A REVIEW OF GAME-TREE PRUNING†

T.A. Marsland

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

ABSTRACT

Chess programs have three major components: move generation, search, and evaluation. All components are important, although evaluation with its quiescence analysis is the part which makes each program's play unique. The speed of a chess program is a function of its move generation cost, the complexity of the position under study and the brevity of its evaluation. More important, however, is the quality of the mechanisms used to discontinue (prune) search of unprofitable continuations. The most reliable pruning method in popular use is the robust alpha-beta algorithm, and its many supporting aids. These essential parts of game-tree searching and pruning are reviewed here, and the performance of refinements, such as aspiration and principal variation search, and aids like transposition and history tables are compared.

† Much of this article is a revision of material condensed from an entry entitled ''Computer Chess Methods,'' prepared for the *Encyclopedia of Artificial Intelligence*, S. Shapiro (editor), to be published by John Wiley & Sons in 1987. The transposition table pseudo code of Figure 7 is similar to that in another paper: ''Parallel Search of Strongly Ordered Game Trees,'' T.A. Marsland and M. Campbell, *ACM Computing Surveys*, Vol 14, No. 4, copyright 1982, Association for Computing Machinery Inc., and is reprinted by permission.

Final draft: ICCA Journal, Vol. 9, No. 1, March 1986, pp. 3-19.

A REVIEW OF GAME-TREE PRUNING†

T.A. Marsland

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

1. INTRODUCTION

A typical chess program contains three distinct elements: board description and move generation, tree searching/pruning, and position evaluation. Several good descriptions of the necessary tables and data structures to represent a chess board exist in readily available books [1, 2] and articles [3, 4]. Even so, there is no general agreement on the best or most efficient representation. From these tables the move list for each position is generated. Sometimes the *Generate* function produces all the feasible moves at once, with the advantage that they may be sorted and tried in the most probable order of success. In small memory computers, on the other hand, the moves are produced one at a time. This saves space and may be quicker if an early move refutes the current line of play. Since only limited sorting is possible (captures might be generated first) the searching efficiency is generally lower, however. Rather than re-address these issues, first-time builders of a chess program are well advised to follow Larry Atkin's excellent Pascal-based model [5].

Perhaps the most important part of a chess program is the *Evaluate* function invoked at the maximum depth of search to assess the merits of the moves, many of which are capturing or forcing moves that are not "dead." Typically a limited search (called a quiescence search) must be carried out to determine the unknown potential of such active moves. The evaluation process estimates the value of chess positions that cannot be fully explored. In the simplest case *Evaluate* only counts the material difference, but 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 [2, 6].

In the area of searching and pruning, all chess programs fit the following general pattern. A full width "exhaustive" search (all moves are considered) is done at the first few layers of the game tree. At depths beyond this exhaustive region some form of selective search is used. Typically, unlikely or unpromising moves are simply dropped from the move list. More sophisticated programs select those discards based on an extensive analysis. Unfortunately, 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, at some maximum depth of search, the evaluation function is invoked; that in turn usually entails a further search of designated moves like captures. Thus all programs employ a model with an implied tapering of the search width, as variations are explored more and more deeply. What differentiates one program from another is the quality of the evaluation, and the severity with which the tapering operation occurs. This paper concentrates on the tree searching and pruning aspects, especially those which are well formulated and have provable characteristics.

2. COMPONENTS OF SEARCH

Since most chess programs examine large trees, a depth-first search is commonly used. That is, the first branch to an immediate successor of the current node is recursively expanded until a leaf node (a node without successors) is reached. The remaining branches are then considered as the search process backs up to the root. Other expansion schemes are possible and the domain is fruitful for testing new search algorithms. Since computer chess is well defined, and absolute measures of performance exist, it is a useful test vehicle for measuring algorithm efficiency. In the simplest case, the best algorithm is the one that visits fewest nodes when determining the true value of a tree. For a two-person game-tree, this value, which is a least upper bound on the score (or merit) for the side to move, can be found through a minimax search. In chess, this so called minimax value is a combination of both ‘‘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, *Evaluate* 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 play or half move) the term ‘‘ply’’ was introduced by Arthur Samuel in his major paper on machine learning [7]. How that word was chosen is not clear, perhaps as a contraction of ‘‘play’’ or maybe by association with forests as in layers of plywood. In either case it was certainly appropriate and it has been universally accepted.

A true minimax search of a game tree may 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 [8], produce narrow trees (fewer than 10 branches per node) that can often be solved exhaustively. In contrast, chess produces bushy trees (average branching factor, W, of about 35 moves [9]). 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 it can be shown that 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 [10], but according to Knuth and Moore the ideas go back further, to John McCarthy and his group at MIT. The first thorough treatment of the topic appears to be Brudno's 1963 paper [11]. 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. Some 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 introduced a recursive algorithm called F2 [12], and used it to prove properties about the α-β algorithm. A ‘‘negamax’’ framework was also employed whose primary advantage is that by always passing back the negative of the subtree value, only maximizing operations are needed. In Figure 1, Pascal-like pseudo code is used to present our α-β function, AB, in the same negamax framework. A *Return* statement has been introduced as the convention for exiting the function and returning the best subtree value or score. Omitted are details of the game-specific functions *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,score) operation represents Fishburn's ‘‘fail-soft’’ condition [13], and ensures that the best available value is returned (rather than an α/β bound), even if it 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 score, j, value : integer;
      posn : ARRAY [1..MAXWIDTH] OF position;
                              { Note: depth must be positive }
BEGIN
  IF depth = 0 THEN               { horizon node, maximum depth? }
     Return(Evaluate(p));

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

Figure 1: Depth-limited Alpha-Beta Function.

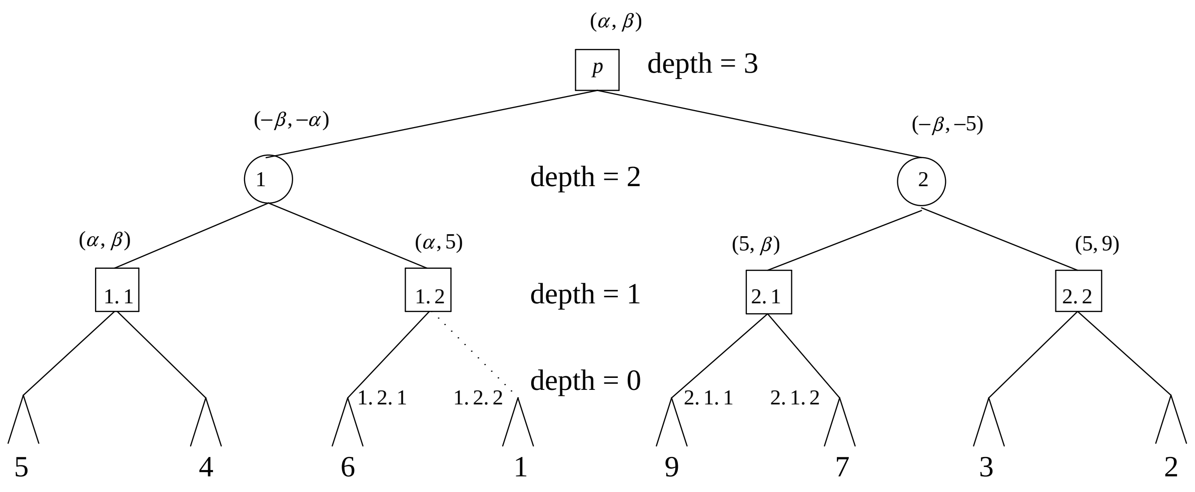

Figure 2: The Effects of $\alpha - \beta$ Pruning.

Although tree-searching topics involving pruning appear routinely in standard Artificial Intelligence texts, chess 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 scheme is used to label the nodes, so that the node name carries information about the path back to the root node. Thus p.2.1.2 is the root of a hidden subtree whose value is shown as 7 in Figure 2. Also shown at each node of Figure 2 is the initial alpha-beta window that is employed by the search. Note that successors to node p.1.2 are searched with an initial window of $(\alpha,5)$. 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 [12], 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 also called a horizon node, with D the distance from the root node to the horizon [14].

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. The method was analyzed by Brudno [11], referred to by Berliner [15], and experimented with by Gillogly [16], but was not consistently successful. A disadvantage is that sometimes the initial bounds do not enclose the minimax value, in which case the search must be repeated with corrected bounds, as the outline of Figure 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.

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

    V := AB (p, $\alpha$, $\beta$, depth);
    IF (V $\geq \beta$) THEN              {   failing high }
      V := AB (p, V, $+\infty$, depth)
    ELSE
    IF (V $\leq \alpha$) THEN             {   failing low  }
      V := AB (p, $-\infty$, V, depth);

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

Figure 3: Narrow Window Aspiration Search.

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 [value] 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’’ [6]. 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 [17].

2.5. Quiescence Search

Even the earliest papers on computer chess recognized the importance of evaluating only those positions which are ‘‘relatively quiescent’’ [18] or ‘‘dead’’ [19]. These are positions which can be assessed accurately without further search. Typically they have no moves, such as checks, promotions or complex captures, whose outcome is unpredictable. Not all the moves at horizon nodes are quiescent (i.e., lead immediately to dead positions), so some must be searched further. To limit the size of this so called quiescence search, only dynamic moves are selected for consideration. These might be as few as the moves that are part of a single complex capture, but can expand to include all capturing moves and all responses to check [20]. Ideally, passed pawn moves (especially those close to promotion) and selected checks should be included [21, 22], but these are often only examined in computationally simple endgames. The goal is always to clarify the node so that a more accurate position evaluation is made. Despite the obvious benefits of these ideas, the realm of quiescence search is unclear, because no theory for selecting and limiting the participation of moves exists. Present quiescent search methods are attractive; 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 [23]. On the other hand, some programs manage quite well without quiescence search, using direct computation to evaluate the exchange of material [24].

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 [14]. 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 [23], 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 [23], 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, ''thinking'' that the queen has been saved at the expense of four pawns! Thus programs with a poor or inadequate quiescence search suffer more from the horizon effect. The best way to provide automatic extension of non-quiescent positions is still an open question, despite proposals such as bandwidth heuristic search [25].

```
   ::   ::    Rb    Kb
::   ::    Qw Pb Qb
   ::   :: Pb ::   ::
:: Pb :: Pb ::    Pw
   :: Pb ::    Pw   ::
:: Pb ::    Pw   ::
   ::    Pw   ::   ::
Bw Kw ::   ::   ::
```

Black's Move

Figure 4: The Horizon Effect.

3. ALPHA-BETA ENHANCEMENTS

3.1. Minimal Window Search

Theoretical advances, such as Scout [26] and the comparable minimal window search techniques [13, 17, 27] 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 [17], 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 [28]. 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 [29], 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 horizon 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; $\alpha$, $\beta$, depth : integer) : integer;
                         { p is pointer to the current node     }
                         { $\alpha$ and $\beta$ are window bounds            }
                         { depth is the remaining search length }
                         { the value of the subtree is returned }
  VAR score, j, value : integer;
     posn : ARRAY [1..MAXWIDTH] OF position;
                               { Note: depth must be positive }
BEGIN
  IF depth = 0 THEN                { horizon node, maximum depth? }
    Return(Evaluate(p));

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

Figure 5: Minimal Window Principal Variation Search.

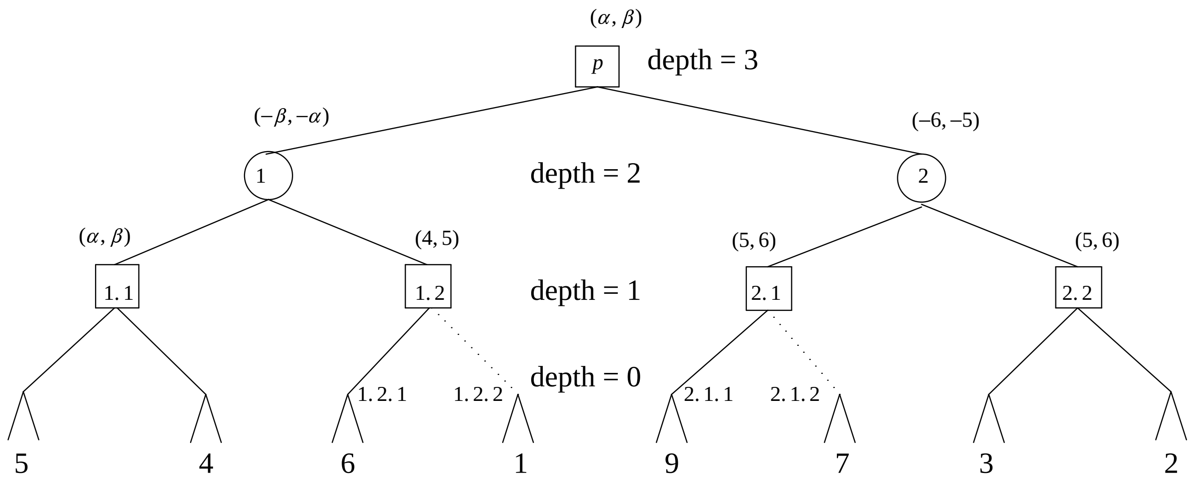

Figure 6: The Effects of PVS Pruning.

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 [30, 31] 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 in the game tree.’’ Good examples supporting this point are found elsewhere [32]. Other methods, such as marginal forward pruning [33] and the gamma algorithm [34], 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 [35], is applied near horizon nodes. The expectation in all forward pruning is that the side to move can improve the current value, so it may be futile to continue. Unfortunately there are cases when the assumption is untrue, for instance in zugzwang positions. As Birmingham and Kent point out ‘‘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’’ [35]. Marginal pruning may also break down when the side to move has more than one piece *en prise* (e.g., is forked), and so the decision to stop the search must be applied cautiously.

Despite these disadvantages, there are sound forward pruning methods and there is every incentive to develop more, since 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.

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’’ [20]. 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 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’’ [20]. Referred to as the killer heuristic, a typical implementation maintains only the two most frequently occurring ‘‘killers’’ at each level [6].

Recently a more powerful and more general scheme for re-ordering moves at an interior node has been introduced. Schaeffer’s history heuristic ‘‘maintains a history for every legal move seen in the search tree. For each move, a record of the move’s ability to cause a refutation is kept, regardless of the line of play’’ [36]. At an interior node the best move is the one that either yields the highest score or causes a cut-off. Many implementations are possible, but a pair of tables (each of 64x64 entries) is enough to keep a frequency count of how often a particular move (defined as a from-to square combination) is best for each side. The available moves are re-ordered so that the most successful ones are tried first. An important property of this so called history table is the sharing of information about the effectiveness of moves throughout the tree, rather than only at nodes at the same search level. The idea is that if a move is frequently good enough to cause a cut-off, it will probably be effective whenever it can be played.

3.4. Progressive and Iterative Deepening

The term progressive deepening was used by de Groot [9] 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. Jim Gillogly, author of the *Tech* chess program [20], coined the term iterative deepening to distinguish a full-width search to increasing depths from the progressively more focused search described by de Groot. About the same time David Slate and Larry Atkin sought a better time control mechanism, and introduced the notion of an iterated search [6] 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 indicated that the overhead cost of the preliminary D-1 iterations was often recovered through a reduced cost for the D-ply search. Later the efficiency of iterative deepening was quantified to assess various refinements, especially memory table assists [17]. Today 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) will return with equal scores. 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 score for one or two iterations. Later the score may improve, but the move could remain at the bottom of a list of all moves of equal score -- 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 data has shown that among moves of equal score the partial ordering should be based on 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 (score, best move, status) of the searches of nodes (subtrees) in the tree can be held in a large direct access table [6, 28, 31]. 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 table score 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 [37]. He observed that a chess position constitutes placement of up to 12 different piece types {K,Q,R,B,N,P,-K ... -P} on to a 64-square board. Thus a set of 12x64 unique integers (plus a few more for *en passant* and castling privileges), $\{R_i\}$, may be used to represent all the possible piece/square combinations. For best results these integers should be at least 32 bits long, and be randomly independent of each other. An index of the position may be produced by doing an exclusive-or on selected integers as follows:

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

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

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

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, but often the cost of the increased complexity of managing the table undermines 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 *Score* 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, rather *Move* is played instead. Figure 7 contains pseudo code showing usage of the entries *Move*, *Score*, *Flag* and *Height*. Not shown there are functions *Retrieve* and *Store*, which access and update the transposition table.

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

Table 1: Typical Transposition Table Entry.

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

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

    IF (height ≥ depth) THEN BEGIN
        IF (flag = VALID) THEN
           Return(score);     { Forward prune, fully seen before }
        IF (flag = LBOUND) THEN
           $\alpha$ := max($\alpha$, score);            { Narrow the window }
        IF (flag = UBOUND) THEN
           $\beta$ := min($\beta$, score);            { Narrow the window }
        IF ($\alpha \geq \beta$) THEN
           Return(score);   { Forward prune, no further interest }
    END;
                  {  Note: update of the $\alpha$ or $\beta$ bound        }
                  {  is not valid in a selective search.     }
                  {  If score in table insufficient to end    }
                  {  search, try best move from table first   }
                  {  before generating other moves.           }

    IF (depth = 0) THEN                     {  horizon node?  }
        Return(Evaluate(p));
    IF (height ≥ 0) THEN BEGIN
                            { Re-order, try 'move' from table }
        score := -AB (posn[move], -$\beta$, -$\alpha$, depth-1);
        IF (score ≥ $\beta$) THEN
           GOTO done;            { Success, omit move generation }
    END ELSE score := -∞;
                             { No cut-off, produce move list }
    posn := Generate(p);
    IF empty(posn) THEN             {  leaf, mate or stalemate?  }
       Return(Evaluate(p));

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

Figure 7: Alpha-Beta Search with Transposition Table.

A transposition table also identifies the preferred move sequences used to guide the next iteration of a progressive deepening search. Only the move is important in this phase, since the subtree length is usually less than the remaining search depth. Transposition tables are particularly advantageous to methods like PVS, since the initial minimal window search loads the table with useful lines that are used in the event of a re-search. On the other hand, for deeper searches, entries are commonly lost as the table is overwritten, even though the table may contain more than a million entries [38]. 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 (DxD/2 entries) is needed to hold the current continuation as it is generated, and these entries in the workspace can also be used as a source of killer moves [39].

3.6. Interpretation

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 analyzing the standard Bratko-Kopec positions [40] with a simple evaluation function. 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 horizon nodes (also called 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 [41]. 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.

One feature of our simple chess program is that an extensive static analysis is done at the root node. The order this analysis provides to the initial moves is retained from iteration to iteration among moves which return the same ‘‘value.’’ At the other interior nodes, if the transposition and/or refutation table options are in effect and either provides a valid move, that move is tried first. Should a cutoff occur the need for a move generation is eliminated. Otherwise the provisional ordering simply places safe captures ahead of other moves. If the history table is enabled, then the move list is reordered 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). 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.

(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.

Odd-ply α-β searches are

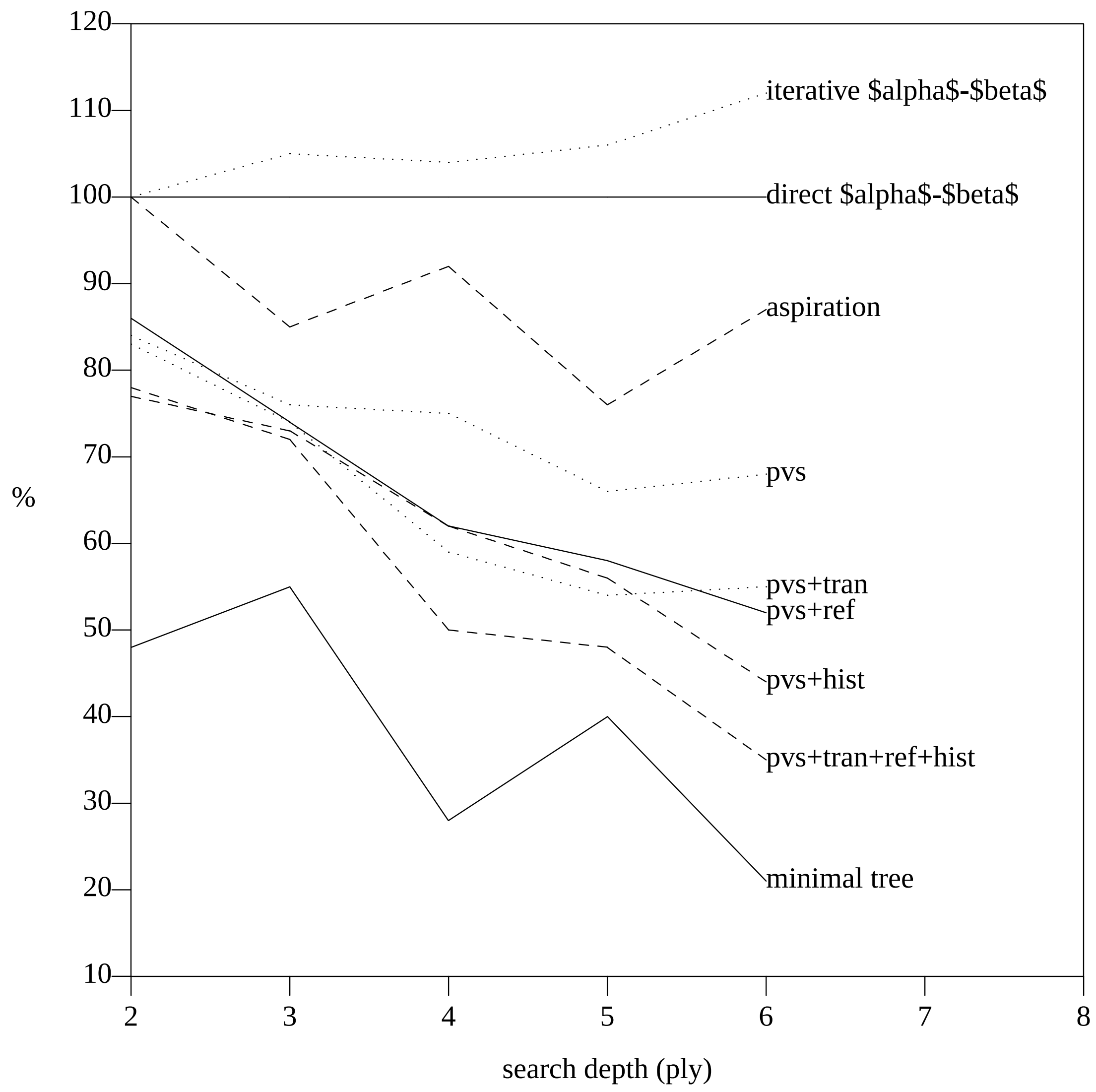

Figure 8: Time Comparison of Alpha-Beta Enhancements

% Performance Relative to a Direct $alpha$-$beta$ Search

4. 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). Programs which 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 employ some form of marginal forward pruning. An evaluation function is applied at the horizon 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 which 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 and one of its refinements like aspiration search or principal variation search, along with some form of iterative deepening.

These methods are significantly improved by dynamic move re-ordering mechanisms like the killer heuristic, refutation tables, transposition tables and the history heuristic. 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 merits of these methods has been encapsulated in a single figure showing their performance relative to a direct α-β search.

Acknowledgements

Don Beal, Peter Frey, Jaap van den Herik, Hermann Kaindl and Jonathan Schaeffer provided helpful definitions of technical terms, and offered constructive criticism that improved the basic organization of the work. To all these people, and also to Tim Breitkreutz who assisted with the experiments, I offer my sincere thanks. Financial support for the preparation of the results was possible through a Canadian Natural Sciences and Engineering Research Council Grant.

References

1. P.W. Frey (editor), *Chess Skill in Man and Machine*, Springer-Verlag, New York, 2nd Edition 1983.
2. D.E. Welsh and B. Baczynskyj, *Computer Chess II*, W.C. Brown Co., Dubuque, Iowa, 1985.
3. A.G. Bell, Algorithm 50: How to Program a Computer to Play Legal Chess, *Computer Journal 13*, 2 (1970), 208-219.
4. S.M. Cracraft, Bitmap Move Generation in Chess, *Int. Computer Chess Assoc. J. 7*, 3 (1984), 146-152.
5. P.W. Frey and L.R. Atkin, Creating a Chess Player, in *The BYTE Book of Pascal*, B.L. Liffick (ed.), BYTE/McGraw-Hill, Peterborough NH, 2nd Edition 1979, 107-155. Also in D. Levy (ed.), *Computer Games 1*, Springer-Verlag, 1988, 226-324.
6. D.J. Slate and L.R. Atkin, CHESS 4.5 - The Northwestern University Chess Program, in *Chess Skill in Man and Machine*, P. Frey (ed.), Springer-Verlag, 1977, 82-118.
7. A.L. Samuel, Some Studies in Machine Learning Using the Game of Checkers, *IBM J. of Res. & Dev. 3*, (1959), 210-229. Also in D. Levy (ed.), *Computer Games 1*, Springer-Verlag, 1988, 335-365.
8. A.G. Bell, *Games Playing with Computers*, Allen and Unwin, London, 1972.
9. A.D. de Groot, *Thought and Choice in Chess*, Mouton, The Hague, 1965. Also 2nd Edition 1978.
10. 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 (1958), 320-335. Also in E. Feigenbaum and J. Feldman (eds.), *Computers and Thought*, 1963, 39-70.
11. A.L. Brudno, Bounds and Valuations for Abridging the Search of Estimates, *Problems of Cybernetics 10*, (1963), 225-241. Translation of Russian original in *Problemy Kibernetiki* **10**, 141-150 (May 1963).
12. D.E. Knuth and R.W. Moore, An Analysis of Alpha-beta Pruning, *Artificial Intelligence 6*, 4 (1975), 293-326.
13. 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, University of Wisconsin, Madison, 118pp.
14. 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.
15. H.J. Berliner, Chess as Problem Solving: The Development of a Tactics Analyzer, Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, March 1974.
16. J.J. Gillogly, Performance Analysis of the Technology Chess Program, Tech. Rept. 189, Computer Science, Carnegie-Mellon University, Pittsburgh, March 1978.
17. 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.
18. C.E. Shannon, Programming a Computer for Playing Chess, *Philosophical Magazine 41*, 7 (1950), 256-275. Also in D. Levy (ed.), *Computer Chess Compendium*, Springer Verlag, 1988, 2-13.
19. A.M. Turing, C. Strachey, M.A. Bates and B.V. Bowden, Digital Computers Applied to Games, in *Faster Than Thought*, B.V. Bowden (ed.), Pitman, 1953, 286-310.
20. J.J. Gillogly, The Technology Chess Program, *Artificial Intelligence 3*, 1-4 (1972), 145-163. Also in D. Levy (ed.), *Computer Chess Compendium*, Springer-Verlag, 1988, 67-79.
21. K. Thompson, Computer Chess Strength, in *Advances in Computer Chess 3*, M. Clarke (ed.), Pergamon Press, Oxford, 1982, 55-56.

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

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

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

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

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

27. M.S. Campbell and T.A. Marsland, A Comparison of Minimax Tree Search Algorithms, *Artificial Intelligence 20*, 4 (1983), 347-367.

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

29. A. Reinefeld, An Improvement of the Scout Tree-Search Algorithm, *Int. Computer Chess Assoc. J. 6*, 4 (1983), 4-14.

30. A. Kotok, A Chess Playing Program for the IBM 7090, B.S. Thesis, MIT, AI Project Memo 41, Computation Center, Cambridge MA, 1962.

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

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

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

34. M.M. Newborn, *Computer Chess*, Academic Press, New York, 1975.

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

36. J. Schaeffer, The History Heuristic, *Int. Computer Chess Assoc. J. 6*, 3 (1983), 16-19.

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

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

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

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

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