RELATIVE PERFORMANCE OF THE ALPHA-BETA ALGORITHM

T.A. Marsland
University of Alberta.

The alpha-beta algorithm implements a minimax search. Its high efficiency may be attributed to the use of two bounds which form an initial <u>window</u>. If this window covers the full range of numbers that the terminal node evaluation function can produce, then a <u>full window</u> search is being done. A call to the alpha-beta function could be:

```
V = AB(p, alpha, beta, depth);
```

where p is a pointer to a position state vector, alpha and beta are the lower and upper bounds on the window, and depth is the specified length of search. The number returned by the function is called the value of the tree, and measures the potential success of the player to move. A skeleton for this function, expressed in the C language with Pascal style declarations and loops, appears in Figure 1. The algorithm is expressed in a negamax framework [KNUT75], and so avoids the need for alternate min/max operations by always returning the negative of the subtree value from node to node. Undefined are functions evaluate(), to assess the value of the terminal nodes, generate(), to list the moves for the current position, make(), to actually play the move under consideration, and undo(), to retract the current move.

```
function AB(p : position; alpha, beta, depth : int) : int;
{
  VAR width, score, i, value : int;

  if (depth ≤ 0)                      /* a terminal node? */
     return(evaluate(p));
                            /* determine successor positions */
  width = generate(p);      /* p.1 ... p.w and return number */
                            /* of moves as function value    */
  if (width == 0)                     /* no legal moves? */
     return(evaluate(p));
  score = -INF;
  for i = 1 to width do {
     make(p.i);
     value = -AB(p.i, -beta, -max(score,alpha), depth-1);
     undo(p.i);

     if (value > score)               /* an improvement? */
        score = value;
     if (score ≥ beta)                /* a cutoff? */
        return(score);
  }
  return(score);
}
```

Figure 1: Depth-limited alpha-beta function.

A major extension to the alpha-beta algorithm involves iterative deepening, in which a sequence of successively deeper and deeper searches is carried out until some (time) limit is exceeded. Thus a search of depth D <u>ply</u> (moves) is used to <u>dynamically reorder</u> (sort) the choices and so prepare the way for a faster D+1 ply search than would be possible directly. Two further refinements are:

(a). <u>Aspiration search</u>, in which the width of the window is typically equal to two times the value of the smallest piece (a Pawn). It is possible for such a search to fail, i.e., to return a value which is outside the window. Two failure modes occur: 'low', in which all the moves are tried but no value reaches the lower limit of the window, and 'high', which stops the

search as soon as a move is found which exceeds the upper expectation. A sample implementation of an aspiration search is shown in Figure 2. Note that function AB() is never called more than twice for each iteration.

```
/*    Assume V = estimated value of position p, and
             e = expected error limit.
*/
V = 0;
for D = 1 to depth do {
    alpha = V - e;
    beta  = V + e;
    V = AB(p, alpha, beta, D);

    if (V ≥ beta)                      /* failing high */
       V = AB(p, V, +INF, D);
    else
    if (V ≤ alpha)                     /* failing low  */
       V = AB(p, -INF, V, D);

    sort(p);       /* best move so far is tried first
                      on next iteration. */
}
```

Figure 2: Iterative deepening with aspiration search.

(b). Minimal window search, in which it is assumed that the first move to be tried is the start of the principal variation. This line is then searched with a full width window, while all the alternate variations are searched with a zero width window, under the assumption that they will fail-low in any case. Should one of the moves not fail this way then it becomes the start of a new principal variation and the search is repeated for this move with a window which covers the new range of possible values*. The nature of principal variation search (PVS) is shown in Figure 3.

*: Ken Thompson employs an interesting variation of this in Belle [MARS83].

```
function PVS( p : position; depth : int) : int;
{
  VAR width, score, i, value : int;
  if (depth ≤ 0)
     return(evaluate(p));
  width = generate(p);
  if (width == 0)
     return(evaluate(p));
  make(p.1);
  score = -PVS(p.1, depth-1);
  undo(p.1);
  for i = 2 to width do {
     make(p.i);
     value = -AB(p.i, -score-1, -score, depth-1);
     if (value > score)
        score = -AB(p.i, -INF, -value, depth-1);
     undo(p.i);
  }
  return(score);
}
```

Figure 3: Minimal window search.

Both aspiration and minimal window searches can benefit further from the use of refutation and transposition tables. Installation of a refutation table is straightforward and has low space overhead. After a search of depth D on a tree of constant width W the table contains W*D entries. Thus for each variation the table contains the sequence of D moves which determined a sufficient value for that variation. Prior to the next iteration the table is sorted so that upon an iteration to depth D+1 there exists a D-ply sequence for each variation that is tried first [This is basically the scheme described by W. Fink, ICCA Newsletter, Vol 5, #1]. The candidate principal variation is fully searched, but for the alternate variations the moves in the refutation table may be sufficient to again cut off the search and thus save the move generation that would normally occur at each node. If the maximum length of

the refutation path is 5 and the maximum tree width is 100, then, if each entry needs 2 bytes, just 1000 bytes are required to hold all the refutation lines for the current position.

A transposition table may also be used to hold refutations but, because it has the capacity for including more information, it has other capabilities too. If the information stored in the entries contains at least the best move in the position and the value and length of the subtree emanating from that point, then the transposition table may be used to extend the effective search depth. This is especially valuable in endgames when the number of possible alternatives is small. As in the other cases, a sorting operation between each iteration ensures that the moves at the first level will be tried in the best possible order. A typical transposition table might contain 10,000 entries, each of 10 bytes [MARS83], for a 100,000 byte total storage overhead.

In comparing algorithms which search game trees, one may either measure the amount of computer time used to search a tree, or count the number of nodes visited in the tree. If the cost of a node is nearly constant, these two measures are effectively the same. However, our test program, and chess programs in general, perform significantly more calculation at a terminal node than at interior nodes in the tree. One reason for this is that a check or capture analysis in the form of an extended tree search is done. Therefore the following comparison is based on the number of terminal nodes examined, especially since it has the additional advantage of being a machine independent measure.

The algorithms were tested on a data set which was used to assess the performance of computer chess programs and human chess players [BRAT82]. That data set contained 24 chess positions [see ICCA Newsletter, Vol 4, #2]. The first position involves a forcing sequence of checks and so was deleted from this study. All the remaining positions were searched with 3, 4 and 5-ply trees, using a combination of alpha-beta refinements. The results are normalized to the cost of a direct (no iteration) alpha-beta search and are summarized in Figure 4. The following cases were considered: (1). Simple iteration, retain first move of principal variation only. (2). Narrow window aspiration search. (3). Full window search with refutation table. (4) Principal variation search with transposition table. A lower bound, in the form of the size of the expected minimal tree that alpha-beta must search, is also included.

The results show that use of a refutation or transposition table is a more important enhancement to iterative deepening than is aspiration searching. However, some form of aspiration searching always provides extra benefit. For trees of less than 6-ply in depth a refutation table is almost as effective as a transposition table, yet incurs a much lower storage overhead. While minimal window searching (PVS) is marginally better than narrow window searching, it involves a more complex implementation. A more complete comparison is to be found in a recent report [MARS82].

BRAT82 I. Bratko and D. Kopec, "A test for comparison of human and computer performance in chess" in Advances in Computer Chess 3, M.R.B. Clarke (editor), Pergamon Press, 1982.

KNUT75 D. Knuth and R. Moore, "An analysis of alpha-beta pruning", Artificial Intelligence 6 (1975), 293-326.

MARS82 T.A. Marsland, "A quantitative study of alpha-beta refinements", TR82-6, Computing Science Dept., Univ. of Alberta, EDMONTON, Canada.

MARS83 T.A. Marsland and M. Campbell, "Parallel search of strongly ordered game trees", Computing Surveys (to appear) 1983.

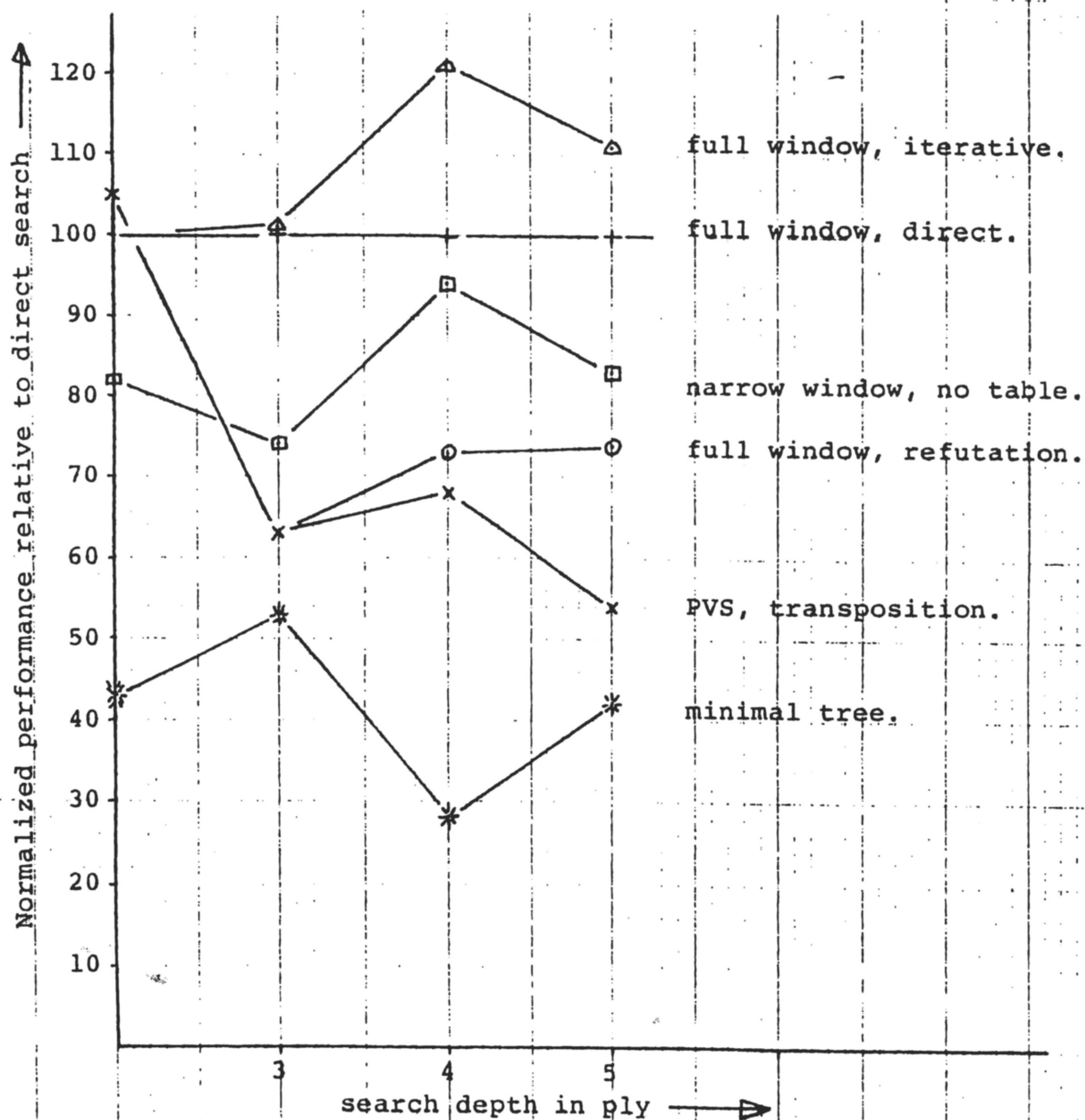

Figure 4: Performance Comparison of alpha-beta enhancements