Experimenting with Revisits in Game Tree Search

Subir Bhattacharya
Indian Institute of Management Calcutta
Joka, Diamond Harbour Road
P. O. Box 16757, Calcutta - 700 027, INDIA

Abstract

The oldest known game tree search algorithm Alpha-Beta is still the most popular one. All other algorithms in this area fall short of Alpha-Beta in one or more of the following three desired characteristics - high pruning power, low storage requirement and low execution time. This paper discusses how revisit of nodes can be used effectively in game tree search. A few strategies of introducing revisits in game tree search are presented. It is demonstrated that for any shape and ordering of the game tree to be searched, there always exists one strategy that, on an average, consistently evaluates less number of terminals than Alpha-Beta in comparable memory and time.

1. Introduction

In Artificial Intelligence, game tree search has ever been an active area of interest Over years, a number of schemes have been proposed for searching game trees, each having its own advantages and disadvantages. Interestingly, the oldest known game tree search algorithm, Alpha-Beta, is still the most popular one. To judge the effectiveness of any game tree search algorithm, we need to consider three different characteristics of the algorithm - its pruning power, its storage requirement and its execution time. Alpha-Beta is a recursive depth-first procedure and needs little memory; it is simple to implement and executes rather fast Other game tree search algorithms like SSS* [Stockman, 1979], SCOUT [Pearl, 1984] and their variations fall short of Alpha-Beta in one or other respect

SSS* is best-first and iterative, and consumes a large amount of storage space. Although the pruning power of SSS* is greater than that of Alpha-Beta, the improvement is more than offset by its excessive memory and overhead requirements. Recently, two recursive variations of SSS*, viz., RecSSS* ([Bhattacharya and Bagchi, 1993]) and RecDual* ([Reinefeld and Ridinger, 1994]) have been proposed. Empirical evidence ([Reinefeld and Ridinger, 1994]) shows that both of them dominate Alpha-Beta so far as terminal nodes examined are concerned, and for any game tree, one of them is always faster than Alpha-Beta. However, the storage requirement of either is even higher than that of SSS*. The cautious test-before-evaluate strategy of SCOUT [Pearl, 1984] is an extension of the Alpha-Beta algorithm. The evaluation phase is essentially Alpha-Beta, and the testing phase prior to it is an additional layer on top of Alpha-Beta, which enhances the pruning power of the evaluation phase. Apart from providing an alternative to Alpha-Beta, SCOUT is important since it introduced the concept of revisits in game tree search. While in Alpha-Beta and SSS* a terminal node in the game tree can be examined atmost once, SCOUT allows revisit of its nodes - once during the testing phase and then, if required, again during the evaluation phase.

The purpose of this paper is to show how revisits can be used effectively to examine less number of terminals than Alpha-Beta in comparable time using affordable amount of storage space. The base algorithm in which different strategies for revisits have been experimented is rather a new algorithm called QuickGame. This algorithm was obtained as a special case of the generalized game tree search algorithm GenGame [Bhattacharya and Bagchi, 1994]. QuickGame uses a mix of depth-first and best-first strategies to search the game tree and, for a uniform game tree of depth d and branching degree b, requires about (b * d)/2 units of storage space. When run on random trees, on an average, QuickGame examines less number of terminals than Alpha-Beta ([Bhattacharya and Bagchi, 1992]).

However, in practice, the game tree to be searched is very often strongly ordered, i.e., for any non-terminal node in the game tree the probability that the leftmost child will prove to be optimal for the node is very high. QuickGame has the drawback that when run on such strongly ordered game trees, it examines, on an average, many more terminals than Alpha-Beta. In this paper, we introduce the concept of revisiting a node in the algorithm QuickGame. We demonstrate that *for any shape and any ordering - random or strong - of the game trees to be searched, there exists at least one version of QuickGame which, on an average, examines less number of terminals than Alpha-Beta in comparable memory and time.*

We first take a quick look at the algorithm QuickGame and its relevant characteristics in section 2. Section 3 improves upon QuickGame by allowing revisit of nodes and

discusses the various ways in which revisits can be done. Section 4 details our experimental results comparing performances of Alpha-Beta, QuickGamc and its variations on game trees with both random and strong orderings. Section 5 concludes the paper.

Throughout the paper the root node s of the game tree T to be searched is assumed, without loss of generality, to be a MAX node. We use d to denote the length of the longest path in T, and b to denote the maximum number of sons that a non-terminal node in T can have, called the *branching degree*. T is a *uniform (b, d) tree* if every non-terminal node has exactly b sons, and if every path from s to a terminal node has length d; otherwise it is *non-uniform*. Individual nodes are named using a Dewey radix-b code [Pearl, 1984], in which the root node is represented by the empty sequence, and the b* < b sons of a non-terminal node x in T as x.i, 0 < i < b' A *static evaluation function v(.)* scores each terminal node from MAX's point of view. For any non-terminal node x in T, let t_x, be the *minimax* value of the subtree rooted at x. When x is a terminal node, t_x, = v(x).

2. A Quick Look at QuickGame

In SSS*, a global list of nodes called OPEN is maintained, which, at any instant during execution, contains one representative node from each solution tree of the game tree T. The high pruning efficiency of SSS* is explained by the fact that *all solution trees* in T are in contention at every instant But in the process, the size of the global OPEN list rises to $b^{d/2}$ units of storage space for a (b, d) uniform tree. Alpha-Beta evaluates more terminals because it does not keep all solution trees in simultaneous contention, but compares in a depth-first scan the current solution tree with the best one found so far. In QuickGame, we intend to go for a compromise. QuickGame is a recursive procedure that gets invoked only at non-terminal MAX nodes and at tenninal nodes. Below any non-terminal MAX node p, the subtree rooted at p is split up into as many subtrees as the number of sons of p, and all the solution trees of these subtrees are kept in simultaneous contention.

Let T be a uniform (b, d) game tree rooted at a MAX node s. For ease of understanding d can be taken to be even, but QuickGame runs just as smoothly on uniform trees of odd depth and on non-uniform trees. Let p be a non-terminal MAX node in T. If QuickGame gets invoked at p, it is passed three parameters: the node p, and the two bounds, alpha and beta. The bounds play a role comparable to that in Alpha-Beta : alpha, the lower bound, is the highest currently known value of all MAX ancestors of p, while beta, the upper bound, is the lowest currently known value of all MIN ancestors of p. At the root node s, alpha = and beta = . When QuickGame returns from the call at node p, the correct minimax value at p has been found, provided that the value lies within the alpha and beta bounds specified in the call. In order to find the best solution tree rooted at p, QuickGame uses a local OPEN list of b entries; an entry for a node x in OPEN has two fields, the Dewey code for x, and the current value h(x) of x. The node p is expanded, and the b *grandsons* p.i.0,0 < i < b, enter OPEN, with their h-values initialized to beta(p). It is as if the set of solution trees rooted at p is split up into b disjoint subsets, with each subset having a representative in the local OPEN of p. For a node x in the local OPEN, h(x) is an upper bound on the minimax value of the subtree rooted at x. QuickGame is now recursively invoked on the highest-valued node among these grandsons, ties being resolved in favour of nodes having lexicographically smaller codes.

Suppose p.i.y is currently the highest-valued node in the local OPEN of p, where 0 < i < b and 0 < y < b. This node being the representative of the currently most promising subset of solution trees, we explore below p.i.y by invoking QuickGame on the node. If p.i.y is terminal, the call returns the minimum over the upper bound on p.i.y and the static evaluation score v(p.i.y); otherwise the call returns the minimax value at p.i.y. On return, if p.i.y is not the rightmost among its brothers we replace p.i.y with its immediate right brother p.i.y+1 which belongs to the same set of solution trees to which p.i.y belongs, p.i.y+1 is the next node to be explored in this set of solution trees if this set is still the most promising one. Since p.i is a MIN node, the minimax value of p.i.y acts as an upper bound on the minimax value of p.i.y+1 and is assigned to h(p.i.y+l). Alternatively, if p.i.y is rightmost among its brothers (i.e., y = b-1), when the call on p.i.y returns, we have examined all those solution trees of the subtree rooted at p sharing the common MIN node p.i. Hence the value returned by the call acts as a lower bound on the minimax value of the MAX node p. So no replacement is made when y is b-1; h(p.i.y) in such situation reflects the value of one completely explored solution tree of the subtree rooted at p. The algorithm now selects the current highest-valued node from OPEN, and the entire cycle is repeated, until the node that gets selected has the form p.i.b-1, *and* QuickGame has already been called at this node at some earlier instant In such a situation, since h(p.i.b-l) is highest among nodes in OPEN and is the value of a solution tree of the subtree rooted at p, h(p.i.b-l) defines the minimax value of p, and the call returns from p to the grandfather of p. Since calls are made only at MAX nodes or terminals, and each call at a non-terminal node involves the use of a local OPEN of b entries, the total storage requirement for all the OPEN lists taken together is about (b * d)/2 units. The algorithm is invoked by calling QuickGame(s, -00 00).

Example 1 : Let the 2^4 = 16 numbers in Table 1 (see section 3) correspond to the static evaluation scores of the terminal nodes in left-to-right order of a uniform (2,4) game tree with a MAX node as root QuickGame evaluates 11 of the 16 terminal nodes in the time sequence indicated below the scores. Alpha-Beta, on the other hand, evaluates 14 of the nodes; owing to its depth-first nature, the terminals get visited in left-to-right order.

It is possible to construct examples where QuickGame evaluates more terminals than Alpha-Beta.

Example 2 : If the 16 static evaluation scores of the (2, 4) game tree were as given in Table 2, QuickGame would evaluate 9 of the terminals, but Alpha-Beta only 7 of them.

```
Function QuickGame(p : node; alpha, beta : real) : real;
var x : node;
    i, j, k : integer;
    r, h[0..b-1] : real;
    OPEN[0..b-1] : array of node;
begin
    If p is terminal then return minimum(beta, v(p));
    for (each child p.i of p) do begin
        h[i] := beta;
        If (p.i is terminal) then OPEN[i] := p.i
        else OPEN[i] := p.i.0;
    end;
    r := alpha; k := 0;
    while (h[k] > r) do begin          (* OPEN[k] = x.j, say *)
        h[k] := QuickGame(OPEN[k], r, h[k]);
        If (OPEN[k] is a terminal MIN node)
            or (OPEN[k] has no brother to its right) then
                r := maximum(r, h[k])
        else OPEN[k] := x.j+1;
        k := index in OPEN corresponding to node with
                maximum h-value;
    end;
    return r;
end;
```

For the (3, 4) uniform tree of [Knuth and Moore, 1975], Alpha-Beta makes 31 terminal node evaluations and QuickGame evaluates 33 terminals.

Thus QuickGame must be making some cutoffs not made by Alpha-Beta, and Example 1 verifies this. But Example 2 shows that on occasions Alpha-Beta makes some cutoffs not made by QuickGame.

The cutoffs made by Alpha-Beta are of two types, *shallow* and *deep* (see [Pearl, 1984]). When a node p1 causes a shallow cutoff below a node p2, the Dewey code of p1 must be lexicographically smaller than the code of p2. Instead of shallow cutoffs, QuickGame makes what may be called *generalized shallow cutoffs* ([Bhattacharya and Bagchi, 1992]). These are just like ordinary shallow cutoffs, except that a node q2 can get cutoff either by a node p1 with a code that is lexicographically smaller or by a node p3 with a code that is lexicographically greater. This added generality is a consequence of the use of local OPENs. It can be shown that for any game tree, all shallow cutoffs made by Alpha-Beta are also made by QuickGame. QuickGame can also influence deep cutoffs. But the number of nodes undergoing deep cutoff in QuickGame is much less than the corresponding number in Alpha-Beta.

In Table 1, the 10th terminal from the left has undergone generalized shallow cutoff while the 6th and the 8th terminals have undergone deep cutoffs in QuickGame. In Table 2, the two terminal nodes evaluated by QuickGame but not by Alpha-Beta have undergone deep cutoffs in Alpha-Beta. In both tables, all shallow cutoffs made by Alpha-Beta have also been made by QuickGame.

3. Introducing Revisits in QuickGame

Empirical results demonstrate ([Bhattacharya and Bagchi, 1992]) that QuickGame, on an average, evaluates fewer terminals than Alpha-Beta when run on random trees. But when game trees are strongly ordered, Alpha-Beta is a much better choice over QuickGame. In actual games, the trees generated very often tend to be strongly ordered, and hence Alpha-Beta will be preferred. Is there any way of improving upon QuickGame so as to dominate over Alpha-Beta even in case of strongly ordered trees ?

The dominance of Alpha-Beta over QuickGame in case of strongly ordered game trees can be explained in terms of deep cutoffs. The more strongly ordered the game tree T is, the higher is the probability that the optimal solution tree lies towards the left of T. When run on T, Alpha-Beta in its strict left-to-right scan of the game tree will encounter the optimal solution tree very quickly, examine all the terminals of it, and the value of the optimal solution tree thus obtained will influence deep cutoffs while scanning the rest of the game tree. Stronger the ordering of the game tree, more pronounced is the effect of deep cutoff while running Alpha-Beta on it. QuickGame, on the other hand, tries to equitably distribute its search effort among disjoint sets of solution trees, and, in the process, delays complete evaluation of any particular solution tree. As a result, when QuickGame is being called on a non-terminal MAX node p, the alpha bound passed as a parameter is a very poor lower bound on the minimax value of the node p, and hence fewer deep cutoffs occur below p.

In short, to make QuickGame competitive to Alpha-Beta for strongly ordered game trees, we have to improve upon its capability to influence deep cutoffs and this can be achieved only by providing improved lower bound while calling QuickGame on a non-terminal MAX node. Let p be a non-terminal MAX node in a (2, d) uniform game tree T and let QuickGame be called on p with alpha bound, lb and beta bound, ub. Grandsons p.0.0, and p. 1.0 will enter OPEN each with an h-value of ub. QuickGame will be recursively called on p.0.0 with bounds lb and ub, and let the call return the minimax value $t_{p.0.0}$, $lb < t_{p.0.0} < ub$ which now becomes the b-value of the replacing node p.0.1. The next node to be selected from OPEN will be p. 1.0 and QuickGame (p. 1.0, lb, ub) will be called. It may be noted that when running Alpha-Beta on the subtree rooted at p, node p. 1.0 will be called with lower bound, $t_{p.0}$ and upper bound, ub. What happens in QuickGame if, instead of lb, we pass h(p.0.1) = $t_{p.0.0}$ as the alpha bound while calling on p. 1.0 ? Clearly, since p.0 is a MIN node, $t_{p.0.0} \geq t_{p.0}$. Hence deep cutoffs below the node p.1.0 will be even more pronounced than the deep cutoffs influenced by Alpha-Beta when called on the same node. QuickGame (p.1.0, h(p.0.1), ub) will continue its search until either the minimax value of p.1.0 has been found out in which case $t_{p.1.0} > h(p.0.1) = t_{p.0.0}$, and the value returned will be minimum $(ub, t_{p.1.0})$; or at some point of time during execution the largest h-value

selected from OPEN is less than lb = h(p.0.1) = $t_{p.0.0}$. Note that in the latter situation node p.1.0 may not get solved, and the value returned by the call is a tighter upper bound on the minimax value of p.1.0. Problem arises if the game tree is such that $t_{p.0.0} > t_{p.1.0} > t_{p.0}$. The subtree rooted at p.1.0 has to be revisited at some later instant of time to find out its exact minimax value.

Extending the above argument, it is easy to see that *in QuickGame, while making a call on a node p, if the lower bound passed as parameter is the second largest h-value from amongst the nodes in local OPEN which are to the left of p in the game tree, then no distinct node pruned by Alpha-Beta will be examined by QuickGame.* When no such second largest h-value is available (for example, when p is a leftmost grandson) the current lower bound of the grandfather of p has to be used.

But when lower bound is chosen in this manner, the gap between the lower and upper bounds tend to become very narrow; and hence substantial amount of revisit may have to be done. As a consequence, the *total* number of terminals examined, including revisits, may become very high.

In our experiments, we have considered three different ways of determining the lower bound to be passed in a call on a non-terminal MAX node p, say, having the highest h-value in the local OPEN.

QG1 : the lower bound passed is the minimum of the h-values of the nodes in OPEN which are lexicographically smaller than p, i.e., to the left of p.

QG2 : the lower bound passed is the minimum of the h-values considering all the nodes in the local OPEN.

QG3 : the lower bound passed is the second highest h-value to the left of p in the local OPEN.

Note that the claim about not examining a node pruned by Alpha-Beta is not valid in QG1 and QG2.

We describe below the resulting algorithm QG which, depending upon how the lower bound has been chosen (the function *SelectLowerBound* in the algorithm), gives rise to QG1, QG2 or QG3. In QG, every node x in OPEN has an additional field *status* having a value of either SOLVED or LIVE. Node x initially enters OPEN with status(x) = LIVE. On examining a terminal, QG sets the corresponding status as SOLVED. A call on a non-terminal MAX node p returns the status of the highest h-valued node in the local OPEN as the status of p. Hence QG returns a pair of values instead of a single value returned by QuickGame. On return, node p will be replaced by its immediate right brother, if any, only if status(p) is SOLVED.

```
Procedure QG(p : node; alpha, beta : real);
var x : node;
    i, j, k : integer;
    lb, r, h[0..b-1] : real;
    OPEN[0..b-1] : array of node;
    status[0..b-1] : (LIVE, SOLVED);
begin
    if p is terminal then return (minimum(beta, v(p)), SOLVED);
    for (each child p.i of p) do begin
        h[i] := beta; status[i] := LIVE;
        if (p.i is terminal) then OPEN[i] := p.i
        else OPEN[i] := p.i.0;
    end;
    lb := alpha; r := alpha; k := 0;                    (* Remark 1 *)
    while (h[k] > r) do begin                  (* OPEN[k] = x.j, say *)
        update h[k], status[k] with QG(OPEN[k], lb, h[k]);
        if (status[k] = SOLVED) then                    (* Remark 2 *)
            if (OPEN[k] is a terminal MIN node)
              or (OPEN[k] has no brother to its right) then
                r := maximum(r, h[k])
            else
              begin OPEN[k] := x.j+1; status[k] := LIVE; end;
        k := index in OPEN corresponding to node with
            maximum h-value;                            (* Remark 3 *)
        lb := SelectLowerBound(OPEN);                   (* Remark 4 *)
        if (lb = h[k]) then lb := r;
    end;
    return (h[k],status[k]);
end;
```

Table 1 : Evaluation of terminals by the algorithms when run on a uniform (2, 4) tree

Terminal scores	5	3	3	4	1	0	1	2	2	7	5	4	1	4	6	2
QuickGame	1	3	2	-	10	-	11	-	4	-	5	6	7	-	8	9
Alpha-Beta	1	2	3	-	4	5	6	7	8	9	10	11	12	-	13	14
QG1,QG2,QG3	1	3	2	-	10	-	11	-	4	-	5	6	7	-	8	9

Table 2 : Evaluation of terminals by the algorithms when run on another uniform (2, 4) tree

Terminal scores	5	8	1	6	9	7	5	3	2	6	3	1	8	6	9	9
QuickGame	1	3	2	-	8	9	-	-	4	7	5	6	-	-	-	-
Alpha-Beta	1	2	3	-	4	5	-	-	6	-	7	-	-	-	-	-
QG1,QG2,QG3	1	3	2	-	6	7	-	-	4	-	5	-	-	-	-	-

Remark 1 : lb, at any instant during the execution of the algorithm, holds the value of the lower bound to be passed in the next call. Initially, since all h-values in OPEN are equal, viz., beta, lb is set to alpha, the lower bound on p.
Remark 2 : If OPEN[k] has not been solved by the call, the node is left in OPEN as it is with its h-value < lb defining a tighter upper bound on the minimax value of OPEN[k].
Remark 3 : Tie in the selection of the highest-valued node is resolved in favour of node having lexicographically smaller code.
Remark 4 : The function SelectLowerBound determines the lower bound according to one of the strategies described above. Depending on the strategy used we get algorithm QG1 or QG2 or QG3.
Example 3 : Let us run QG1, QG2 and QG3 on the (2, 4) game trees of Examples 1 and 2 in that order (Tables 1 and 2 respectively). It may be noted here that when the branching degree is 2, most of the times all the three strategies - lowest to the left, lowest among all and second highest - will find the same lower bound. In Table 1, all the versions of QG examine the same sequence of terminals as QuickGame. In Table 2, all the versions of QG examine the same set of terminals as Alpha-Beta but in a different order.

In the (3, 4) uniform tree of [Knuth and Moore, 1975], both OG1 and QG2 examine 32 terminals and QG3 examines 30 terminals compared to 31 terminals and 33 terminals examined by Alpha-Beta and QuickGame respectively.

It may be noted that the storage requirement of each of these algorithms is identical to that of QuickGame, i,e., b * d/2 units for a uniform (b, d) tree.

4. Experimental Results

The performances of the algorithms Alpha-Beta, QuickGame, QG1, QG2 and QG3 have been compared experimentally on a 486-based Vectra 33VL machine. For each (b, d) pair, 100 random uniform trees and 100 strongly ordered uniform game trees were generated. A *node ordering* tells us, for any node x.i in a game tree, the probability that x.i determines the minimax value of x. In a *random* ordering, all x.i's have equal probabilities. The strongly ordered trees were generated following the suggestion of Richard Korf [1993]. This technique of generating game trees first appeared in [Fuller et al., 1973]. The scheme is as follows : Assign random independent values to the edges of the tree. The static value of a node is computed as the sum of the edge costs from the root to that node. Fully expand each node, and then order the children of the node by their static values - increasing order for the children of MIN nodes, and decreasing order for the children of MAX nodes. "It makes no sense to fully expand nodes one level above the terminal nodes". Following Korf s suggestion, the strongly ordered trees have been generated by doing full expansion with sorting all the way down to one level above the terminal nodes. To allow comparisons of different algorithms on the same tree, whenever we needed a random value to be assigned to an edge, we have made use of the static value of the parent node and the tree number (between 1 and 100) to determine the seed for initializing the random number generator.

The ordering of the trees thus generated have been found to be marginally stronger than the strongly ordered trees of Marsland et al. [1987]. For example, when run on one hundred (10,6) strongly ordered uniform trees generated using both the methods (Korf s and Marsland's), the average number of terminals examined by Alpha-Beta is 3327 on Korf trees and 3453 on Marsland trees. Note that the number of terminals examined is 0.33 percent of the total number of terminals in the case of Korf trees.

Table 3 compares the number of terminals evaluated and the time taken in seconds by each of the five algorithms. For each (b, d) pair, the number of terminals examined by an algorithm was averaged over 100 uniform trees and rounded off to the nearest integer. For QG1, QG2 and QG3, the terminal count shown is the *total* number of terminals examined, including revisits. The second column of the table gives the value of $b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$, the minimum number of terminals that any minimax algorithm has to examine.

It may be noted that in our implementation, no time is spent in 'evaluating' a terminal and substantial amount of time is spent in generating the tree in a pre-specified order. This, in turn, implies that in our experiments if an algorithm A examines less number of terminals than some other algorithm B in comparable time, one can expect algorithm A to run faster than algorithm B in actual game playing situations. This is because in actual games, terminal evaluation takes significant time and the tree is quite naturally generated in a strongly ordered manner.

On the basis of the experimental results given in Table 3, the following observations are in order:

a) On random trees, the performance of QuickGame is the best among all the algorithms in terms of average number of terminals examined, execution time, and number of times it examines fewer terminals than Alpha-Beta. In other words, *in actual games, QuickGame is expected to run faster than Alpha-Beta if the tree generated is known to be random.* On strong trees, the performance of QuickGame is worst among the algorithms.
b) On strongly ordered trees, the average *total* number of terminals (including revisits) examined by the algorithm QG3 is consistently less than that of Alpha-Beta. Recall that in QG3 the lower bound passed is the second highest h-value to the left of the selected node in the local OPEN. As expected, on random trees, performance of QG3 is worst among QuickGame and its variations due to too many revisits.
c) The average execution time of QG3, when run on strongly ordered trees, is marginally worse than that of Alpha-Beta. It may be noted that, since our implementation requires no time to evaluate a terminal, the timings reflect only the overheads of the algorithms. Thus the

Table 3 : Comparison of the algorithms Alpha-Beta, QuickGame, QG1, QG2 and QG3

(b, d)	minm. no. of terminals to be examined	Order	Alpha-Beta		QuickGame			QG1			QG2			QG3		
			terminal count	time taken	terminal count	worse* count	time taken	terminal count	worse* count	time taken	terminal count	worse* count	time taken	terminal count	worse* count	time taken
6,6	431	random	5947	0.18	4474	4	0.16	4706	15	0.18	4833	18	0.19	5595	43	0.21
		strong	767	0.02	1099	93	0.04	743	56	0.03	763	57	0.03	691	38	0.03
6,7	1511	random	22540	0.70	18010	5	0.63	19493	23	0.81	20042	32	0.84	22217	51	0.91
		strong	2219	0.06	2825	91	0.10	2284	71	0.09	2312	75	0.09	2092	32	0.08
6,8	2591	random	76029	2.77	61481	16	2.65	65924	26	2.94	66312	31	2.99	76416	51	3.38
		strong	4696	0.14	8705	98	0.30	4902	66	0.20	5101	68	0.21	4254	42	0.18
8,6	1023	random	22037	0.65	17259	1	0.66	19078	22	0.76	19263	23	0.77	25229	72	0.98
		strong	1733	0.05	2669	96	0.09	1675	55	0.07	1736	57	0.07	1555	39	0.07
9,6	1457	random	36880	1.10	28855	4	1.12	31573	21	1.28	32147	26	1.31	43365	77	1.71
		strong	2313	0.07	3845	99	0.14	2318	60	0.10	2404	63	0.10	2128	44	0.10
10,5	1099	random	11541	0.34	9431	3	0.32	10274	30	0.45	10450	30	0.47	13025	69	0.55
		strong	1426	0.04	1520	75	0.05	1382	50	0.05	1385	50	0.05	1315	15	0.05
10,6	1999	random	59090	1.77	44955	1	1.78	49972	16	2.08	50697	22	2.12	71651	71	2.89
		strong	3327	0.10	5459	100	0.19	3219	56	0.14	3342	57	0.15	3096	45	0.14
10,7	10999	random	314417	9.51	250219	3	8.68	275679	24	12.43	282393	31	12.75	368547	71	15.99
		strong	13916	0.39	17783	97	0.73	14911	80	0.61	14994	81	0.61	13381	34	0.53

* Number of trees, out of the 100 trees, where this algorithm has examined more number of terminals than Alpha-Beta.

overhead of QG3 is marginally higher than that of Alpha-Beta.

d) As argued in section 3, no node pruned by Alpha-Beta will be examined by QG3, although QG3 may need to revisit some of the nodes. In actual games, the time for evaluating a terminal during revisit can be saved by keeping track of the already evaluated terminals in a hashed table. Hence, compared to Alpha-Beta, the total time that would be saved in terminal evaluation by QG3, when embedded in game playing programs, is expected to be more than what is reflected by the difference between the average number of terminals examined by the two algorithms.

e) Considering the observations in (b), (c) and (d) above, it is reasonable to expect that in actual games where (i) no special effort is required to generate the tree in strongly ordered manner and (ii) evaluation of a terminal takes substantial time, QG3 will run faster than Alpha-Beta.

f) Between QG1 and QG2, performance of QG1 (lower bound passed is the lowest h-value to the left of the selected node) is consistently better than that of QG2 (lower bound passed is the lowest among all h-values in local OPEN). On random trees, both QG1 and QG2 perform worse than QuickGamc but better than both Alpha-Beta and QG3. On strongly ordered trees, both QG1 and QG2 perform worse than QG3 and better than QuickGamc We conjecture that for some ordering between random and strong, these algorithms will perform better than both QuickGame and QG3.

5. Conclusion

In this paper different strategies for allowing revisits in game tree search have been discussed. It has been shown that on random trees, the algorithm QuickGame, which does not allow revisit of a node, outperforms Alpha-Beta in comparable time and memory. On strongly ordered trees, QG3, which is an extension over QuickGame and allows revisits in a specific manner, is a close competitor of Alpha-Beta. The other strategies discussed may prove useful for orderings in between random and strong.

We expect that the algorithms discussed will stimulate further research in using revisit of nodes effectively in game tree search.

References

[Bhattacharya and Bagchi, 1994] Subir Bhattacharya and A. Bagchi, A General framework for minimax search in game trees, *Information Processing letters,* vol 52, 1994, pp 295-301.

[Bhattacharya and Bagchi, 1993] Subir Bhattacharya and A. Bagchi, A Faster alternative to SSS* with extension to variable memory, *Information Processing letters,* vol 47, 1993, pp 209-214.

[Bhattacharya and Bagchi, 1992] Subir Bhattacharya and A. Bagchi, QuickGame : A compromise between pure depth-first and pure best-first game tree search strategies, *Proc. International Workshop on Automated Reasoning,* IWAR'92, Beijing, China, July 13-16,1992, pp 211-220.

[Fuller et al., 1973] S. H. Fuller, J. G. Gaschnig and J. J. Gillogly, An analysis of the Alpha-Beta pruning algorithm, Department of Computer Science Technical Report, Carnegie-Mellon University, Pittsburgh, Pa, 1973.

[Knuth and Moore, 1975] Donald £. Knuth and Ronald W. Moore, An analysis of Alpha-Beta pruning, *Artificial Intelligence,* vol 6, 1975, pp 293-326.

[Korf, 1993] Richard Korf; 1993, Personal communication to A. Bagchi.

[Marsland et al., 1987] T. A. Marsland, Alexander Reinefeld and Jonathan Schaeffer, Low overhead alternatives to SSS*, *Artificial Intelligence,* vol 31, 1987, pp 185-199.

[Pearl, 1984] J. Pearl, *Heuristics : Intelligent Search Strategies for Computer Problem Solving,* Addison-Wesley, Reading, Massachusetts, U.S.A., 1984.

[Reinefeld and Ridinger, 1994] Alexander Reinefeld and Peter Ridinger, Time-efficient state space search, *Artificial Intelligence,* vol 71, 1994, pp 397-408.

[Stockman, 1979] G. C. Stockman, A minimax algorithm better than Alpha-Beta ?, *Artificial Intelligence,* vol 12, 1979, pp 179-196.