INTERNATIONAL
COMPUTER CHESS ASSOCIATION

ICCA JOURNAL

CONTRIBUTIONS

Vol. 6, No. 4
December 1983

AN IMPROVEMENT TO THE SCOUT TREE SEARCH ALGORITHM

Alexander Reinefeld
University of Hamburg

ABSTRACT

This paper presents a new algorithm for pruning game trees, named "Negascout" because of its relation to Pearl's "Scout" algorithm. As Negascout has been developed on the basis of the well-known alpha-beta algorithm, its program structure is conceptually simple and can easily be implemented in chess programs.

To demonstrate the potential of Negascout as a practical game searching tool, empirical results are reported comparing the performance of Negascout to alpha-beta. It is shown that Negascout greatly benefits from the application of a transposition table, which is commonly used in chess programs to prevent re-searching of already evaluated positions. Using this technique, Negascout examines 20 to 30 percent fewer terminal nodes than alpha-beta.

INTRODUCTION

Most game-playing programs choose their moves by searching a large tree of potential continuations. The problem with tree searching is that the search space grows exponentially with the depth of the search. Efficient pruning techniques are required to reduce the search space without losing any information concerning the optimal move.

The most widely used method for pruning trees of two-person zero-sum games like chess is the alpha-beta algorithm [1]. Since its introduction in 1958 by Newell, Shaw and Simon [2], several additional heuristics have been applied to achieve further speed-up.

In 1980 Pearl proposed a new pruning algorithm called Scout [3] which offers some advantages over alpha-beta when searching certain large game trees. The practical efficiency of Scout is due to the fact that it traverses most of

the nodes without evaluating them precisely. A boolean function determines at each visited node whether its value exceeds some previously computed reference value. Since most tests result in exempting the node from further consideration, few subtrees remain to be revisited to compute their precise minimax value.

The Negascout algorithm presented in this paper searches the game tree in a way similar to Scout's. But instead of computing the root value by a minimax function, Negascout uses the negamax approach which results in a simpler program structure resembling the alpha-beta algorithm. Thus Negascout can easily be implemented in chess-playing programs.

In addition, Negascout often searches fewer nodes than Scout because it traverses the tree with two bounds (like alpha-beta) rather than using a boolean function (like Scout). In this paper it is suggested that Negascout becomes even more effective than alpha-beta when searching large game trees.

Recently, Campbell and Marsland published a similar algorithm named PVS - for principal variation search [4,5]. Since PVS consists of two functions (PVS and falphabeta), its implementation seems to be more complex than that of Negascout.

THE ALPHA-BETA ALGORITHM

The alpha-beta algorithm speeds up the search by pruning irrelevant branches from the lookahead tree without loss of information. In the PASCAL-like notation following the descendants of node p are called p.1, ..., p.b, assuming a constant branching factor b at each internal position. The function "evaluate" assigns static values to each terminal node at depth = d from the point of view of the side to move.

The interval enclosed by the two bounds alpha and beta is commonly referred to as a search window. In order to return the correct minimax value, alpha-beta must be invoked with a search window that covers the full range of values which can be produced by the evaluation function.

If the root value does not lie within the initial window, the tree search is said to fail. It must then be repeated with a larger window to determine the correct minimax value. In this case the "fail-soft alpha-beta" [6] shown in figure 1 often returns a tighter bound on the true score than the original alpha-beta algorithm [1].

```
 1 FUNCTION falphabeta (p: POSITION; alpha, beta, depth: INTEGER) : INTEGER;
 2 VAR i,t,m: INTEGER;
 3 BEGIN
 4     IF depth = d THEN RETURN (evaluate(p))
 5     ELSE
 6     BEGIN m := -∞;
 7           FOR i := 1 TO b DO
 8           BEGIN t := -falphabeta (p.i, -beta, -max(alpha,m), depth + 1);
 9                 IF t > m THEN m := t;
10                 IF m >= beta THEN RETURN(m);
11           END;
12           RETURN(m);
13     END;
14 END;
```

Figure 1: The "fail-soft" alpha-beta algorithm.

THE NEGASCOUT ALGORITHM

The Negascout algorithm (figure 2) computes the minimax value of a uniform game tree with branching factor b and search depth d. It requires four more statements in addition to the alpha-beta function.

The starting statements are the same as in alpha-beta: If the position p is a leaf, Negascout returns its static value (line 4). Otherwise the variables m and n are initialized with $-\infty$ and beta respectively (line 6 and 7). Then Negascout "scouts" the successors of p from left to right. The leftmost son p.1 is searched with the interval (-beta, -alpha) whereas the rest of the sons p.2, ..., p.b are searched using the zero-width window (-m-1, -m) which has been assigned at line 15 immediately after searching the leftmost son.

Since this null window contains no element (assuming integer values), the search is bound to fail. The direction of the failure (high or low) tells whether the move can be discarded from further consideration.

```
 1 FUNCTION negascout (p: POSITION; alpha, beta, depth: INTEGER) : INTEGER;
 2 VAR i,t,m,n: INTEGER;
 3 BEGIN
 4    IF depth = d THEN RETURN (evaluate(p))
 5    ELSE
 6    BEGIN m := -∞;
 7          n := beta;
 8          FOR i := 1 TO b DO
 9          BEGIN t := -negascout (p.i, -n, -max(alpha,m), depth+1);
10                IF t > m THEN
11                   IF (n = beta) OR (depth >= d-2)
12                   THEN m := t
13                   ELSE m := -negascout (p.i, -beta, -t, depth+1);
14                IF m >= beta THEN RETURN (m);
15                n := max (alpha,m) +1;
16          END;
17          RETURN (m);
18    END;
19 END;
```

Figure 2: The Negascout algorithm.

If the null window search fails high ("t > m" at line 10), Negascout must revisit the same subtree with a wider window to determine the true back-up value. Only in two cases there is no need to perform any re-search: Firstly, the back-up values returned by a full-width window search are correct (see condition "n = beta" at line 11). Secondly, Negascout's "fail-soft refinement" always returns correct minimax scores at the two lowest tree levels ("depth >= d-2"). In all other cases the search must be repeated with the new window (-beta, -t) (line 13). Note, that the re-search is done with a more realistic window than alpha-beta uses because of the tighter t-value.

The pruning condition (line 14) is the same as in alpha-beta: if m is greater or equal to beta the remaining sons can be discarded from further search.

In order to get a closer insight into the search process an example will be discussed. Figure 3 shows the smallest uniform game tree in which alpha-beta visits one node (p.2.1.2) which is exempted by Negascout.

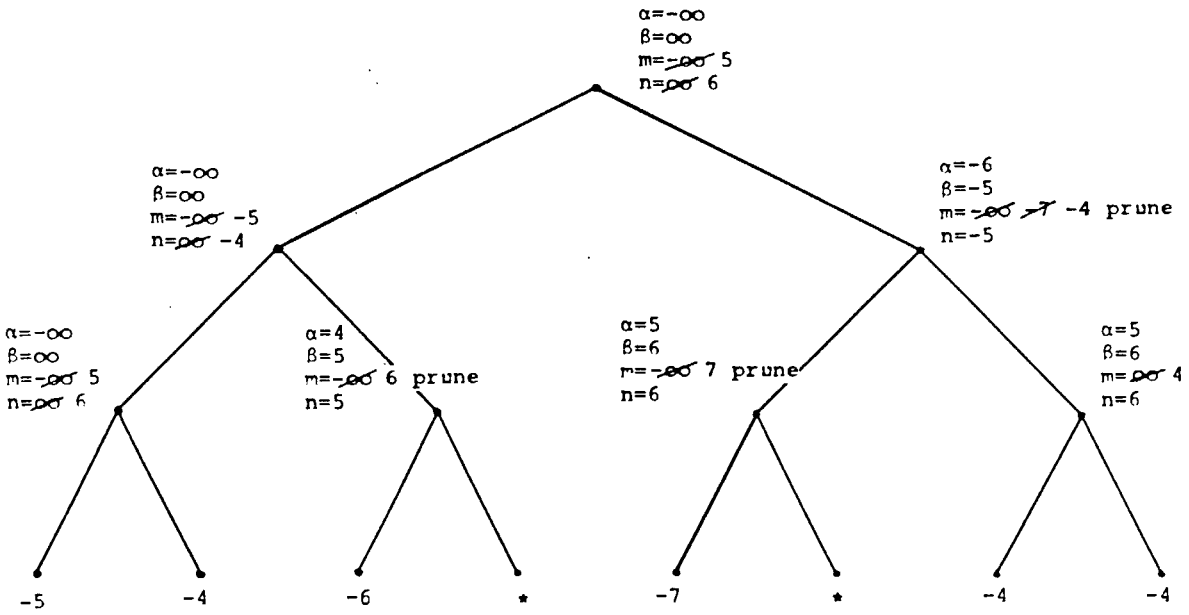

Figure 3.: Smallest uniform tree showing Negascout to advantage

The leftmost branches are traversed with the initial window $(-\infty, +\infty)$. After having computed the temporary minimax value m = 5, Negascout starts visiting the right subtree of the root with the null window (-6, -5). At node p.2.1 only the left son needs to be considered because its value (-7) results in pruning the right successor. Finally Negascout visits the two rightmost leaves p.2.2.1 and p.2.2.2 which cannot affect the ultimate minimax value m = 5.

REFINEMENTS OF NEGASCOUT

When performing a re-search, Negascout traverses the same subtree from left to right once more. Since the new minimax value normally is not located at the leftmost terminal node, subsequent re-searches are required. So it would be helpful to save more information than just the fact of failing high or low.

The actual implementation maintains a path to the leftmost leaf which caused the failure. The second search descends along this path and traverses the rest of the subtree from left to right in a bottom-up manner. The time and space overhead of the improvement are the same as those to maintain a principal variation of the whole game tree. In fact, the determination of the principal variation is a side-effect of this refinement. Experiments have shown that the number of leaf visits is typically reduced by 8% when searching trees (d = 7, b = 10) with uniform distributed terminal values.

Note that Negascout is less effective than alpha-beta when searching tall game trees of depth less than three. (Figure 3 is the smallest uniform tree which is favorable to Negascout.) Due to alpha-beta's directional search each node is visited at most once. Negascout, in contrast, often revisits the same nodes several times because it is called recursively with a null window when re-searching subtrees. So it should be profitable to call alpha-beta instead of Negascout when performing a re-search of a small subtree.

We could go to the extreme of substituting the Negascout call at line 13 (figure 2) by an alpha-beta call, yielding an algorithm which resembles the principal variation search [4]. (There is still one difference: Negascout never performs any re-search at the two lowest tree levels.)

Other enhancements used in chess programs such as "iterative deepening", the "killer" heuristic or the application of transposition tables are expected to be even more favorable to Negascout than to alpha-beta, because they are ordering the moves in a best-first list, thus reducing the number of re-searches.

EMPIRICAL COMPARISON OF NEGASCOUT AND ALPHA-BETA

In order to demonstrate the potential of Negascout as a practical game-searching tool some experiments have been run using uniform game trees of different sizes.

The main problem of experiments like these is to choose a realistic distribution function for assigning terminal values. When analysing evaluation functions of chess playing programs (e.g., [7]) we found that they return only a few different scores when searching end-game positions, in contrast to several thousand different scores in the middle game. Even more embarrassing is the fact that it seems quite impossible to simulate the various dependencies among different terminal nodes all over the three.

For the experiment presented in this paper we have chosen a uniform distribution of 200 different terminal values, allowing possible duplicate scores. The trees searched were uniform with several width-depth combinations. Of each tree size, 30 different trees were generated independently and each searched by alpha-beta and by Negascout.

We used two different methods to measure the performance of Negascout: The first method is to determine the total number of terminal node evaluations (NBP) and compare it to alpha-beta's. Assuming that the terminal evaluation is the major cost of tree searching, this method provides a quite realistic performance measure. Note: those leaves which are visited n times by Negascout due to re-searches are counted for n.

The second performance measure is to count each different leaf visited exactly once, regardless whether it has been searched several times or not. This is simply done by compiling a large table which contains all leaves searched by Negascout. Before evaluating any position, a table look-up determines whether the leaf has already been searched before. This corresponds to the "hash table" method used in most chess-playing programs.

Note that we are considering artificial game trees in which any position is distinct from any other position. In this case alpha-beta does not profit by such a hash table because each leaf is visited at most once. In the game of

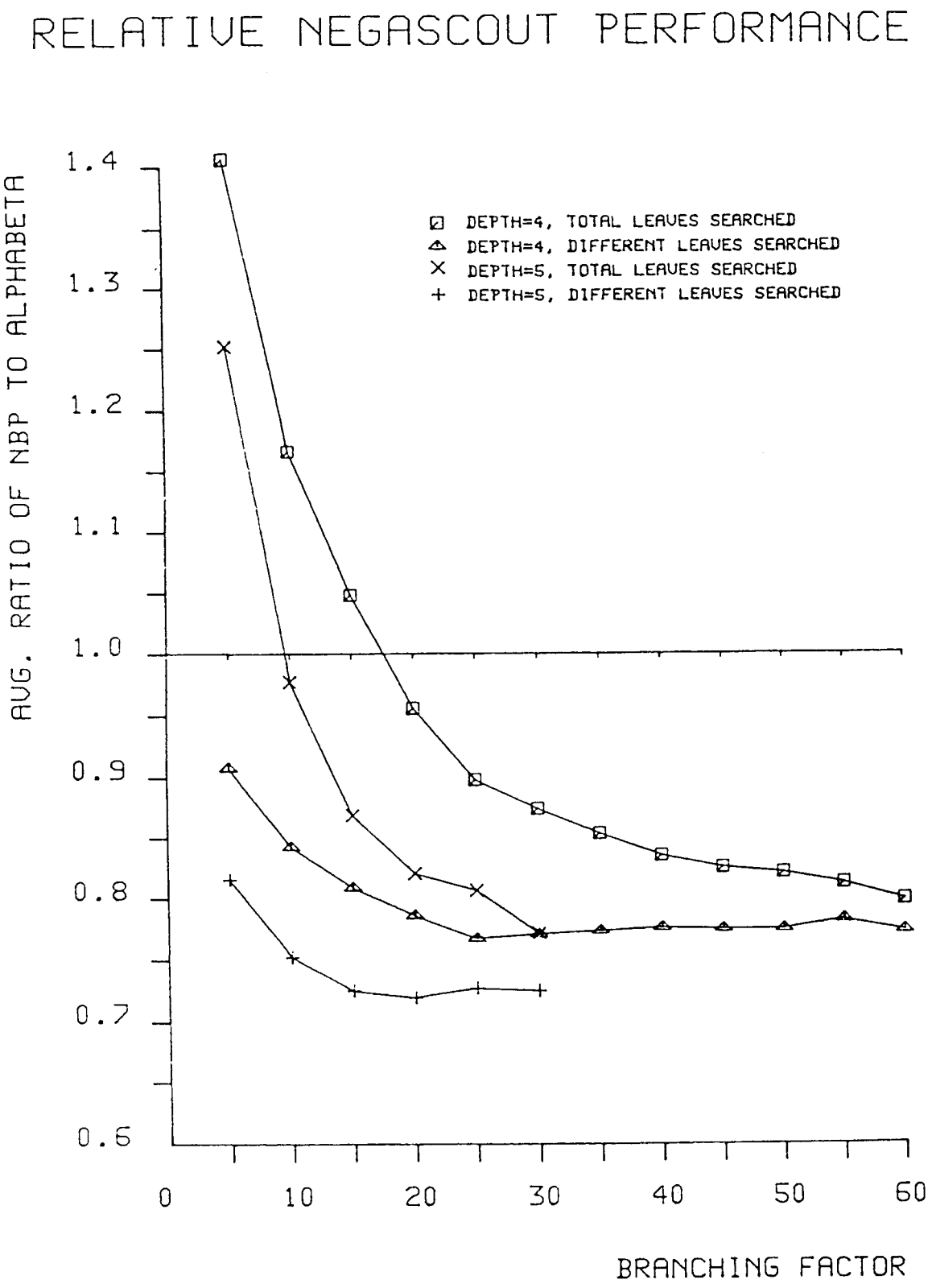

Figure 4.: Relative Negascout performance

chess, in contrast, it is not uncommon for positions to recur at several places throughout the tree. Due to this fact most chess programs are using a transposition table.

RESULTS

The curves presented in figure 4 show that Negascout is superior to alpha-beta when searching game trees with branching factors from 20 to 60, which are typical for the game of chess.

First consider the data obtained by counting the total number of leaves searched. As expected, Negascout is outperformed by alpha-beta when searching (tall) trees with low branching factors. In this case any single re-search greatly affects the performance of Negascout because the total number of leaves searched is small.

From the exact data (not reported here) it can be concluded that most of the trees with low branching factors ($b < 15$) are searched more or less effectively by Negascout; only a few of them require several re-searches. Of course, this results in a large standard deviation of the total number of leaf visits.

However, when increasing the branching factor, the speed-up due to the null window search compensates for the time-consuming re-searches. Negascout visits approximately 20 percent fewer leaves than alpha-beta when considering typical chess tree sizes.

If a hash table is used (i.e., the different leaf-visits are counted only) Negascout is superior to alpha-beta at any tree size. Due to the application of the zero-width window each such search fails. Thus Negascout prunes the maximal number of nodes while traversing the tree with a null window. Even when re-visiting a subtree with an opened window, Negascout searches less nodes than alpha-beta because its window is narrower than the alpha-beta window. This is caused by the preceding null window search which has determined a tighter t-value.

Nevertheless, all curves exhibit saturation at high branching factors. Since there exist only 200 different terminal scores, the principal variation is likely to be located in the left part of large trees. Thus the trees are almost perfectly ordered and both algorithms approach the optimal case.

CONCLUSION

The Negascout pruning algorithm has been empirically compared to the alpha-beta function. The results confirm that Negascout outperforms alpha-beta when searching game trees of useful size.

Especially the application of hash tables which are commonly used in chess programs to prevent the re-evaluation of already visited positions are more favorable to Negascout than to alpha-beta. Other alpha-beta enhancements are also expected to increase Negascout's performance because most of them are ordering the moves in a best-first list and therefore reducing the number of re-searches.

However, the point should be stressed that the efficiency of any search algorithm greatly depends on the distribution function of the terminal values. In order to achieve a realistic performance measure a further investigation is to implement Negascout in an actual game-playing program. For this purpose the game of chess is ideally suited because its average tree width is large enough to allow efficient application of Negascout.

ACKNOWLEDGEMENTS

I would like to thank Professor F. Schwenkel (University of Hamburg) and Professor T.A. Marsland (University of Alberta) for their many constructive comments which helped reduce the potential ambiguities in this paper.

REFERENCES

[1] Knuth, D.E. & Moore, R.W. (1975). An analysis of alpha-beta pruning. Artificial Intelligence, 6, pp. 293-326.

[2] Newell, A. & Shaw, J.C. & Simon, H.A. (1963). Chess-playing program and the problem of complexity. Computers and Thought (eds. E.A. Feigenbaum and J. Feldman), pp. 109-133. McGraw-Hill, New York.

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

[4] Marsland, T.A. (1982). Relative performance of the alpha-beta algorithm. ICCA Newsletter, Vol. 5, No. 2, pp. 21-24.

[5] Campbell, M.S. & Marsland, T.A. (1983). A comparison of minimax trees search algorithms. Artificial Intelligence, 20, pp. 347-368.

[6] Fisburn, J. (1981). Three optimizations of alpha-beta search. Computer Science Department, University of Wisconsin-Madison (May 1981). Appendix to Ph.D. Thesis.

[7] Slate, D. & Atkin, L. (1977). Chess 4.5 - The Northwestern University chess program. Chess Skill in Man and Machine (ed. P.W. Frey), pp. 82-118. Springer-Verlag, New York.