Computer Chess Methods

Article · October 2001

Source: CiteSeer

1 author:

T. Anthony Marsland
University of Alberta

COMPUTER CHESS METHODS

T.A. Marsland

Computing Science Department,
University of Alberta,
EDMONTON,
Canada T6G 2H1

ABSTRACT

Article prepared for the ENCYCLOPEDIA OF ARTIFICIAL INTELLIGENCE, S. Shapiro (editor), D. Eckroth (Managing Editor) to be published by John Wiley, 1987.

Final draft; DO NOT REPRODUCE OR CIRCULATE.

This copy is for review only. Please do not cite or copy.

Acknowledgements

Don Beal of London, Jaap van den Herik of Amsterdam and Hermann Kaindl of Vienna provided helpful responses to my request for definitions of technical terms. They and Peter Frey of Evanston also read and offered constructive criticism of the first draft, thus helping improve the basic organization of the work. In addition, my many friends and colleagues, too numerous to mention individually, offered perceptive observations. To all these people, and to Ken Thompson for typesetting the chess diagrams, I offer my sincere thanks. The experimental work to support results in this article was possible through Canadian Natural Sciences and Engineering Research Council Grant A7902.

December 15, 1990

COMPUTER CHESS METHODS

T.A. Marsland

Computing Science Department,
University of Alberta,
EDMONTON,
Canada T6G 2H1

1. HISTORICAL PERSPECTIVE

Of the early chess-playing machines the best known was exhibited by Baron von Kempelen of Vienna in 1769. Like its relations it was a conjurer's box and a grand hoax [1, 2]. In contrast, about 1890 a Spanish engineer, Torres y Quevedo, designed a true mechanical player for king-and-rook against king endgames. A later version of that machine was displayed at the Paris Exhibition of 1914 and now resides in a museum at Madrid's Polytechnic University [2]. Despite the success of this electro-mechanical device, further advances on chess automata did not come until the 1940's. During that decade there was a sudden spurt of activity as several leading engineers and mathematicians, intrigued by the power of computers and fascinated by chess, began to express their ideas on computer chess. Some, like Tihamer Nemes of Budapest [3] and Konrad Zuse [4], tried a hardware approach but their computer chess works did not find wide acceptance. Others, like noted computer scientist Alan Turing, found success with a more philosophical tone, stressing the importance of the stored program concept [5]. Today, best recognized are the 1965 translation of Adriaan de Groot's 1946 doctoral dissertation [6] and the much referenced paper on algorithms for playing chess by Claude Shannon [7]. Shannon's inspirational paper was read and reread by computer chess enthusiasts, and provided a basis for most early chess programs. Despite the passage of time, that paper is still worthy of study.

1.1. Landmarks in Chess Program Development

The first computer model in the 1950's was a hand simulation [5]; programs for subsets of chess followed [8] and the first full working program was reported in 1958 [9]. By the mid 1960's there was an international computer-computer match [10] between a program backed by John McCarthy of Stanford (developed by a group of students from MIT [11]) and one from the Institute for Theoretical and Experimental Physics (ITEP) in Moscow [12]. The ITEP group's program (under the guidance of the well-known mathematician Georgi Adelson-Velskiy) won the match, and the scientists involved went on to develop *Kaissa*†, which became the first world computer chess champion in 1974 [13]. Meanwhile there emerged from MIT another program, *Mac Hack Six* [16], which boosted interest in Artificial Intelligence. First, *Mac Hack* was demonstrably superior not only to all previous chess programs, but also to most casual chess players. Secondly, it contained more sophisticated move ordering and position evaluation methods. Finally, the program incorporated a memory table to keep track of the values of chess positions that were seen more than once. In the late 60's, spurred by the early promise of *Mac Hack*, several people began developing chess programs and writing proposals. Most substantial of the proposals was the twenty-nine point plan by Jack Good [17]. By and large experimenters did not make effective use of these works, at least nobody claimed a program based on those designs, partly because it was not clear how some of the ideas could be addressed and partly because some points were too naive. Even so, by 1970 there was enough progress that Monroe Newborn was able to convert a suggestion for a public demonstration of chess playing computers into a competition that attracted eight participants [18]. Due mainly to Newborn's careful planning and organization this event continues today under the title "The ACM North American Computer Chess Championship."

In a similar vein, under the auspices of the International Computer Chess Association, a worldwide computer chess competition has evolved. Initial sponsors were the IFIP triennial conference in Stockholm (1974) and Toronto (1977), and later independent backers such as the Linz (Austria) Chamber of Commerce (1980), ACM New York (1983) and for 1986, the city of Cologne, West Germany. In the first world championship for computers *Kaissa* won all its games, including a defeat of the eventual second place

† The names of programs mentioned here will be written in italics. Descriptions of these programs can be found in various books [13, 14]. Interviews with some of the designers have also appeared [15].

finisher, *Chaos*. An exhibition match against the 1973 North American Champion, *Chess 4.0*, was drawn [10]. *Kaissa* was at its peak, backed by a team of outstanding experts on tree searching methods. In the second Championship (Toronto, 1977), *Chess 4.6* finished first with *Duchess* [19] and *Kaissa* tied for second place. Meanwhile both *Chess 4.6* and *Kaissa* had acquired faster computers, a Cyber 176 and an IBM 370/165 respectively. The traditional exhibition match was won by *Chess 4.6*, indicating that in the interim it had undergone far more development and testing [20]. The 3rd World Championship (Linz, 1980) finished in a tie between *Belle* and *Chaos*. In the playoff *Belle* won convincingly, providing perhaps the best evidence yet that a deeper search more than compensates for an apparent lack of knowledge. In the past, this counter-intuitive idea had not found ready acceptance in the Artificial Intelligence community.

More recently, in the New York 1983 championship another new winner emerged, *Cray Blitz* [21]. More than any other, that program drew on the power of a fast computer, here a Cray X-MP. Originally *Blitz* was a selective search program, in the sense that it could discard some moves from every position, based on a local evaluation. Often the time saved was not worth the attendant risks. The availability of a faster computer made it possible to use a purely algorithmic approach and yet retain much of the expensive chess knowledge. Although a mainframe won that event, small machines made their mark and seem to have a great future [22]. For instance, *Bebe* with special purpose hardware finished second, and even experimental versions of commercial products did well.

1.2. Implications

All this leads to the common question: When will a computer be the unassailed expert on chess? This issue was discussed at length during a "Chess on non-standard Architectures" panel discussion at the ACM 1984 National Conference in San Francisco. It is too early to give a definitive answer, even the experts cannot agree; their responses covered the whole range of possible answers from "in five years" (Newborn), "about the end of the century" (Scherzer and Hyatt), "eventually. - it is inevitable" (Thompson) and "never, or not until the limits on human skill are known" (Marsland). Even so there was a sense that production of an artificial Grand Master was possible, and that a realistic challenge would occur during the first quarter of the 21st century. As added motivation, Edward Fredkin (MIT professor and well-known inventor) has created a special incentive prize for computer chess. The trustee for the Fredkin Prize is Carnegie-Mellon University and the fund is administered by Hans Berliner. Much like the Kremer prize for man-powered flight, awards are offered in three categories. The smallest prize of $5000 has already been presented to Ken Thompson and Joe Condon, when their *Belle* program achieved a US Master rating in 1983. The other awards of $10,000 for the first Grand Master program, and $100,000 for achieving world champion status remain unclaimed. To sustain interest in this activity, each year a $1500 prize match is played between the currently best computer and a comparably rated human.

One might well ask whether such a problem is worth all this effort, but when one considers some of the emerging uses of computers in important decision-making processes the answer must be positive. If computers cannot even solve a decision making problem in an area of perfect knowledge (like chess), then how can we be sure that computers make better decisions than humans in other complex domains -- especially in domains where the rules are ill-defined, or those exhibiting high levels of uncertainty? Unlike some problems, for chess there are well established standards against which to measure performance, not only through a rating scale [23] but also using standard tests [24] and relative performance measures [25]. The ACM sponsored competitions have provided fifteen years of continuing experimental data about the effective speed of computers and their operating system support. They have also afforded a public testing ground for new algorithms and data structures for speeding the traversal of search trees. These tests have provided growing proof of the increased understanding about chess by computers, and the encoding of a wealth of expert knowledge. Another potentially valuable aspect of computer chess is its usefulness in demonstrating the power of man-machine cooperation. One would hope, for instance, that a computer could be a useful adjunct to the decision-making process, providing perhaps a steadying influence, and protecting against errors introduced by impulsive short-cuts of the kind people might try in a careless or angry moment. In this and other respects it is easy to understand Donald Michie's belief that computer chess is the "*Drosophila melanogaster* [fruit fly] of machine intelligence" [26].

2. TERMINOLOGY

There are several aspects of computer chess of interest to Artificial Intelligence researchers. One area involves the description and encoding of chess knowledge, in a form that enables both rapid access and logical deduction in the expert system sense. Another fundamental domain is that of search. Since computer chess programs examine large trees, a depth-first search is commonly used. That is, the first branch to an immediate successor of the current node is recursively expanded until a leaf node (a node without successors) is reached. The remaining branches are then considered as the search process backs up to the root. Other expansion schemes are possible and the domain is fruitful for testing new search algorithms. Since computer chess is well defined, and absolute measures of performance exist, it is a useful test vehicle for measuring algorithm efficiency. In the simplest case, the best algorithm is the one which visits fewest nodes when determining the true value of a tree. For a two-person game-tree this value, which is a least upper bound on the merit for the side to move, can be found through a minimax search. In chess, this so called minimax value is a combination of both the "MaterialBalance" (i.e., the difference in value of the pieces held by each side) and the "StrategicBalance," e.g., a composite measure of such things as mobility, square control, pawn formation structure and king safety. Usually MaterialBalance is dominant.

2.1. Minimax Search

For chess, the nodes in a two-person game-tree represent positions and the branches correspond to moves. The aim of the search is to find a path from the root to the highest valued terminal node that can be reached, under the assumption of best play by both sides. To represent a level in the tree (that is, a play or half move) the term "ply" was introduced by Arthur Samuel in his major paper on machine learning [27]. How that word was chosen is not clear, perhaps as a contraction of "play" or maybe by association with forests as in layers of plywood. In either case it was certainly appropriate and it has been universally accepted.

A true minimax search is expensive since every leaf node in the tree must be visited. For a tree of uniform width W and fixed depth D there are W^D terminal nodes. Some games, like Fox and Geese [28], produce narrow trees (fewer than 10 branches per node) that can often be solved exhaustively. In contrast, chess produces bushy trees (average branching factor about 35 moves). Because of the magnitude of the game tree, it is not possible to search until a mate or stalemate position (a leaf node) is reached, so some maximum depth of search (i.e., a horizon) is specified. Even so, an exhaustive search of all chess game trees involving more than a few moves for each side is impossible. Fortunately the work can be reduced, since it can be shown that the search of some nodes is unnecessary.

2.2. Alpha-beta Algorithm

As the search of the game tree proceeds, the value of the best terminal node found so far changes. It has been known since 1958 that pruning was possible in a minimax search [29], but according to Knuth and Moore the ideas go back further, to John McCarthy and his group at MIT. The first thorough treatment of the topic appears to be Brudno's 1963 paper [30]. The alpha-beta algorithm employs lower (alpha) and upper (beta) bounds on the expected value of the tree. These bounds may be used to prove that certain moves cannot affect the outcome of the search, and hence that they can be pruned or cut off. As part of the early descriptions about how subtrees were pruned, a distinction between deep and shallow cut-offs was made. Some versions of the alpha-beta algorithm used only a single bound (alpha), and repeatedly reset the beta bound to infinity, so that deep cut-offs were not achieved. Knuth and Moore's recursive F2 algorithm [31] corrected that flaw. In Figure 1, Pascal-like pseudo code is used to present the alpha-beta algorithm, AB, in Knuth and Moore's negamax framework. A <u>Return</u> statement has been introduced as the convention for exiting the function and returning the best subtree value or score. Omitted are details of the game-specific functions <u>Make</u> and <u>Undo</u> (to update the game board), <u>Generate</u> (to find moves) and <u>Evaluate</u> (to assess terminal nodes). In the pseudo code of Figure 1, the max(alpha,merit) operation represents Fishburn's "fail-soft" condition [32], and ensures that the best available value is returned (rather than an alpha/beta bound). This idea is usefully employed in some of the newer refinements to the alpha-beta algorithm.

```
FUNCTION AB (p : position; alpha, beta, depth : integer) : integer;
                              { p is pointer to the current node     }
                              { alpha and beta are window bounds     }
                              { depth is the remaining search length }
                              { the value of the subtree is returned }
  VAR merit, j, value : integer;
      posn : ARRAY [1..MAXWIDTH] OF position;
                                      { Note: depth must be positive }
BEGIN
  IF depth = 0 THEN                   { horizon node, maximum depth? }
     Return(Evaluate(p));

  posn := Generate(p);                { point to successor positions }
  IF empty(posn) THEN                         { leaf, no moves?      }
     Return(Evaluate(p));
                                      { find merit of best variation }
  merit := -MAXINT;
  FOR j := 1 TO sizeof(posn) DO BEGIN
     Make(posn[j]);                              { make current move }
     value := -AB (posn[j], -beta, -max(alpha,merit), depth-1);
     IF (value > merit) THEN                    { note new best score  }
        merit := value;
     Undo(posn[j]);                           { retract current move }
     IF (merit ≥1 beta) THEN              { cutoff? }
    GOTO done;
 END ;
done:
 Return(merit);
END ;
```

Figure 1: Depth-limited Alpha-beta Function.

2.3. Minimal Game Tree

If the "best" move is examined first at every node, then the tree traversed by the alpha-beta algorithm is referred to as the minimal game tree. This minimal tree is of theoretical importance since its size is a measure of a lower bound on the search. For uniform trees of width W branches per node and a search depth of D ply, there are

$$W^{\left\lceil \frac{D}{2} \right\rceil} + W^{\left\lfloor \frac{D}{2} \right\rfloor} - 1$$

terminal nodes in the minimal game tree. Although others derived this result, the most direct proof was given by Knuth and Moore [31]. Since a terminal node is rarely a leaf it is often called a horizon node, with

D the distance to the horizon [33].

2.4. Aspiration Search

An alpha-beta search can be carried out with the initial bounds covering a narrow range, one that spans the expected value of the tree. In chess these bounds might be (MaterialBalance -Pawn, MaterialBalance +Pawn). If the minimax value falls within this range, no additional work is necessary and the search usually completes in measurably less time. The method was analyzed by Brudno [30], referred to by Berliner [34], and experimented with in *Tech* [35], but was not consistently successful. A disadvantage is that sometimes the initial bounds do not enclose the minimax value, in which case the search must be repeated with corrected bounds as the outline of Figure 2 shows.

```
{      Assume V = estimated value of position p, and     }
{             e = expected error limit                   }
{         depth = current distance to horizon            }
{             p = position being searched                }
     alpha := V - e;                     { lower bound  }
     beta  := V + e;                     { upper bound  }

     V := AB (p, alpha, beta, depth);
     IF (V ≥1 beta) THEN          {  failing high  }
       V := AB (p, V, +MAXINT, depth)
     ELSE
     IF (V ≤ alpha) THEN          {  failing low   }
       V := AB (p, -MAXINT, V, depth);

{     A successful search has now been completed     }
{     V now holds the current value of the tree      }
```

Figure 2: Narrow Window Aspiration Search.

Typically these failures occur only when material is being won or lost, in which case the increased cost of a more thorough search is warranted. Because these re-searches use a semi-infinite window, from time to time people experiment with a "sliding window" of (V, V +PieceValue), instead of (V, +MAXINT). This method is often effective, but can lead to excessive re-searching when mate or large material gain/loss is in the offing. After 1974, "iterated aspiration search" came into general use, as follows:

> "Before each iteration starts, alpha and beta are not set to -infinity and +infinity as one might expect, but to a window only a few pawns wide, centered roughly on the final score [value] from the previous iteration (or previous move in the case of the first iteration). This setting of ’high hopes’ increases the number of alpha-beta cutoffs" [36].

Even so, although aspiration searching is still popular and has much to commend it, minimal window search seems to be more efficient and requires no assumptions about the choice of aspiration window [37].

2.5. Minimal Window Search

Theoretical advances, such as Scout [38] and the comparable minimal window search techniques [32, 37] were the next products of research. The basic idea behind these methods is that it is cheaper to prove a subtree inferior, than to determine its exact value. Even though it has been shown that for bushy trees minimal window techniques provide a significant advantage [37], for random game trees it is known that even these refinements are asymptotically equivalent to the simpler alpha-beta algorithm. Bushy trees are typical for chess and so many contemporary chess programs use minimal window techniques through the Principal Variation Search (PVS) algorithm. In Figure 3, a Pascal-like pseudo code is used to describe PVS in a negamax framework, but with game-specific functions <u>Make</u> and <u>Undo</u> omitted for clarity. Here the original version of PVS has also been improved by using Reinefeld’s depth=2 idea [39], which ensures that re-searches are only done when the remaining depth of search is greater than 2.

```
FUNCTION PVS (p : position; alpha, beta, depth : integer) : integer;
                               { p is pointer to the current node     }
                               { alpha and beta are window bounds     }
                               { depth is the remaining search length }
                               { the value of the subtree is returned }
   VAR merit, j, value : integer;
       posn : ARRAY [1..MAXWIDTH] OF position;
                                       { Note: depth must be positive }
BEGIN
   IF depth = 0 THEN                   { horizon node, maximum depth? }
      Return(Evaluate(p));

   posn := Generate(p);                { point to successor positions }
   IF empty(posn) THEN                              { leaf, no moves? }
      Return(Evaluate(p));
                                               { principal variation? }
   merit := -PVS (posn[1], -beta, -alpha, depth-1);
   FOR j := 2 TO sizeof(posn) DO BEGIN
      IF (merit ≥1 beta) THEN          { cutoff? }
    GOTO done;
   alpha := max(merit, alpha);         { fail-soft condition }
                        { zero-width minimal-window search }
   value := -PVS (posn[j], -alpha-1, -alpha, depth-1);
   IF (value > merit) THEN             { re-search, if "fail-high" }
     IF (alpha < value) AND (value < beta) AND (depth > 2) THEN
       merit := -pvs (posn[j], -beta, -value, depth-1)
     ELSE merit := value;
  END ;
done:
  Return(merit);
END ;
```

Figure 3: Minimal Window Principal Variation Search.

2.6. Forward Pruning

To reduce the size of the tree that should be traversed and to provide a weak form of selective search, techniques that discard some branches have been tried. For example, tapered N-best search [11, 16] considers only the N-best moves at each node. N usually decreases with increasing depth of the node from the root of the tree. As Slate and Atkin observe

> "The major design problem in selective search is the possibility that the lookahead process will exclude a key move at a low level in the game tree."

Good examples supporting this point are found elsewhere [40]. Other methods, such as marginal forward pruning [41] and the gamma algorithm [18], omit moves whose immediate value is worse than the current best of the values from nodes already searched, since the expectation is that the opponent's move is only going to make things worse. Generally speaking these forward pruning methods are not reliable and should be avoided. They have no theoretical basis, although it may be possible to develop statistically sound methods which use the probability that the remaining moves are inferior to the best found so far.

One version of marginal forward pruning, referred to as razoring [42], is applied near horizon nodes. The expectation in all forward pruning is that the side to move can improve the current value, so it may be futile to continue. Unfortunately there are cases when the assumption is untrue, for instance in zugzwang positions. As Birmingham and Kent point out, their *Master* program

> "defines zugzwang precisely as a state in which every move available to one player creates a position having a lower value to him (in its own evaluation terms) than the present bound for the position" [42].

Marginal pruning may also break down when the side to move has more than one piece en prise (e.g., is forked), and so the decision to stop the search must be applied cautiously.

Despite these disadvantages, there are sound forward pruning methods and there is every incentive to develop more, since it is one way to reduce the size of the tree traversed, perhaps to less than the minimal game tree. A good prospect is through the development of programs that can deduce which branches can be neglected, by reasoning about the tree they traverse.

2.7. Move Re-ordering Mechanisms

For efficiency (traversal of a smaller portion of the tree) the moves at each node should be ordered so that the more plausible ones are searched soonest. Various ordering schemes may be used. For example,

> "since the refutation of a bad move is often a capture, all captures are considered first in the tree, starting with the highest valued piece captured" [43].

Special techniques are used at interior nodes for dynamically re-ordering moves during a search. In the simplest case, at every level in the tree a record is kept of the moves that have been assessed as being best, or good enough to refute a line of play and so cause a cut-off. As Gillogly observed

> "If a move is a refutation for one line, it may also refute another line, so it should be considered first if it appears in the legal move list" [43].

Referred to as the killer heuristic, a typical implementation maintains only the two most frequently occurring "killers" at each level [36].

Recently a more powerful scheme for re-ordering moves at an interior node has been introduced. Named the history heuristic it

> "maintains a history for every legal move seen in the search tree. For each move, a record of the move's ability to cause a refutation is kept, regardless of the line of play" [44].

At an interior node the best move is the one that either yields the highest merit or causes a cut-off. Many implementations are possible, but a pair of tables (each of 64x64 entries) is enough to keep a frequency count of how often a particular move (defined as a from-to square combination) is best for each side. The available moves are re-ordered so that the most successful ones are tried first. An important property of this so called history table is the sharing of information about the effectiveness of moves throughout the tree, rather than only at nodes at the same search level. The idea is that if a move is frequently good enough to cause a cut-off, it will probably be effective whenever it can be played.

2.8. Quiescence Search

Even the earliest papers on computer chess recognized the importance of evaluating only those positions which are "relatively quiescent" [7] or "dead" [5]. These are positions which can be assessed accurately without further search. Typically they have no moves, such as checks, promotions or complex captures, whose outcome is unpredictable. Not all the moves at horizon nodes are quiescent (i.e., lead immediately to dead positions), so some must be searched further. To limit the size of this so called quiescence search, only dynamic moves are selected for consideration. These might be as few as the moves that are part of a single complex capture, but can expand to include all capturing moves and all responses to check [43]. Ideally, passed pawn moves (especially those close to promotion) and selected checks should be included [21, 25], but these are often only examined in computationally simple endgames. The goal is always to clarify the node so that a more accurate position evaluation is made. Despite the obvious benefits of these ideas, the realm of quiescence search is unclear; because no theory for selecting and limiting the participation of moves exists. Present quiescent search methods are attractive because they are simple, but from a chess standpoint they leave much to be desired, especially when it comes to handling forking moves and mate threats. Even though the current approaches are reasonably effective, a more sophisticated method of extending the search, or of identifying relevant moves to participate in the selective quiescence search, is needed [45]. On the other hand, *Sargon* managed quite well without quiescence search, using direct computation to evaluate the exchange of material [46].

2.9. Horizon Effect

An unresolved defect of chess programs is the insertion of delaying moves that cause any inevitable loss of material to occur beyond the program’s horizon (maximum search depth), so that the loss is hidden [33]. The "horizon effect" is said to occur when the delaying moves give up additional material to postpone the eventual loss. The effect is less apparent in programs with more knowledgeable quiescence searches [45], but all programs exhibit this phenomenon. There are many illustrations of the difficulty; the example in Figure 4, which is based on a study by Kaindl [45], is clear. Here a program with a simple quiescence search involving only captures would assume that any blocking move saves the queen. Even an 8-ply search (b3-b2, Bxb2; c4-c3, Bxc3; d5-d4, Bxd4; e6-e5, Bxe5) would not see the inevitable, "thinking" that the queen has been saved at the expense of four pawns! Thus programs with a poor or inadequate quiescence search suffer more from the horizon effect. The best way to provide automatic extension of non-quiescent positions is still an open question, despite proposals such as bandwidth heuristic search [47].

	: :		: :		R b		K b
: :		: :		Q w	P b	Q b	
	: :		: :	P b	: :		: :
: :	P b	: :	P b	: :		P w	
	: :	P b	: :		P w		: :
: :	P b	: :		P w		: :	
	: :		P w		: :		: :
B w	K w	: :		: :		: :	

Black to move

Figure 4: The Horizon Effect.

2.10. Progressive and Iterative Deepening

The term progressive deepening was used by de Groot [6] to encompass the notion of selectively extending the main continuation of interest. This type of selective expansion is not performed by programs employing the alpha-beta algorithm, except in the sense of increasing the search depth by one for each checking move on the current continuation (path from root to horizon), or by performing a quiescence search from horizon nodes until dead positions are reached.

In the early 1970's several people tried a variety of ways to control the exponential growth of the tree search. A simple fixed depth search is inflexible, especially if it must be completed within a specified time. Jim Gillogly, author of *Tech* [43], coined the term iterative deepening to distinguish a full-width search to increasing depths from the progressively more focused search described by de Groot. About the same time David Slate and Larry Atkin sought a better time control mechanism, and introduced the notion of an iterated search [36] for carrying out a progressively deeper and deeper analysis. For example, an iterated series of 1-ply, 2-ply, 3-ply ... searches is carried out, with each new search first retracing the best path from the previous iteration and then extending the search by one ply. Early experimenters with this scheme were surprised to find that the iterated search often required less time than an equivalent direct search. It is not immediately obvious why iterative deepening is effective; as indeed it is not, unless the search is guided by the entries in a transposition table (or the more specialized refutation table), which holds the best moves from subtrees traversed during the previous iteration. All the early experimental evidence indicated that the overhead cost of the preliminary D-1 iterations was often recovered through a reduced cost for the D-ply search. Later the efficiency of iterative deepening was quantified to assess various refinements, especially memory table assists [37]. Today the terms progressive and iterative deepening are often used synonymously.

2.11. Transposition and Refutation Tables

The results (merit, best move, status) of the searches of nodes (subtrees) in the tree can be held in a large hash table [16, 36, 48]. Such a table serves several purposes, but primarily it enables recognition of move transposition, leading to a subtree that has been seen before, and so eliminate the need to search. Thus, successful use of a transposition table is an example of exact forward pruning. Many programs also store their opening book, where different move orders are common, in a way that is compatible with access to the transposition table. Another important purpose of a transposition table is as an implied move re-ordering mechanism. By trying first the available move in the table, an expensive move generation may be avoided [48].

By far the most popular table-access method is the one proposed by Zobrist [49]. He observed that a chess position constitutes placement of up to 12 different piece types {K,Q,R,B,N,P,-K ... -P} on to a 64-square board. Thus a set of 12x64 unique integers (plus a few more for en passant and castling privileges), $\{R_i\}$, may be used to represent all the possible piece/square combinations. For best results these integers should be at least 32 bits long, and be randomly independent of each other. An index of the position may be produced by doing an exclusive-or on selected integers as follows:

$$P_j = R_a \ xor \ R_b \ xor \ \cdots \ xor \ R_x$$

where the R_a etc. are integers associated with the piece placements. Movement of a "man" from the piece-square associated with R_f to the piece-square associated with R_t yields a new index

$$P_k = (P_j \ xor \ R_f) \ xor \ R_t$$

One advantage of hash tables is the rapid access that is possible, and for further speed and simplicity only a single probe of the table is normally made. More elaborate schemes have been tried, but often the cost of the increased complexity of managing the table swamps any benefits from improved table usage. Table 1 shows the usual fields of each entry in the hash table. Figure 5 contains sample pseudo code showing how the entries Move, Merit, Flag and Height are used. Not shown there are functions Retrieve and Store, which access and update the transposition table.

Lock	To ensure the table position is identical to the tree position.
Move	Best move in the position, determined from a previous search.
Merit	Value of subtree, computed previously.
Flag	Indicates whether merit is upper bound, lower bound or true merit.
Height	Length of subtree upon which merit is based.

Table 1: Typical Transposition Table Entry.

A transposition table also identifies the preferred move sequences used to guide the next iteration of a progressive deepening search. Only the move is important in this phase, since the subtree length is usually less than the remaining search depth. Transposition tables are particularly advantageous to methods like PVS, since the initial minimal window search loads the table with useful lines that are used in the event of a re-search. On the other hand, for deeper searches, entries are commonly lost as the table is overwritten, even though the table may contain more than a million entries [50]. Under these conditions a small fixed size transposition table may be overused (overloaded) until it is ineffective as a means of storing the continuations. To overcome this fault, a special table for holding these main continuations (the refutation lines) is also used. The table has W entries containing the D elements of each continuation. For shallow searches (D < 6) a refutation table guides a progressive deepening search just as well as a transposition table. In fact, a refutation table is the preferred choice of commercial systems or users of memory limited processors. A small triangular workspace (DxD/2 entries) is needed to hold the current continuation as it is generated, and these entries in the workspace can also be used as a source of killer moves [51].

2.12. Summary

The various terms and techniques described have evolved over the years. The superiority of one method over another often depends on how the elements are combined. The utility of iterative deepening, aspiration search, PVS, and transposition and refutation tables is perhaps best summarized by a revised version of an established performance graph [37], Figure 6. That graph was made from data gathered by a simple chess program when analyzing the twenty-four standard positions of the Bratko-Kopec test [24]. Analysis of those positions requires the search of trees whose nodes have an average width of W = 34 branches. Thus it is possible to use the formula for the terminal (horizon) nodes in a uniform minimal game tree as an estimate of the lower bound on the search size, see Figure 6. For the results presented in Figure 6 the transposition table was fixed at eight thousand entries, so that the effects of table overloading may be seen. Figure 6 shows that:

(a). Iterative deepening has low cost, so is useful as a time-control mechanism.
(b). PVS is superior to aspiration search.
(c). A refutation table is a space efficient alternative to a transposition table for guiding both the next iteration and a re-search.
(d). Odd-ply alpha-beta searches are more efficient than even-ply ones.
(e). Transposition table size must increase with depth of search.
(f). Transposition and/or refutation tables plus the history heuristic are an effective combination, achieving search results close to the minimal game tree for odd-ply search depths.

```
FUNCTION AB (p : position; alpha, beta, depth : integer) : integer;
    VAR value, height, merit : integer;
        j, move : 1..MAXWIDTH ;
        flag : (VALID, LBOUND, UBOUND);
        posn : ARRAY [1..MAXWIDTH] OF position;
BEGIN
         { retrieve merit and best move for the current position }
    Retrieve(p, height, merit, flag, move);

                        {  height is the effective subtree length.  }
                        {  height < 0 - position not in table.      }
                        {  height ≥1 0 - position in table.      }

  IF (height ≥ depth) THEN BEGIN
    IF (flag = VALID) THEN
      Return(merit);
    IF (flag = LBOUND) THEN
      alpha := max(alpha, merit);
    IF (flag = UBOUND) THEN
      beta := min(beta, merit);
    IF (alpha ≥ beta) THEN
      Return(merit);
  END;
             {  Note: update of the alpha or beta bound  }
             {  is not valid in a selective search.      }
             {  If merit in table insufficient to end    }
             {  search try best move (from table) first, }
             {  before generating other moves.           }

  IF (depth = 0) THEN                  {  horizon node?  }
    Return(Evaluate(p));
  IF (height ≥ 0) THEN BEGIN
                          { first try move from table }
    merit := -AB (posn[move], -beta, -alpha, depth-1);
    IF (merit ≥ beta) THEN
      GOTO done;
  END ELSE merit := -MAXINT;
                       {  No cut-off, generate moves   }
  posn := Generate(p);
  IF empty(posn) THEN            {  leaf, mate or stalemate?  }
    Return(Evaluate(p));

  FOR j := 1 TO sizeof(posn) DO
  IF j ≠ move THEN BEGIN
                          { using fail-soft condition }
    value := -AB (posn[j], -beta, -max(alpha,merit), depth-1);
    IF (value > merit) THEN BEGIN
      merit := value;
      move := j;
      IF (merit ≥ beta) THEN
        GOTO done;
    END;
  END;
done:
  flag := VALID;
  IF (merit ≤ alpha) THEN
    flag := UBOUND;
  IF (merit ≥ beta) THEN
    flag := LBOUND;
  IF (height ≤ depth) THEN             { update hash table }



    Store(p, depth, merit, flag, move);
  Return(merit);
END;
```

Figure 5: Alpha-beta with Transposition Table.

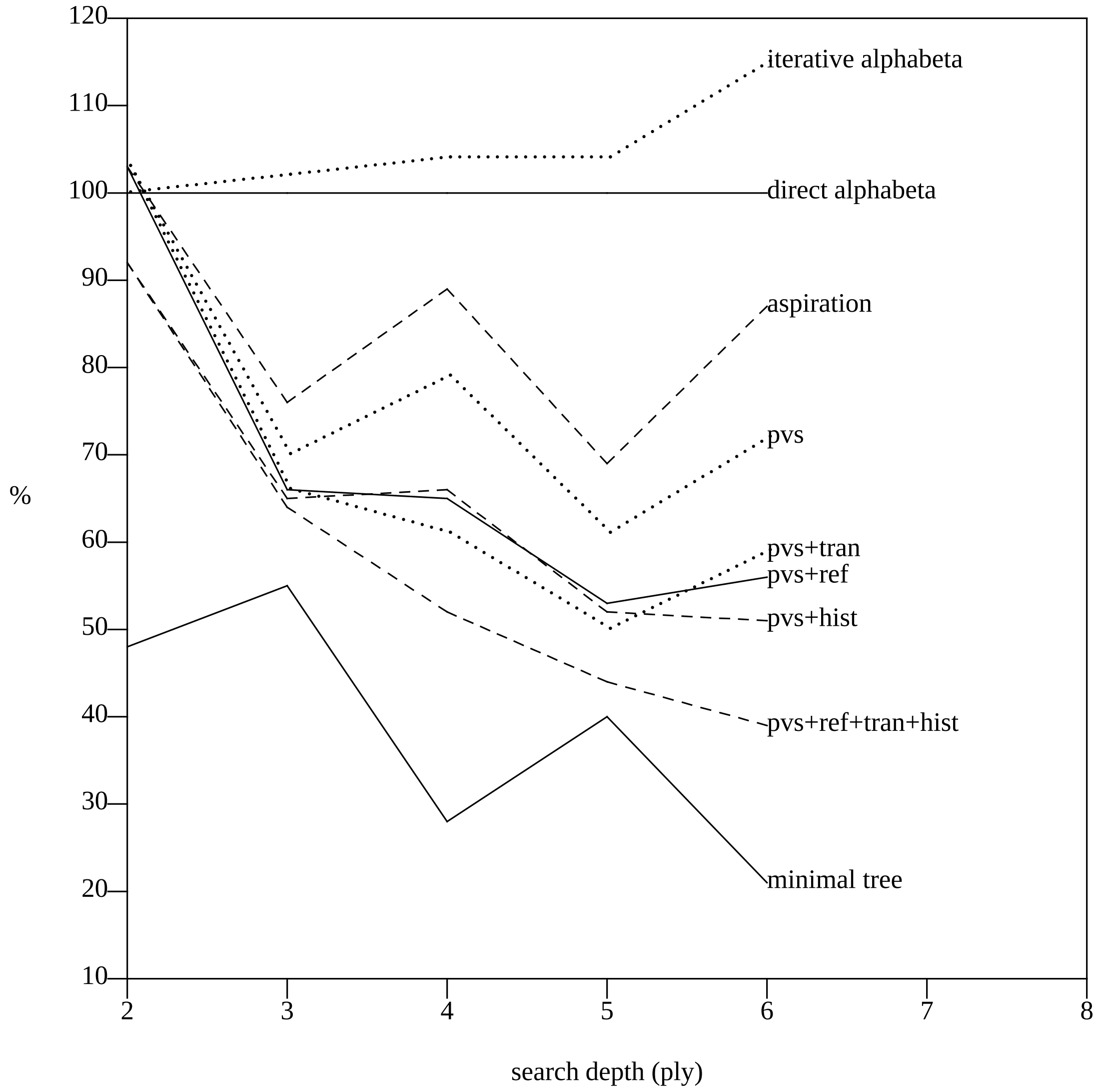

Figure 6: Node Comparison of Alpha-beta Enhancements

% Performance Relative to a Direct Alphabeta Search

3. STRENGTHS AND WEAKNESSES

3.1. Anatomy of a chess program

A typical chess program contains the following three distinct elements: Board description and move generation, tree searching/pruning, and position evaluation. Many people have based their first chess program on Larry Atkin's instructive Pascal-based model [52]. Although several good proposals exist in readily available books [14, 20] and articles [53, 54], the most efficient way of representing all the tables and data structures necessary to describe a chess board is not yet known. From these tables the move list for each position can be generated. Sometimes the Generate function produces all the feasible moves at once, which has the advantage that the moves may be sorted to improve the probability of a cut-off. In small memory computers, on the other hand, the moves are produced one at a time. This saves space and perhaps time whenever an early cut-off occurs. Since only limited sorting is possible (captures might be generated first) the searching efficiency is generally lower, however.

In the area of searching/pruning methods, variations on the depth-limited alpha-beta algorithm remain the preferred choice. All chess programs fit the following general model. A full width "exhaustive" search (all moves are considered) is done at the first few ply from the root node. At depths beyond this exhaustive layer some form of selective search is used. Typically, unlikely or unpromising moves are simply dropped from the move list. More sophisticated programs carry out an extensive analysis to select those moves which are to be discarded at an interior node. Even so, this type of forward pruning is known to be error prone and dangerous; it is attractive because of the big reduction in tree size that ensues. Finally, the Evaluate function is invoked at the horizon nodes to assess the merits of the moves. Many of these are captures or other forcing moves which are not "dead," and so a limited quiescence search is carried out to resolve the unknown potential of the move. The evaluation process is the most important part of a chess program, because it estimates the value of the subtrees that extend beyond the horizon. Although in the simplest case Evaluate simply counts the material balance, for superior play it is also necessary to measure many positional factors, such as pawn structures. These aspects are still not formalized, but adequate descriptions by computer chess practitioners are available in books [14, 36].

3.2. Hardware Advances

Computer chess has consistently been in the forefront of the application of high technology. With *Cheops* [55], the 1970's saw the introduction of special purpose hardware for chess. Later networks of computers were tried; in New York, 1983, *Ostrich* used an eight processor Data General system [56] and *Cray Blitz* a dual processor Cray X-MP [21]. Some programs used special purpose hardware (see for example *Belle* [57, 58], *Bebe*, *Advance 3.0* and *BCP* [14]), and there were several experimental commercial systems employing custom VLSI chips. This trend towards the use of custom chips will continue, as evidenced by the success of the latest master-calibre chess program, *Hitech* from Carnegie-Mellon University, based on a new chip for generating moves [59]. While mainframes will continue to be faster for the near future, it is only a matter of time before massive parallelism is applied to computer chess. The problem is a natural demonstration piece for the power of distributed computation, since it is processor intensive and the work can be partitioned in many ways. Not only can the game trees be split into similar subtrees, but parallel computation of such components as move generation, position evaluation, and quiescence search is possible.

Improvements in hardware speed have been an important contributor to computer chess performance. These improvements will continue, not only through faster special purpose processors, but also by using many processing elements.

3.3. Software Advances

Many observers attributed the advances in computer chess through the 1970's to better hardware, particularly faster processors. Much evidence supports that point of view, but major improvements also stemmed from a better understanding of quiescence and the horizon effect, and a better encoding of chess knowledge. The benefits of aspiration search [43], iterative deepening [36] (especially when used with a refutation table [51]), the killer heuristic [43] and transposition tables [16, 36] were also appreciated, and by 1980 all were in general use. One other advance was the simple expedient of "thinking on the opponent's time" [43], which involved selecting a response for the opponent, usually the move predicted by the computer, and searching the predicted position for the next reply. Nothing is lost by this tactic, and when a successful prediction is made the time saved may be accumulated until it is necessary or possible to do a deeper search. Anticipating the opponent's response has been embraced by all microprocessor based systems, since it increases their effective speed.

Not all advances work out in practice. For example, in a test with *Kaissa* the method of analogies

> "reduced the search by a factor of 4 while the time for studying one position was increased by a factor of 1.5" [60].

Thus a dramatic reduction in the positions evaluated occurred, but the total execution time went up and so the method was not effective. This sophisticated technique has not been tried in other competitive chess programs. The essence of the idea is that captures in chess are often invariant with respect to several minor moves. That is to say, some minor moves have no influence on the outcome of a specific capture. Thus the true results of a capture need be computed only once, and stored for immediate use in the evaluation of other positions that contain this identical capture! Unfortunately, the relation (sphere of influence) between a move and those pieces involved in a capture is complex, and it can be as much work to determine this relationship as it would be to simply re-evaluate the exchange. However, the method is elegant and appealing on many grounds and should be a fruitful area for further research, as a promising variant restricted to pawn moves illustrates [61].

3.4. Endgame Play

During the 1970's there developed a better understanding of the power of pawns in chess, and a general improvement in endgame play. Even so, endgames remained a weak feature of computer chess. Almost every game illustrated some deficiency, through inexact play or conceptual blunders. More commonly, however, the programs were seen to wallow and move pieces aimlessly around the board. A good illustration of such difficulties is a position from a game between *Duchess* and *Chaos* (Detroit, 1979), which was analysed extensively in an appendix to a major reference [20].

	: :		: :		Kb		Bb	Chaos
: :		: :		: :		: :	Pw	
	: :	Bw	: :	Kw	Pw		Pw	
: :		: :		: :		: :		
	: :		: :		: :		: :	
Pb		: :		: :		: :		
Pw	: :		: :		: :		: :	
: :		: :		: :		: :		Duchess

White to move

Figure 7: Lack of Endgame Plan.

After more than ten hours of play the position in Figure 7 was reached, and since neither side was making progress the game was adjudicated after white's 111th move of Bc6-d5. White had just completed a sequence of 21 reversible moves with only the bishop, and black had responded correctly by simply moving the king to and fro. *Duchess* had only the most rudimentary plan for winning endgames. Specifically, it knew about avoiding a 50-move rule draw. Had the game continued, then within the next 29 moves it would either play an irreversible move like Pf6-f7, or give up the pawn on f6. Another 50-move cycle would then ensue and perhaps eventually the possibility of winning the pawn on a3 might be found. Even six years later I doubt if many programs could handle this situation any better. There is simply nothing much to be learned through search. What is needed here is some higher notion involving goal seeking plans. All the time a solution must be sought which avoids a draw. This latter aspect is important since in many variations black can simply offer the sacrifice bishop takes pawn on f6, because if the white king recaptures a stalemate results.

Sometimes, however, chess programs are supreme. At Toronto in 1977, in particular, *Belle* demonstrated a new strategy for defending the lost ending KQ vs KR against chess masters. While the ending still favors the side with the queen, precise play is required to win within 50 moves, as several chess masters were embarrassed to discover. In speed chess *Belle* also often dominates masters, as many examples in the literature show [20]. Increasingly, chess programs are teaching even experts new tricks and insights. As long ago as 1970 Thomas Strohlein built a database to find optimal solutions to several simple three and four piece endgames (kings plus one or two pieces) [62]. Using a Telefunken TR4 (48-bit word, 8 μsec. operations) he obtained the results summarized in Table 2. Many other early workers on endgames built databases of the simplest endings. Their approach was to develop optimal sequences backward from all possible winning positions (mate or reduction to a known subproblem) [63, 64]. These works have recently been reviewed and put into perspective [65].

Pieces	Moves	Computation time
Queen	10	6.5 mins.
Rook	16	9 mins.
Rook vs Bishop	18	6 hours 30 mins.
Rook vs Knight	27	14 hours 16 mins.
Queen vs Rook	31	29 hours 9 mins.

Table 2: Maximum Moves to Win Simple Endgames.

The biggest contributions to chess theory, however, have been made by *Belle* and Ken Thompson. They have built databases to solve five piece endgames. Specifically, KQX vs KQ (where X = Q, R, B or N), KRX vs KR and KBB vs KN. This last case may prompt another revision to the 50-move rule, since in general KBB vs KN is won (not drawn) and less than 67 moves are needed to mate or safely capture the knight [66]. Also completed is a major study of the complex KQP vs KQ ending. Again, often more than 50 manoeuvres are required before a pawn can advance [66]. For more complex endings involving several pawns, the most exciting new ideas are those on chunking. Based on these ideas, it is claimed that the "world's foremost expert" has been generated for endings where each side has a king and three pawns [67, 68].

3.5. Memory Tables

Others have pointed out [36, 50] that a hash table can also be used to store information about pawn formations. Since there are usually far more moves by pieces than by pawns, the value of the base pawn formation for a position must be re-computed several times. It is a simple matter to build a hash key based on the location of pawns alone, and so store the values of pawn formations in a hash table for immediate retrieval. Hyatt found this table to be effective [21], since otherwise 10-20% of the search time was taken up with evaluation of pawn structures. A high (98-99%) success rate was reported [21]. King safety can also be handled similarly [36, 50], since the king has few moves and for long periods is not under attack.

Transposition and other memory tables come into their own in endgames, since there are fewer pieces and more reversible moves. Search time reduction by a factor of five is common, and in certain types of king and pawn endings, it is claimed that experiments with *Cray Blitz* and *Belle* have produced trees of more than 30 ply, representing speedups of well over a hundred-fold. Even in complex middle games, however, significant performance improvement is observed. Thus, use of a transposition table provides an exact form of forward pruning and as such reduces the size of the search space, in endgames often to less than the minimal game tree! The power of forward pruning is well illustrated by the following study of "Problem No. 70" [69], Figure 8, which was apparently first solved [52] by *Chess 4.9* and then by *Belle*.

	: :		: :		: :		: :
Kb		: :		: :		: :	
	: :		Pb		: :		: :
Pb		: :	Pw	: :	Pb	: :	
Pw	: :		Pw		Pw		: :
: :		: :		: :		: :	
	: :		: :		: :		: :
Kw		: :		: :		: :	

White to move

Figure 8: Transposition Table Necessity.

The only complete computer analysis of this position was provided later [21]. As Robert Hyatt puts it, a solution is possible because "The search tree is quite narrow due to the locked pawns." Here *Cray Blitz* is able to find the correct move of Ka1-b1 at the 18th iteration. The complete line of the best continuation was found at the 33rd iteration, after examining four million nodes in about 65 seconds of Cray-1 time. This was possible because the transposition table had become loaded with the results of draws by repetition, and so the normal exponential growth of the tree was inhibited. Also, at every iteration, the transposition table was loaded with losing defences corresponding to lengthy searches. Thus the current iteration often yielded results equivalent to a much longer 2(D-1) ply search. Ken Thompson refers to this phenomenon as "seeing over the horizon."

3.6. Selective Search

Many software advances came from a better understanding of how the various components in evaluation and search interact. The first step was a move away from selective search, by providing a clear separation between the algorithmic component, search, and the heuristic component, chess position evaluation. The essence of the selective approach is to narrow the width of search by forward pruning. Some selection processes removed implausible moves only [70], thus abbreviating the width of search in a variable manner not necessarily dependent on node level in the tree. This technique was only slightly more successful than other forms of forward pruning, and required more computation. Even so, it too could not retain sacrificial moves. So the death knell of selective search was its inability to predict the future with a static evaluation function. It was particularly susceptible to the decoy sacrifice and subsequent entrapment of a piece. Interior node evaluation functions that attempted to deal with these problems became too expensive. Even so, in the eyes of some, selective methods remain as a future prospect since

> "Selective search will always loom as a potentially faster road to high level play. That road, however, requires an intellectual break-through rather than a simple application of known techniques" [58].

The reason for this belief is that chess game trees grow exponentially with depth of search. Ultimately it will become impossible to obtain the necessary computing power to search deeper within normal time constraints. For this reason most chess programs already incorporate some form of selective search, often as forward pruning. These methods are quite ad hoc since they are not based on a theory of selective search.

Although nearly all chess programs have some form of selective search, even if it is no more than the discarding of unlikely moves, at present only two major programs (*Awit* and *Chaos*) do not consider all moves at the root node. Despite their occasional successes, these programs can no longer compete in the race for Grand Master status. Nevertheless, while the main advantage of a program that is exhaustive to some chosen search depth is its tactical strength, it has been shown that the selective approach can also be effective in tactical situations. In particular, Wilkin's *Paradise* program demonstrated superior performance in "tactically sharp middle game positions" on a standard suite of tests [71]. *Paradise* was designed to illustrate that a selective search program can also find the best continuation when there is material to be gained, though searching but a fraction of the game tree viewed by such programs as *Chess 4.4* and *Tech*. Furthermore it can do so with greater success than either program or a typical A-class player [71]. However, a nine to one speed handicap was necessary, to allow adequate time for the interpretation of the MacLisp program. *Paradise's* approach is to use an extensive static analysis to produce a small set of plausible winning plans. Once a plan is selected "it is used until it is exhausted or until the program determines that it is not working." In addition, *Paradise* can "detect when a plan has been tried earlier along the line of play and avoid searching again if nothing has changed" [71]. This is the essence of the method of analogies too. As Wilkins says the

> "goal is to build an expert knowledge base and to reason with it to discover plans and verify them within a small tree."

Although *Paradise* is successful in this regard, part of its strength lies in its quiescence search, which is seen to be "inexpensive compared to regular search," despite the fact that this search "investigates not only captures but forks, pins, multimove mating sequences, and other threats" [71]. The efficiency of the program lies in its powerful evaluation, so that usually "only one move is investigated at each node, except when a defensive move fails." Jacques Pitrat has also written extensively on the subject of finding plans that win material [72], but neither his ideas nor those in *Paradise* have been incorporated into the competitive chess programs of the 1980’s.

3.7. Search and Knowledge Errors

The following game was the climax of the 15th ACM NACCC, in which all the important programs of the day participated. Had *Nuchess* won its final match against *Cray Blitz* there would have been a 5-way tie between these two programs and *Bebe*, *Chaos* and *Fidelity X*. Such a result almost came to pass, but suddenly *Nuchess* "snatched defeat from the jaws of victory," as chess computers are prone to do. Complete details about the game are not important, but the position shown in Figure 9 was reached.

```
     ::  Nb  ::  Kb  ::  Rb  ::      Cray Blitz
 Pb      ::      ::      ::
     ::      Bw      Rw  Pb  ::
 ::      Pw      ::  Pb  Kw  Pb
     ::      ::      ::      ::
 ::      ::      Pw      Pw
     ::      ::      Pw      Pw
 ::      ::      ::      ::          Nuchess
White's Move 45.
```

Figure 9: A Costly Miscalculation.

Here, with Rf6xg6, *Nuchess* wins another pawn, but in so doing enters a forced sequence that leaves *Cray Blitz* with an unstoppable pawn on a7, as follows:

45. Rf6xg6 ? Rg8xg6+
46. Kg5xg6 Nc8xd6
47. Pc5xd6

Many explanations can be given for this error, but all have to do with a lack of knowledge about the value of pawns. Perhaps black's passed pawn was ignored because it was still on its home square, or perhaps *Nuchess* simply miscalculated and "forgot" that such pawns may initially advance two rows? Another possibility is that white became lost in some deep searches in which its own pawn promotes. Even a good quiescence search might not recognize the danger of a passed pawn, especially one so far from its destination. In either case, this example illustrates the need for knowledge of a type that cannot be obtained easily through search, yet which humans are able to see at a glance [6]. The game continued 47. ... Pa5 and white was neither able to prevent promotion nor advance its own pawn.

There are many opportunities for contradictory knowledge interactions in chess programs. Sometimes chess folklore provides ground rules which must be applied selectively. Such advice as "a knight on the rim is dim" is usually appropriate, but in special cases placing a knight on the edge of the board is sound, especially if it forms part of an attacking theme and is unassailable. Not enough work has been done to assess the utility of such knowledge and to measure its importance. Recently, Jonathan Schaeffer completed an interesting doctoral thesis [73] which addressed this issue; a thesis which could also have some impact on the way expert systems are tested and built, since it demonstrates that there is a correct order to the acquisition of knowledge, if the newer knowledge is to build effectively on the old.

3.8. Areas of Future Progress

Although most chess programs are now using all the available refinements and tables to reduce the game tree traversal time, only in the ending is it possible to search consistently less than the minimal game tree. Selective search and forward pruning methods are the only real hope for reducing further the magnitude of the search. Before this is possible, it is necessary for the programs to reason about the trees they see and deduce which branches can be ignored. Typically these will be branches which create permanent weaknesses, or are inconsistent with the current themes. The difficulty will be to do this without losing sight of tactical factors.

Improved performance will also come about by using faster computers, and through the construction of multiprocessor systems. One early multiprocessor chess program was *Ostrich* [56, 74]. Other experimental systems followed including *Parabelle* [75] and *ParaPhoenix* [76]. None of these systems, nor the strongest multiprocessor program *Cray Blitz* [21], consistently achieves more than a 5-fold speed-up, even when eight processors are used [76]. There is no apparent theoretical limit to the parallelism, but the practical restrictions are great and may require some new ideas on partitioning the work, as well as more involved scheduling methods.

Another major area of research is the derivation of strategies from databases of chess endgames. It is now easy to build expert system databases for the classical endgames involving four or five pieces. At present these databases can only supply the optimal move in any position (although a short principal continuation can be provided by way of expert advise). What is needed now is a program to deduce from these databases optimally correct strategies for playing the endgame. Here the database could either serve as a teacher of a deductive inference program, or as a tester of plans and hypotheses for a general learning program. Perhaps a good test of these methods would be the production of a program which could derive strategies for the well-defined KBB vs KN endgame. A solution to this problem would provide a great advance to the whole of Artificial Intelligence.

4. BIBLIOGRAPHY

References

1. A.G. Bell, *The Machine Plays Chess?*, Pergamon Press, Oxford, 1978.
2. D.N.L. Levy, *Chess and Computers*, Batsford Press, London, 1976.
3. T. Nemes, The Chess-Playing Machine, *Acta Technica*, Hungarian Academy of Sciences, Budapest, 1951, 215-239.
4. K. Zuse, Chess Programs, in *The Plankalkuel*, Rept. No. 106, Gesellschaft fuer Mathematik und Datenverarbeitung, Bonn, 1976, 201-244. (Translation of German original, 1945).
5. A.M. Turing, Digital Computers Applied to Games, in *Faster Than Thought*, B.V. Bowden (ed.), Pitman, London, 1953, 286-297.
6. A.D. de Groot, *Thought and Choice in Chess*, Mouton, The Hague, 1965 (2nd Edition 1978).
7. C.E. Shannon, Programming a Computer for Playing Chess, *Philosophical Magazine 41*, (1950), 256-275.
8. J. Kister, P. Stein, S. Ulam, W. Walden and M. Wells, Experiments in Chess, *J. of the ACM 4*, (1957), 174-177.
9. A. Bernstein, M. de V. Roberts, T. Arbuckle and M.A. Belsky, A Chess Playing Program for the IBM 704, *Western Joint Computer Conf. Procs.*, (New York: AIEE), Los Angeles, 1958, 157-159.
10. B. Mittman, A Brief History of Computer Chess Tournaments: 1970-1975, in *Chess Skill in Man and Machine*, P. Frey (ed.), Springer-Verlag, 1st edition 1977, 1-33.
11. A. Kotok, A Chess Playing Program for the IBM 7090, B.S. Thesis, MIT, AI Project Memo 41, Computation Center, Cambridge MA, 1962.
12. G.M. Adelson-Velskii, V.L. Arlazarov, A.R. Bitman, A.A. Zhivotovskii and A.V. Uskov, Programming a Computer to Play Chess, *Russian Math. Surveys 25*, (Mar-Apr 1970), 221-262, Cleaver-Hume Press, London. (Translation of Proc. 1st summer school Math. Prog. v 2, 1969, 216-252).
13. J.E. Hayes and D.N.L. Levy, *The World Computer Chess Championship*, Edinburgh Univ. Press, Edinburgh, 1976.
14. D.E. Welsh and B. Baczynskyj, *Computer Chess II*, W.C. Brown Co., Dubuque, Iowa, 1985.
15. H.J. van den Herik, Computerschaak, Schaakwereld en Kunstmatige Intelligentie, Ph.D. Thesis, Technische Hogeschool Delft, Academic Service, 's-Gravenhage, The Netherlands, 1983.
16. R.D. Greenblatt, D.E. Eastlake III and S.D. Crocker, The Greenblatt Chess Program, *Fall Joint Computing Conf. Procs. 31*, (New York: ACM), San Francisco, 1967, 801-810.
17. I.J. Good, A Five-Year Plan for Automatic Chess, in *Machine Intelligence 2*, E. Dale and D. Michie (ed.), Elsevier, New York, 1968, 89-118.
18. M.M. Newborn, *Computer Chess*, Academic Press, New York, 1975.

19. T.R. Truscott, Techniques used in Minimax Game-playing Programs, M.S. thesis, Duke University, Durham NC, April, 1981.

20. P.W. Frey (editor), *Chess Skill in Man and Machine*, Springer-Verlag, New York, 2nd Edition 1983.

21. R.M. Hyatt, A.E. Gower and H.L. Nelson, Cray Blitz, in *Advances in Computer Chess 4*, D. Beal (ed.), Pergamon Press, Oxford, 1985, 8-18.

22. D. Levy and M. Newborn, *More Chess and Computers*, Computer Science Press, Rockville MD, 2nd Edition 1981.

23. A.E. Elo, *The Rating of Chessplayers, Past and Present*, Arco Publishing, New York, 1978.

24. D. Kopec and I. Bratko, The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess, in *Advances in Computer Chess 3*, M. Clarke (ed.), Pergamon Press, Oxford, 1982, 57-72.

25. K. Thompson, Computer Chess Strength, in *Advances in Computer Chess 3*, M. Clarke (ed.), Pergamon Press, Oxford, 1982, 55-56.

26. D. Michie, Chess with Computers, *Interdisciplinary Science Reviews 5(3)*, (1980), 215-227.

27. A.L. Samuel, Some Studies in Machine Learning Using the Game of Checkers, *IBM J. of Res. & Dev. 3*, (1959), 210-229. (Also in Computers & Thought, E.Feigenbaum and J.Feldman (eds.), McGraw-Hill, l963, 71-105).

28. A.G. Bell, *Games Playing with Computers*, Allen and Unwin, London, 1972.

29. A. Newell, J.C. Shaw and H.A. Simon, Chess Playing Programs and the Problem of Complexity, *IBM J. of Res. & Dev. 4(2)*, (1958), 320-335. (Also in Computers and Thought, E.Feigenbaum and J.Feldman (eds.), McGraw-Hill, 1963, 39-70).

30. A.L. Brudno, Bounds and Valuations for Abridging the Search of Estimates, *Problems of Cybernetics 10*, (1963), 225-241, Pergamon Press. (Translation of Russian original in Problemy Kibernetiki Vol. 10, May 1963, 141-150).

31. D.E. Knuth and R.W. Moore, An Analysis of Alpha-beta Pruning, *Artificial Intelligence 6(4)*, (1975), 293-326.

32. J.P. Fishburn, *Analysis of Speedup in Distributed Algorithms*, UMI Research Press, Ann Arbor, Michigan, 1984. Comp. Sci. report Tech. Rep. 431, Univ. of Wisconsin, May 1981.

33. H.J. Berliner, Some Necessary Conditions for a Master Chess Program, *Procs. 3rd Int. Joint Conf. on Art. Intell.*, (Menlo Park: SRI), Stanford, 1973, 77-85.

34. H.J. Berliner, Chess as Problem Solving: The Development of a Tactics Analyzer, Ph.D. Thesis, Carnegie-Mellon University, Pittsburg, March 1974.

35. J.J. Gillogly, Performance Analysis of the Technology Chess Program, Tech. Rept. CMU-CS-78-189, Computer Science, Carnegie-Mellon University, Pittsburg, March 1978.

36. D.J. Slate and L.R. Atkin, CHESS 4.5 - The Northwestern University Chess Program, in *Chess Skill in Man and Machine*, P. Frey (ed.), Springer-Verlag, 1st edition 1977, 82-118.

37. T.A. Marsland, Relative Efficiency of Alpha-beta Implementations, *Procs. 8th Int. Joint Conf. on Art. Intell.*, (Los Altos: Kaufmann), Karlsruhe, West Germany, Aug. 1983, 763-766.

38. J. Pearl, Asymptotic Properties of Minimax Trees and Game Searching Procedures, *Artificial Intelligence 14(2)*, (1980), 113-138.

39. A. Reinefeld, J. Schaeffer and T.A. Marsland, Information Acquisition in Minimal Window Search, *Procs. 9th Int. Joint Conf. on Art. Intell.*, (Los Altos: Kaufmann), Los Angeles, Aug. 1985, 1040-1043.

40. P.W. Frey, An Introduction to Computer Chess, in *Chess Skill in Man and Machine*, P. Frey (ed.), Springer-Verlag, New York, 1977, 54-81.

41. J.R. Slagle, *Artificial Intelligence: The Heuristic Programming Approach*, McGraw-Hill, New York, 1971.

42. J.A. Birmingham and P. Kent, Tree-searching and Tree-pruning Techniques, in *Advances in Computer Chess 1*, M. Clarke (ed.), Edinburgh Univ. Press, Edinburgh, 1977, 89-107.

43. J.J. Gillogly, The Technology Chess Program, *Artificial Intelligence 3(1-4)*, (1972), 145-163.

44. J. Schaeffer, The History Heuristic, *ICCA Journal 6(3)*, (1983), 16-19.

45. H. Kaindl, Dynamic Control of the Quiescence Search in Computer Chess, in *Cybernetics and Systems Research*, R. Trappl (ed.), North-Holland, Amsterdam, 1982, 973-977.

46. D. Spracklen and K. Spracklen, An Exchange Evaluator for Computer Chess, *Byte*, Nov. 1978, 16-28.

47. L.R. Harris, The Heuristic Search and the Game of Chess, *Procs. 4th Int. Joint Conf. on Art. Intel.*, (Cambridge: MIT), Tbilisi, Sept. 1975, 334-339.

48. T.A. Marsland and M. Campbell, Parallel Search of Strongly Ordered Game Trees, *Computing Surveys 14(4)*, (1982), 533-551.

49. A.L. Zobrist, A Hashing Method with Applications for Game Playing, Technical Report 88, Computer Sciences Dept., Univ. of Wisconsin, Madison WI, April, 1970.

50. H.L. Nelson, Hash Tables in Cray Blitz, *ICCA Journal 8(1)*, (1985), 3-13.

51. S.G. Akl and M.M. Newborn, The Principal Continuation and the Killer Heuristic, *1977 ACM Ann. Conf. Procs.*, (New York: ACM), Seattle, Oct. 1977, 466-473.

52. P.W. Frey and L.R. Atkin, Creating a Chess Player, in *The BYTE Book of Pascal*, B.L. Liffick (ed.), BYTE/McGraw-Hill, Peterborough NH, 2nd Edition 1979, 107-155.

53. A.G. Bell, Algorithm 50: How to Program a Computer to Play Legal Chess, *Computer Journal 13(2)*, (1970), 208-219.

54. S.M. Cracraft, Bitmap Move Generation in Chess, *ICCA Journal 7(3)*, (1984), 146-152.

55. J. Moussouris, J. Holloway and R. Greenblatt, CHEOPS: A Chess-oriented Processing System, in *Machine Intelligence 9*, J. Hayes, D. Michie and L. Michulich (ed.), Ellis Horwood, Chichester, 1979, 351-360.

56. M. Newborn, A Parallel Search Chess Program, *Procs. ACM Ann. Conf.*, (New York: ACM), Denver, Oct 1985, 272-277.

57. J.H. Condon and K. Thompson, Belle Chess Hardware, in *Advances in Computer Chess 3*, M. Clarke (ed.), Pergamon Press, Oxford, 1982, 45-54.

58. J.H. Condon and K. Thompson, Belle, in *Chess Skill in Man and Machine*, P. Frey (ed.), Springer-Verlag, 2nd Edition 1983, 201-210.

59. C. Ebeling and A. Palay, The Design and Implementation of a VLSI Chess Move Generator, *11th Ann. Int. Symp. on Comp. Arch.*, (New York: IEEE), Ann Arbor, June 1984, 74-80.

60. G.M. Adelson-Velsky, V.L. Arlazarov and M.V. Donskoy, Algorithms of Adaptive Search, in *Machine Intelligence 9*, J. Hayes, D. Michie and L. Michulich (ed.), Ellis Horwood, Chichester, 1979, 373-384.

61. H. Horacek, Knowledge-based Move Selection and Evaluation to Guide the Search in Chess Pawn Endings, *ICCA Journal 6(3)*, (1983), 20-37.

62. T. Strohlein, Untersuchungen uber Kombinatorische Speile, Doctoral Thesis, Technischen Hochschule Munchen, Munich, West Germany, Jan. 1970.

63. M.A. Bramer and M.R.B. Clarke, A Model for the Representation of Pattern-knowledge for the Endgame in Chess, *Int. J. Man-Machine Studies 11*, (1979), 635-649.

64. I. Bratko and D. Michie, A Representation for Pattern-knowledge in Chess Endgames, in *Advances in Computer Chess 2*, M. Clarke (ed.), Edinburgh Univ. Press, Edinburgh, 1980, 31-56.

65. H.J. van den Herik and I.S. Herschberg, The Construction of an Omniscient Endgame Database, *ICCA Journal 8(2)*, (1985), 66-87.

66. K. Thompson, Retrograde Analysis of Certain Endgames, *ICCA Journal 9(3)*, (1986), 131-139.

67. Murray Campbell, A Chess Program That Chunks, *Proc. of Nat. Conf. on Art. Intell.*, (Los Altos: Kaufmann), Washington, Aug. 1983, 49-53.

68. H. Berliner and M. Campbell, Using Chunking to Solve Chess Pawn Endgames, *Artificial Intelligence 23(1)*, (1984), 97-120.

69. R. Fine, *Basic Chess Endings*, David McKay, New York, 1941.

70. E.W. Kozdrowicki and D.W. Cooper, COKO III: The Cooper-Kozdrowicki Chess Program, *Int. J. Man-Machine Studies 6*, (1974), 627-699.

71. David Wilkins, Using Chess Knowledge to Reduce Speed, in *Chess Skill in Man and Machine*, P. Frey (ed.), Springer-Verlag, 2nd Edition 1983, 211-242.

72. J. Pitrat, A Chess Combination Program which uses Plans, *Artifical Intelligence 8(3)*, (1977), 275-321.

73. J. Schaeffer, Experiments in Search and Knowledge, Ph.D. thesis, Univ. of Waterloo, Waterloo, Canada, Spring 1986 .

74. M. Newborn, OSTRICH/P - a parallel search chess program, Tech. Rep. SOCS 82.3, Computer Science, McGill Univ., Montreal, Canada, March 1982.

75. T.A. Marsland and F. Popowich, Parallel Game-Tree Search, *IEEE Trans. on Pattern Anal. and Mach. Intell. 7(4)*, (July 1985), 442-452.

76. T.A. Marsland, M. Olafsson and J. Schaeffer, Multiprocessor Tree-Search Experiments, in *Advances in Computer Chess 4*, D. Beal (ed.), Pergamon Press, Oxford, 1985, 37-51.

5. ABBREVIATIONS

1. IEEE: The Institute of Electrical and Electronics Engineers, 345 E. 47th St., New York, 10017

2. ACM: The Association for Computing Machinery, 11 W 42nd St., New York 10036.

3. ICCA Journal: The International Computer Chess Association Journal, Published by the Dept. of Math. and Informatics, Delft Technical Univ., DELFT, The Netherlands.

4. IFIP: International Federation for Information Processing.

5. AFIPS: American Federation of Information Processing Societies.

6. IJCAI: International Joint Conference on Artificial Intelligence.

7. ACM NACCC: ACM North American Computer Chess Championship.

8. SRI: Stanford Research Institute, Menlo Park CA.

9. AIEE: American Institute of Electrical Engineers (now IEEE)

10. AAAI: American Association for Artificial Intelligence, 445 Burgess Drive, Menlo Park, CA 94025.