ALPHA-BBTA PRUNING

ON EVOLVING GAME TREES*

by

Gary Lindstrom

UUCS 79-101

Department of Computer Science

University of Utah

Salt Lake City, Utah 84112

March 1979

*This work was supported in part by the National Science Foundation under grant MCS 78-03832.

ABSTRACT

The alpha-beta strategy is a widely used method for economizing on the size of game trees. Heretofore, its application has been limited to depth-first tree growth in recursive search functions. However, many modern game players use retentive (i.e. coroutine-based) control to achieve greater attention mobility in the game tree, e.g. for heuristically guided "best-first" searching. This paper reformulates the alpha-beta strategy for this generalized control setting. Algorithms are provided (in complete PASCAL code) for the following operations on appropriate nodes arbitrarily selected from a game tree: terminal node expansion, resumption of heuristically suspended move generation, tree re-rooting (i.e. top-level move selection), subtree redevelopment to satisfy a new search thoroughness condition, including restart of nodes that were cut-off but may no longer be. Empirical results are presented indicating that, in addition to heuristic freedom, this method typically offers trees with fewer terminal nodes than in the recursive case, due to best-first descendant ordering, and the availability on the average of greater tree context for node cutting.

CR categories: 3.66; 4.22.

Key words and phrases: alpha-beta pruning, game trees, heuristic search, minimaxing, retentive control, PASCAL.

1. MOTIVATION.

1.1. The alpha-beta strategy.

The alpha-beta strategy is a familiar method for economizing on the growth of game trees [Sl71], [Nl71], [Wn77]. Under the alpha-beta strategy, a nonterminal node is "cut-off" or abandoned whenever it is known from the node's context that its ultimate value cannot possibly rise to the root of the tree under the rules of minimax value propagation. Although the alpha-beta strategy has at times been called a heuristic, it is rather an optimization admitting no possibility of error in top-level move selection.

A number of studies have estimated the savings offered by the alpha-beta strategy under various conditions [Bd78a], [FGG73], [Gr76], [KM75], [Nw77]. In sum, these findings indicate that the alpha-beta strategy significantly slows (but does not eliminate) the exponential growth problem for game tree searching. In view of its ease of implementation in ordinary depth-first searching via recursion, the method has seen wide application.

1.2. Game playing under retentive control.

At the time of the alpha-beta strategy's original discovery (e.g. [HE63]; see [KM75] for a historical summary), recursion was the most advanced control form generally available in higher-level programming languages. Today, however, more advanced control structures are common, particularly in AI languages [BR74]. If one puts aside true concurrency (see [Kl78] and [Bd78b] for approaches to alpha-beta pruning in this setting), the central feature of these advanced structures is retentive control, whereby activations of program units are not necessarily deleted when exited. This capability offers a logical

parallelism that goes beyond recursion, and can support such programming styles as coroutines, backtracking, and demand-driven generator functions.

In the specific domain of game playing, retentive control offers the following attractive search methods as alternatives to depth-first search:

i) breadth-first search, in which the game tree is searched in ply-order;

ii) best-first search, in which nodes are examined in an order dictated by some dynamic measure of "promise", and

iii) evolving tree search, in which a significant portion of the game tree is retained from one move cycle to the next, in order to eliminate its reconstruction in subsequent searches.

1.3. Alpha-beta pruning under retentive control.

Each of the search strategies above goes beyond ordinary depth-first search in that some degree of lateral mobility in the game tree is required. Nevertheless, the notion of alpha-beta pruning is still sensible. Our goal here is to show the feasibility and attractiveness of adapting alpha-beta pruning to this logically parallel setting. In particular, procedures will be defined for performing the following actions in any order on appropriate nodes arbitrarily selected from a game tree:

i) expand a node that is currently terminal;

ii) resume move generation from a nonterminal node that was heuristically suspended;

iii) re-root the tree by selecting a top-level descendant to be the new root (thereby "evolving" the game tree one level through a move decision);

iv) redevelop a subtree, i.e. expand and resume enough of its nodes to satisfy some extended measure of thoroughness (e.g. depth of terminals), and

v) restart move generation from a nonterminal node that was previously cut-off, but may no longer be due to the alteration of its cutting value through actions of type (i) - (iv).

Our correctness and economy conditions will be the following:

i) minimax correctness: upon the completion of each action of type (i) - (iv), including any indicated node restarts (v), the root node will possess the correct minimax value for the current tree (and hence will indicate the correct next move according to the search done thus far);

ii) cutting correctness: after each move selection, i.e. rerooting and tree redevelopment according to some lookahead heuristic (e.g. the desired depth of terminals), the resulting root value and indicated move will be consistent with that of the full tree delimited by that heuristic, even though the actual tree will usually be smaller (e.g. have some "high" terminals), and

iii) cutting effectiveness: whenever move generation is done, it will be done under the most stringent alpha-beta thresholds dictated by the current tree at large.

2. A-B TREES.

2.1. Implicit vs. explicit tree representations.

In traditional depth-first game players, the game state information (we will use "board position") is typically represented in the following fashion:

i) there is a single global data structure B representing the current board position;

ii) as move lookahead is done by the recursive search function, incremental changes are made to B for each hypothesized move, and

iii) as each such move is retracted, its incremental change is explicitly "undone" on B.

This approach does not conveniently generalize to the logically parallel search methods outlined in section 1.2. Instead, one of the following two alternative approaches to game tree representation is generally used:

a) [implicit] The global data structure B reflects the given board position associated with the root of the tree and is left unchanged during the search. The incremental change to B associated with each move is retained as data local to the search process at each node. The board access functions then examine these changes in LIFO order along the path from the current node to the root. The game tree thus exists implicitly through the control relationships of the suspended search functions at each node. By coupling coroutine-based control with this distributed representation for B, rapid "context switching" or lateral tree

mobility is attained.

b) [explicit] There is only one re-entrant search process, and all state information associated with each board position is directly encoded into an explicit context tree [MPST78] of board positions. Each node may possess a complete B value, or an incremental change as in (a). In either case context switching is rapid, since the single re-entrant search process may be redirected to any existing node without extensive global or local data manipulations.

Although approach (a) is a recognizable trend in AI programming (due in part to the availability of supporting features in languages such as CONNIVER [MS72]), we will adopt approach (b) here. While the benefits of our method may ultimately be greater under approach (a), using approach (b) here offers the following immediate advantages:

i) exposition will be simpler since there will be no need to define the coroutine and context management facilities requisite to approach (a), and

ii) the results obtained will be available immediately to AI researchers using conventional high-level languages.

Given this choice, it is useful to reformulate the classical alpha-beta strategy in terms of the static properties it induces on such trees, as opposed to its more customary dynamic pruning effect. This will be done in three stages. First, a particular representation of ordinary minimax game trees will be defined in section 2.2. Second, that representation will be extended in section 2.3 to "A-B trees", which include node cutting relationships. Third, a

notion of A-B tree "validity" (thoroughness with respect to a particular lookahead condition) will be defined in section 3.1. Finally, the classical alpha-beta algorithm will be presented recast in these terms in the remainder of section 3.

2.2. A game tree representation.

A game tree T is a set of nodes obeying the following conditions:

2.2.1. Nodes. A node in T is a record defined as follows (in PASCAL [JW74] terms):

```
type    nodedeg  = 0 .. maxnrofdesc;
        descnr   = 1 .. maxnrofdesc;
        player   = (max, min);
        position = ... {representation of board position};
        nodeval  = -statvalmax .. statvalmax;

        ptrnode  = ^node;

        node     = record nrdesc: nodedeg;
                          desc: array [descnr] of ptrnode;
                          parent: ptrnode;
                          onmove: player;
                          pos: position;
                          value: nodeval
                   end;
```

2.2.2. Terminal nodes. A node n in T is terminal if n.nrdesc=0.

2.2.3. Nonterminal nodes. A node n in T is nonterminal if n.nrdesc>0. Then for each i, 1<=i<=n.nrdesc, the descendant link n.desc[i] points to a descendant node d of n, with d^.parent = n; d is said to be a rank i node. The descendant links of nodes in T form a tree structure on T in the obvious manner. The root node of T is denoted root(T); the subtree rooted by any node n in T is denoted tree(n).

2.2.4. onmove. Root(T) may have either max or min in its onmove field, reflecting the player whose turn it currently is to move. However, each other node n in T must obey the following recursive rule:

if n.onmove = max (min), then for each i, 1<=i<=n.nrdesc,

n.desc[i]^.onmove = min (max).

2.2.5. pos. The pos field of each node n represents the board position associated with that node in the game tree. The exact nature of this value (e.g. its structure, whether the pos value is incremental or complete, etc.) is unimportant to us. Naturally there are some game-dependent conditions that apply (e.g. that the net change in pos values from a nonterminal node to each of its descendants corresponds to a different legal move by n.onmove, etc.), but we leave these unstated here.

2.2.6. value. The value at a node n is determined as follows:

i) if n is a terminal node, n.value is the static value of n.pos.

ii) if n is a nonterminal node and n.onmove = max (min), then n.value is the maximum (minimum) over 1<=i<=n.nrdesc of n.desc[i]^.value.

2.2.7. desc order. We assume (for reasons to become clear in section 3) that the descendants of each nonterminal node n are sorted in best-first order of desirability (from n's viewpoint). That is, if n.onmove = max (min), for each i, 1<=i<=n.nrdesc-1, n.desc[i]^.value >= (<=) n.desc[i+1]^.value. Thus n.value = n.desc[1]^.value.

2.3. A-B trees and their representation.

Our goal of attention mobility in the game tree demands knowledge not only of the cut-off status of each node, but also explicit cutter-cuttee relationships. To that end, we now extend our representation of game trees to that of A-B trees, which include such information. (The more perspicuous name "alpha-beta trees" has, unfortunately, a quite different extant meaning [CW78].) We assume the following new fields are added to node records:

```
node    = record   ... {fields as before}
                   cut: boolean;
                   cutter, cutnodes, nextcut: ptrnode
          end;
```

Definition. An A-B tree is a game tree (as per section 2.2), in which the node fields adjoined above obey the following conditions:

If n.cut=true, then node n is currently cut-off by some other node in T. In this case:

i) n is a nonterminal node, not all of whose immediate descendants have been generated;

ii) n.cutter points to the node cutting n (call it c). The following must be true of c and n:

a) c.cut = false;

b) c.onmove = n.onmove;

c) c is an immediate descendant of an ancestor of n, but not itself an ancestor of n, and

d) if c.onmove=n.onmove=min, then c.value>n.value

(alpha cut-off); otherwise (i.e. if c.onmove=n.onmove=max), c.value<n.value (beta cut-off);

iii) n.cutnodes = nil.

If n.cut=false, then node n is currently not cut-off; instead, n itself may be cutting one or more other nodes in T. Let S be that set, i.e. S = { m in T such that m.cut=true and m.cutter=n }. Then:

i) if S is empty, then n.cutnodes=nil; otherwise:

ii) n.cutnodes points to one node in S, and the remaining nodes in S are chained together via their nextcut fields (with a final nil terminator).

Definition. An A-B subtree is a subtree of an A-B tree. Note that an A-B subtree is itself not necessarily an A-B tree, since some of its nodes may be cut by external nodes. When a subtree in an A-B tree is in fact an A-B tree itself, we say it is an independent A-B subtree.

Readers well-acquainted with the alpha-beta method may have found three aspects of this representation notable. We point out each of these here, along with a brief word of motivation.

First, one may note that, contrary to the normal depth-first viewpoint, the cutter c of a cut-off node n is not a complementary (i.e. max for min, etc.) ancestor of n, but rather a like (max for max, etc.) sibling or ancestor sibling. Two observations are relevant:

i) this approach is consistent with the customary viewpoint, in that we have simply focused on a potential source of a cutting ancestor's value, rather than the ancestor itself, and

ii) this approach is likely to economize on node restarting, for changes in the ancestor's value will not bring into question the restart of n unless caused by the strengthening of c's value (more on this later in section 4.3).

Second, one may note that while the values of cut-off nodes are permitted to rise in the tree's minimax value computation, cut-off nodes are not themselves eligible to do cutting. There are several reasons for this, including:

i) node record size can then be minimized through the sharing of fields that cannot simultaneously be active (e.g. cutter and cutnodes --- unexploited here), and

ii) a cutter-cuttee dichotomy is maintained which will simplify our algorithms, while

iii) no loss of cutting power results, as shown by the following:

Theorem 1. A cut-off node immediately below a live node (i.e. one currently undergoing move generation) cannot contribute a sharpened alpha or beta value for its siblings.

Proof (Max case; min is symmetric). Let n1 be a min node whose descendant n0 has just undergone (beta) cut-off. Let the (alpha, beta) values prevailing on n0 and n1 be (a0, b0) and (a1, b1) respectively. Then v0>b0. Two cases apply.

If v1<b1 then b0=v1. Thus v0>b0=v1, so v1 is not dislodged from n1 and b1 will prevail on the next descendant of n1.

If v1>=b1, then b0=b1, and v0>b0=b1, so b1 will again remain in effect, even if v0 is installed at n1 due to v0<v1. The alpha value a1, of course, is unchanged in either case.

Finally, it may be noted that node cutting is not done on tie values (i.e. strict inequalities must hold on cutting tests). There are two justifications for this deviation from customary practice:

i) the equality case is sufficiently rare that no noticeable performance degradation occurs, and

ii) a pathological A-B tree condition termed "cross-cutting" (see section 3.1) is thereby avoided.

3. GROWING VALID A-B TREES.

Our introductory treatment of A-B trees will now be concluded in two final stages. First, the notion of A-B tree "validity" will be defined; secondly, the classical alpha-beta algorithm will be presented, reformulated to generate valid A-B subtrees.

3.1. Valid A-B trees.

The A-B tree data structure defined in section 2 provides a concrete representation for game trees with explicit node cutting relationships. In addition to the structural well-formedness rules given there, a further notion of A-B tree correctness will be necessary to validate the tree expansion algorithms to be presented in subsequent sections. We begin with a series of definitions.

3.1.1. Full game trees. We say a game tree T is a full game tree with respect to the "horizon" function DEEPENOUGH if:

i) for each node n in T: n is terminal iff DEEPENOUGH(n.pos) = true, and

ii) for each nonterminal node n in T: the full complement of n's descendants has been generated (denoted n.nrdesc = NROFMOVESFROM(n.pos)).

3.1.2. Key nodes. Let T now be an A-B tree. Then the key nodes of T are the smallest set satisfying the following recursive definition:

i) Root(T) is a key node of T.

ii) Let n be a key nonterminal node of T. Then:

a) if n is uncut, then for each i, 1<=i<=n.nrdesc, n.desc[i] is a key node, otherwise:

b) if n is cut-off, then n.desc[1] is a key node.

3.1.2. A-B subtree validity. Let T be an A-B tree, and n a node in T. We say tree(n) is a valid A-B subtree with respect to DEEPENOUGH if when n is assumed to be key in T:

i) for every key node m in tree(n): m is terminal iff DEEPENOUGH(m.pos) = true, and

ii) for every uncut key nonterminal node m in tree(n): m.nrdesc = NROFMOVESFROM(m.pos).

Notes: Since root(T) must be key, saying T is valid means (i) and (ii) are true of every key node in T. Henceforth, for brevity, "valid A-B subtree" and "full game tree" will both implicitly mean "with respect to DEEPENOUGH".

3.1.3. Underlying game trees. Let T be an A-B subtree and G be a full game tree, with root(T).pos = root(G).pos. Then we say G underlies T.

Our general objective in subsequent sections will be to provide a flexible means for generating A-B trees that are consistent with, but typically more compact than, all underlying full game trees. As we shall see, subtree validity will be our principal means toward that end. These notions will be sharpened in section 3.1.5, after one further lemma.

3.1.4. Rank 1 cutter nodes. It will be useful at times to put A-B subtrees into a certain standard form, as indicated by Lemma 1.

Lemma 1. Let T be an A-B subtree. Then T can be modified so that every cut-off node in T is either (i) cut-off from outside T, or (ii) cut-off by a rank 1 node in T. Moreover, if T is valid then the modified form of T is also valid.

Proof. Let n1 be a cut-off node in T with n1.cutter = n0 in T possessing rank > 1. Assume inductively that n1 is the first such node in postorder. Denote n1.parent as n2, etc. Let nk be the sibling of n0 on the path from n1 to n0.parent. We assume n0 and n1 are min nodes (the alpha cut-off case; the beta case is symmetric).

Denote nj.value as vj, for 0 <= j <= k. Now let ni be the first min node on the path from n1 to nk that possesses a rank 1 sibling ni* with value vi* such that vi* > vi. Observe that if ni exists, then v1 = v2 >= v3 = ... >= vi, since the odd nodes are rank 1 min nodes. Hence v1 >= vi.

Case 1. Node ni exists; since vi* > vi, ni* is a potential cutter of n1. If ni* is uncut, make it the new cutter of n1. Otherwise, make ni*.cutter the new cutter of n1. By induction, ni*.cutter is outside T, or is rank 1. Finally, note that since we have neither expanded any terminals nor uncut any nodes, if T was valid then it remains so.

Case 2. No such node ni exists; hence nk has no weaker siblings and vk >= v0. Moreover, v1 >= vk by the path reasoning used above. Hence v1 >= v0, refuting alpha cut-off of n1 by n0, i.e. v1 < v0.

Note: Observe that this proof would not hold if cutting were done on equality, e.g. on v1 <= v0 above. In such a case subtree cross-cutting can occur, in which two sibling nodes each cut-off nodes in the other's subtree, with no alternative cutter available higher in the tree. This pathological condition can block the correctness of an otherwise valid A-B tree.

3.1.5. Consistency of valid A-B subtrees. Our principal result relating valid A-B subtrees and underlying full game trees will now be presented, after a preliminary lemma.

Lemma 2. Let tree(n) be an A-B subtree. Suppose n.desc[i] = n0 cuts some n1. Then if n1 is uncut and resumed, no value change will result at n0.parent in the resulting reminimaximization along the path from n1 to n0.parent.

Proof (min case; max case is symmetric). Define n2, ..., nk and v0, ..., vk as in the proof of Lemma 1. As a result of resuming min node n1, v1 cannot be weakened, i.e. v1' <= v1. Consequently, the value at max node n2 cannot be strengthened, i.e. v2' <= v2, etc.; ultimately we have vk' <= vk. But by hypothesis n0 is rank 1, so v0 >= vk, whence v0 >= vk'. (Note that any additional node resumptions in tree(nk) would have to be triggered by a strengthened min node ni; by analogous reasoning these can only affect vk by strengthening it further.)

If vk did not cut-off any nodes in tree(n0), v0 remains unchanged and continues to dominate at n0.parent. Now suppose vk did cut-off some node n* in tree(n0). Then restarting n* due to vk' < vk could only cause v0' < v0 if v0 = v*. But then v0 = v* < vk, contradicting the original rank 1 hypothesis for n0. Hence again v0 continues to

dominate at n0.parent.

Theorem 2. Let T be an A-B tree, and n such that tree(n) is a valid A-B subtree of T. If tree(n) is an independent A-B subtree, then n.value = root(Gn).value, for any full game tree Gn underlying tree(n).

Proof. By Lemma 1, we can assume all cut-off nodes in tree(n) are cut from outside tree(n), or possess rank 1 cutters. Now assume n is key; we proceed by induction on the key nodes of tree(n) in postorder. Without altering n.value, we will ensure by construction that the following property is true of tree(n) after our visit:

P(n): every key node in tree(n) rooting an independent A-B subtree possesses a value equal to that of its corresponding node in Gn.

Case 1: n is terminal. Then tree(n) is trivially an independent A-B subtree, and P(n) is true directly by the horizon condition shared by tree(n) and Gn.

Case 2a: n is a cut-off nonterminal. Node n.desc[1] = d is then key; by induction, P(d). Moreover, by our rank 1 cutter assumption, if d is cut then d.cutter is outside tree(n). Node n itself is clearly cut from outside tree(n). Delete any descendants of n of rank > 1; this does not alter n.value. One may then conclude P(n).

Case 2b: n is an uncut nonterminal. Then all descendants n.desc[i] = ni exist and are key; hence P(ni). Suppose n1 cuts some node n* in tree(ni). Replace the missing descendants at n* with the corresponding subtrees from Gn. By Lemma 3, this does not change the

value at n. Repeat this process exhaustively; by our rank 1 cutter hypothesis the only cut-off nodes remaining in tree(ni) are cut from outside tree(n). If no such nodes exist in tree(ni), then tree(ni) is identical to its corresponding subtree in Gn by construction. If all ni meet this condition, then tree(n) is similarly correct; otherwise n.value may still be incorrect. In either case, P(n).

Our final result is obtained through Corollary 1, which follows directly from Theorem 2:

Corollary 1. Let G be a full game tree underlying a valid A-B tree T. Then the top-level move indicated by T is consistent with that indicated by G.

3.2. A valid A-B subtree algorithm.

3.2.1. Specification. We now present GROWTREE, a method for generating valid A-B subtrees. The arguments to GROWTREE are:

n: a pointer to an uncut node in an A-B tree T. We assume any existing descendants of n either root valid A-B subtrees, or span a node m heuristically suspended (denoted HEURISTICSTOP(m)), and

alpha, beta: pointers to cutting threshold nodes above n in its tree, in the sense of section 2.3; if no alpha cutting value has yet been established for n (e.g. n is the global root), then alpha = topmax, a pointer to a special pseudo node with topmax.value = -statvalmax = INITVAL(max) (similarly for beta and topmin, a pointer to a pseudo node with topmin.value = statvalmax = INITVAL(min)).

The effect of GROWTREE under the input assumptions above is to produce an A-B subtree tree(n) which is valid if it does not span a heuristically suspended node.

3.2.2. Code for GROWTREE. Code for GROWTREE and a few of its auxiliary routines will now be given. The other routines referenced may be found in the appendices, organized as follows:

Appendix A: miscellaneous auxiliary routines (in complete PASCAL);

Appendix B: game-dependent functions (in skeletal PASCAL), and

Appendix C: heuristic control functions (in skeletal PASCAL).

```
procedure GROWTREE(n, alpha, beta: ptrnode);

var     a, b: ptrnode;

begin   {generate new descendants (with subtrees) from node n
                under cutting thresholds given by alpha and beta}
        if DEEPENOUGH(n^.pos) {horizon limit} then
                n^.value:=STATICVALUE(n^.pos)
        else
        begin   {expand n}
                a:=alpha;  b:=beta;
                if n^.nrdesc>0 then {compute local alpha & beta,
                                relying on sorted descendants}
                        LOCALPHABETA(n^.desc[1], a, b);
                while not n^.cut and
                        (n^.nrdesc<NROFMOVESFROM(n)) and
                        not HEURISTICSTOP(n) do
                begin   CREATENEXTDESC(n);
                        GROWTREE(n^.desc[n^.nrdesc], a, b);
                        CHECKDESC(n, n^.desc[n^.nrdesc], a, b);
                        if SHOULDBECUT(n, alpha, beta) then
                                CUTOFF(n, alpha, beta)
                end
        end;
        SORTDESC(n)
end {GROWTREE};
```

```
procedure CHECKDESC(p, q: ptrnode; var alpha, beta: ptrnode);

begin   {see if value at desc q is new local best for p;
                update alpha or beta as appropriate}
        if BETTER(q^.value, p^.value, p^.onmove) then
                INSTALLVALUE(p, q, alpha, beta)
end {CHECKDESC};


function SHOULDBECUT(n, alpha, beta: ptrnode): boolean;

begin   {test if node n should be cut-off under thresholds
                given by alpha and beta}
        if n^.nrdesc<NROFMOVESFROM(n^.pos) then
                case n^.onmove of
                        max:    SHOULDBECUT:=CUTS(beta, n);
                        min:    SHOULDBECUT:=CUTS(alpha, n)
                end
        else    SHOULDBECUT:=false
end {SHOULDBECUT};


procedure CUTOFF(n, alpha, beta: ptrnode);

begin   {insert n on appropriate cut list of alpha or beta}
        n^.cut:=true;
        case n^.onmove of
                max:    INSERT(n, beta);
                min:    INSERT(n, alpha)
        end
end {CUTOFF};
```

3.3. Correctness.

The correctness of GROWTREE can be argued as follows:

Theorem 3. Let n, alpha, and beta be as specified in section 3.2.1. Then GROWTREE(n, alpha, beta) performs as specified.

Proof (informal). Assume n is key. If n.pos is DEEPENOUGH, n will be left terminal, with tree(n) thereby valid. Otherwise, some number of subtrees will be grown by GROWTREE recursively; by induction these will be valid or contain suspended nodes. If any suspended nodes result in tree(n), then GROWTREE performs as desired.

Otherwise, all subtrees rooted by descendants of n will be valid. Hence tree(n.desc[1]) will surely be valid. If n does not become cut-off, then n.nrdesc = NROFMOVESFROM(n.pos) after GROWTREE. In either case, tree(n) is valid.

Note: Henceforth (except in section 5), we will assume no nodes have been suspended through HEURISTICSTOP. This will expedite our presentation, and permit stronger validity results.

3.4. Appraisal.

The worst-case economic behavior of GROWTREE may be estimated as follows: Let f be the maximum degree (fan-out) of any node grown, d the depth (relative to n) of the deepest new terminal, and m the number of new nodes grown. Then:

space: GROWTREE consumes O(m) space for the new nodes produced. Its working space (recursion stack, with locals) is O(d).

time: Let O(1) denote fixed running time. Then all routines other than GROWTREE itself called within GROWTREE's loop are O(1). Moreover, one new node is created on each loop cycle; hence the time spent local to those loops is O(m). The time spent outside the loop in each call is O(f log f) (from SORTDESC). Thus the time overall for GROWTREE is O(m) * O(f log f) = O(m), assuming m>>f.

4. EXPANDING A TERMINAL NODE.

Our first "random access" operation to be defined on an A-B tree T will be to expand an arbitrary terminal node t. This operation will be performed in four stages:

i) calculation of the cutting thresholds at each node along the path from root(T) to t;

ii) expansion of t into an A-B subtree;

iii) reminimaximization of the values along t's path to root(T), and

iv) restart of any previously cut-off nodes no longer cut-off due to the rise of t's new value.

4.1. FINDPATH.

Step (i) above will be accomplished with the aid of a global stack ps (for "path stack") and a procedure FINDPATH. The stack ps is defined as follows:

```
type    psentry = record alpha, beta: ptrnode;
                         wascut: boolean;
                         oldvalue: nodeval
                  end;

var     ps: array [0..depthmax] of psentry;
        pstop: -1..depthmax;  {top pointer for ps}
```

FINDPATH behaves as follows:

1) The path from t to root(T) (t's "root path") is determined by recursive descent along parent links.

2) The path stack ps is then constructed working backward from the root. At each node n along that path a new record is pushed onto ps, and:

a) n.value is saved in the oldvalue field of that record;

b) n.cut is saved in the wascut field of the record, and n is uncut if n.cut = true;

c) a new n.value is computed ignoring the immediate descendant of n along the path to t, and

d) the (alpha, beta) pair prevailing on that immediate descendant is determined and saved in the record.

Code for FINDPATH:

```
procedure FINDPATH(n, d: ptrnode);

begin   {retract values along root path from descendant d of n;
         build in ps the path stack prevailing on d}
        if d=root then
                PSPUSH(d^.value, false, topmax, topmin)
        else
        begin   {build pathstack tail first}
                FINDPATH(n^.parent, n);
                PSPUSH(d^.value, d^.cut, ps[pstop].alpha
                                         ps[pstop].beta);
                if d^.cut then
                        UNCUT(d);  {kills any restart request as well}
                REOPTIMIZE(n, d, ps[pstop].alpha, ps[pstop].beta)
        end
end {FINDPATH};
```

```
procedure REOPTIMIZE(p, q, alpha, beta: ptrnode);

begin   {recalculate value at p, ignoring the value at immediate
                desc q; update alpha or beta as approp.}
        if p^.nrdesc=1 then
                p^.value:=INITVAL(p^.onmove)
        else    INSTALLVALUE(p, BESTSIB(q), alpha, beta)
end {REOPTIMIZE};


function BESTSIB(p: ptrnode): ptrnode;

begin   {find best sibling of p; at least one assumed to exist}
        if p^.parent^.desc[1]=p then
                BESTSIB:=p^.parent^.desc[2]
        else    BESTSIB:=p^.parent^.desc[1]
end {BESTSIB};
```

Theorem 4. Suppose FINDPATH(n, d) is invoked, with d a descendant of n in an A-B tree T. Then upon FINDPATH's completion the nodes on d's root path are configured as they would be in the depth-first case were d the newest node created (ignoring descendant ordering).

Proof. Follows directly from the description above.

4.2. Expanding t.

Once FINDPATH has completed, the expansion of t can be accomplished by simply:

a) initializing t's value to INITVAL(t.onmove), and

b) using the (alpha, beta) pair in the top record of ps as cutting thresholds for GROWTREE.

4.3. Reminimaxing the root path from t.

Definition. An extended A-B tree is an A-B tree as defined in section 2.3, but with perhaps some of its cut-off nodes (those "marked for restart") on the cutnodes list of the pseudo node restarthead.

Note: the previously defined concepts of "key node" and "valid A-B subtree" apply without modification to extended A-B subtrees.

Once the new value at t has been established through GROWTREE, the path to the root is examined in upward order. At each node n along that path we must:

a) recalculate n's minimax value, and

b) re-establish n's cutting relationships. In particular:

1) using the (alpha, beta) pair for n saved on ps, determine whether n should now be cut-off. If so, do so; otherwise:

2) if n was cut prior to FINDPATH (perhaps by being restart marked), mark it for restart;

3) if n is thereby cut-off, mark for restart all of the nodes it previously cut-off; otherwise:

4) see if the new value at n has been strengthened with respect to the stacked old value. If so, re-check its cut-off nodes; otherwise, retain its entire cut-off list.

The full details of this logic are given in BACKUP, NEWNODEVAL and EXPANDTERM below:

```
procedure BACKUP(d: ptrnode);

var     q: ptrnode;

begin   {back up newly installed value from d using path stack ps}
        q:=d;
        while q<>root do
        begin   PSPOP;  q:=q^.parent;
                SORTDESC(q);  {could do linear merge}
                q^.value:=q^.desc[1]^.value;
                if SHOULDBECUT(q, ps[pstop].alpha, ps[pstop].beta) then
                        CUTOFF(q, ps[pstop].alpha, ps[pstop].beta)
                else
                if ps[pstop].wascut then
                        ADDRESTART(c);
                NEWNODEVAL(q, ps[pstop].oldvalue)
        end;
        PSPOP  {for root entry}
end {BACKUP}


procedure NEWNODEVAL(q: ptrnode; oldvalue: nodeval)

begin   {check implications of newly installed value at
                        previously uncut node q on root path}
        if q^.cut then {mark all nodes cut by q for restart}
                RESTARTALL(q);
        else
        if BETTER(q^.value, oldvalue, q^.onmove) then
                CHECKCUTS(q, q^.cutnodes, q^.cutnodes)
end {NEWNODEVAL};


procedure EXPANDTERM(t: ptrnode);

begin   {expand terminal node t}
        FINDPATH(t^.parent, t);
        t^.value:=INITVAL(t^.onmove);
        GROWTREE(t, ps[pstop].alpha, ps[pstop].beta);
        NEWNODEVAL(t, ps[pstop].oldvalue);
        BACKUP(t)
end {EXPANDTERM};
```

Theorem 5. Let t be a terminal node in an extended A-B tree T. Then after EXPANDTERM(t): (i) tree(t) is valid, and (ii) if T was valid before, it remains so.

Proof (informal). Through its use of GROWTREE, EXPANDTERM produces a valid subtree at t, given the (alpha, beta) prevailing on t as determined by FINDPATH. As a result of NEWNODEVAL and BACKUP, any node n whose cutter is removed as a result of a new value along t's root path is marked for restart, and still is cut-off. Hence the set of key nodes in T - tree(t) is unchanged, and validity is preserved.

4.4. Restarting newly uncut nodes.

As described above, EXPANDTERM can leave marked for restart some number of previously cut-off nodes. In general, it is preferable to accumulate these requests and batch process them; mechanisms for this purpose are given in section 8.

4.5. Appraisal.

We now consider the economic behavior of EXPANDTERM. Let d be the depth of t relative to root(T), d' the depth (relative to t) of its deepest descendant, m the number of nodes in t's new subtree, and r the length of the longest nextcut chain encountered by CHECKCUTS and RESTARTALL. Then:

space: through GROWTREE, EXPANDTERM consumes space O(m). Its temporary workspace is O(d) + O(d'), for ps and GROWTREE's recursion.

time: FINDPATH runs in time O(d), if we assume UNCUT to be O(1),

which it could be by doubly linking the nextcut fields. GROWTREE runs in time O(m) as before, and BACKUP runs in O(d) * [O(f) + O(r)], with the O(f) term coming from the new value merge. Moreover, an obligation for some number of node restarts may be incurred, which must be considered part of EXPANDTERM's overhead. If the time associated with nextcut list scanning and node restarting can be ignored (as will be suggested in section 9), then the overall time for EXPANDTERM is O(d) * O(f) + O(m).

5. RESUMING A SUSPENDED NODE.

5.1. The meaning of node resumption.

The function HEURISTICSTOP called in GROWTREE provides for a node under move generation to be heuristically abandoned. The mechanism for later resuming move generation from such a node is closely related to the corresponding code for terminal node expansion. The only differences are (a) the omission of INITVAL invocation, and (b) the preliminary screening for current cut-off (which is not a problem for terminal nodes).

5.2. Code.

The code for RESUMENODE is as follows:

```
procedure RESUMENODE(p: ptrnode);

begin   {resume move generation from p if not cut}
        if p^.cut then
                {no point in resuming}
        else
        begin   FINDPATH(p^.parent, p);
                GROWTREE(p, ps[pstop].alpha, ps[pstop].beta);
                NEWNODEVAL(p, ps[pstop].oldvalue);
                BACKUP(p)
        end
end {RESUMENODE};
```

5.3. Appraisal.

The validity implications of RESUMENODE can be argued in a style analogous to that of Theorem 5. Moreover, the economic analysis for EXPANDTERM holds equally well for RESUMENODE. Thus we have space consumed = O(m), workspace = O(d) + O(d'), and time = O(d) * O(f) + O(m).

6. RE-ROOTING A-B TREES.

6.1. Steps involved.

Selecting a top-level move in an A-B tree for evolution of the game can readily be accomplished as follows. Suppose the rank i move has been selected; thus root.desc[i] = m is to become the new global root. Then the following steps suffice:

i) mark for restart all nodes in m's subtree that are cut-off by siblings of m;

ii) clear the cutnodes list of m (since these must all be in subtrees about to be erased);

iii) erase the root node and all subtrees rooted by siblings of m, and

iv) make m the new global root.

6.2. Code.

Code for this process is given in MAKEMOVE:

```
procedure MAKEMOVE(i: descnr);

var     j: descnr;
        m: ptrnode;

begin   {make i-th best move from root}
        m:=root^.desc[i];
        for j:=1 to root^.nrdesc do
                if j<>i then
                        {find nodes in m's tree cut by siblings of m}
                        CLEARCUTSINTREE(root^.desc[j]^.cutnodes, m);
        m^.cutnodes:=nil;
```

```
        for j:=1 to root^.nrdesc do
                if j<>i then
                        ERASETREE(root^.desc[j]);
        dispose(root);
        root:=m
end {MAKEMOVE};
```

6.3. Correctness of MAKEMOVE.

The correctness of MAKEMOVE can be argued as follows.

Theorem 6. Let tree(root) be a valid A-B tree such that DEEPENOUGH(root.pos) = false (i.e. root is a nonterminal). Furthermore, let i be such that 1 <= i <= NROFMOVESFROM(root.pos). Then root.desc[i] exists (call it m), and after MAKEMOVE(i) tree(m) is a valid extended A-B tree.

Proof (informal). Since tree(root) is an A-B tree, root cannot be cut-off. Since tree(root) is valid, root.nrdesc = NROFMOVESFROM(root.pos), so m exists. Any node in tree(m) cut-off from outside tree(m) must have a sibling of m as its cutter. Hence by the operation of CLEARCUTSINTREE, each such node will be restart marked. Since MAKEMOVE uncuts no nodes in tree(m), the validity of tree(m) as an extended A-B tree is assured.

6.4. Appraisal.

The economic behavior of MAKEMOVE can be assessed as follows. Let m be the number of nodes currently in the tree and d its maximum depth. Then:

space: MAKEMOVE uses O(d) workspace to support the execution of ERASETREE, and results in a net gain in node working space.

time: MAKEMOVE is O(m), due to ERASETREE, ignoring node restart overhead.

7. SUBTREE REVALIDATION.

7.1. Why subtree revalidation?

Thus far, "random access" algorithms have been presented for terminal node expansion, suspended node resumption and tree re-rooting. These three operations might be sufficient in a game player whose search attention is fully heuristically controlled. That is, if one knows at all times exactly which tree nodes need consideration in order to satisfy current lookahead needs, then no further attention control facilities are needed. However, such direct attention control has the following drawbacks:

i) such fully random-access node processing incurs root path overhead (from FINDPATH and BACKUP) for each node;

ii) generally, attention control heuristics have only a part-time effect, and some means of "browsing" the tree is needed, and

iii) some systematic means must be provided for handling the node restart requests that result from node value changes.

Moreover, in the absence of attention control heuristics, some efficient means must be found for revalidating an entire A-B tree given an extended horizon condition DEEPENOUGH (e.g., when the game has evolved one move, and the resulting tree after MAKEMOVE is to be pursued further). These needs will be met through the notions of A-B subtree revalidation and A-B tree redevelopment.

Definition. Let T be an extended A-B tree, and n a node in T. If tree(n) is modified so as to become a valid A-B subtree, then we say tree(n) has been revalidated.

Definition. Let T be a valid extended A-B tree. If T is modified so as to become a valid A-B tree, then we say T has been redeveloped.

This section presents an approach to subtree revalidation; this technique will be then be applied in section 8 to the problem of A-B tree redevelopment.

7.2. An approach to subtree revalidation.

Suppose r is the root of an A-B subtree. Revalidating r's subtree may be accomplished as follows:

i) traversal of r's subtree in depth-first order including consideration (in best-first order) of existing moves at each node encountered, and generation of new descendant subtrees until all are generated or r is cut-off;

ii) expansion of terminal nodes encountered until stopped by DEEPENOUGH;

iii) re-calculation of the values at each nonterminal node n in r's subtree ab initio, i.e. as each new value is reported, ignoring previous values of yet-unrevalidated descendants of n, but

iv) inclusion of the full complement of sibling values, both revalidated and unrevalidated, in determining downward (alpha, beta) thresholds (i.e. the "rightmost spur" viewpoint of section 4.1 is retained).

7.3. Code.

Code for this process is embodied in REVALIDATE, SWEEPTREE, and related routines.

```
procedure REVALIDATE(n: ptrnode);

begin   {revalidate tree rooted at n, consistent with DEEPENOUGH}
        FINDPATH(n^.parent, n);
        SWEEPTREE(n, ps[pstop].alpha, ps[pstop].beta);
        BACKUP(n)
end {REVALIDATE};


procedure SWEEPTREE(n, alpha, beta: ptrnode);

var     i: integer;
        a, b: ptrnode;
        oldvalue: nodeval;

begin   {revalidate tree rooted at n, given alpha & beta}
        if DEEPENOUGH(n) then
                {do nothing}
        else
        begin   oldvalue:=n^.value;
                if n^.cut then
                        UNCUT(n) {clears any restart as well};
                i:=1;  n^.value:=INITVAL(n^.onmove);
                while (i<=n^.nrdesc) and not n^.cut do
                begin   a:=alpha;  b:=beta;
                        NEWALPHABETA(n, n^.desc[i], a, b);
                        SWEEPTREE(n^.desc[i], a, b);
                        if BETTER(n^.desc[i]^.value, n^.value,
                                        n^.onmove) then
                        begin   n^.value:=n^desc[i]^.value;
                                if (i=n^.nrdesc) or
                                        BETTER(n^.value,
                                                n^.desc[i+1]^.value,
                                                n^.onmove) then
                                if SHOULDBECUT(n, alpha, beta) then
                                        CUTOFF(n, alpha, beta)
                        end;
                        i:=i+1
                end;
                GROWTREE(n, alpha, beta); {does SORTDESC as well}
                NEWNODEVAL(n, oldvalue)
        end
end {SWEEPTREE};
```

Theorem 7. Let T be an extended A-B tree, and n be a node in T. Then upon completion of REVALIDATE(n), tree(n) is a valid subtree of T.

Proof (informal). The correctness of FINDPATH, BACKUP and NEWNODEVAL were argued previously in Theorems 4 and 5. Thus it remains simply to show that given cutting thresholds alpha and beta prevailing on n, SWEEPTREE(n, alpha, beta) produces a valid subtree at n.

We argue inductively. Suppose n is terminal. If DEEPENOUGH(n.pos), then n is unchanged and tree(n) is valid. Otherwise, GROWTREE(n, alpha, beta) is called, and by Theorem 3, a valid subtree at n results.

Now suppose n is nonterminal. SWEEPTREE will then be applied recursively to each of its existing descendants until all are so treated or until n is cut-off. NEWALPHABETA ensures that such recursive calls are given proper alpha and beta arguments. By induction, each of the resulting subtrees will be revalidated.

Now suppose n is cut-off as a result of a revalidated tree(d) for some descendant d. By the code for SWEEPTREE, that cut-off test will be made only if d would sort into rank 1 among its siblings. Hence if n is cut-off through d, tree(n) is valid.

On the other hand, if n is not cut-off through the root value of any revalidated subtree, then again GROWTREE(n, alpha, beta) will insure a valid subtree at n.

7.4. Appraisal.

The economic behavior of REVALIDATE(n) may be assessed as follows. Assume n is at depth d and its subtree contains m nodes before and m' nodes after revalidation, with maximum relative depth d'. Then:

space: REVALIDATE uses O(d) + O(d') workspace and consumes O(m') - O(m) added node space.

time: the phases of REVALIDATE each run in the following time:

i) FINDPATH: O(d).

ii) SWEEPTREE: O(m') * O(f log f), the latter factor arising from the SORTDESC called within GROWTREE at each level. Note that NEWALPHABETA as written causes an O(f^2) factor, but this can be reduced to O(f) through O(1) determination at each cycle of the best revalidated and the best uncut unrevalidated descendants. Hence SWEEPTREE and GROWTREE have identical worst case time behaviors.

iii) BACKUP: O(d) * O(f), ignoring nextcut chain scanning.

Overall, we have O(m') * O(f log f) + O(d) * O(f). The first term will dominate for subtrees rooted high in the tree, and the latter for subtrees placed more deeply.

8. TREE REDEVELOPMENT.

8.1. Why tree redevelopment?

A revalidated extended A-B tree may be obtained systematically via REVALIDATE(root), as just discussed, or through customized attention control via EXPANDTERM and RESUMENODE (sections 4 and 5). In either case, it then becomes necessary to establish the validity of the tree as an independent A-B tree, by processing all nodes marked for restart. As defined in section 7, this process is termed tree redevelopment.

8.2. Restart list processing.

Throughout the previous sections, nodes marked for restart have simply been collected on the cutnodes list of the pseudo node restarthead for later processing. There are several good reasons for this deferred processing approach:

i) a good probability exists that restart marked nodes will be recut (e.g. during SWEEPTREE), thereby obviating the need for a special restart;

ii) node restarting, as we shall see, involves a possible "rippling" effect causing other nodes to become restart marked (hence restart buffering is inescapable), and

iii) by deferring node restart until after revalidation of the entire tree, the available cutting context for their subsequent processing is maximized, thereby economizing on ultimate tree size.

Tree redevelopment may be accomplished by the following procedures:

```
procedure REDEVELOPTREE;

var     p: ptrnode;

begin   {given valid extended A-B tree at global root,
                make it valid as A-B tree}
        loop    p:=restarthead^.cutnodes;
        exit if p=nil;
                UNCUT(p);  {removes p from restart list}
                TRYRECUT(p);
                if not p^.cut then
                        REVALIDATE(p)
        end
end; {REDEVELOPTREE}


procedure TRYRECUT(p: ptrnode);

var     a, b, q: ptrnode;

begin   {see if p can be recut}
        a:=topmax;  b:=topmin;  q:=p;
        while q<>root do
        begin   if q^.parent^.nrdesc>1 then
                        LOCALPHABETA(BESTSIB(q), a, b);
                q:=q^.parent
        end;
        if SHOULDBECUT(p, a, b) then
                CUTOFF(p, a, b)
end; {TRYRECUT}
```

8.3. Correctness of REDEVELOPTREE.

The correctness of REDEVELOPTREE can be argued in the following three stages:

Theorem 8. Let tree(root) be a valid extended A-B tree. Then at the beginning of each cycle of REDEVELOPTREE's loop, tree(root) is once again a valid extended A-B tree.

Proof (informal). By Theorem 7, REVALIDATE(p) will preserve the validity of T as an extended A-B tree. Hence the only issue is whether the optimization of TRYRECUT could possibly compromise the validity of T as an extended A-B tree. The answer is no, argued as follows.

Suppose TRYRECUT(p) causes p to be recut. If p is not key in tree(root), it does not matter whether tree(p) is valid or not, so the recutting is safe. Now suppose p is key in tree(root). Then tree(p) was valid when cut-off by being restart marked; hence it remains so when recut by TRYRECUT.

Theorem 9. Let tree(root) be a valid extended A-B tree. Then REDEVELOPTREE terminates.

Proof (informal). By the well-formedness conditions of extended A-B trees, UNCUT, TRYRECUT, and REVALIDATE clearly terminate for each of the given arguments in REDEVELOPTREE. The only question remaining is whether an infinite sequence of node requests can result. But this is impossible, as follows.

Suppose REVALIDATE(p) is invoked. If p is recut as a consequence, since TRYRECUT(p) did not recut it, p's value must have changed (strengthened, in fact). But then at least one new terminal node must have been added to tree(p). Similarly, if p is not recut, it must have had at least one new descendant grown from it (since tree(p) is revalidated, and p must now possess its full complement of immediate descendants). In either case, tree(root) has increased in node count; since tree(root) can be no larger than any underlying

full game tree, this process must terminate.

Theorem 10. Let tree(root) be a valid extended A-B tree. Then REDEVELOPTREE produces a valid A-B tree at root.

Proof (informal). By Theorems 8 and 9, REDEVELOPTREE terminates with a valid extended A-B tree at root. But the restart list must then be empty, so tree(root) contains no externally cut-off nodes; hence it is valid.

8.4. Appraisal.

The economic behavior for REDEVELOPTREE may be estimated as follows.

space: REDEVELOPTREE uses bounded workspace itself, so the overall workspace is dominated by that of REVALIDATE, proportional to the maximum terminal depth in the final tree. The space consumed, of course, is proportional to the number of new nodes created.

time: Let k be the number of restart recuests processed by REDEVELOPTREE, and d the maximum depth of their root nodes. Then each of the k TRYRECUT calls runs in O(d) time. Now let k' be the number of nodes that fail to be recut by TRYRECUT, and m' the maximum node count of the revalidated trees. Then as per section 7.4, we have $O(m') * O(f \log f) + O(d) * O(f)$ time for each of these k' nodes. Overall, then, the maximum worst case time is $O(d) * O(k) + O(m') * O(f \log f) + O(d) * O(f)$.

9. EVALUATION.

9.1. Methods of evaluation.

The performance of the algorithms presented here might be evaluated in several ways, including:

i) estimation of their worst-case space-time behavior;

ii) probabilistic analysis;

iii) empirical evaluation in a real game player, and

iv) monte carlo studies in a simulated game environment.

Approach (i) has been taken informally throughout the previous sections. However, the insights gained thereby do not relate directly to actual expected performance. Approach (ii) is at present infeasible, given the complexity of the algorithms and the limited results this approach has yielded on the simpler depth-first case. While approach (iii) appears feasible, this avenue must await the attention of an expert possessing a suitable existing game player. A small study using approach (iv) will now be summarized in the interest of providing some preliminary estimates of the method's actual performance.

9.2. Studies performed.

9.2.1. Realistic environment. Previous monte carlo studies of the alpha-beta method have been oriented toward the non-retentive, depth-first case. In such a setting, static values are needed only at a fixed tree horizon. In our "evolutionary" tree setting, however, it is important to model the incremental nature of static values along tree paths. While "catastrophic"

static value changes can occur (e.g. loss of one's queen in chess), these are infrequent in comparison with small changes reflecting minor variation in such factors as comparative board control and material balance. Moreover, if such an incremental assumption is not made, the utility of full local cutting context in SWEEPTREE (i.e. via NEWALPHABETA) is dubious, given the statistical independence of terminal node values from ply to ply.

Our modeling approach, then, is simply to assign a random, uniformly distributed static value increment to each node, and compute its full static value by summing those increments along the path to the initial tree root (equivalently, by adding its increment to the full static value of its parent node). One may argue that this permits unrealistic "lop-sided" paths to result in the tree, but this is proper since one player may be seriously overmatched; if the players are fairly matched, such paths will rapidly be pruned.

9.2.2. Results. Table 1 summarizes the results of our preliminary monte carlo studies. It was felt that heuristic attention control could not seriously be modeled, so only tree re-rooting, revalidation and redevelopment of the entire new tree were performed. In each case (i.e. for various node degree and horizon depth combinations), the following measures were obtained:

i) the number of terminals in a full tree of those dimensions;

ii) the number of terminals predicted for the classical method, assuming independent static values;

iii) the measured average number of terminals under the classical method (to reveal the effect, if any, of our sense of static value dependence), and

iv) the average observed number of terminals in the evolving tree, along with the average number of node restarts required and their average depth, for two representative subcases:

a) consistent selection of the best top-level move (presumably statistically frequent), and

b) consistent selection of the worst top-level move (presumably statistically infrequent).

For each tree generated, a consistency check was made by comparing its root value with that of a randomly selected underlying full game tree.

9.2.3. Interpretation. Two principal insights about evolving tree game players may be gained from this small study:

i) that significantly smaller trees result in comparison with the classical approach, and

ii) node restarting is a negligible overhead due to

a) the low number of node restarts required (less than 6 per cent of NRP in Table 1), and

b) their deep placement, on average, in the current tree (average move height of 1 in Table 1).

9.3. Conclusions.

A method has been presented that adapts the classical alpha-beta method to exploit the attention mobility offered by retentive control. The following advantages and disadvantages appear to result:

advantages:

A1) heuristic freedom;

A2) elimination of subtree regeneration from one top-level move to another, and

A3) smaller trees in certain statistical settings.

disadvantages:

D1) space required for tree representation (but this is optionally controllable through selective subtree non-retention), and

D2) time overhead for random-access node processing (if desired), and slightly slower subtree traversal during tree redevelopment (due to descendant sorting in GROWTREE); however, this is presumably faster than actual move regeneration, and contributes to advantage A3 above.

Acknowledgement. The author is indebted to Prof. Robert Keller for his interest and insights in this work.

Table 1. Summary of Empirical Results.

N (degree)	D (depth)	NBP[1] full	NBP pred.[2]	NBP observ.	rank picked	NBP evolv.	restart[3] count	restart depth
2	3	8	7.0	6.8	1	6.2	0.1	2.0
					2	6.1	0.1	2.0
2	4	16	12.4	11.5	1	10.0	0.2	3.0
					2	10.1	0.0	3.0
2	5	32	22.2	20.7	1	16.1	0.2	4.0
					2	17.2	0.1	4.0
2	6	64	38.5	34.9	1	26.8	0.5	4.8
					2	27.8	0.2	4.8
3	3	27	18.9	17.6	1	15.6	0.6	2.0
					3	15.9	0.4	2.0
3	4	81	44.6	41.2	1	30.0	0.9	2.9
					3	33.4	0.2	2.9
3	5	243	106.9	102.6	1	68.9	2.0	4.0
					3	77.6	1.4	3.8
3	6	729	248.2	201.5	1	139.2	3.1	4.8
					3	143.5	0.7	4.7
4	3	64	38.1	37.2	1	32.7	1.9	2.0
					4	34.3	1.5	2.0
4	4	256	110.6	94.4	1	71.2	3.3	2.9
					4	73.1	0.8	3.0
4	5	1024	326.0	271.7	1	188.2	5.9	3.9
					4	215.2	3.7	3.9

Experimental conditions: static value increments uniformly distributed over [-1000, 1000]; 100 trials for each average.

Footnotes: [1]NBP ≡ number of bottom positions, i.e. terminal nodes.

[2]from [FGG73]; shallow cut-off only, independently distributed terminal values.

[3]restart ≡ REVALIDATE calls from within REDEVELOPTREE.

REFERENCES.

[Bd78a] Baudet, Gerard, "On the branching factor of the alpha-beta pruning algorithm", Artificial Intelligence 10,2 (April 1978) 173-199.

[Bd78b] ____, "The design and analysis of algorithms for asynchronous multiprocessors", Ph.D. disser., Carnegie-Mellon Univ., Report CMU-CS-78-116 (April 28, 1978) 182 pp.

[BR74] Bobrow, D., and Raphael, B., "New programming languages for artificial intelligence research, Computing Surveys 6,3 (Sept. 1974) 155-174.

[CW78] Choy, D. M., and Wong, C. K., "Optimal alpha-beta trees with capacity constraint", Acta Informatica 10 (1978) 273-296.

[FGG73] Fuller, S. H., Gaschnig, J. G., and Gillogly, J. J., "Analysis of the alpha-beta pruning algorithm", Technical Report, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa. (July 1973) 51 pp.

[Gr76] Griffith, Arnold K., "Empirical exploration of the performance of the alpha-beta tree searching heuristic", IEEE Transactions on Electronic Computers C-25,1 (Jan. 1976) 6-11.

[HE63] Hart, T. P., and Edwards, D. J., "The alpha-beta heuristic", MIT AI Lab Memo (Oct. 28,1963) 4 pp.

[JW74] Jensen, Kathleen, and Wirth, Niklaus, PASCAL User Manual and Report, Springer-Verlag (Berlin) 1974. 170 pp.

[KM75] Knuth, D. E., and Moore, Ronald W., "An analysis of alpha-beta pruning", Artificial Intelligence 6,4 (Winter 1975) 293-326.

[Kl78] Keller, Robert M., "An approach to determinacy proofs", Technical Report UUCS 78-102, Dept. of Computer Science, Univ. of Utah (March 1978).

[MPST78] Montangero, Carlo, Pacini, Giuliano, Simi, Maria, and Turini, Franco, "Information management in context trees", Acta Informatica 10 (1978) 85-94.

[MS72] McDermott, Drew, and Sussman, Gerald Jay, "The CONNIVER reference manual", MIT AI Lab Memo 259 (May 1972) 91 pp.

[Nw77] Newborn, N. M., "The efficiency of the alpha-beta search on trees with branch-dependent terminal node scores", Artificial Intelligence 8 (1977) 137-153.

[Nl71] Nilsson, Nils J., Problem-solving methods in artificial intelligence, McGraw-Hill, New York (1971) 255 pp.

[Sl71] Slagle, J. R., Artificial Intelligence: the heuristic programming approach, Mc-Graw-Hill, New York (1971).

[Wn77] Winston, Patrick Henry, Artificial Intelligence, Addison-Wesley, Reading, Mass. (1977) 444 pp.

APPENDIX A: Miscellaneous Auxiliary Routines.

```
procedure ADDRESTART(p: ptrnode);

begin   {add p^ to list of nodes to be restarted}
        p^.cut:=true;
        INSERT(p, restarthead)
end {ADDRESTART};


function BETTER(v, w: nodeval; pl: player): boolean;

begin   {test if v is better than w from viewpoint of pl}
        case pl of
                max:    BETTER:=v>w;
                min:    BETTER:=v<w
        end
end {BETTER};


procedure CHECKCUTS(p, cl: ptrnode; var result: ptrnode);

var     q: ptrnode;

begin   {examine nodes on nextcut list headed by cl
                to see which still should be cut by p; mark others for restart}
        if cl=nil then result:=nil
        else
        if not CUTS(p, cl) then
        begin   q:=cl^.nextcut;
                ADDRESTART(cl);
                CHECKCUTS(p, q, result)
        end
        else
        begin   CHECKCUTS(p, cl^.nextcut, cl^.nextcut);
                result:=cl
        end
end {CHECKCUTS};


procedure CLEARCUTSINTREE(cl, r: ptrnode);

var     p, q: ptrnode;

begin   {restart tree(r) nodes in nextcut chain headed by cl}
        p:=cl;
        while p<>nil do
        begin   q:=p^.nextcut;
                if PATH(p, r) then
                        ADDRESTART(p);
                p:=q
        end
end {CLEARCUTSINTREE};
```

```
procedure CREATENEXTDESC(p: ptrnode);

var     n: ptrnode;

begin   {create next descendant of p}
        new(n);  INITNODE(n);
        n^.parent:=p;
        p^.nrdesc:=p^.nrdesc+1;
        p^.desc[p^.nrdesc]:=n;
        n^.onmove:=OTHERPLAYER(p^.onmove);
        n^.value:=INITVAL(n^.onmove);
        n^.pos:=GENPOS(p^.pos, p^.nrdesc)
end {CREATENEXTDESC};


function CUTS(c, n: ptrnode): boolean;

begin   {test if value at node c can cut value at node n}
        CUTS:=BETTER(n^.value, c^.value, n^.onmove)
end {CUTS};


procedure ERASETREE(r: ptrnode);

var     d: descnr;

begin   {free each node in tree rooted at r}
        for d:=1 to r^.nrdesc do
                ERASETREE(r^.desc[d]);
        dispose(r)
end {ERASETREE};


procedure INITNODE(n: ptrnode);

begin   {initialize miscellaneous fields of node n}
        n^.nrdesc:=0;
        n^.cut:=false;
        n^.cutnodes:=nil
end {INITNODE};


function INITVAL(pl: player): nodeval;

begin   {initial value for nodes where pl is on move}
        case pl of
                max:    INITVAL:=-statvalmax;
                min:    INITVAL:=statvalmax
        end
end {INITVAL};
```

```
procedure INSERT(n, c: ptrnode);

begin   {insert n into cutnodes list of c}
        n^.cutter:=c;
        n^.nextcut:=c^.cutnodes;
        c^.cutnodes:=n
end {INSERT};


procedure INSTALLVALUE(p, q: ptrnode; var alpha, beta: ptrnode);

begin   {put value from q into p; update alpha or beta
                if new value beats previous threshold}
        p^.value:=q^.value;
        LOCALPHABETA(q, alpha, beta)
end {INSTALLVALUE};


procedure LOCALPHABETA(q: ptrnode; var alpha, beta: ptrnode);

begin   {use value at q (if uncut) to update alpha or beta if appropriate}
        if not q^.cut then
        case q^.onmove of
                max:    if q^.value<beta^.value then
                                beta:=q;
                min:    if q^.value>alpha^.value then
                                alpha:=q
        end
end {LOCALPHABETA};


procedure NEWALPHABETA(p, d: ptrnode; var a, b: ptrnode);

var     i: descnr;

begin   {find local alpha and beta bearing on desc d of p}
        case p^.onmove of
                max:    for i:=1 to p^.nrdesc do
                                if (p^.desc[i]<>d) and not p^.desc[i]^.cut
                                        and (p^.desc[i]^.value>a^.value) then
                                        a:=p^.desc[i];
                min:    for i:=1 to p^.nrdesc do
                                if (p^.desc[i]<>d) and not p^.desc[i]^.cut
                                        and (p^.desc[i]^.value<b^.value) then
                                        b:=p^.desc[i]
        end
end {NEWALPHABETA};
```

```
function OTHERPLAYER(pl: player): player;

begin   {invert player name}
        case pl of
                max:    OTHERPLAYER:=min;
                min:    OTHERPLAYER:=max
        end
end {OTHERPLAYER};


function PATH(p, q: ptrnode): boolean;

var     r: ptrnode;

begin   {test if q is on path from p to root}
        r:=p;
        while (r<>root) and (r<>q) do
                r:=r^.parent;
        PATH:=r=q
end {PATH};


procedure PSPOP;

begin   {pop top record off global path stack ps}
        pstop:=pstop-1
end {PSPOP};


procedure PSPUSH(ov: nodeval; wc: boolean; a, b: ptrnode);

begin   {push new entry onto path stack ps}
        pstop:=pstop+1;
        with ps[pstop] do
        begin   oldvalue:=ov;  wascut:=wc;
                alpha:=a;  beta:=b
        end
end {PSPUSH};


procedure REMOVE(n, m: ptrnode);

var     p, q: ptrnode;

begin   {remove node n from cutnodes list of node m}
        p:=m^.cutnodes;
        if p=n then
                m^.cutnodes:=n^.nextcut
        else
```

```
        begin   q:=nil;
                while p<>n do
                begin   q:=p;
                        p:=p^.nextcut
                end;
                q^.nextout:=n^.nextcut
        end
end {REMOVE};


procedure RESTARTALL(p: ptrnode);

var     q: ptrnode;

begin   {schedule all nodes on cutnodes list of p for restart}
        loop    q:=p^.cutnodes;
        exit if q=nil;
                REMOVE(q, p);
                ADDRESTART(q)
        end
end {RESTARTALL};


procedure SORTDESC(p: ptrnode);

var     i, j: descnr;
        q: ptrnode;

begin   {sort descendants of p, weakest first;
                we give a simple exchange sort here}
        for i:=1 to p^.nrdesc-1 do
                for j:=1 to p^.nrdesc-i do
                        if BETTER(p^.desc[j+1]^.value, p^.desc[j]^.value,
                                        p^.onmove) then
                        begin   q:=p^.desc[j];
                                p^.desc[j]:=p^.desc[j+1];
                                p^.desc[j+1]:=q
                        end
end {SORTDESC};


procedure UNCUT(p: ptrnode);

begin   {change p from cut to uncut status}
        p^.cut:=false;
        REMOVE(p, p^.cutter)
end {UNCUT};
```

APPENDIX B: Game-Dependent Functions.

```
function GENPOS(p: position; i: descnr): position;
begin   {generate position for i-th legal move from p}
        ...
end {GENPOS};

function NROFMOVESFROM(p: position): nodedeg;
begin   {number of legal moves from position p}
        ...
end {NROFMOVESFROM};

function STATICVALUE(p: position): nodeval;
begin   {compute static value of position p}
        ...
end {STATICVALUE};
```

APPENDIX C: Heuristic Control Functions.

```
function DEEPENOUGH(p: position): boolean;
begin   {see if horizon heuristic should be invoked}
        ...
end {DEEPENOUGH};

function HEURISTICSTOP(p: ptrnode): boolean;
begin   {see if node abandonment heuristic should be invoked}
        ...
end {HEURISTICSTOP};
```