COMPUTER CHESS AND SEARCH

T.A. Marsland

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

ABSTRACT

Article prepared for the 2nd edition of the ENCYCLOPEDIA OF ARTIFICIAL INTELLIGENCE, S. Shapiro (editor), to be published by John Wiley, 1992.

April 3, 1991

COMPUTER CHESS AND SEARCH

T.A. Marsland

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

1. HISTORICAL PERSPECTIVE

Of the early chess-playing machines the most famous was exhibited by Baron von Kempelen of Vienna in 1769. As is well-known, von Kempelen's machine and the others were conjurer's tricks and grand hoaxes. In contrast, around 1890 a Spanish engineer, Torres y Quevedo, designed a true mechanical player for KR vs K (king and rook against king) endgames (Bell 1978). 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. Despite the success of this electro-mechanical device, further advances on chess automata did not come until the 1940s. During that decade there was a sudden spurt of activity as several leading engineers and mathematicians, intrigued by the power of computers, began to express their ideas about computer chess. Some, like Tihamer Nemes (1951) and Konrad Zuse (1945) tried a hardware approach, but their computer-chess works did not find wide acceptance. Others, like noted scientist Alan Turing, found success with a more philosophical tone, stressing the importance of the stored program concept (Turing *et al*., 1953).[1] Today, best recognized are Adriaan de Groot's 1946 doctoral dissertation (de Groot, 1965) and the much referenced paper on algorithms for playing chess by Claude Shannon (1950), whose inspirational work provided a basis for most early chess programs. Despite the passage of time, Shannon's paper is still worthy of study.

1.1. Landmarks in Chess Program Development

The first computer-chess model in the 1950s was a hand simulation. Programs for subsets of chess followed, and the first full working program was reported in 1958. Most of the landmark papers reporting these results have now been collected together (Levy, 1988). By the mid 1960s there was an international computer-computer match, later reported by Mittman (1977), between a program backed by John McCarthy of Stanford (developed by Alan Kotok and a group of students from MIT) and one from the Institute for Theoretical and Experimental Physics (ITEP) in Moscow. The ITEP group's program won the match, and the scientists involved went on to develop *Kaissa*,[2] which became the first World Computer Chess Champion in 1974 (Hayes and Levy 1976). Meanwhile there emerged from MIT another program, *Mac Hack Six* (Greenblatt, Eastlake and Crocker, 1967), which boosted interest in artificial intelligence. Firstly, *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 1960s, spurred by the early promise of *Mac*

---

[1] The chess portion of that paper is normally attributed to Turing, the draughts (checkers) part to Strachey, and the balance to the other co-authors.

[2] Descriptions of *Kaissa*, and other chess programs not discussed here, can be found elsewhere, e.g., the books by Hayes and Levy (1976), Welsh and Baczynskyj (1985) and by Marsland and Schaeffer (1990).

Hack, several people began developing chess programs and writing proposals. Most substantial of the proposals was the twenty-nine point plan by Jack Good (1968). 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. Due mainly to Newborn's careful planning and organization this event continues today under the title ''The North American Computer Chess Championship,'' with the sponsorship of the ACM.

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 at Stockholm in 1974 and Toronto in 1977, and later independent backers such as the Linz (Austria) Chamber of Commerce for 1980, ACM New York for 1983, the city of Cologne in Germany for 1986 and AGT/CIPS for 1989 in Edmonton, Canada. In the first World Championship for computers *Kaissa* won all its games, including a defeat of the *Chaos* program that had beaten the favorite, *Chess 4.0*. An exhibition match between the new champion, *Kaissa*, and the eventual second place finisher, *Chess 4.0* the 1973 North American Champion, was drawn (Mittman, 1977). *Kaissa* was at its peak, backed by a team of outstanding experts on tree-searching methods (Adelson-Velsky, Arlazarov and Donskoy, 1988). In the second Championship at Toronto in 1977, *Chess 4.6* finished first with *Duchess* 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 exhibition match between *Chess 4.6* and *Kaissa* was won by the former, indicating that in the interim it had undergone far more development and testing, as the appendix to Frey's book shows (Frey, 1983). The 3rd World Championship at Linz in 1980 finished with a tie between *Belle* and *Chaos*. *Belle* represented the first of a new generation of hardware assists for chess, specifically support for position maintenance and evaluation, while *Chaos* was one of the few remaining selective search programs. 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. Even today, this counter-intuitive idea does not find ready acceptance in the artificial intelligence community.

At the 4th World Championship (1983 in New York) yet another new winner emerged, *Cray Blitz* (Hyatt, Gower and Nelson, 1990). More than any other, that program drew on the power of a fast computer, here a Cray XMP. Originally Blitz was a selective search program, in the sense that it used a local evaluation function to discard some moves from every position, but often the time saved was not worth the attendant risks. The availability of a faster computer made it possible for *Cray Blitz* to switch to a purely algorithmic approach and yet retain much of the expensive chess knowledge. Although a mainframe program won the 1983 event, small machines made their mark and were seen to have a great future. For instance, *Bebe* with special-purpose hardware finished second (Scherzer, Scherzer and Tjaden, 1990), and even experimental versions of commercial products did well. The 5th World Championship (1986 in Cologne) was especially exciting. At that time *Hitech*, with the latest VLSI technology for move generation, seemed all powerful (Berliner and Ebeling, 1989), but faltered in a better position against *Cray Blitz* allowing a four-way tie for first place. As a consequence, had an unknown microprocessor system, *Rebel*, capitalized on its advantages in the final round game, it would have been the first micro-system to win an open championship. Finally we come to the most recent event of this type, the 6th World Championship (1989 in Edmonton). Here the Carnegie Mellon favorite (*Deep Thought*) won convincingly, even though the program exhibited several programming errors. Still luck favors the strong, as the full report of the largest and strongest computer chess event ever held shows (Schaeffer, 1990). Although *Deep Thought* dominated the world championship, at the 20th North American Tournament that followed a bare six months later it lost a game against

Mephisto, and so only tied for first place with its deadly rival and stable-mate *Hitech*. All these programs were relying on advanced hardware technologies. *Deep Thought* was being likened to ''*Belle* on a chip'', showing how much more accessible increased speed through special integrated circuits had become.

From the foregoing one might reasonably assume that most computer chess programs have been developed in the USA, and yet for the past two decades participants form Canada have also been active and successful. Two programs, *Ostrich* and *Wita*, were at the inauguration of computer-chess tournaments at New York in 1970, and their authors went on to produce and instigate fundamental research in practical aspects of game-tree search (Campbell and Marsland, 1983; Newborn, 1988; Marsland, Reinefeld and Schaeffer, 1987). Before its retirement, *Ostrich* (McGill University) participated in more championships than any other program. Its contemporary, renamed *Awit* (University of Alberta), had a checkered career as a Shannon type-B (selective search) program, finally achieving its best result with a second place tie at New York in 1983. Other active programs have included *Ribbit* (University of Waterloo), which tied for second at Stockholm in 1974, *L'Excentrique* and *Brute Force*. By 1986 the strongest Canadian program was *Phoenix* (University of Alberta), a multiprocessor-based system using workstations (Schaeffer, 1989b). It tied for first place with three others at Cologne.

While the biggest and highest performing computers were being used in North America, European developers concentrated on microcomputer systems. Especially noteworthy are the Hegener & Glaser products based on the *Mephisto* program developed by Richard Lang of England, and the *Rebel* program by Ed Schröder from the Netherlands.

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 panel discussion at the ACM 1984 National Conference in San Francisco. At that time it was too early to give a definitive answer, since even the experts could not agree. Their responses covered the whole range of possible answers with different degrees of optimism. Monty Newborn enthusiastically supported ''in five years,'' while Tony Scherzer and Bob Hyatt held to ''about the end of the century.'' Ken Thompson was more cautious with his ''eventually, it is inevitable,'' but most pessimistic was Tony Marsland who said ''never, or not until the limits on human skill are known.'' Even so, there was a sense that production of an artificial Grandmaster 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 was presented to Ken Thompson and Joe Condon, when their *Belle* program earned a US Master rating in 1983. The second prize of $10,000 for the first program to achieve a USCF 2500 rating (players who attain this rating may reasonably aspire to becoming Grandmasters) was awarded to *Deep Thought* in August 1989 (Hsu, Anantharaman, Campbell and Nowatzyk, 1990), but the $100,000 for attaining world-champion status remains unclaimed. To sustain interest in this activity, Fredkin funds are available each year for a prize match 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 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 the Elo rating scale but also

using standard tests (Kopec and Bratko, 1982) and relative performance measures (Thompson, 1982). The ACM-sponsored competitions have provided twenty 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 how to program computers for chess, and how to encode the wealth of expert knowledge needed.

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 support for the view that computer chess is the ''*Drosophila melanogaster* (fruit fly) of machine intelligence'' (Michie, 1980).

What then has been the effect of computer chess on artificial intelligence (AI)? First, each doubter who dared assert the superiority of human thought processes over mechanical algorithms for chess has been discredited. All that remains is to remove the mysticism of the world's greatest chess players. Exactly why seemingly mechanical means have worked, when almost every method proposed by reputable AI experts failed, remains a mystery for some. Clearly hard work, direct application of simple ideas and substantial public testing played a major role, as did improvements in hardware/software support systems. More than anything, this failure of traditional AI techniques for selection in decision-making, leads to the unnatural notion that many ''intellectual and creative'' activities can be reduced to fundamental computations. Ultimately this means that computers will make major contributions to Music and Writing; indeed some will argue that they have already done so. Thus one effect of computer chess has been to force an initially reluctant acceptance of ''brute-force'' methods as an essential component in ''intelligent systems,'' and to encourage growing use of search in problem-solving and planning applications. Several articles discussing these issues appear in a recent edited volume (Marsland and Schaeffer, 1990).

2. SEARCHING FOR CHESS

Since most chess programs work by examining large game 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 in turn as the search process backs up to the root. In practice, since leaf nodes are rarely encountered, search proceeds until some limiting depth (the horizon or frontier) is reached. Each frontier node is treated as if it were terminal and its value fed back. Since computer chess is well defined, and absolute measures of performance exist, it is a useful test vehicle for measuring efficiency of new search algorithms. In the simplest case, the best algorithm is the one that visits fewest nodes when determining the expected value of a tree. For a two-person game-tree, this value, which is a least upper bound on the merit (or score) for the side to move, can be found through a minimax search. In chess, this so called minimax value is a combination of both ''MaterialBalance'' (i.e., the difference in value of the pieces held by each side) and ''StrategicBalance'' (e.g., a composite measure of such things as mobility, square control, pawn formation structure and king safety) components. Normally, an Evaluate procedure computes these components in such a way that the MaterialBalance dominates all positional factors.

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 ''leaf node'' that can be reached, under the assumption of best play by both sides. To represent a level in the tree (that is, a move by one side) the term ''ply'' was introduced by Arthur Samuel in his major paper on machine learning (Samuel, 1959). 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.

In general, a true minimax search of a game tree will be expensive since every leaf node must be visited. For a uniform tree with exactly W moves at each node, there are W^D nodes at the layer of the tree that is D ply from the root. Nodes at this deepest layer will be referred to as terminal nodes, and will serve as leaf nodes in our discussion. Some games, like Fox and Geese, produce narrow trees (fewer than 10 branches per node) that can often be expanded to true leaf nodes and solved exhaustively. In contrast, chess produces bushy trees with an average branching factor, W, of about 35 moves (de Groot, 1965). Because of the size of the game tree, it is not possible to search until a mate or stalemate position (a true 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 the search of some nodes is unnecessary.

2.2. The Alpha-Beta (α-β) 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 (Newell, Shaw and Simon, 1958), but according to Knuth and Moore (1975) 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 paper (Brudno 1963). The α-β algorithm employs lower (α) and upper (β) 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. Early versions of the α-β algorithm used only a single bound (α), and repeatedly reset the β bound to infinity, so that deep cut-offs were not achieved. To correct this flaw, Knuth and Moore (1975) introduced a recursive algorithm called F2 to prove properties about pruning in search. They also employed a ''negamax'' framework whose primary advantage is that by always passing back the negative of the subtree value, only maximizing operations are needed. Figure 1 uses a Pascal-like pseudo code to present our α-β function, AB, in the same negamax framework. Here a Return statement is the convention for exiting the function and returning the best subtree value or merit. Omitted are details of the game-specific functions Make and Undo (to update the game board), Generate (to find moves) and Evaluate (to assess terminal nodes). In the pseudo code of Figure 1, the max(α,merit) operation represents Fishburn's ''fail-soft'' condition (Fishburn, 1984), and ensures that the best available value is returned (rather than an α/β bound), even if the value lies outside the α-β window. This idea is usefully employed in some of the newer refinements to the α-β algorithm.

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

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

Figure 1: Depth-limited 'fail-soft' Alpha-Beta Function under Negamax Search.

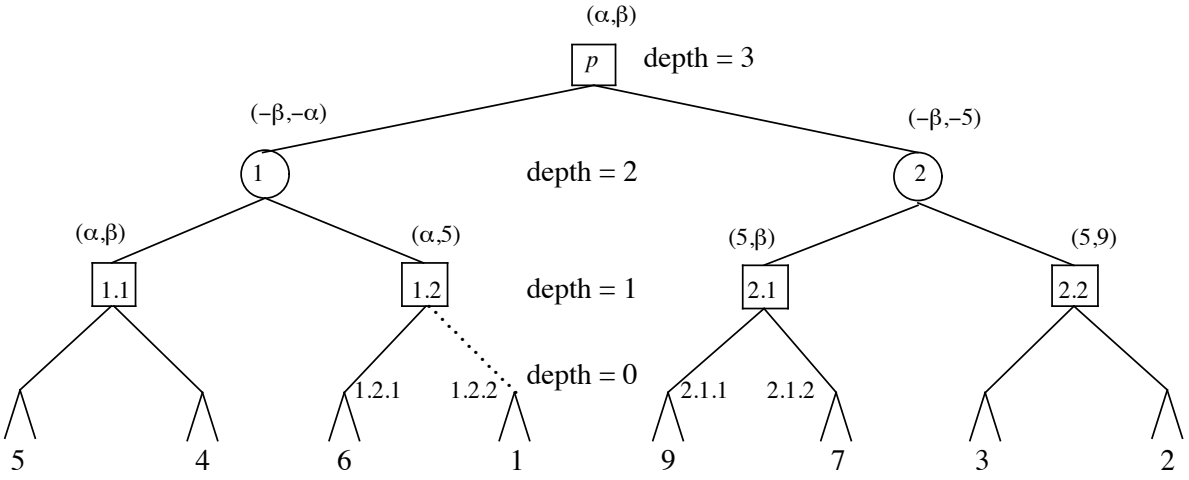


Figure 2: The Effects of α–β Pruning under Negamax Search.

Although tree-searching topics involving pruning appear routinely in standard artificial intelligence texts, game-playing programs remain the major application for the α-β algorithm. In the texts, a typical discussion about game-tree search is based on alternate use of minimizing and

maximizing operations. In practice, the negamax approach is preferred, since the programming is simpler. Figure 2 contains a small 3-ply tree in which a Dewey-decimal notation is used to label the nodes, so that the node name identifies the path from the root node. Thus, in Figure 2, p.2.1.2 is the root of a hidden subtree whose value is shown as 7. Also shown at each node of Figure 2 is the initial alpha-beta window that is employed by the negamax search. Note that successors to node p.1.2 are searched with an initial window of (α,5). Since the value of node p.1.2.1 is 6, which is greater than 5, a cut-off is said to occur, and node p.1.2.2 is not visited by the α-β algorithm.

2.3. Minimal Game Tree

If the ''best'' move is examined first at every node, the minimax value is obtained from a traversal of the minimal game tree. This minimal tree is of theoretical importance since its size is a lower bound on the search. For uniform trees of width W branches per node and a search depth of D ply, Knuth and Moore provide the most elegant proof that 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 (Knuth and Moore, 1975), where $\lceil x \rceil$ is the smallest integer $\geq x$, and $\lfloor x \rfloor$ is the largest integer $\leq x$. Since such a terminal node rarely has no successors (i.e., is not a leaf) it is often referred to as a horizon node, with D the distance from the root to the horizon (Berliner, 1973).

2.4. Aspiration Search

An α-β 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. This aspiration search method—analyzed by Brudno (1963), referred to by Berliner (1973) and experimented with by Gillogly (1978)—has been popular, though it has its problems (Kaindl, 1990). 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 3 shows. Typically these failures occur only when material is being won or lost, in which case the increased cost of a more thorough search is acceptable. 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, $+\infty$). 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, α and β are not set to $-\infty$ and $+\infty$ as one might expect, but to a window only a few pawns wide, centered roughly on the final score [merit] from the previous iteration (or previous move in the case of the first iteration). This setting of 'high hopes' increases the number of α-β cutoffs'' (Slate and Atkin, 1977).

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 (Marsland, 1983).

2.5. Quiescence Search

Even the earliest papers on computer chess recognized the importance of evaluating only positions which are ''relatively quiescent'' (Shannon, 1950) or ''dead'' (Turing *et al.*, 1953). These are positions that can be assessed accurately without further search. Typically they have no

```
{     Assume V = estimated value of position p, and    }
{            e = expected error limit                  }
{        depth = current distance to the frontier      }
{            p = position being searched               }
     α := V – e;                          { lower bound  }
     β := V + e;                          { upper bound  }

     V := AB (p, α, β, depth);
     IF (V ≥ β) THEN                  {   failing high  }
        V := AB (p, V, +∞, depth)
     ELSE
     IF (V ≤ α) THEN                  {   failing low   }
        V := AB (p, –∞, V, depth);

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

Figure 3: Narrow Window Aspiration Search.

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 (Gillogly, 1972). Ideally, passed pawn moves (especially those close to promotion) and selected checks should be included (Slate and Atkin, 1977; Hyatt, Gower and Nelson, 1985), 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 best form of the quiescence search remains unclear, although some theories for controlling the search depth and limiting the participation of moves are emerging. Present quiescent search methods are attractive; they are simple, but from a chess standpoint 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 is needed for extending the search, or for identifying relevant moves to participate in the selective quiescence search (Kaindl, 1982). A first step in this direction is the notion of a singular extension (Anantharaman, Campbell and Hsu, 1988). On the other hand, some commercial chess programs have managed well without quiescence search, using direct computation to evaluate the exchange of material. Another favored technique for assessing dynamic positions is use of the null move (Beal 1989), which assumes that there is nothing worse than not making a move!

2.6. 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 (Berliner, 1973). The ''horizon effect'' is said to occur when the delaying moves unnecessarily weaken the position or give up additional material to postpone the eventual loss. The effect is less apparent in programs with more knowledgeable quiescence searches (Kaindl, 1982), 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, 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 (..., Pb2; Bxb2, Pc3; Bxc3, Pd4; Bxd4, Pe5; Bxe5) might not show the inevitable, ''believing'' 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 (Harris, 1974).

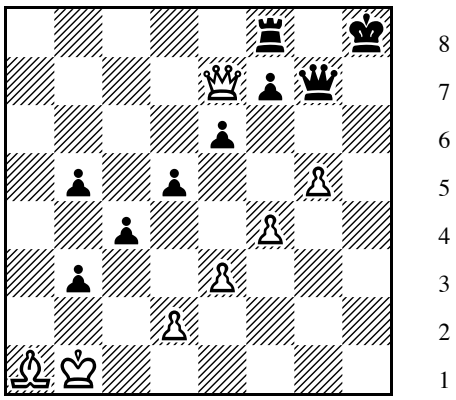


Figure 4: The Horizon Effect.

3. ALPHA-BETA ENHANCEMENTS

Although the α-β algorithm is extremely efficient in comparison to a pure minimax search, it is improved dramatically both in the general case, and for chess in particular, by heuristic move-ordering mechanisms. When the heuristically superior moves are tried first there is always a statistical improvement in the pruning efficiency. Another important mechanism is the use of an iteratively deepening search, it too has the effect of dynamically re-ordering the move list at the root position, with the idea of reducing the search to that of the minimal game tree. Iteratively deepening searches are made more effective by the use of transposition tables to store results of searches from earlier iterations and use them to guide the current search more quickly to its best result. Finally, the α-β implementation itself has a more efficient implementation, based on the notion of a minimal (null) window search to prove more quickly the inferiority of competing variations.

3.1. Minimal Window Search

Theoretical advances, such as SCOUT (Pearl, 1980) and the comparable minimal window search techniques (Fishburn, 1984; Marsland, 1983; Campbell and Marsland, 1983) came in the late 1970's. 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 (Marsland, 1983), for random game trees it is known that even these refinements are asymptotically equivalent to the simpler α-β algorithm. Bushy trees are typical for chess and so many contemporary chess programs use minimal window techniques through the Principal Variation Search (PVS) algorithm (Marsland and Campbell, 1982). In Figure 5, a Pascal-like pseudo code is used to describe PVS in a negamax framework. The chess-specific functions Make and Undo have been omitted for clarity. Also, the original version of PVS has been improved by using Reinefeld's depth=2 idea, which shows that re-searches need only be performed when the remaining depth of search is greater than 2. This point, and the general advantages of PVS, is illustrated by Figure 6, which shows the traversal of the same tree presented in Figure 2. Note that using narrow windows to prove the inferiority of the subtrees leads to the pruning of an additional frontier node (the node p.2.1.2). This is typical of the savings that are possible, although there is a risk that some subtrees will have to be re-searched.

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

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

Figure 5: Minimal Window Principal Variation Search.

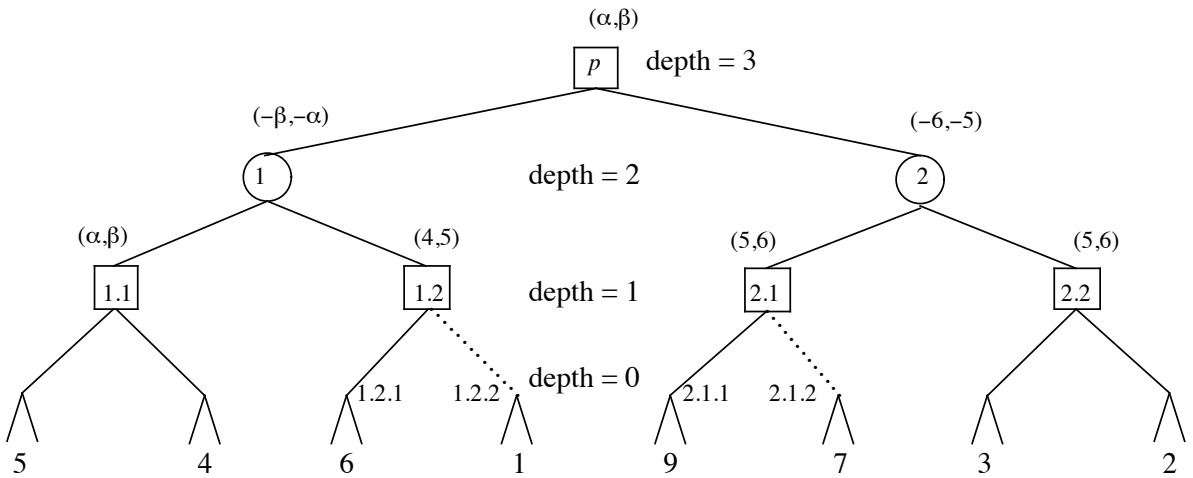


Figure 6: The Effects of PVS Pruning (Negamax Framework).

3.2. 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 (Greenblatt, Eastlake and Crocker, 1967) considers only the N-best moves at each node, where N usually decreases with increasing depth of the node from the root of the tree. As noted by Slate and Atkin ''The major design problem in selective search is the possibility that the lookahead process will exclude a key move at a low level [closer to the root] in the game tree.'' Good examples supporting this point are found elsewhere (Frey, 1983). Other methods, such as marginal forward pruning and the gamma algorithm, 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 (Birmingham and Kent, 1977), is applied near horizon nodes. The expectation in all forward pruning is that the side to move can always 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 (1977) point out ''the 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''. 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. On the other hand, use of the null move heuristic (Beal 1989; Goetsch and Campbell, 1990) may be valuable here.

Despite these disadvantages, there are sound forward pruning methods and there is every incentive to develop more, since this 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 (Horacek 1983).

3.3. Move 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'' (Gillogly, 1972). 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 (1972) puts it: ''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''. Referred to as the killer heuristic, a typical implementation maintains only the two most frequently occurring ''killers'' at each level (Slate and Atkin, 1977).

Later a more powerful and more general scheme for re-ordering moves at an interior node was introduced. For every legal move seen in the search tree, Schaeffer's history heuristic maintains a record of the move's success as a refutation, regardless of the line of play (Schaeffer, 1989a). At any point the best refutation 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 64×64 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. Thus at each new interior node, the available moves are re-ordered so that the ones that have been most successful elsewhere are tried first. An important

property of this so called history table is the ability to share 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.

3.4. Progressive and Iterative Deepening

The term progressive deepening was used by de Groot (1965) to encompass the notion of selectively extending the main continuation of interest. This type of selective expansion is not performed by programs employing the α-β 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. This difficulty was noted by Scott who reported in 1969 on the effective use of an iterated search (Scott, 1969). Jim Gillogly, author of the *Tech* chess program, 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 (1977) sought a better time control mechanism, and introduced an improved iterated search 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 memory table (such as a transposition or refutation table) which holds the best moves from subtrees traversed during the previous iteration. All the early experimental evidence suggests that the overhead cost of the preliminary 1 iterations is usually 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 (Marsland, 1983). Now the terms progressive and iterative deepening are often used synonymously.

One important aspect of these searches is the role played by re-sorting root node moves between iterations. Because there is only one root node, an extensive positional analysis of the moves can be done. Even ranking them according to consistency with continuing themes or a long range plan is possible. However, in chess programs which rate terminal positions primarily on material balance, many of the moves (subtrees) return with equal merits. Thus at least a stable sort should be used to preserve an initial order of preferences. Even so, that may not be enough. In the early iterations moves are not assessed accurately. Some initially good moves may return with a poor expected merit for one or two iterations. Later the merit may improve, but the move could remain at the bottom of a list of all moves of equal merit—not near the top as the initial ranking recommended. Should this move ultimately prove to be best, then far too many moves may precede it at the discovery iteration, and disposing of those moves may be inordinately expensive. Experience with our test program, *Parabelle* (Marsland and Popowich, 1985), has shown that among moves of apparently equal merit the partial ordering should be based on the order provided by an extensive pre-analysis at the root node, and not on the vagaries of a sorting algorithm.

3.5. 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 direct access table (Greenblatt, Eastlake and Crocker 1967; Slate and Atkin, 1977). Re-visits of positions that have been seen before are common, especially if a minimal window

search is used. When a position is reached again, the corresponding table entry serves three purposes. First, it may be possible to use the merit value in the table to narrow the (α,β) window bounds. Secondly, the best move that was found before can be tried immediately. It had probably caused a cut-off and may do so again, thus eliminating the need to generate the remaining moves. Here the table entry is being used as a move re-ordering mechanism. Finally, the primary purpose of the table is to enable recognition of move transpositions that have lead to a position (subtree) that has already been completely examined. In such a case there is no need to search again. This use of a transposition table is an example of exact forward pruning. Many programs also store their opening book in a way that is compatible with access to the transposition table. In this way they are protected against the myriad of small variations in move order that are common in the opening.

By far the most popular table-access method is the one proposed by Zobrist (1970). He observed that a chess position constitutes placement of up to 12 different piece types {K,Q,R,B,N,P,–K ... –P} onto a 64-square board. Thus a set of 12×64 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:

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

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

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

By using this index as a hash key to the transposition table, direct and rapid access is possible. For further speed and simplicity, and unlike a normal hash table, only a single probe is made. More elaborate schemes have been tried, and can be effective if the cost of the increased complexity of managing the table does not undermine the benefits from improved table usage. Table 1 shows the usual fields for each entry in the hash table. *Flag* specifies whether the entry corresponds to a position that has been fully searched, or whether *Merit* can only be used to adjust the α-β bounds. *Height* ensures that the value of a fully evaluated position is not used if the subtree length is less than the current search depth, instead *Move* is played.

Table 1: Typical Transposition Table Entry.

Lock	To ensure the table entry corresponds to the tree position.
Move	Preferred move in the position, determined from a previous search.
Merit	Value of subtree, computed previously.
Flag	Is the merit an upper bound, a lower bound or an exact value?
Height	Length of subtree upon which merit is based.

Correctly embedding transposition table code into the α-β algorithm needs care and attention to details. It can be especially awkward to install in the more efficient Principal Variation Search algorithm. To simplify matters, consider a revised version of Figure 5 in which the line

```
value := –PVS (moves[j], –α–1, –α, depth–1);
```

is replaced by

```
value := –MWS (moves[j], –α, depth–1);
```

```
FUNCTION MWS (p : position; β, depth : integer) : integer;
   VAR value, Height, Merit : integer;
       Move, TableMove, BestMove : 1..MAXWIDTH;
       Flag : (VALID, LBOUND, UBOUND);
       moves : ARRAY [1..MAXWIDTH] OF position;
BEGIN
   Retrieve(p, Height, Merit, Flag, TableMove);
            { if no entry in hash-transposition table then       }
            {    TableMove = 0, Merit = −∞ and Height < 0        }
   IF (Height ≥ depth) THEN BEGIN                {Node seen before}
      IF (Flag ≡ VALID) OR (Flag ≡ LBOUND AND Merit ≥ β)
                        OR (Flag ≡ UBOUND AND Merit < β) THEN
         Return(Merit);
   END;
   IF (Height > 0) THEN BEGIN             {Save a move Generation?}
      Merit := −MWS (moves[TableMove], −β+1, depth−1);
      if (Merit ≥ β) THEN
         Return(CUTOFF(p, Merit, depth, Height, TableMove));
   END;

   IF (depth ≡ 0) THEN
      Return(Evaluate(p));                              {Frontier node}

   moves := Generate(p);
   IF empty(moves) THEN
      Return(Evaluate(p));                                 {Leaf node}
   BestMove := TableMove;
   FOR Move := 1 TO sizeof(moves) DO
   IF Move ≠ TableMove THEN BEGIN
      IF (Merit ≥ β) THEN
         Return(CUTOFF(p, Merit, depth, Height, BestMove));
      value := −MWS (moves[Move], −β+1, depth−1);
      IF (value > Merit) THEN BEGIN
         Merit := value;
         BestMove := Move;
      END;
   END;

   IF (Height ≤ depth) THEN
      Store(p, depth, Merit, UBOUND, BestMove);
   Return(Merit);                                    {full-width node}
END;

FUNCTION CUTOFF (p: position; Merit, depth, Height : integer;
                 Move : 1..MAXWIDTH) : integer;
BEGIN
   IF (Height ≤ depth) THEN
      Store(p, depth, Merit, LBOUND, Move);
   return(Merit);                                      {pruned node}
END;
```

Figure 7: Minimal Window Search with Transposition Table.

Basically the minimal window search portion is being split into its own procedure (this formulation also has some advantages for parallel implementations). Figure 7 contains pseudo code for MWS and shows not only the usage of the entries *Move*, *Merit*, *Flag* and *Height* from Table 1, but does so in the negamax framework of the null window search portion of PVS. Of course the transposition access methods must also be put into PVS. It is here, for example, that Store sets

Flag to its EXACT value. Note too, in Figure 7, the introduction of the CUTOFF function to ensure that the LBOUND marker is stored in the transposition table when a cutoff occurs, while UBOUND is used when all the successors are examined. The contents of functions Retrieve and Store, which access and update the transposition table, are not shown here.

Transposition tables have found many applications in chess programs, not only to help detect replicated positions, but also to assess king safety and pawn formations (Nelson, 1985). Further these tables have been used to support a form of rote learning first explored by Arthur Samuel (1958) for checkers. Two major examples of improving performance in chess programs through learning are the works of Slate (1987) and Scherzer, Scherzer and Tjaden (1990).

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 beneficial to methods like PVS, since the initial minimal window search loads the table with useful lines that will be used if a re-search is needed. 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 (Nelson, 1985). 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. Thus a refutation table is the preferred choice of commercial systems or users of memory limited processors. A small triangular workspace ($D \times D/2$ 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. A good alternative description of refutation and transposition techniques appears in the recent book by Levy and Newborn (1990).

3.6. Combined Enhancements

The various terms and techniques described have evolved over the years, with the superiority of one method over another often depending on which elements are combined. Iterative deepening versions of aspiration and Principal Variation Search (PVS), along with transposition, refutation and history memory tables are all useful refinements to the α-β algorithm. Their relative performance is adequately characterized by Figure 8. That graph was made from data gathered by a chess program's simple evaluation function, when analyzing the standard Bratko-Kopec positions (Kopec and Bratko, 1982). Other programs may achieve slightly different results, reflecting differences in the evaluation function, but the relative performance of the methods should not be affected. Normally, the basis of such a comparison is the number of frontier nodes (also called horizon nodes, bottom positions or terminal nodes) visited. Evaluation of these nodes is usually more expensive than the predecessors, since a quiescence search is carried out there. However, these horizon nodes are of two types, ALL nodes, where every move is generated and evaluated, and CUT nodes from which only as many moves as necessary to cause a cut-off are assessed (Marsland and Popowich, 1985). For the minimal game tree these nodes can be counted, but there is no simple formula for the general α-β search case. Thus the basis of comparison for Figure 8 is the amount of CPU time required for each algorithm, rather than the leaf node count. Although a somewhat different graph is produced as a consequence, the relative performance of the methods does not change. The CPU comparison assesses the various enhancements more usefully, and also makes them look even better than on a node count basis. Analysis of the Bratko-Kopec positions requires the search of trees whose nodes have an average width (branching factor) of $W = 34$ branches. Thus it is possible to use the formula for horizon node count in a uniform minimal game tree to provide a lower bound on the search size, as drawn in Figure 8. Since search was not possible for this case, the trace represents the % performance relative to direct α-

β, but on a node count basis. Even so, the trace is a good estimate of the lower bound on the time required.

Figure 8 shows the effect of various performance enhancing mechanisms. At interior nodes, if the transposition (+trans) and/or refutation (+ref) table options are enabled, any valid table move is tried first. By this means, should a cut-off occur the need for a move generation is eliminated. Otherwise the initial ordering simply places safe captures ahead of other moves. When the history table (+hist) is enabled, the move list is further re-ordered to ensure that the most frequently effective moves from elsewhere in the tree are tried soonest. For the results presented in Figure 8, transposition, refutation and heuristic tables were in effect only for the traces whose label is extended with +trans, +ref and/or +hist respectively. Also, the transposition table was fixed at eight thousand entries, so the effects of table overloading may be seen when the search depth reaches 6-ply. Figure 8 shows that:

(a). Pure iterative deepening costs little over a direct search, and so can be effectively used as a time control mechanism. In the graph presented an average overhead of only 5% is shown, even though memory assists like transposition, refutation or history tables were not used.

(b). When iterative deepening is used, PVS is superior to aspiration search (with a ± pawn window).

(c). A refutation table is a space efficient alternative to a transposition table for guiding the early iterations.

(d). Odd-ply α-β searches are more efficient than even-ply ones.

(e). Transposition table size must increase with depth of search, or else too many entries will be overlaid before they can be used. The individual contributions of the transposition table, through move re-ordering, bounds narrowing and forward pruning are not brought out in this study.

(f). Transposition and/or refutation tables combine effectively with the history heuristic, achieving search results close to the minimal game tree for odd-ply search depths. It is these combinations that have the most dramatic effect on iterative deepening's efficiency.

3.7. Overview

A model chess program has three phases to its search. Typically, from the root node an exhaustive examination of layers of moves occurs, and this is followed by a phase of selective searches up to a limiting depth (the horizon). This limiting depth is not necessarily a constant, since responses to check are not usually counted in the path length. The reason is clear, there are only a few responses to checking moves and so their cost is negligible. Programs that have no selective search component might be termed ''brute force,'' while those lacking an initial exhaustive phase are often selective only in the sense that they use some form of marginal forward pruning. An evaluation function is applied at the frontier nodes to assess the material balance and the structural properties of the position (e.g., relative placement of pawns). To aid in this assessment a third phase is used, a variable depth quiescence search of those moves that are not dead (i.e., cannot be accurately assessed). It is the quality of this quiescence search which controls the severity of the horizon effect exhibited by all chess programs. Since the evaluation function is expensive, the best pruning must be used. All major programs use the ubiquitous α-β algorithm, in its aspiration or principal variation search form, along with iterative deepening.

Dynamic move re-ordering mechanisms like the killer heuristic, refutation tables, transposition tables and the history heuristic significantly improve these methods. Forward pruning methods are also sometimes effective. The transposition table is especially important because it improves the handling of endgames where the potential for a draw by repetition is high. Like the history heuristic, it is also a powerful predictor of cut-off moves, thus saving a move generation. The properties of these methods has been encapsulated in Figure 8, which shows their

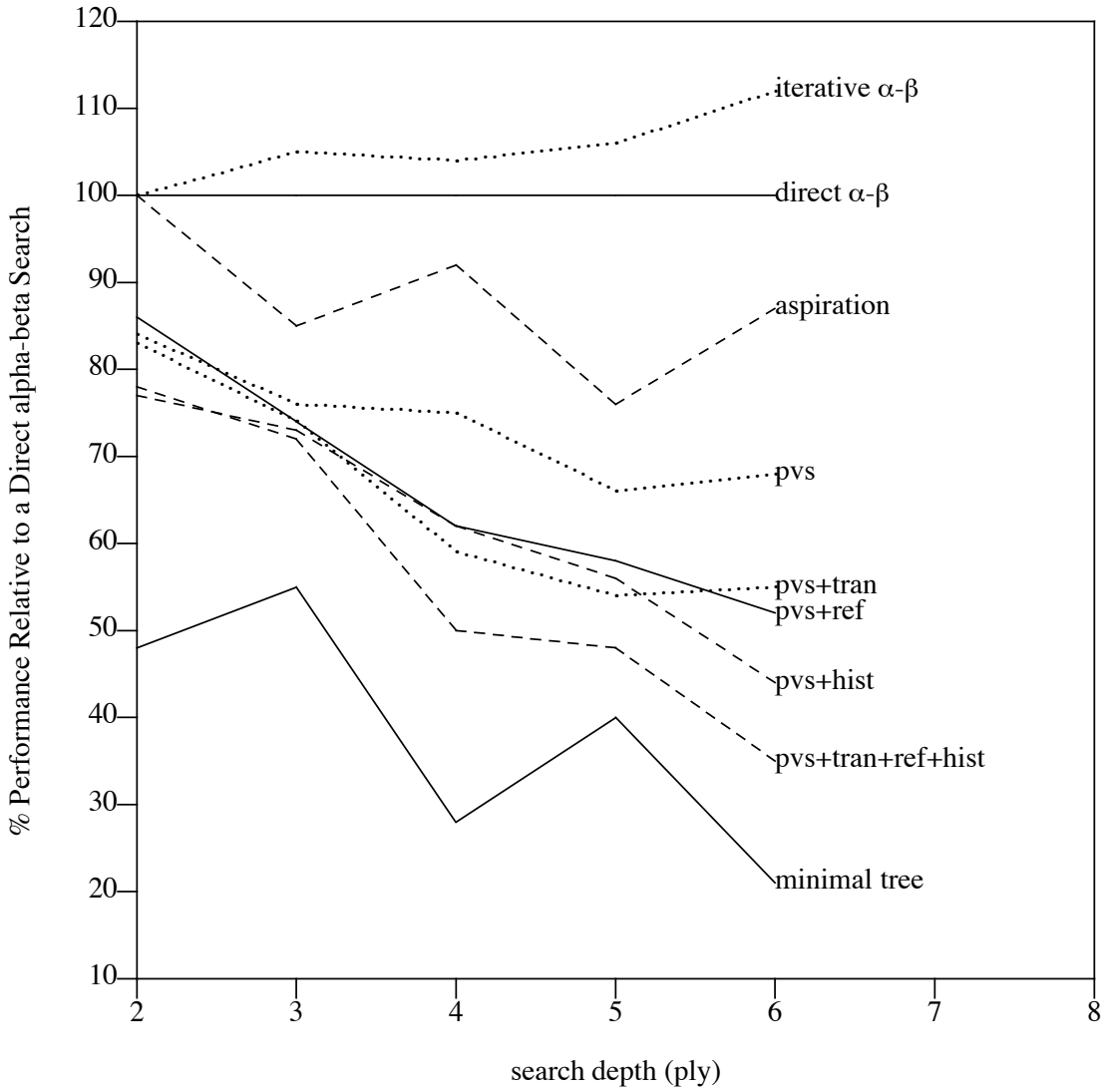


Figure 8: Time Comparison of Alpha-Beta Enhancements.

performance relative to a direct α-β search.

4. THE ANATOMY OF CHESS PROGRAMS

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 Frey and Atkin's (1979) instructive Pascal-based model. Even so, the most efficient way of representing all the tables and data structures necessary to describe a chess board is not yet known, although good proposals exist in readily available books (Frey, 1983; Welsh and Baczynskyj, 1985). Sometimes the Generate function produces all the feasible moves at once from these tables. This 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'' (all moves are considered) search 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 do an extensive analysis to select those moves that 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 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 (Slate and Atkin, 1977; Ebeling, 1987; Hyatt, Gower and Nelson, 1990).

4.1. Hardware Assists

Computer chess has consistently been in the forefront of the application of high technology. With *Cheops* (Moussouris, Holloway and Greenblatt, 1979), the 1970's saw the introduction of special purpose hardware for chess. Later networks of computers were tried; in 1983 (New York), *Ostrich* used an eight processor Data General system (Newborn, 1985) and *Cray Blitz* a dual processor Cray X-MP (Hyatt, Gower and Nelson, 1985). Some programs used special purpose hardware [see for example *Belle* (Condon and Thompson, 1982), *Bebe* (Scherzer, Scherzer and Tjaden, 1990), *Advance 3.0* and *BCP* (Welsh and Baczynskyj, 1985)], and there were several experimental commercial systems employing high-performance VLSI chips. The 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 circuit design for generating moves. Most recently *Deep Thought*, which has been likened to *Belle* on a chip, often runs as a multiprocessor system (Hsu *et al*., 1990; Hsu, 1990). The advantages and difficulties of parallel systems have been described by Hyatt, Suter and Nelson (1989) and Schaeffer (1989b). Although mainframes continue to be faster, the programs on smaller more special purpose computers are advancing more rapidly. It is only a matter of time before massive parallelism is applied to computer chess, as Stiller's endgame studies show (Stiller, 1989). 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.

4.2. 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, iterative deepening (especially when used with a refutation table), the killer heuristic and transposition tables 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'' (Gillogly, 1972), which involved selecting a

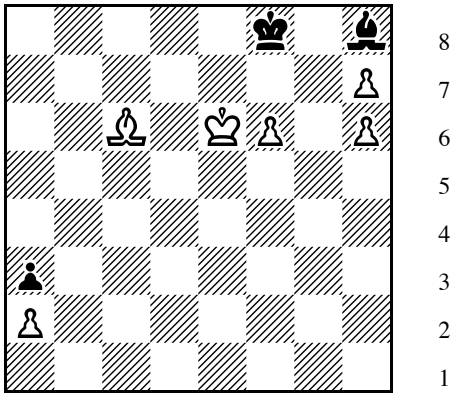
response for the opponent, usually the move predicted by the computer, and searching the predicted position for the next reply. Nothing is lost by this tactic, and when a successful prediction is made the time saved may be accumulated until it is necessary or possible to do a deeper search. Anticipating 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'' (Adelson-Velsky, Arlazarov and Donskoy, 1979). 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 (Horacek, 1983).

Perhaps the most important improvements in software have come from the new ideas in selective and quiescent search. Great emphasis has been placed on self-limiting variable depth search (Kaindl 1982) and on the method of singular extensions (Anantharaman, Campbell and Hsu 1988) to guide the quiescent down appropriately forcing lines. Also the widely mentioned idea of a null-move search, based on the assumption that nothing is worse than non move, has led Beal (1989) to formulate a theory of 1st-order and higher-order null-move searches. These ideas tie in with McAllister's (1988) proposed use of ''conspiracy numbers'', although they have not yet been applied in a major way to chess. Some of these methods are still somewhat special purpose, but they all point to the existence of a theory for controlled search for games.

4.3. 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 1979 game between *Duchess* and *Chaos*, which was analyzed extensively in an appendix to an important reference (Frey, 1983). After more than ten hours of play the position in Figure 9 was reached, and since neither side was making progress the game was adjudicated after white's 111^{th} 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 today I doubt if many programs can do 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 which avoids a draw must be sought. 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.



White to move

Figure 9: Lack of an Endgame Plan.

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 (Kopec, 1990). In speed chess even *Belle* often dominated the masters, and the newer machines like *Deep Thought* and *Hitech* are truly formidable. Increasingly, chess programs are teaching even experts new tricks and insights. Long ago Thomas Ströhlein built a database to find optimal solutions to several simple three and four piece endgames (kings plus one or two pieces). Using a Telefunken TR4 (48-bit word, 8 μsec. operations) he obtained the results summarized in Table 2 (Ströhlein 1970). When one considers that it took more than 29 hours to solve the KQ vs KR endgame, one realizes what a major undertaking it was in 1969. In the next decade, many others built databases of the simplest endings. Their approach (Bramer and Clark, 1979; Bratko and Michie, 1980) was to develop optimal sequences backward from all possible winning positions, building all paths to mate (i.e., reducing to a known subproblem). These works have recently been reviewed and put into perspective (van den Herik and Herschberg, 1985), but the other main results summarized in Table 2 were obtained by Thompson (1986).

Table 2: Maximum Moves to Mate or Win an Endgame.

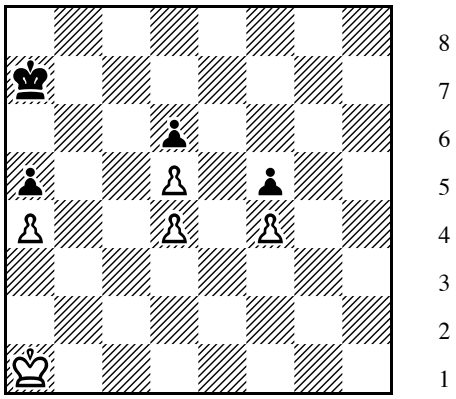
Ströhlein(1970)		Thompson(1986)			
Pieces	Moves to Mate	Pieces	Moves to Win	Pieces	Moves to Mate
KQ vs K	10	KBB vs KN	66	KRR vs KR	31
KR vs K	16	KQ vs KNN	63	KRQ vs KR	35
KR vs KB	18	KQ vs KNB	42	KQN vs KQ	41
KR vs KN	27	KQ vs KBB	71	KQB vs KQ	33
KQ vs KR	31	KRN vs KR	33	KQR vs KQ	67
		KRB vs KR	59	KQQ vs KQ	30

In the five-piece endgames not all the positions are necessarily won, but of those that are the maximum moves to mate or capture of a black piece is now known. These results were obtained in 1985/86 using a Sequent Balance 8000 computer, on which a ‘‘typical pure-piece endgame would be solved in two or three weeks of real time’’ (Thompson, 1986)! Although the recent work by Stiller (1989) is interesting, the major contribution by *Belle* and Ken Thompson was the building of databases to solve five-piece endgames—specifically, KQ# vs KQ (where # = Q, R, B

or N) and KR# vs KR. Furthermore Thompson discovered that in general KBB vs KN is won (not drawn) and less than 67 moves are needed to mate or safely capture the knight, raising questions about revisions to the 50-move rule. Also completed was a major study of the complex KQP vs KQ ending. Again, often more than 50 manoeuvres are required before a pawn can advance (Thompson, 1986). Although hard to top, in 1988 Lewis Stiller used a Connection Machine 2 to confirm these results and develop new ones, solving nearly all the five-piece endgames in the process (Stiller, 1989). He also exhaustively studied many of the four-piece plus pawn endgames, but had no means of storing the results to produce databases. For more complex endings involving several pawns, some 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 (Berliner and Campbell, 1984).

4.4. Memory Tables

Others have pointed out (Slate and Atkin, 1977; Nelson, 1985) 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. A 98-99% success rate was reported (Hyatt, Gower and Nelson, 1985) for a pawn hash table, since otherwise 10-20% of the search time was taken up with evaluation of pawn structures. King safety can also be handled similarly (Nelson, 1985), because the king has few moves and for long periods is not under attack.



White to move

Figure 10: Transposition Table Necessity.

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 position in Figure 10, which was apparently first solved by *Chess 4.9* (Frey and Atkin, 1979) and then by *Belle*. The only complete computer analysis of this position was provided later. As Hyatt, Gower and Nelson (1985) put 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 real 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.''

4.5. 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, thus abbreviating the width of search in a variable way 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'' (Condon and Thompson, 1983).

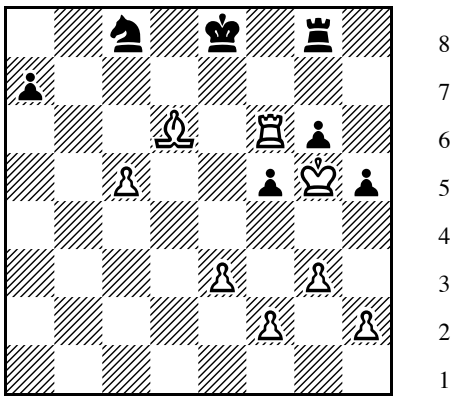
The reason for this belief is that chess game trees grow exponentially with depth of search. Ultimately it will become impossible to obtain the necessary computing power to search deeper within normal time constraints. For this reason most chess programs already incorporate some form of selective search, often as forward pruning. These methods are quite *ad hoc* since they have a weak theoretical base.

Although nearly all chess programs have some form of selective search deep in the tree, even if it is no more than discarding unlikely moves, so far only two major programs (*Awit* and *Chaos*) had noted successes while not considering all the moves at the root node. Despite their past occasional good results, these programs no longer compete in the race for Grand Master status. Nevertheless, although 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 handling tactics. In particular, Wilkin's *Paradise* program demonstrated superior performance in ''tactically sharp middle game positions'' on a standard suite of tests (Wilkins, 1983). *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. However, a nine to one time advantage 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'' (Wilkins, 1983). 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* was successful in this regard, part of its strength lay in its quiescence search, which was 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'' (Wilkins, 1983). 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 also wrote extensively on the subject of finding plans that win material (Pitrat, 1977), but neither his ideas nor those in *Paradise* were incorporated into the competitive chess programs of the 1980's.

4.6. 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 were prone to do. Complete details about the game are not important, but the position shown in Figure 11 was reached.



White's move 45.

Figure 11: 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, and yet humans recognize at a glance (de Groot, 1965). 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 that 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. In 1986, Jonathan Schaeffer completed an interesting doctoral thesis (Schaeffer, 1986)

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 (Schaeffer and Marsland, 1985).

5. 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 that 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. Perhaps the earliest multiprocessor chess program was *Ostrich* (Newborn, 1985). Other experimental systems followed including *Parabelle* (Marsland and Popowich, 1985) and *ParaPhoenix*. None of these early systems, nor the strongest multiprocessor program *Cray Blitz* (Hyatt, Gower and Nelson, 1990), consistently achieved more than a 5-fold speed-up, even when eight processors were used. There is no apparent theoretical limit to the parallelism. Although the practical restrictions are great, some new ideas on partitioning the work and better scheduling methods have begun to yield improved performance (Hyatt, Suter and Nelson, 1989; Schaeffer, 1989b).

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 that derives 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.

References

[AAD79] G.M. Adelson-Velsky, V.L. Arlazarov and M.V. Donskoy, ''Algorithms of Adaptive Search'' in J. Hayes, D. Michie and L. Michulich, eds., *Machine Intelligence 9*, Ellis Horwood, Chichester, 1979, 373-384.

[AAD88] G.M. Adelson-Velsky, V.L. Arlazarov and M.V. Donskoy, Algorithms for Games, Springer-Verlag, New York, 1988. Translation of Russian original (1978).

[ACH88] T. Anantharaman, M. Campbell and F. Hsu, ''Singular Extensions: Adding Selectivity to Brute-Force Searching,'' *Int. Computer Chess Assoc. J. 11*(4), 135-143 (1988). Also in *Artificial Intelligence* **43**(1), 99-110 (1990).

[Bea89] D. Beal, ''Experiments with the Null Move'' in D. Beal, ed., *Advances in Computer Chess 5*, Elsevier, 1989, 65-79. Revised as ''A Generalized Quiescence Search Algorithm'' in *Artificial Intelligence* **43**(1), 85-98 (1990).

[Bel78] A.G. Bell, The Machine Plays Chess?, Pergamon Press, Oxford, 1978.

[Ber73] 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.

[BeC84] H. Berliner and M. Campbell, ''Using Chunking to Solve Chess Pawn Endgames,'' *Artificial Intelligence 23*(1), 97-120 (1984).

[BeE89] H.J. Berliner and C. Ebeling, ''Pattern Knowledge and Search: The SUPREM Architecture,'' *Artificial Intelligence 38*(2), 161-198 (1989). A revised version appears as ''Hitech'' in *Computers, Chess, and Cognition*, 1990.

[BiK77] J.A. Birmingham and P. Kent, ''Tree-searching and Tree-pruning Techniques'' in M. Clarke, ed., *Advances in Computer Chess 1*, Edinburgh Univ. Press, Edinburgh, 1977, 89-107.

[BrC79] 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*, 635-649 (1979).

[BrM80] I. Bratko and D. Michie, ''A Representation for Pattern-knowledge in Chess Endgames'' in M. Clarke, ed., *Advances in Computer Chess 2*, Edinburgh Univ. Press, Edinburgh, 1980, 31-56.

[Bru63] A.L. Brudno, ''Bounds and Valuations for Abridging the Search of Estimates,'' *Problems of Cybernetics 10*, 225-241 (1963). Translation of Russian original in *Problemy Kibernetiki* **10**, 141-150 (May 1963).

[CaM83] M.S. Campbell and T.A. Marsland, ''A Comparison of Minimax Tree Search Algorithms,'' *Artificial Intelligence 20*(4), 347-367 (1983).

[CoT82] J.H. Condon and K. Thompson, ''Belle Chess Hardware'' in M. Clarke, ed., *Advances in Computer Chess 3*, Pergamon Press, Oxford, 1982, 45-54.

[CoT83] J.H. Condon and K. Thompson, ''Belle'' in P. Frey, ed., *Chess Skill in Man and Machine*, Springer-Verlag, 2nd Edition 1983, 201-210.

[Ebe87] C. Ebeling, All the Right Moves: A VLSI Architecture for Chess, MIT Press, 1987. See (1986) Ph.D. thesis, Carnegie-Mellon Univ., Pittsburgh, 145pp.

[Fis84] J.P. Fishburn, Analysis of Speedup in Distributed Algorithms, UMI Research Press, Ann Arbor, Michigan, 1984. See earlier PhD thesis (May 1981) Comp. Sci. Tech. Rep. 431, Univ. of Wisconsin, Madison, 118pp.

[FrA79] P.W. Frey and L.R. Atkin, ''Creating a Chess Player'' in B.L. Liffick, ed., *The BYTE Book of Pascal*, BYTE/McGraw-Hill, Peterborough NH, 2nd Edition 1979, 107-155. Also in D. Levy (ed.), *Computer Games 1*, Springer-Verlag, 1988, 226-324.

[Fre83] P.W. Frey(editor), Chess Skill in Man and Machine, Springer-Verlag, New York, 2nd Edition 1983.

[Gil72] J.J. Gillogly, ''The Technology Chess Program,'' *Artificial Intelligence 3*(1-4), 145-163 (1972). Also in D. Levy (ed.), *Computer Chess Compendium*, Springer-Verlag, 1988, 67-79.

[Gil78] J.J. Gillogly, Performance Analysis of the Technology Chess Program, Tech. Rept. 189, Computer Science, Carnegie-Mellon Univ., Pittsburgh, March 1978.

[GoC90] G. Goetsch and M.S. Campbell, ''Experiments with the Null-Move Heuristic'' in T.A. Marsland and J. Schaeffer, eds., *Computers, Chess, and Cognition*, Springer-Verlag, New York, 1990, 159-168.

[Goo68] I.J. Good, ''A Five-Year Plan for Automatic Chess'' in E. Dale and D. Michie, eds., *Machine Intelligence 2*, Elsevier, New York, 1968, 89-118.

[GEC67] R.D. Greenblatt, D.E. Eastlake and S.D. Crocker, ''The Greenblatt Chess Program,'' *Fall Joint Computing Conf. Procs. vol. 31*, 801-810 (San Francisco, 1967). Also in D. Levy (ed.), *Computer Chess Compendium*, Springer-Verlag, 1988, 56-66.

[Gro65] A.D. de Groot, Thought and Choice in Chess, Mouton, The Hague, 1965. Also 2nd Edition 1978.

[Har74] L.R. Harris, ''Heuristic Search under Conditions of Error,'' *Artificial Intelligence 5*(3), 217-234 (1974).

[HaL76] J.E. Hayes and D.N.L. Levy, The World Computer Chess Championship, Edinburgh Univ. Press, Edinburgh, 1976.

[HeH85] H.J. van den Herik and I.S. Herschberg, ''The Construction of an Omniscient Endgame Database,'' *Int. Computer Chess Assoc. J. 8*(2), 66-87 (1985).

[Hor83] H. Horacek, ''Knowledge-based Move Selection and Evaluation to Guide the Search in Chess Pawn Endings,'' *Int. Computer Chess Assoc. J. 6*(3), 20-37 (1983).

[Hsu90] F-h. Hsu, Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess, CMU-CS-90-108, Ph.D. thesis, Carnegie-Mellon Univ., Pittsburgh, Feb. 1990.

[HAC90] F-h. Hsu, T.S. Anantharaman, M.S. Campbell and A. Nowatzyk, ''Deep Thought'' in T.A. Marsland and J. Schaeffer, eds., *Computers, Chess, and Cognition*, Springer-Verlag, New York, 1990, 55-78.

[HGN85] R.M. Hyatt, A.E. Gower and H.L. Nelson, ''Cray Blitz'' in D. Beal, ed., *Advances in Computer Chess 4*, Pergamon Press, Oxford, 1985, 8-18.

[HSN89] R.M. Hyatt, B.W. Suter and H.L. Nelson, ''A Parallel Alpha/Beta Tree Searching Algorithm,'' *Parallel Computing 10*(3), 299-308 (1989).

[HGN90] R.M. Hyatt, A.E. Gower and H.L. Nelson, ''Cray Blitz'' in T.A. Marsland and J. Schaeffer, eds., *Computers, Chess, and Cognition*, Springer-Verlag, New York, 1990, 111-130.

[Kai82] H. Kaindl, ''Dynamic Control of the Quiescence Search in Computer Chess'' in R. Trappl, ed., *Cybernetics and Systems Research*, North-Holland, Amsterdam, 1982, 973-977.

[Kai90] H. Kaindl, ''Tree Searching Algorithms'' in T.A. Marsland and J. Schaeffer, eds., *Computers, Chess, and Cognition*, Springer-Verlag, New York, 1990, 133-158.

[KnM75] D.E. Knuth and R.W. Moore, ''An Analysis of Alpha-beta Pruning,'' *Artificial Intelligence 6*(4), 293-326 (1975).

[KoB82] D. Kopec and I. Bratko, ''The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess'' in M. Clarke, ed., *Advances in Computer Chess 3*, Pergamon Press, Oxford, 1982, 57-72.

[Kop90] D. Kopec, ''Advances in Man-Machine Play'' in T.A. Marsland and J. Schaeffer, eds., *Computers, Chess, and Cognition*, Springer-Verlag, New York, 1990, 9-32.

[LeN90] D.N.L. Levy and M.M. Newborn, How Computers Play Chess, W.H. Freeman & Co., New York, 1990.

[Lev88] D.N.L. Levy(editor), Computer Chess Compendium, Springer-Verlag, New York, 1988.

[MaC82] T.A. Marsland and M. Campbell, ''Parallel Search of Strongly Ordered Game Trees,'' *Computing Surveys 14*(4), 533-551 (1982).

[Mar83] T.A. Marsland, "Relative Efficiency of Alpha-beta Implementations," *Procs. 8th Int. Joint Conf. on Art. Intell.*, (Los Altos: Kaufmann), Karlsruhe, Germany, Aug. 1983, 763-766.

[MaP85] T.A. Marsland and F. Popowich, ''Parallel Game-Tree Search,'' *IEEE Trans. on Pattern Anal. and Mach. Intell. 7*(4), 442-452 (July 1985).

[MRS87] T.A. Marsland, A. Reinefeld and J. Schaeffer, ''Low Overhead Alternatives to SSS*,'' *Artificial Intelligence 31*(2), 185-199 (1987).

[Ma(90] T.A. Marsland and J. Schaeffer (eds.), Computers, Chess, and Cognition, Springer-Verlag, New York, 1990.

[Mic80] D. Michie, ''Chess with Computers,'' *Interdisciplinary Science Reviews 5*(3), 215-227 (1980).

[Mit77] B. Mittman, ''A Brief History of Computer Chess Tournaments: 1970-1975'' in P. Frey, ed., *Chess Skill in Man and Machine*, Springer-Verlag, 1977, 1-33.

[MHG79] J. Moussouris, J. Holloway and R. Greenblatt, ''CHEOPS: A Chess-oriented Processing System'' in J. Hayes, D. Michie and L. Michulich, eds., *Machine Intelligence 9*, Ellis Horwood, Chichester, 1979, 351-360.

[Nel85] H.L. Nelson, ''Hash Tables in Cray Blitz,'' *Int. Computer Chess Assoc. J. 8*(1), 3-13 (1985).

[Nem51] T. Nemes, "The Chess-Playing Machine," *Acta Technica*, Hungarian Academy of Sciences, Budapest, 1951, 215-239.

[New85] M.M. Newborn, "A Parallel Search Chess Program," *Procs. ACM Ann. Conf.*, (New York: ACM), Denver, Oct 1985, 272-277. See also (March 1982) Tech. Rep. SOCS 82.3, Computer Science, McGill Univ., Montreal, Canada, 20pp.

[New88] M.M. Newborn, ''Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search,'' *IEEE Trans. on Pattern Anal. and Mach. Intell. 10*(5), 687-694 (1988).

[NSS58] A. Newell, J.C. Shaw and H.A. Simon, ''Chess Playing Programs and the Problem of Complexity,'' *IBM J. of Research and Development 4*(2), 320-335 (1958). Also in E. Feigenbaum and J. Feldman (eds.), *Computers and Thought*, 1963, 39-70.

[Pea80] J. Pearl, ''Asymptotic Properties of Minimax Trees and Game Searching Procedures,'' *Artificial Intelligence 14*(2), 113-138 (1980).

[Pit77] J. Pitrat, ''A Chess Combination Program which uses Plans,'' *Artificial Intelligence 8*(3), 275-321 (1977).

[Sam59] A.L. Samuel, ''Some Studies in Machine Learning Using the Game of Checkers,'' *IBM J. of Res. & Dev. 3*, 210-229 (1959). Also in D. Levy (ed.), *Computer Games 1*, Springer-Verlag, 1988, 335-365.

[ScM85] J. Schaeffer and T.A. Marsland, "The Utility of Expert Knowledge," *Procs. 9th Int. Joint Conf. on Art. Intell.*, Los Angeles, 1985, 585-587.

[Sch86] J. Schaeffer, Experiments in Search and Knowledge, Ph.D. thesis, Univ. of Waterloo, Waterloo, Canada, Spring 1986.

[Sch89a] J. Schaeffer, ''Distributed Game-Tree Search,'' *J. of Parallel and Distributed Computing 6*(2), 90-114 (1989).

[Sch89b] J. Schaeffer, ''The History Heuristic and Alpha-Beta Search Enhancements in Practice,'' *IEEE Trans. on Pattern Anal. and Mach. Intell. 11*(11), 1203-1212 (1989).

[Sch90] J. Schaeffer, ''1989 World Computer Chess Championship'' in T.A. Marsland and J. Schaeffer, eds., *Computers, Chess, and Cognition*, Springer-Verlag, New York, 1990, 33-46.

[SST90] T. Scherzer, L. Scherzer and D. Tjaden, ''Learning in Bebe'' in T.A. Marsland and J. Schaeffer, eds., *Computers, Chess, and Cognition*, Springer-Verlag, New York, 1990, 197-216.

[Sco69] J.J. Scott, ''A Chess-Playing Program'' in B. Meltzer and D. Michie, eds., *Machine Intelligence 4*, Edinburgh Univ. Press, 1969, 255-265.

[Sha50] C.E. Shannon, ''Programming a Computer for Playing Chess,'' *Philosophical Magazine 41*(7), 256-275 (1950). Also in D. Levy (ed.), *Computer Chess Compendium*, Springer Verlag, 1988, 2-13.

[SlA77] D.J. Slate and L.R. Atkin, ''CHESS 4.5 - The Northwestern Univ. Chess Program'' in P. Frey, ed., *Chess Skill in Man and Machine*, Springer-Verlag, 1977, 82-118.

[Sla87] D. Slate, ''A Chess Program that uses its Transposition Table to Learn from Experience,'' *Int. Computer Chess Assoc. J. 10*(2), 59-71 (1987).

[Sti89] L. Stiller, ''Parallel Analysis of Certain Endgames,'' *Int. Computer Chess Assoc. J. 12*(2), 55-64 (1989).

[Str70] T. Ströhlein, Untersuchungen über Kombinatorische Speile, Doctoral Thesis, Technischen Hochschule München, Munich, Germany, Jan. 1970.

[Tho82] K. Thompson, ''Computer Chess Strength'' in M. Clarke, ed., *Advances in Computer Chess 3*, Pergamon Press, Oxford, 1982, 55-56.

[Tho86] K. Thompson, ''Retrograde Analysis of Certain Endgames,'' *Int. Computer Chess Assoc. J. 9*(3), 131-139 (1986).

[TSB53] A.M. Turing, C. Strachey, M.A. Bates and B.V. Bowden, ''Digital Computers Applied to Games'' in B.V. Bowden, ed., *Faster Than Thought*, Pitman, 1953, 286-310.

[WeB85] D.E. Welsh and B. Baczynskyj, Computer Chess II, W.C. Brown Co., Dubuque, Iowa, 1985.

[Wil83] David Wilkins, ''Using Chess Knowledge to Reduce Speed'' in P. Frey, ed., *Chess Skill in Man and Machine*, Springer-Verlag, 2nd Edition 1983, 211-242.

[Zob70] A.L. Zobrist, A New Hashing Method with Applications for Game Playing, Tech. Rep. 88, Computer Sciences Dept., Univ. of Wisconsin, Madison, April, 1970. Also in *Int. Computer Chess Assoc. J.* **13**(2), 169-173 (1990).

[Zus76] K. Zuse, ''Chess Programs'' in *The Plankalkül*, Rept. No. 106, Gesellschaft für Mathematik und Datenverarbeitung, Bonn, Germany, 1976, 201-244. Translation of German original, 1945. Also as Rept. No. 175, Oldenbourg Verlag, Munich, 1989.