A SURVEY OF ENHANCEMENTS TO THE ALPHA-BETA ALGORITHM

T.A. Marsland and M. Campbell

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

ABSTRACT

Current game-playing programs have developed numerous move ordering and search reduction techniques in order to improve the effectiveness of the alpha-beta algorithm. A critical review of these search modifications is provided, and a recursive formula to estimate the search time is proposed, one which reflects the characteristics of the strongly ordered trees produced through use of improved search enhancements.

I. THE ALPHA-BETA ALGORITHM

With few exceptions [NEWB], much of the existing theoretical work on sequential game tree searching has been restricted to random trees. However, in practice, truly random trees are quite uncommon. In addition, special techniques have been developed to improve the effectiveness of the principal searching method, the alpha-beta algorithm. Thus, we will assess these enhancements and show why strongly ordered trees are more realistic, and possess properties that can be exploited

A complete description of the alpha-beta algorithm can be found elsewhere [KNUT]. Rather than duplicate that work we will simply clarify some relevant facts and terminology used in our paper. A typical procedure heading might be alphabeta(p, alpha, beta, depth), where p represents a position, (alpha,beta) the <u>search window</u> or range of values over which the search is to be made, and depth the intended length of the search path. The basic structure of the depth-limited alpha-beta algorithm can be seen in the following procedure.

```
alphabeta(p, alpha, beta, depth)
 position p;
 int alpha, beta, depth;
{
  int w, m, i, t;

  if (depth < 0) return(evaluate(p));
  w = generate(p);
    /* determine successor positions */
    /* p.1 ... p.w and return number */
    /* of moves as function value    */
  if (w == 0) /* no legal moves */
    return(evaluate(p));
  m = alpha;
  for i = 1 to w do
  {  t = -alphabeta(p.i,-beta,-m,depth-1);
     if (t > m) m = t;
     if (m >= beta)  /* cutoff */
       return(m);
  }
  return(m);
}
```

For purposes of analysis, it is convenient to study the performance of the minimax and alpha-beta algorithms on uniform trees of depth D and constant width W. It is also usual to measure the relative efficiency of tree-searching algorithms in terms of the number of terminal nodes scored. The minimax algorithm will always examine M(W,D) = W**D terminal nodes, while under ideal conditions the alpha-beta algorithm, under ideal conditions, scores only

$$B(W,D) = W^{**}\lceil D/2\rceil + W^{**}\lfloor D/2\rfloor - 1 \quad \text{nodes.}$$

Thus the potential efficiency of the alpha-beta algorithm is very good, examining close to the square root of the maximum number of nodes while still generating the same solution path (principal variation) from the root node. However, optimal performance is achieved only when the first move considered at each node is the best one. Under more realistic assumptions, we can define the following quantities.

	D=3	B	A	R	M
W	8	71	105 (21)	181 (36)	512
	16	271	405 (64)	786 (114)	4096
	24	599	857 (115)	1752 (250)	13824

	D=4	B	A	R	M
W	8	127	281 (88)	690 (153)	4096
	16	511	1286 (430)	4125 (875)	65536
	24	1151	2946 (1013)	10425 (1891)	331776

Table 1: Expected search costs for trees of width W and depth D.

R(W,D) = average number of terminal nodes scored in a random uniform game tree
A(W,D) = average number of terminal nodes scored in a strongly ordered uniform game tree

For the purposes of this paper, we will define a tree to be strongly ordered if the search finds (1) the first branch from each node best 70% of the time, and (2) the best move in the first 25% of the branches 90% of the time. Of course, this definition is totally arbitrary, but it is meant to produce trees similar in character to those generated by contemporary chess programs using search enhancements.

While the performance of alpha-beta on random trees has a solid theoretical basis [FULL], at present only empirical evidence is available for strongly ordered trees. Nevertheless, on a statistical basis, it seems clear that we have the relation B(W,D) < A(W,D) << R(W,D) << M(W,D) = W**D Relative values for these terms can be seen from our Monte Carlo simulation results, presented in Table 1. The simulations were carried out on trees of depths up to 5 and width W, with scores in the range 0 - 127. To estimate R, the values were assigned randomly to the terminal nodes, while the calculation of A relied on branch-dependent scores. The bracketed numbers represent the standard deviation for 100 independent search trials. Table 1 will be used later to support a proposed formula which estimates A(W,D).

II. ENHANCEMENTS TO ALPHA-BETA SEARCHING

Many of the following techniques have been developed in efficiency-conscious full-width chess programs. The basic methods, however, are applicable to most programs that use the alpha-beta algorithm.

A. Aspiration search: The interval enclosed by (alpha, beta) is referred to as the alpha-beta window. For the alpha-beta algorithm to be effective, the minimax score of the root position must lie within the initial window. Generally speaking, however, the narrower the initial window, the better the algorithm's performance. In many problem domains such as chess, there are reliable methods to estimate the score that will be returned by the search. Thus, instead of using an initial window of (-INF, +INF) (where INF is a number larger than evaluate() will return), one can use (V-e,V+e), where V is the estimated score, and e the expected error. There are three possible outcomes of this so-called aspiration search, depending on S, the actual (minimax) score of a position p.

1. if S <= V-e,
   alphabeta(p,V-e,V+e,D) <= V-e
2. if S >= V+e,
   alphabeta(p,V-e,V+e,D) >= V+e
3. if V-e < S < V+e,
   alphabeta(p,V-e,V+e,D) = S

Cases 1 and 2 are referred to as failing low and failing high respectively [FISH]. Only in case 3 is the true score of the position p found, using a smaller search space -- bounded by B(W,D) and A(W,D).

In the failed low case, it is necessary for the search to show that each alternative from the root is less than V-e. Assuming perfect ordering,

$$W ** \lceil D/2 \rceil \text{ nodes must be examined.}$$

In the failed high case, it is sufficient for the search to show one alternative greater than V+e. Again under perfect ordering conditions, only

$$W ** \lfloor D/2 \rfloor \text{ nodes need be examined.}$$

Either way the search must be repeated, for example alphabeta(p,V+e,+INF,D) for the failed high case. Empirical evidence has shown aspiration searches to be very effective; in TECH[7], search time reductions averaging 23% were noted [GIL2]. This result is confirmed by Baudet by adapting his results for parallel tree search to the sequential case [BAUD].

B. Transposition Table: In carrying out a search of a chess game tree, it is not uncommon for positions to recur in numerous places throughout the tree. Rather than rebuild the subtrees associated with the transposed positions, it may be possible to simply retrieve the results stored in a table by a previous search. A transposition table is a large hash table, with each entry representing a position. For game modelling nearly perfect hashing functions can be produced[ZOBR]. Although there are many table management problems which must be solved, the technique has very low overhead for the large potential gains.

A typical hash index generation method is the one proposed by Zobrist[ZOBR], who observed that a chess position constitutes placement of up to 12 different piece types {K,Q,R,B,N,P,-K ... -P} onto a 64-square board. Thus a set of 12x64 integers (plus a few for enpassant and castling privileges), {Ri}, may be used to represent all the possible piece/square combinations. An index of the position may be given by

$$Pj = Ra \ \underline{xor} \ Rb \ \underline{xor} \ \ldots \ \underline{xor} \ Rw$$

where the Ra etc. are integers associated with the piece placements for the particular position under consideration. Movement of a piece from a square associated with Rf to the piece/square associated with Rt yields a new index

$$Pk = (Pj \ \underline{xor} \ Rf) \ \underline{xor} \ Rt$$

More importantly, if the Ri are uniformly distributed in the interval [0,2**N], then so are the Pk. Typically N is 32 and so 2**N is too large for direct use of Pk as an index into a transposition table, rather

Hk = Pk mod T   is used, where T << 2**N.

Clearly, all the possible chess positions cannot be represented uniquely by Hk, but even so this is quite sufficient as a basis for a successful entry point. A minimal table entry could have the following format:

lock	move	score	flag	len	prio

lock	to ensure the table position is identical to the tree position,
move	best move in the position, determined from previous search,
score	of subtree computed previously,
flag	indicating whether score is upper bound, lower bound or true score,
len	length of subtree that score is based on,
prio	used in table management, to select entries for deletion.

When a position reached during a search is located in the table (i.e. the lock matches), there are a number of possible actions:

(1) If len is less than remaining length to be searched, score is ignored and the search is carried out as usual. However move is tried first in the position. The main advantage of this is that it saves a move generation, and also, since move has previously (in a shallower search) proven best, it is likely to be so again. Furthermore, move will direct the search toward positions that have been seen before, hence increasing the effectiveness of the table.

(2) if len >= remaining length to be searched
   (a) if score was the true score, this value is returned without further searching
   (b) otherwise, score is used to adjust the current alpha-beta bounds. This could either cause an immediate cutoff, or allow the search to continue with a reduced window. If a search must be done, move will be tried first.

There are also further enhancements possible. For example, DUCHESS[5] maintains both upper and lower bounds on the position score, with separate lengths for each.

Transposition tables are most effective in chess endgames, where there are fewer pieces and more reversible moves. Gains of a factor of 5 or more are typical, and in certain types of King and pawn endings, experiments with BLITZ[3] and BELLE[2] have produced trees of more than 30 ply, representing speedups of well over a hundred-fold. Even in complex middlegames, however, significant performance improvement is observed. An implementation of alpha-beta employing a transposition table is presented in the Appendix.

C. Killer Heuristics: The killer heuristic is based on the premise that if move My 'refutes' move Mx, it is more likely that My (the 'Killer') will be effective in other positions. Any move which causes a cutoff at level N is said to have refuted the move at level N-1. There are many ways of using this information. For example, the program CHESS[4] maintains a short list of killers at each level in the tree, and attempts to apply them early in the search in the hope of producing a quick cutoff. A further advantage of the killer heuristic is that it tends to increase the usefulness of the transposition table. By continually suggesting the same moves, there is a greater possibility of reaching a position already in the table.

In its full generality, the killer

heuristic can be used to dynamically reorder moves as the search progresses. For example, if a move My at level N refutes a move at level N-1, and My remains to be searched at level N-2, it is worth considering next. An additional method, used by AWIT[1], seeks out defensive moves at ply N-1 which counteract killers from level N. The idea behind the generalized killer heuristic mechanism is to allow information gathered deep in the tree to be redistributed to shallower levels. This is not usually done by the full-width programs, however, since it is not yet clear that the potential gains exceed the overhead.

The actual search reductions produced by the killer heuristic are not clear. In TECH[7], no improvement was noted, but CHESS[4], DUCHESS[5] and BLITZ[3] continue to employ the mechanism.

D. Iterative Deepening: Iterative deepening refers to the procedure of using an N-1 ply search to prepare for an N ply search. The cost of such a search is given by an equation of the form

A(W,D) = A(W,D-1) + E(W,D),

where E(W,D) is the expected cost of an alpha-beta search given the first D-1 moves of the principal variation. This technique has certain immediately obvious advantages.

(1) It can be used as a method for controlling the time spent in a search. In the simplest case, new iterations can be tried until a preset time threshold is passed.
(2) An N-1 ply search can provide a principal continuation which, with high probability, contains a prefix of the N ply principal continuation. This allows the alpha-beta search to proceed more quickly.
(3) The score returned from a N-1 ply search can be used as the center of an alpha-beta window for the N ply search. It is probable that this window will contain the N ply score, thus increasing search speed.

These last two points, though significant, are not really complete justifications for the use of iterative deepening from a tree searching point of view. In fact, in experiments with checkers game trees [FISH], it was found that iterative deepening increased the number of nodes searched by 20% (apparently only using point (2), however). In addition, studies with TECH[7] using a generalized version of (2), but not (3), noted a 5% increase in search times when iterative deepening was applied [GIL2]. It appears that a strong initial move ordering, together with a good alpha-beta window estimate, can approximately match iterative deepening. The real searching advantage of iterative deepening is:

(4) The transposition table and killer lists are filled with useful values and moves.

The importance of this fact is illustrated by the performance of the BELLE[2] chess machine. Typical chess middlegame positions have branching factors of 35-40. It has been found that in such positions, it normally costs BELLE a factor of 5 - 6 to go one further ply, i.e. less than the expected cost of optimal alpha-beta.

A variation of this basic scheme, one which is especially appropriate if transposition tables are not used, is employed by L'EXCENTRIQUE[6]. A 2 or 4-ply minimax search is first performed to obtain W move-pairs (moves and their best refutation). These are then sorted and a 6, 8, 10 etc -ply iterative deepening cycle initiated. The rationale behind two ply increments is to preserve a consistent theme between iterations, so that the principal variation will not flip-flop between attacking and defensive lines. To our knowledge, no analytical comparison between this and conventional iterative deepening has been done.

However, we do hypothesize that the incremental cost is of the form

E(W,D) = B(W,D) + (W-1)*F(W-1,D-2)

A study of Table 1 leads us to refine the function F to fit the available data and to propose that the recurrence relation

A(W,D) =:= A(W,D-1) + B(W,D)
+ (W-1)*B(W-1,D-2)

be valid for trees of the type searched by chess programs, using iterative deepening in conjunction with transposition tables. From the above equation, and the data in Table 1, the estimated value for A(24,4) is 3066 while the experimental value was 2946. Similarly, the value for A(24,5) from the recurrence relationship is 30018 and the experimental value from fifty Monte Carlo trials was about 28500. For typical values of W and D, (W-1)*B(W-1,D-2) is approximately equal to B(W,D) and A(W,D-1) is small in comparison. Hence we may say that

A(W,D) =:= 2*B(W,D)

for strongly ordered trees with W > 20 and D > 4.

III. MODIFICATIONS TO THE ALPHA-BETA ALGORITHM

A number of modifications to the alpha-beta algorithm have been proposed [FISH]. They are examined here mainly for compatibility with the other search enhancements discussed.

Falphabeta, for 'fail-soft alphabeta', is useful when aspiration searching is employed. Though always examining the same nodes as alpha-beta, falphabeta can give a tighter bound on the true score of the tree when the search fails high or low. Although falphabeta requires a slight constant overhead, any system which uses aspiration searches should find the technique a practical one. The concept of a minimal window, an alpha-beta window of (-m-1,-m) where m is the best score so far, was introduced and used to search the last subtree [FISH]. Slight searching improvement was noted for no cost.

Palphabeta is an interesting modification of alpha-beta which operates only on nodes along the principal variation. Once a candidate principal variation is obtained, the balance of the tree is searched with a minimal window. However, each subtree that is better than its elder siblings must be searched twice, if the tree is poorly ordered. Hence there is some risk that palphabeta will examine more nodes than alpha-beta. Iterative deepening provides a principal variation with reasonable reliability, and makes this technique more feasible. The following code has been adapted from [FISH].

```
palphabeta(position p, int depth)
{
  int w, m, i, t;
  if (depth < 0) return(evaluate(p));
  w = generate(p);
  if (w == 0) return(evaluate(p));
  m = -palphabeta(p.1, depth-1);
  for i = 2 to w do
  {  t = -falphabeta(p.i,-m-1,-m,depth-1);
     if (t > m)
        m = -alphabeta(p.i,-INF,-t,depth-1);
  }
  return(m);
}
```

It could also be pointed out that it is not necessary to carry palphabeta all the way to the terminal nodes. In fact, since only the first few moves of a principal continuation are usually reliable, carrying palphabeta to, say, N-2 ply on an N ply iteration may be sufficient. Another objection could be made on the grounds that, for programs employing transposition tables, the values that will be stored in the table are rather loose bounds, and hence less likely to cause later cutoffs. The effects of this are not clear.

SCOUT [PEAR] is a further generalization of palphabeta, in which the call to alphabeta is replaced by

```
          m = -palphabeta(p.i, depth-1);
```

In its original form, SCOUT does not use the minimal window idea, but rather an equivalent test procedure. Our initial simulation results indicate that palphabeta out-performs SCOUT on strongly ordered trees.

IV. CONCLUSIONS

A number of techniques for improving the searching performance of the alpha-beta algorithm have been discussed. The experiences of current game playing programs have demonstrated the effectiveness of aspiration searches, transposition tables, the killer heuristic and iterative deepening. Modifications like palphabeta and SCOUT deserve further attention in programs that search strongly ordered trees, particularly to determine their interaction with the other searching enhancements. There is also growing interest in parallel implementations of alpha-beta [BAUD], [MARS], and it is important that these parallel methods retain the advantages obtained in the sequential case.

REFERENCES

[BAUD] G. Baudet, "The design and analysis of algorithms for asynchronous multiprocessors", Ph.D. thesis, Computer Science Dept., Carnegie-Mellon Univ. Pittsburgh, (1978).

[FISH] J. Fishburn and R. Finkel, "Parallel alpha-beta search on Arachne", TR 394, Computer Science Dept., Univ. Wisconsin, Madison, (1980).

[FULL] S. Fuller, J. Gaschnig and J. Gillogly, "Analysis of the alpha-beta pruning algorithm", Computer Science Dept., Carnegie-Mellon University, Pittsburgh, (1973)

[GIL1] J. Gillogly, "The technology chess program", Artificial Intelligence 3, 145-163, (1972).

[GIL2] J. Gillogly, "Performance analysis of the Technology chess program", Ph.D. thesis, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, (1978).

[KNUT] D. Knuth and R. Moore, "An analysis of alpha-beta pruning", Artificial Intelligence 6, 293-326, (1975).

[MARS] T.A. Marsland, M.S. Campbell and A.L. Rivera, "Parallel search of game trees", TR 80-7, Computing Science Dept., Univ. of Alberta, Edmonton, (1980).

[NEWB] M. Newborn, "The efficiency of the alpha-beta search in trees with branch dependent terminal node scores", Artificial Intelligence, Vol 8, #2, 137-153, (1977).

[PEAR] J. Pearl, "Asymptotic properties of minimax trees and game-searching procedures", Artificial Intelligence 14, 113-138, (1980).

[ZOBR] A.L. Zobrist, "A hashing method with applications for game playing", TR 88, Computer Science Dept., Univ. of Wisconsin, Madison (1970).

CHESS PROGRAMS REFERENCED

1. AWIT - T.A. Marsland; University of Alberta
2. BELLE - K. Thompson, J. Condon; Bell Telephone Laboratories
3. BLITZ - R. Hyatt, A. Gower; Univ. of Southern Mississippi
4. CHESS - D. Slate, L. Atkin; Northwestern University
5. DUCHESS - T. Truscott, B. Wright, E. Jensen; Duke University
6. L'EXCENTRIQUE - C. Jarry; McGill University
7. TECH - J. Gillogly; Carnegie-Mellon University

```
AB(position p, int alpha, int beta, int depth)
{   int i, t, w, type, score, flag;
    position p.opt;
    type = retrieve(p, depth, score, flag, p.opt);
      /* type <  0 - position not in table
         type == 0 - position in table, but length < depth
         type >  0 - position in table, length >= depth
      */
    if (type > 0)
    {   if (flag == VALID) goto done;
        if (flag == LBOUND)
            alpha = max(alpha, score);
        else /* flag == UBOUND */
            beta = min(beta, score);
        if (score >= beta)  goto done;
    }
      /* Note beneficial update of alpha or beta
         bound assumes full width search.
         Score in table insufficient to terminate search
         so continue as usual, but try p.opt (from table)
         before generating other moves, if p is non-terminal.
      */
    score = alpha;
    if  ((type >= 0) and (p.opt != NULL))
    {   t = -AB(p.opt, -beta, -score, depth-1);
        if (t > score) score = t;
        if (score >= beta)  goto done;
    }
      /* no cutoff. Generate moves, put p.opt first.
      */
    w = generate(p);
    if (w == 0)             /* mate or stalemate */
    {   p.opt = NULL;
        score = evaluate(p);
        goto done;
    }
    for i = 2 to w do
    {
       if (depth == 0)
         t = evaluate(p.i);
       else
          t = -AB(p.i, -beta, -score, depth-1);
        if (t > score)
        {  score = t;
           p.opt = p.i;     /* note best successor */
           if (score >= beta) goto done;
        }
    }
done:
    flag = VALID;
    if (score <= alpha) flag = UBOUND;
    if (score >= beta) flag = LBOUND;
    store(p, depth, score, flag, p.opt);
    return(score);
}
```

Appendix: Alpha-beta implementation using transposition table