The History Heuristic and

Alpha-Beta Search Enhancements in Practice

Jonathan Schaeffer

Abstract - Many enhancements to the alpha-beta algorithm have been proposed to help reduce the size of minimax trees. A recent enhancement, the history heuristic, is described that improves the order in which branches are considered at interior nodes. A comprehensive set of experiments is reported which tries all combinations of enhancements to determine which one yields the best performance. Previous work on assessing their performance has concentrated on the benefits of individual enhancements or a few combinations. However, each enhancement should not be taken in isolation; one would like to find the combination that provides the greatest reduction in tree size. Results indicate that the history heuristic and transposition tables significantly out-perform other alpha-beta enhancements in application generated game trees. For trees up to depth 8, when taken together, they account for over 99% of the possible reductions in tree size, with the other enhancements yielding insignificant gains.

Index Terms - Alpha-beta search, minimax search, game trees, history heuristic, transposition tables, minimal window search, killer heuristic.

1. Introduction

Many modifications and enhancements to the alpha-beta ($\alpha\beta$) algorithm have been proposed to increase the efficiency of minimax game tree searching. Some of the more prominent ones in the literature include iterative deepening [1], transposition tables [2], refutation tables [3], minimal window search [4], aspiration search [5] and the killer heuristic [3]. Some of these search aids seem to be beneficial while others appear to have questionable merit. However, each enhancement should not be taken in isolation; one would like to find the combination of features that provides the greatest reduction in tree size. Several experiments assessing the relative merits of some of these features have been reported in the literature, using both artificially constructed [6] and application generated [1, 7, 8] trees.

The size of the search tree built by a depth-first $\alpha\beta$ search largely depends on the order in which branches are considered at interior nodes. The minimal $\alpha\beta$ tree arises if the branch leading to the best minimax score is considered first at all interior nodes. Examining them in worst to best order results in the maximal tree. Since the difference between the two extremes is large, it is imperative to obtain a good ordering of branches at interior nodes. Typically, application dependent knowledge is applied to make a "best guess" decision on an order to consider them in.

In this paper, a recent enhancement to the $\alpha\beta$ algorithm, the history heuristic, is described [9, 10]. The heuristic achieves its performance by improving the order in which branches are considered at interior nodes. In game trees, the same branch, or *move*, will occur many times at different nodes, or *positions*. A history is maintained of how successful each move is in leading to the highest minimax score at an interior node. This information is maintained for every different move, regardless of the originating position. At interior nodes of the tree, moves are examined in order of their prior history of success. In this manner, previous search information is accumulated and distributed throughout the tree.

A series of experiments is reported that assess the performance of the history heuristic and the prominent $\alpha\beta$ search enhancements. The experiments consisted of trying all possible combinations of enhancements in a chess program to find out which provides the best results. The

reductions in tree size achievable by each of these enhancements is quantified, providing evidence of their (in)significance. This is the first comprehensive test performed in this area; previous work has been limited to a few select combinations [1, 7, 8]. Further, this work takes into account the effect on tree size of ordering branches at interior nodes, something not addressed by previous work.

The results show that the history heuristic combined with transposition tables provides more than 99% of the possible reductions in tree size; the others combining for an insignificant improvement. The history heuristic is a simple, mechanical way to order branches at interior nodes. It has none of the implementation, debugging, execution time and space overheads of the knowledge-based alternatives. Instead of using explicit knowledge, the heuristic uses the implicit knowledge of the "experience" it has gained from visiting other parts of the tree. This gives rise to the apparent paradox that less knowledge is better in that an application dependent knowledge-based ordering method can be approximated by the history heuristic.

2. αβ and its Enhancements

An *adversary* or *game* tree is one in which two players alternate making moves. Two types of nodes in the tree are distinguished. Nodes at the bottom of the tree are called *leaf* nodes while those with successors are *interior* nodes. An interior or leaf node may be referred to as a *position*. An *evaluator* is used at leaf nodes to assign a score measuring the merit of the position. A *move* is the operation by which a position is transformed into a successor position.

For each subtree, αβ maintains lower (α) and upper (β) bounds on the range of minimax values that can be *backed-up* to affect the value at the root of the root. The benefits of the algorithm come from the elimination of sub-trees without search once it is proven their value must lie outside the αβ search window. Sub-trees eliminated in this manner are said to be *cutoff*. αβ does not eliminate the exponential growth of the trees; it merely dampens it. In the optimal case, for uniform trees of depth d (or *d ply*) and branching factor w, $w^{\lceil d/2 \rceil} + w^{\lfloor d/2 \rfloor} - 1$ leaf nodes need be considered [11]. In the worst case, αβ becomes a full minimax search, evaluating w^d leaf nodes. The size of the tree depends on the order in which moves are examined at interior nodes.

Considering the move producing the best minimax score first at each interior node in the tree produces the *minimal* or *optimally-ordered* $\alpha\beta$ tree.

Many enhancements to $\alpha\beta$ have been suggested in the literature. They are usually based on one or more of the following principals:

1) *ordering*, improving the order in which moves are examined at interior nodes ,

2) *window size*, the smaller the search window (absolute difference between α and β), the greater the chance for a cutoff to occur, and

3) *re-using information*, saving the results of sub-trees examined in the event that this sub-tree re-appears at a later time.

Following is a brief discussion of the major $\alpha\beta$ enhancements used in practice.

A) Iterative deepening [1, 8]. This technique is used to increase the probability that the best move is searched first at the root of the tree. A series of staged searches are carried out to successively larger search depths, using the best move found for a $d-1$ ply search as the first examined in a d ply search. As the search progresses deeper, more and more confidence is gained in the best move. The cost of an iteration is strongly influenced by the branching factor w. For chess programs (with w typically around 35), the cost of iterating an extra ply to an odd depth is about 8 times the cost of the previous search, whereas to an even depth, about 3 times (a consequence of the $\lfloor\rfloor$ and $\lceil\rceil$ operators in the minimal tree formula) [1]. Thus the cost of performing iterations to $1, 2, \cdots, d-1$ ply is only a small price to pay for having high confidence that the best move is being searched first for the larger d ply search.

Iterative deepening is useful when the search is constrained by time (as, for example, in a tournament game). At any point, the search can be terminated with the best move (to that point) known.

B) Transposition tables [2, 12]. Interior nodes of game trees are not necessarily distinct; it may be possible to reach the same position by more than one path. These tables take advantage of this by recording information about each sub-tree searched in the event that the identical position occurs again. The information saved would include the value of the sub-tree, the move that leads

to this value, and the depth to which the tree was searched. Before searching a new sub-tree, the table is interrogated to see if there is information relevant to the current node. If so, depending on the quality of the information found, the entire sub-tree may not need to be searched.

The tables are used in two ways. If the position is found in the table and it was previously searched to at least the desired depth, then the value for the sub-tree can be retrieved from the table. Note that this has the potential for building trees *smaller* than the minimal game tree by eliminating sub-trees without performing any search. If the information is not as reliable as desired (was not previously searched deep enough) the best move from the previous search can be retrieved. Since this move was previously best (albeit for a shallower search) there is a high probability it is also best for the current depth and should be tried first. Thus transposition tables allow for the reuse of information and help improve ordering.

To minimize access time, transposition tables are typically implemented as a hash table. As the table becomes full, collisions occur and information lost, potentially reducing their effectiveness. As a consequence, the tables are usually as large as possible.

C) Refutation tables [3]. The major disadvantage of transposition tables is their size. Refutation tables attempt to retain one of the advantages of transposition tables, when used with iterative deepening, but with smaller memory requirements. For each iteration, the search yields a path for each move from the root to a leaf node that results in either the correct minimax score or an upper bound on its value. This path from the $d-1$ ply search can be used as the basis for the search to d ply. Often, searching the previous iteration's path or *refutation* for a move as the initial path examined for the current iteration will prove sufficient to refute the move one ply deeper.

D) Minimal window [4, 8, 13]. Minimal window searching is a variant of $\alpha\beta$ based on the assumption that the first move considered is best and the remaining moves are inferior until proven otherwise. At an interior node, given that the first branch has been searched with a full window (α, β) and produced a value v inside the window, the remaining moves are searched with the minimal window $(v, v+1)$. The motivation for minimal window searching is that a smaller window generates a smaller tree. Essentially, minimal window searching is a probabilistic gamble

that most moves can be shown to be inferior with little effort. If the gamble proves incorrect, then additional work is required to do a re-search (to encompass the range of values $(v+1, \beta)$). At a node with w successors, $\alpha\beta$ only recognizes only one best move, which implies $w-1$ inferior ones.

E) Aspiration search [5, 14]. The $\alpha\beta$ search is usually started with a $(-\infty, \infty)$ search window. If some idea of the range in which the value of the search will fall is available, then tighter bounds can be placed on the initial window. Aspiration search involves using a smaller initial search window. If the value falls within the window, then the original window was adequate. Otherwise, one must re-search with a wider window.

F) Killer Heuristic [3]. Often during a search, most moves tried in a position are quickly refuted, usually by the same move. The killer heuristic involves remembering at each depth of search the move(s) which seem to be causing the most cutoffs ("killers"). The next time the same depth in the tree is reached, the killer move is retrieved and used, if valid in the current position.

Since searching the best move first at interior nodes has a dramatic effect on tree size, it is important to have a good move ordering. Devoting resources to achieving this ordering usually pays off. In positions where the transposition or refutation table can be used, they help order the moves by suggesting a likely candidate for best move based on information acquired from previous searches. But they don't tell how to order the remaining moves, nor give any information on how to order them at nodes where the tables provide no clues. The method used by most game playing programs is to apply application dependent heuristic knowledge to each move indicating how good the move is likely to be and sort based on these assessments. Of course, any ordering is probabilistic guessing since the desirability of a move usually cannot be reliably determined without performing some search. Depending on the quality of the knowledge used, this approach is usually superior to a random ordering.

Several studies have been performed investigating how close $\alpha\beta$ search with its enhancements can come to achieving the minimal tree. Campbell and Marsland [6] have investigated this

question using artificially-constructed trees, while Gillogly [1] and Marsland [7, 8] experimented using a chess program. Marsland's experiments considered trees of depths 2 through 6 ply [8]. For even depths, the results showed that αβ, enhanced with minimal window searching (the Principal Variation Search algorithm) and refutation and transposition tables, was able to build trees within twice the size of the minimal tree. For odd depths, this was reduced to a factor of 1.5. The factor of 2 for even depths is roughly the same as in Campbell and Marsland [6], but these results were achieved *without* using transposition and refutation tables. This illustrates the danger of using results from artificially-constructed trees as a guideline for how to search application generated trees. The study also showed that for shallow searches, the performance of refutation tables was comparable to that of transposition tables, but as the search depth increased, refutation tables out-performed transposition tables (probably because of transposition table over-loading).

A variety of studies have been performed demonstrating that minimal window search is usually superior to just αβ. The advantage is more pronounced in artificially constructed game trees [6, 13, 15] than in application generated trees [7, 8], where the stronger ordering of moves at interior nodes results in smaller trees.

The effectiveness of the killer heuristic has been questioned. Gillogly observed no benefits [1], whereas Hyatt claims significant reductions in tree size - as much as 80% [16]. This is a popular αβ enhancement, yet no one really seems to know how effective it is.

3. The History Heuristic

Using the terminology of Newell and Simon [17], a *problem space* is a set of *states* and *operators* that map states into states. Starting from an initial state of knowledge, one repeatedly applies operators to states, looking for a goal state. Operators can be viewed as parameterized functions; the parameters providing the specifics of what the operator should do to the state. In the context of game trees, states are positions and moves, operators. The move operation could be viewed as a function of three parameters: the initial position and the from-square and to-square of the move, producing a new successor position.

For many types of search trees, the parameters of an operator are not unique within the tree: two different states can have the same operator applied yielding new states. The history heuristic maintains information on whether there is a correlation between an operator and any success that the operator has in achieving a goal. At an arbitrary state in the problem space, the history heuristic prefers to explore operators with a history of success first.

For many types of game trees, the making of a move from one position yields a successor position that has many of the same properties as its predecessor. It is quite likely that the important features of the position, the ones that dictate what the good moves are, have not significantly changed and that a move considered good in the previous position will still be good. Since the two positions are similar, one could take a sophisticated approach and use analogies to show the similarity (for example [18]). However, this is an expensive proposition.

As an example, consider the trees built by chess programs. A move *M* may be shown to be best in one position. Later on in the search tree a similar position may occur, perhaps only differing in the location of one piece. This minor difference may not change the position enough to alter move *M* from still being best. *M*, with its prior history of success, may now be the first move considered in this position. This improves the ordering of moves, increasing the chances of a cutoff occurring.

In an $\alpha\beta$ framework, a *sufficient* (or good) move at an interior node is defined as one that

1) causes a cutoff, or

2) if no cutoff occurs, the one yielding the best minimax score.

Note that this does not imply that a *sufficient* move is the *best* move. A move causing a cutoff was *sufficient* to cause the search to stop at that node, but there is no guarantee that had the search continued, a better (but irrelevant) score might have been obtained by a latter move.

Every time a *sufficient* move is found, the history score associated with that move is increased. Moves that are consistently good quickly achieve high scores. Upon reaching an interior node, moves are ordered according to their prior history of success. Rather than using a ran-

dom or knowledge-based approach for deciding the order to consider moves in, a move's prior history of success can be used as the ordering criteria. The heuristic is experienced-based, acquiring information on what has been seen as an indication of what will be useful in the future. In effect, previous search information is being accumulated and distributed throughout the tree. This implies that the heuristic is *dynamic*; the ordering of branches being adaptive to previous conditions seen in the search tree.

Figure 1 illustrates the history heuristic in the $\alpha\beta$ algorithm. The lines marked with a * are those that differ from the standard algorithm. Note that the sorting operation is not specific to the history heuristic; the usual technique of assigning a heuristic-based score to each move requires a sort as well. However, some applications avoid the sort by generating moves as needed in some reasonable order. Also note that the heuristic is applied at all interior nodes, since one can never tell in advance with certainty whether a cutoff will occur or not.

Figure 2 shows an example of how the heuristic can affect tree searching. Assuming a depth-first, left to right, traversal of the tree, at interior node *I1*, move *M* was found to be sufficient. Later on in the search, at interior node *I2*, move *M* was one of the successor branches. Given no information, the successors of *I2* would be considered in arbitrary order. With the prior history of success of *M*, the successors of *I2* are re-ordered, allowing *M* to be considered first. Since nodes *I1* and *I2* are not dis-similar, there is a high probability that *M* will also be sufficient at *I2*, thereby reducing the size of tree traversed.

The implementation is based on two parameters; the *mapping* of moves to history and the *weight* of a *sufficient* move. The *mapping* used in the chess program was to specify a move as a 12-bit number (6-bits from-square, 6-bits to-square). Essentially, the *mapping* can be viewed as the parameters to the move operator. Although this simplistic representation hides a lot of the move details in the operator (for example, which piece is making the move), it allows for compact tables (2^{12} entries) that are easily accessed. Including more context information (for example, the piece moving) did not significantly increase performance (for a complete discussion, see [9]). Note that if you carry the idea of including context to the extreme, the result is a transposition

table.

What should be the history *weight* of a *sufficient* move? There are two considerations: First, the deeper the sub-tree searched, the more reliable the minimax value (except in *pathological* trees, rarely seen in practice [19]). Therefore, one idea is to associate higher history scores for moves successful on deep rather than shallow searches. Second, the deeper the search tree (and hence larger), the greater the differences between two arbitrary positions in the tree and the less they may have in common. Hence, sufficient moves near the root of the tree have more potential for being useful throughout the tree than sufficient moves near the leaf nodes. These considerations led the usage of a *weight* of 2^{depth}, where *depth* is the depth of the sub-tree searched. This scheme gives more weight to moves that are *sufficient* closer to the root of the tree than those near the leaves of the tree and favors moves that are roots of larger, rather than smaller, sub-trees. Several other *weights*, such as 1 and *depth*, were tried and found to be experimentally inferior to 2^{depth} (for a complete discussion, see [9]).

Note that the killer heuristic is just a special case of the history heuristic. Whereas the killer heuristic keeps track of one or two successful moves per depth of search, the history heuristic maintains the success for all moves at all depths. With the history heuristic, "killer" moves would earn high history scores. The generality of the history mechanism provides valuable ordering information at all nodes; not just those with "killer" moves.

A final point is that there is nothing particular to the history heuristic that restricts its usage to the αβ algorithm. Other types of search algorithms and applications may be able to use this technique. The success depends on whether the tree type is suitable (operators are not unique to each node and there is a correlation between using an operator and sub-tree values) and a suitable *mapping* and *weight* can be found.

4. Experiment Design

To assess the importance of various search features, the chess program *Phoenix* was enhanced to turn each one on or off selectively. Starting with a base program that performs αβ

with iterative deepening, a series of experiments was then performed whereby all possible combinations of search features were tried on a test set of positions. The Bratko-Kopec positions [20] have been used extensively for the benchmarking of sequential and parallel tree search performance in chess programs [6, 21, 22]. Iterative deepening is necessary to be able to properly evaluate refutation tables and hence is not one of the experiment parameters.

The search features used were:

1) Transposition Tables (*T*). The table contained $2^{13} = 8,192$ entries for depths 2 through 6, and $2^{16} = 65,536$ for depths 7 and 8 .

2) Refutation Tables (*R*).

3) Minimal Window Searching (*N*). The NegaScout variant [13].

4) Aspiration Searching (*S*), using a one pawn window around the result of the previous iteration.

5) Ordering using the History Heuristic (*H*).

A more detailed description of the implementation of each of these can be found in [9].

To assess the history heuristic as a move ordering mechanism, it is necessary to have some basis of comparison. Most search programs apply application dependent heuristics to try and suggest which moves might be good. Of course, the heuristics are only guesses; their validity often cannot be ascertained without search. This approach has two undesirable properties. The first is that the knowledge is (usually) static; it is unable to adapt to changing conditions on the board. The second problem is that the knowledge does not handle exceptions well. Both problems are symptoms of the same problem: inadequate knowledge. The real problem is that there is too much knowledge to draw on and the designer must make important cost/benefit decisions. Most of the knowledge one could use is not cost effective to implement and so the designer draws on the heuristics that are effective 90% of the time and accepts the resulting consequences. For interior node move ordering, the consequences can be a poor ordering resulting in a larger search tree.

To have a means of comparison for the history heuristic, *Phoenix* also has a heuristic-based

ordering mechanism (developed prior to the history heuristic). The methods used for ordering the moves are similar to most chess programs, although it is probably more sophisticated than most. It is, in fact, superior to the ordering mechanisms used in [7, 8]. This move ordering method will be referred to as the *knowledge heuristics* since the ordering is based on well-known, human-understandable, chess knowledge. The heuristics usually give an integral value in the range -25 to +25. When used with the history heuristic, the scores are combined, but the larger *H* scores quickly swamp the smaller knowledge heuristic scores. A detailed description of the routine can be found in [9]. Thus, the experiments have another parameter:

6) Ordering using Knowledge Heuristics (K).

This implies that a program without *K* or *H* has no explicit move ordering and just searches the moves in the order they were generated in. In fact, this is not a completely random order, since there is some bias in the order moves are generated. In the absence of a transposition or refutation table move, all program variants consider capture moves before non-capture ones.

Finally, for comparison purposes, the experiment has been enhanced to include the killer heuristic. Since there seems to be considerable doubt as to its effectiveness, a controlled experiment might provide insight into its effective behavior.

7) Killer Heuristic (L).

Previous experiments with $\alpha\beta$ enhancements have neglected the issue of interior move ordering. In [7, 8] since the experiments were performed using a chess program, it is reasonable to assume that some application specific knowledge was used to order the moves and thereby reduce tree size. One might question the conclusions they draw from their experiments, since the ordering will influence tree size, and tree size may influence the effectiveness of some enhancements.

Excluding the killer heuristic, there are 6 parameters resulting in $2^6 = 64$ different sets of enhancements that can be tried. All possible combinations were searched to depths of 2, 3, 4, and 5 ply. Since the killer heuristic is just a special case of the history heuristic, only a few of the combinations with it present were tried just to establish the relationship between *L* and *H*. To

depths of 6, 7, and 8 ply, experiments continued with only those combinations that showed significant reductions in tree size through 5 ply. A total of 2000 hours of VAX 11/780 equivalent time was required to perform the experiments.

5. Experiment Results

One measure for comparing search algorithm performance is elapsed CPU time. From the implementor's point of view, this is the most important consideration. However, any timing results are machine and implementation dependent. Another measure is the number of bottom positions (leaf nodes) examined (NBP), which has been used extensively in the literature [1, 6, 23, 24]. However, NBP is an inadequate measure since it treats all leaf nodes equally and assumes interior nodes are of negligible cost. In fact, for most applications, neither is a reasonable assumption. NBP may be a good theoretical measure, but not a good practical one.

A better measure would be one that measures the size of the tree in a manner that is correlated with program running time. The *node count (NC)* measure counts all nodes in the tree where computation occurs. This includes interior, leaf, and any nodes in sub-trees built as part of leaf node evaluation. At least for *Phoenix*, this count has been shown to be strongly correlated with program running time [9]. Note that since the execution time overhead of most of the enhancements is negligible (the cost being dominated by leaf node evaluation), *NC* accurately reflects the relative running time of the program variants. The exception is the *K* heuristic, since you can add as much or as little knowledge to it as you want. *NC* has the advantage of factoring this out of the results. If you view the *NC* graphs as measuring time, it is important to note that *K*'s relative time performance is over-stated since it added an additional 5% on to the execution time. In this paper, both the NBP and NC measures will be used.

The best measure of the performance of a combination of search enhancements would be to compare its tree sizes with that of the minimal tree. Unfortunately, it is difficult to know the size of the minimal tree, due to the variable branching factor and presence of terminal nodes in the tree (although Ebeling has recently devised a good method for approximating the minimal tree [25]). Instead, performance will be measured as the percentage improvement in search efficiency as

compared to that achievable by the best combination of search features. For each depth, all possible combinations of enhancements were tried, with the one giving the best result (using the *NC* or *NBP* measure) being labeled N_{best}. The program was also run with none of the enhancements (just αβ with iterative deepening) yielding N_{none}. For a given combination of enhancements *enh*, its percentage improvements reflects how much of the reduction in tree size from N_{none} to N_{best} the presence of *enh* achieved. The calculation is done using the formula:

$$reduction_{enh} = \frac{N_{none} - N_{enh}}{N_{none} - N_{best}} * 100$$

Figure 3 shows the percentage reduction in tree size attributable to the addition of a single search enhancement for depths 3 - 8 ply using NBP as a measure. Figure 4 shows all the combinations of two enhancements that exceed a 70% reduction.

Figure 3 shows that history heuristic performs well, but its efficiency appears to drop after depth 7. The knowledge heuristic's performance runs almost parallel to *H*, albeit with less search efficiency. The effectiveness of transposition tables, on the other hand, increases with search depth. Refutation tables seem to provide constant performance, regardless of depth, and appear to be a poor substitute for transposition tables. Aspiration and minimal window search provide small benefits. With two enhancements (Figure 4), the combination of transposition tables and the history heuristic achieves over 99% of the possible reduction in tree size. Since transposition tables provide a different type of search reduction than the history heuristic (eliminating sub-trees without search and providing limited move ordering, versus more comprehensive move ordering), it is not surprising that the two work well together. The performance of these two features indicates that the other enhancements can provide at most a small percentage of improvement. Combining three or more enhancements does not change this observation (details can be found in [9]).

Figures 5 and 6 present the same data using the *NC* measure and provide a better indication of how tree size effects execution time. Using this measure, transposition tables and history heuristic show up even better. Both methods have the advantage of steering the search towards previously encountered parts of the tree, thereby increasing the computations that can be re-used.

In both Figures 4 and 6, the performance of some enhancements appear to oscillate with depth. This is largely attributable to the odd/even depth of the alpha-beta search. As mentioned in section 2, the incremental cost of growing the tree an additional ply to an odd depth is greater than for an even depth. Since some combinations of enhancements are tied to search depth, it suggests that a different method for representing the data might be more informative. One could separately plot the odd and even depth results, but given the paucity of depths for which results are available, this was not done.

From 7 to 8 ply, the percentage improvement attributable to the history heuristic decreases (although significantly less for *NC* than *NBP*). There are two reasons for this. The first is that the trees are getting so large that, although the history heuristic is just as effective as before, its performance relative to the larger tree seems poorer. Recall that the percentage reduction is relative to the maximum possible savings. If an enhancement starts to perform well (in this case, the transposition tables), it can increase the possible savings and as a result, decrease the percentage improvement attributable to the other enhancements.

A second reason is that the history heuristic tables are becoming over-loaded. As the search becomes deeper, there is a greater variation in the positions that occur in the tree. Table entries can be flooded with scores from the lower depths that affect the orderings of moves near the root of the tree. In fact, the results reported here were constrained to only save history information within the first $depth - 1$ ply of the tree, but to use the information throughout the entire tree. This constraint provided a small improvement in performance, confirming that history tables can become flooded with information, decreasing their usefulness.

As an interior node ordering heuristic, the history heuristic dominates the knowledge heuristics approach under all scenarios. Again, this result may be a comment on the quality of the chess specific knowledge used to order the moves and that a better routine might be written with increased effort. An interesting feature in the diagrams is that sometimes the combination of both *H* and *K degrades* the performance compared to *H* alone. This condition is caused by the two ordering mechanisms disagreeing over the importance of a move. *K* may rate some moves highly

that superficially appear to be good but in the current context, are not. Recall that the knowledge heuristics statically assign scores to moves without performing any search. The history heuristic also does not perform search, but makes use of previous search results to do its ordering. The history heuristic is *adaptive* to the conditions in the search tree and is *insensitive* to application dependent pre-conceptions. The knowledge heuristics fail under conditions of exception, where exception is defined as any condition not covered by the heuristics. Consequently, inappropriate *K* scores can decrease the effectiveness of ordering, resulting in larger trees.

Not surprisingly, the killer heuristic shows up as being inferior to the history heuristic. However, a consideration is that there is less overhead in maintaining killer than history information; in particular, no sorting. For high performance machines that implement $\alpha\beta$ in hardware or microcode, the simplicity of implementing the killer heuristic may make it preferable (for example, [25]).

The performance of transposition tables consistently increases with depth. As the trees become larger, there are potentially more transpositions and therefore more savings to be found. However, the savings are limited by the size of the tables used and to 6 ply, it appears there were few problems with table over-loading problems. For depths 7 and 8, over-loading became a serious problem and table sizes had to be increased. At further depths, if performance starts to decrease, one can always increase the size of the tables, reducing the number of collisions that occur.

In all cases, the transposition table significantly out-performs the refutation table. This is not surprising since the refutation tables contain only a small subset of the information available with transposition tables. Marsland [8] has published results comparing refutation and transposition tables. His experiments included 2 through 6 ply searches using transposition tables of size 8K. For all depths examined, refutation tables were found to be roughly comparable in performance to transposition tables. When comparing these results with those reported here, it is important to note that tree sizes obtained by Marsland's program are 3-4 times larger than those generated by *Phoenix* while using the same data set (the 24 Kopec-Bratko positions). Because of

this, the transposition table fills up more quickly and may become over-loaded, reducing its effectiveness and making refutation tables look more attractive. Marsland concludes that an 8K transposition table is too small for 6 ply searches of complex middlegame positions. In *Phoenix*, refutation tables and 8K entry transposition have been examined to 6 ply, again showing little justification for refutation tables. The main contribution of the refutation table is to remember an important subset of the transposition table information. This is an important enhancement on a memory constrained system where transposition tables are not possible. But for machines with available memory, as long as the transposition table does not become over-loaded, refutation tables will always be of minimal benefit.

The two search window enhancements, aspiration and minimal window, usually provide a small improvement in performance. When used with the history heuristic and transposition tables, both enhancements singlely and together provide a small reduction in tree size. Since both are easy to implement and cost nothing in additional storage or computation, one might as well include them in any real tree searching program. The benefits, however, are probably going to be small.

In conclusion, what is the best combination of alpha-beta enhancements? Clearly, the history heuristic and transposition tables provide most of the savings. One loses nothing and gains slightly by also including aspiration and minimal window search.

Since the experiments reported here illustrate the importance of move ordering, it is difficult to compare these results with those of other researchers. Without having addressed the issue of interior node move ordering, some reported results may be heavily biased by a good or poor ordering scheme (for example [1, 7, 8]) but it is impossible for us to know. By considering a range of ordering schemes (from no ordering to the knowledge and history heuristics), the results reported here illustrate the range of efficient versus inefficient alpha-beta searching and can be used to assess the effectiveness of interior node ordering methods.

The history heuristic was added to the chess program *TinkerBelle*. This program used transposition tables and simple move ordering heuristics. The addition of the history heuristic

reduced the size of the average tree by 25%.

How close are we to the minimal tree? For artificially constructed trees it is easy to calculate the size of the minimal tree. Unfortunately, this is not necessarily true for application generated trees. Using an average branching factor, one can come up with a rough approximation of the minimal tree size. Eliminating transposition tables, the *NBP* results show the trees being built to be within a factor of 1.5 times that of the minimal tree. With transposition tables, the trees become smaller than the minimal tree.

6. Conclusions

The history heuristic is an inexpensive method for ordering descendants of interior nodes that is largely application independent. It has the advantage of being simple to implement, as compared to the time consuming, trial-and-error, knowledge-based approach. The algorithm substitutes "experience" for explicit knowledge and, at least in the case of $\alpha\beta$ searching applied to the game of chess, provides a viable alternative.

The heuristic has proven useful for parallel $\alpha\beta$ implementations. In [26] on a network of computers, the history tables were used two ways. First, each processor would maintain its own local history table. Second, periodically all the local tables would be merged and broadcast to all processors to achieve global sharing. This resulted in a noticeable improvement in program performance.

Given that it is relatively easy to build trees close to the minimal tree in size, there is nothing of interest left to explore in terms of *fixed depth* $\alpha\beta$ searching. New research efforts are concentrating on *variable depth* searches, so called *selective deepening* (for example, null moves [27, 28] and singular extensions [29] as enhancements to $\alpha\beta$, and conspiracy numbers [30, 31] as a new approach to minimax search).

Acknowledgments

Many thanks to Tony Marsland for his guidance and constructive criticism. Alexander Reinefeld, Murray Campbell, and the anonymous referees provided valuable suggestions.

References

1. J. Gillogly, *Performance Analysis of the Technology Chess Program*, Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, 1978.
2. D.J. Slate and L.R. Atkin, Chess 4.5 - The Northwestern University Chess Program, in *Chess Skill in Man and Machine*, P.W. Frey (ed.), Springer-Verlag, New York, 1977, 82-118.
3. S.G. Akl and M.M. Newborn, The Principle Continuation and the Killer Heuristic, *ACM Annual Conference*, 1977, 466-473.
4. J. Pearl, Scout: A Simple Game-Searching Algorithm with Proven Optimal Properties, *First Annual National Conference on Artificial Intelligence*, Stanford, 1980.
5. G. Baudet, *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*, Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, 1978.
6. M.S. Campbell and T.A. Marsland, A Comparison of Minimax Tree Search Algorithms, *Artificial Intelligence 20*, (1983), 347-367.
7. T.A. Marsland, A Review of Game-Tree Pruning, *Journal of the International Computer Chess Association 9*, 1 (1986), 3-19.
8. T.A. Marsland, Relative Efficiency of Alpha-Beta Implementations, *International Joint Conference on Artificial Intelligence*, Karlsruhe, 1983, 763-766.
9. J. Schaeffer, Experiments in Search and Knowledge, Ph.D. thesis, Dept. of Computer Science, University of Waterloo, 1986.
10. J. Schaeffer, The History Heuristic, *Journal of the International Computer Chess Association 6*, 3 (1983), 16-19.
11. D.E. Knuth and R.W. Moore, An Analysis of Alpha-Beta Pruning, *Artificial Intelligence 6*, (1975), 293-326.
12. R.D. Greenblatt, D.E. Eastlake and S.D. Crocker, The Greenblatt Chess Program, *Fall Joint Computer Conference 31*, (1967), 801-810.
13. A. Reinefeld, An Improvement of the Scout Tree Search Algorithm, *Journal of the International Computer Chess Association 6*, 4 (1983), 4-14.
14. R.A. Finkel, J. Fishburn and S.A. Lawless, Parallel Alpha-Beta Search on Arachne, *International Conference on Parallel Processing*, 1980, 235-243.
15. T.A. Marsland, A. Reinefeld and J. Schaeffer, Low Overhead Alternatives to SSS*, *Artificial Intelligence 31*, 1 (1987), 185-199.
16. R.M. Hyatt, *Cray Blitz - A Computer Chess Playing Program*, M.Sc. thesis, University of Southern Mississippi, 1983.
17. A. Newell and H.A. Simon, in *Human Problem Solving*, Prentice-Hall, 1972.
18. G.M. Adelson-Velskiy, V.L. Arlazarov and M.V. Donskoy, Some Methods of Controlling the Tree Search in Chess Programs, *Artificial Intelligence 6*, (1975), 361-371.
19. D.S. Nau, Quality of Decision Versus Depth of Search on Game Trees, Ph.D. thesis, Dept. of Computer Science, Duke University, 1979.
20. D. Kopec and I. Bratko, The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess, in *Advances in Computer Chess 3*, M.R.B. Clarke (ed.), Pergamon Press, 1982, 57-72.
21. M.M. Newborn, A Parallel Search Chess Program, *ACM Annual Conference*, 1985, 272-277.
22. T.A. Marsland, M. Olafsson and J. Schaeffer, Multiprocessor Tree-Search Experiments, in *Advances in Computer Chess 4*, D. Beal (ed.), Pergamon Press, 1985, 37-51.
23. J. Slagle and J. Dixon, Experiments with some Programs that Search Game Trees, *Journal*

of the ACM 2, (1969), 189-207.

24. A. Muszycka and R. Shinghal, An Empirical Comparison of Pruning Strategies in Game Trees, *IEEE Transactions on Systems, Man, and Cybernetics SMC-15*, 3 (1985), 389-399.
25. Carl Ebeling, All the Right Moves: A VLSI Architecture for Chess, Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon University, 1986.
26. J. Schaeffer, Distributed Game-Tree Searching, *Journal of Parallel and Distributed Computing*, 1989. To appear.
27. D. Beal, Null Moves, *Advances in Computer Chess V*, 1987.
28. G. Goetsch and M. Campbell, Experiments with the Null Move in Chess, *AAAI Sprint Symposium*, 1988, 14-18.
29. T. Anantharaman, M. Campbell and F.H. Hsu, Singular Extensions: Adding Selectivity to brute-Force Searching, *AAAI Spring Symposium*, 1988, 8-13.
30. D.A. McAllester, Conspiracy Numbers for Min-Max Search, *Artificial Intelligence 35*, 3 (1988), 287-310.
31. J. Schaeffer, Conspiracy Numbers, *Artificial Intelligence*, 1989. In press. Also to appear in *Advances in Computer Chess V*, D. Beal and H. Berliner (ed.), Elsevier Press, 1989.

Affiliation of author:

The author is with the Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1. This work was supported by the National Science and Engineering Research Council of Canada.

Footnotes:

1. A competitor in the 1986 World Computer Chess Championship, Cologne, West Germany.

2. The original experiments were conducted through to depth 6 and showed minimal signs of over-loading. When extended to depths 7 and 8, table conflicts necessitated an increase in table size.

3. A software predecessor of the chess machine *Belle.*

```
AlphaBeta( p : position; α, β, depth : integer ) : integer;
var
    bestmove, score, width, m, result : integer;
    rating : array[ 1..MAX_WIDTH ] of integer;

begin
    if depth = 0 then        { At a leaf node }
        return( Evaluate( p ) );

    width := GenerateMoves( moves );
    if width = 0 then        { No moves in this position }
        return( Evaluate( p ) );

    { Assign history heuristic score to each move }
    for m := 1 to width do
*       rating[ m ] = HistoryTable[ moves[ m ] ];
    Sort( moves, rating );   { Put highest rated moves first }

    score := -∞4;
  for m := 1 to width do
    begin
     { Recurse using nega-max variant of αβ }
     result := -AlphaBeta( p.moves[m], -β, -α, depth-1 );
     if result > score then
        score := result;

     { Check for cut-off }
     if score >= β then
       begin
         { Cut-off: no further sub-trees need be examined }
         bestmove := moves[ m ];
         goto done;
       end;

     α := MAX( α, score );
    end;

 done:
   { Update history score for best move }
*  HistoryTable[ bestmove ] = HistoryTable[ bestmove ] + 2^depth ;
    return( score );
end.
```

Fig. 1. The History Heuristic and Alpha-Beta.

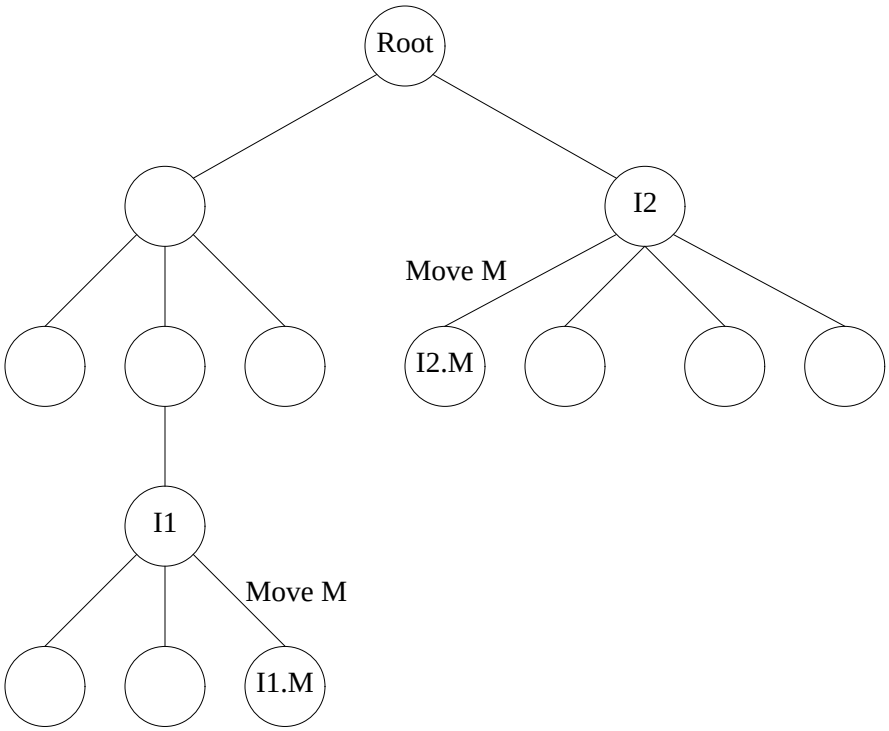

Fig. 2. History Heuristic Example.

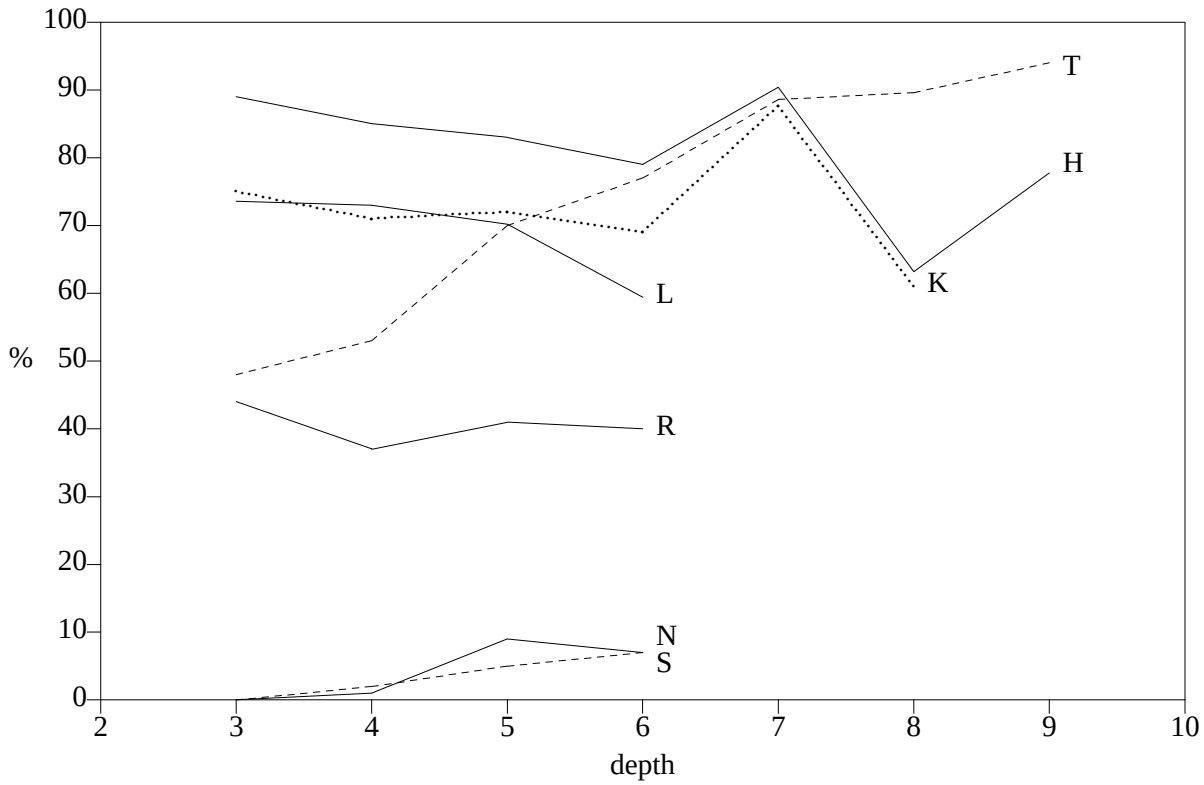

Fig. 3. Single Search Enhancement (NBP).

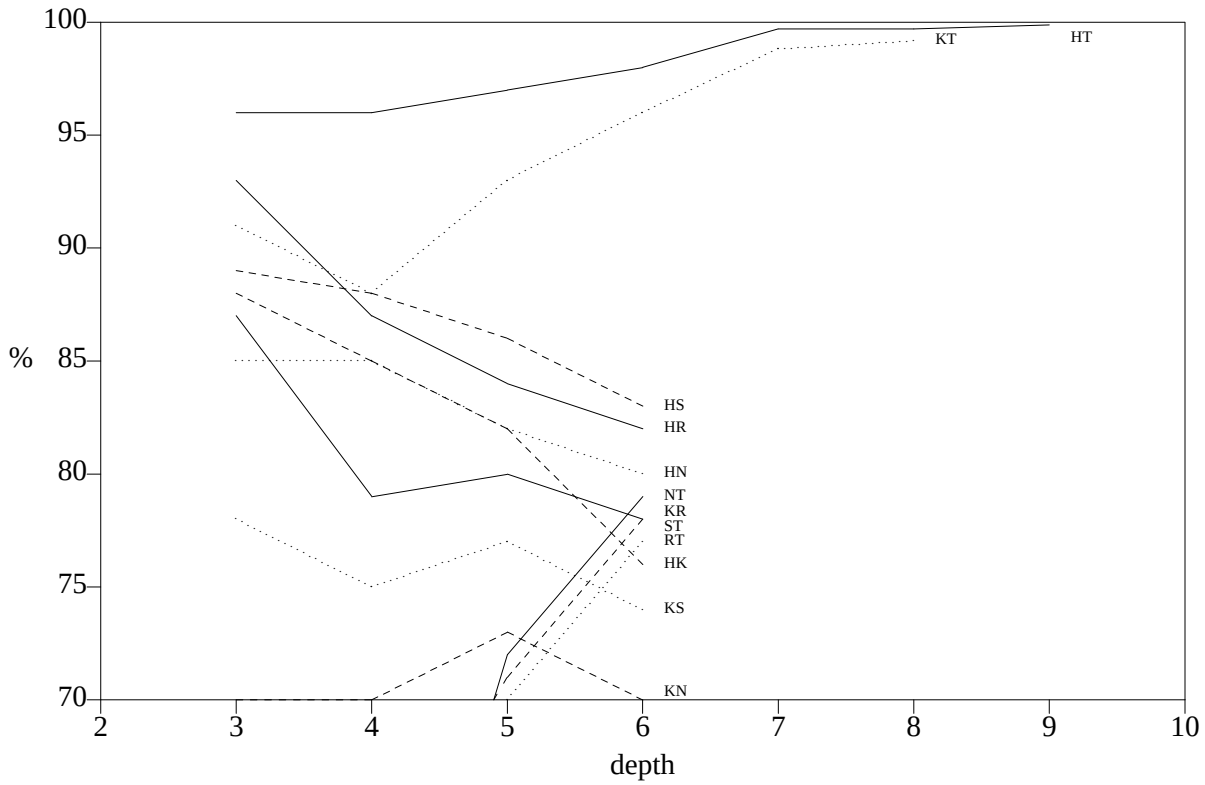

Fig. 4. Two Search Enhancements (NBP).

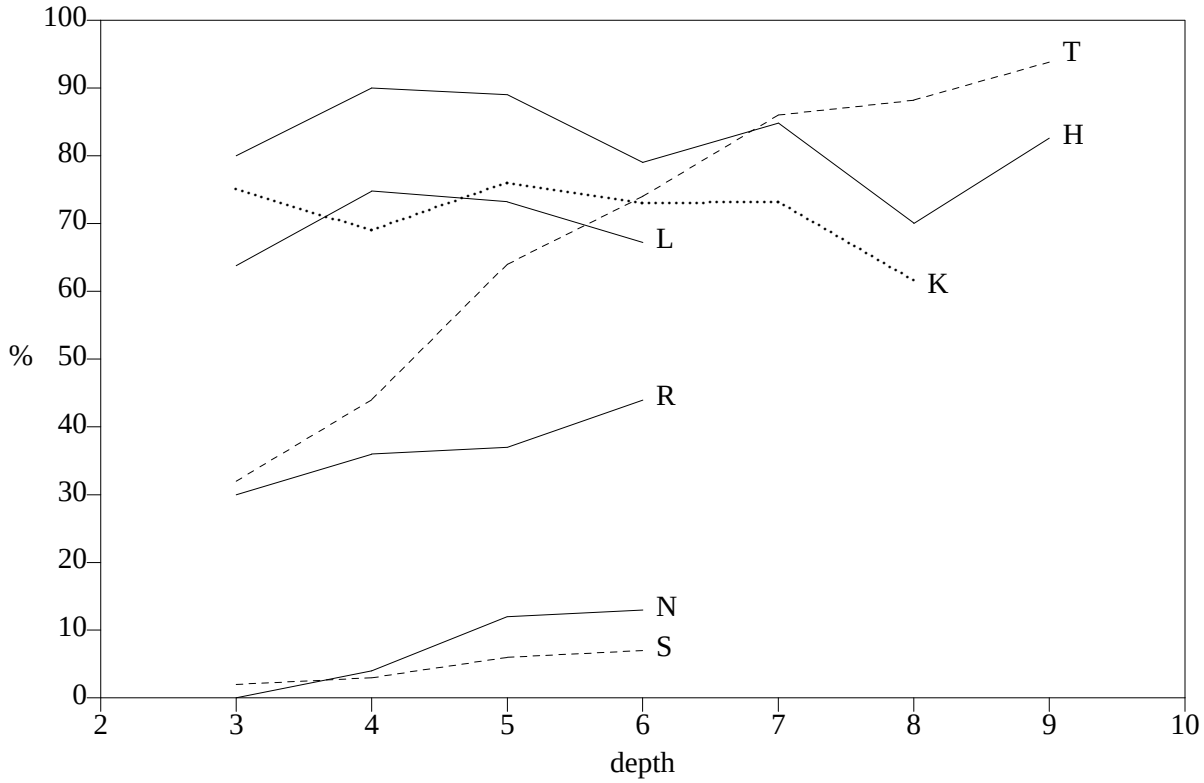

Fig. 5. Single Search Enhancement (NC).

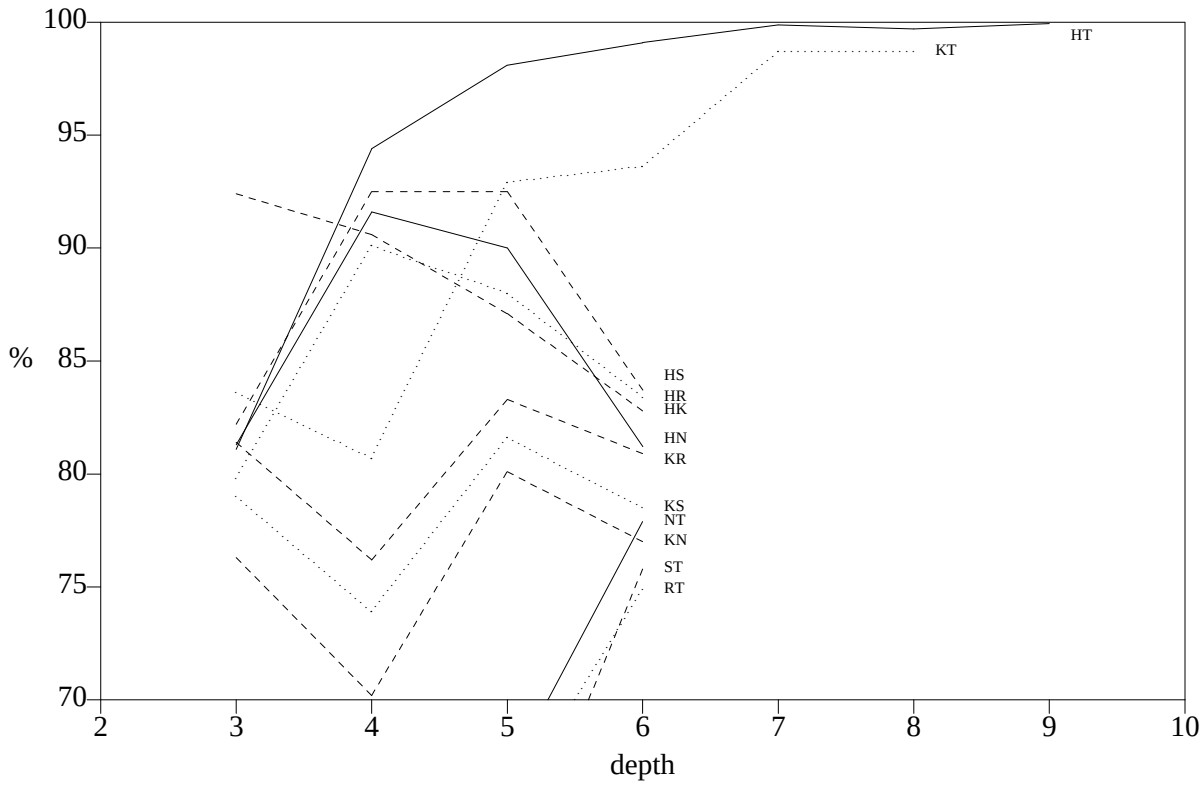

Fig. 6. Two Search Enhancements (NC).

Fig. 1. The History Heuristic and Alpha-Beta.

Fig. 2. History Heuristic Example.

Fig. 3. Single Search Enhancement (NBP).

Fig. 4. Two Search Enhancements (NBP).

Fig. 5. Single Search Enhancement (NC).

Fig. 6. Two Search Enhancements (NC).