current move as *forbidden*, a score worse than *mate*. In all other cases when a threefold repetition is detected, a *draw* score is assigned. We think such an approach can correctly handle many of the difficult repetition conditions in Chinese Chess. Also, it provides a conceptually sound base for further improvements and should be able to manage even more repetitions when the definition of threat is refined.

2.5.2 The Definition of Threat

In *Abyss* a *threat* is defined as a:

Checking threat

If a move delivers check, it is a threat. This is the simplest case.

Mate threat

If after one side has made a move, the opponent can be mated by a series of checking moves, the first move is considered to be a *mate threat*. In *Abyss*, an approximation is adopted. We do a search to a depth of 3, considering only checking moves and replies to check, and assume that there is a mate if a mate score is returned for this search. To reduce the search cost to a reasonable level, detection of a mate-in-n ($n > 3$) threat is temporarily not considered, although according to rules, all mate threats should be equally treated. For faster hardware however, it may be better to search to unlimited depth, terminating only when a repetition (of any kind) or a mate is seen or when there are no more checking moves.

Piece-winning threat

Again some simplifications are made, and we only consider those moves that threaten to win an unprotected piece (pawns that pass the river and all minor and major pieces in Chinese Chess). The expense of *threat* detection isn't as large as it seems, because the operations above are only carried out when a third-time-repetition is seen, and the test is only done once during the search.

Figure 2.1 summarizes our treatment on how to distinguish a legal repetition from an illegal one. Further work is required to consider more backward positions when a repetition is found. At the moment we can handle some difficult repetition situations, e.g., two threats over two threats (a draw) or two threats over one threat (a loss). Also, because of time limits, search to some predefined fixed depth might be required to detect whether a side has a *piece-winning threat*. By restricting the moves to only captures, checking moves and replies to checks (an extended quiescence search), we can do a search after making a null move (in this case making two consecutive moves for the side causing the repetition). If the value returned exceeds a certain amount (the *threat margin*), the first move can be thought of as a *threat* and can be treated accordingly.

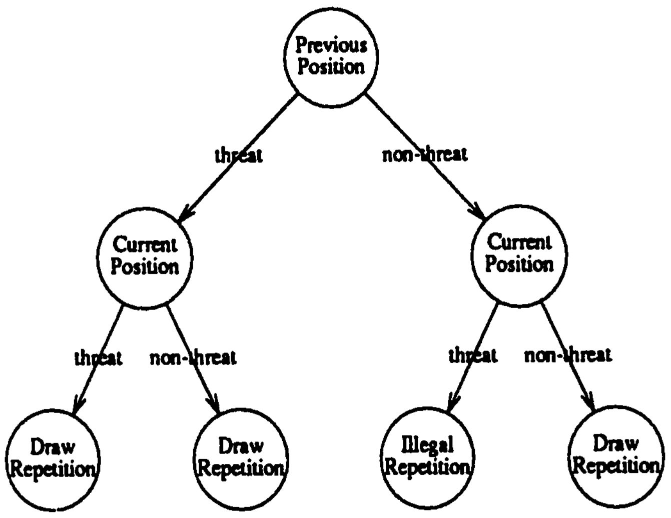


Figure 2.1: Legal and Illegal Repetitions

2.6 Other Miscellaneous Features

In this section, we give some descriptions of two other features which are implemented in *Abyss*: time control and opening book. These functions were newly added to *Abyss* to suit the needs of the 3rd Computer Olympiad.

2.6.1 Opening Book

There are many reasons for using an opening book for a computer chess playing program. First, the moves are made almost instantly, thus saving time for considering subsequent moves. Second, no mistakes will be made by the program on book-opening moves provided their correctness have been checked carefully. Because of these, we seldom see a chess playing program without an opening book.

Many different strategies can be used to store the opening moves in the computer memory, but normally, moves are stored in two ways: either as complete chess positions with best moves attached or as move strings [White 1990]. Although the former method has the advantage of allowing programs to stay in the book after transpositions, it has the drawbacks of using large memory space as well as the need for auxiliary programs to encode the positions and moves [Levy 1985, White 1990]. This is perhaps why the latter method is preferred, especially for microcomputer programmers.

The opening book of *Abyss* follows the same method called the *amateurs' opening book* as described by White [1990] which is based on storing move strings. This method was adopted because of the need to modify our opening book frequently to test what opening variations best suit our program's style, and the limit of time available before the 3rd Computer Olympiad. To make it work, the opening moves are first stored in the memory in the from-to notation (see Table 2.2). Notice one opening variation is split over several lines for improved readability. The "#" character is added for comments and delimiting different opening lines (as based on observations of the opening book for *Gnu Chess*†).

The above mentioned file for move strings is then compressed using the method described by White [1990]. During the opening, the line in the book is read in and used

† *Gnu Chess* is a public chess playing program distributed by Free Software Foundations, Inc.

Table 2.2: Sample Opening Book Lines

```
# Center Cannon vs. Screen Knights
h2e2 h9g7
h0g2 i9h9
i0h0 b9c7
h0h6 h7i7
#
h2e2 h9g7
h0g2 i9h9
g3g4 h7i7
#
h2e2 h9g7
h0g2 i9h9
i0h0 g6g5
h0h6 b9c7
```

to match the actual move sequence. If this move sequence is in the opening book line, the leftmost move in the line that is not matched with the actual moves will be selected as the response. Since the board of Chinese Chess is symmetric, special effort is made to exploit this fact. We view one opening book as two with moves in one book representing reflective moves in the other. Therefore, if a move sequence is not found in the book first, we would then reflect all the moves in the book and search one more time. Only if neither search finds the move sequence, do we call the search routine and "think" on our own.

2.6.2 Time Utilizations

The other major improvement added to *Abyss* is a feature which lets the program utilize time more efficiently. Searching to a fixed depth during the tournament is either dangerous if the depth is set too great, resulting in a time-forfeit before reaching the required time-limit, or it can be conservative if the depth is too shallow, leaving much idle time. One way to alleviate this is to choose different depths for different game phases and the depth increases as the game goes on. But, a far better way is to allocate a certain amount of time on each move.

Suppose we have to make *n* moves in a certain time *C* (usually C is 2 hours or 7200 seconds and n is 40 from the beginning of a game). Experience has told us to use the following formula to calculate the amount of time *t* (seconds) to spend on each move:

$t = (C - O) / n.$

Here *O* (usually 15 minutes or 900 seconds) is the operation time which is predefined to reflect the time spent on making the moves on the board and hitting the clock. As the game goes on, the values of *n*, *C* and *O* should all be modified to reflect the remaining status and the new time to spend on a move be obtained.

Abyss uses a slightly different formula to calculate the time *t*. We use a new parameter *R* which represents the actual time remaining on the clock. So we can use the following simpler formula to decide *t*:

$t = R / n.$

Whenever *Abyss* makes a move, it will prompt the operator to make the move on the board, hit the clock and then hit the return key indicating the completion of a move. The interval from starting the search to hitting the return key will be subtracted from R. Also at any time when it is the opponent's turn to move, the operator is allowed to reset R according to the time remaining on the clock. We feel that in this way both the possible wasted time and the chances of over-stepping the time control will be insignificant.

However, no matter how you obtain your time to spend on each move, you always come up with the problem of how to treat the current move and score when the time is up. Do you accept the current best move or do you want to search a bit deeper hoping to find some even better moves? Some innovation on time utilization has already been introduced by Hyatt [1984] for the chess playing program *Cray Blitz* [Hyatt *et al.* 1990]. In *Abyss*, the solution is to accept the move as long as the score of the best move in the current iteration is no less than the score of the best move from the previous iteration minus some tolerance (set to 1/10th of a pawn value initially and increases as the game goes on), or when the program is in time trouble (defined as whether it has entered the last 5 minutes in the time control). However, if such a condition is not met, we search a maximum of 10 more moves in the hope that a better move can be found. Here, a second alarm has to be set (amount usually equal to which has been spent already on the current move; or less when we are close to the time control) to avoid the possibility that the program may step out of the time control†.

There is also another way that *Abyss* tries to utilize the time more efficiently; it will keep the program busy by "thinking" when it is the opponent's turn to move. Generally speaking, two strategies may be used to implement such a feature. One is to simply think for the opponent and fill the results in the transposition tables, storing the information useful for the next search. The other is to think about an assumed move to be made by the opponent (found in the principal variation refutation table). The second way is better and is adopted because the chances that the opponent makes our guessed move is high and we can search deeper in this case.

All the time utilization functions have been implemented in *Abyss* using the signal facilities (`signal`, `alarm`, `setjmp` and `longjmp`) and the file control facility (`fcntl` to open the keyboard in an asynchronous mode) provided in UNIX.

† Because of our failure to do this, we lost the second game to *Surprise* during the Chinese Chess tournament at the 3rd Computer Olympiad.

Chapter 3

Minimax Search in Chinese Chess

3.1 Introduction

Here, we present experimental results on four different essential minimax enhancements; they are: heuristics for interior move orderings, transposition table management, null move search and futility cutoffs. The purpose of this suite of experiments is to give some basic feelings for the efficiencies of these minimax enhancements in the domain of Chinese Chess.

3.2 Heuristics for Interior Move Ordering

The following heuristics are used for interior move orderings in *Abyss* (the relative importance is given in that order); they are the transposition tables (`+tra`), the refutation tables (`+ref`), the captures move ordering (largest material gain first; `+cap`) and the history heuristic (`+his`). The experiments to test the relative efficiencies of these heuristics and their results are presented in Table 3.1. For these experiments, the transposition table consists of 32K entries and the size for the history table is 8100 entries (the Chinese Chess board is 9X10). In the table, *nodes* stands for the total nodes (interior nodes plus capture nodes) searched for a particular experiment (ordering) over 50 test positions and *ratio* is the node ratio for one experiment over the original experiment (with captures move ordering only). Using the definition of correct moves and correct move scores in Section 1.4, the move quality for these experiments can be represented by both *success rate* (percentage of correct moves found with correct score) and *correct move hits* (percentage of correct moves found no matter what scores are). A more general term *scores* is introduced to reflect this move quality and it is presented in the following form:

score = % success rate (% correct move hits);

with a score reflecting the predicted outcome. For later experiments throughout the thesis, the same terminology will be used wherever applicable.

Table 3.1: Experiments on Interior Move Orderings

Orderings	Depth = 3		Depth = 4		Depth = 5	
	nodes	ratio	nodes	ratio	nodes	ratio
+cap	263085	1.00	1978554	1.00	8883580	1.00
+cap+ref	236363	0.90	1867159	0.94	7925428	0.89
+cap+his	182961	0.70	1025896	0.52	6467181	0.73
+cap+tra	209638	0.80	1403093	0.71	5452502	0.61
+cap+ref+his	167348	0.64	953147	0.48	5872280	0.66
+cap+tra+ref	207375	0.79	1398513	0.71	5429484	0.61
+cap+tra+his	158359	0.60	788646	0.40	4238259	0.48
+cap+tra+ref+his	157352	0.60	787382	0.40	4153931	0.47
scores	6(18)		16(30)		22(34)	

From the results in Table 3.1, we see the *scores* improve as the search depth increases. But for each depth, some correct moves is found for a "wrong" reason, i.e., the value of the move (c.f. Section 1.4) doesn't fall within the window of the expected position score. Obviously, all these positions are beyond the search horizon and the correct moves are returned only on strategic concerns. Regarding search efficiencies, we see that *history heuristic*, in terms of memory occupancy (about 16K bytes), performs well at shallow search depth but is outperformed by *transposition tables* at depth 5 (compare experiment `+cap+his` with `+cap+ref`, `+cap+tra` and even `+cap+tra+ref`). This is also true for experiment `+cap+tra` and `+cap+ref+his` where the former is less efficient at shallow depth but becomes better when the search depth is deep (Depth = 5). One reason that makes the transposition table more efficient only at deep search depth can be explained as lacking enough transpositions at shallow depth, since not like history heuristic, which exploits information across the whole game tree, information stored in transposition tables can only be used by those positions seen previously. Another reason is that transposition tables, apart from their function in move ordering, also provide a means for some forward prunings to the encountered positions. Therefore, although transposition tables can result in great total search efficiency, in the perspective of memory occupancy and move orderings alone, history heuristic proves to be the best. Combining all these heuristics gives even better search efficiency and a total of 53% node count saving (defined as 1 − *ratio*) is achieved at depth 5 (`+cap+tra+ref+his`). It can be seen that all the interior move ordering heuristics that perform well for chess also perform well here for Chinese Chess. Also it can be obtained from Table 3.1 that the average branching factor† (as compared with *nodes* at different search depths) is around 5 which again shows the efficiencies of these interior move ordering heuristics and the search algorithms implemented in *Abyss*.

3.3 Transposition Table Management

A *transposition table* [Greenblatt *et al.* 1967, Slate and Atkin 1977] is a large hash table used to store the useful information during the search. The information usually represents the nodes (positions) during the search like its associated score, move and depth. Since a transposition is frequent in Chinese Chess as in chess, such information can help reduce the search effort if the position has already been encountered. Also, the transposition table serves as a means for move reordering, especially when combining other minimax search enhancement like iterative deepening.

In *Abyss*, the transposition table is implemented using the method as described by Marsland and Campbell [1982] with a 32-bit hash key†† and a depth-based replacement algorithm to handle collisions. Further improvements are possible, e.g., dividing the nodes in the tree into privileged nodes, which are replaced only by deeper results, and non-privileged nodes, which are replaced in simple FIFO order [Ebeling 1987]. The random numbers used to calculate the *lock* [Zobrist 1970] are obtained with the UNIX

† A better way to calculate the branching factor is introduced in Chapter 5 when discussing the null move quiescence search.

†† It has been proved that a 32-bit hash key is inadequate for larger trees (depth equals to 6 or greater) searched by today's faster processors [Warnock and Wendroff 1988]. In our experiments however, no such problems were experienced, because the trees searched here are relatively smaller (a maximum depth of 5 is reached).

pseudo random number generator `random()`. Notice for these random numbers, a good distribution of 1's and 0's is essential to the performance of a transposition table. The experimental results show that the random numbers generated here meet with this property.

For Chinese Chess, it is also possible to exploit the board symmetry when calculating locks (two symmetrical positions should be treated as identical), but the chance of such symmetries is too rare to occur in the whole game (except in the opening stage since the initial board configuration is symmetrical) to make it worth being included in the transposition table.

Here we give some experimental results for using the transposition table during the search. All the positions are search to a fixed depth 5 with different transposition table sizes. The experimental results are presented in Table 3.2.

Table 3.2: Transposition Tables with Different Sizes

size	nodes	time	hits	t-stores	overs	%	est. overs	est. %
2K	5090307	3613	61503	415542	293355	70.6	317454	76.4
4K	4711910	3272	82502	390690	201001	51.4	224765	57.5
8K	4446413	3065	100105	374505	123489	33.0	139835	37.3
16K	4235483	2880	113334	357099	65834	18.4	75314	21.1
32K	4153931	2808	121376	349564	33891	9.7	39306	11.2
64K	4116806	2800	126126	346580	17238	5.0	20188	5.8
128K	4092139	2822	128645	344146	8700	2.5	10166	3.0
256K	4047571	2769	130203	343038	4465	1.3	5108	1.5
512K	4041773	2708	130734	342314	2255	0.7	2558	0.7
1024K	4036961	2702	131008	342111	1160	0.3	1281	0.4

In Table 3.2, the *size* for the transposition tables is represented in thousands of entries and *time* is the total CPU time (in seconds) spent on 50 test positions. The *hits* is the count of positions that are found in the transposition table when retrieving. The actual overwrites (*overs*) is the count of updating an already occupied transposition table entry; and the number of estimated overwrites (*est. overs*) for each search problem is calculated using the following formula (as given by Feldmann *et al.* [1992]):

overs = stores – size * (1 – ((size – 1) / size) ^ stores)

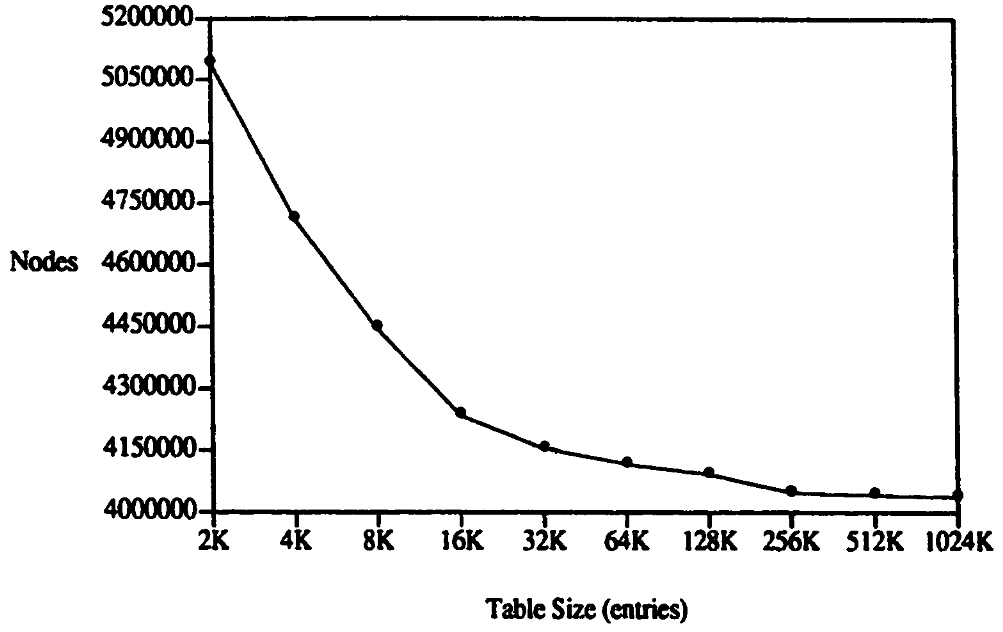


Figure 3.1: Total Nodes Searched versus Transposition Table Size

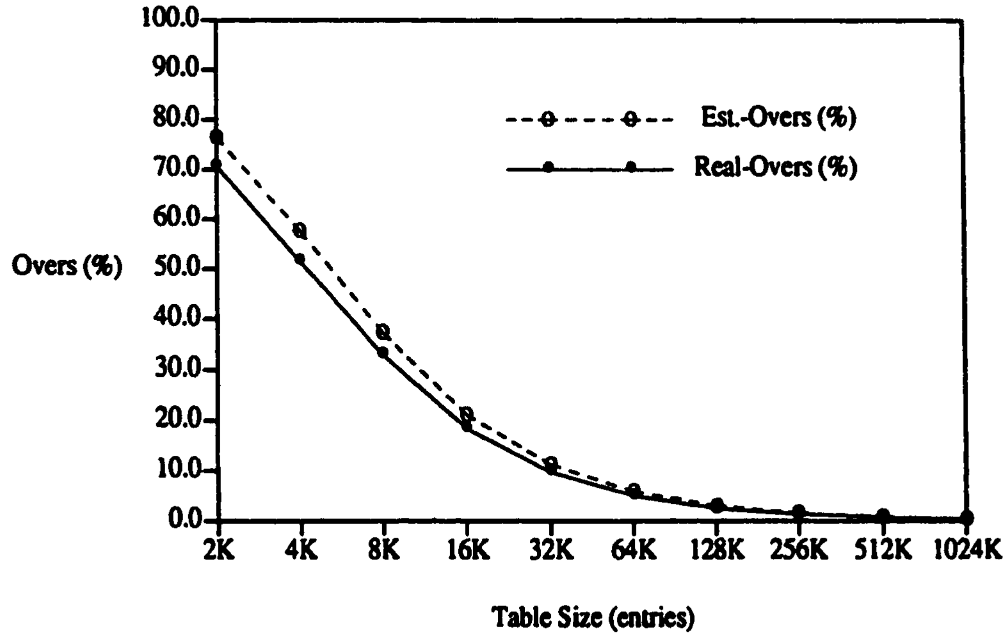


Figure 3.2: Overwrite Percentages versus Transposition Table Size

Here *stores* is a frequency count of the writings of the transposition table happened during the search on each particular problem. The total stores (*t-stores*) is the sum of all *stores* over 50 test positions. The percentage of actual overwrites (%) and percentage of estimate overwrites (*est.* %) are relative to the total stores (*t-stores*) which occurred during the search.

There are two results we are interested in: the total number of nodes searched and the overwrites of the transposition table during the search, and they are presented in Figure 3.1 and Figure 3.2 respectively. In both figures, *Nodes* stands for the total number of nodes expanded for each transposition table size (in entries); "Real-Overs (%)" and "Est.-Overs (%)" are percentages of actual and estimated overwrites relative to the total stores occurring during the search for the 50 test positions. The results provided in Figure 3.1 and Figure 3.2 show that the transposition tables perform equally well for Chinese Chess as for the optimal implementation of this heuristic in *Abyss*, although a search to depth 5 may seem inadequate to fully expose the relative efficiency of a transposition table.

3.4 The Null Move Heuristic

The *null move* heuristic [Goetsch and Campbell 1990] is a means of improving search speed with little risk. *Abyss* tries a null move search in the internal nodes with a depth reduction of 1 ply before it starts searching legal moves. If the value returned exceeds the β bound, the value is accepted as a true cutoff; otherwise, the value is used to improve the α bound.

It is straightforward to add the null move heuristic in the normal $\alpha\beta$ search algorithm and a C-like pseudo code is illustrated in Figure 3.3. Here, function *repetition()* is used to decide whether a position has been seen before and returns a value to indicate whether a repetition is legal (a positive score) or illegal (a negative score). Function *check(side, position)* determines whether the king of *side* in the *position* is in check. Notice function *generate()* here generates a list of legal moves instead of pseudo legal moves as described in Section 2.2 and the details for checking this legality as well as the treatment for leaf node (where no legal move exists) are omitted.

For the experiments, the interior move ordering heuristics tested in Section 3.2 are all enabled. The results of adding the null move heuristic are given in Table 3.3. Two

Table 3.3: Experimental Results for Null Move Heuristics

Experiments	Depth = 3			Depth = 4			Depth = 5		
	nodes	ratio	scores	nodes	ratio	scores	nodes	ratio	scores
-null	157352	1.00	6(18)	787382	1.00	16(30)	4153931	1.00	22(34)
+null_all	137854	0.88	6(18)	751018	0.95	16(30)	3042999	0.73	18(30)
+null_front	133220	0.85	6(18)	706844	0.90	16(30)	2563303	0.62	18(30)

```
int pvs(side, position, α, β, depth)
  int side, position[90], α, β, depth;
{
  int successor[90], merit, rept, num_moves, p, score, lower;
  struct { int from, to; } moves[MAX_MOVES];

  if(rept = repetition(side, position))      /* position duplicated? */
    return((rept > 0)? DRAW: ILLEGAL);
  else if(depth <= 0)                                /* horizon node? */
    return(evaluate(side, position, α, β));          /* capture search */

  merit = -∞;
  if(depth > 1 && !endgame(position) && !check(side, position)) {
    /* null move search */
    merit = -pvs(!side, position, -β, -α, depth-2);
    if(merit >= β) {
      score = merit;
      goto out;
    }
  }
  lower = max(α, merit);

  /* normal αβ search */
  num_moves = generate(side, position, moves);
               /* details of special case (num_moves == 0) omitted */
  successor = make(moves[0], position);                  /* first move */
  score = -pvs(!side, successor, -β, -lower, depth-1);

  for(p = 1; p < num_moves; p++) {
    if(score >= β)
      goto out;
    successor = make(moves[p], position);
    lower = max(lower, score);                  /* fail-soft condition */
    merit = -pvs(!side, successor, -lower-1, -lower, depth-1);
    if(merit > score)
      if(merit > α && merit < β && depth > 1)
        score = -pvs(!side, successor, -β, -merit, depth-1);
      else
        score = merit;
  }

  out:
  return(score);
}
```

Figure 3.3: Principal Variation Search with Null Move Heuristic

different experiments were carried out to determine whether to apply the null move at all interior nodes (`+null_all`), or only before frontier nodes (a layer before horizon nodes [Marsland 1992]; `+null_front`). The terms for *nodes*, *ratio* and *scores* [*success rate (correct move hits)*] are as defined in Section 3.2.

As we can see in Table 3.3, the heuristic of using the null move except frontier nodes (`null_front`) performs better than the heuristic that applies the null move at all interior nodes (`null_all`). This difference comes from search at frontier nodes where search after making a null move and search after making a legal move both finally lead to a capture search, and therefore a null move search (with depth reduction) can hardly be more efficient since it doesn't give as much savings as it does before frontier nodes. The average savings (the arithmetic sum of savings, defined as 1 – *ratio*, over all depths) are 21% (`+null_front`) and 15% (`+null_all`) respectively. Also, both implementations yielded a same result on success rate, and a drop of 4 percentage points (the difference of success rate between the experiment with the null move search and the one without the null move search) occurred at depth 5. One reason for this deterioration could be the shallow depth for the null move search. Also it may be relevant to our particular implementation here where null move search is applied recursively (Figure 3.3), resulting a higher chance of inaccuracies when more null move searches are possible for deeper total search depth. Notice for *Hitech*, null move search is not used recursively and, as well, the searches below depth 5 are done by hardware, which guarantees at least a 5-ply search after making a null move. Even with that help, a small error was still observed [Goetsch and Campbell 1990].

3.5 Futility Cutoffs

The idea of futility cutoffs isn't new. Similar ideas have already appeared in the computer chess literature, e.g., the *Gamma Algorithm* by Newborn [1975, pp. 177-178], the *razoring* technique by Birmingham and Kent [1977] and some special forward pruning methods used in *Chess 4.5* [Slate and Atkin 1977]. All these heuristics are generally applicable but using them involves some risk. A safer variation is the heuristic which Jonathan Schaeffer calls the *futility cutoffs* [Schaeffer 1986, pp. 33-34]. The futility cutoffs differs from the above heuristics in the following: first, use of a futility-cutoff is restricted to the layer before the frontier nodes in the search tree. Second, material merit is used to decide whether to stop the search or not, but here the total value of the material merit and the maximum positional value is used. Third, the search doesn't stop when such a criterion is met, instead, it uses this information to forward prune most of the moves and only considers those which bring the material merit into the current window; this consists of all checking moves and some of the captures. In other words, the futility cutoffs is a low risk transformation of nodes near the frontier into tip nodes when certain criteria are met. Although this heuristic is often mentioned, only Schaeffer [1986] provides some quantitative data to show its effectiveness.

To give an illustration of using the futility cutoffs in the αβ search algorithm, we show our implementation of this heuristic in the C-like pseudo code in Figure 3.4. Function *value(side, position)* returns the static value (material balance) of *position* plus a maximum positional score for *side* which is used to determine whether to apply the futility cutoffs or not. The experimental result on using the futility cutoffs heuristic (`+fcut`) can be found in Table 3.4. The result of using both the futility cutoffs and the null move heuristics (`+fcut+null`) is also included.

Table 3.4: Experiments on Futility Cutoffs

Experiments	Depth = 3			Depth = 4			Depth = 5		
	nodes	ratio	scores	nodes	ratio	scores	node	ratios	scores
–null–fcut	157352	1.00	6(18)	787382	1.00	16(30)	4153931	1.00	22(34)
+null–fcut	133220	0.85	6(18)	706844	0.90	16(30)	2563303	0.62	18(30)
–null+fcut	115169	0.73	6(18)	706245	0.90	16(30)	2392147	0.58	20(34)
+null+fcut	104234	0.66	6(18)	641469	0.81	16(30)	1887356	0.45	18(30)

It can be seen from Table 3.4 that the heuristic of futility cutoffs does slightly better here than the null move heuristic since the average savings across depth 3, 4 and 5 for the

```
int pvs(side, position, α, β, depth)
  int side, position[90], α, β, depth;
{
  int successor[90], merit, rept, num_moves, p, score, lower, fcut;
  struct { int from, to; } moves[MAX_MOVES];

  if(rept = repetition(side, position))
    return((rept > 0)? DRAW: ILLEGAL);
  else if(depth <= 0)
    return(evaluate(side, position, α, β));

  /* decide whether we should apply futility cutoffs;
   * value() is material balance plus maximum positional score.
   */
  fcut = (depth == 1) && (value(side, position) <= α);

  num_moves = generate(side, position, moves);
  score = -∞; p = 0;

  while(p < num_moves && score == -∞) {
    successor = make(moves[p], position);
    if(!fcut || value(side, successor) > α || check(!side, successor))
      score = -pvs(!side, successor, -β, -α, depth-1);
    p++;
  }

  while(p < num_moves) {
    if(score >= β)
      goto out;
    successor = make(moves[p], position);
    if(!fcut || value(side, successor) > α || check(!side, successor))
    {
      lower = max(α, score);
      merit = -pvs(!side, successor, -lower-1, -lower, depth-1);
      if(merit > score)
        if(merit > α && merit < β && depth > 1)
          score = -pvs(!side, successor, -β, -merit, depth-1);
        else
          score = merit;
    }
    p++;
  }

  out:
  return(score);
}
```

Figure 3.4: Principal Variation Search with Futility Cutoffs

futility cutoffs is 26% (`-null+fcut`), as compared to 21% (`+null-fcut`) of the null move heuristic. Also, the success rate at depth 5 for futility cutoffs dropped from 20% to 18% of the null move heuristic. This deterioration in success rate comes partly from the nature of these two heuristics themselves but it could depend to the test suite we choose as well; using different test positions may yield different results. Combining both heuristics (`+null+fcut`) gives even greater savings (an average of 36%) without deteriorating the performance any further. For this reason, both the null move heuristic and the futility cutoffs are included in the current version of *Abyss* and will be enabled for all the later experiments in the thesis unless otherwise stated.

Chapter 4

Extensions With Domain Specific Knowledge

4.1 Introduction

With today's faster hardware and enhanced search algorithms, it is possible for the best chess playing programs to search to a formidable depth (say 9-ply or even deeper) during most of the middle game. Even for those programs, moves beyond the game tree horizon may still be neglected. To alleviate this problem, a more promising approach is adopted viz., *selective extensions*, increasing the search by an extra ply (plies) when certain criteria are met.

For instance, chess playing programs will usually extend the search by an extra ply when the side to move is in check, since checking usually consists of a serious threat. The safety of a deeper search is worth the extra cost, which isn't high since the number of replies to a checking move is small. This is one example of using domain-specific knowledge to extend the search depth. Other approaches include extending on recaptures [Ebeling 1987, pp. 101-102], pawn moves to the 6th and 7th rank in chess [Kaindl 1982, Scherzer *et al.* 1990], moves near the territory of the opponent's king [Anantharaman 1991], strictly forced moves (say if one side has only one legal move) [Uiterwijk 1991] and certain piece evading moves to bring a piece out of the opponent's attack (an *ad hoc* heuristic tried in *Abyss*). The latter two have not yet been adequately explored in computer chess literature, but are nonetheless two important aspects that are worth considering in extensions when using domain specific knowledge. Here, we shall provide the descriptions of these heuristics with experimental results, and share our experience in implementing them in the *Abyss* Chinese Chess playing program.

Using domain specific knowledge is one means to carry out selective extensions. Because of its simplicity in implementation and efficiency in performance, most chess playing programs adopt such an extension heuristic to some extent.

Note that the implementation of these heuristics is simple and straightforward. Suppose we have a function called *forcing(move, position)* which indicates whether the *move*, if made on the current *position*, falls in the definition of a *forcing move* (either a check evasion move, a recapture, a king-threat move, a piece evading move or a strictly forced move), the brute-force αβ search algorithm can be easily modified to suit this new need (Figure 4.1). So the remaining task is how to define such a forcing move and whether it should be used for extension during the search.

```
int pvs(side, position, α, β, depth)
  int side, position[90], α, β, depth;
{
  int successor[90], merit, ndepth, num_moves, p, score, lower;
  struct { int from, to; } moves[MAX_MOVES];

  if(depth <= 0)
    return(evaluate(side, position, α, β));

  num_moves = generate(side, position, moves);
  successor = make(moves[0], position);
  if(forcing(moves[0], position))
    score = -pvs(!side, successor, -β, -α, depth);
  else
    score = -pvs(!side, successor, -β, -α, depth-1);

  for(p = 1; p < num_moves; p++) {
    if(score >= β)
      goto out;
    lower = max(α, score);
    successor = make(moves[p], position);
    if(forcing(moves[p], position))
      ndepth = depth
    else
      ndepth = depth - 1;
    merit = -pvs(!side, successor, -lower-1, -lower, ndepth);
    if(merit > score)
      if(merit > α && merit < β)
        score = -pvs(!side, successor, -β, -merit, ndepth);
      else
        score = merit;
    }
  }

  out:
  return(score);
}
```

Figure 4.1: Principal Variation Search with Extension on Forcing Moves

As one might have noticed from Figure 4.1, the function *forcing()* here slightly deviates from the definition of forcing moves. For example, in the check evasion case, *forcing(move, position)* returns true if *move* is a check, therefore all nodes expanded in this layer (where the opponent's king is all in check) will be searched without a depth reduction. Of course, this is only one way of adding the extension heuristics on forcing moves. Other implementations are also possible.

4.2 Check Evasions

Because of its simplicity and efficiency, *check evasion* is perhaps the most commonly found feature in chess programs. A checking move usually forms a major threat and is a forcing move, therefore a one ply deeper search might reveal some tactics that are beyond the original horizon. The situation is of course substantially the same in Chinese Chess, so one can expect a similar benefit from adding such a heuristic.

The experimental results on the check evasion heuristic can be found in Table 4.1. The terms for *nodes*, *ratio* and *scores* have been defined in Section 3.2. For all the experiments throughout this chapter, the null move heuristic and futility cutoffs are both used as an enhancement for the αβ search algorithm. Also, the result from the original experiment (without any knowledge extensions) is included as a basis for comparisons.

Table 4.1: Experiment on Check Evasions

Extensions	Depth = 3			Depth = 4			Depth = 5		
	nodes	ratio	scores	nodes	ratio	scores	nodes	ratio	scores
-check	104234	1.00	6(18)	641469	1.00	16(30)	1887356	1.00	18(30)
+check	121842	1.17	20(32)	659071	1.03	24(38)	2684653	1.42	32(50)

Our results confirm the benefits of adding this heuristic (see Table 4.1) since, although adding the *check evasions* heuristic increased the nodes being searched (an average of 21% over depths 3 to 5), the cost is worthwhile because the success rate is increased (an average of 12 percentage points) as well as the correct move hits (an average of 14 percentage points).

4.3 Recaptures

Capturing is the essence of tactics in chess (and Chinese Chess), and a capture search [Bettadapur and Marsland 1988] forms the kernel part of quiescence search. In the exhaustive search region, some captures are more or less forced, e.g., *recaptures*, as defined by Carl Ebeling [1987, pp. 101-102]. Therefore it might be worthwhile to extend one more ply on recaptures with the hope that some deep tactics can be revealed.

Note that extending all recaptures could be expensive since a capture doesn't restrict the move choices by the opponent, therefore care is necessary to avoid a search explosion. In *Abyss*, we adopt the same rule as in *Hitech* [Ebeling 1987, Berliner 1989]; only recaptures that bring the material merit into a window of the initial root value are considered.

Table 4.2: Experiment on Recaptures

Extensions	Depth = 3			Depth = 4			Depth = 5		
	nodes	ratio	scores	nodes	ratio	scores	nodes	ratio	scores
–recapture	104234	1.00	6(18)	641469	1.00	16(30)	1887356	1.00	18(30)
+recapture	111711	1.07	6(22)	661337	1.03	16(28)	2290755	1.21	18(34)

Our experiments show that adding the recaptures alone doesn't improve the success rate (it however does improve the correct move hits by an average of 2 percentage points). Later we shall see how well the recaptures perform when combining other extension heuristics like check evasions (see Section 4.7).

4.4 King Threats

It has already been mentioned that the *king threat* heuristic can be used to alleviate the problem that computer chess playing programs normally lack the knowledge of recognizing the long-term threats against the king [Anantharaman 1991]. In Chinese Chess, the king is perhaps more vulnerable to attack than in chess, since it is confined to only nine squares (called *palace*). Since there is no pawn promotion rule, the chance is higher that the king is assaulted by the opponent. In fact, there is an adage in Chinese

Chess which says "Three pieces beside (the opponent's palace) wins the game"; therefore, it is more reasonable to use the king-threat heuristic in Chinese Chess.

Although, for simplification, the palace can be naturally used as the squares around one side's king, such a treatment might cause evaluation inaccuracies. In *Abyss* the same definition as in *Deep Thought* [Anantharaman 1991] is adopted. We consider the king is in jeopardy if the number of squares around one side's king (exactly one square away) that are under the opponent's attack exceeds a certain threshold (3 is chosen) and we extend the search for one more ply in this case.

The experimental results of adding the king-threat heuristic can be found in Table 4.3 (solely the king-threat). We have set up two different experiments to test the effectiveness of adding this heuristic. They are mainly aimed to decide how many plies (1 or 2 plies) we should extend in the case that a king-threat is detected, as well as to decide whether we should consider all the positions where the squares around one side's king exceed the threshold (`+all.1` and `+all.2`) or only consider it when such a position occurs after making a move (`+third.1` and `+third.2`) but not so before the move is made.

Table 4.3: Experiment on King-Threats

Extensions	Depth = 3			Depth = 4			Depth = 5		
	nodes	ratio	scores	nodes	ratio	scores	nodes	ratio	scores
-king	104234	1.00	6(18)	641469	1.00	16(30)	1887356	1.00	18(30)
+all.1	144057	1.38	8(24)	774636	1.21	16(30)	2613998	1.39	20(32)
+all.2	214165	2.05	8(24)	1136650	1.77	16(30)	4050674	2.15	22(34)
+third.1	144948	1.39	8(24)	802759	1.25	16(30)	2890065	1.53	22(34)
+third.2	155539	1.49	8(24)	871982	1.40	16(30)	3630385	1.92	22(34)

From the results shown in Table 4.3, we achieve an average improvement in the success rate (2 percentage points) and correct move hits (3 percentage points) but the nodes being searched also increase dramatically [a node ratio of 2.15 in the worst case (`all.2` at depth 5)]. Notice that all implementations yield the same results on move selections except one case (`all.1` at depth 5). It seems the best result comes from using the king-threat extension only for one ply, and only considering the situation where the squares around the king exceeds a threshold after one side has made a move (`+third.1`). Using this implementation, the same improvements on move quality can be maintained but with a moderate node increase (an average of 39% across depth 3, 4 and 5). Therefore, such a method will be chosen in *Abyss* as well as in the later experiments when king-threats are included. Further experiments on combining king-threats with other heuristics will be presented in Table 4.6 in Section 4.7.

4.5 Piece Evading Moves

Another possibility is to extend the search for moves that bring to safety a piece under attack. This may be viewed as a generalized case of the *check evasion* heuristic, since otherwise the piece under attack will be captured by the opponent on the next move, resulting in a material deficit. Moving the piece to safety (to a square where it is no longer under the opponent's attack or is protected by pieces of its own side) is somehow forced.

The other reason for using this *ad hoc* extension heuristic in the search algorithm in *Abyss* stems from consideration of the repetition rule of Chinese Chess. *Null moves* can usually be used to detect threats, but not all threats identified by the *null move* follow the rules of Chinese Chess, and the expense of such a detection is high. Therefore, some simplification to restrict the type of moves which are considered "threats" is made in *Abyss* (see Section 2.5 for more details about the repetition check algorithm). As a byproduct of detecting a *piece evading* move, which reveals one certain type of threat, we gain knowledge about how to distinguish a legal repetition from an illegal one.

In *Abyss*, a simplified version of this heuristic is included; it only considers moving a major piece out of attack, since including all types of pieces proved to be too expensive. Two experiments were conducted; in the first case (`+evade.1`), we try to check whether there is at least one opponent's piece under attack after one side has made a move (including discovering attacks). In the second case (+evade.2), we restrict the pieces being attacked to only those attacked by the last piece moved by the opponent.

Table 4.4: Experiment on Piece Evading Moves

Extensions	Depth = 3			Depth = 4			Depth = 5		
	nodes	ratio	scores	nodes	ratio	scores	nodes	ratio	scores
-evade	104234	1.00	6(18)	641469	1.00	16(30)	1887356	1.00	18(30)
+evade.1	125612	1.21	8(20)	786182	1.23	20(32)	4185770	2.22	26(40)
+evade.2	124867	1.20	8(18)	705624	1.10	18(30)	2420428	1.28	24(34)

The average node increase of using the `evade.1` and `evade.2` extension heuristics are 55% and 19% respectively, with the average success rate being increased by 5 and 3 percentage points (the correct move hits increased by 4 and 2 percentage points). In all our experiments, we see an improvement on move selections with a moderate ratio increase, except one case (+evade.1 at depth 5), where a node ratio of 2.22 was observed. Although the former implementation (+evade.1) yields a better performance in terms of success rate, we think the cost is perhaps not worthwhile since the search ratio is too high, although results from further experiments may reveal whether such a high ratio should still maintain at deeper search depth. For further experiments in the thesis, we shall use only the second implementation (+evade.2) for the piece-evading move heuristic (considering those pieces attacked by the last moved piece only).

4.6 Strictly Forced Moves

There is still another heuristic which can be used for extensions, strictly forced moves. This heuristic has been implemented in the chess playing program *Touch* [Uiterwijk 1991]. If there is only one legal move in a position, it is of course forced and a less turbulent and more reliable value may be returned after searching by an extra ply. Such a heuristic is especially useful in situations where one side can make a move which leads to a decisive advantage (like a mate threat) but the opponent can "thwart" this threat by making some delaying moves like checks. By disregarding these moves, it is possible that we can avoid being "fooled" into missing the threat. Notice that when there is only one legal move in a particular position, this usually means the king of the side to move is under check. Therefore a total of two extra plies will be extended when combining with the *check evasion* heuristic, searching deeper in a forced line without giving too much extra cost.

Table 4.5: Experiment on Strictly Forced Moves

Extensions	Depth = 3			Depth = 4			Depth = 5		
	nodes	ratio	scores	nodes	ratio	scores	nodes	ratio	scores
-strict	104234	1.00	6(18)	641469	1.00	16(30)	1887356	1.00	18(30)
+strict	107812	1.03	6(18)	678275	1.06	16(30)	2013468	1.07	18(36)

The heuristic for strictly forcing moves is implemented in *Abyss* and the result can be found in Table 4.5. No success rate increase was achieved in this case (although correct move hits increased by 6 percentage points at depth 5) with an average node increase of 5%. This is probably what we have expected, since the chance of having only one legal move is rare in the middle games. This heuristic was introduced to supplement the check evasion heuristic we have just discussed. It is possible that the extension on strictly forced moves may perform better when combined with check evasions (see Section 4.7).

However, such a heuristic might not be worth implementing at all when other search extension heuristics like singular extensions are included; a strictly forced move represents one special case for a PV-singular move (the only move is also the "significantly" best move). Please refer to Chapter 5 for discussions on singular extensions.

4.7 Combining Extensions With Domain Specific Knowledge

The experimental results of different combinations of search extensions with domain specific knowledge have been provided here in Table 4.6 (also used to support Table 4.8), based on the extension combinations provided in Table 4.7.

To decide the relative importance of each knowledge extension heuristic and their combinations, we have to compare not only the performance measures, but also the

Table 4.6: Experimental Results for Different Knowledge Extensions

Exp. No.	Depth = 3			Depth = 4			Depth = 5		
	nodes	ratio	scores	nodes	ratio	scores	nodes	ratio	scores
1	104234	1.00	6(18)	641469	1.00	16(30)	1887356	1.00	18(30)
2	121842	1.17	20(32)	659071	1.03	24(38)	2684653	1.42	32(50)
3	111711	1.07	6(22)	661337	1.03	16(28)	2290755	1.21	18(34)
4	144948	1.39	8(24)	802759	1.25	16(30)	2890065	1.53	22(34)
5	125612	1.21	6(18)	705624	1.10	20(32)	2420428	1.28	24(34)
6	107812	1.03	6(18)	678275	1.06	16(30)	2013468	1.07	18(36)
7	127350	1.22	18(32)	658167	1.03	22(36)	3062946	1.62	30(48)
8	155909	1.50	18(30)	869377	1.36	26(42)	4493750	2.38	30(52)
9	167284	1.60	24(34)	759998	1.18	24(38)	3510887	1.86	34(50)
10	136747	1.31	18(30)	854828	1.33	24(42)	3645802	1.93	36(50)
11	167878	1.61	16(30)	893620	1.39	24(40)	5309546	2.81	36(52)
12	166342	1.60	22(34)	784635	1.22	22(36)	4040137	2.14	32(46)
13	144298	1.38	16(30)	855107	1.33	24(42)	4467591	2.37	36(50)
14	214200	2.05	22(32)	1033796	1.61	24(40)	6248260	3.31	38(52)
15	190384	1.83	16(30)	1104580	1.72	30(46)	6693856	3.55	42(54)
16	195093	1.87	20(32)	1140422	1.78	26(42)	7364632	3.90	38(52)
17	249328	2.39	22(32)	1632788	2.55	30(46)	11574022	6.13	44(58)

Table 4.7: Key to Different Combinations for Knowledge Extensions

Exp. No.	Combinations
1	no extensions at all
2	check evasions
3	recaptures
4	king-threats
5	evading moves
6	strictly forced moves
7	check evasions + recaptures
8	check evasions + evading moves
9	check evasions + king-threats
10	check evasions + strictly forced moves
11	check evasions + recapture + evading moves
12	check evasions + recapture + king-threats
13	check evasions + recapture + strictly forced moves
14	check evasions + recapture + king-threats + evading moves
15	check evasions + recapture + strictly forced moves + evading moves
16	check evasions + recapture + strictly forced moves + king-threats
17	check evasions + recapture + strictly forced moves + evading moves + king-threats

search efficiency. Since the relative importance increases as the success rate increases, but decreases as the node ratio increases, a funtion defined as success rate over node ratio can be used. As a secondary factor, we should also consider the number of correct moves found (disregarding the scores), since this number shows the move quality as well. Therefore, to measure the relative importance *RI* of one particular extension heuristic, the

Table 4.8: Relative Importance of Knowledge Extensions

Exp. No.	Depth = 3			Depth = 4			Depth = 5			Avg. RI
	R	S(H)	RI	R	S(H)	RI	R	S(H)	RI	
1	1.00	6(18)	15.00	1.00	16(30)	31.00	1.00	18(30)	33.00	26.33
2	1.17	20(32)	30.77	1.03	24(38)	41.75	1.42	32(50)	40.14	37.55
3	1.07	6(22)	15.89	1.03	16(28)	29.13	1.21	18(34)	28.93	24.65
4	1.39	8(24)	14.39	1.25	16(30)	24.80	1.53	22(34)	25.49	21.56
5	1.21	6(18)	12.40	1.10	20(32)	32.73	1.28	24(34)	32.03	25.72
6	1.03	6(18)	14.56	1.06	16(30)	29.25	1.07	18(36)	33.64	25.82
7	1.22	18(32)	27.87	1.03	22(36)	38.83	1.62	30(48)	33.33	33.34
8	1.50	18(30)	22.00	1.36	26(42)	34.56	2.38	30(52)	23.53	26.70
9	1.60	24(34)	25.62	1.18	24(38)	36.44	1.86	34(50)	31.72	31.26
10	1.31	18(30)	25.19	1.33	24(42)	33.83	1.93	36(50)	31.61	30.21
11	1.61	16(30)	19.25	1.39	24(40)	31.65	2.81	36(52)	22.06	24.32
12	1.60	22(34)	24.37	1.22	22(36)	32.79	2.14	32(46)	25.70	27.62
13	1.38	16(30)	22.46	1.33	24(42)	33.83	2.37	36(50)	25.74	27.34
14	2.05	22(32)	18.54	1.61	24(40)	27.33	3.31	38(52)	19.34	21.74
15	1.83	16(30)	16.94	1.72	30(46)	30.81	3.55	42(54)	19.44	22.40
16	1.87	20(32)	19.25	1.78	26(42)	26.40	3.90	38(52)	16.41	20.69
17	2.39	22(32)	15.90	2.55	30(46)	20.78	6.13	44(58)	11.91	16.20

following formula is proposed:

$$RI = (S + H / 2) / R.$$

Here, *S* stands for the success rate, *H* is the correct move hits, and *R* represents the node ratio as compared to that case when no extensions are used (i.e., compare to Experiment No. 1). Using the above formula, the results of all the extensions experimented are presented in Table 4.8, where *Avg. RI* stands for average *RI*'s across depth 3, 4 and 5.

Although the criterion used here to calculate relative importance is crude, these statistics provide evidence to support use of certain extension heuristics. From Table 4.8, it can be seen that check evasion is the heuristic with the highest relative importance, both among those with a single extension heuristic and when the extensions are combined. Using more than two extension heuristics leads to too many nodes being searched without improving the success rate significantly. Also, we see that when combining only two extension heuristics, the combination of check evasions plus recaptures performs best, although the other two combinations, check evasions plus king-threats and check evasions plus strictly forced moves, are not far behind (based on average *RI*'s). This is also true for depth 5, a depth we are more interested in, since a search to depth 5 is perhaps the deepest level that a micro-ranged chess (Chinese Chess) program can achieve. Notice the *RI* values for Experiment 5 and 6 at depth 5 are also high, but nether is recommended because of their poorer move qualities.

There is another interesting phenomenon from Table 4.8. For most experiments with selective extensions, we see the relative importance increases from depth 3 to depth 4 but drops from depth 4 to depth 5. Perhaps this is related to the odd even search depth that is used. It could also be the effect of the law of diminishing returns on adding these extension heuristics. Searching to depth 6 (or deeper) may make us further understand and explain this phenomenon.

We can draw some conclusions from the experiments of using knowledge extensions. First, check evasion (Experiment 2) performs well in all cases and should be included in the chess playing programs when adopting selective extensions. Second, combining more than one extension heuristic yields a better success rate and correct move hits, but it is perhaps not worthwhile to combine more than two of them since the excessive nodes searched neutralizes the performance (e.g., Experiment 17). Third, when combining two extension heuristics, the best choice is to use check evasions plus recaptures (Experiment 7; already the most popular choice in chess playing programs design), although further experiments may be necessary to see whether the other two choices, check evasions plus king-threats (Experiment 9) and check evasions plus strictly forced moves (Experiment 10), can possibly improve the strength of the play.

Chapter 5

Extensions Without Domain Specific Knowledge

5.1 Introduction

Using domain-specific knowledge to extend search to a variable depth has proved to be beneficial, based on the experimental results in the previous chapter. Although experiments [Hyatt *et al.* 1990, Ye and Marsland 1992] show that a n-ply search enabled with extension heuristics still performs no better than a (n+2)-ply exhaustive search, it outperforms (n+1)-ply exhaustive search in two ways: first, it searches fewer nodes (time) and second, the move quality is better.

Nevertheless, there are two problems when adopting the selective extensions with domain-specific knowledge. As Anantharaman *et al.* [1988] state:

> "First, it is difficult to provide enough knowledge to cover all or most of the interesting cases. Second, the knowledge is usually based only on the static features of the moves without taking into account the dynamics of the position, and the search extensions based on such knowledge may be grossly irrelevant and wasteful."

A more powerful search extension heuristic called *Singular Extensions* was presented by Anantharaman *et al.* [1988]. Initially it was thought to be a major advance, although later experiments on singular extensions show that this heuristic is not as promising as it first appeared [Anantharaman 1991]. The idea of singular extensions is to use information gathered in the search itself to extend the search whenever one move is significantly better than the sibling moves.

Apart from applying the singular extensions, there is still another promising way to extend the search without using any domain-specific knowledge: the *Null Move Quiescence Search* (or *Second-order Quiescence Search*) as proposed by Beal [1989]. Originally, the null move quiescence search serves as a means to reduce the errors in the normal quiescence search phase. However, it we view this approach from another angle, we see that null move quiescence search provides an excellent means of extending the search based on the information gained from the search itself (the null move value from the first-order quiescence search). The search will thus be divided into three phases as the full-width search, the null move quiescence search and the normal quiescence search.

In this chapter, we give the descriptions of these search extension heuristics, as well as the implementation details from program *Abyss*. The experimental results are presented, together with a brief analysis of the performance.

5.2 Singular Extensions

There are two types of singular moves which are considered during the search; they are *Singular PV* moves and *Singular Fail-high* moves, the former applies to PV moves and the latter applies to non-PV moves. The principle of the singular extensions is to extend the search by an extra ply whenever a move is found to be "significantly" better than the sibling moves. Both extensions are implemented in the current version of *Abyss* and will be discussed in this section.

5.2.1 Extensions on Singular PV Moves

As defined by Anantharaman *et al.* [1988], "A move *m* is *singular* at some depth *d* if the value returned by a *d*-ply search of *m* is better than all siblings of *m* by a *significant* amount S, referred to as the *singular margin*." A minimal window search with the window *(V – S)* can be used to test the singularity, which costs nothing extra if the test finally succeeds. However, a re-search with the correct minimal window is necessary if the test fails. Another problem related to implementing the PV-singular extension is how to treat the new score returned from the extended search. If the new score drops more than the singular margin (in which case some other moves may be best), a re-search of all the siblings is required.

A more detailed discussion on the implementation issues can be found elsewhere [Anantharaman 1991]. In Figure 5.1, we give our illustrative algorithm of the function *pvs()* with consideration of the PV-singular extension. Here, *S* represents the singular margin and is chosen as 1/4 of a pawn's value. The functions *in_smoves()* and *add_smove()* are used for re-search purposes: *in_smoves()* determines whether a move has already been found PV-singular and extended; and *add_smove()* records every move that has been found PV-singular.

The result of adding solely the PV-singular extension heuristic can be found in Table 5.1. Two experimental results are presented in Table 5.1: (a) use the PV-singular extension only (+spv-chk) and (b) combine PV-singular extension with check evasions (+spv+chk). As mentioned earlier, the null move heuristic and the futility cutoffs are both turned on when carrying out these experiments.

Table 5.1: Experimental Results for PV-Singular Extensions

Experiment	Depth = 3			Depth = 4			Depth = 5		
	nodes	ratio	scores	nodes	ratio	scores	nodes	ratio	scores
-spv-chk	104234	1.00	6(18)	641469	1.00	16(30)	1887356	1.00	18(30)
+spv-chk	184719	1.77	6(24)	1195163	1.86	18(30)	3600092	1.91	18(32)
-spv+chk	121842	1.17	20(32)	659071	1.03	24(38)	2684653	1.42	32(50)
+spv+chk	254201	2.44	20(32)	1390137	2.17	24(36)	5741655	3.04	34(50)

It can be seen from Table 5.1 that adding the PV-singular extension brings little improvement in terms of success rate (a 2 percentage points increase was noticed in both experiments; -chk, depth 4 and +chk, depth 5) and the node increase is huge (an average increase of 85% and 134% for each experiment as compared with the version without the PV-singular extension). One reason may be the search depth chosen here (maximum 5) is too shallow to discover deep tactics. Further experiments will be carried out to determine whether adding the singular extensions can improve the play of the program (see Section 5.3).

```
#define Spv 25                              /* PV singular margin (1/4 pawn) */

int pvs(side, position, α, β, depth)
  int side, position[90], α, β, depth;
{
  int successor[90], merit, num_moves, p, score, bscore;
  boolean pv_singular, no_move;
  struct { int from, to; } moves[MAX_MOVES], bmove;
  struct { int moves[MAX_MOVES]; int num_moves = 0; } smoves;

  if(depth <= 0)
    return(evaluate(side, position, α, β));

  num_moves = generate(side, position, moves);
  bscore = -∞;

  re-search:          /* first move or re-search after singular extension */
  p = -1; score = -∞; pv_singular = no_move = TRUE;
  while(++p < num_moves && score == -∞)
    if(!in_smoves(moves[p], &smoves)) {                   /* found a move? */
      successor = make(moves[p], position);
      score = -pvs(!side, successor, -β, -α, depth-1);
      bmove = moves[p]; no_move = FALSE;
    }

  while(++p < num_moves) {
    if(score >= β)
      goto out;
    if(!in_smoves(moves[p], &smoves)) {
      successor = make(moves[p], position);
      if(pv_singular) {      /* test PV singularity, assumed true initially */
        merit = -pvs(!side, successor, -score-1+Spv, -score+Spv, depth-1);
        if((score - merit) <= Spv) {
          /* test fails, search with correct window */
          merit = -pvs(!side, successor, -score-1, -score, depth-1);
          pv_singular = FALSE;          /* current PV can't be PV singular */
        }
      }
      else
        merit = -pvs(!side, successor, -score-1, -score, depth-1);
      if(merit > score) {                      /* possible singular PV move */
        score = -pvs(!side, successor, -β, -merit, depth-1);
        bmove = moves[p]; pv_singular = TRUE;
      }
    }
  }

  if(pv_singular && !no_move) {                /* is best move PV singular? */
    successor = make(bmove, position);
    merit = -pvs(!side, successor, -β, -α, depth);          /* PV extension */
    if(merit > score || (score - merit) <= Spv)           /*not fail low? */
      score = merit;                              /* accept the new score */
    else {
      add_smove(bmove, &smoves);
      bscore = score;
      goto re-search;       /* re-search discounting the current best move */
    }
  }
  else if(no_move)              /* no moves, accept the previous best score */
    score = bscore;

  out:
  return(score);
}
```

Figure 5.1: Principal Variation Search with PV Singular Extension

5.2.2 Extensions on Singular Fail-high Moves

In the previous section, we have discussed the extension on PV-singular moves and its experimental results. In this section, we shall discuss the other way to apply the singular extensions to non-PV moves, i.e., *Fail-high Singular Extension.*

As defined by Anantharaman *et al.* [1988], "A *fail-high* move *h* is *fail-high* at depth *d* with a test reduction factor of *r* if the present PV value *v* is better than the (*d*–*r*)-ply values of all siblings of *h* by a *significant* amount *Sh*, referred to as the *singular fail-high margin.*"

The method for determining the fail-high singular moves is illustrated in Figure 5.2. The fail-high margin *Sh* is chosen as 75 points (3/4 of a pawn's value) and the reduction factor is fixed at 2 for all search depths. The experimental results are presented in Table 5.2 (only considering fail-high singular moves) and Table 5.3 (considering both PV-singular moves and fail-high singular moves). Two experimental results are presented in Table 5.2: (a) extend with fail-high singular moves only (`+sfh-chk`) and (b) combine fail-high singular extension with check evasions (`+sfh+chk`).

Table 5.2: Experimental Results for Fail-high Singular Extensions

Experiment	Depth = 3			Depth = 4			Depth = 5		
	nodes	ratio	scores	nodes	ratio	scores	nodes	ratio	scores
-sfh-chk	104234	1.00	6(18)	641469	1.00	16(30)	1887356	1.00	18(30)
+sfh-chk	176352	1.69	6(24)	1216168	1.90	18(28)	3585024	1.90	18(30)
-sfh+chk	121842	1.17	20(32)	659071	1.03	24(38)	2684653	1.42	32(50)
+sfh+chk	220643	2.12	20(28)	1278800	1.99	22(34)	5001444	2.65	34(54)

Like PV-singular extension, adding fail-high singular extension brings little improvement in terms of success rate. Up to 2 percentage points improvement was observed with an average node increase of 83% (`-sfh-chk` vs. `+sfh-chk`) and 105% (`-sfh+chk` vs. `+sfh+chk`) as compared with the version without fail-high singular extensions. Notice the 2 percentage points drop when combining the fail-high singular extension with check evasions, but we think this is perhaps more attributable to the non-

```
#define Sh 75                  /* fail-high singular margin (3/4 pawn) */

int pvs(side, position, α, β, depth)
  int side, position[90], α, β, depth;
{
  int successor[90], t_posn[90], score, merit, value, num_moves, p, i;
  boolean fail_high;
  struct { int from, to; } moves[MAX_MOVES];

  if(depth <= 0)
    return(evaluate(side, position, α, β));

  num_moves = generate(side, position, moves);
  successor = make(moves[0], position);                    /* first move */
  score = -pvs(!side, successor, -β, -α, depth-1);

  for(p = 1; p < num_moves; p++) {
    if(score >= β)
      goto out;
    successor = make(moves[p], position);
    merit = -pvs(!side, successor, -score-1, -score, depth-1);
    if(merit > score && merit > α && merit < β) {
      fail_high = TRUE;           /* assume it is fail-high singular */
      for(i = p + 1; i < num_moves && fail_high; i++) {
         /* test fail-high singularity, depth reduction 3 is chosen */
        t_posn = make(moves[i], position);
        value = -pvs(!side, t_posn, -score-1+Sh, -score+Sh, depth-3);
        if((score - value) <= Sh)                        /* test fails */
           fail_high = FALSE;
      }
      if(fail_high) {           /* new PV move is fail-high singular? */
        value = -pvs(!side, successor, -β, -merit, depth);
        if(value > score)          /* not fail low after extension? */
          score = value;
      }
      else
        score = -pvs(!side, successor, -β, -merit, depth-1);
    }
    else if(merit > score)
      score = merit;
  }

  out:
  return(score);
}
```

Figure 5.2: Principal Variation Search with Fail-high Singular Extension

perfect success rate calculation method rather than to the heuristic of fail-high singular extension itself. Further experiments will be carried out to determine whether singular extensions can improve the play of the program (see Section 5.3).

5.2.3 Experimental Results on Singular Extensions

In the above sections, we have presented our initial experimental results on adding the singular extensions heuristics in *Abyss* where the PV-singular moves and the fail-high singular moves were considered separately. In this section, we shall give our results of combining both PV-singular and fail-high moves together with some other knowledge extension heuristics like check evasions and recaptures. The results for these experiments can be found in Table 5.3 and the experiment setups can be found in Table 5.4.

Table 5.3: Experimental Results for Singular Extensions

Exp. No.	Depth = 3			Depth = 4			Depth = 5		
	nodes	ratio	scores	nodes	ratio	scores	nodes	ratio	scores
1	104234	1.00	6(18)	641469	1.00	16(30)	1887356	1.00	18(30)
2	184719	1.77	6(24)	1195163	1.86	18(30)	3600092	1.91	18(32)
3	176352	1.69	6(24)	1216168	1.90	18(28)	3585024	1.90	18(30)
4	188481	1.81	6(24)	1201698	1.87	18(30)	3551248	1.88	18(34)
5	121842	1.17	20(32)	659071	1.03	24(38)	2684653	1.42	32(50)
6	254201	2.44	20(32)	1390137	2.17	24(36)	5741655	3.04	34(50)
7	220643	2.12	20(28)	1278800	1.99	22(34)	5001444	2.65	34(54)
8	258545	2.48	20(32)	1400774	2.18	24(36)	5683926	3.01	34(50)
9	268708	2.58	20(32)	1411165	2.20	24(36)	5764677	3.05	34(50)

Table 5.4: Experiment Setup for Singular Extensions

Exp. No.	Combinations
1	– PV-singular extensions – Fail-high singular extensions – check evasions – recaptures
2	+ PV-singular extensions – Fail-high singular extensions – check evasions – recaptures
3	– PV-singular extensions + Fail-high singular extensions – check evasions – recaptures
4	+ PV-singular extensions + Fail-high singular extensions – check evasions – recaptures
5	– PV-singular extensions – Fail-high singular extensions + check evasions – recaptures
6	+ PV-singular extensions – Fail-high singular extensions + check evasions – recaptures
7	– PV-singular extensions + Fail-high singular extensions + check evasions – recaptures
8	+ PV-singular extensions + Fail-high singular extensions + check evasions – recaptures
9	+ PV-singular extensions + Fail-high singular extensions + check evasions + recaptures

Table 5.3 shows that adding the singular extensions hardly improves the success rate in our experiments (no improvement comparing Experiment 1 with Experiment 4 and a total of 2 percentage points comparing Experiment 5 with Experiment 8). Also, adding recapture to the suite of both singular extensions and check evasions (Experiment 9) yields no performance improvement. One explanation is that the test positions chosen are too sophisticated to be understood by a search to depth 5. Not including all the

enhancements for singular extensions is another reason. Also, the results could be related to the quality of the evaluation function. Nevertheless, it is still possible the program with singular extensions will play better. Therefore, two supplementary experiments were conducted below for further comparisons.

We decide to let the two versions of *Abyss*, *Abyss+SIN* (with singular extensions) and *Abyss–SIN* (without singular extensions), play a certain number of games. In both programs, the extensions for check evasions and recaptures are turned on. We randomly choose 12 frequently-occurring opening positions (5 to 8 moves from the start) and let two programs play each other with one opening exactly twice (one with red and one with black). This gives a total of 24 games. We believe the number 24 is large enough to reduce any abnormalities that may occur during the play.

To make the comparison "fair" for each version with different extension heuristics, we decide to spend a fixed amount of time on each move. Two time intervals were chosen: (1) 30 seconds per move, which normally guarantees a depth 4 search; and (2) 100 seconds per move, which normally guarantees a depth 5 search. When time is up, the best move found so far is chosen and there are no further searches. Also, the game is adjudicated by the author after 150 moves to save time on the meaningless plays by the programs. With all these conditions, a total of 48 games were played and the results can be found in Table 5.5. The opening positions that are used to play these 48 games can be found in Appendix 3.

In Table 5.5, a win counts for 1 point, a draw 0.5 point and a loss 0 point. It can be seen that the version with singular extensions performs consistently better than the one without this heuristic (the margins are 3 and 5 points or a winning rate of 56% and 60% respectively). Many other enhancements [Anantharaman 1991] can possibly make the search effort even more worthwhile. Nevertheless, even with our basic implementation of the singular extensions, the improvement is still obvious enough to suggest that it is

Table 5.5: Game Play Scores Between Two Different Versions of Abyss

Openings	30 sec. per move		100 sec. per move	
	Abyss+SIN	Abyss-SIN	Abyss+SIN	Abyss-SIN
1	0	2	1	1
2	1	1	0.5	1.5
3	1.5	0.5	1.5	0.5
4	2	0	1.5	0.5
5	1.5	0.5	1	1
6	1.5	0.5	1	1
7	0.5	1.5	1	1
8	1	1	2	0
9	1	1	2	0
10	1	1	0.5	1.5
11	1	1	2	0
12	1.5	0.5	0.5	1.5
Totals	13.5	10.5	14.5	9.5

worthwhile to include these extensions. Although for microcomputers, it may be more efficient to consider only one of them (e.g., PV-singular extensions only).

5.3 Null Move Quiescence Search

Null Move Quiescence Search, or Second-Order Quiescence Search [Beal 1989] provides a different means of carrying out a selective search. Defining the usual quiescence search (e.g., capture search) as the first-order quiescence search, this second-level quiescence search is able to discover as many tactical threats as the full-width search without adding any more special domain-specific knowledge. Since null move quiescence search provides a cheaper means of detecting tactics in chess (or Chinese Chess), a new search framework can be introduced. We shall be able to do a full-width to some shallower depth (as compared with solely full-width search) using the second-order quiescence for its terminal nodes. A slower static evaluation function can be installed in this case. In the extreme, we shall be able to do a 1-ply full-width search with positional factors being considered and then null move quiescence search considering only materials [Beal 1991].

5.3.1 Basic Implementation Issues

Some implementation issues of the null move quiescence search are discussed here. These are mainly the enhancements with the transposition table management, the iterative deepening and the futility cutoffs. All these heuristics provide a means to increase the search efficiency.

5.3.1.1 Transposition Table Management

When implementing the null move quiescence search, transposition tables can not only be used in the full-width search, but in the null move quiescence search as well. However, since different criteria are used in the two different search phases, some considerations have to be made to correctly and efficiently utilize the transposition tables.

The difference between the information stored in the two transposition tables is that the first-order quiescence search (capture search) has been used to either raise the lower bound or give cutoffs. In other words, the value obtained from the second-order quiescence search is different (or less reliable) than the value returned from a full-width search. Hence, the value returned from a second-order quiescence search cannot be used as a valid value or to update bounds in the full-width search.

The content of each transposition table entry can be found in Figure 5.3. Notice two depth values are stored in the transposition table: one for full-width search (*fdepth*; stored as *flength*) and the other for null move quiescence search (*qdepth*; stored as *qlength*). Figure 5.4 shows how the two search depths correspond to the search phases. The pseudo codes for implementing the transposition table are illustrated in Figure 5.5 and Figure 5.6, (function *pvs()* is for full-width search and function *nmqs()* is for second-order quiescence search). Here we use the standard treatment originally given by Marsland and Campbell [1982] and later refined by Marsland [1992].

In function *pvs()*, the values stored in the transposition table can only be used when

```
struct {
  unsigned long lock;
  short move;                /* best move for the current position */
  short score;                 /* score for the current position */
  char  flength;                  /* depth for full-width search */
  char  qlength;                  /* depth for quiescence search */
  char  flag;                /* either VALID or LBOUND or UBOUND */
}
```

Figure 5.3: Information Stored in Transposition Tables

the two depths for the previously searched sub-tree are no less than that of the current sub-tree. In function *nmqs()*, however, we shall accept the values stored in the table whenever the sum of the depths from the table is no less than that for the current position. Since full-width search is more reliable than null move quiescence search, we should be able to use the value from a full-width search to a null move quiescence search. For function *pvs()*, two sub-tree depths are used; *fdepth* is for the depth in the full-width search phase and, *qdepth* is for null move quiescence search. However, only one sub-tree depth (*qdepth*) is provided for function *nmqs()*, since *nmqs()* is called at the tip node of *pvs()*, therefore the full-width search depth is always zero for the null move quiescence search and thus can be omitted.

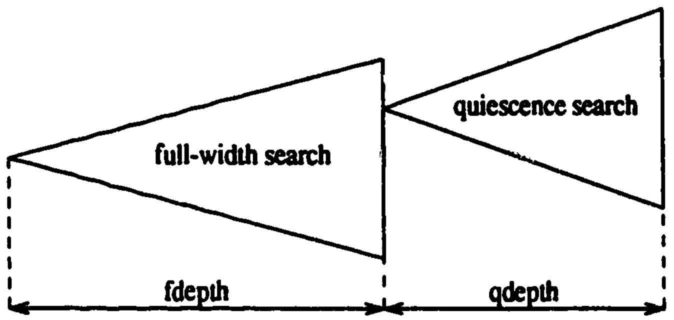


Figure 5.4: Depths Used for Null Move Quiescence Search

There is another possibility of utilizing the transposition tables when applying the null move quiescence search, i.e., the value for the first-order quiescence search (capture search). However, it should be noted that some different treatment is necessary if a transposition table is to be used for this purpose, since a capture search, unlike a full-width search, considers capture moves only and therefore some information stored in the

```
int pvs(side, position, α, β, fdepth, qdepth)
  int side, position[90], α, β, fdepth, qdepth;
{
  int successor[90], score, merit, lower, num_moves, p,
      flength, qlength, flag;
  struct { int from, to; } moves[MAX_MOVES], tmove;

  if(retrieve(position, &merit, &flag, &tmove, &flength, &qlength))
    if(legal(tmove) && flength >= fdepth && qlength >= qdepth) {
      switch(flag) {
      case VALID:
        return(merit);
      case LBOUND:
        α = max(α, merit); break;
      case UBOUND:
        β = min(β, merit); break;
      }
      if(α >= β)
        return(merit);
    }

  if(fdepth <= 0)              /* horizon node in full-width search? */
    return(nmqs(side, position, α, β, qdepth)); /* null move search */

  score = -∞;
  if(legal(tmove)) {           /* move found in transposition table? */
    successor = make(tmove, position); bmove = tmove;
    score = -pvs(!side, successor, -β, -α, fdepth-1, qdepth);
    if(score >= β)
      goto out;
  }

  num_moves = generate(side, position, moves);
  for(p = 0; p < num_moves; p++)
    if(moves[p] != tmove) {
      successor = make(moves[p], position);
      lower = max(α, score);
      merit = -pvs(!side, successor, -β, -lower, fdepth-1, qdepth);
      if(merit > score) {
        score = merit;
        bmove = moves[p];
      }
      if(score >= β)
        goto out;
    }

  out:
  if(fdepth >= flength && qdepth >= qlength && score > -∞) {
    flag = (score >= β)? LBOUND: (score <= α)? UBOUND: VALID;
    store(score, flag, bmove, fdepth, qdepth);
  }
  return(score);
}
```

Figure 5.5: Transposition Tables in Full-width Search

```
int nmqs(side, position, α, β, qdepth)
  int side, position[90], α, β, qdepth;
{
  int successor[90], score, merit, lower, num_moves, p,
      flength, qlength, flag,
  struct { int from, to; } moves[MAX_MOVES], tmove;

  if(retrieve(position, &merit, &flag, &tmove, &flength, &qlength))
    if(legal(tmove) && (flength + qlength) >= qdepth) {
      switch(flag) {
        case VALID:
          return(merit);
        case LBOUND:
          α = max(α, merit); break;
        case UBOUND:
          β = min(β, merit); break;
      }
      if(α >= β)
        return(merit);
    }

  if(qdepth <= 0)    /* horizon node in null move quiescence search? */
    return(evaluate(side, position, α, β));       /* capture search */

  score = -∞;
  if(legal(tmove)) {        /* search transposition table move first */
    successor = make(tmove, position); bmove = tmove;
    score = -nmqs(!side, successor, -β, -α, qdepth-1);
    if(score >= β)
      goto out;
  }

  /* null move quiescence search */
  merit = -evaluate(!side, position, -β, -max(α, score));
  if(merit > score) {
    score = merit;
    if(score >= β)
      goto out;
  }

  num_moves = generate(side, position, moves);
  for(p = 0; p < num_moves; p++)
    if(moves[p] != tmove) {
      successor = make(moves[p], position);
      lower = max(α, score);
      merit = -nmqs(!side, successor, -β, -lower, qdepth-1);
      if(merit > score) {
        score = merit;
        bmove = moves[p];
      }
      if(score >= β)
        goto out;
    }

  out:
  if(qdepth >= (flength + qlength) && score > -∞) {
    flag = (score >= β)? LBOUND: (score <= α)? UBOUND: VALID;
    store(score, flag, bmove, 0, qdepth);
              /* full-width search depth for current sub-tree is zero */
  }
  return(score);

}
```

Figure 5.6: Transposition Tables in Null Move Quiescence Search

transposition table, like bounds for a position, cannot be efficiently exploited. The current version of *Abyss* doesn't include this enhancement and it will be left for future work.

5.3.1.2 Iterative Deepening

It has been pointed out by Beal [1991] that when implementing the null move quiescence search, it is necessary to use iterative deepening to keep the cost to a reasonable level. In practice, three alternative ways may be considered. Suppose we are doing a search with full-width depth of *d* and null move quiescence depth of *q*, and we represent each iteration with the pair (full-depth, quiescence-depth). The first approach is to finish the null move quiescence search with the shallowest full-width depth and then keep on incrementing the full-width depth with the deepest null move quiescence depth; i.e., the iterations will go like (1, 1), (1 ,2), ... (1, q), (2, q), ... (d, q).

The second approach is to finish the full-width search with the shallowest quiescence depth first before going deeper in the null move quiescence search; i.e., the iterations will go like (1, 1), (2, 1), ... (d, 1), (d, 2), ... (d, q). Both of these approaches have been implemented in *Abyss* and are illustrated in Figure 5.7 and Figure 5.8, respectively. In both figures, *fdepth* stands for the maximum full-width search depth and *qdepth* is the maximum null move quiescence search depth.

```
for(d = 1; d <= fdepth; d++) {
  if(d == 1)
    q = 1;
  else
    q = qdepth;
  while(q <= qdepth) {
    ...
    pvs(α, β, d, q)
    ...
    q++;
  }
}
```

Figure 5.7: Iterative Deepening on Null Move Quiescence Search First

```
for(d = 1; d <= fdepth; d++) {
  if(d < fdepth)
    q = 1;
  else
    q = qdepth;
  for(i = 1; i <= q; i++) {
    ...
    pvs(α, β, d, i);
    ...
    q++;
  }
}
```

Figure 5.8: Iterative Deepening on Full-width Search First

There is also a third way of adding the iterative deepening, i.e., the internal iterative deepening. In this case, the iterative deepening applies not only to the root, but to the tip node for full-width search (root for null move quiescence search) as well. However, no experiments were carried out on this implementation because we believe the difference lies only in search efficiency and it should have little effect on move selections.

From Table 5.6, we can see the difference is obvious: the iterative deepening on null move (quiescence) search first searches more nodes as well as yielding a poorer success rate. In the subsequent experiments, we shall use the second implementation of the null move quiescence search, i.e., iterative deepening on full-width search first.

Table 5.6: Iterative Deepening in Null Move Quiescence Search

(d, q)	Iterative deepening on null move search first			Iterative deepening on full-width search first		
	nodes	ratio	scores	nodes	ratio	scores
(2, 1)	196059	1.00	4(22)	190724	0.97	2(20)
(2, 2)	955020	1.00	18(32)	890854	0.93	18(34)
(2, 3)	2227904	1.00	20(32)	2129815	0.96	20(36)
(2, 4)	9363120	1.00	36(44)	8240188	0.88	40(48)
(2, 5)	30344799	1.00	38(50)	17992862	0.59	40(48)

5.3.1.3 Futility Cutoffs

In Chapter 3, we show that the heuristic of futility cutoffs has proved to be successful in the full-width search. In the case where null move quiescence search is adopted, it is still possible to exploit this heuristic as an attempt to reduce some unsuccessful node expansions. Although the heuristic is best applied at the frontier nodes (in the null move quiescence search phase), it is possible that such a heuristic could also be worth trying at all nodes in the quiescence search. There is even another possibility, i.e., to use futility cutoffs at frontier nodes of the full-width search, but the chance for success here seems intuitively to be doomed since this will mainly discard the moves that cover the threats found by the null move quiescence search.

Further discussion may be necessary, but the best solution is through practice. Here we provide our experimental results on the possible improvements of null move quiescence search when futility cutoffs are used (Table 5.7); they are, applying the futility cutoff only at the frontier nodes of the quiescence search (`+fut_front`) and at all nodes of the quiescence search (`+fut_all`). In all experiments, a fixed depth (1-ply) for full-width search is adopted and the depth for null move quiescence search is represented as *Qdepth*.

Table 5.7: Futility Cutoffs in Null Move Quiescence Search

Qdepth	-fut			+fut_front			+fut_all		
	nodes	ratio	scores	nodes	ratio	scores	nodes	ratio	scores
1	44615	1.0	4(16)	44615	1.0	4(16)	44615	1.0	4(16)
2	160107	1.0	2(12)	158634	0.99	2(12)	158634	0.99	2(12)
3	498790	1.0	12(24)	496286	0.99	12(24)	479815	0.96	10(24)
4	1196771	1.0	12(26)	1185584	0.99	12(26)	1118394	0.93	12(24)
5	3456981	1.0	30(46)	3443252	1.0	30(46)	3126651	0.90	28(44)
6	7699981	1.0	34(48)	7647224	0.99	34(48)	6911076	0.90	30(46)

From Table 5.7, it can be concluded that adding futility cutoffs in the null move quiescence search brings little success. If applying the futility cutoffs at all nodes during the null move quiescence search, a maximum node account saving of 10% is possible but the success rate also deteriorates (up to 4 percentage points). Using the futility cutoffs only at frontier nodes gives almost no savings in node account (up to 1%) but preserves the success rate. This is because the game-tree of null move quiescence search naturally shrinks to avoid searching those "non-tactical" moves and explains why no futility cutoffs will be used in all the following experiments.

5.3.2 Experimental Results on Null Move Quiescence Search

The experimental results on null move quiescence search are presented in Table 5.8. A total maximum depth of 7 is chosen, bringing the maximum node count close to that of the full-width search, so further cross-comparisons with different extension heuristics are possible. Some experiments with full-width search depth of 2 are also carried out. In all the experiments, positional evaluations are only considered in the full-width search but not in the null move quiescence search and the capture search. In Table 5.8, *Depth* and *Qdepth* are the search depth for full-width and null move quiescence search respectively. The average branching factors (*a.b.f.*) between two consecutive quiescence search depths are also calculated (which is the geometric average on every single branching factor). The node ratio is obtained from comparing two searches with the same total search depth.

Table 5.8: Experiments on Null Move Quiescence Search

Qdepth	Depth = 1				Qdepth	Depth = 2			
	nodes	ratio	a.b.f.	scores		nodes	ratio	a.b.f.	scores
1	44615	1.00	1.00	4(16)	0	74565	1.67	1.00	2(22)
2	160107	1.00	3.80	2(12)	1	192764	1.20	3.57	2(20)
3	498790	1.00	2.96	12(24)	2	890854	1.79	4.41	18(34)
4	1196771	1.00	2.40	12(26)	3	2227904	1.86	2.52	20(36)
5	3456981	1.00	2.55	30(46)	4	8240188	2.38	3.03	40(48)
6	7699981	1.00	2.07	34(48)	5	17992862	2.34	2.05	40(48)

From Table 5.8, we can see null move quiescence search performs well in terms of success rate relative to the node count. For instance, a null move quiescence search with depths (1, 5) yields a success rate of 30(46) with a total node count of 3456981. On the other hand, a full-width search at depth 5 when not considering any extensions only gives a success rate of 22(34) with a total node count of 4153931 (Table 3.1). This clearly shows the effectiveness of the null move quiescence search in detecting tactics. In the above case, a total depth of 6 is achieved when using the null move quiescence search, and fewer nodes are expanded as compared to a full-width search at depth 5.

5.3.3 Combining Other Extensions

When using the null move quiescence search, it is also possible to add other selective extension heuristics like knowledge extensions and singular extensions as we have discussed previously. In this section, we consider experiments with one particular extension heuristic, the check evasions. Adding other extension heuristics is worth trying but will be left for future work.

In Table 5.9, we present experimental results on adding the check evasions when implementing the null move quiescence search. Two more experiments are done: (a) to add check evasions only in null move quiescence search (`-chk1+chk2`) and (b) add check evasions in both full-width and null move quiescence search (`+chk1+chk2`). The two depths for full-width and null move search are presented in a depth pair *(d, q)*.

Table 5.9: Adding Check Evasions in Null Move Quiescence Search

(d, q)	No extensions -chk1-chk2			Extensions in null move search -chk1+chk2			Extensions for both phases +chk1+chk2		
	nodes	ratio	scores	nodes	ratio	scores	nodes	ratio	scores
(1, 1)	44615	1.00	4(16)	45611	1.02	4(16)	45611	1.02	4(16)
(1, 2)	160107	1.00	2(12)	158322	0.99	2(18)	170317	1.06	6(18)
(1, 3)	498790	1.00	12(24)	523382	1.05	14(30)	567449	1.14	16(28)
(1, 4)	1196771	1.00	12(26)	1427207	1.19	26(46)	1608529	1.34	30(44)
(1, 5)	3456981	1.00	30(46)	3951971	1.14	32(54)	4481973	1.30	34(50)
(1, 6)	7699981	1.00	34(48)	12316527	1.60	36(56)	13751614	1.79	38(52)
(2, 1)	190724	1.00	2(20)	186890	0.98	10(34)	201980	1.06	16(36)
(2, 2)	890854	1.00	18(34)	966927	1.09	28(40)	1009684	1.13	28(40)
(2, 3)	2227904	1.00	20(36)	2743197	1.23	38(50)	3055821	1.37	38(48)
(2, 4)	8240188	1.00	40(48)	9096365	1.10	40(60)	10264895	1.25	42(58)
(2, 5)	17992862	1.00	40(48)	26591660	1.48	42(62)	31619370	1.76	42(60)

The extension on check evasions, as in the full-width search, combines well with the null move quiescence search. The experiments of adding the check evasions solely in the null move search phase gives an average success rate increase of 5 percentage points (mostly comes from the search at a total depth of 5) with a node count increase of 17%. Including the check evasions at full-width search gives a further 2 percentage points increase on success rate with an average node increase of 29%. We can thus conclude that the check evasion heuristic performs well when combined with null move quiescence search. Also, for searches with the same total depth, the one with a deeper full-width search depth performs significantly better than the other but also yields a higher node ratio. However, the encouraging result is that with a deeper full-width search depth (2 here), a search with a shallower null move search depth can outperform a search with a shallower full-width search depth but a deeper total search depth. For example, compare experiments with check evasions and search depths (2, 4) and (1, 6): the former performs clearly better both in terms of node count and success rate. Further, we see little improvement on success rate from search depths (2, 4) to (2, 5). This could be temporary; searching to deeper depths may result in a better success rate, but we think this can be more related to the nature of null move quiescence search itself. As pointed out by Newborn when discussing the *Gamma Algorithm* [Newborn 1975], most of the tactical threats in chess can be detected by a 4-ply full-width search. Null move quiescence search should be able to detect threats more efficiently, but loses its benefit when its depth goes beyond 4. Perhaps the full-width search depth should be deep enough to hold this assumption and that's why we don't see such a phenomenon for a shallower full-width search depth.

Apart from the check evasions, there are of course other possibilities for adding extension heuristics in the null move quiescence search. Knowledge extensions, as described in Chapter 4, and singular extensions can all be included for better performance. Further experiments are necessary to determine which heuristics work well in the null move quiescence search. Also, we may search to a deeper full-width depth to further reveal the impacts of different depth combinations.

5.4 Relative Importance of Extension Heuristics

In the earlier sections of this thesis, we have discussed the selective extension heuristics and presented our experimental results. The heuristics we are interested in include knowledge extensions, singular extensions and null move quiescence search. We have shown the effectiveness of adding these heuristics to the normal $\alpha\beta$ search framework. Yet we have to determine the relative importance among these heuristics and this will be carried out in this section.

According to Marsland and Rushton [1973], Gillogly [1978], Thompson [1982] and Schaeffer [1986], there are two traditional techniques for comparing two programs without human intervention. They are: (1) let two programs play a certain number of games, using some standard opening positions; (2) measure the performance of two programs on a benchmark and compare the number of correct solutions found by each program. The former technique is perhaps the more accepted way nowadays, but it has a drawback of demanding too much time. We have already used this technique as a supplementary way to compare the program with singular extensions against the program without singular extensions in Section 5.2.3. However, since we are carrying out the experiments in a new domain, we are interested not only in the performance of different extension heuristics, but also in their relative search efficiencies. To keep consistency, only the latter technique (measure the performance on a benchmark) is used.

In the previous experiments, we have compared the performance on an average basis. Here, we shall only compare a heuristic if it achieves a success rate better than 30%, excluding those with a node count more than triple the one with no extensions at all (*ratio* less than 3.0). For full-width search, such a success rate can be achieved by a search to depth 5 when combining at least check evasions. The depth 5 is critical for microcomputers where a search only to depth 5 can be acquired under normal tournament conditions (40 moves per 2 hours). Null move quiescence search with a total depth of 5 can achieve the same result when combining check evasions, but with one difference, it searches fewer nodes. Notice the results on singular extensions will not be included under this design since the node ratios all exceed the required bound (Table 5.3).

The same formula introduced in Section 4.7 to calculate relative importance will be used here. The results of these calculations can be found in Table 5.10. Apart from the total nodes and the items used to calculate the relative importance, the rank for each heuristic combination is also appended. All combinations excluding *Null Move* (which refers to null move quiescence search) are full-width searches to a depth 5 with the extensions shown in the table. If it is a null move quiescence search, two depths are given in brackets, indicating the depth for full-width search and null move quiescence search. The check evasion combinations for null move quiescence search are described either as *2nd-order* (only in null move quiescence search) or *Both* (in both full-width search and null move quiescence search), if they are enabled for one experiment. *Rate* and *hits* stand for *success rate* and *correct move hits*, respectively.

Table 5.10: Relative Importance of Different Extensions

Combination	nodes	ratio	rate	hits	RI	rank
No extensions	1887356	1.00	18	30	33.00	6
Check Evasions	2684653	1.42	32	50	40.14	3
Check Evasions & Recaptures	3062946	1.62	30	48	33.33	5
Check Evasions & Piece Evasions	4493750	2.38	30	52	23.53	14
Check Evasions & King-Threats	3510887	1.86	34	50	31.72	7
Check Evasions & Forced Moves	3645802	1.93	36	50	31.61	8
Check Evasions & Recaptures & Piece Evasions	5309546	2.81	36	52	22.06	15
Check Evasions & Recaptures & King-Threats	4040137	2.14	32	46	25.70	12
Check Evasions & Recaptures & Forced Moves	4467591	2.37	36	50	25.74	11
Null Move (1, 5) & No Extensions	3456981	1.83	30	46	28.96	9
Null Move (1, 5) & Check Evasions 2nd-order	3941971	2.09	32	54	28.23	10
Null Move (2, 3) & Check Evasions 2nd-order	2743197	1.45	38	50	43.45	2
Null Move (1, 4) & Check Evasions Both	1608529	0.85	30	44	61.18	1
Null Move (1, 5) & Check Evasions Both	4481973	2.37	34	50	24.89	13
Null Move (2, 3) & Check Evasions Both	3055821	1.62	38	48	38.27	4

Table 5.10 clearly shows that null move quiescence search, when combined with other extensions like check evasions, performs much better than other heuristics in terms of success rate and node count [Null Move (1, 4) & Check Evasions Both, ranking 1st; Null Move (2, 3) & Check Evasions 2nd-order, ranking 2nd]. It also shows that searching with more than 1 ply in the full-width phase can increase the success rate [an 8 percentage points increase from Null Move (1, 4) & Check Evasions Both to Null Move (2,

3) & Check Evasions 2nd-order]. If we only consider full-width search, check evasions (ranking 3rd) is the most important extension heuristic and check evasions plus recaptures (ranking 5th) is next, though in terms of success rate, check evasions plus king-threat (ranking 7th) and check evasions plus strictly forced moves (ranking 8th) are relatively better.

However, we still have to solve the problem of determining the relative playing strength. Since the strength of null move quiescence search is on tactics, it is possible that a program with null move quiescence search will play positionally worse than a program with only full-width search. Another related problem is how to implement the time-control routine for null move quiescence search if we want to spend a fixed amount of time on each move, because we have two depths to increase now. Should we finish a full-width search to a fixed depth first (e.g., full-width search to depth 2 and then null move quiescence search forever until the alarm rings)? Or should we finish a null move quiescence search to a fixed depth first (e.g., null move quiescence search to depth 4 with 1-ply full-width search and then full-width search forever)? Or should we increase both depths in turns? Certainly, further experiments are required to answer these questions.

Chapter 6

Conclusions

In this thesis, we have experimented with the selective search extension heuristics as well as the enhancements on the $\alpha\beta$ search algorithm in the domain of Chinese Chess. The extensions include knowledge extensions such as check evasions, recaptures, king-threats, piece-evading moves and strictly forced moves, singular extensions and null move quiescence search. The results for $\alpha\beta$ enhancements like interior move ordering heuristics (transposition tables, refutation tables and history heuristic), null move search and futility cutoffs are also presented as a basis for the experiments on selective extensions. The following conclusions are based on the experimental results as presented in the previous chapters.

First, we conclude that the $\alpha\beta$ enhancements successfully carry over from chess to a game like Chinese Chess. The interior move ordering heuristics experimented proved to be as efficient as for chess and more than a half node count saving was achieved when combining heuristics such as transposition tables, refutation tables and history heuristic (Section 3.2). Null move search and futility cutoffs can decrease the node count although they slightly lower the move qualities sometimes. However, as our experiments show, the savings for the null move heuristic [Goetsch and Campbell 1990] in Chinese Chess is not as great as for chess. This could be a property of the test suite we chose for the experiments, but it also could be attributed to the difference between the two games. Because of the more tactical nature of Chinese Chess, it is possible that heuristics like null move search and futility cutoffs perform less efficiently and error-free. Nevertheless, because of their great savings, even for Chinese Chess, it is still worthwhile to include them as an enhancement for normal $\alpha\beta$ search.

Second, we show that check evasion is the heuristic with the highest relative importance; it proved to be most search efficient with most improvement on move qualities.

Also, when combining all extension heuristics together (excluding null move quiescence search), we see that it is possible to have better move selections when adding more extensions. However, combining all of them proved to be not worthwhile, as this brings the node count to a level higher than that of a brute-force search with a deeper search depth. Adding one more carefully chosen extension heuristic to check evasions can improve the performance and we see that check evasion plus recaptures is the best combination in this case, although two other combinations, check evasion plus king-threats and check evasion plus strictly forced moves, are both worth trying (Section 4.7).

Third, using the basic implementation of singular extensions didn't prove to be efficient in terms of correct moves found relative to node count: the nodes searched increased dramatically with a marginal performance improvement. Further experiments show that adding singular extensions can reasonably make the program play stronger (Section 5.2.3) and therefore it is still worthwhile to consider them in the chess (Chinese Chess) playing programs.

Finally, among all the selective search extensions in our experiments, null move quiescence search with check evasions proves to be the most efficient means of carrying out a selective search. Usually, null move quiescence search is able to search at least a ply deeper than the normal $\alpha\beta$ search and this extra depth is crucial for revealing more tactics. Regarding the implementation issues of null move quiescence search, we see that most of the $\alpha\beta$ enhancements like transposition tables, iterative deepening and extension heuristics like check evasions all successfully carry over. And, when applying iterative deepening to null move quiescence search, finishing iterations on full-width search (nodes before null move quiescence search is applied) first and then iterating on null move quiescence search with the maximum full-width search depth provides a higher search efficiency.

There is however some work remaining to be done. First, the experimental results achieved in this thesis are obtained by a search to a maximum depth 5 (or equivalent to depth 5 for null move quiescence search). While depth 5 is the normal search depth for microcomputers under tournament conditions, it is probably not enough to show that the same conclusions drawn in the thesis will stand for deeper searches. Because of hardware limitations, we were only able to complete the search to depth 5: search to deeper depth is recommended when faster hardware or software (e.g., a better move generator) is available.

Also, although we have concluded null move quiescence search does well in finding tactics, we still have to show how it can play equally well against a program that employs the best full-width search model (normal $\alpha\beta$ search with best selective extensions excluding null move quiescence search). When implementing a program that adopts null move quiescence search, some experiments are necessary to decide what depth combination and means of depth increment should be used if a fixed amount of time is allocated for each move selection. Still other experiments may be required on considering more extension heuristics (like singular extensions and other knowledge extensions) for null move quiescence search.

Bibliography

Akl, S.G. and Newborn, M.M. (1977). The Principle Continuation and the Killer Heuristic. *ACM Annual Conference*, pp. 466-473.

Anantharaman, T.S., Campbell, M.S. and Hsu F-h. (1988). Singular Extensions: Adding Selectivity to Brute-Force Searching. *Proceedings AAAI 1988 Spring Symposium*. Also in *ICCA Journal*, Vol. 11, No. 4, pp. 135-143 and *Artificial Intelligence*, Vol. 43, No. 1, pp. 99-109.

Anantharaman, T.S. (1991). Extensions Heuristics. *ICCA Journal*, Vol. 14, No. 2, pp. 47-65. See also *A Statistical Study of Selective Min-Max Search in Computer Chess*. Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University.

Baudet, G. (1978). *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*. Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University.

Beal, D.F. (1989). Experiments with the Null Move. *Advances in Computer Chess 5* (Ed. D.F. Beal), pp. 65-79. Elsevier Science Publishers B. V., North-Holland.

Beal, D.F. (1991). Private communications with D.F. Beal (Department of Computer Science, Queen Mary and Westfield College, England) during his visit University of Alberta (March-May, 1991).

Bell, A.G. (1984). How to Program a Computer to Play Legal Chess. *Computer Journal*, Vol. 13, No. 2, pp. 208-219.

Berliner, H. (1989). Some Innovations Introduced by Hitech. *Advances in Computer Chess 5* (Ed. D.F. Beal), pp. 283-293. Elsevier Science Publishers B. V., North-Holland.

Bettadapur, P. and Marsland, T.A. (1988). Accuracy and Savings in Depth-limited Capture Search. *Int. J. Man Machine Studies*, pp. 497-502.

Birmingham, J.A. and Kent, P. (1977). Tree-Searching and Tree-Pruning Techniques. *Advances in Computer Chess 1* (Ed. M. Clarke), pp. 89-107. Edinburgh Univ. Press, Edinburgh.

Ebeling, C. (1987). *All the Right Moves, a VLSI Architecture for Chess*. MIT Press.

Cracraft, S.M. (1984). Bitmap Move Generation in Chess. *ICCA Journal*, Vol. 7, No. 3, pp. 146-152.

Feldmann, R., Mysliwietz, P. and Monien, B. (1992). Experiments With a Fully Distributed Chess Program. *Heuristic Programming in Artificial Intelligence 3* (Eds. J. van den Herik and V. Allis), pp. 72-87. Ellis Horwood, London.

Finkel, R.A., Fishburn, J. and Lawless, S.A. (1980). Parallel Alpha-Beta Search on Arachne. *International Conference on Parallel Processing*, pp. 235-243.

Fishburn, J.P. (1984). *Analysis of Speedup in Distributed Algorithms*. UMI Research Press, Ann Arbor. See also *Analysis of Speedup in Distributed Algorithms*. Ph.D. Thesis, Department of Computer Science, University of Wisconsin, 1981.

Frey, P.W. (1983). An Introduction to Computer Chess. *Chess Skill in Man and Machine* (Ed. P.W. Frey), pp. 54-81. Springer-Verlag, New York.

Gillogly, J. (1978). *Performance Analysis of the Technology Chess Program*. Ph.D. Thesis, Department of Computing Science, Carnegie-Mellon University.

Goetsch, G. and Campbell, M. (1990). Experiments with the Null Move Heuristic in Chess. *Computers, Chess, and Cognition* (Eds. T.A. Marsland and J. Schaeffer), pp. 159-168. Springer-Verlag, New York.

Greenblatt, R.D., Eastlake, D.E. and Crocker, S.D. (1967). The Greenblatt Chess Program. *Fall Joint Computer Conference 31*, pp. 801-810.

Hartmann, D. (1989). Notions of Evaluation Functions tested against Grandmaster Games. *Advances in Computer Chess 5* (Ed. D.F. Beal), pp. 91-142. Elsevier Science Publishers B. V., North-Holland.

Huang, S-I. (1986). *On Strategies of Chinese Chess*. Shu Rong Chess Publishers, Chengdu, Shichuan, P.R.C.

Hyatt, R.M. (1984). Using Time Wisely. *ICCA Journal*, Vol. 7, No. 1, pp. 4-9.

Hyatt, R.M., Gower, A.E. and Nelson, H.L. (1990). Cray Blitz. *Computers, Chess, and Cognition* (Eds. T.A. Marsland and J. Schaeffer), pp. 110-130. Springer-Verlag, New York.

Hsu, F-h., Anantharaman, T.S., Campbell, M.S. and Nowatzyk, A. (1990). Deep Thought. *Computers, Chess, and Cognition* (Eds. T.A. Marsland and J. Schaeffer), pp. 55-78. Springer-Verlag, New York.

Jacobs, N.J.D. (1989). Xian, a Chinese Chess Program. *Heuristic Programming in Artificial Intelligence* (Eds. D. Levy and D.F. Beal), pp. 104-112. Ellis Horwood, London.

Knuth, D.E. and Moore R.W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, pp. 293-326.

Lau, H-t. (1985). *Chinese Chess*, Charles E. Tuttle Co. Inc., Japan.

Levy, D.N.L. (1985). *The New Chess Computer Book*. pp. 238-239. Pergamon Press, London.

Kaindl, H. (1982). Quiescence Search in Computer Chess. *SIGART Newsletter*, Vol. 80, pp. 124-131.

Marsland, T.A. and Rushton, P.G. (1973). Mechanisms for Comparing Chess Programs. *ACM Annual Conference*, pp. 202-205.

Marsland, T.A. and Campbell M.S. (1981). A Survey of Enhancement to the Alpha-Beta Algorithms. *ACM Annual Conference*, pp. 109-114.

Marsland, T.A. and Campbell, M.S. (1982). Parallel Search of Strongly Ordered Game Trees. *Computing Surveys*, Vol. 14, No. 4, pp. 533-551.

Marsland, T.A. (1983). Relative Efficiency of Alpha-Beta Implementations. *International Joint Conference on Artificial Intelligence*, Karlsruhe, pp. 763-766.

Marsland, T.A. and Popowich, F. (1985). Parallel Game-Tree Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-7, No. 7, pp. 442-452.

Marsland, T.A. (1985). Evaluation-Function Factors. *ICCA Journal*, Vol. 8, No. 2, pp. 47-57.

Marsland, T.A. (1992). Computer Chess and Search. *Encyclopedia of Artificial Intelligence* (Ed. S. Shapiro), Second Edition, pp. 224-241. John Wiley and Sons, Inc.

Marsland, T.A., Breitkreutz, T. and Sutphen, S. (1991). A Network Multi-Processor for Experiments in Parallelism. *Concurrency: Practice and Experience*, Vol. 3, No. 3, pp. 203-219.

Newborn, M. (1975). *Computer Chess*. Academic Press.

Palay, A.J. (1983). *Searching with Probabilities*. Ph.D. Thesis, Department of Computer Science, Carneige Mellon University. See also book with same title, Pitman, 1985.

Pearl, J. (1980). Asymptotic Properties of Minimax Trees and Game Searching Procedures. *Artificial Intelligence*, Vol. 14, pp. 113-138.

Reinefeld, A. (1983). An Improvement of the Scout Tree Search Algorithm. *ICCA Journal*, Vol. 6, No. 4, pp. 4-14.

Slate, D.J. and Atkin, L.R. (1977). Chess 4.5 - The Northwestern University Chess Program. *Chess Skill in Man and Machine* (Ed. P.W. Frey), pp. 82-118. Springer-Verlag, New York.

Schaeffer, J. (1986). *Experiments in Search and Knowledge*. Ph.D. Thesis, Department of Computer Science, University of Waterloo. See also *Experiments in Search and Knowledge*. Technical Report TR 86-12, Department of Computing Science, University of Alberta.

Schaeffer, J. (1989). The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-11, No. 11, pp. 1203-1212.

Schrüfer, G. (1989). A Strategic Quiescence Search. *ICCA Journal*, Vol. 12, No. 1, pp. 3-9.

Scherzer, T., Scherzer, L. and Tjaden, D. (1990). Learning in Bebe. *Computers, Chess, and Cognition* (Eds. T.A. Marsland and J. Schaeffer), pp. 197-216. Springer-Verlag, New York.

Thompson, K. (1982). Computer Chess Strength. *Advances in Computer Chess 3* (Ed. D.F. Beal), pp. 55-56. Pergamon Press, Oxford.

Tsao, K-m. (1988). *Design and Implementation of an Intelligent Chinese Chess Program*. M.Sc. Thesis, Department of Computer Science and Information Engineering, National Taiwan University.

Tsao, K-m., Li, H. and Hsu, S-c. (1990). Design and Implementation of a Chinese Chess Program. *Heuristic Programming in Artificial Intelligence 2* (Eds. D. Levy and D.F. Beal), pp. 108-118. Ellis Horwood, London.

Tu, J-m. (1985). *Chinese Chess Dictionary*. Shanghai Cultural Press, Shanghai, P.R.C.

Tunstall-Pedoe, W. (1991). Genetic Algorithms Optimizing Evaluation Functions. *ICCA Journal*, Vol. 14, No. 3, pp. 119-128.

Uiterwijk, J. (1991). Private communications with J. Uiterwijk (Department of Computer Science, University of Limburg, Maastricht, The Netherlands) during the 3rd Computer Olympiad, Maastricht, The Netherlands.

Warnock, T. and Wendroff, B. (1988). Search Tables in Computer Chess. *ICCA Journal*, Vol. 11, No. 1, pp. 10-13.

White, J.F. (1990). The Amateurs' Book-Opening Routine. *ICCA Journal*, Vol. 13, No. 1, pp. 22-26.

Wu, R. (1991). Private communications with Ren Wu (Computer and System Science Department, Nankai University, 94 Weijin Road, Tianjin, P.R. China 300071).

Ye, C. and Marsland, T.A. (1992). Selective Extensions in Game-Tree Search. *Heuristic Programming in Artificial Intelligence 3* (Eds. J. van den Herik and L.V. Allis), pp. 112-122. Ellis Horwood, London.

Zhang, T-y. (1981). *The Application of Artificial Intelligence in Computer Chinese Chess*. M.Sc. Thesis, Institute of Electrical Engineering, Taiwan University.

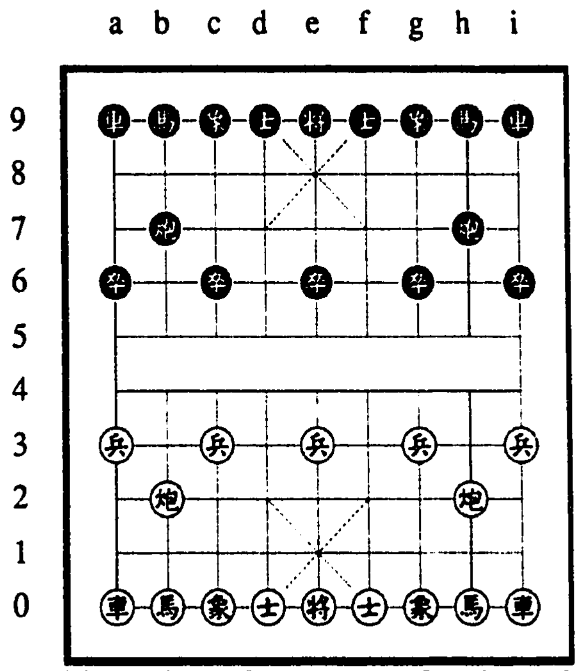
Zielinski, G. (1976). Arrays for Programming Chess. *Kybernetes*, Vol. 5, pp. 91-96.

Zobrist, A.L. (1970). *A New Hashing Method with Applications for Game Playing*. Technical Report TR 88, Computer Science Department, University of Wisconsin, Madison. Also in *ICCA Journal*, Vol. 13, No. 2, pp. 169-173.

Appendix 1
The Basics of Chinese Chess

In this appendix, we give a brief description of the game of Chinese chess, its outlook (board and pieces), rules, relative value of each piece and the notation used to record a game. A more detailed description can be found in a variety of books on Chinese Chess (e.g., H-t. Lau, *Chinese Chess*, Charles E. Tuttle Co. Inc., Japan, 1985).

1.1. Board and Pieces



Initial Chinese Chess Board Configuration

The Chinese chess board consists of 9 vertical lines and 10 horizontal lines. There are two horizontal lines in the middle of the board called *river* which separates the two sides. Each of the two players (called either *Red* or *Black*), starts the game with 16 pieces: one *King*, two each of *Guards*, *Bishops*, *Knights*, *Rooks* and *Cannons*, and five *Pawns*. Unlike chess, the pieces are placed at intersections rather than squares. The notation used in this thesis is algebraic: vertical lines are denoted by the letters *a* to *i*, from left to right; horizontal lines are denoted by the numbers *0* to *9*, from bottom to top. Using this notation, it is easy to present each move with its *from* square and *to* square, i.e., b2e2 means the b2 Cannon moves horizontally to the center file (e-file) in the starting position. The initial board setup is shown in the above figure. It can also be represented in co-ordinates below:

Initial Board Position

Red:

K(e0), R(a0, i0), C(b2, h2), N(b0, h0), B(c0, g0), G(d0, f0), P(a3, c3, e3, g3, i3)

Black:

K(e9), R(a9, i9), C(b7, h7), N(b9, h9), B(c9, g9), G(d9, f9), P(a6, c6, e6, g6, i6)

1.2. Rules of Chinese Chess

In Chinese chess, each piece moves identically as follows:

King, or General

The King moves either vertically or horizontally in a confined area called *Palace* (marked by diagonals on a Chinese Chess board). It is illegal to move the King into a position that makes it oppose the enemy's King in the same line without any intervening pieces. When the King is in check and cannot avoid being captured in the next move, it is said to be *checkmated* and the side who gets checkmated loses the game. A player also loses the game if he has no legal moves left on the board; there is no *stalemate* rule as in chess.

Rook, or Warrior

The Rook has exactly the same properties as the Rook in chess.

Cannon, or Gunner

The Cannon is a special piece in Chinese chess. It moves similarly to a Rook, except when it captures an enemy's piece. When capturing, and only when capturing, a Cannon jumps over one piece, of either side, which must be between the cannon and the piece captured. The piece captured must be in the same line as the Cannon and the jumped piece.

Knight, or Horse

The Knight's move resembles the Knight in chess, with an important difference that a Knight cannot jump over pieces (it can be blocked). To determine whether a Knight is blocked or not, we can separate the Knight's move into two steps: first it moves one unit parallel to a board edge, and then it diagonally moves across one square; the intermediate point must be vacant for the move to be legal.

Pawn, or Soldier

The Pawn always moves one square at a time. While still on its own side of the river, it can only move directly forward. Once it crosses the river, it gains extra moves and it can move sideways as well. The Pawn doesn't promote as in chess so placing a Pawn on the opponent's back rank offers no special advantage.

All the above mentioned pieces except the King are called *attacking pieces* because they can cross the river. The following two pieces, Bishop and Guard, can only remain in their own territories and thus are called *defensive pieces.*

Bishop, or Elephant

A Bishop moves from one corner to the opposite corner of a 2X2 rectangle. It cannot jump over a piece, so if the center of the rectangle is occupied, the move cannot be made. There are only seven points that a Bishop can go to.

Guard, or Assistant

The Guard is another defensive piece. It moves diagonally across one square, and must stay inside the Palace. There are only five points that a Guard can reach.

Chinese chess has no artificial exceptions to the general move rules necessary in chess (pawn promotion, castling, stalemate and *en passant* captures). It does however have one special rule: it is illegal to play a move that repeats the position for the third time if the move is a threat (either a check, threat to mate, or threat to win material by forced moves). There are however some exceptions to this repetition rule. Besides, different Chinese Chess associations in the world have their own definitions of this special rule and it is hard to say which one is most correct. However, it is usually safe to count the number of threat moves between two repetitive positions for both sides and determine the legalness of the last threat move; it is illegal to make a threat move for a side if this makes his total number of threat moves exceed the opponent's. Please refer to Section 2.5 for a more detailed discussion on the repetition rule in the thesis. A game of Chinese Chess ends by a checkmate (*win* or *loss* depending on who gets checkmated), a legal third-time-repetition (*draw*), or if no capture is made in a consecutive 60 moves (*draw*).

1.3. Piece Rankings

It is difficult to give a precise value to each piece in a particular position. However, they can be ranked roughly as follows:

Piece	Ranking	Value used in Abyss
King	∞	7000
Rook	9	1800
Cannon	4.5	900 in opening, 800 in endgame
Knight	4	800 in opening, 900 in endgame
Bishop	2	300
Guard	2	300
Pawn	1	100

The values given here are only static values. Some positional factors like *head-cannon*, *side-cannon*, *passed pawns* and *three-pieces-beside (king)* have to be considered in the evaluation function to make the board assessment more accurate.

Appendix 2

The Fifty Middle Game Positions Used for Experiments

In this appendix, we present the 50 middle game positions used for the experiments in the thesis and their solutions. These positions are extracted from a standard work by Tu Jing-ming [*Chinese Chess Dictionary*, pp. 105-124, Shanghai Cultural Press, Shanghai, P.R.C., 1985]. The Chinese Chess board is represented by all the existing pieces and their corresponding squares (in brackets). The piece names used are K for King, R for Rook, N for Knight, C for Cannon, B for Bishop, G for Guard and P for Pawn. Each square is defined by a file number (vertically from *a* to *h*) and a rank number (horizontally from *0* to *9*).

Position #1
Red: K(e0), R(d9, b0), C(a9, a2), B(e2, g0), G(f0), P(a4, c4, i3)
Black: K(e8), R(b6, c3), C(b4), N(g5), B(g9, e7), P(i6, e5)
Solution: 1. b0d0 b6b8 2. a9a8 *black has to sacrifice his rook or he gets mated.*

Position #2
Red: K(e0), R(c8), C(b9, c2), N(g2), B(a2, e2), G(e1, f0), P(a3, e3, g3, i3)
Black: K(e9), R(d4), C(e8, i7), N(g7), B(c9, e7), G(d9, f9), P(a6, e6, i6, g5)
Solution: 1. a2c4 e8d8 2. c8c9 *with a winning attack.*

Position #3
Red: K(e0), R(b0), C(d1), N(f6, g4), B(e2, g0), G(e1, f0), P(c4, a3, i3)
Black: K(e9), R(b5), C(b2), N(f5, d3), B(g9, e7), G(d9, e8), P(a6, c6, i6)
Solution: 1. e1d2 d3f4 2. f6g8 e9f9 3. d1f1 b2b3 4. d2e1 *red should win a piece.*

Position #4
Red: K(e0), R(d8), C(h6, b2), N(b8), B(c0, g0), G(d0, f0), P(g4, a3, i3)
Black: K(e9), R(b7), C(d9), N(i7, f4), B(c9, g9), G(f9, e8), P(a6, e6, g6, i6, d3)
Solution: 1. b2e2 f4e2 2. h6e6 e8f7 3. c0e2 *and 4. d8d9, red wins a piece.*

Position #5
Red: K(e0), R(d8), C(e2, d1), N(a2, g2), B(c0, g0), G(d0, f0), P(a4, e4, g4, c3)
Black: K(e9), R(a6), C(b7, i5), N(g7, b5), B(c9, e7), G(d9, e8), P(e6, g6, i6, c5)
Solution: 1. a4a5 a6a5 2. d1a1 b5a3 3. e2e3 b7a7 4. d8a8 *red should win a piece.*

Position #6
Red: K(e0), R(d4), C(g4, g2), N(c2), B(e2, g0), G(d0, f0), P(a3, i3)
Black: K(e9), R(b5), C(c7, i7), N(d5), B(e7, g5), G(f9, e8), P(a6, i6)
Solution: 1. g4h4 i7h7 2. h4h5 b5c5 3. c2b4 c7d7 4. d4d5 *red wins a piece.*

Position #7
Red: K(e0), R(d6, b2), C(e4, c2), N(g2), B(g4, e2), G(d0, f0), P(c6, a4, i3)
Black: K(e9), R(c3, f1), C(g9, g3), N(g7), B(c9, e7), G(d9, e8), P(a6, g6, i6)
Solution: 1. d6d3 c3c4 2. d3e3 c4c6 3. e3g3 *red wins a piece.*

Position #8
Red: K(e0), R(d8, c0), C(c2, e1), N(e2, g2), B(a2, g0), G(d0, f0), P(a3, e3, g3, i3)
Black: K(e9), R(b4, c4), C(e7, b3), N(c7, f5), B(c9, g9), G(d9, f9), P(a6, e6, i6, c5, g5)
Solution: 1. e1b1 c4d4 2. d8d4 b4d4 3. c2c7 *red wins a piece.*

Position #9
Red: K(e0), R(h4, b0), C(b2, e2), N(b4, g2), B(c0, g0), G(e1, f0), P(a4, c3, i3)
Black: K(e9), R(a9, g3), C(e7, e3), N(c7, g7), B(c9, g9), G(d9, f9), P(a6, c6, e6, g6,

i6)
Solution: 1. b4d5 c7a8 2. g2e3 e7e3 3. b2b3 *pin, red should win a piece.*

Position #10
Red: K(e0), R(f6), C(e2), N(e3, c2), B(c0), G(e1, d0), P(a3, c3)
Black: K(e9), R(g5), C(i3), N(c7, e5), B(g9, e7), G(d9), P(a6, i6, c5)
Solution: 1. e2e5 g5e5 2. e3g4 *fork, red should win a piece.*

Position #11
Red: K(e0), R(f5, f1), C(b4, e2), N(h3, c2), B(c0, g0), G(d0, f0), P(c4, a3, e3, i3)
Black: K(e9), R(h8, g4), C(b5, h5), N(c7, d4), B(g9, e7), G(d9, e8), P(a6, e6, i6, c5)
Solution: 1. c2d4 b5f5 2. d4f5 g4c4 *or g4g6 3. f5d6 e8d7 4. c4c5 red has more pawns***3. f5d6 c4d4 4. d6c8 d4d8 5. b4b9** *red wins.*

Position #12
Red: K(e0), R(f5, f1), C(c7, a2), N(c2), B(e2), G(e1), P(c4, a3, e3)
Black: K(e9), R(h9, g0), C(i0), N(f0), B(c9, e7), G(d9, f9), P(a6, c6, e6, i6)
Solution: 1. f5f9 h9f9 2. f1f9 e9f9 3. e2g0 *red maintains material advantage.*

Position #13
Red: K(e0), R(b0), C(g1), N(e6), B(e2, g0), G(d0, f0), P(i4, a3, e3)
Black: K(e9), R(e5), N(d5, b4), B(g9, e7), G(d9, f9), P(a5, e4)
Solution: 1. e6c5 e7c5 2. b0b4 e4e3 3. g1e1 *equal.*

Position #14
Red: K(e0), R(h9), C(g0), N(f2), B(e2, c0), G(e1, d0), P(f5, a3, i3)
Black: K(e9), R(g6), C(f7), N(b5), B(c9, e7), G(d9, f9), P(a6, e6, i6)
Solution: 1. f2g4 e7g5 2. g4i5 *double attack, red wins the exchange.*

Position #15
Red: K(e0), C(g6, h5), N(g8, a2), B(e2, c0), G(e1, d0), P(e4, g3)
Black: K(e8), C(i3, b1), N(b7, i6), B(g9, e7), G(d9, f9), P(e6, g5)
Solution: 1. g6g7 b7c5 2. g7i7 e8d8 3. i7i3 *red wins a piece.*

Position #16
Red: K(e0), R(h4, a0), C(b4, e2), N(g2), B(c0, g0), G(d0, f0), P(g4, a3, e3, i3)
Black: K(e9), R(a9, h9), C(h7, b5), N(c7), B(c9, g9), G(d9, f9), P(a6, e6, i6, c2)
Solution: 1. a0a2 c2b2 2. b4e4 c9e7 3. a2b2 *with initiative.*

Position #17
Red: K(e0), R(h6), C(e5, d2), N(d3, g2), B(e2, c0), G(e1, d0), P(e3, i3)
Black: K(e9), R(f5), C(f7, c3), N(a7, f6), B(g9, e7), G(f9, e8), P(c6, a5)
Solution: 1. g2h4 f5f1 2. d2d1 f1f4 e3e4 *red should win a piece.*

Position #18
Red: K(e0), R(e4, i0), C(h2), N(a4, c2), B(e2, c0), G(e1, d0), P(g4, a3, i3)
Black: K(e9), R(i9, b5), C(e5), N(h9, c7), B(c9, g9), G(d9, f9), P(a6, g6, i6)
Solution: 1. c2d4 b5d5 2. a4b6 d5d6 3. b6c8 d6d8 4. e3e4 f9e8 5. h2h8 *red wins.*

Position #19
Red: K(e0), R(c0), C(e5), N(e3, i2), B(e2, g0), G(f2, d0), P(a3, i3)
Black: K(e9), R(b5), C(e4, h2), N(f5), B(g9, a7), G(d9, f9), P(a6, i6)
Solution: 1. f2e1 b5e5 2. e3g4 *red wins back the piece, equal.*

Position #20
Red: K(e0), R(d6, h0), C(e2, c1), N(e5, a2), B(i2, c0), G(d0, f0), P(g4, e3, i3)
Black: K(e9), R(h6, c4), C(d7, h3), N(g7, d5), B(c9, e7), G(d9, e8), P(a6, e6, i6, c5)
Solution: 1. g4g5 e7g5 2. d6d5 e6e5 3. c1c3 h3h4 4. e2c2 h4e4 5. e3e4 c4e4 6. d0e1 *still threatening mate therefore red wins material.*

Position #21
Red: K(e0), C(d2, f2), N(f5, g2), B(i2, c0), G(e1, d0), P(a4, e3)
Black: K(e9), C(g7, g3), N(b5, d5), B(g9, e7), G(f9, e8), P(a6, g6, i5)
Solution: 1. i2g4 g6g5 2. f5g7 d5e3 3. f2e2 *red wins a piece.*

Position #22
Red: K(e0), R(c6), C(d2, e2), N(c2, g2), B(c0, g0), G(e1, d0), P(g4, a3, e3, i3)
Black: K(e9), R(f8), C(b7, e7), N(g7, d5), B(g9, a7), G(d9, f9), P(a6, e6, g6, i6)
Solution: 1. c2d4 d5f6 2. c6c7 f8b8 3. d2b2 g7e8 4. c7e7 b7c7 5. e7f7 b8b2 6. e2e6 e8c9 7. f7c7 *red wins a piece.*

Position #23
Red: K(e0), R(e5, h3), C(e2), N(a2, f0), B(c0, g0), G(e1, d0), P(a4, g4, i3)
Black: K(e9), R(h9, b3), C(h4, g3), N(g7), B(g9, e7), G(f9, e8), P(a6, g6, i6)
Solution: 1. e2e3 h4a4 *or g3g1 2. e3e7 and 3. h3b3* **2. h3g3** *red wins a piece.*

Position #24
Red: K(e0), R(b9, h0), C(h6, d2), N(g2, a1), B(e2, g0), G(e1, d0), P(g4, a3, i3)
Black: K(e9), R(c8, h8), C(e7, b4), N(g7, d5), B(g9), G(d9, e8), P(a6, e6, g6, i6)
Solution: 1. h6e6 *threatening mate therefore wins material.*

Position #25
Red: K(e0), R(f6), C(e4), N(a2, g2), B(e2, c0), G(e1, d0), P(a4, c3, e3, i3)
Black: K(e9), R(b2), C(g3), N(h8, a7), B(c9, e7), G(f9, e8), P(a6, i6, c5)
Solution: 1. f6f8 h8g6 2. f8f3 g3g5 3. f3f5 e9d9 4. f5d5 d9e9 5. f5g5 *red wins a piece.*

Position #26
Red: K(e0), R(h4, a0), C(b4, d2), N(f5, d4), B(e2, g0), G(d0, f0), P(c4, a3, e3)
Black: K(e9), R(a9, h6), C(g5, i3), N(c7, h5), B(g9, e7), G(f9, e8), P(a6, c6, e5)
Solution: 1. b4b1 i3i8 2. b1h1 i8h8 3. f5g3 *red should win a piece.*

Position #27
Red: K(e0), R(g6), C(c4, d2), N(h6, g0), B(e2, c0), G(d0, f0), P(d6, a3, e3, i3)
Black: K(e9), R(h5), C(c8, b6), N(e8, g7), B(c9, e7), G(d9, f9), P(a6, e6, i6, g3)
Solution: 1. d6c6 c8a8 2. d2d8 e8g9 3. d8h8 h5g5 4. g6g5 e7g5 5. c4c9 d9e8 6. h8a8 *red wins a piece.*

Position #28
Red: K(e0), R(f3), C(g2), N(c2, h0), B(e2, c0), G(e1, d0), P(c4, e4, g4, i4, a3)
Black: K(e9), R(c3), C(d3), N(c7, i7), B(g9, e7), G(f9, e8), P(a6, c6, e6, g6, i6)
Solution: 1. c2b0 c3a3 2. b0a2 *threatening 3. g2g3, red should win a piece.*

Position #29
Red: K(e0), R(f6), C(c6, e2), B(i2, c0), G(e1, f0), P(a3, i3)
Black: K(e9), R(h3), C(g7), N(h4), B(c9, e7), G(d9, e8), P(a6, i6, c4, g3)
Solution: 1. f6f7 g7g6 2. f7e7 e8f8 3. e7e4 *with counterplay.*

Position #30
Red: K(e0), R(c7, b5), C(e4, a2), N(c2), B(g4, c0), G(e1, d0), P(c4, a3, e3, i3)
Black: K(e9), R(h9, a8), C(g7, i0), N(f4), B(c9, g9), G(d9, f9), P(a6, c6, i6, f2)
Solution: 1. b5e5 a8e8 2. e5h5 e8h8 3. c7e7 d9e8 4. e7g7 c9e7 5. h5h8 h9h8 6. e1f2 *red wins a piece.*

Position #31
Red: K(e1), R(d2), C(b8, c8), N(c6), G(f0), P(a3, e3, g3, i3)
Black: K(e9), R(c0), C(i7), N(c7, g1), B(c9, e7), G(f9), P(a6, e6, i6, g5, f2)
Solution: 1. e1d1 c0c1 2. d1d0 g1e2 3. f0e1 f9e8 4. b8b9 e8d9 5. d2d9 *mate.*

Position #32
Red: K(e0), R(h1), C(g6, c2), N(b4, g2), B(e2, g0), G(d0, f0), P(a3, e3, i3)
Black: K(e9), R(h5), C(i5, h3), N(a7, g3), B(c9, e7), G(d9, e8), P(a6, e5)
Solution: 1. e2g4 e7g5 2. b4c6 e8f9 3. c2c9 d9e8 4. c6a7 *winning.*

Position #33
Red: K(e0), R(b6, h4), C(g6, e2), B(c0, g0), G(d0, f0), P(a3, i3)
Black: K(e9), R(d9, c7), C(b7, g2), B(g9, e7), G(f9, e8), P(a6, i6)
Solution: 1. f0e1 b7b9 2. g6a6 b9a9 3. b6g6 *red should win a piece.*

Position #34
Red: K(e0). R(h3), C(g4), N(f6, a2), B(c0, g0), G(d0, f0), P(e5, c3, i3)
Black: K(e9), R(d1), C(e3), N(f5, a3), B(g9, e7), G(f9, e8), P(i6, a5, c5)
Solution: 1. a2b0 d1b1 2. h3h2 b1d1 3. h2f2 *and 4. f2f3, red should win a piece.*

Position #35
Red: K(e0), R(g6, d3), C(b2), N(c2, g2), B(e2, g0), G(d0, f0), P(c4, e4, a3, i3)
Black: K(e9), R(h8, f7), C(g7, b6), N(c7), B(c9, e7), G(d9, f9), P(a6, c6, e6, i6, g5)
Solution: 1. d3b3 *eliminating threat b6b8 and b8g8 by black* **b6b2 2. b3b2** *equal.*

Position #36
Red: K(e0), R(d6), C(b2, e2), N(e3, g2), B(c0, g0), G(d0, f0), P(c4, a3, g3, i3)
Black: K(e9), R(h9), C(a7, h7), N(c7, e6), B(g9, e7), G(d9, f9), P(a6, i6, c5, e5, g5)
Solution: 1. b2b6 e6g7 *or h7h6 2. b6e6* **2. e2e5 f9e8 3. c4c5** *with initiative.*

Position #37
Red: K(e0), R(h4, b0), C(b8, g6), N(d4, f4), B(e2, c0), G(e1, f0), P(a3, e3, i3)
Black: K(e9), R(i9, a8), C(f7, g7), N(c7, e7), B(g9, c5), G(d9, f9), P(a6, i6, e5, g5)
Solution: 1. f4e6 c7e6 2. d4e6 f7f6 3. e6c5 e7g6 4. c5d7 e9e8 5. d7f6 g7f7 6. h4h6 g6f8 7. b8f8 *since e8f8? 8. h6h8 mate, therefore red wins a piece.*

Position #38
Red: K(e0), R(f6, d3), N(c6, d5), B(e2, g0), G(e1, d0), P(a4, c4, e3)
Black: K(e9), R(i3, c1), C(f7), N(c7, f5), B(g9, e7), G(f9, e8), P(a6, e6, i5)
Solution: 1. d5b6 e8d7 2. f6f7 f5h4 3. b6c8 e9e8 4. f7h7 *threatening mate therefore red wins a piece.*

Position #39
Red: K(f0), R(b7, f6), C(e2, g2), B(g4, c0), P(a3, i3)
Black: K(e9), R(d9, e3), C(e1, d0), B(e7, i7), G(f9, e8), P(a6, i6, c5)
Solution: 1. f6f9 e8f9 2. b7e7 f9e8 3. e7e8 *mate.*

Position #40
Red: K(e0), R(e5), C(e4, d2), N(h6), B(e2, c0), G(e1, d0), P(h5, e3)
Black: K(e9), R(f6), C(c9, h7), N(c3), B(g9, e7), G(f9, e8), P(i6, a3)
Solution: 1. e4e7 g9e7 2. e5e7 h7h5 3. h6g8 f6f8 4. d2d8 c3d5 5. e7e5 c9c5 6. e5d5 e8d7 7. e5c5 *threatening 8. c5c9 therefore red wins a piece.*

Position #41
Red: K(e0), R(f8, h1), C(g4), N(g6), B(e2, c0), G(d0, f0), P(a5, c4, i3)
Black: K(e9), R(i9, c3), C(d8), N(c7), B(g9, e7), G(f9, e8), P(c6, e6, i6)
Solution: 1. g6e7 d8f8 2. e7g8 g9i7 3. g8i9 *red wins a bishop.*

Position #42
Red: K(e0), R(f6), C(e7, d4), N(c2), B(e2, c0), G(d0, f0), P(c4, a3, e3, i3)
Black: K(f9), R(i9), C(f8, f7), N(c7, g7), B(g5), G(d9, e8), P(a6, c6, e6, i6, g3)
Solution: 1. e7g7 f8f6 2. g7c7 i9i8 3. c7c9 f9f8 4. d4d8 *red should win a piece.*

Position #43
Red: K(e0), R(d8, b3), C(b5), N(f4), B(g0), G(f0), P(c4, g4, a3, i3)

Black: K(e9), R(b7, g1), C(d9, i0), B(g9, e7), G(e8), P(a6, c6, g6, i6)
Solution: 1. b5e5 b7b3 2. d8e8 e9f9 3. e8e9 f9f8 4. f4g6 f8f7 5. e9f9 *mate.*

Position #44
Red: K(f1), R(f8, b6), N(f3), G(e1, d0), P(a3, c3)
Black: K(e9), R(h5), C(c0), N(i6, c1), B(g9, e7), G(f9, e8), P(a6)
Solution: 1. f3e5 e7c9 2. e5d7 e9d9 3. b6d6 e8d7 4. d6d7 d9e9 5. f8f9 e9e8 6. d7d9 *red mates once opponent stops repetition checks.*

Position #45
Red: K(e0), R(c6), C(e5, c3), N(c5), B(c0, g0), G(d0, f0), P(c4, a3, i3)
Black: K(e9), R(f2), C(d7, i7), N(h1), B(g9, e7), G(f9, e8), P(a6, i6)
Solution: 1. c6c9 d7d9 2. c3d3 f2f0 3. e0e1 f0d0 4. c5d7 i7d7 5. c9d9 *mate.*

Position #46
Red: K(d0), R(d1), C(e6, e2), N(a2), B(c0, g0), G(e1, f0), P(a4, e4, c3, i3)
Black: K(e9), R(g4), C(a9, e7), N(g7), B(c9, g9), G(f9, e8), P(a6, i6, c5)
Solution: 1. e2b2 a9b9 2. a2b4 b9a9 3. b4c6 a9b9 4. c6a7 *red should win a piece.*

Position #47
Red: K(e0), R(h4), C(a2, g1), N(g6), B(c0, g0), G(d0, f0), P(c4, e3, i3)
Black: K(e9), R(h9), C(a3), N(c7, i7), B(g9, e7), G(f9, e8), P(c6, a5, e5, i5)
Solution: 1. g6i7 h9h4 2. i7g8 e9d9 3. g1d1 h4h1 4. d0e1 *and a2d2 mate.*

Position #48
Red: K(e0), R(i1), C(e5), N(b5, d5), B(e2, g0), G(d0, f0), P(c4, a3)
Black: K(e9), R(i4), C(b7, i3), N(h9), B(e7, g5), G(f9, e8), P(a6, i6, f3)
Solution: 1. d5b6 b7b5 *or e9d9 2. i1d1 b7d7 3. b6d7 e8d7 4. d1d7 mate* **2. b6d7 e9d9 3. e5d5** *mate.*

Position #49
Red: K(e0), R(i6, b0), C(e5, d3), N(d5, c2), B(e2, c0), G(e1, f0), P(c4, a3, e3)
Black: K(e9), R(b7, h0), C(f7, g7), N(i7, g5), B(a7, e7), G(f9, e8), P(a6, f2)
Solution: 1. b0b7 f7b7 2. d5e7 *since g5i6 3. e7c8 e9d9 4. e5d5 mate, so red at least wins a bishop.*

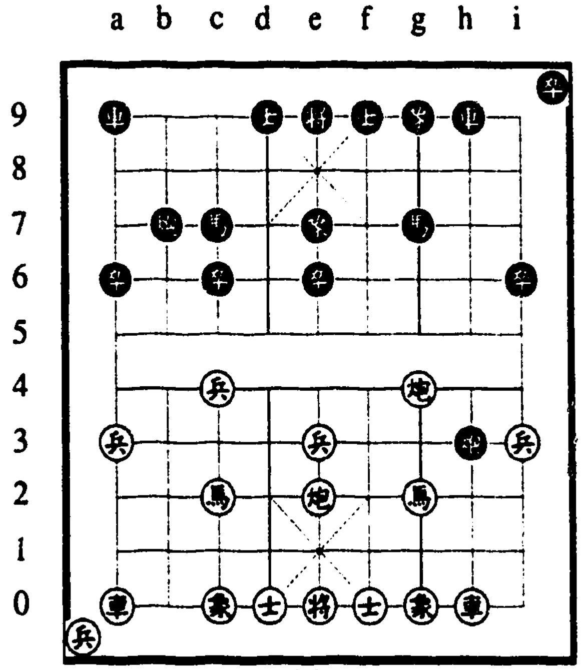
Position #50
Red: K(e0), R(f2), C(e6, c1), N(c5, g2), B(c0, g0), G(e1, d0), P(a3, e3, i3)
Black: K(e9), R(h3), C(e7, g7), N(c7, i7), B(g9, a7), G(d9, e8), P(a6, g6, i6, c4)
Solution: 1. c5d7 g7d7 2. e0f0 *mating.*

Appendix 3

The Twelve Openings Used for Experiments

Here we present 12 openings that are used to play 48 games between two versions of *Abyss* with different extension heuristics. These randomly chosen openings are mostly 5 to 8 moves away from the initial position and are common in Chinese Chess. For one opening position, which is reached by using an opening book limited with only those moves in bold font, we play 2 games each between two different programs at a different time control, so we have altogether 4 games played by 4 different programs. The following names will be used for the programs being tested: *Abyss+SIN'30—Abyss* with singular extensions at 30 seconds per move, *Abyss-SIN'30—Abyss* without singular extensions at 30 seconds per move, *Abyss+SIN'100—Abyss* with singular extensions at 100 seconds per move and *Abyss-SIN'100—Abyss* without singular extensions at 100 seconds per move. Notice for all programs, the extension heuristics of check evasions and recaptures are always included. Apart from the opening moves, the non-drawn game scores are also included in this appendix.

Opening #1:



Position #1

Abyss+SIN'30—Abyss-SIN'30: **1. h2e2 h9g7 2. h0g2 b9c7 3. i0h0 i9h9 4. c3c4 g6g5 5. b0c2 h7h3 6. b2b4 c9e7 7. g3g4 g5g4 8. b4g4** (Position #1) a9b9 9. a0b0 b7b5 10. f0e1 g7f5 11. g2f4 h9h4 12. g4g2 h4f4 13. h0h3 f4g4 14. h3h2 f5h4 15. e1f2 h4g2 16. g0i2 g4c4 17. e2g2 b5e5 18. e0f0 e5f5 19. f0e0 f5e5 20. e0f0 e5f5 21. f0e0 b9b0 22. c2b0 c4c0 23. b0c2 f5e5 24. e0f0 e5f5 25. f0e0 f5e5 26. e0f0 f9e8 27. h2h6 e5f5 28. f0e0 f5e5 29. e0f0 e5f5 30. f0e0 c0c1 31. h6i6 f5e5 32. d0e1 c6c5 33. e3e4 c1c0 34. c2d0 e5d5 35. g2g7 e7c9 36. i6i9 d5g5 37. g7h7 e8f9 38. i9g9 g5f5 39. h7h9 f5f7 40. h9f9 c9e7 41. g9i9 d9e8 42. f9f8 e8f9 43. i9i7 f9e8 44. i7i9 e8f9 45. i3i4 c5c4 46. i4i5 c4d4 47. i9i7 f9e8 48. i7i9 e8f9 49. i9i7 f9e8 50. i7i9 e8f9 51. f8b8 d4e4 52. b8b7 e4e3 53. i9i7 f7f3 54. b7e7 c7d5 55. e7a7 f3h3 56. i7d7 c0c5 57. d7d6 d5f4 58. i5i6 c5i5 59. i6h6 i5i2 60.

d0b1 i2i0 61. e1f0 h3g3 62. h6g6 g3g0 63. f0e1 g0g5 64. e1f0 g5g0 65. f0e1 g0g3 66. e1f0 i0i1 67. b1d2 e3e2 68. d2c4 i1c1 69. c4b2 c1c2 70. d6b6 g3e3 71. e0d0 c2d2 72. b2d1 e2f2 73. b6b9 e9e8 74. b9f9 d2d4 75. f9f4 d4f4 76. f0e1 e8d8 77. e1d2 e3d3 78. g6f6 f2e2 79. d0e0 d3d1 80. f6e6 f4e4 81. e6f6 e2d2 82. e0f0 e4f4 83. f0e0 f4f6 84. a7i7 f6e6 85. e0f0 d8e8 86. i7i1 d1h1 87. i1i5 (0 - 1)

Abyss-SIN'30—Abyss+SIN'30: **1. h2e2 h9g7 2. h0g2 b9c7 3. i0h0 i9h9 4. c3c4 g6g5 5. b0c2 h7h3 6. b2b4 c9e7 7. g3g4 g5g4 8. b4g4** a9b9 9. a0b0 b7b1 10. c2b4 b1a1 11. h0h1 c6c5 12. e2b2 b9b4 13. g4b4 c5c4 14. b4b6 h3g3 15. h1a1 g3g0 16. f0e1 g0i0 17. e0f0 h9h0 18. f0f1 h0h1 19. f1f0 h1h0 20. f0f1 h0h1 21. f1f0 h1g1 22. b2c1 g1g0 23. f0f1 i0d0 24. b0b2 g0g2 25. b6i6 g7i6 26. a1b1 g2g1 27. f1f2 c7d5 28. e1d0 g1g3 29. b1d1 g3f3 30. f2e2 f3e3 31. e2f2 e3f3 32. f2e2 d5f4 33. b2a2 f3i3 34. e2f2 i3f3 35. f2e2 f3a3 36. e2e1 a3e3 37. c2e2 c4d4 38. d1d4 f4g2 39. e1d1 d9e8 40. d4d6 i6g5 41. a2a6 e3d3 42. d6d3 g2f0 43. e2e0 e8d7 (1 - 0)

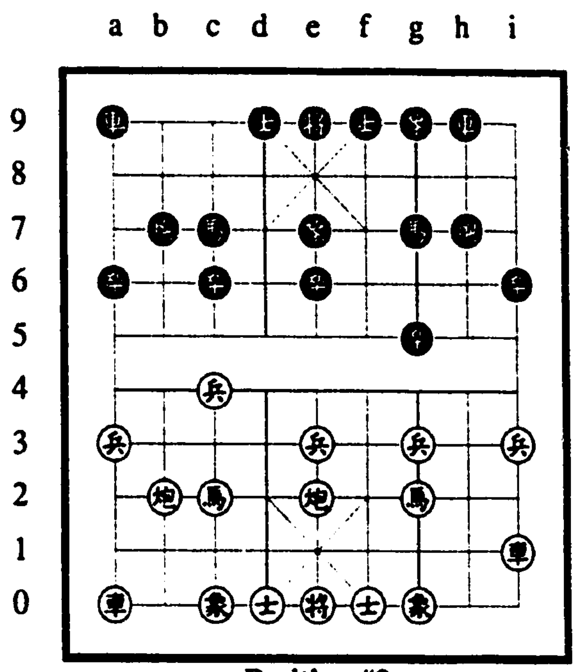
Abyss+SIN'100—Abyss-SIN'100: **1. h2e2 h9g7 2. h0g2 b9c7 3. i0h0 i9h9 4. c3c4 g6g5 5. b0c2 h7h3 6. b2b4 c9e7 7. g3g4 g5g4 8. b4g4** a9b9 9. g4g6 b7b2 10. g6c6 b9b6 11. g2f4 b6c6 12. e2b2 c6c4 13. h0h3 h9h3 14. f4h3 c4c2 15. b2b0 c2c3 16. h3f2 c3e3 17. f0e1 e3i3 18. f2g4 i3b3 19. c0e2 c7d5 20. b0c0 d9e8 21. a0a1 g7f5 22. a3a4 d5f4 23. c0c6 f5h4 24. a1d1 i6i5 25. c6c4 b3a3 26. d1d4 f4d3 27. d4d6 h4g2 28. c4c2 g2f4 29. g4h6 e8d7 30. d6a6 d3c1 31. e0f0 a3d3 32. a6a9 e9e8 33. a9a8 e8e9 34. a8a9 e9e8 35. a9f9 d3f3 36. e1f2 c1a2 37. a4a5 a2b4 38. c2d2 b4d3 39. h6f5 f3g3 40. a5b5 e6e5 41. d2d7 f4g2 42. f0f1 g3h3 43. f5g7 e8d8 44. f9f8 d8d7 45. f8h8 (0 - 1)

Abyss-SIN'100—Abyss+SIN'100: **1. h2e2 h9g7 2. h0g2 b9c7 3. i0h0 i9h9 4. c3c4 g6g5 5. b0c2 h7h3 6. b2b4 c9e7 7. g3g4 g5g4 8. b4g4** a9b9 9. g4g6 b7a7 10. a0a1 a6a5 11. a1f1 d9e8 12. f0e1 b9c9 13. g2f4 h3h4 14. c0a2 h4h2 15. g6c6 c7a6 16. c4c5 a6c5 17. c6b6 c5d7 18. b6d6 h2c2 19. h0h9 g7h9 20. d6i6 c2b2 21. a2c0 c9c0 22. e1d2 b2b0 23. f1b1 b0d0 24. e0e1 a7a3 25. b1b9 d7c9 26. b9b3 a3a0 27. e2e6 c0c1 28. e1e2 c1c4 29. f4h5 a0g0 30. b3b5 a5a4 31. b5g5 c4c3 32. e3e4 c3e3 33. e6e3 e7g5 34. i6i4 a4a3 35. e4e5 d0a0 36. e5f5 g5e7 37. i4i9 e9d9 38. e3h3 a0a2 39. e2e1 g0h0 40. h3h9 h0h9 41. h5g7 h9h1 42. g7e8 c9d7 43. e8g7 d7c5 44. g7f9 c5b3 45. e1d1 a2c2 46. i3i4 c2c7 47. f5f6 c7d7 48. d1e1 b3d2 49. i4i5 a3b3 50. f6e6 d2f3 51. e1e0 h1f1 52. f9g7 d9d8 53. g7h5 b3c3 54. h5g3 f1a1 55. e6d6 f3g1 56. e0e1 d7b7 57. e1d1 c3d3 58. d1d0 b7b0 59. g3e4 b0b4 60. d0d1 (0 - 1)

Opening #2:

Abyss+SIN'30—Abyss-SIN'30: **1. h2e2 h9g7 2. h0g2 i9h9 3. c3c4 g6g5 4. b0c2 b9c7 5. i0i1 c9e7** (Position #2) 6. c2d4 h7h6 7. b2c2 b7b8 8. d4c6 a9c9 9. a0b0 b8a8 10. c6d4 a8a3 11. b0b3 a3a4 12. b3b4 a6a5 13. d4e6 g7e6 14. e2e6 c7e6 15. c2c9 e7c9 16. f0e1 h6g6 17. b4b5 e6f4 18. g2f0 a4a1 19. e1d2 a1a0 20. i1a1 g6e6 21. d2e1 f4h3 22. b5f5 a0b0 23. a1a5 h3g1 24. f5f1 b0b1 25. f1g1 b1g1 26. a5e5 e6e8 27. e5e6 c9e7 28. e6i6 e8e3 29. c0e2 e3f3 30. i6d6 f3i3 31. c4c5 i3i0 32. c5c6 h9h3 33. d6d3 f9e8 34. c6c7 i0i3 35. c7c8 i3g3 36. d3d6 g3f3 37. d6d4 g1f1 38. e1f2 h3g3 39. c8d8 g3g0 40. d0e1 g5g4 41. e0d0 g0g1 42. d8d9 e8d9 43. d4d9 e9e8 44. d9d8 e8e9 45. d8d9 e9e8 46. d9d8 e8e9 47. f0d1 f1d1 48. d8d1 f3f9 49. d1d8 g4g3 50. d8d6 g1f1 51. d6f6 g3g2 52. f6f7 f1h1 53. f7f8 g2f2 54. e1f2 h1f1 55. d0e0 e7c9 56. f8f5 c9e7 57. f5f3 f9f2 58. e2c4 e9e8 59. c4a2 e8e9 60. a2c0 e9e8 61. f3g3 f1c1 62. g3g8 f2f8 63. c0e2 c1c9 64. e0f0 c9f9 65. f0f1 e8d8 66. f1e1 f8e8 67. g8g6 e7g5 68. e2g4 g5e7 69. g4e2 e8e9 70. g6c6 e7g5 71. c6e6 d8d9 72. e6e4 f9f2 73. e4e5 f2e2 74. e1f1 e2e5 75. f1f2 (0 - 1)

Abyss-SIN'30—Abyss+SIN'30: **1. h2e2 h9g7 2. h0g2 i9h9 3. c3c4 g6g5 4. b0c2 b9c7 5. i0i1 c9e7** 6. c2d4 h7h6 7. b2c2 b7b8 8. d4c6 a9c9 9. a0b0 b8a8 10. c6d4 a8a3 11. b0b3 a3a4 12. c4c5 a4c4 13. b3c3 c4c2 14. c3c2 e7c5 15. c2c5 g9e7 16. c5c4 h6h4 17. d4c6 g7f5 18. i1f1 h4h5 19. c4h4 c7e8 20. c6b8 c9c8 21. e2e6 f5g3 22. h4b4 h9h6 23. b4f4 g3f5 24. f4h4 h6e6 25. h4h5 e8g7 26. h5h3 c8b8 27. f1f4 f9e8 28. g2h4 b8b5 29. f0e1


Position #2

b5e5 30. c0e2 a6a5 31. h3f3 g5g4 32. f4g4 e5e3 33. f3e3 e6e3 34. h4g6 e3e6 35. g6i7 e6e3 36. i7g8 e9f9 37. g4g6 e3e6 38. g6g4 g7h5 39. i3i4 e8d7 40. e0f0 h5i3 41. g4g6 i3h1 42. g6g1 h1i3 43. g1g6 e6g6 44. g8e7 f9e9 45. e7g6 i3g2 46. f0f1 a5a4 47. e1f2 f5h6 48. g6e5 d7e8 49. e5c4 h6i4 50. c4b6 a4b4 51. b6d5 b4b3 52. d5b6 e9f9 53. b6d5 i4g3 54. f1e1 i6i5 55. d5e3 g2i3 56. e1d1 g3e4 57. f2e1 b3c3 58. e3c4 c3c2 59. c4e3 c2c1 60. d1d2 i5i4 61. g0i2 i3g2 62. e1f2 e8d7 63. i2g4 i4h4 64. g4i2 h4h3 65. e2g0 h3g3 66. d2e2 g3f3 67. e2d2 f3e3 68. i2g4 e3f3 69. d2e2 f3f2 70. e2e1 f2f1 71. e1e0 f1f0 (0 - 1)

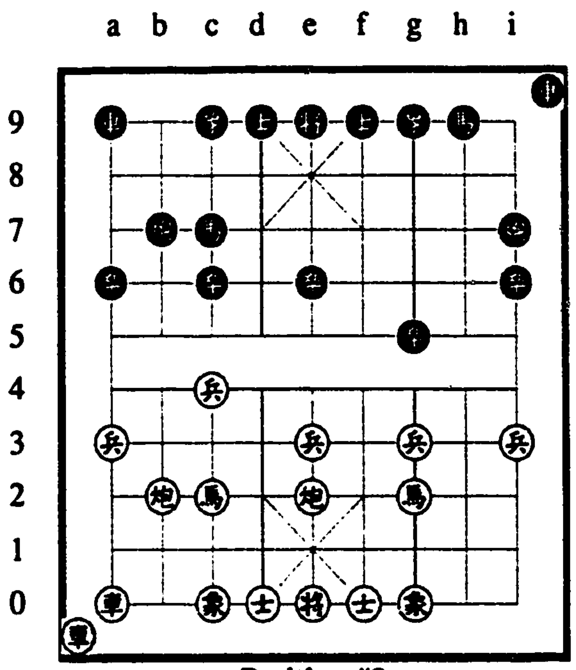
Abyss+SIN'100—Abyss-SIN'100: **1. h2e2 h9g7 2. h0g2 i9h9 3. c3c4 g6g5 4. b0c2 b9c7 5. i0i1 c9e7** 6. c2d4 h7h3 7. b2c2 h9h4 8. i1d1 a9a8 9. c2c6 b7b2 10. d1d2 b2b3 11. c4c5 b3g3 12. d4b5 c7b9 13. d2d6 g3g0 14. f0e1 g0i0 15. e2e6 g7e6 16. c6e6 f9e8 17. e0f0 h4h6 18. e6g6 h3h0 19. f0f1 h0c0 20. a0a1 g5g4 21. g2f0 i0g0 22. g6f6 a8d8 23. d6a6 c0c2 24. b5c3 d8d2 25. c3e2 (0 - 1)

Abyss-SIN'100—Abyss+SIN'100: **1. h2e2 h9g7 2. h0g2 i9h9 3. c3c4 g6g5 4. b0c2 b9c7 5. i0i1 c9e7** draw after 143 moves.

Opening #3:

Abyss+SIN'30—Abyss-SIN'30: **1. h2e2 h9g7 2. h0g2 i9h9 3. i0h0 b9c7 4. c3c4 g6g5 5. h0h6 h7i7 6. h6h9 g7h9 7. b0c2** (Position #3) b7b3 8. c2d4 b3g3 9. g0i2 a9b9 10. b2c2 c9e7 11. d4e6 c7e6 12. e2e6 d9e8 13. e6a6 b9b4 14. c0e2 b4b2 15. a0c0 g3a3 16. f0e1 a3a0 17. c2c6 b2e2 18. g2f4 e2e3 19. a6a9 a0b0 20. c0b0 e9d9 21. c6c9 d9d8 22. b0b8 d8d7 23. b8b1 d7d8 24. b1d1 e8d7 25. d1b1 i7i8 26. a9f9 h9g7 27. f9f8 e3i3 28. f4g6 d8e8 29. b1b8 e8e9 30. c9c8 i8c8 31. b8c8 i3i2 32. g6e7 g7h5 33. f8i8 g9e7 34. i8i2 g5g4 35. i2e2 e7g5 36. c8c9 e9e8 37. c9c6 i6i5 38. e2e5 h5g3 39. e5i5 g3i2 40. c6e6 e8d8 41. c4c5 g4g3 42. e6g6 g5e7 43. i5d5 d8e8 44. g6g8 e8e9 45. g8d8 e7c5 46. d8d7 e9e8 47. d5i5 i2g1 48. e0f0 c5e7 49. i5e5 e7g9 50. d7g7 g1e2 51. g7g9 g3f3 52. g9g8 e8e7 53. e5a5 e2d4 54. a5a7 f3e3 55. g8g7 e7e8 56. g7g4 d4b5 57. g4e4 e8d8 58. e4b4 b5a7 59. b4b8 d8d9 60. b8b7 d9d8 61. b7a7 e3e2 62. a7a8 d8d7 (1 - 0)

Abyss-SIN'30—Abyss+SIN'30: **1. h2e2 h9g7 2. h0g2 i9h9 3. i0h0 b9c7 4. c3c4 g6g5 5. h0h6 h7i7 6. h6h9 g7h9 7. b0c2** draw after 150 moves.



Position #3

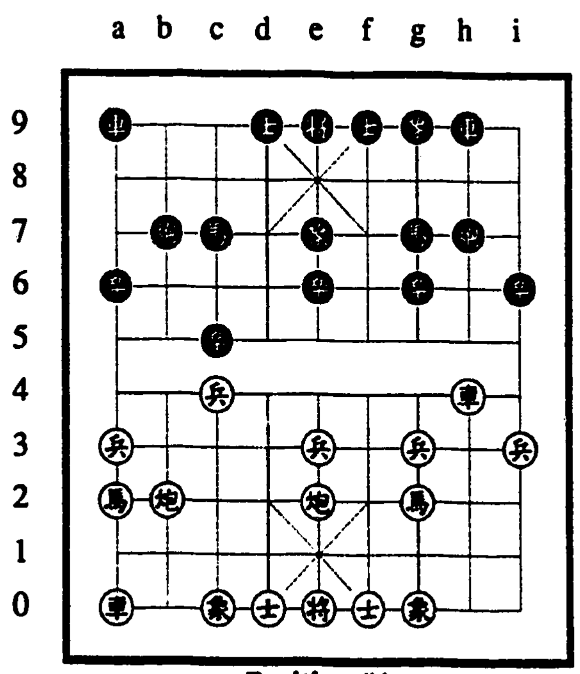
Abyss+SIN'100—Abyss–SIN'100: **1. h2e2 h9g7 2. h0g2 i9h9 3. i0h0 b9c7 4. c3c4 g6g5 5. h0h6 h7i7 6. h6h9 g7h9 7. b0c2** draw after 63 moves.

Abyss–SIN'100—Abyss+SIN'100: **1. h2e2 h9g7 2. h0g2 i9h9 3. i0h0 b9c7 4. c3c4 g6g5 5. h0h6 h7i7 6. h6h9 g7h9 7. b0c2** b7b3 8. c2d4 b3g3 9. g0i2 a9b9 10. a0b0 h9g7 11. d4c6 c9e7 12. b2b6 d9e8 13. c4c5 e7c5 14. b6e6 g7e6 15. b0b9 c7b9 16. e2e6 e8f7 17. e6a6 g3a3 18. a6i6 a3i3 19. c6e5 b9d8 20. i6f6 i3h3 21. c0e2 f7e8 22. d0e1 i7e7 23. i2g0 h3g3 24. f6b6 d8e6 25. e3e4 e8f7 26. b6b9 f7e8 27. b9b6 e7c7 28. g2e3 c7e7 29. e5c6 e6d4 30. c6e7 c5e7 31. e3c4 g3h3 32. b6b4 d4c6 33. b4b2 h3h4 34. e4e5 g5g4 35. c4d6 g4g3 36. b2b0 h4h8 37. b0b9 g3f3 38. e5e6 c6d4 39. b9b6 f3g3 40. d6e4 g3g2 41. e6f6 h8h0 42. e4g3 d4b3 43. e0d0 h0h8 44. g3i4 h8h5 45. i4h6 h5d5 46. d0e0 g2g1 47. b6a6 g1f1 48. a6a9 b3c1 49. e0d0 d5d8 50. e2c0 e8d7 51. e1d2 c1b3 52. f6e6 b3d2 53. c0e2 d2b3 54. e6d6 d8d6 55. h6f7 e9e8 56. f7e5 b3d4 57. d0e0 d4e2 58. e5g4 d6e6 59. f0e1 f1e1 60. e0f0 e2f4 61. a9g9 (0 - 1)

Opening #4:

Abyss+SIN'30—Abyss–SIN'30: **1. h2e2 h9g7 2. h0g2 i9h9 3. i0h0 c6c5 4. h0h4 b9c7 5. b0a2 c9e7 6. c3c4** (Position #4) c5c4 7. h4c4 g7e8 8. c4i4 b7b6 9. i4i6 g6g5 10. i6i4 h7h4 11. b2b1 b6b4 12. i4i5 b4b2 13. g2e1 b2b3 14. a2c1 b3g3 15. e2h2 h4f4 16. h2e2 g3g1 17. i5i4 f4f1 18. a0a2 f1c1 19. i4b4 a9a8 20. a2d2 h9h3 21. e1c2 g1b1 22. b4b1 c1h1 23. d2d7 g5g4 24. e2e6 a8a7 25. b1b4 g4g3 26. e3e4 h3i3 27. e4e5 h1h7 28. d7d8 h7h8 29. d8d7 i3i5 30. b4e4 h8h7 31. d7d8 c7e6 32. e5e6 h7h0 33. e4d4 i5e5 34. d0e1 a7a9 35. e0d0 e8g7 36. d8d9 a9d9 37. d4d9 e9e8 38. e6f6 g3g2 39. d9d8 e8e9 40. d8d9 e9e8 41. d9d8 e8e9 42. d8d9 e9e8 43. f6f7 g7f5 44. d9d8 e8e9 45. f7f8 e5d5 46. d8d5 f9e8 47. d5d8 e8d7 48. d8d7 f5d6 (1 - 0)

Abyss–SIN'30—Abyss+SIN'30: **1. h2e2 h9g7 2. h0g2 i9h9 3. i0h0 c6c5 4. h0h4 b9c7 5. b0a2 c9e7 6. c3c4** c5c4 7. h4c4 g7e8 8. c4i4 b7b3 9. g3g4 h9i9 10. a0a1 h7g7 11. f0e1 i9i8 12. b2c2 a9b9 13. a2c3 i6i5 14. i4h4 b3b4 15. h4h6 g7g4 16. g0i2 g4g5 17. a1c1 b4b6 18. a3a4 g5g3 19. c3e4 g6g5 20. h6h4 g3f3 21. e4d6 e8g7 22. h4c4 g7f5 23. c2b2 b6c6 24. c4c6 b9b2 25. g2h4 g5g4 26. i2g4 i8d8 27. d6c4 b2b7 28. h4g6 d8d4 29. g4i2 f5e3 30. g6i5 b7b4 31. c4b6 f3f7 32. c1c3 e3f5 33. i5h7 g9i7 34. h7f6 d4d8 35. c3c4 b4b1 36. c4c1 b1b3 37. c1c3 b3b1 38. c3c1 b1b3 39. c1c3 b3c3 40. c6c3 d8d6 41. c3b3


Position #4

e6e5 42. b6c8 d6d8 43. c8a7 c7a8 44. b3f3 e5e4 45. e2f2 f5d4 46. f2d2 d8h8 47. a7c8 h8c8 48. f3d3 c8c4 49. f6e4 d4e6 50. e4c3 e6f4 51. c0a2 c4a4 52. d3d8 a4a3 53. c3b5 a3h3 54. d8a8 h3h0 55. e1f0 h0f0 56. e0e1 f4g2 57. e1d1 f0f1 58. d0e1 f1e1 59. d1d0 g2e3 60. d2d3 f7f5 61. i2g4 f5d5 62. b5d4 d5d3 63. g4e2 e1e2 64. a8a6 e2e1 65. a6a3 d3d2 66. a3e3 e1e3 67. i3i4 e7g9 68. d4c2 e3c3 69. c2b0 (0 - 1)

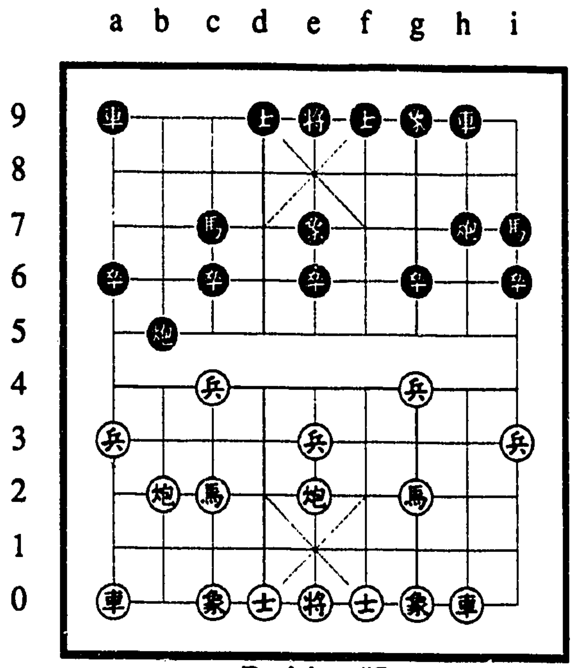
Abyss+SIN'100—Abyss-SIN'100: **1. h2e2 h9g7 2. h0g2 i9h9 3. i0h0 c6c5 4. h0h4 b9c7 5. b0a2 c9e7 6. c3c4** a9a8 7. c4c5 e7c5 8. h4h6 c7b5 9. b2b7 h7b7 10. h6h9 g7h9 11. e2e6 a8i8 12. a2c1 b7g7 13. c1e2 g6g5 14. d0e1 i8f8 15. a3a4 h9i7 16. a4a5 b5c7 17. e6e5 f8h8 18. e3e4 g5g4 19. e2d4 a6a5 20. d4e6 c7e8 21. e6g7 e8g7 22. g3g4 h8h3 23. a0a5 h3c3 24. g0e2 i7h9 25. a5a6 i6i5 26. a6e6 f9e8 27. e6i6 c5e7 28. i6i5 c3c6 29. e5a5 h9i7 30. a5a9 c6c9 31. a9a2 i7g6 32. i5d5 g7e6 33. d5d6 g6f8 34. a2a8 e8f9 35. e4e5 c9a9 36. d6d8 d9e8 37. e5e6 f8e6 38. d8d6 e6g7 39. d6g6 g7h9 40. g6a6 a9b9 41. a8a9 b9b3 42. a6c6 b3a3 43. c6c9 e8d9 44. c9b9 f9e8 45. e0d0 a3d3 46. d0e0 d3a3 47. g2f4 a3a4 48. f4d5 a4a7 49. d5f6 g9i7 50. e0d0 a7a9 51. b9a9 i7g9 52. i3i4 h9g7 53. i4i5 e9f9 54. g4g5 g7e6 55. i5h5 e6c7 56. g5g6 f9e9 57. f6g8 e9f9 58. a9a5 e8f7 59. g6f6 f7e8 60. a5f5 f9f8 61. f6e6 e8f7 62. e6e7 d9e8 63. e2c4 g9i7 64. e7e8 c7e8 65. g8i7 e8d6 66. i7g6 f8f9 (1 - 0)

Abyss-SIN'100—Abyss+SIN'100: **1. h2e2 h9g7 2. h0g2 i9h9 3. i0h0 c6c5 4. h0h4 b9c7 5. b0a2 c9e7 6. c3c4** draw after 45 moves.

Opening #5:

Abyss+SIN'30—Abyss-SIN'30: **1. h2e2 b9c7 2. h0g2 h9i7 3. i0h0 i9h9 4. c3c4 c9e7 5. b0c2 b7b5 6. g3g4** (Position #5) h7g7 7. h0h9 i7h9 8. c2d4 g7g8 9. d4e6 a9a7 10. b2b4 g8e8 11. e6d4 b5d5 12. d0e1 a7b7 13. a0b0 c7e6 14. e2e6 e8e6 15. g2f4 e6d6 16. d4c2 d5f5 17. c4c5 d9e8 18. c5c6 d6e6 19. c6d6 e6f6 20. f4d5 b7c7 21. c2d4 f5e5 22. e3e4 c7c4 23. e4e5 c4d4 24. d6e6 f6f1 25. g0i2 f1i1 26. c0e2 d4e4 27. d5b6 i1i0 28. i2g0 e4d4 29. b0d0 d4d0 30. e0d0 e8d7 (1 - 0)

Abyss-SIN'30—Abyss+SIN'30: **1. h2e2 b9c7 2. h0g2 h9i7 3. i0h0 i9h9 4. c3c4 c9e7 5. b0c2 b7b5 6. g3g4** draw after 150 moves.



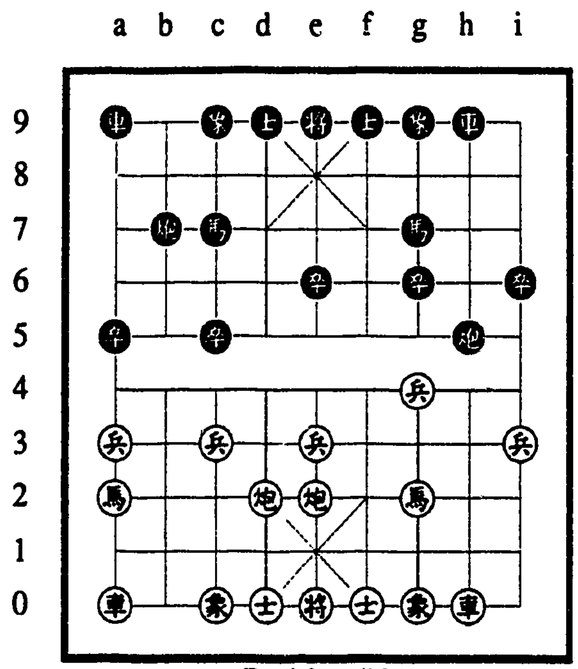
Position #5

Abyss+SIN'100—Abyss–SIN'100: **1. h2e2 b9c7 2. h0g2 h9i7 3. i0h0 i9h9 4. c3c4 c9e7 5. b0c2 b7b5 6. g3g4** i6i5 7. h0h6 h7g7 8. h6h9 i7h9 9. g2f4 g6g5 10. g0i2 b5a5 11. b2a2 g5g4 12. a2a5 a6a5 13. e2e6 c7e6 14. f4e6 g4h4 15. c2d4 g7i7 16. e3e4 a9c9 17. e4e5 i7i3 18. a3a4 a5a4 19. a0a4 i5i4 20. a4a6 i3c3 21. i2g0 d9e8 22. a6b6 h9i7 23. b6b3 c3i3 24. b3b6 i3c3 25. b6b3 c3c1 26. b3b6 i7g6 27. e5f5 h4g4 28. f5f6 g6f4 29. e6f4 g4f4 30. f0e1 c1c3 31. e0f0 i4h4 32. b6b4 c9d9 33. d4e6 h4g4 34. f6g6 d9d6 35. b4b9 e8d9 36. e6g5 e7g5 37. g6g7 g5e7 38. b9b8 g4g3 39. g7g8 f9e8 40. b8b9 c3f3 41. b9b6 d6g6 42. g8g9 g3g2 43. g0i2 g6h6 44. i2g0 h6h0 45. f0e0 h0g0 46. e1f0 g2f2 47. d0e1 f2f1 48. g9f9 e8f9 49. c0e2 f3i3 50. e0d0 i3i1 51. b6c6 i1e1 52. d0d1 g0f0 53. c6d6 f9e8 54. c4c5 f4f3 55. c5d5 f0c0 56. d6a6 c0c1 57. d1d0 e1e0 58. a6c6 c1c6 59. d5c5 c6d6 60. c5d5 (0 - 1)

Abyss–SIN'100—Abyss+SIN'100: **1. h2e2 b9c7 2. h0g2 h9i7 3. i0h0 i9h9 4. c3c4 c9e7 5. b0c2 b7b5 6. g3g4** i6i5 7. h0h6 h7g7 8. h6h9 i7h9 9. g2f4 g6g5 10. g0i2 b5a5 11. c0a2 g5g4 12. i2g4 g7h7 13. f4e6 h7h0 14. f0e1 c7e6 15. e2e6 d9e8 16. e0f0 h9g7 17. e6g6 a9b9 18. b2b0 h0h6 19. a3a4 a5g5 20. b0b6 h6i6 21. b6b4 i6i3 22. g6a6 g7f5 23. a2c0 f5g3 24. a0b0 b9b5 25. a6a9 g5h5 26. g4i2 g3i2 27. b4b2 i2h0 28. c0e2 b5f5 29. e1f2 (0 - 1)

Opening #6:

Abyss+SIN'30—Abyss–SIN'30: **1. h2e2 h9g7 2. h0g2 i9h9 3. i0h0 b9c7 4. g3g4 c6c5 5. b0a2 a6a5 6. b2d2 h7h5** (Position #6) 7. h0h4 a9b9 8. d2c2 c7b5 9. c2c5 b5a3 10. g2f4 b7e7 11. f0e1 a5a4 12. f4e6 h5f5 13. h4h9 g7h9 14. e6c7 b9a9 15. a0a1 d9e8 16. a1d1 a9a7 17. d1d5 f5c5 18. d5c5 e7e2 19. c0e2 c9e7 20. c5c6 a7b7 21. e3e4 b7b2 22. c6a6 e8d9 23. a6a4 b2a2 24. c7b5 a2a1 25. a4a3 a1d1 26. e4e5 h9g7 27. a3a6 g7h9 28. a6g6 i6i5 29. g6i6 h9g7 30. i6i5 f9e8 31. e5d5 d1d3 32. i3i4 d3d1 33. d5d6 d1d3 34. c3c4 g7h9 35. i5f5 h9g7 36. f5e5 g7i6 37. i4i5 i6h8 38. d6e6 d3b3 39. b5d6 b3f3 40. c4c5 h8g6 41. e5d5 e9f9 42. c5c6 e8d7 43. c6c7 d7e8 44. i5h5 f3f7 45. e6e7 g6e7 46. d5g5 f7f6 47. g5g9 f9f8 48. d6b5 f6b6 49. g9g5 b6f6 50. g5g7 e7d5 51. g4g5 f6b6 52. b5d4 b6b4 53. d4f5 b4f4 54. c7c8 d5e3 55. f5d6 f4f6 56. g7g8 f8f9 57. g8g9 f9f8 58. g9g6 f6g6 59. g5g6 e3g2 60. g6g7 g2f4 61. e0f0 f8f9 62. h5h6 f9e9 63. g7g8 f4d5 64. g8f8 d5f6 65. c8d8 f6d7 66. f8g8 d7f6 67. g8h8 f6h5 68. [illegible] h5f4 69. h8g8 f4e6 70. d8c8



Position #6

e6f4 71. c8d8 f4e6 72. d6b7 e6f4 73. b7c5 e9f9 74. e0f0 f9e9 75. h6h7 f4g2 76. f0e0 e9f9 77. d8d9 g2h4 78. h7g7 h4f5 79. g8g9 f9f8 80. c5e6 f5e7 81. g7g8 f8f7 82. e6g5 (1 - 0)

Abyss-SIN'30—Abyss+SIN'30: **1. h2e2 h9g7 2. h0g2 i9h9 3. i0h0 b9c7 4. g3g4 c6c5 5. b0a2 a6a5 6. b2d2 h7h5** draw after 90 moves.

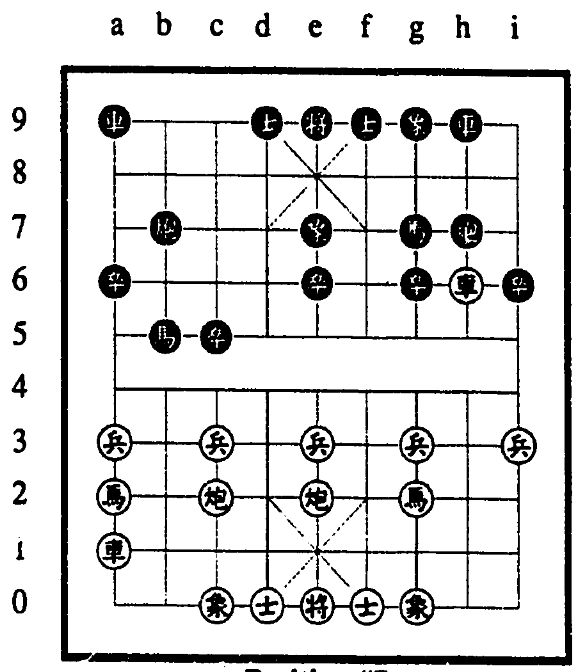
Abyss+SIN'100—Abyss-SIN'100: **1. h2e2 h9g7 2. h0g2 i9h9 3. i0h0 b9c7 4. g3g4 c6c5 5. b0a2 a6a5 6. b2d2 h7h5** 7. e3e4 c9e7 8. a0a1 a5a4 9. a3a4 a9a4 10. a1f1 a4e4 11. d2c2 h5e5 12. d0e1 h9h0 13. g2h0 e4a4 14. f1f8 b7b1 15. a2b0 a4a0 16. f8d8 a0b0 17. d8d0 b0a0 18. g4g5 g6g5 19. c3c4 b1b0 20. c4c5 g7e8 21. c5c6 b0d0 22. e0d0 a0c0 23. d0d1 c0c2 24. c6c7 e8c7 25. h0g2 c2c1 26. d1d2 c1c4 27. d2d1 c4d4 28. e1d2 e5d5 29. d1e1 d5d2 30. g2e3 d4d3 31. e3f5 d3i3 32. f5d4 g5g4 33. e1d1 i3d3 34. d4c6 d2b2 35. e2d2 b2b8 36. c6b4 d3d4 37. b4c6 d4d3 38. c6b4 d3b3 39. b4c6 b3b1 40. d1d0 b1b0 41. d0d1 b8c8 42. c6d4 b0f0 43. g0i2 f0f1 44. d1d0 f1f2 45. d0d1 f2i2 46. d2e2 e6e5 47. d1d0 c8d8 48. d0e0 c7d5 (0 - 1)

Abyss-SIN'100—Abyss+SIN'100: **1. h2e2 h9g7 2. h0g2 i9h9 3. i0h0 b9c7 4. g3g4 c6c5 5. b0a2 a6a5 6. b2d2 h7h5** 7. e3e4 c9e7 8. a0a1 a5a4 9. a3a4 a9a4 10. a1f1 a4e4 11. h0h4 h5e5 12. d0e1 h9h4 13. g2h4 e5e2 14. c0e2 e4a4 15. a2c1 g6g5 16. f1g1 a4a1 17. e1f2 g5g4 18. g1g4 a1c1 19. g4g7 c1c3 20. h4i6 c3i3 21. i6h8 c7d5 22. g7g4 i3h3 23. h8g6 h3f3 24. g4d4 d5c7 25. d4b4 b7b5 26. f2e1 d9e8 27. g6f4 e6e5 28. f4g6 f3f6 29. g6h4 b5b6 30. h4g2 c7a6 31. b4d4 b6b4 32. d4g4 b4b0 33. d2c2 b0b4 34. c2b2 b4e4 35. e0d0 a6b4 36. b2d2 b4c2 37. d0e0 f6a6 38. d2d0 a6a0 39. g4e4 e5e4 40. g2i3 c2b4 41. i3g4 b4d3 42. d0c0 c5c4 43. g4h6 c4c3 44. g0i2 c3c2 45. e0d0 e9d9 46. h6g8 c2c1 47. g8h6 c1c0 48. e2c0 a0c0 49. d0d1 e4d4 50. d1d2 c0c2 51. d2d1 d3b2 52. i2g0 c2c6 53. d1d2 (0 - 1)

Opening #7:

Abyss+SIN'30—Abyss-SIN'30: **1. h2e2 h9g7 2. h0g2 b9c7 3. i0h0 i9h9 4. b0a2 c6c5 5. b2c2 c7b5 6. h0h6 c9e7 7. a0a1** (Position #7) draw after 44 moves.

Abyss-SIN'30—Abyss+SIN'30: **1. h2e2 h9g7 2. h0g2 b9c7 3. i0h0 i9h9 4. b0a2 c6c5 5. b2c2 c7b5 6. h0h6 c9e7 7. a0a1** b5a3 8. c2c1 g6g5 9. a1b1 a9b9 10. b1b3 g5g4 11. g3g4 b7b6 12. h6h0 f9e8 13. b3a3 h7h6 14. d0e1 b6b1 15. a3a4 b1b3 16. h0h1 i6i5 17. g2f4



Position #7

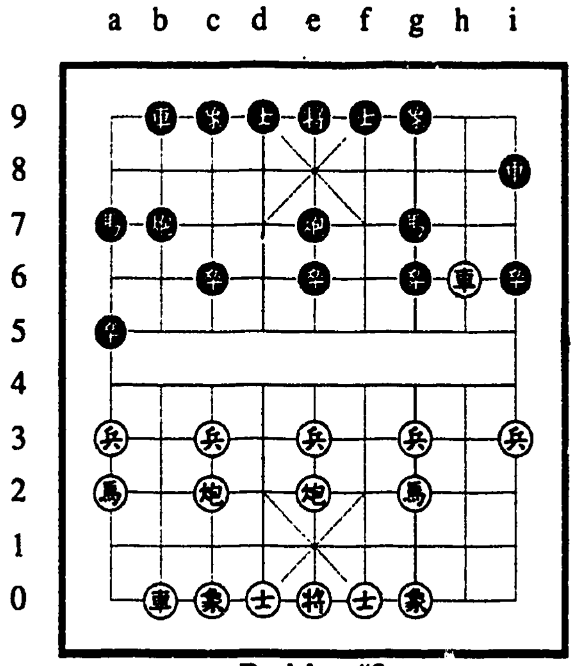
h6h4 18. h1h3 a6a5 19. a4d4 g7f5 20. d4e4 h9h6 21. g4g5 h4e4 22. h3h6 e4e2 23. c0e2 e7g5 24. h6e6 g5e7 25. f4d5 b3b5 26. d5f6 b5b8 27. e6b6 f5h6 28. c1b1 a5a4 29. b1b8 a4a3 30. a2b0 h6f7 31. f6g4 f7g5 32. b0c2 a3a2 33. e2c0 b9a9 34. b8b9 e7c9 35. g4i5 a2a1 36. b6b5 g5e6 37. b5b4 a9a5 38. i5h7 g9i7 39. h7f6 a5a7 40. b4e4 a7a6 41. c2a1 i7g5 42. i3i4 a6b6 43. i4i5 g5i7 44. b9a9 b6a6 45. a1c2 i7g9 46. a9a8 e9f9 47. a8c8 a6d6 48. e4e5 d6b6 49. f6g8 c9a7 50. e5f5 e8f7 51. f5f7 e6f8 52. f7a7 b6c6 53. c8a8 c6g6 54. a7b7 g6c6 55. a8a9 c6c9 56. a9b9 g9i7 57. b7i7 f8g6 58. i7f7 g6f8 59. g8f6 c9b9 60. f6d7 b9b8 61. d7b8 d9e8 62. f7f6 e8f7 63. f6f7 f9e9 64. f7f8 c5c4 65. c3c4 (1 - 0)

Abyss+SIN'100—Abyss–SIN'100: **1. h2e2 h9g7 2. h0g2 b9c7 3. i0h0 i9h9 4. b0a2 c6c5 5. b2c2 c7b5 6. h0h6 c9e7 7. a0a1** b5a3 8. c2c1 g6g5 9. c1h1 a3c2 10. a1f1 h7h1 11. e2e6 d9e8 12. h6h9 a9d9 13. f1c1 g7h9 14. c1c2 h9g7 15. e6f6 g7h5 16. c2c1 h1d1 17. c1b1 d9d2 18. g2i1 d2a2 19. b1b7 a2a3 20. f6f3 c5c4 21. b7c7 c4d4 22. c7c6 h5g3 23. i1g2 d1d3 24. f3f6 a6a5 25. f6e6 e9d9 26. e6g6 g5g4 27. g6d6 d9e9 28. d6e6 d4e4 29. g0e2 g4f4 30. e3e4 g3e4 31. c6c9 d3d9 (0 - 1)

Abyss–SIN'100—Abyss+SIN'100: **1. h2e2 h9g7 2. h0g2 b9c7 3. i0h0 i9h9 4. b0a2 c6c5 5. b2c2 c7b5 6. h0h6 c9e7 7. a0a1** b5a3 8. a1b1 b7d7 9. c2b2 a9b9 10. h6g6 b9b3 11. b1d1 f9e8 12. b2b1 h7h8 13. g6f6 h8g8 14. f0e1 g8g3 15. g2i1 g3c3 16. a2c3 c5c4 17. e2e6 g7e6 18. f6e6 c4c3 19. b1a1 a3b1 20. d1c1 h9h1 21. e1f2 h1c1 22. a1c1 c3d3 23. e6a6 d3e3 24. a6i6 b3c3 25. c1h1 b1d2 26. e0e1 d2c0 27. i6d6 c0b2 28. d6e6 e8f9 29. e1e0 d7a7 30. e6a6 a7d7 31. i3i4 c3d3 32. f2e1 d7c7 33. a6c6 c7a7 34. c6a6 a7c7 35. a6c6 c7a7 36. c6a6 a7b7 37. a6b6 b7c7 38. e1f0 b2c4 39. b6b1 c4d2 40. b1d1 d2f3 41. d1d3 f3g1 42. e0e1 g1f3 43. e1f1 e3d3 44. i4i5 e7g5 45. i1h3 f3d2 46. f1f2 d3e3 47. g0e2 (0 - 1)

Opening #8:

Abyss+SIN'30—Abyss–SIN'30: **1. h2e2 h7e7 2. h0g2 h9g7 3. i0h0 i9i8 4. b0a2 a6a5 5. b2c2 b9a7 6. a0b0 a9b9 7. h0h6** (Position #8) g6g5 8. h6f6 g9i7 9. b0b6 b7d7 10. b6b9 a7b9 11. c2c6 b9a7 12. c6c5 f9e8 13. e2c2 d7d6 14. f6f4 d6a6 15. c5c7 e8d7 16. f4f7 g7f9 17. c7e7 c9e7 18. f7f5 a6c6 19. f5a5 e7c9 20. c2c6 a7c6 21. a5c5 i8c8 22. a3a4 c9e7 23. c5d5 f9g7 24. a4a5 c8f8 25. a5b5 f8f2 26. b5b6 f2g2 27. b6c6 d7e8 28. g3g4 g2g4 29. d5d3 g4g0 30. a2b4 g0g3 31. b4d5 i7g9 32. i3i4 g3g4 33. c6c7 g4i4 34. c0e2 i4i3 35.


Position #8

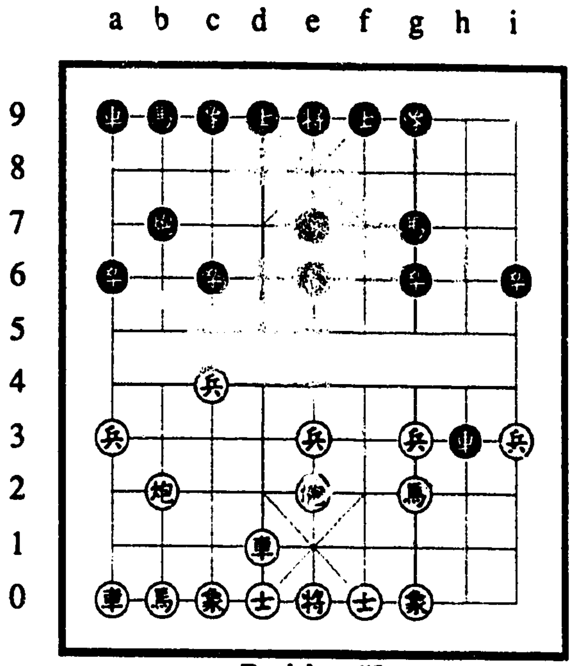
d5b4 g7f5 36. c7c8 i3e3 37. e0e1 e3e4 38. b4d5 g5g4 39. c8d8 i6i5 40. d5c7 f5d4 41. e1d1 g4f4 42. c3c4 i5i4 43. c7b5 d4c6 44. d8d9 e9f9 45. b5d6 e4e2 46. d6e8 f9f8 47. e8c7 e2c2 48. d3d8 f8f7 49. d8d6 c2c1 50. d1d2 c6a5 51. d6e6 c1c2 52. d2d1 c2c1 53. d1d2 a5b3 54. d2e2 c1c2 55. e2e1 c2c1 56. e1e2 b3d4 57. e2d2 d4e6 58. c7e6 c1c4 59. f0e1 f4e4 60. e1f2 c4c6 61. e6g7 e4e3 62. d2d1 e3e2 63. d0e1 c6d6 64. e1d2 d6d2 (0 - 1)

Abyss-SIN'30—Abyss+SIN'30: **1. h2e2 h7e7 2. h0g2 h9g7 3. i0h0 i9i8 4. b0a2 a6a5 5. b2c2 b9a7 6. a0b0 a9b9 7. h0h6** g6g5 8. b0b6 g7f5 9. e2e6 d9e8 10. h6f6 f5d4 11. c2c6 d4c6 12. b6c6 b7b6 13. c6b6 b9b6 14. c3c4 b6c6 15. g0e2 a7b5 16. f0e1 b5a3 17. e0f0 e9d9 18. a2c3 a3c2 19. e3e4 c9a7 20. g2i1 c6d6 21. c4c5 a7c5 22. e4e5 e7e5 23. c0a2 c2e3 24. a2c0 d6d3 25. e6c6 g9e7 26. f6f3 i6i5 27. i1g2 i8i6 28. c6c8 i6c6 29. c8a8 d3c3 30. f3e3 c3e3 31. g2e3 c6f6 32. f0e0 (0 - 1)

Abyss+SIN'100—Abyss-SIN'100: **1. h2e2 h7e7 2. h0g2 h9g7 3. i0h0 i9i8 4. b0a2 a6a5 5. b2c2 b9a7 6. a0b0 a9b9 7. h0h6** g6g5 8. h6g6 g9i7 9. c2c6 b7b6 10. g6g7 a7c6 11. e2c2 i8c8 12. c3c4 b6b7 13. c4c5 b9b8 14. g7g6 c6e5 15. g6e6 e5c4 16. c0e2 c4d2 17. e0e1 c8c5 18. c2c9 e9e8 19. e6g6 e8d8 20. c9f9 c5f5 21. g6d6 e7d7 22. f9i9 f5f2 23. g2h0 f2h2 24. d6d2 h2h0 25. i9i8 b8b9 26. b0c0 h0h1 27. e1e0 d8e8 28. d2d7 h1a1 29. c0c8 e8e9 30. d7h7 b7b0 31. e2c0 d9e8 32. i8i9 b9b1 33. h7h9 i7g9 (1 - 0)

Abyss-SIN'100—Abyss+SIN'100: **1. h2e2 h7e7 2. h0g2 h9g7 3. i0h0 i9i8 4. b0a2 a6a5 5. b2c2 b9a7 6. a0b0 a9b9 7. h0h6** g6g5 8. h6g6 g9i7 9. c3c4 i8b8 10. g0i2 b8f8 11. d0e1 f9e8 12. c2c6 f8f5 13. b0b6 b7d7 14. b6b9 a7b9 15. e2e6 g7e6 16. g6e6 d7a7 17. e6i6 f5e5 18. c6a6 e7e3 19. g2e3 e5e3 20. i6i7 a7e7 21. a6a9 e3c3 22. e1d0 c3c4 23. i7g7 a5a4 24. a9c9 c4c9 25. a3a4 e7e6 26. g7g9 e8f9 27. g9g6 e6e8 28. g6b6 c9c5 29. i2g0 b9c7 30. b6b4 c7d5 31. b4b2 d5e3 32. f0e1 e3g2 33. e1f0 g2i3 34. b2h2 c5e5 35. d0e1 e5e3 36. g0e2 e3g3 37. a4a5 i3g2 38. e0d0 g3d3 39. d0e0 d3g3 40. e0d0 g2f0 41. h2h4 e8e1 42. h4f4 f0g2 43. f4f1 g3h3 44. a5b5 e1a1 45. f1g1 a1a0 46. a2b0 h3b3 47. g1g2 b3b0 48. e2g0 b0b5 49. c0e2 b5d5 50. d0e0 a0a8 51. g2g3 a8e8 52. e2c0 e8e6 53. e0e1 d5c5 54. g0i2 c5e5 55. e1f1 e6f6 56. c0e2 e5e2 57. g3g5 e2e3 58. g5g2 (0 - 1)

Opening #9:


Position #9

Abyss+SIN'30—Abyss-SIN'30: **1. h2e2 h7e7 2. i0i1 h9g7 3. h0g2 i9h9 4. i1d1 h9h3 5. c3c4** (Position #9) h3g3 6. b0c2 g3g2 7. d1d7 b7a7 8. d7b7 g2g0 9. b2b9 g0g3 10. a0b0 a7a8 11. b7d7 f9e8 12. d7c7 g3i3 13. c7c9 e8f9 14. c9c6 d9e8 15. b0b8 i3i4 16. c6c9 e8d9 17. b8d8 e7e3 18. c2e3 a9b9 19. c9b9 (1 - 0)

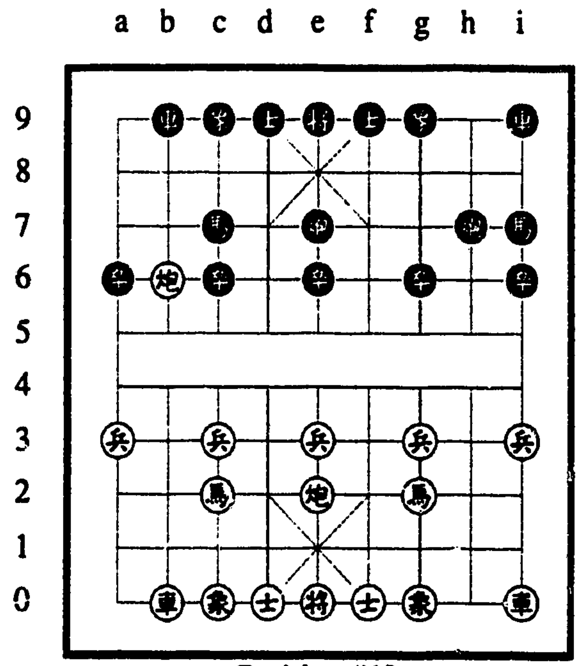
Abyss-SIN'30—Abyss+SIN'30: **1. h2e2 h7e7 2. i0i1 h9g7 3. h0g2 i9h9 4. i1d1 h9h3 5.** c3c4 h3g3 6. b0c2 g3g2 7. e2e6 g7e6 8. b2g2 e6f4 9. g2e2 f4e2 10. g0e2 b7c7 11. d1g1 c6c5 12. g1g6 f9e8 13. g6g9 e8f9 14. c2e1 c5c4 15. e2c4 e7e1 16. f0e1 c7i7 17. g9g6 i7i3 18. g6i6 i3h3 19. e3e4 d9e8 20. a0b0 b9c7 21. e4e5 h3h0 22. i6c6 a9a7 23. c4e2 h0h7 24. e5e6 h7i7 25. b0b4 i7i6 26. e6e7 i6i2 27. b4e4 i2i0 28. e7e8 f9e8 29. c6i6 e9d9 30. i6i0 e8d7 31. i0i9 d9d8 32. i9c9 a7a8 33. c9c7 a6a5 34. e4d4 d8e8 35. c7d7 e8f8 36. d4h4 f8e8 37. h4e4 e8f8 38. d7e7 f8f9 (1 - 0)

Abyss+SIN'100—Abyss-SIN'100: **1. h2e2 h7e7 2. i0i1 h9g7 3. h0g2 i9h9 4. i1d1 h9h3 5. c3c4** h3g3 6. d1d6 c9a7 7. d6c6 a6a5 8. b2c2 b7c7 9. a0a1 e7e8 10. c2a2 e8c8 11. c6d6 c7c0 12. d0e1 b9c7 13. d6d8 a9c9 14. d8d7 g7h9 15. e2e6 e9e8 16. g0e2 c0c3 17. e3e4 c8a8 18. a1b1 c3b3 19. g2e3 b3b5 20. e6c6 a8c8 21. b1b5 c8c6 22. b5e5 e8f8 23. e5h5 f9e8 24. h5h8 f8f9 25. d7g7 f9e9 26. a2a5 e8f9 27. a5e5 g3f3 28. g7g8 f3f8 (1 - 0)

Abyss-SIN'100—Abyss+SIN'100: **1. h2e2 h7e7 2. i0i1 h9g7 3. h0g2 i9h9 4. i1d1 h9h3 5. c3c4** h3g3 6. d1d6 b7c7 7. d6c6 g6g5 8. c4c5 g5g4 9. c5d5 e7e8 10. c6b6 c7e7 11. b6b8 e7f7 12. g2e1 e8e3 13. g0i2 g3f3 14. b2b3 e3e4 15. b8b7 g9e7 16. b3b5 g4f4 17. b[illegible]7 (0 - 1)

Opening #10:

Abyss+SIN'30—Abyss-SIN'30: **1. h2e2 b7e7 2. h0g2 b9c7 3. b0c2 h9i7 4. a0b0 a9b9 5. b2b6** (Position #10) d9e8 6. c3c4 i9h9 7. i0i1 c7a8 8. b6b1 h7g7 9. g2e1 a8c7 10. e2i2 b9b3 11. i2i6 g7g3 12. i3i4 e7e3 13. c2e3 b3e3 14. i4i5 g3a3 15. g0e2 a3d3 16. e1c2 e3e4 17. b1c1 d3h3 18. c1h1 h3d3 19. h1c1 d3h3 20. c1h1 h3d3 21. b0b7 d3d1 22. h1e1 e4e2 23. b7c7 g9e7 24. i1i2 e2i2 25. i6i2 d1d7 26. i2e2 d7d5 27. e1e6 e9d9 28. e2d2 d5e5 29. c2d4 e5d5 30. d2d5 h9h3 31. e6a6 h3h4 32. a6a9 c9a7 33. d4f5 h4e4 34. f0e1 e4e5 35. f5e7 d9d8 36. e7c6 e5e1 (1 - 0)



Position #10

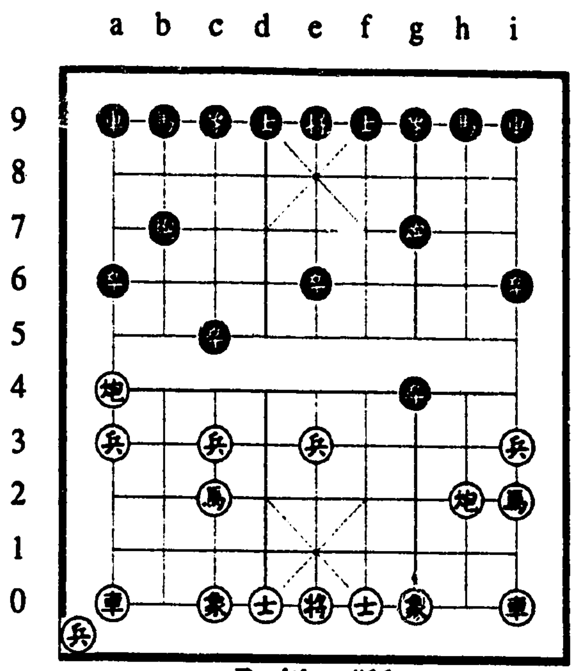
Abyss–SIN'30—Abyss+SIN'30: **1. h2e2 b7e7 2. h0g2 b9c7 3. b0c2 h9i7 4. a0b0 a9b9 5. b2b6** d9e8 6. c3c4 i9h9 7. i0i1 c7a8 8. b6b8 h7h2 9. c2d4 h2e2 10. c0e2 h9h2 11. i1i2 h2i2 12. g0i2 e7a7 13. d4e6 a7a3 14. e6d8 b9a9 15. b8b3 a8c7 16. d8c6 c9e7 17. b3c3 c7e6 18. g3g4 a3a2 19. g2h4 a2a3 20. c3c0 a3i3 21. h4i6 e6f4 22. b0b2 a9d9 23. e3e4 i3c3 24. c6b8 d9d4 25. b8a6 d4e4 26. e0e1 f4g2 27. e1e0 c3e3 28. e2g0 e3h3 29. d0e1 e4g4 30. b2b9 e8d9 31. a6c7 g4d4 32. c0c2 f9e8 33. g0e2 g2e3 34. c2c0 h3h6 35. b9b3 h6h0 36. i2g0 e3d5 37. c7d5 d4d5 38. b3b7 h0h7 39. b7b4 h7h6 40. i6h4 i7h5 41. h4g2 h6i6 42. c0a0 d5a5 43. a0d0 e7c9 44. g2h4 g9e7 45. g0i2 a5e5 46. b4b2 e5e4 47. h4f5 h5f4 48. i2g0 i6i0 49. f5h6 f4e2 50. d0d1 e2c1 51. h6g8 e9f9 52. b2f2 e4f4 (1 - 0)

Abyss+SIN'100—Abyss–SIN'100: **1. h2e2 b7e7 2. h0g2 b9c7 3. b0c2** [illegible] **4. a0b0 a9b9 5. b2b6** draw after 80 moves.

Abyss–SIN'100—Abyss+SIN'100: **1. h2e2 b7e7 2. h0g2 b9c7 3. b0c2 h9i7 4. a0b0 a9b9 5. b2b6** d9e8 6. g3g4 h7h5 7. c3c4 i9h9 8. i0h0 a6a5 9. c2d4 h5e5 10. h0h9 i7h9 11. d4c6 c7a8 12. b6b8 e5e2 13. c0e2 e7c7 14. d0e1 h9g7 15. c6a5 c7f7 16. c4c5 c9e7 17. c5c6 f7f8 18. b8b7 f8f3 19. g2f4 a8c9 20. e3e4 b9a9 21. a5c4 c9b7 22. b0b7 a9a6 23. b7b9 e8d9 24. b9b3 f3f1 25. c6b6 a6a7 26. f4g6 f1g1 27. b6b7 a7a8 28. b7b8 a8a6 29. g6h4 a6a7 30. b8c8 f9e8 31. c8d8 a7d7 32. b3b8 d7d3 33. h4i2 g1h1 34. i3i4 e9f9 35. i2h4 d3i3 36. d8e8 g7e8 37. b8d8 f9e9 38. e0d0 e8g7 39. d8d9 e9e8 40. d9d8 e8e9 41. h4g6 i3h3 42. d8d9 e9e8 43. a3a4 h3b3 44. d0e0 g9i7 45. c4d6 e8f8 46. d9d8 f8f7 47. d8g8 b3b0 48. e1d0 (1 - 0)

Opening #11:

Abyss+SIN'30—Abyss–SIN'30: **1. g3g4 h7g7 2. b0c2 g6g5 3. h0i2 g5g4 4. b2b4 c6c5 5. b4a4** (Position #11) b7a7 6. a4e4 g9e7 7. h2e2 g4f4 8. e4b4 f4f3 9. i2g1 f3f2 10. e2e6 d9e8 11. b4i4 h9i7 12. g1f3 f2f1 13. i0i1 i6i5 14. i4e4 i7g6 15. i1f1 g7g0 16. e0e1 g6f4 17. e6e5 i9h9 18. f1g1 g0i0 19. g1g4 h9h1 20. f3g1 i0i1 21. e1e0 i1g1 22. a0a1 c5c4 23. a1g1 h1g1 24. g4g1 f4h3 25. g1f1 c4c3 26. e5h5 e9d9 27. f1d1 a7d7 28. e4a4 c9a7 29. h5h9 e7g9 30. c2a1 c3d3 31. d1h1 h3f2 32. h1f1 f2g4 33. f1f8 a9a8 34. e3e4 g4i3 35. h9f9 b9c7 36. f8f3 i3h1 37. f9f8 e8f9 38. e4e5 i5i4 39. a1b3 i4h4 40. f8f4 a8b8 41. b3d4 a6a5 42. a4c4 d3d2 43. f4e4 f9e8 44. d4e6 c7b5 45. c4h4 d2d1 46. h4h9 g9e7 47. f3f9 d9d8 48. f9f1 d1d0 49. e0e1 b8b7 50. f1h1 b5a3 51. e4d4 d7c7 52. e5d5 c7d7 53. d5c5



Position #11

d7c7 54. d4d1 d8d9 55. e6d4 d9e9 56. h9i9 a5a4 57. h1h9 e8f9 58. h9h2 f9e8 59. h2h9 e8f9 60. h9h6 f9e8 61. c5c6 c7c9 62. h6h9 e8f9 63. h9h6 f9e8 64. e1f1 c9i9 65. h6h9 e8f9 66. d1e1 e7g9 67. d4e6 e9d9 68. e6c7 b7c7 69. c6c7 i9i2 70. c0e2 a3c2 71. h9g9 c2e1 72. g9f9 d9d8 73. f9f4 a7c9 74. f1e1 i2i6 75. f4a4 d8e8 76. a4e4 e8f8 77. e2g4 i6i7 78. e4e8 f8f9 79. e8e9 f9f8 80. e9c9 i7i1 81. c9d9 d0c0 82. c7c8 i1i8 83. c8d8 i8g8 84. d8e8 g8e8 85. d9d8 f8f9 86. d8e8 c0d0 (1 - 0)

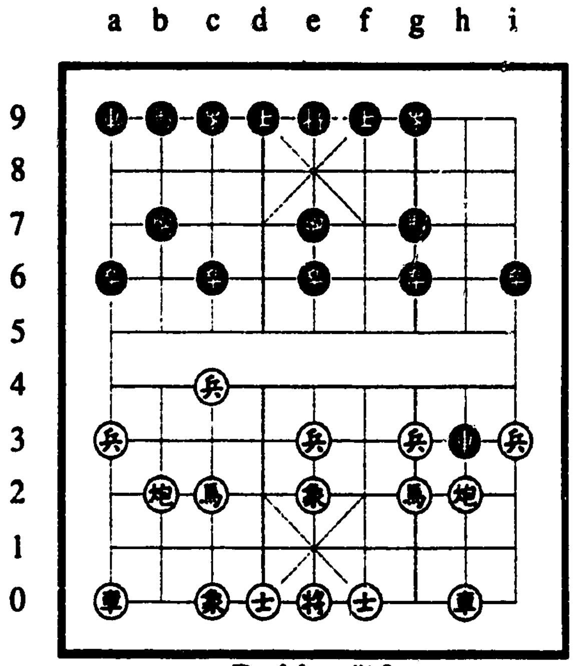
Abyss–SIN'30—Abyss+SIN'30: **1. g3g4 h7g7 2. b0c2 g6g5 3. h0i2 g5g4 4. b2b4 c6c5 5. b4a4** b7a7 6. c2a1 a7a4 7. a3a4 b9c7 8. g0e2 g4f4 9. a1c2 c7b5 10. i0g0 b5c3 11. g0g6 e6e5 12. g6e6 g9e7 13. e6e5 h9f8 14. h2f2 f8g6 15. e5f5 g6h4 16. f5f7 f4f3 17. f7g7 f3f2 18. i2g3 a9b9 19. c2e1 i9i8 20. g7g4 i8h8 21. e1g0 c3d1 22. a0a3 b9b1 23. f0e1 f2e2 24. g3e2 c9a7 25. e2g1 b1b8 26. g1f3 b8g8 27. f3e5 g8g4 28. e5g4 c5c4 29. e3e4 d1b2 30. a3b3 b2d1 31. b3b6 d1c3 32. b6a6 h8g8 33. g0h2 c4b4 34. a4a5 a7c9 35. a6h6 h4g2 36. e4e5 g8a8 37. a5a6 a8i8 38. h6b6 b4a4 39. b6c6 c3d1 40. a6b6 a4b4 41. b6b7 g2f4 42. b7c7 f9e8 43. c7c8 d1e3 44. c8d8 e8d7 45. c6c8 d7e8 46. c8c9 i8h8 47. g4f6 e9f9 48. h2f3 h8h0 49. e1f0 h0h3 50. f3g1 h3h1 51. g1f3 h1f1 52. f6e8 b4a4 53. e8g7 f9f8 54. c9d9 f1f0 55. e0f0 (1 - 0)

Abyss+SIN'100—Abyss–SIN'100: **1. g3g4 h7g7 2. b0c2 g6g5 3. h0i2 g5g4 4. b2b4 c6c5 5. b4a4** b7a7 6. h2e2 b9c7 7. a0a1 h9i7 8. i0h0 i9h9 9. c2e1 h9h0 10. i2h0 a7a4 11. a3a4 a9b9 12. a1c1 c7d5 13. c1d1 d5f4 14. e2a2 g7e7 15. a2a6 e7e3 16. e1g2 f4g2 17. h0g2 e3g3 18. g0e2 g3f3 19. d1d6 g4g3 20. d6e6 c9e7 21. g2e3 f3i3 22. e3f5 g3f3 23. f5e7 g9e7 24. e6i6 i3c3 25. i6i7 e7c9 26. f0e1 b9b5 27. a6e6 b5b6 28. e6e5 b6c6 29. i7f7 f3e3 30. e0f0 c6d6 31. f7f9 e9e8 32. a4a5 d6d1 33. e5h5 d1d5 34. h5h8 e3d3 35. f9f4 c9e7 36. a5a6 d5d6 37. a6a7 d6d7 38. a7a8 d3e3 39. a8b8 c3a3 40. f4f8 e8e9 41. b8c8 d9e8 42. h8h9 e8f7 43. f8f7 a3a7 44. f7f9 e9e8 45. f9f8 e8e9 46. c8d8 d7d8 47. f8d8 (1 - 0)

Abyss–SIN'100—Abyss+SIN'100: **1. g3g4 h7g7 2. b0c2 g6g5 3. h0i2 g5g4 4. b2b4 c6c5 5. b4a4** b7a7 6. g0e2 a7a4 7. a3a4 g4g3 8. h2h4 g3g2 9. i2h0 g2h2 10. i0i1 a9a8 11. a0b0 b9c7 12. h4i4 h9i7 13. i1g1 a8g8 14. g1g4 i6i5 15. i4i7 i9i7 16. g4h4 g7g2 17. c2e1 g2g3 18. h4h2 g3c3 19. h2i2 i7d7 20. i3i4 g8d8 21. e1c2 d7d2 22. b0b2 d8h8 23. h0g2 i5i4 24. i2i4 c3c0 25. e2c0 d2g2 26. b2a2 d9e8 27. d0e1 g2g5 28. i4e4 c9e7 29. c0e2 h8h3 30. a2b2 e6e5 31. e4b4 g5g2 32. e3e4 g2e2 33. e4e5 h3c3 34. b4b7 c7d9 35. c2d0 e2e5 36.

b2b6 c5c4 37. b6a6 c3a3 38. a6a8 c4c3 39. b7a7 e7c9 40. a7g7 g9e7 41. g7g3 d9c7 42. a8a9 e9d9 43. g3g6 c7b5 44. g6g2 b5d4 45. a9a6 d9e9 46. g2d2 d4f3 47. d2f2 f3g5 48. f2i2 c3d3 49. i2e2 d3e3 50. e2f2 a3a0 51. f2g2 g5e4 52. a6d6 e3f3 53. g2g1 a0a2 54. d0b1 a2b2 55. g1g4 b2b1 56. g4e4 e5e4 57. a4a5 e4a4 58. d6d5 a4a0 59. e1d0 f3f2 60. f0e1 f2f1 61. e1f0 e7c5 62. a5a6 a0a6 63. d0e1 f1f0 64. e1f0 (0 - 1)

Opening #12:



Position #12

Abyss+SIN'30—Abyss-SIN'30: **1. g0e2 h7e7 2. h0g2 h9g7 3. i0h0 i9h9 4. b0c2 h9h3 5. c3c4** (Position #12) h3g3 6. c2d4 g3i3 7. a0a1 i3g3 8. d4c6 e7c7 9. f0e1 g6g5 10. h2h6 c9e7 11. h6g6 g3f3 12. c6d4 f3f1 13. b2b1 f1f6 14. h0h6 b7b6 15. d4c6 f6f7 16. c6e7 g9e7 17. g6b6 g5g4 18. h6g6 g4f4 19. c4c5 b9d8 20. c5d5 a9a8 21. d5d6 e6e5 22. d6d7 c7a7 23. g6e6 f7f5 24. e6g6 f5h5 25. g6g7 h5h6 26. b6b4 h6d6 27. d7c7 d8c6 28. b4a4 d6f6 29. b1b9 e7c9 30. a1d1 a7g7 31. d1d9 e9e8 32. a4a8 f6g6 33. d9e9 e8f8 34. e9f9 f8e8 35. c7d7 g7g9 36. b9b8 (1 - 0)

Abyss-SIN'30—Abyss+SIN'30: **1. g0e2 h7e7 2. h0g2 h9g7 3. i0h0 i9h9 4. b0c2 h9h3 5. c3c4** draw after 66 moves.

Abyss+SIN'100—Abyss-SIN'100: **1. g0e2 h7e7 2. h0g2 h9g7 3. i0h0 i9h9 4. b0c2 h9h3 5. c3c4** h3i3 6. g3g4 i3i4 7. b2b4 i4i5 8. g2h4 g6g5 9. h4g6 i5h5 10. g4g5 h5g5 11. g6e7 c9e7 12. h2h7 b7c7 13. a0a2 a9a8 14. h0h6 i6i5 15. f0e1 g5d5 16. e0f0 g7f5 17. h6h0 a8h8 18. h7h1 f5g3 19. h1h6 i5i4 20. f0e0 c7d7 21. b4b6 f9e8 22. b6e6 b9c7 23. e6f6 g3f5 24. h6h5 d5d6 25. e3e4 d6f6 26. e4e5 f5g3 27. e5e6 f6e6 28. c2g4 e6h6 29. h0h3 h6h5 30. h3g3 h5g5 31. c0e2 h8h0 32. e1f0 h0h1 33. d0e1 c6c5 34. c4c5 g5c5 35. g3d3 c7d5 36. d3b3 i4h4 37. b3b9 c5c3 38. b9b5 d5f4 39. g4i2 f4e2 40. a2a1 e2g1 41. e0d0 c3d3 42. a1d1 d7d1 43. c2b4 d3b3 44. d0d1 g1e2 45. b4d3 b3d3 (0 - 1)

Abyss-SIN'100—Abyss+SIN'100: **1. g0e2 h7e7 2. h0g2 h9g7 3. i0h0 i9h9 4. b0c2 h9h3 5. c3c4** draw after 130 moves.