The University of Alberta

EXPERIMENTS IN CHESS CAPTURE SEARCH

by

Prakash Bettadapur

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

Department of Computing Science

Edmonton, Alberta
Spring, 1986

ISBN 0-315-30253-4

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Prakash Bettadapur

TITLE OF THESIS: EXPERIMENTS IN CHESS CAPTURE SEARCH

DEGREE : Master of Science

YEAR THIS DEGREE GRANTED: 1986

(Signed) [illegible]

Permanent Address:

1320, Nehru Nagar,
MANDYA, INDIA 571401.

Dated 21 March 1986

THE 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 Chess Capture Search** submitted by **Prakash Bettadapur** in partial fulfilment of the requirements for the degree of **Master of Science**

[illegible]
Supervisor

[illegible]

[illegible]

[illegible]

Date March 21, 1986

To My Parents

ABSTRACT

Capture search in chess is an example of a tree searching application. The capture analysis, an expensive part of any chess program, checks for quiescence at every leaf node of a depth limited game tree. A simple capture search is implemented with limited hardware support, to study which hardware and software aspects can improve search efficiency. Three experiments are discussed, specifically, the performance comparisons, order experiments, and the depth experiments study the effect of applying algorithms and heuristics to capture search.

Performance measurements suggest that the expensive part of capture search, move generation, can be made faster using bitmaps, but this improved performance may be lost if the moves cannot be provided to the search mechanism efficiently. Move ordering is important in search efficiency. In the order experiments, the capture search is done with different capture orders and their efficiencies are compared. The experiments indicate that ordering the moves using the heuristic of *first* capturing the *biggest pieces* by the *smallest pieces* leads to the most efficient search. In the depth experiments, the capture trees are searched to varying depths. The resulting error and the reduction in tree size for depth limited capture trees are compared to the correct values obtained with no depth limit. The experiments show that selecting a search depth leads to a *tradeoff* between error and savings. There is no optimum search depth with maximum savings and minimum error, so it is better to do capture search to an unlimited depth.

Acknowledgements

I would like to thank my supervisor, Dr. T. A. Marsland for his guidance and support, and for his patient reading of the thesis to suggest corrections. I would like to express my appreciation and thank Dr. J. Schaeffer for his constant encouragement and guidance.

My thanks are to the other committee members, Dr. W. Dobosiewicz and Prof. W. Joerg for their valuable comments on improving this thesis.

Finally, I gratefully acknowledge Pinky for her continual love and support, which made things much easier.

Table of Contents

List of Tables

List of Figures

Chapter 1
INTRODUCTION

Conventional computers are not suitable for some well defined specialized applications, because traditional machine architecture is designed for general computation. Computers are inadequate because they cannot support specialized data structures and the necessary control structures to handle such data. This causes a gulf between the architecture and the application [Mye82] and is the reason for inefficient implementation of many problems.

One approach to efficient problem solving is to build a machine with an appropriate architecture for each class of problems. Another approach is to build a microprogrammable machine and an emulator for each problem domain. Yet another method is to reduce the gap between the architecture on one side, and the High-Level Language and user's application on the other, by designing specialized primitives to handle complex components of a problem. The best solution might be a combination of these approaches.

Many other factors for example, the algorithms chosen, the heuristics used, and the techniques applied, also influence efficient problem solving. A discussion of these aspects and measurement of the influencing parameters help to make the application efficient.

Accuracy of solution must be considered in problem solving applications. Much effort may be necessary for an accurate solution. If some error can be tolerated, then problems can be solved with less expense. A knowledge of the resulting error can help in solving a problem easily, with less effort.

1.1. Tree Searching

Tree searching is the only method available for solving some important problems. Programs adopt many heuristics and techniques to limit the size of the search tree, since searching a large problem space can be expensive. Heuristics can be applied to guide the search towards a solution, as for example, in the traveling salesman problem. A city that is near is considered before a city that is far away, in solving for minimum total travel distance. Techniques can reduce the size of a search tree by adopting efficient algorithms. The alpha-beta algorithm applied to search a well ordered game tree cuts the size of the search tree significantly.

Game playing programs adopt tree-searching and have long been a popular area of Artificial Intelligence research. They are easy to implement since they are well defined in a formal, rule-bound manner. Many games can be considered to represent models of real world situations. They form an experimental domain where progress can be quantified, and any progress in research is available for display.

1.2. Capture search

In chess, programs search a depth-limited tree and evaluate the leaf nodes. The evaluation may result in wrong values if the nodes contain forcing moves. So the leaf nodes are checked for quiescence by limiting the search to only captures, checks, forks and threat moves. This forms the quiescence search. Some programs consider only capture moves for analysis. This capture search forms a computationally expensive part of the node evaluation in chess programs, usually constituting about 50% of total search effort [SIA83].

It is well defined and self terminating, but involves the complexities of move generation.

The aim was to discuss the hardware and software assists that can be provided for capture search. Implementing the capture search in hardware is desirable for better performance, but simplicity is also a factor that must be considered. Best capture ordering can be determined by performing the search in various orders and comparing their performances. To compromise the search effort with error in search, it is necessary to terminate the search to different depths and compare the error and savings values. Several experiments were conducted to address these issues in this thesis.

1.2.1. Hardware support

Game playing programs usually require a step towards the goal node as solution (like a "move" in chess), instead of a path from the start node to the goal-node (as in the traveling salesman problem). So programs search a depth-limited tree with the assumption that the evaluation of leaf nodes truly represents "nearness" to the goal node. Searching a deeper tree gives a more accurate solution (as seen in chess programs), so programmers demand faster execution to search deeper within the specified time constraints (like having to make a chess move every three minutes). Since the final problem solving is on hardware, faster execution requires more efficient hardware.

The structure of most computers is not well suited for chess, since the fundamental data and control structures are not supported. Many chess machines are built with specialized hardware. Although successful, they are complex and cannot be easily improved (e.g., BELLE [CoT82]). Some problems are caused by an inadequate understanding of the problem itself. Only a well defined application can be completely supported by hardware. The capture search is suited for such an implementation.

Effective and simple hardware support can be provided by supporting the expensive and complex part of tree searching, the move generation. By encoding moves of a piece from a square to all other squares of the chess board in an array of sixty-four bits, we can form a move bitmap. Simple logical operations on bitmaps can generate the moves. Such a design provides a simple alternative to complex move generation logic of special hardware chess machines. With the alpha-beta algorithm, more cut-offs are possible if the best move is examined first. The hardware chess machines generate and examine moves in some particular order, by having a complex priority circuit to provide the "next best move" for search. A simple mechanism for providing the next best move is possible by storing all branches that can be generated at a node, and examining them in the desired order. A large ROM stores every piece-pair combination, so the branch of the tree that must be generated next at a node is conveniently obtained.

With these considerations, a program for capture analysis was written in MC68000 assembler, and interfaced to the TinkerBelle* program. The performance is compared to TinkerBelle's capture search. Some problems and limitations in utilizing hardware for tree searching applications are explored and discussed.

* A chess program, developed by K. Thompson (Bell Laboratories, Murray Hill, NJ), which participated at the U.S. Computer Chess Championship, ACM National Conference, San Diego, CA, 1975.

When all the forcing moves must be considered in a hardware implementation of quiescence search, what aspects are involved? What hardware support can be provided? How can the hardware be provided for functions like node evaluation, move selection and move ordering in a typical tree searching application? These are some issues that are also addressed.

1.2.2. Effect of order on capture search

Move ordering is known to be important for search efficiency, since the alpha-beta algorithm produces many more cut-offs on a well ordered tree. In game trees, the value of the pieces and their positions must be considered in obtaining the best move order. Programs search capture trees in different orders; some consider the value of attacked pieces only in ordering the search, whereas others consider the difference in the value of attacked and attacking pieces. For example, BELLE [CoT82] examines the capture of biggest pieces by smallest pieces first, whereas some others [GEC67] prefer to examine the captures with highest material value difference captures first. It is not clear which is the desirable order, nor how the efficiency of search is affected. The pieces on the destination square of a capture move play a greater role since they alter the total material estimate of the node after the move. When the moves are generated in a random way, i.e., the value of attacked pieces are not considered in a move ordering mechanism (which is likely the case when the moves are generated for each attacking piece), it is interesting to quantify the search efficiency. Many other orderings are possible, and how they affect the search can be seen only by conducting capture search with different orders and comparing them.

The flexible move ordering provided by ROM data is used to search the capture tree with different orders and explore the effect of order on the efficiency of the search algorithm. These "order" experiments are performed on two sets of test positions and the results are discussed. The experiments clearly show that capturing the biggest pieces from smallest pieces provides the best order for examining the moves.

1.2.3. Search to different depths

When search is used as a strategy for problem solving, it is possible to look at all the possibilities and get an optimal solution. A partial search results in an approximate solution. A compromise with execution time and resources available may force one to live with some error. In capture analysis, the search should terminate in a quiescent position, but a depth-limited capture tree search results in evaluating non-quiescent positions, thus returning with some unknown error value.

Many competent chess programs search capture trees to different depths and compromise the effort spent and the error that can be tolerated. For example, Phoenix [Sch86] searches a capture tree to an arbitrary depth. The TinkerBelle program searches to only 8-ply deep, before returning the value of the subtree. Many microcomputer programs do only a static analysis of captures to estimate the quiescent value of a position [Kai85]. The experiments to quantify the error and the savings incurred in a search to different depths, help in designing capture search for future chess programs.

The error in searching a capture tree of depth d is defined as the ratio of the number of times the search returns an error value as compared with searching a capture tree of arbitrary depth, to the number of capture trees examined. The savings is the ratio of the saved nodes on searching a capture tree of depth d, to the total nodes searched on an arbitrary depth tree. A set of "depth" experiments have been performed on two sets of test positions and the results are discussed. Though the savings values obtained are different for the two test positions, the error values suggest that a deeper search is desirable for an accurate solution.

Chapter 2

BACKGROUND AND SURVEY OF TREE SEARCHING IN CHESS

2.1. Why Chess?

Chess has been an intriguing problem for researchers interested in machine intelligence and is an ideal problem for experimentation, since the game is clearly defined in terms of the allowed operations (the legal moves), and in the final goal. It is neither too simple to be trivial, nor too complex to obtain a satisfactory solution. Mathematically, the problem can be solved, but computationally it is intractable. The game tree for chess consists of over 10^{43} legal chess positions [Sha50] and at a rate of 10^6 nodes per second, it could take up to 10^{30} years to search. The problem is solved to the best approximation within a time constraint, like making a chess move in three minutes.

The type of trees that are looked at are for two person, zero sum, perfect information games, known as adversary trees. A game tree need not be an adversary tree, for example, in Go, a player may pass his turn with the opponent's permission, thus allowing him to make two or more successive moves. They need not be perfect information games e.g., Backgammon, where a move is based on the results of dice. They need not be two person, zero sum games, e.g., Bridge, where there can be collaboration. Thus chess forms special trees and this makes the search much simpler,

Currently available chess programs play much better than those in the 70's. Since Shannon's classic paper [Sha50] appeared in 1950, performance of chess programs has improved considerably. Research has resulted in better algorithms, techniques and representations. The computers are more capable, many run chess specific algorithms. A computer that runs six times faster usually adds one ply to the search depth of a brute force chess program, thus increasing its playing strength by nearly 100 Elo points [WeB85]. Master level programs like BELLE [CoT82], HITECH [EbP84] and CRAY BLITZ [Hya85] search a depth of eight to nine ply in a typical middle game, where there is an average branching factor of thirty-five. The annual computer chess championships provide a good measure of progress in the area.

2.2. Definition of terms

To map a chess problem to a tree searching application, it is necessary to describe some chess specific terms. These terms are used frequently in this thesis. Definitions of many other terms can be found elsewhere [AAB70, Ber74].

An *estimator or evaluator* for the terminal nodes of the search tree consists of a material component and a positional component. The material component is obtained by adding the (arbitrarily, but consistently) assigned values for the pieces on the board. The positional component is based on complex factors, such as, pawn structure, piece position and king safety. Each component's relative weightings vary, depending on the situation of the game and the strategy adopted.

A *capture* is a move from a node at depth d to a node at depth $d+1$ (a ply), in which there is a change in the material balance. A king is in *check* when he is attacked and is subject to capture in the next move. A *check mate* is a "game end" condition, in which the king in check cannot escape the attack.

A move by the opponent creates a *threat* on a piece, if the piece is forced to move from its present location. When two or more pieces are threatened by moving a piece to a key square, the threatened pieces are said to be *forked*. Berliner defines that, "A piece is *pinned*, if moving it would allow the capture of a more valuable piece of its own side, when such a capture would not otherwise be possible" [Ber74]. A piece is said to be *en prise* if it is currently under attack. A piece is said to be *hung* if it remains under attack even after a move.

A *capture search* is the search through the subtree in which branches are the capture moves and evaluation involves only the material balance. A *quiescence search* is more general in that, the subtree searched terminates only in a tactically "quiet" or "dead" position. Complex conditions like forks and threats may also be considered.

A *brute force* search is one in which all moves are considered to a depth D, which may be followed by a quiescence search up to a dead position. A *selective* search is one in which implausible moves are eliminated, and amongst the rest, only a few are selected for examination by the search algorithm.

2.3. Tree Searching in Chess

A chess program searches a tree whose branches represent moves and whose nodes represent positions. The program searches minimax trees like those produced by any two person, zero sum, perfect information game. The object is to find a best move to play, so the program chooses a path that can lead to the highest valued leaf node, under the assumption of best play by both sides. Searching the entire game tree is impossible because of its size, programs usually limit the search tree to a fixed depth. Terminal nodes are then evaluated to estimate their "nearness" to the goal node. Often, terminal nodes may contain forcing moves, which will be examined by a quiescence search.

Many algorithms can be applied to the search procedure of a minimax tree. The most important one comes from the *alpha-beta algorithm* [KnM75], because of its simplicity. The algorithm is directional, and the moves are examined depth first. Searching the best move first results in maximum cut-offs. This suggests that when moves are generated at a node, they must be sorted before being provided to the search algorithm. A knowledge of the preferred order may permit the generation of moves in that order and make the search more efficient.

2.4. Components in tree searching

There are three main components in tree searching; subtree generation, evaluation of the nodes, and examining them to find a variation. The alpha-beta algorithm (Algorithm 2.1) [CaM82] is illustrated below in a C/Pascal-like language to explain these components.

```
1.  alphabeta(p : position; a, b : integer)
2.  {
3.    m, i, t, w : integer;
4.    if (TERMINAL(p)) return (EVALUATION(p));
5.    w = GENERATE(p);          /* Successors p_i */
6.    m = a;
7.    for i = 1 to w do
8.    {
9.        t = -alphabeta(p_i, -b, -m);
10.       if (t > m) m = t;
11.       if (m >= b) return(m);  /* cut-off */
12.   }
13.   return(m);
14. }
```

Algorithm 2.1: alpha-beta

All the moves are generated by the function GENERATE, sorted and supplied to the search mechanism (lines 7 to 12). TERMINAL(position) is true if the position is a leaf node of the search tree. EVALUATION(position) returns the estimated value of the position. Programs use the *iterative deepening technique* [SlA83] to search, so that the successors can be evaluated and re-ordered before the next level is explored. Efficiency is obtained because the moves are "examined" in the preferred order. In Algorithm 2.2, NEXTSUCCESSOR has to "generate" branches in a suitable order to obtain cut-offs using the alpha-beta. Although, each of these versions maintains data structures that are specialized to the application, the first algorithm must also

```
alphabeta(p : position, a, b : integer)
{
        t : integer; p' : position;
        if (TERMINAL(p)) return EVALUATION(p);
loop:
        p' = NEXTSUCCESSOR(p);
        if (p' == nil) return (a);
        t = -alphabeta( p', -b, -a);
        if (t > a)   a = t;
        if (a >= b) return (a);
        goto loop;
}
```

Algorithm 2.2: alpha-beta for generating moves one by one

maintain all the branches at each node the algorithm has traversed. In the second case, a status word at each node can indicate what branch must be generated next.

Subtree generation mechanisms in software typically generate all the moves into a move-list. This approach has some advantages and disadvantages. The move-list can be generated in any convenient way, while a separate sorting phase can follow. It requires much less effort to generate all the moves "from a piece," than generating all the moves "to a square." The latter approach produces the moves in a desirable order (shown later in Chapter 5) while the former one can be too expensive to search (also illustrated in Chapter 5).

Generating the move-lists can benefit from parallelism in the representation. For example, when bitmaps are used, all the moves "from a piece" can be obtained with relative ease. These moves can be sorted and given to the search mechanism. Sorting adds complexity to the move generation process, since additional data structures are required to store move-lists at each node. Sorting is inexpensive compared to the cost of move generation (illustrated in Chapter 4).

When an ordered move-list is examined by a search mechanism, a beta cut-off can occur early, and terminate processing at that node. Much effort is wasted in generating moves that are eventually cut-off. However, when moves are generated one by one in an order that induces cut-offs, less effort is wasted.

Several other optimizations can be applied to move generation in software. Vector processing in machines like Cray [Ibb82] helps in vectorizing sequences of operations for move generation. Move-list generation requires the interaction of pieces with all the other squares of the chess board (sixty-four squares). The Cray computer supports sixty-four element vector registers and the control structures for handling vector data. By choosing appropriate data structures (like holding a piece of information corresponding to a square in an element of vector register), vectorization can be made more suitable for chess [Wen85]. CRAY BLITZ [Hya85], a chess program running on a Cray computer uses such techniques.

When a single processor handles both move generation and search, the problem of dividing the work, or optimally utilizing the hardware for move generation and search, does not arise. When move generation is done by chess specific hardware, generating all moves at a time into a move-list under utilizes such hardware. For example, during the search phase, the generated move-list is used and the move generator remains idle. There is also an extra cost associated with updating the board with each move, whereas generating and making a move at the same time is less expensive.

Move generation mechanisms in hardware are characterized by extracting the moves one by one. This technique has the advantage of not requiring support for complex structure like a list of moves. Sorting is eliminated by generating the moves in a desirable order. The implementation efficiency is improved by not generating the moves that are cut-off by the search algorithm. NEXTSUCCESSOR of Algorithm 2.2 must be able to generate the "possibly next best move" from the moves not yet generated at a node. To select the "next best move" from a set of all possible moves at a position, a hardware system must provide the following:

(1) A method for generating all possible moves at the same instant, to provide input to a priority circuit that can select the best move.

(2) A masking mechanism to disable the moves that are already examined and so are not inputs to the priority circuit, and

(3) A priority mechanism to extract the best of the available moves.

We study how these mechanisms are implemented in the existing systems.

BELLE [CoT82] uses the above technique for move generation. Its hardware system is an 8X8 array of combinatorial circuits. Each square has a transmitter and a receiver circuit. Each transmitter connects to its neighbors corresponding to the king's move, knight's move, pawn move, and manhattan and diagonal squares corresponding to the sliding moves. Each receiver has similar input connections. The priority circuit of BELLE extracts the best move from the move generator as follows:

> "The microcode asserts FIND_VICTIM. This causes each transmitter to activate signals corresponding to the piece on each occupied square. The receiver sections then activate the priority leads for *all* enabled

attacks. The priority tree finds the *highest* enabled attacked piece. The microcode latches the address of the attacked piece and then asserts FIND_AGGRESSOR. The addressed victim's transmitter radiates signals as a super piece. The priorities are inverted by the PLA so that the lowest valued aggressor on the selected victim is found" [CoT82].

The "next best move" is selected by

"After processing each move, the victim and aggressor are restored. The aggressor is disabled and next lowest priority aggressor is found until there are no more aggressors on this victim. The victim is disabled and all potential aggressors are re-enabled" [CoT82].

This helps in finding the next best victim. This process continues until all the victims are exhausted. Thus BELLE generates moves, in the order *biggest pieces being captured by smallest pieces* first. The disabling mechanism is by a sixty-four word stack, each entry of which is sixty-four bits long. Moves already seen are disabled and the next best move is generated.

The VLSI implementation of chess machine HITECH, uses a similar approach for move generation [EbP84]. It contains an 8X8 array of VLSI chips for move generation. As communication is expensive in VLSI, all the necessary data, e.g., board position, ROM data for move information and a context stack for disabling the examined moves, are locally stored in the chip. Each chip independently calculates the set of legal moves to its own destination square. The problem of selecting the best move is solved in two stages: first, each chip decides which of its moves is the best, and then a global voting procedure decides which of the 64 chips has the best move. The legal moves on each chip are ordered by the heuristic that lower valued pieces should be tried before moving higher valued pieces. Each chip then calculates a value of the highest priority legal move based on the type of moving piece, type of the captured piece (if any), and an estimation of the *safety* of the destination square. Safety information of the origin square is not used since it is not available in an efficient way, and is considered less important. The chips vote on the priority value, resulting in the selection of the best move.

BELLE and HITECH emphasize the "order" of move generation. BELLE considers only the material value of affected pieces for ordering. HITECH goes a step further in considering the safety of participating squares. Work in a later chapter quantifies how this extra consideration can influence the efficiency of alpha-beta algorithm.

Another design for a move generator as a VLSI chip [SPJ83] has the logic for move generation, but there is no support for ordering moves or saving context. All moves have to be generated for each position. Thus, this requires additional work to integrate the chip into a chess program.

Incorporating a hardware move generator to a search program is important in obtaining good performance. When moves are generated, extracted and provided to the search algorithm in an appropriate order, the node evaluation must also be available in a comparable time. BELLE achieves this by having two evaluations: a fast evaluation that just considers the material balance and the safety of pieces, and a slow evaluation that considers the pawn structure and ray evaluation. Fast evaluation is available in less than an instruction time and often results in a beta cut-off. Absence of hardware evaluator can result in the loss of performance, as illustrated in CHEOPS [MHG79]. CHEOPS derives its chess specific power from a Chess Array Module (CHARM), an 8X8 array of circuits. CHARM maintains the board data in piece list and square list, and S_SCAN and P_SCAN circuits to scan the squares. The module makes the next move in a *raster scan order* (the only distinctions in moves are noble captures, pawn captures and non-capture moves) and reports this to the processor, which then performs the node evaluation. Absence of move ordering to suit the search algorithm and not incorporating an evaluation function in hardware resulted in a performance loss in CHEOPS.

It has been reported that in ADVANCE 3.0, a ninety-six-bit high speed sequencer controls a sixteen-bit bit-slice processor (handling the search) and a sixty-four-bit TTL processor (handling chess specific operations) [WeB85]. Complex functions such as attack propagation and locating and counting bits on the bit board are available as operations. Optimum utilization of processor capacity by splitting the task into sixteen-bit and sixty-four-bit operations makes a good architecture for a chess machine.

BEBE's move generator, as described by Welsh, adopts a software approach. Here, move-list generation is microprogrammed to the processor instruction set so that it appears as a single instruction [WeB85]. This way, the requirement of complex logic for move generation and the global priority circuit for move extraction are eliminated. The same processor implements move generation, node evaluation and search, so the problem of integrating the search components does not arise.

2.5. Role of quiescence search

The size of the minimax search tree in a game playing program is controlled by limiting the depth. The leaf nodes of the search tree must be assessed by an evaluator. If the leaf nodes represent positions that are non-quiescent, the evaluation may not be accurate. This effect, known as "Horizon effect," is defined by Berliner:

> "A problem that exists in the tree searches to fixed depths which causes errors in terminal evaluation since not all terms in the evaluation function are driven to quiescence" [Ber74].

Quiescence search reduces the horizon effect by considering the forcing moves (like captures, checks, threats, forks and pins) from non-quiescent positions. As Kaindl points out,

> "The aim of a quiescence search is to find a quiescent position, which should be distinguished from a normal game tree search where the aim is to find a combination" [Kai85].

A quiescence search must terminate in a tactically "quiet" position, and not be depth limited. What moves to consider for implementation depends on the board position: whether the position has threat moves, or whether it has fork moves. Kaindl suggests that the static analysis by a program should guide a selective quiescence search [Kai85]. Typical implementations simplify this by considering only capturing and checking moves.

The capture search is computationally expensive, consuming about 50% of the total execution time, because it is executed at every leaf node. Slate and Atkin observe

> "Typically, the capture search constitutes about 20 to 70 percent of the total search. On rare occasions the search does blow up and this fraction reaches as high as 95 percent" [SlA83].

A study of the aspects of capture search implementation can help in reducing this fraction of computation time.

In some chess programs, a quiescence search is often replaced by a static analysis. Microcomputer programs typically adopt this technique [Kai85]. It is reported that CRAY BLITZ uses a complex static exchange evaluator in place of a full width capture search [WeB85]. The exchange evaluator does a fast evaluation of selective chess combinations without including unrelated captures elsewhere, and considers only the material balance. Even though this saves much execution time, it could lead to large errors.

A better alternative to static capture analysis is the dynamic capture search, where capture moves are made in succession, and the quiescent nature of a position is determined. In contrast to a static analysis, a dynamic search can expose side effects like hidden threats, justifying its implementation. The capture search is mainly an exchange-analysis. A major component is the material balance, although less subtle aspects such as safety and positional factors can also be considered. It is not clear what kinds of material exchanges are preferred, e.g., whether an exchange ordering involving difference in material value is preferred over the absolute value of the captured piece. The node evaluation function of capture search is well informed (material evaluation closely corresponds to actual material value of the node [Pea85]). The capture ordering function can use this information to implement the capture search efficiently, because the amount of material taken off the board at each capture, is known.

Game tree search programs produce shallow, bushy trees since they have a large branching factor and are depth limited. Capture search, on the other hand, produces arbitrarily deep, narrow trees, because there are limited captures at a node and the search is self terminating. Theorists have analyzed the application of alpha-beta algorithm on the game trees [Bau78, KnM75, Pea80]. Performance measurements of alpha-beta and its refinements on the game trees have also been made [CaM83, Fre85]. Work about the efficiency of applying alpha-beta on capture trees is not found in the literature. This thesis provides some of these performance measurements.

Chapter 3

IMPLEMENTATION OF CAPTURE SEARCH

3.1. Why Capture search?

Capture analysis is a computationally expensive part of a game tree search program because it is done at every leaf node. Implementation of capture search can expose the problems involved and lead to improvements. Results from capture search can be applied to other search applications.

Capture search builds a tree in which the branches are capture moves only, and forms a well defined subset of quiescence search. It involves the intricacies of move generation, but the evaluation function is simple and well defined. So it can benefit from hardware implementation. Capture search logically forms a separate block and hence can be implemented independently.

Generating captures one by one can be useful in capture search because they can be preordered in a preferred way even without sorting. Nodes of the capture tree differ by material balance values only, so the node evaluation function is well informed. This information can be used for preordering the captures. Hardware chess machines choose to generate the moves one by one, so capture search makes a good application.

3.2. Implementation considerations

Capture search forms a logically separate block for implementation. Its implementation is simple even if it involves move generation and node evaluation. Move generation is kept simple by using bitmaps, which are formed by encoding the moves of a piece from a given square in sixty-four bits. A capture is asserted when the accessed move bitmap has a "1" on the destination square. Move ordering is predefined by a ROM to help in selecting the "next capture." Status at a node of a capture tree is represented by a word capture state, so when the search returns to the node, next capture is generated easily. This mechanism also avoids the requirement for a global priority circuit.

Data structures were chosen to suit the TinkerBelle chess program. For example, board data is stored as a piece-list and also as a bitmap. Data for all the legal moves and information about the moves that must be generated at a node are stored in a ROM. An indexed access with offsets as the piece and square numbers is necessary to extract appropriate bitmaps from the ROM that has the legal moves. To program these operations as implemented in hardware, the capture search program is implemented in assembler.

3.3. Capture search implementation rules

A capture search consists of the following rules [Bea80]:

(1) All captures are searched.

(2) A position where the side to play has no captures is terminal, and the static material balance value is taken.

(3) At any node, if all the captures for the side to play prove to be losing, the

static material balance value is taken.
Rule (3) above corresponds to the consistency cut-off, and the capture search corresponds to a consistency search [Bea80].

Some rules specific to the implementation include:

(1) A king's capture indicates that the previous capture move was illegal; all captures that prevent the king's check are tried in the previous level of the capture tree. Other capture moves that do not take the king out of check are declared illegal, and are not explored. This produces more cut-offs in the search tree.

(2) Branches of the tree are cut-off, when the beta bounds are reached.

(3) When the pieces are considered in the biggest to smallest value of attacked pieces, an additional cut-off can be obtained because the value of the piece that can be captured at that node is not any better than that is already seen. This cut-off can be applied when the subtree is generated in a specific order (more details in Chapter 5).

(4) The capture search returns the material balance value of the quiescent position.

(5) Captures are generated one by one, so the effort is not wasted in generating moves that are eventually cut-off by the search algorithm.

3.4. Architecture of capture search

3.4.1. BOARD

The board is represented as a *piece-square* memory, where each piece is allocated a fixed memory location. If a piece exists on the board, the corresponding square number is written in its position, else the location is filled with an invalid value like 'FF.' An additional data structure is maintained as a bitmap of pieces (PXR) to handle the bit operations of the board data. A bit is set to "1" if a piece is present on a square, reset if it is not. Separate PXRs can be maintained for each color and type of piece if necessary. Incremental updating of the piece-square memory and PXR takes place when a move is made or unmade during the search.

3.4.2. Capture ordering ROM

The capture ordering ROM (CAP_ROM) stores the order in which the capture moves must be examined at a node. This is necessary because it helps in generating the "next best move" without having a global priority circuit. An alternative to this is to generate ~~all the~~ moves at a time and sort them, but it is simpler to generate one move at a time, if they can be preordered, and CAP_ROM helps this implementation. The status of move generation and search at each node is stored in a word as the CAPTURE_STATE, and produces an index into the CAP_ROM. A list of piece-pair combinations that can make up the potential captures are stored in CAP_ROM. Sequentially going through each piece-pair exhaustively tests the existence of a capture for the board position. CAP_ROM also stores other data like material gained by a capture (a property of the TO_PIECE), and type of FROM_PIECE (i.e., if it is a sliding piece or not or if it is a pawn). Corresponding to sixteen pieces, the capacity of CAP_ROM is 256 words for each side. CAPTURE_STATE forms an eight-bit word, the ninth bit (of a combined sixteen-bit word) represents the side to move. Thus CAPTURE_STATE directly indexes into the CAP_ROM.

3.4.3. Legal moves ROM

The bitmap of moves of each piece on each square are stored in the Legal Moves ROM (LEG_ROM). A bit is set to 1 in a move bitmap, if a piece can move to the square number represented by the bit, from a given origin square. This representation simplifies the move generation by replacing the complex hardware, like that used by BELLE to determine if a particular piece can move to a particular square. Entries of LEG_ROM are arranged so that a simple indexed access, with the piece number and the square number as the index, gives the corresponding move bitmap.

Some additional data is also arranged in a similar way. This is the bitmap of all squares that can attack any particular square, stored as the ATTACKED_BY bitmap. When a TO_PIECE is not on the attack path of any FROM_PIECE, a single test using the ATTACKED_BY bitmap helps in skipping the TO_PIECE, and a test through each FROM_PIECE is avoided.

3.4.4. ARRAY_LOGIC

A simple indexed access of an entry from LEG_ROM can yield a move bitmap, so this approach for generating the moves can be fast. Unfortunately, the moves of sliding pieces, that is, queen, rook and bishop moves cannot be generated so easily. With the bitmap representation, it is possible to know the destination squares of a sliding piece, from a given origin square, if there are no other pieces on the board. The actual squares where a sliding piece can move, depends on the board configuration (i.e., the location of blocking pieces). The information on a TO_SQUARE is insufficient to determine the legal moves, and it is necessary to know the occupancy of all the squares intervening the FROM_SQUARE and the TO_SQUARE.

Processors generally permit simple AND/OR operations on the bitmaps. When only the information on a TO_SQUARE is sufficient to determine a legal move or capture (for example, to determine a knight's move, it is enough to know that the destination square is empty or an any piece is present), AND/OR operations can be used to generate the moves. For sliding pieces, the presence of other pieces on manhattan and diagonal neighbors of a chess board must also be considered. On a bitmap representation, bits representing these neighbors are not adjacent, and the general purpose processors do not provide instructions to handle such operations on bitmaps.

When the moves of a sliding piece are generated in software, each square is checked to see if it is on the attack path of the FROM_PIECE. If the destination square is empty, the attack path is extended. If the destination square has a friendly piece, the attack is blocked. If an enemy piece is present, an attack is asserted on that square, but the attack path is not extended. These operations can be done in parallel and fast, when implemented in hardware.

One approach to solve this in hardware is to use complex logic, like that used by BELLE. A simpler approach is to use the bitmap representation and some simple logic. Since the bitmap move of a sliding piece, when no other piece is present on the board, is available from LEG_ROM, it can be used with friendly pieces and enemy pieces bitmaps, to yield a bitmap of moves valid for a given board position. A sequence of simple logic operations on an array of bits (of a bitmap) results in generating a move bitmap for a sliding piece. This is the ARRAY_LOGIC, and the algorithm for its implementation is given in Figure 3.1.

```
1) Send out "1" in all eight directions from FROM_SQUARE
2) If "1" comes in (COME_IN) then
    If "1" is on that square (from LMREG (bitmap move)) then
        If NO_PIECE then
            ASSERT "1";
            SEND_OUT "1";      /* Move         */
        else if ENEMY_PIECE (EPXR) then
            ASSERT "1";
            SEND_OUT "0";      /* Capture      */
        else if FRIENDLY_PIECE (FPXR) then
            ASSERT "0";
            SEND_OUT "0";      /* Blocker      */
    else
        ASSERT "0";    /* Bits adjacent to move bits */
        SEND_OUT "0";
  else
    DISABLE input to ASSERT;
    SEND_OUT "0";            /* Affects squares set previously */
```

Figure 3.1 ARRAY_LOGIC - Logic for legal moves of Sliding Pieces

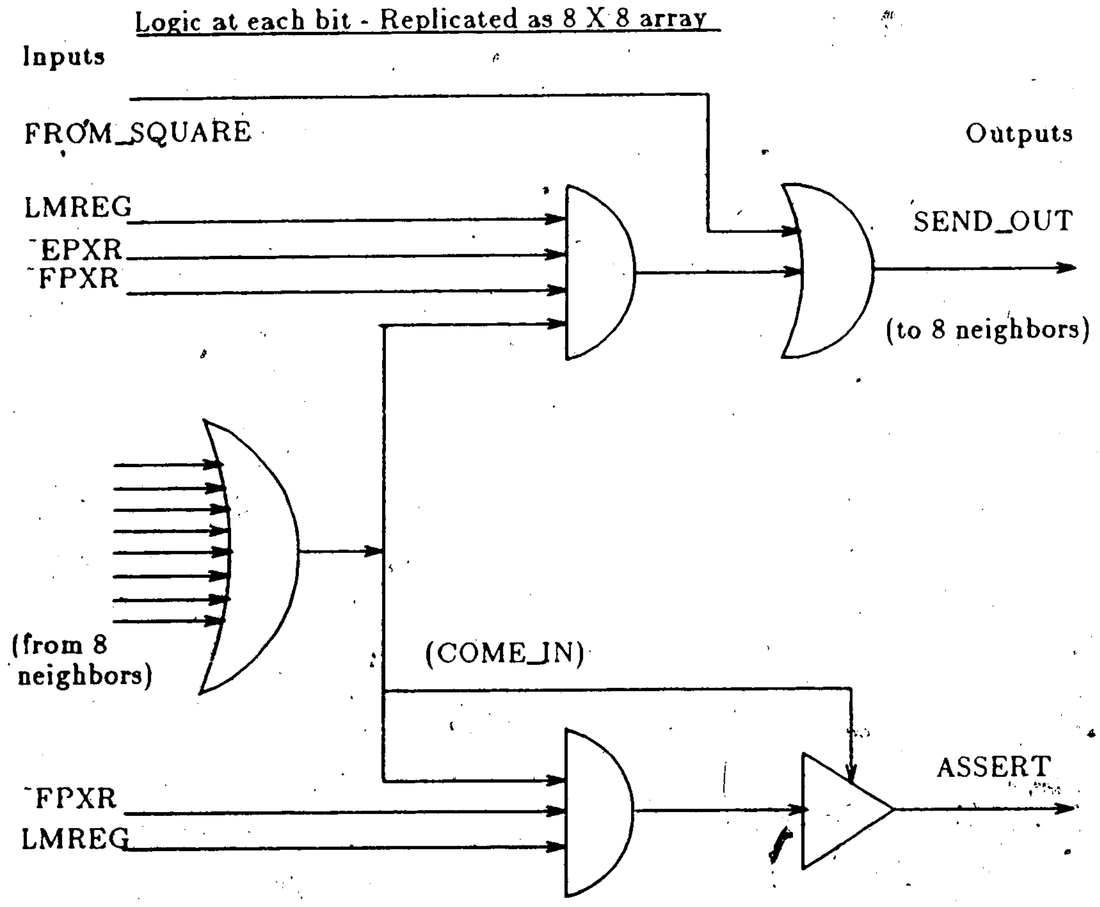
The circuit diagram implementing ARRAY_LOGIC, given in Figure 3.2, shows the logic for one bit. Each signal for the circuit is sixty-four bits long (corresponding to sixty-four bits of a chess board). SEND_OUT is connected to the eight neighbors corresponding to the manhattan and diagonal squares (logical neighbors on a chess board). EPXR (a bitmap for Enemy Piece eXistence Register) and FPXR (Friendly Piece eXistence Register) are given as inverted signals for this implementation. Data for EPXR and FPXR can be input earlier and latched in the chip. This data is altered only when there is a capture or a move. An input from LMREG (Legal Moves REGister) will ASSERT the appropriate bits on the output bitmap, which forms the legal moves for a sliding piece. This output bitmap can be handled similar to a bitmap obtained from LEG_ROM for non-sliding pieces.

In the capture search program, the ARRAY_LOGIC is simulated in a routine SLIDING. The operations are similar to that given in Figure 3.1. The logic of the circuit corresponding to one bit has been simulated and tested using RNL [Not84], a logic simulator. The hardware is not implemented.

ARRAY_LOGIC requires many input/output connections. By using a two bit "mode-select" for handling different input data, sixty-four pins can handle the sixty-four bit input/output signals. A chip that implements the ARRAY_LOGIC can be interfaced to a processor that handles bitmap move generation, to ease the task of generating sliding moves.

3.4.5. Stack management

Tree searching requires stack management for storing the data at the nodes that will be used when the search returns to the parent node. The volume of information stored depends on the problem and the search algorithm chosen. One approach for stacking is to store all the data on the same stack with only one pointer maintaining it, which is convenient when pro-



¯Inputs and outputs are 64 bits.

Figure 3.2 ARRAY_LOGIC

grammed in software. In hardware implementations, maintaining separate stacks is not hard and is desirable since their structure can be modified to suit the data. In the implementation on a MC68000 processor, the instruction set supports stack management, but if hardware is built, stacks can be constructed by shift register controlled memory.

For capture search, the information stored at each node include the moves made (represented by the piece and square numbers of origin and destination squares), the status at each node, the alpha and beta bounds. In the implementation, the data stored on the stack consist of (1) the move made: FROM_PIECE, FROM_SQUARE, TO_PIECE and TO_SQUARE, (2) the alpha-beta window, and (3) CAPTURE_STATE, MATERIAL_BALANCE and some flags.

3.4.6. CONTROL

Figure 3.3 illustrates a schematic block diagram for the capture search, with blocks in the diagram representing the different data and control structures. The status of search, shown by the block "state," produces an index into CAP_ROM, to give a TO_PIECE-FROM_PIECE combination. FROM_PIECE, with its square number retrieved from BOARD produces an index into LEG_ROM to yield a move bitmap. A logic operation of the square number location of TO_PIECE and the bitmap determines a capture. To generate a bitmap for sliding moves, ARRAY_LOGIC (in this implementation, the routine SLIDING) takes the Piece Existence Register (PXR) and the move bitmap as input. A logical "test bit" operation of the TO_PIECE's square number with the sliding move bitmap asserts a capture for sliding pieces.

On a capture, CONTROL stacks the information, makes the move, updates the board and initializes the capture state. The search starts at a successor node. When all the captures are searched on a level, or a cut-off occurs, CONTROL unstacks the information, unmakes the move, updates board data, to resume operation at a higher level. When the analysis is complete, CONTROL returns the result of search. The recursive search algo-

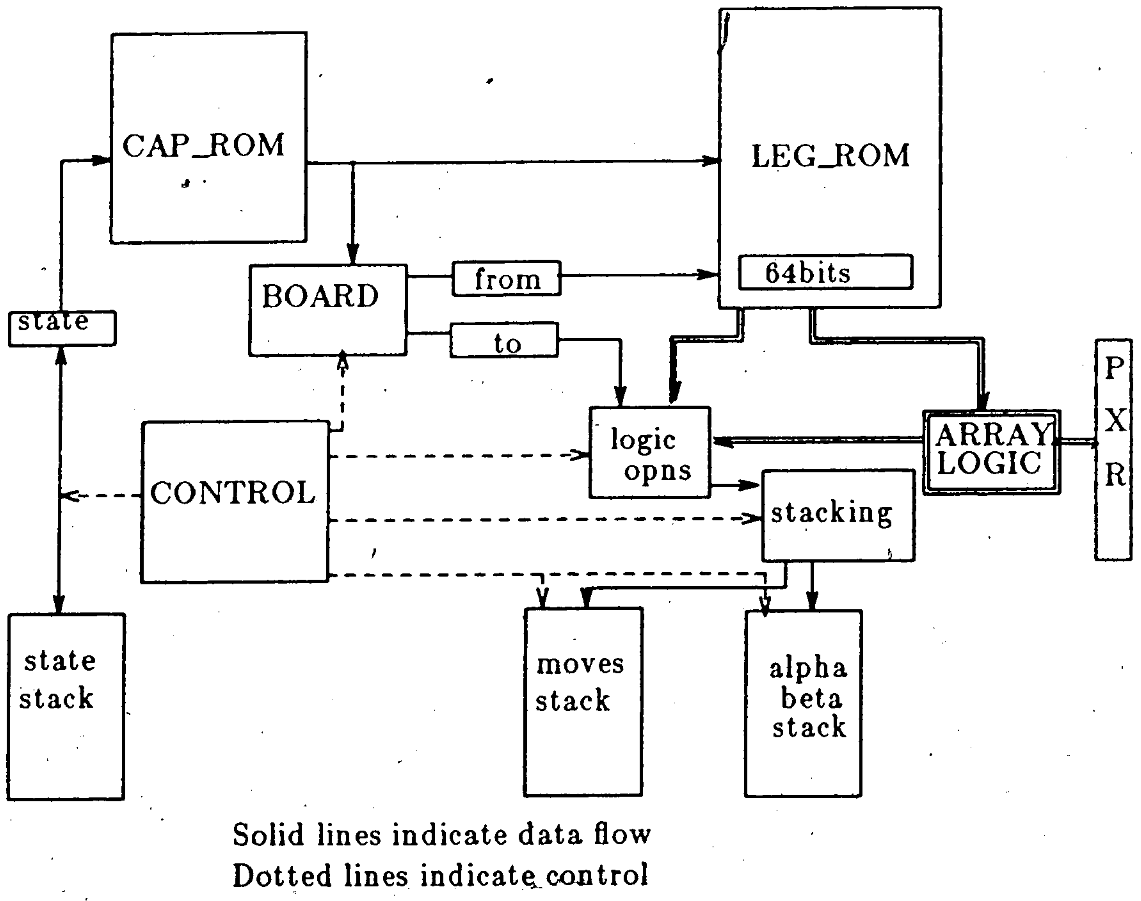


Figure 3.3 Block Diagram for Capture Search

rithm is mapped to the hardware as a non-recursive algorithm. In a hardware design, various blocks can operate in parallel, but in this simulation, the operations must be sequential. The non-recursive algorithm used in the implementation is given in Figure 3.4.

```
CAPANAL:
        Load address registers
        Initialize the flags
        Initialize with alpha, beta = [ 0, 127 ]
GO_DEEP:
        Clear other registers.
NXT_CAP:
        If no more captures, goto NXT_HIG_LEV.
        Get one piece-pair combination of CAP_ROM.
        If any of the pieces do not exist, goto NXT_CAP.
        Get the bitmap moves of FROM_PIECE from LEG_ROM.
        If it is a sliding piece, generate sliding moves.
        Test with TO_PIECE for capture, if so, goto CAPTURE.
        Else goto NXT_CAP.
        If any error, goto ERROR.
ERROR:
        Set flag; goto EXIT.
CAPTURE:
        Compute material balance.
        If King is in check, goto KING_CHK.
        Store move, make move, update BOARD, update PXR.
        Store alpha, beta.
        Store capture state and material balance.
        Negate and interchange alpha, beta.
        If no captures, update alpha with Material Balance
                (or its negation for MIN nodes).
        If alpha >= beta, goto NXT_HIG_LEV.
        Else goto GO_DEEP.
KING_CHK:
                /* Don't make move or store any data */
        Update alpha with material balance
                (or its negation for MIN nodes).
        Goto NXT_HIG_LEV.
NXT_HIG_LEV:
        If top of stack, goto EXIT.
        Retrieve move, unmake move.
        Update the return value.
        If alpha >= beta goto NXT_HIG_LEV.
        Restore capture state and material balance.
        Goto NXT_CAP.
EXIT:
        Return alpha as RESULT.
```

Figure 3.4 The Non-recursive Algorithm used in Capture Search

3.5. Implementation on the MC68000 processor

Bitmaps can be handled better, if the application is microprogrammed. But, to test the capture search with a chess program (the TinkerBelle program), it was decided to implement the capture search routine on the MC68000 processor of a SUN work station. The MC68000 supports thirty-two-bit operations, so a bitmap (of 64 bits) needs two instructions. The control for move generation is arranged to suit the instruction set of MC68000. For example, special instructions for extracting the bits from a bitmap are not available for MC68000, and so the SET, SHIFT and TEST bit instructions are used. The processor also supports eight-bit and sixteen-bit operations, which are helpful in implementing the search. Stack management is well supported with special instructions available for the purpose. The indexed operations to access the data for moves and capture states are also supported on MC68000 efficiently. This capture search implementation is referred to as CAPANAL (capture analyzer) during the discussions in later chapters.

3.6. Limitations of the Implementation

The main drawback is the way in which the moves are generated and provided to the search mechanism. In an attempt to generate the moves one at a time and eliminate the wasted effort in generating moves that are cut off by search, CAP_ROM is formed as a move arbitrator. But, looping through every piece-pair using CAP_ROM to check if a capture exists, takes about 80% of the execution time (Chapter 4).

Inherent parallelism in the bitmaps is not used since the moves are generated one by one. This forms another defect of the implementation. There is no parallelism in the operations shown in the block diagram. All operations in the program are sequential and a single processor generates the moves and implements the search.

Pawn promotions and en passant captures also result in a change in material balance, so it is logical to consider them in capture analysis. But, pawn promotions are not well defined, because the value of material changed depends on the type of piece to which the pawn is promoted. En passant captures can be considered only after special pawn moves that require the maintenance of extra data structures. These increase the complexity of implementation and do not fit into our simple model.

Some capture search programs include the moves that evade checks. Infinite checking sequences may result sometimes, where some artificial criterion must be applied to terminate the search. This violates the self-terminating feature of capture search and makes the implementation difficult. In CAPANAL, checking moves are considered by assigning a high value to the king, and returning that value for a king's capture. This is not a desirable approach since only the re-capture moves that take a king out of check are considered. To include the moves that evade checks, it becomes necessary to generate all the moves. Using the capture generation mechanism as discussed earlier, requires a CAP_ROM of capacity 16X64 words for generating all the moves, with each entry providing the move of each piece to every square. That would be impractical, since looping through each entry of CAP_ROM at every node, causes a large delay.

Sometimes, a quiescence search is adopted instead of a capture search, so all the forcing moves may be included. If threats, forks and pinned pieces must be considered, it is difficult to predefine the contents of CAP_ROM, since the number of branches from a node and the order for move generation to suit the search depends on the position. So the approach described in this chapter cannot be adopted in that case.

Bitmap move generation is considered to be a time efficient, storage expensive method [Cra84]. On the main frames, sufficient storage is available, so it is not a problem. The technique can be time efficient only if suitable instructions are available to extract the moves (destination squares of moves are available as set bits in a bitmap). Deciding "which move to generate next?" is solved by using a CAP_ROM, but this introduces a delay in looping through each combination.

3.7. Limitations of hardware for tree searching

Capture search, although considers only well defined moves and involves only the material balance for node evaluation, has limitations for an effective implementation. For a game tree searching program, there are more serious limitations, since its components are less defined. Only the task of *move generation* is well defined (except for the special moves), and the hardware chess machines generate the moves in a fast and efficient way. A *move selection* may be necessary, to limit the size of the search tree and permit a deeper search for more accurate results, but no good move selection mechanisms are known, not only for hardware, but also for software implementations. The *order* for generating the moves plays an important role in the search efficiency. Move generation mechanisms adopt some simple heuristics for move ordering, whereas more sophisticated algorithms could result in a more efficient search. A *node evaluation* is necessary at each leaf node, to determine the best move, but what features must be considered in such an evaluation are not known [Mar85b]. Hardware chess machines implement only a few, well defined features; Improvements in the quality of game is possible only if more features are implemented in hardware. Every leaf node must be checked for *quiescence*, to prevent blunders resulting from applying node evaluation on non-quiescent positions. The types of moves to be considered for an ideal quiescence search (like forking moves, threat moves and moves involving pinned pieces) are difficult to identify in a strict hardware implementation.

In addition to implementing each component of a chess program, it becomes necessary to incorporate and integrate the components properly into the chess program. An efficient move generation does not yield any benefits, if the node evaluation is not available in a comparable time, to the search mechanism. Therefore, it is much harder to implement a game tree searching application mechanism in hardware.

Chapter 4

PERFORMANCE COMPARISONS

The performance of the CAPANAL routine is compared to TinkerBelle's capture search. The programs are profiled to determine the time spent in each part of the program. CAPANAL was improved based on the results obtained from the profiler. This chapter discusses these details.

4.1. Test positions

The comparisons are made using two sets of test positions. The first, referred to as the KOPEC1 positions, is a set of twenty-four positions, and has been extensively used to evaluate chess program performance [KoB82]. The second set, referred to as the KOPEC2 positions, has an additional twenty-five positions [KNY85]. TinkerBelle is used for alpha-beta search to two-ply and three-ply deep. The collection of terminal nodes from these searches provides over 35000 non-quiescent positions from the first set, and over 45000 positions from the second set. These 80000 positions are used as the test set of positions for the experiments.

4.2. Measurements

A good measure of search efficiency is to determine the nodes searched per unit time. For capture search, the number of capture nodes (here after referred as Cnodes) searched per unit time are compared. The UNIX* system call *times()* is used to obtain the elapsed execution time. Cnodes examined per unit time are obtained by dividing the total Cnodes by the total time.

* UNIX is a trade mark of AT&T Bell Laboratories

The execution time returned from the system routines may vary depending on the load conditions, so these values are approximate. The results have been averaged over a large set of positions, thereby removing any errors.

Another interesting measurement is the time spent in different parts of the capture search routine. This profiles the behavior of the program, and helps in studying any defects in the implementation. One way to measure this is by calculating the time between the entry and the exit of each part of the program. Another approach is to measure the time spent in each part of the program statistically. A profiler that uses the second approach has been implemented.

4.2.1. Profiler

The profiler uses the *signal()* facility of the UNIX system. It sets up an interval for interrupting the processor. The capture search sets specific values on a flag when it is in different parts of the program. At every interrupt, the profiler checks the flag to determine which part of the program is currently being executed and increments a corresponding count. When the execution completes, the counts give a relative measure of the time spent in each part of the program.

4.3. Results

4.3.1. Initial performance of capture search

The CAPANAL routine, written in assembler was expected to do much better than TinkerBelle's capture search, but on the contrary, its initial performance was inferior. This is illustrated in the second and third columns of Table 4.1. On the KOPEC1 positions, TinkerBelle examined 2.48 Cnodes per unit time, while CAPANAL could examine only 2.28 Cnodes per unit time. Similar, but higher results were obtained with the KOPEC2 positions.

4.3.2. Improvements

A major improvement in execution efficiency can result if the part of the program causing the maximum delay is improved. The profiler showed that looping through each of the 256 piece-pair combinations of CAP_ROM, consumed 80% of the execution time. This looping may not be a problem in a true hardware implementation, since the CAP_ROM is usually replaced by a complex priority circuit that can extract the next best move. However, here, an improvement was made by checking every TO_PIECE, if it was on the attack path of any FROM_PIECE, using the ATTACKED_BY bitmap. If the FROM_PIECE was not attacked by any TO_PIECE, sixteen combinations were skipped. With this improvement, the CAPANAL routine could examine over 4 nodes per unit time, as shown in the last column of Table 4.1.

Cnodes/unit time	TinkerBelle	Old version	New version
KOPEC1 positions	2.48	2.28	4.07
KOPEC2 positions	2.86	2.83	4.47

Table 4.1 Performance Comparison

4.3.3. Profiling capture search

TinkerBelle's capture search was profiled to determine what parts of the program were expensive. The results showed that the move generation took up to 80% of the total execution time. Sorting, recursive calls, making and unmaking moves took only 20%. This suggests that move generation is expensive and must be made more efficient to improve the overall performance. Move generation could be made simpler by generating moves for each FROM_PIECE, and efficient by taking advantage of parallelism in bitmap representation. An *inexpensive* sort can reorder the moves in a desirable way.

The improved CAPANAL routine was also profiled to determine how much time was spent in each part of the program. The results are shown in Table 4.2. "Looping" corresponds to the time spent looping through the piece-pair combinations checking for the existence of pieces, "Move Gen" corresponds to the time spent in accessing appropriate bitmaps to assert a capture, "Slide Mov" corresponds to the time spent in generating the sliding moves, and "Stacking" to the time spent in making and unmaking moves. The KOPEC2 positions spend more time in move generation and less in looping. This suggests that these latter positions have more captures and are more complex.

Percentage	Looping	Move Gen	Slide Mov	Stacking
KOPEC1 positions	67%	23%	5.5%	4.5%
KOPEC2 positions	61%	32%	3.7%	3.3%

Table 4.2 Profiler results

TinkerBelle's capture search spent 80% of its execution time in move generation, whereas the new CAPANAL routine spent less than 30% on average. This is a significant improvement in terms of the efficiency of move generation. The total time spent by CAPANAL for capture searching was 68% to that spent by TinkerBelle.

(1) Efficient SHIFT instructions
(2) SIXTY_FOUR_BIT operations
(3) WHICH_BIT_SET operation
(4) FIRST_BIT_SET operation
(5) COUNT of set bits

Figure 4.1 Desirable instructions on MC68000 for efficient implementatio

Move generation using the bitmaps can be more efficient, if finding out what moves must be generated is inexpensive (looping through the entries of CAP_ROM is seen to be expensive). Move generation on the MC68000 can be better, if more suitable instructions to handle the bitmaps are available. For example, shift instructions are frequently used in the basic loop of the implementation. Each shift operation takes $8+2n$ clock cycles to move n bits. With $n=8$, each shift can take 24 clock cycles, whereas other register operations take only 4 to 6 clock cycles. Figure 4.1 lists some desirable instructions on the MC68000 that can help in manipulating bitmaps better.

Chapter 5

EXPERIMENTS

This chapter discusses the results of two experiments. By conducting the search with different capture orders, the order experiments quantify the efficiency obtained using the alpha-beta algorithm on capture trees. In the depth experiments, the resulting error and the reduction in tree size for depth limited capture trees are compared to the correct values obtained with no depth limit. The order experiments use the CAPANAL routine; the depth experiments use TinkerBelle's capture search.

5.1. Measurements

The number of bottom positions (NBP) is a common measure of search performance and is said to reflect the total nodes of the search tree [CaM83]. This may be true for game trees where all the bottom positions are leaf nodes, and are at the same depth. In capture trees, where the search is terminated at an arbitrary depth, only a count of all the nodes in the tree represents the true cost of search. Therefore, total capture nodes is taken for a measure of performance in these experiments.

A set of twenty-four Bratko-Kopec positions [KoB82], referred to as the KOPEC1 positions, and an additional set of twenty-five positions [KNY85], referred to as the KOPEC2 positions, are considered for the experiments. TinkerBelle uses its move ordering mechanism for iterative ***alpha-beta search*** to two-ply and three-ply on these forty-nine positions. This initial search results in over 80000 non-quiescent terminal positions, and forms the test suite for the capture search experiments.

PRIMARY ORDERING BASED ON THE VALUE OF TO_PIECES:

Order 1 -- Capture of big pieces first
Best order of from pieces

Order 1a - Same as Order 1, with an extra cut-off for nodes

Order 3 -- Capture of smallest pieces first

Order 4 -- Capture of big pieces first
Capturing pieces in king to pawn order
All captures by king, followed by queen, etc.

Order 7 -- Big White pieces captures first
Small Black re-captures first

Order 8 -- Small white pieces captures first
Big black re-captures first

PRIMARY ORDERING BASED ON MATERIAL VALUE DIFFERENCE

Order 2 -- High material value difference captures first

Order 5 -- Equal pieces capture first
Good captures: positive material value difference first
Bad captures: negative material value difference last

Order 6 -- Good captures: positive material value difference first
Bad captures: negative material value difference next
Uninteresting equal captures last

Order 9 -- Highest material value difference on white first
Smallest material value difference on black first

Order A -- Smallest material value difference on white first
Highest material value difference on black first

Order D -- Lowest material value difference captures first

PRIMARY ORDERING BASED ON THE VALUE OF FROM_PIECES

Order B -- Captures "from small pieces first" -- Random ordering
All captures from pawns, then all from knight, etc
Captured pieces in king to pawn order

Order C -- Captures "from big pieces first" -- Random ordering
All captures from king, then all from queen, etc
Captured pieces in king to pawn order

Figure 5.1 Capture orderings for experiments

5.2. Order experiments

Move ordering is important in searching the game trees, since alpha-beta applied to a well ordered tree induces more cut-offs. As discussed in Chapter 2, there are two approaches to generate an ordered tree. One approach is to generate all the successor nodes and order them by sorting. This approach provides the moves in either the best or the worst order, but is not suitable for the order experiments considered here. These experiments consider more than these two cases, so they require a preordering for captures. In capture search, the exact change in the material value of the node on making a move is known, since the material evaluation function is well informed. CAP_ROMs are constructed for each capture order and tested with the CAPANAL routine, to measure their performance.

Figure 5.1 lists the capture orders considered in this analysis, which are grouped into three categories.

(1) Primary order based on the value of to-pieces. Capture orders 1a, 1, 3, 4, 7 and 8 are included here. The value of the piece on the destination square of a capture move, plays a major role in determining the desirable capture order, since removing this piece changes the material balance at the node, by the value of the piece.

(2) Primary order based on the material value difference. Capture orders 2, 5, 6, 9, A and D are grouped here. The piece-pair combinations considered here have the ordering based both on the from-pieces and the to-pieces.

(3) Primary order based on the value of from-pieces only. Capture orders B and C are grouped here.

The capture orders are numbered in the sequence in which they were generated and the experiments were conducted.

The performance of CAPANAL with some orders were compared to the performance with the others. There are certain comparisons that are of interest and thus require some discussion. They are listed in Figure 5.2. Comparison 4 and comparison 6 are not discussed since their results were not found interesting. They are only mentioned for completeness.

Comparison 1: Orders 1,1a and 2.
Determines the most desirable type of capture.

Comparison 2: Orders 1 and 4.
Shows the effect of from-pieces on capture order.

Comparison 3: Orders 1-3.
Alpha-beta on well ordered and worst ordered trees.

Comparison 4: Orders 1-7-8-3
Does order of capture benefit from which side starts first?
This comparison is not discussed.

Comparison 5: Orders 2-5-6
How do zero material difference captures effect search tree?
Are pieces placed to result in a chain of captures?

Comparison 6: Orders 2-9-A-D
Similar to comparison 4, but for material value difference.
This comparison is also not discussed.

Comparison 7: Orders 1-B
Benefit of sorting a generated move-list
and searching, to searching from-piece order.

Comparison 8: 1a-1-2-4-5-6-B-C
A graph with these orders.
Provides an overview of different orders.

Figure 5.2 Comparisons

5.2.1. Comparing orders 1 and 2

The most popular order for capture search and move generation is to capture the "Biggest pieces from smallest pieces first" [CoT82,EbP84]. Some people also consider the material value difference between the capturing and the captured pieces for ordering [GEC67,Mar85a]. For example, the capture of a rook by a pawn is considered before the capture of a queen by a queen. This ordering is considered better, based on the assumption that the boards would be set up in such a way that a capturing piece will also be captured, so considering those with high material value difference first might result in an early cut-off.

Capturing the biggest pieces first can benefit from an additional cut-off. Since it is known how much material will be added on the remaining captures at a particular node, it is easy to decide whether to proceed with the capture analysis or cut-off (Algorithm 5.1). This consistency cut-off can be applied earlier for order 1a since the captured pieces are preordered in a desirable way.

```
If (value of piece captured < material balance already lost)
then
    { Now know remaining captures are no better;
      Return to higher level }
else
    { Assert capture; Add material balance;
      Continue with capture search }
```

Algorithm 5.1: Extra cut-off for order 1a

Order 1 is the capture analysis with the biggest captures first, but without the extra cut-off (Algorithm 5.2). This order provides a true comparison of the effect of only alpha-beta cut-offs on the search order, since the extra cut-off cannot be applied on some capture orders.

```
If (value of piece captured < material balance already lost)
then
      { This capture is not sufficient;
        Try other captures on the same node }
else
      { Assert capture; Add material balance;
        Continue with capture search }
```

Algorithm 5.2: Normal Case with no extra cut-off

In capture order 2, the piece-pair combinations with the higher material value differences are considered first.

Table 5.1 compares the nodes in the capture tree for the two sets of test positions. The capture trees built with capture order 2 result in 40% to 80% larger trees compared to order 1. With the additional cut-offs on order 1, it is possible to build capture trees 35% smaller (order 1a). Order 2 analyzes the captures with the human way of thinking, by predicting that a capturing piece is likely to be captured in the next move. This is possible if the captured piece is defended, but may not be the case in the millions of nodes analyzed by the program.

		Order 1a	Order 1	Order 2
KOPEC1 positions	Total nodes	309729	481383	685196
	Ratio	0.643	1	1.423
KOPEC2 positions	Total nodes	504002	846743	1526170
	Ratio	0.59	1	1.802

Table 5.1 Comparison 1: Comparing orders 1a, 1 and 2

5.2.2. Effect of from-pieces on the best order

Pieces on the destination square of a capture move (to-pieces) play a greater role in ordering the captures, since they alter the material balance of a position after the move. How do the capturing pieces (attacking pieces) affect the efficiency of search? This is discussed here, in Comparison 2.

Capture order 1 corresponds to captures *from smallest* pieces to biggest pieces first. Capture order 4 corresponds to captures *from biggest* pieces to biggest pieces first. The to-pieces are arranged as biggest to smallest pieces in both the cases. The results of capture search for the two sets of test positions with these orders are shown in Table 5.2. The capture order 4 builds about 20% to 35% larger trees than order 1, so order 4 has a lower search efficiency. The results also suggest that a capturing piece is more susceptible to capture on the next move, than other squares. This provides the logical reasoning for some static exchange evaluators (for example, CRAY BLITZ), that have a fast evaluation of selective chess combinations, and do not analyze captures elsewhere on the board [WeB85]. The results also imply that the destination squares are less safe, and can have some effect on move ordering. For example, HITECH considers the safety of the destination squares for its move ordering mechanism.

		Order 1	Order 4
KOPEC1 positions	Total nodes	481383	659229
	Ratio	1	1.369
KOPEC2 positions	Total nodes	846743	1054186
	Ratio	1	1.245

Table 5.2 Comparison 2: Comparing effect of attacking pieces

5.2.3. Efficiency of alpha-beta with worst order

It is known that the worst case order for alpha-beta search results in the largest tree, and corresponds to a minimax search. For game trees of an average uniform branching factor and fixed depth, the size of the minimax tree can be estimated. Efficiency of alpha-beta on such trees can be easily seen. Capture trees are self terminating, with non-uniform branching factor and arbitrary depth. Therefore, it is much harder to estimate their size. By searching the tree with the worst order, it is possible to measure the maximum size of the capture trees, and compare the efficiencies obtained from an alpha-beta algorithm.

A set of five simple positions is considered for measuring the efficiency of alpha-beta on worst order capture trees. Table 5.3 compares the nodes searched for the best order and the worst order. The positions that do not have many captures are considered here for analysis, since the worst order for search in more complex positions produced much larger trees and did not terminate even after two or three days of computation. So these results give only a best case estimate of the inefficiency of alpha-beta on badly ordered trees, i.e., searching a capture tree generated with a bad order ~~can~~ be at least this inefficient.

		Order 1	Order 3
A sample of 5 positions	Total nodes	13063	51974666
	Ratio	1	3978.8

Table 5.3 Comparison 3: Worst case alpha-beta

5.2.4. Comparing orders 2, 5 and 6

This comparison lets us to assess the effect of the role played by the material value difference of to and from-pieces. When analyzing the capture trees, it is generally felt that the captures with equal valued from and to-pieces would only extend the depth of the capture search tree, without inducing extra cut-offs, or providing material benefits. This suggests that equal captures must be considered last as in capture order 6. Another variation of the constituents of capture order 2, i.e., considering captures with equal valued to and from pieces before the positive material value captures, forms the capture order 5.

Table 5.4 tabulates the total of searched nodes for orders 2, 5 and 6. The results show that the capture order 2 is much better than the other two. Search efficiency is better when positive material value difference captures are considered first, followed by captures with equal valued to and from pieces, and last, the negative material value difference captures. Orders 5 and 6 are seen to be inefficient, so this comparison suggests that they are not worth considering.

		Order 2	Order 5	Order 6
KOPEC1 positions	Total nodes	685196	1708750	4153507
	Ratio	1.423	3.55	8.628
KOPEC2 positions	Total nodes	1526170	2816330	14090611
	Ratio	1.802	3.326	16.64

Table 5.4 Comparison 5: Comparing orders 2,5 and 6

5.2.5. Comparison 7: Benefit of sorting moves

Move-list generation is convenient when all the moves for a particular from-piece is generated, and is repeated for all the from-pieces. If the from-pieces are chosen in pawn to king order, we obtain the capture order B, which is a random order in terms of to-pieces. Capture order 1 is generated in the best order of to-pieces, thus requiring a complex method for move generation, or requiring a sorting phase after move generation.

A comparison between the two shows that capture order 1 is nearly 70 to 150 times more efficient than capture order B as in Table 5.5. For moves to be in the best order of to-pieces requires that they must be generated one by one for each from-piece and to-piece. Generating all the moves for a from-piece can be convenient because the parallelism in the bitmaps can be utilized. Sorting the moves generated from the latter order can provide the moves in the best order of to-pieces. Since sorting is inexpensive (results of Chapter 4) and capture order B for search is expensive, greater search efficiency can be achieved by generating moves conveniently, followed by an inexpensive sort. The comparison here helps to assess this fact.

		Order 1	Order B
KOPEC1 positions	Total nodes	481383	34422497
	Ratio	1	71.7
KOPEC2 positions	Total nodes	846743	127483242
	Ratio	1	150.5

Table 5.5 Comparison 7: Comparing order 1 and order B

5.2.6. Overview of the order experiments

Figure 5.3 plots the expense of searching a capture tree with the discussed orders. The KOPEC2 positions are represented by the dashed line. It is seen that the various heuristics in capture ordering lie between the best order and the random order. The worst order (not shown in the graph), i.e., capturing the smallest pieces first searches the entire tree, and shows how the alpha-beta algorithm can be inefficient on badly ordered trees. Unordered trees correspond to the last two positions on the graph of Figure 5.3 and are many times inefficient compared to the best order. Data corresponding to all the order experiments are given in the Appendix A1.

For KOPEC1 positions, it is seen that the order B is better than order C, while for the KOPEC2 positions, order C is better. In capture order B, all captures from smallest to biggest from-pieces are generated. In capture order C, all captures from biggest to smallest from-pieces are generated. The attacked pieces (to-pieces) are distributed in a arbitrary order. Since the attacked pieces play a major role in ordering the capture search, but are distributed randomly, order B is better some times and order C is better otherwise. This is the reason for the discrepancy in the total nodes searched for capture order C, in the KOPEC1 and KOPEC2 positions.

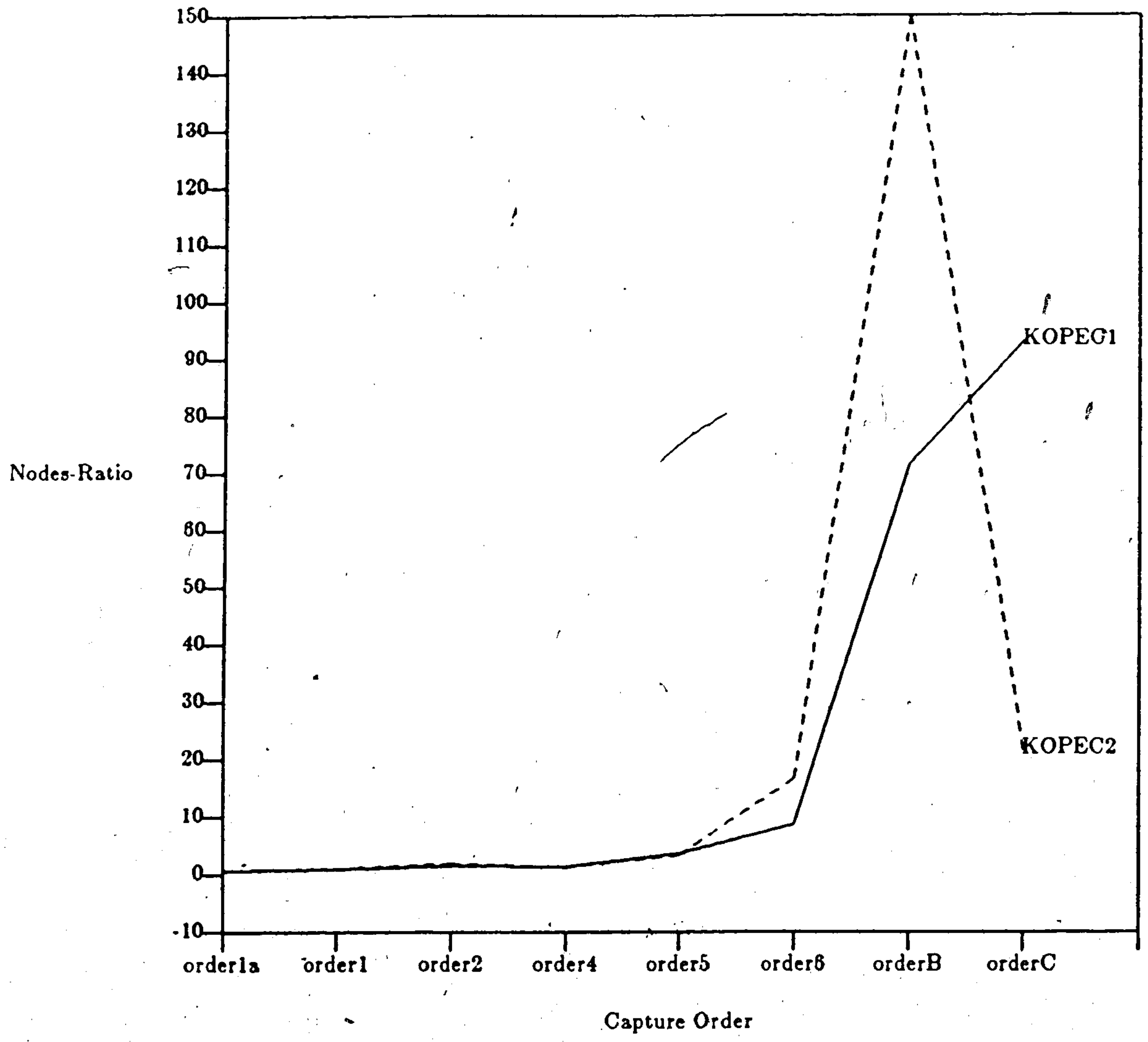


Figure 5.3 Nodes searched Versus Capture Orders

5.3. Depth experiments

Game tree search programs are known to do better if they search deeper trees. This is because they can explore favorable combinations of moves, or expose defects, better. Look-ahead is widely accepted in minimax game tree search, and there are many attempts to analyze this strategy [Bea80, Bea82, BrG82, Pea83]. When an approximate evaluation function is used for the terminal positions, the minimax backed up values are statistically more precise than the value produced by a static evaluation function. Game trees for chess generally have a uniform branching factor and a fixed depth. If the search must be extended by one ply, the number of extra nodes to be searched, can be easily estimated. Searching deeper increases the required effort, but results in playing a better game. It is much harder to quantify the error as the evaluation functions are not well defined.

In capture search, the trees are narrow (because of lower branching factor), and arbitrarily deep (because the search must end in quiescent positions), so it is not easy to estimate the size of a capture tree. Typically, 50% of the total time spent in a chess program, is spent in capture search [SlA83]. So, the performance can be improved if the capture search looks at smaller trees. If depth is changed to reduce the size of the tree, then the search must be terminated at non-quiescent positions, whose static evaluation is inexact, and therefore may return an error value. Because the static evaluation on quiescent positions is exact (only material balance is considered in capture search), it is easy to quantify the resulting error.

Two types of measurements are made in these experiments. (1) Error and (2) Savings. By searching deeper, more errors can be eliminated as the static evaluation on more nodes gets accurate. Such a search is also expensive since larger trees are searched. Thus these experiments provide interesting results about the error and savings involved, when the search ~~is~~ terminated to different depths. The results the experiments are plotted in the graphs, with the KOPEC1 positions represented by a solid line and the KOPEC2 positions represented by a dashed line.

5.3.1. Experimental set-up

The TinkerBelle chess program's capture search routine is used to conduct the depth experiments. Capture analysis is made to variable depth, after two-ply and three-ply full width search on the two sets of test positions. Capture analysis for depth equal to sixteen is taken as the reference for the total nodes searched and the return value of the search tree. Rarely the capture tree depth equals sixteen, so the results from an arbitrary depth capture tree and the one with depth equal to sixteen are almost same.

5.3.2. Error measurements

The capture search is performed at each non-quiescent (all depth 2 and depth 3 nodes are considered) position. Each time the search returns a different value to that when the depth is sixteen, it is recorded as an error. For every board position, ratio of the number of times a different value is returned, to the number of non-quiescent positions searched represents the percentage error. This is computed over the two sets of test positions.

Figure 5.4 plots the %error in capture search with respect to depth. It is seen that as the search depth is increased, the return value of search becomes more accurate. This is because more nodes become quiescent, and the static evaluation on *more nodes* becomes accurate. Minimaxing accurate values through the tree increases the accuracy at the root node. The KOPEC1 and the KOPEC2 positions have nearly the same error values at different depths.

Some programs adopt static evaluation of a position instead of capture search. It is believed that such analysis could be inaccurate, and may lead to many blunders during a chess game. A static analysis cannot effectively

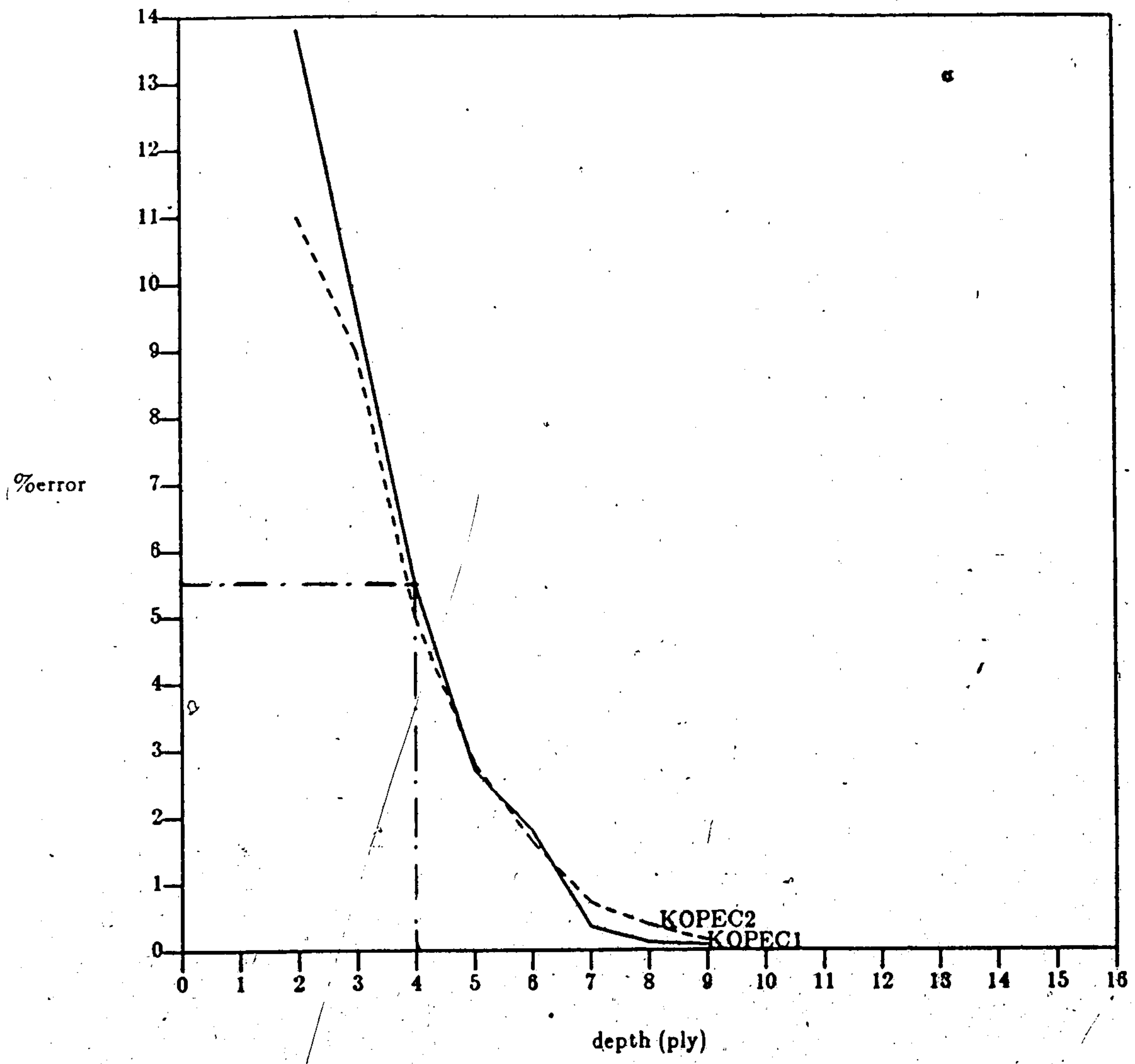


Figure 5.4 Error versus Depth

consider the side effects on captures. For example, a pinned piece can be exposed when a capture is actually made, but cannot be evaluated by a static analysis. Even if we consider the static analysis as equivalent to a full 4-ply capture search, the error can be as high as 5.5%.

5.3.3. Savings

Capture trees may have a lower branching factor at deeper levels of the tree, so the shape of the plot savings versus depth can be considerably different from that for a game tree. Searching deeper than a certain level of the search tree can taper off the savings, even with the search becoming more accurate. Experiments in this section try to characterize this behavior of capture trees.

The number of board positions analyzed by the capture search routine, either when it recursively calls itself, or when called by the main search program are referred to as capture nodes. Restricting the search depth can result in smaller capture trees, reducing the capture nodes searched. Percentage savings is the ratio of this reduction of capture nodes to the total capture nodes searched when depth is sixteen.

Figure 5.5 plots the savings in searched nodes for various depths. As expected, the savings tapers off at higher depths, but is significant for a shallow search. The KOPEC2 positions represented by the dashed line have a larger savings than the KOPEC1 positions, because they are generally complex, and build large capture trees. For example, at search depth=7, there is a 15% savings in the KOPEC1 positions, while it is 21% in the KOPEC2 positions. The higher slope of the savings curve, from the graph of Figure 5.5 suggests that the KOPEC2 positions have a larger branching factor even at depth=9 ply.

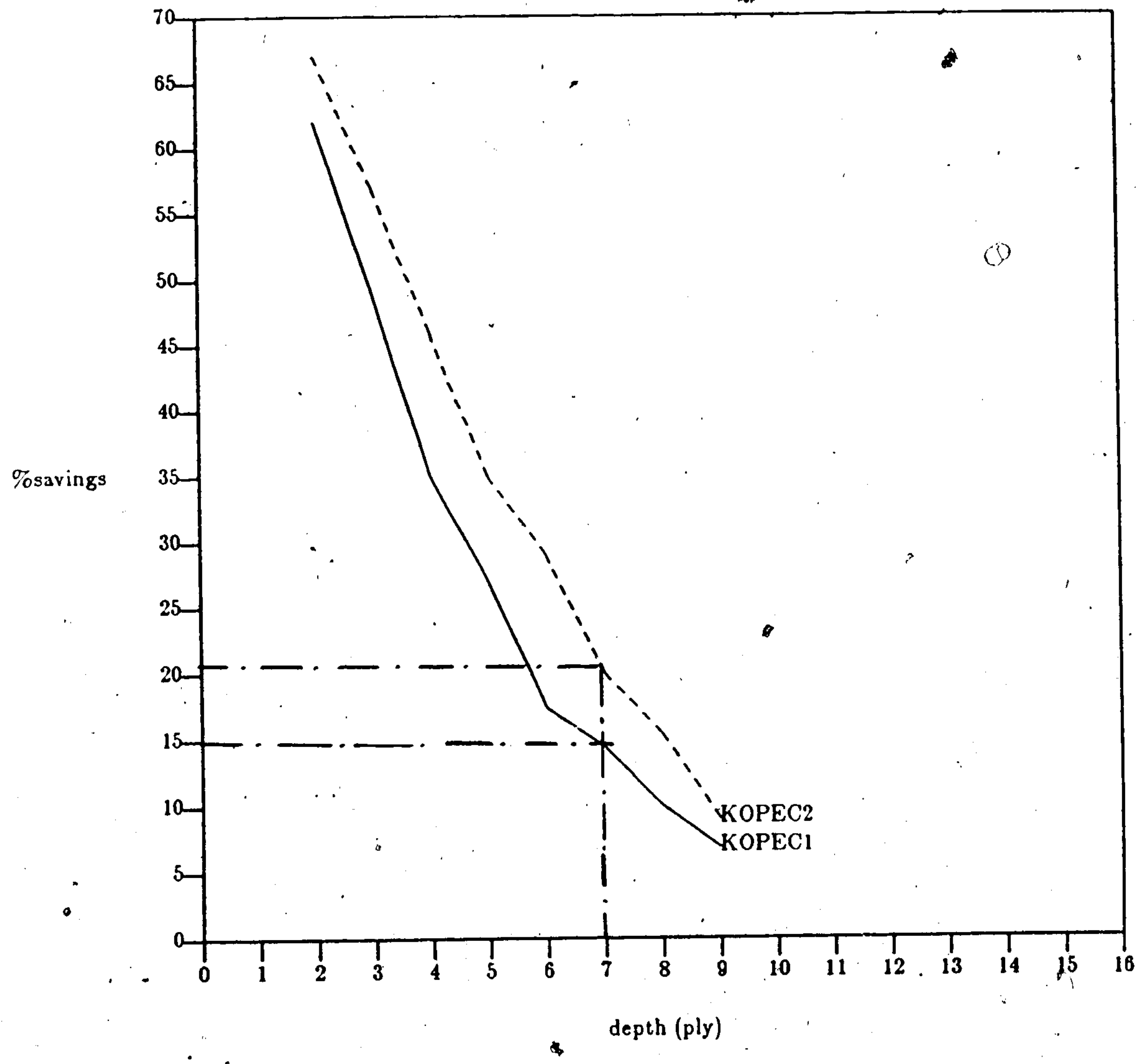


Figure 5.5 Savings versus Depth

5.3.4. Savings versus error

A plot of savings versus error helps in deciding the depth for capture search by combining error and savings information on the same graph. This graph can be used to choose an optimum search depth, where the error is not high, but there is significant savings. This can also help in choosing a search depth for an estimated error and savings values.

Figure 5.6 plots savings versus error for the two sets of test positions. The KOPEC2 positions represented by the dashed line result in larger savings and smaller error than the KOPEC1 positions.

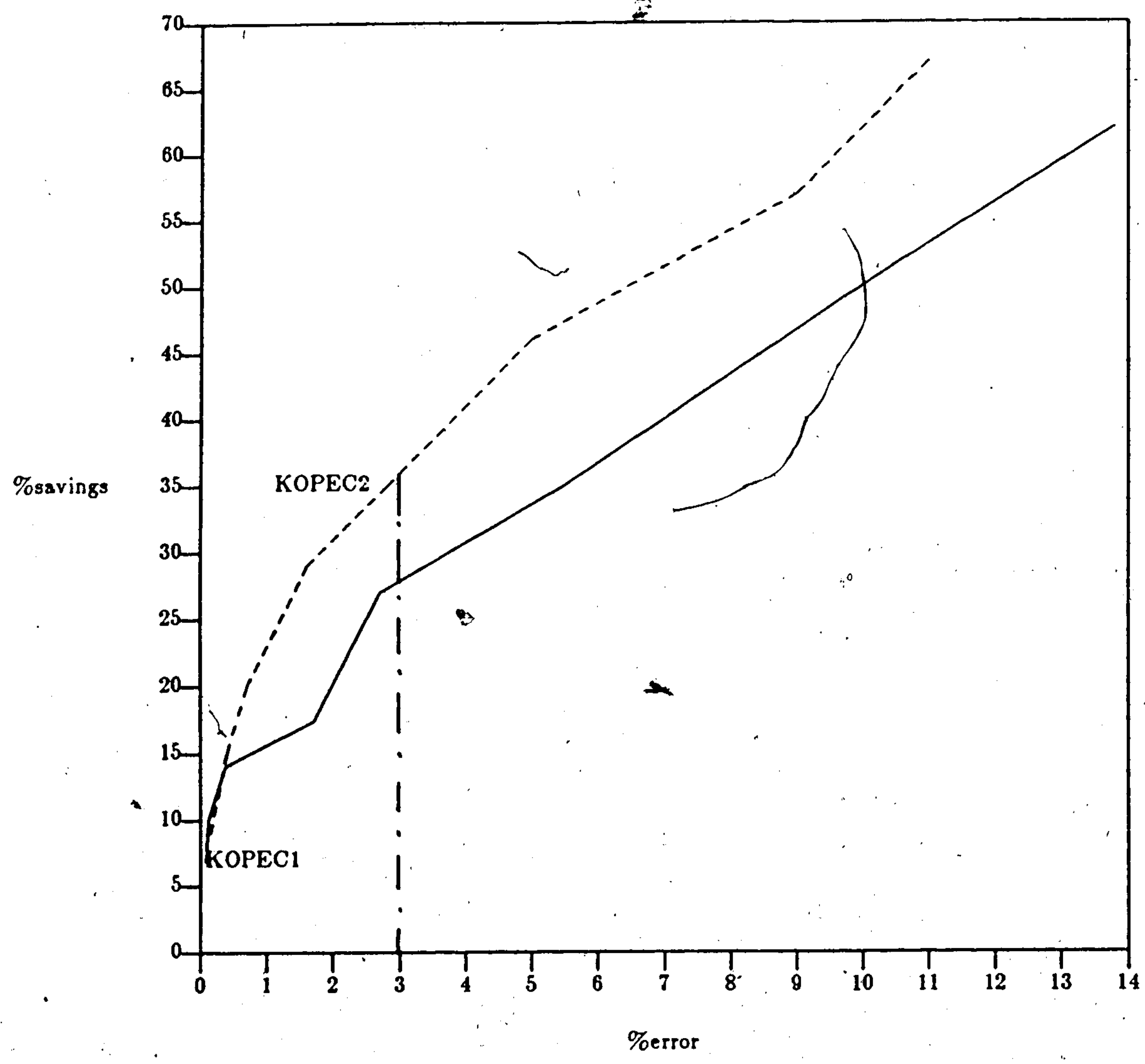


Figure 5.6 Savings versus Error

The shape of the plot does not suggest a point at which there is high savings for a given error. The difference in KOPEC1 and KOPEC2 values makes it hard to choose the curve for deciding search depth. The true range for savings versus error curve may lie in the vicinity, so we can get an approximate value of savings for a specified error. Moreover, the returned val·e from capture search itself may be erroneous, since ideally, a quiescence search is preferred. So these error results may not be accurate.

5.3.5. Search to even and odd depths

When the search is terminated by specifying depth, the error and savings values can differ for odd and even ply, and is interesting to plot them separately. Some positions had significant difference in the number of total nodes searched and the number of error nodes, for odd and even ply, which prompted the plotting this graph.

Figure 5.7 plots the error versus savings for even and odd depths for the KOPEC1 positions only. The dotted line represents the odd depth values. It is seen that the search to odd depth has consistently less error for a given savings, and more savings for a given error value, than that for even depth. These are shown by the horizontal and vertical pointers on the graph. Search to an odd depth results in the last move by a piece of the color that made the first move. This has more effect on the returned value and so a greater accuracy. It is thus a good idea to terminate the capture search to odd depths.

The data corresponding to all the depth experiments are given in the Appendix A2.

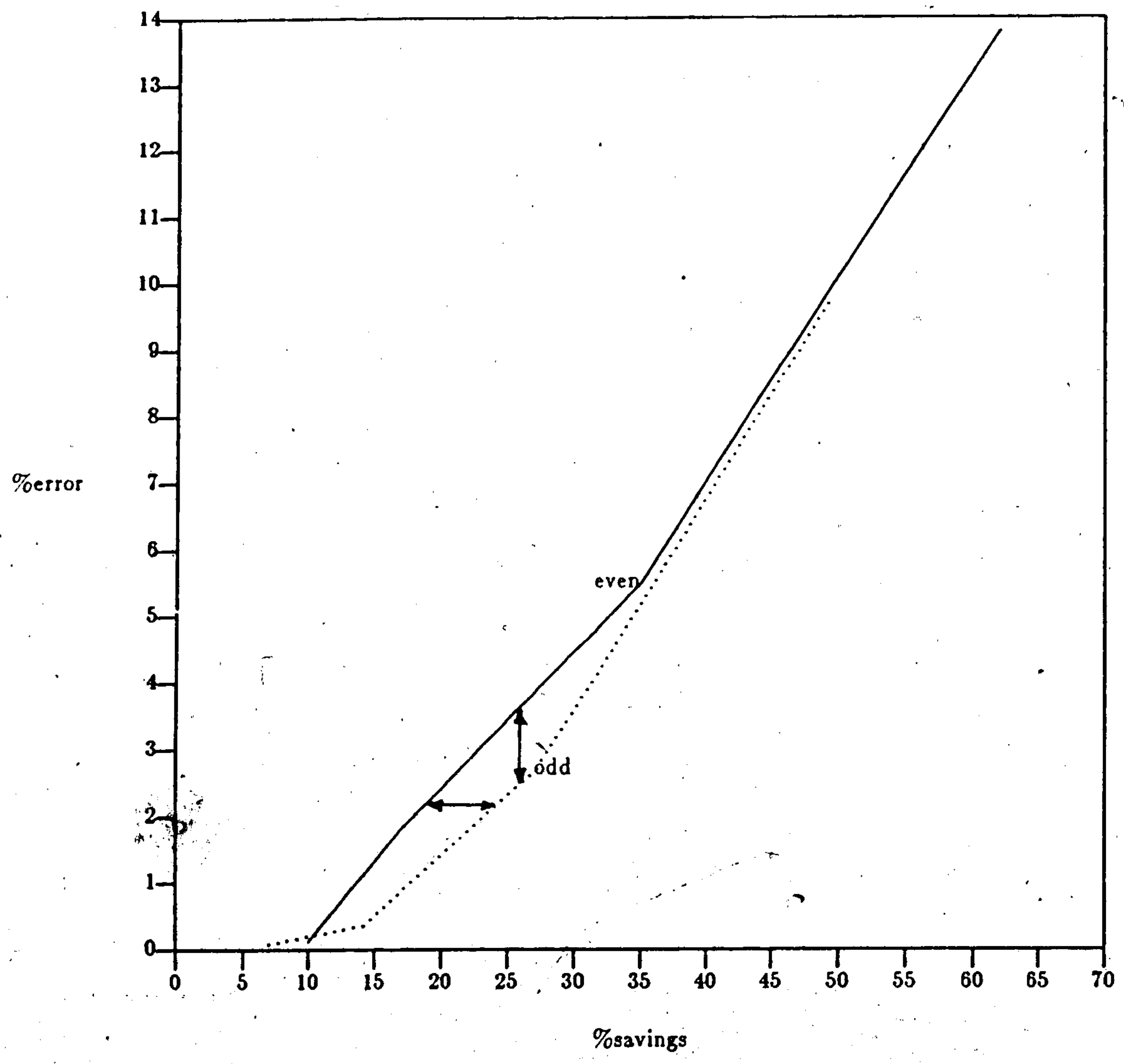


Figure 5.7 Search to Even and Odd depths

5.4. Conclusions

The experiments provide a good insight into the capture searching in chess. The order experiments suggest that the capture order 1 is the best, and cuts the size of search tree drastically. The worst order could be many times inefficient and even the random orders (orders B and C) are not desirable. The value of to-pieces plays a major role in determining the order, although the from-pieces also have an effect on the efficiency of search.

The depth experiments give an idea of what savings can be expected from a capture search to various depths and what errors are involved. A 3% error, allowed by engineering approximations, results in a savings of about 30% on average. There is no value for an optimum depth that has a maximum savings with a tolerable error. The choice of depth for search is only a compromise between the error and savings involved. If depth must be chosen as the criterion for terminating the search, it is better to terminate at an odd ply.

Chapter 6

CONCLUSIONS

The work in this thesis provides some insights into the tree searching mechanism in general, and into the capture search of chess programs in particular. A simple design that uses limited hardware support has been implemented and its performance is compared to TinkerBelle's capture search. The amount by which the efficiency of implementation is improved by applying heuristics on the search order, and the amount of error-savings values for search to varying depths are also quantified.

The search algorithm requires that the move generator must provide moves one by one, to simplify the interface between them. For efficiency, the search algorithm prefers the moves (or captures) in a desirable order, which is achieved in hardware by large combinatorial circuits and a complex priority circuit. An implementation with a simple hardware alternative that requires the CAP_ROM to extract the "next best move," reveals that looping through the entries of CAP_ROM is expensive, and is not a preferred alternative for capture search. Move generation using bitmaps is seen to be fast, but mode of extracting moves from bitmaps is inefficient. Since sorting is cheap (Chapter 4), moves can be generated conveniently (Capture Order B) and then sorted, so that search can be efficient. Handling bitmaps can be expensive on conventional processors, so these operations can be microprogrammed on specialized processors, or processors like MC68020 can be used. Thus there is no "simple hardware alternative" to search the tree with the hardware search mechanism discussed in Chapter 2.

Generalized tree searching mechanisms require more issues to be considered. In a quiescence search, what moves can be generated must be dynamically determined, so with the CAP_ROM approach, the contents cannot be predefined. Implementing a sophisticated node evaluation function, move selection mechanism, quiescence search, are harder to implement with a pure hardware approach. However, hardware assists can be provided to handle the expensive components. For example, move-list generation can be microprogrammed as a function; ARRAY_LOGIC, used for generating the sliding moves, can be built as separate hardware and interfaced.

The order experiments let us to assess the role played by the value of participating pieces in capture search. The experiments clearly indicate that the capture order 1, in which the biggest pieces captured by the smallest pieces are considered first, is the best order. With the additional cut-off, the search can be made more efficient. It is also seen that the order of to-pieces plays a greater role, but the order of from-pieces also has some effect on the search efficiency. The worst order for searching the tree, i.e., capturing the smallest pieces first suggests the maximum size of capture trees. Even ~~the~~ random orders are not suitable for tree searching, and moves must be ordered before the search algorithm examines them.

Building ordering functions in hardware is important as seen in the practical chess machines. The CHEOPS does not have proper move ordering built in hardware, whereas, BELLE has a simple ordering mechanism. HITECH has a more sophisticated ordering mechanism -- perhaps HITECH's performance is superior.

A deeper search results in more accurate values. The extra savings obtained in searching a depth limited capture tree is significant only when large error is tolerated. The depth experiments also suggest that there is no optimum depth for which the savings/error is a maximum. Thus we have to compromise between the accuracy of the solution (or error) and the effort to be spent (or savings) in choosing a search depth. Even a 7-ply deep search with 0.4 to 0.7% error gives less than 20% savings, so an arbitrary depth capture search may be preferred.

References

[AAB70] G. M. Adel'son-Vel'skii, V. L. Arlazarov, A. R. Bitman, A. A. Zhivotovskii and A. V. Uskov, Programming a computer to play chess, *Russian Mathematical Surveys 25*, 2 (March-April 1970), 221-257.

[Bau78] G. M. Baudet, On the branching factor of alpha-beta algorithm, *Artificial Intelligence 10*, (1978), 173-199.

[Bea80] D. F. Beal, An Analysis of Minimax, in *Advances in Computer Chess 2*, M. R. B. Clarke (ed.), Edinburgh Univ. Press, Edinburgh, 1980, 103-109.

[Bea82] D. F. Beal, Benefits of Minimax Search, in *Advances in Computer Chess 3*, M. R. B. Clarke (ed.), Pergamon Press, Oxford, 1982, 1-16.

[Ber74] Hans Berliner, Chess as Problem Solving: The Development of a Tactics Analyzer, Ph.D Thesis, Department of Computer Science, Carnegie-Mellon University, 1974.

[BrG82] I. Bratko and M. Gams, Error Analysis of the Minimax principle, in *Advances in Computer Chess 3*, M. R. B. Clarke (ed.), Pergamon Press, Oxford, 1982, 1-16.

[CaM83] Murray S. Campbell and T. A. Marsland, A comparison of Minimax Tree Search Algorithms, *Artificial Intelligence 20*, (1983), 347-367.

[CaM82] M. S. Campbell and T. A. Marsland, A Comparison of Minimax Tree Search Algorithms, Tech. Rep. 82-3, Dept. of Comp. Sc., Univ of Alberta, Edmonton, July 1982.

[CoT82] Joe Condon and Ken Thompson, Belle Chess Hardware, in *Advances in Computer Chess 3*, M. R. B. Clarke (ed.), Pergamon Press, Oxford, 1982, 45-54.

[Cra84] Stuart M. Cracraft, Bitmap Move Generation in Chess, *ICCA Journal 7*, 3 (September 1984), 146-153.

[EbP84] Carl Ebeling and Andrew Palay, The Design and Implementation of a VLSI Chess Move Generator, *Proceedings 11th Annual Symposium on Computer Architecture*, 1984.

[Fre85] Peter W. Frey, The Alpha-Beta algorithm: incremental updating, well-behaved evaluation functions, and non-speculative forward pruning, in *Computer Game Playing*, M. A. Bramer (ed.), Ellis Horwood Limited Publishers, Chichester, 1985, 285-289.

[GEC67] R. D. Greenblatt, D. E. Eas[illegible]ke III and S. D. Crocker, The Greenblatt Chess Program, *Proc. AFIPS Fall Joint Computer Conference 31*, (1967), 801-810.

[Hya85] Robert M. Hyatt, Parallel Chess on the CRA~~Y X-M~~P/4[illegible] *ICCA Journal 8*, 2 (June 1985), 90-99.

[Ibb82] Roland N. Ibbe[illegible], in *The Architecture of High Performance Computers*, Springer-Verlag New York Inc., 1982, 112-125.

[Kai85] Hermann Kaindl, Quiescence Search in Computer Chess, in *Computer Game Playing*, M. A. Bramer (ed.), Ellis Horwood Limited Publishers, Chichester, 1985, 38-52.

[KnM75] D. E. Knuth and R. Moore, An Analysis of Alpha-Beta pruning, *Artificial Intelligence 6*, (1975), 293-326.

[KoB82] D. Kopec and I. Bratko, The Bratko-Kopec Experiment: A comparision of Human and computer performance in chess, in *Advances in Computer Chess 3*, M. R. B. Clarke (ed.), Pergamon Press, Oxford, 1982, 57-72.

[KNY85] Danny Kopec, M. M. Newborn and Winston Yu, Experiments in Chess Cognition, in *Advances in Computer Chess 4*, D. Beal (ed.), Pergamon Press, Oxford, 1985, 59-79.

[Mar85a] T. A. Marsland, Personal Communication, 1985.

[Mar85b] T. A. Marsland, Evaluation Function Factors, *ICCA Journal 8*, 2 (June 1985), 47-57.

[MHG79] J. Moussouris, J. Holloway and R. Greenblatt, CHEOPS: A Chess-oriented Processing System, in *Machine Intelligence*, vol. 9, J. E. Hayes, D. Michie and L. I. Mikulich (ed.), Horwood-Wiley, Chichester, UK, 1979, 351-360.

[Mye82] Glenford Myers, in *Advances in Computer Architecture II*, Wiley, New York, 1982, 25-35.

[Not84] Rudolf W. Nottrott, NETLIST and RNL - Tutorial for Beginners, *UW/NW VLSI Consortium*, 1984.

[Pea80] Judea Pearl, Asymptotic properties of minimax trees and game searching procedures, *Artificial Intelligence 14*, (1980), 113-138.

[Pea83] Judea Pearl, On the nature of pathology in game searching, *Artificial Intelligence 20*, (1983), 427-453.

[Pea85] Judea Pearl, Game searching theory: Survey of recent results, in *Computer Game Playing*, M. A. Bramer (ed.), Ellis Horwood Limited Publishers, Chichester, 1985, 276-284.

[Sch86] J. Schaeffer, *Experiments in Search and Knowledge*, Ph.D. thesis, Computer Science Department, University of Waterloo, 1986.

[SPJ83] Jonathan Schaeffer, Patrick A. D. Powell and Jim Jonkman, *A VLSI Chess Legal Move Generator*, Technical report 4, University of Waterloo, Feb 1983.

[Sha50] C. Shannon, Programming a computer for playing chess, *Philosophical Magazine 41*, (1950), 256-275.

[SlA83] David J. Slate and Lawrence R. Atkin, Chess 4.5-The Northwestern University chess program, in *Chess skill in man and machine*, Peter W. Frey (ed.), Springer-Verlag, New York, 1983.

[WeB85] David E. Welsh and Boris Baczynskyj, in *Computer Chess II*, Wm. C. Brown Publishers, Dubuque, Iowa, 1985, 6-97.

[Wen85] Burton Wendroff, Attack detection and move generation on the X-MP/48, *ICCA Journal 8*, 2 (June 1985), 58-65.

Appendix A1

Order Experiments

Effect of capture ordering on alpha-beta search algorithm

KOPEC1 positions

	Order1a	Order1	Order2	Order4	Order5	Order6	Order8	OrderC
Total	309729	481383	685196	659229	1708750	4153507	34422497	44802627
Ratio	0.643	1	1.423	1.369	3.55	8.628	71.5	93.07

KOPEC2 positions

	Order1a	Order1	Order2	Order4	Order5	Order6	Order8	OrderC
Total	504002	846743	1526170	1054186	2816330	14090611	127483242	18337620
Ratio	0.59	1	1.802	1.245	3.326	16.64	150.55	21.66

Appendix A2

Depth Experiments

Error and Savings values for capture search to various depths

KOPEC1 positions

Total Number of Quiescent nodes analyzed: 37138

	depth=16	depth=9	depth=8	depth=7	depth=6	depth=5	depth=4	depth=3	depth=2
Searched nodes:	233377	217547	209438	199677	192859	169876	151717	119861	89064
Saved Cnodes:	0	15830	23939	33700	40518	63501	81660	113516	144313
% of savings:	0.00	6.78	10.26	14.44	17.36	27.21	34.99	48.64	61.84
Ret val chg:	0	29	45	133	658	1008	2028	3593	5113
% Error :	0.00	0.08	0.12	0.36	1.77	2.71	5.46	9.67	13.77

KOPEC2 positions

Total Number of Quiescent nodes analyzed: 45842

	depth=16	depth=9	depth=8	depth=7	depth=6	depth=5	depth=4	depth=3	depth=2
Searched nodes:	309254	281500	261673	247453	220686	200208	166240	132337	103331
Saved cnodes:	0	27754	47581	61801	88568	109046	143014	176917	205923
% of savings:	0.00	8.97	15.39	19.98	28.64	35.26	46.24	57.21	66.59
Ret val chg:	0	68	173	328	749	1288	2287	4146	5057
% Error :	0.00	0.15	0.38	0.72	1.63	2.81	4.99	9.04	11.03