# An Algorithm Faster than NegaScout and SSS* in Practice*

Aske Plaat, Erasmus University, *plaat@theory.lcs.mit.edu*
Jonathan Schaeffer, University of Alberta, *jonathan@cs.ualberta.ca*
Wim Pijls, Erasmus University, *whlmp@cs.few.eur.nl*
Arie de Bruin, Erasmus University, *arie@cs.few.eur.nl*

Erasmus University,
Department of Computer Science,
Room H4-31, P.O. Box 1738,
3000 DR Rotterdam,
The Netherlands

University of Alberta,
Department of Computing Science,
615 General Services Building,
Edmonton, Alberta,
Canada T6G 2H1

**Abstract**

In this paper we introduce a framework for best first search of minimax trees. Existing best first algorithms like SSS* and DUAL* are formulated as instances of this framework. The framework is built around the Alpha-Beta procedure. Its instances are highly practical, and readily implementable. Our reformulations of SSS* and DUAL* solve the perceived drawbacks of these algorithms. We prove their suitability for practical use by presenting test results with a tournament level chess program.

In addition to reformulating old best first algorithms, we introduce an improved instance of the framework: MTD($f$). This new algorithm outperforms NegaScout, the current algorithm of choice of most chess programs. Again, these are not simulation results, but results of tests with an actual chess program, Phoenix.

## 1 Introduction

Over the last thirty years most practitioners have used depth-first search algorithms like Alpha-Beta in their game playing programs. Since the introduction of Alpha-Beta, numerous enhancements have been introduced to reduce the search space of game playing programs, improving the performance tremendously. Among the enhancements are: iterative deepening, transposition tables, the history heuristic, and minimal search windows (see for example [28] for an assessment). Whereas in other fields best first approaches have been successful, in minimax search these have been largely ignored by practitioners, the reason being the good results of enhanced depth first searches, and the reportedly excessive memory usage of best first searches.

In this paper we will show that best first search can be implemented using a minimal window depth first search procedure, solving numerous reported problems, creating a number of practical best first algorithms. In particular, we describe the following additions to our understanding of depth first and best first minimax search:

- Null window search, with a form of storage, can be used to construct best first minimax algorithms. For storage a conventional transposition table can be used. The null window calls generate a sequence of bounds on the minimax value. The storage contains the part of the search tree that establishes these bounds, to be refined in subsequent passes.

*This paper is submitted for publication. Some of these results were originally published in [22].

- A framework is presented for algorithms that generate such sequences of bounds in different ways. Interestingly, certain instances of this framework expand the same leaf nodes in the same order as SSS* and DUAL*. These algorithms, called AB-SSS* and AB-DUAL*, are much simpler than the original formulations: they consist of a single loop of Alpha-Beta calls.
- To circumvent the problems of artifically generating realistic trees, we have used a real chess playing program for our tests. Using the Alpha-Beta based framework, both depth first and best first algorithms are given the same storage, in contrast to previous comparisons. The results confirm that AB-SSS* and AB-DUAL* are practical algorithms, contradicting previous publications [8, 12, 26].
- We have tested instances of the framework against the depth first algorithm implemented in our chess program, i.e. Informed NegaScout with aspiration windows, representing the current choice of the game programming community. One instance, MTD($f$), outperforms NegaScout on leaf node count, total node count, and execution time, by a wider margin than NegaScout's gain over Alpha-Beta. We provide some experimental evidence as to why MTD($f$) performs so well.
- Interestingly, all the tested algorithms perform relatively close together, much closer than previous simulation results have indicated. We conclude that, a) artificially generated game trees often do not capture all essential aspects of real trees, and b) often more gains are obtained from the so-called Alpha-Beta enhancements, than from better algorithms.

# 2 Null Window Search and Memory

## 2.1 Null Window Search

In the Alpha-Beta procedure, a node is searched with a search window of a certain size. A narrow search window $\langle a, b\rangle$ causes more nodes to be cutoff than a wide window $\langle c, d\rangle$ (where $c < a$ and $d > b$). The narrowest window possible—the one causing the most cutoffs—is the null window, where $\alpha = \beta - 1$ (assuming integer valued leaves). Many people have noted that null-window search has a great potential for efficient search algorithms [1, 3, 4, 6, 14, 27]. The widely used NegaScout algorithm derives its superiority over Alpha-Beta from null-window search. Because it has the potential of creating very efficient algorithms, a number of algorithms have been proposed that are solely based on null window search. For example, Coplan presented the C* algorithm [4, 30], and Schaeffer and Fishburn have experimented with Alpha Bounding [6, 27].

Knuth and Moore have shown that the return value $g$ of an Alpha-Beta search with window $\langle\alpha, \beta\rangle$ can be one of three things [9]:

1. $\alpha < g < \beta$. $g$ is equal to the minimax value $f$ of the game tree $G$.
2. $g \leq \alpha$ (failing low). $g$ is an upper bound on the minimax value $f$ of $G$, or $f \leq g$.
3. $g \geq \beta$ (failing high). $g$ is a lower bound on the minimax value $f$ of $G$, $f \geq g$.

We see that a single null window search will never return the true minimax value $f$, but only a bound on it (which may happen to coincide with $f$, but this cannot be inferred from the result of the null window call). A fail low results in an upper bound, denoted by $f^+$. A fail high returns a lower bound, denoted by $f^-$. Algorithms like C* and Alpha Bounding use multiple null window calls to generate bounds to home in on the minimax value. A potential problem with these repetitive calls is that they re-expand previously searched nodes. For NegaScout it appears that the gains of the tighter bounds out-weigh the costs of re-expansions, compared to a single wide-window Alpha-Beta call [16].

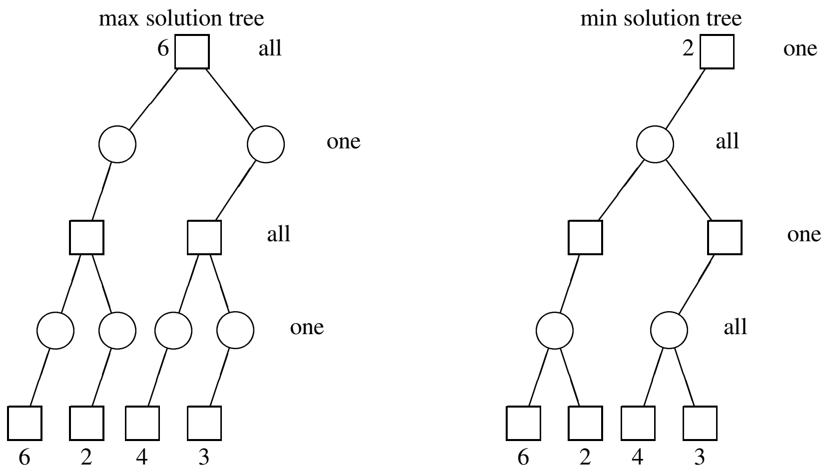


Figure 1: Solution Trees.

## 2.2 Memory

An idea to prevent the re-expansion of previously searched nodes is to store them in memory. It is often said that since minimax trees are of exponential size, this approach is infeasible since it needs exponentially growing amounts of memory. We will show in this paper that although the search information that must be stored is indeed of exponential size, it is certainly less than what is often assumed. For today's search depths it fits comfortably in today's memories. Projections of tomorrow's search depths and memory sizes predict that this situation will persist in the foreseeable future.

The information returned by an Alpha-Beta call is either a bound or the minimax value. Knuth and Moore have shown that the essential part of the search tree that proves the minimax value is the so called *minimal tree*. For a minimax tree of uniform width $w$ and depth $d$, it has $w^{\lfloor d/2 \rfloor} + w^{\lceil d/2 \rceil} - 1$ leaves, or, its size is $O(w^{\lceil d/2 \rceil})$. If Alpha-Beta returns an upper bound, then its value is defined by a so called max solution tree, in a sense one half of a minimal tree, of size $O(w^{\lceil d/2 \rceil})$. If Alpha-Beta returns a lower bound, then its value is defined by a min solution tree, the other half of a minimal tree, of size $O(w^{\lfloor d/2 \rfloor})$. The (theoretical) background for these statements can be found in [5, 10, 21, 29]. Figure 1 shows a max and a min solution tree.

Since recursive null window Alpha-Beta calls return only bounds, storing the previous search results comes down to storing a max or a min solution tree. Although not exactly small for the case of chess, assuming a search depth of 8 or 9 and a branching factor of 36, the numbers are not completely out of reach of todays technology. In addition, these are numbers for (uniform) trees. Real games usually have transpositions, that can reduce the search space tremendously [19].

Here we arrive at an important decision: to decide on a method that is suited to store the information of previous searches. For game playing programs an obvious choice is the transposition table. According to the literature, it is used for two reasons in current programs [11]. The foremost reason is to increase the move ordering in conjunction with iterative deepening, to increase the number of Alpha-Beta cutoffs. The second reason is to prevent re-expansion of nodes that have already been searched, due to transpositions in the search space.

For algorithms using multiple (null-window) passes, or re-searches, a third reason can be added: to prevent the re-expansion of nodes searched in a previous pass. The only

```
/* Transposition table (TT) enhanced Alpha-Beta */
function Alpha-Beta(n, α, β) → g;
    /* Check if position is in TT and has been searched to sufficient depth */
    if retrieve(n) = found then
        if n.f⁺ ≤ α or n.f⁺ = n.f⁻ then return n.f⁺;
        if n.f⁻ ≥ β then return n.f⁻;
    /* Reached the maximum search depth */
    if n = leaf then
        n.f⁻ := n.f⁺ := g := eval(n);
    else
        g := −∞;
        c := firstchild(n);
        /* Search until a cutoff occurs or all children have been considered */
        while g < β and c ≠ ⊥ do
            g := max(g, −Alpha-Beta(c, −β, −α));
            α := max(α, g);
            c := nextbrother(c);
        /* Save in transposition table */
        if g ≤ α then n.f⁺ := g;
        if α < g < β then n.f⁺ := n.f⁻ := g;
        if g ≥ β then n.f⁻ := g;
    store(n);
    return g;
```

Figure 2: Alpha-Beta for use with Transposition Tables.

difference with the second reason is that the table entries are not caused by transpositions, but by a previous search pass; there is no difference as far as the algorithm is concerned. The basic idea is that it is possible to store the search tree as known from more theoretical work [7, 10, 17, 29] in the transposition table. Some background explaining in greater detail why the transposition table is a suitable structure for storing search or solution trees, and why it gives correct results in algorithms doing repetitive null window searches, can be found in [5, 21].

# 3 MT: Memory enhanced Test

The concept of null window search was introduced by Pearl, who called the procedure Test [15]. For that reason, we have named our memory enhanced version MT, for memory enhanced Test. (Our Test returns a bound, not just a boolean value. This version is sometimes called *informed* Test). The code for MT, in its Alpha-Beta form, can be found in figure 2. In this paper we will interchangeably use the terms MT and Alpha-Beta procedure (as opposed to *algorithm*). (Storing both bounds at the same time is necessary for some algorithms, while for others it improves the efficiency only slightly [21]. If the algorithm does not need to store two solution trees at the same time, then in practice the extra memory needed is considered too much.)

So far, we have discussed the following two mechanisms to be used in building efficient algorithms: (1) null window searches cutoff more nodes than wide search windows, and (2) we can use transposition tables to glue multiple passes of null window calls together, so that they can be used to home in on the minimax value, without re-expanding nodes searched in previous passes. We can use these building blocks to construct a multitude of different algorithms. One option is to construct drivers that repeatedly call MT at the root of the game tree. Three of these MT-drivers are shown in the figures 3, 5, and 6. The

**function** AB-SSS*($n$) $\rightarrow f$;
  $g := +\infty$;
  **repeat**
    $\gamma := g$;
    $g :=$ Alpha-Beta($n, \gamma - 1, \gamma$);
  **until** $g = \gamma$;
  **return** $g$;

Figure 3: SSS* as a Sequence of memory enhanced Alpha-Beta Searches.

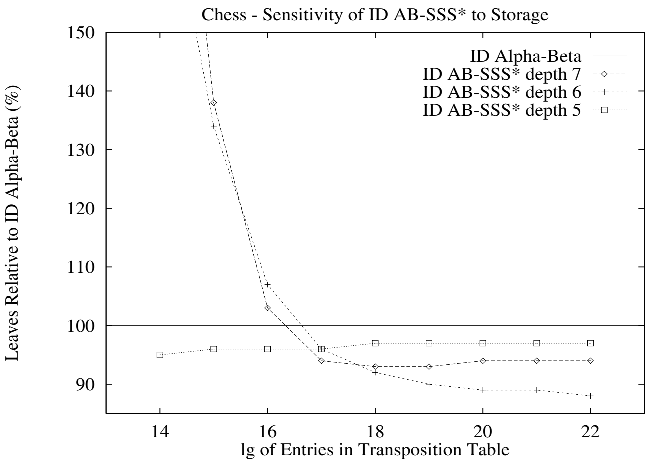


Figure 4: Leaf node count ID AB-SSS*.

drivers differ in the way the null window is chosen (denoted by $\gamma$ in the figures).

## 3.1 SSS*

The driver in figure 3 constructs an algorithm that starts with an upper bound of $+\infty$. From Alpha-Beta's postcondition we see that this call will fail low, yielding an upper bound. By feeding this upper bound $f^+$ again to an MT call, we will get a sequence of fail lows. In the end, if $g = \gamma$, we will have a fail high with $g = f^- = \gamma = f^+$, which means the game value $f$ is found. To compute the next upper bound, AB-SSS* refines a max solution tree [20, 17, 18].

This driver expands the same leaf nodes in the same order as Stockman's SSS* [18, 21]. In this sense, we have constructed an equivalent formulation of SSS*, constructing a best first algorithm using depth first, memory enhanced, search. Formulating SSS* as a sequence of null window searches makes clear *why* it searches less leaves than a single wide window Alpha-Beta call, as has been proven by Stockman: because of the smaller search windows.

The new formulation has a number of practical advantages over the old Stockman formulation. The biggest advantage is that this formulation is readily implementable in a regular Alpha-Beta based chess program. This enables us to finally test the performance of SSS* in a real application. Section 4 gives these test results.

Another quite interesting test is to see how much storage is needed to store the re-search information—is it indeed of the order of the max solution tree?—in the transposition table. Of the three uses of the transposition table—better move ordering, no re-expansions through transpositions, and no re-expansions from previous passes—we are interested in

```
function AB-DUAL*(n) → f;
    g := −∞;
    repeat
        γ := g;
        g := Alpha-Beta(n, γ, γ + 1);
    until g = γ;
    return g;
```

Figure 5: DUAL* as a Sequence of memory enhanced Alpha-Beta Searches.

the third one. That would show us whether the assumption of section 2.2, that the re-search information fits in memory, is valid. Therefore, we have conducted tests with a variable size transposition table, and show the number of expanded nodes for a multi-pass algorithm divided by the nodes expanded by a one-pass algorithm. To put it differently, how much memory does AB-SSS* need to beat Alpha-Beta? Figure 4 shows the test results for Phoenix. The graph shows the cumulative number of leaf node evaluations for iterative deepening versions of AB-SSS* and Alpha-Beta. The $x$-axis gives the number of entries in the table in powers of 2. As will be explained in section 4.1, these numbers are averaged over a test-set of 20 positions chosen to be representative for real games.

For small tables, the solution tree does not fit in memory, and has to be re-expanded for each pass of AB-SSS*. As soon as the storage reaches a critical level and essentially the max solution tree fits in memory, AB-SSS*'s tree size stabilizes, and becomes better than Alpha-Beta. Since the line is more or less straight, it appears that after a certain size, hash key collisions play no measurable role.

Many reseachers have conjectured that best-first algorithms such as SSS* would need too much memory too be practical alternatives for depth first algorithms like Alpha-Beta [29, 8, 12, 13, 23, 26]. The tests with our reformulation of SSS* prove otherwise. For present-day search depths in applications like chess playing programs, using present-day memory sizes, and conventional transposition tables as storage structure, we see that SSS*'s search trees fit without problems in the available memory. For most real-world programs a transposition table size of 10 Megabyte will be more than adequate for AB-SSS*.

The literature cites three main drawbacks of SSS*: it is hard to understand, it performs operations on a sorted list that are slow, and it uses too much memory to be practical. We think that our reformulation as a sequence of null window Alpha-Beta calls is easy to understand. The slow list operations are traded in for hash table lookups that are as fast as for Alpha-Beta, and the experiments show that AB-SSS* does not need too much memory. We conclude that the drawbacks of SSS* are solved in the new formulation.

## 3.2 DUAL*

A dual version of SSS*, aptly named DUAL*, can be created by inverting SSS*'s operations: use an ascendingly sorted list instead of descending, swap max and min operations, and start at $-\infty$ instead of $+\infty$ [12, 23].

To demonstrate the power of the MT driver framework, we show the reformulation called AB-DUAL* in figure 5. The only difference with AB-SSS* is the initialization of the bound to $-\infty$, and a change to the way Alpha-Beta is called. This reformulation focusses attention on one item only: the bound starts at the bottom of the scale, implying that the only fundamental difference between SSS* and DUAL* is that upper bounds are replaced by lower bounds (which implies that the max solution tree that is refined by AB-SSS* has become a min solution tree in AB-DUAL*). All other differences are apparently insubstantial, since Alpha-Beta does not have to be changed. (That Alpha-Beta can be used to return both upper and lower bounds, can be seen from case (2) and (3) of its postcondition.

```
function MTD-f(n, g) → f;
    f⁺ := +∞; f⁻ := −∞;
    repeat
        if g = f⁻ then γ := g + 1 else γ := g;
        g := Alpha-Beta(n, γ − 1, γ);
        if g < γ then f⁺ := g else f⁻ := g;
    until f⁻ = f⁺;
    return g;
```

Figure 6: MTD($f$), a Better Sequence of MT searches.

That it can be used to refine both max and min solution trees is shown in [5, 21].)

### 3.3 Other Options for the choice of Start Value

Having seen that MT calls are flexible enough to be started off with the two extremes of the scale, and having seen that MT can be used to refine both upper and lower bounds, we can try other values. An inuitively appealing option is to choose a start value closer to the expected outcome. One option is to keep bisecting the interval between the upper and lower bound, to reduce the number of MT calls. This idea has been described in [4, 20, 30]. Another idea is to use a heuristic guess as the start value. In an iterative deepening framework it is natural to use the score from the previous iteration for this purpose, since it is expected to be a close approximation of the score for the current depth. We have called this MT driver MTD($f$). The pseudo code is shown in figure 6. The first call acts to decide which way the search will go. If it is a fail high, MTD($f$) will behave like AB-DUAL*, and keep increasing the lower bound returned by MT. If the first call fails low, MTD($f$) will, like AB-SSS*, decrease the upper bound until the minimax value is reached. AB-SSS* starts off optimistic, AB-DUAL* starts off pessimistic, and MTD($f$) starts off in the middle, possibly realistic.

One of the drawbacks of AB-SSS* and AB-DUAL* is the high number of calls to MT that it takes them to converge on the minimax value. Most of the MT calls make small improvements to the bound, causing the process to continue in small steps. By starting closer to the minimax value, many intermediate MT calls are skipped. MTD($f$) takes one big leap to come close to the minimax value, dramatically dropping the number of intermediate MT calls. The lower number of calls has the advantage that MTD($f$) performs relatively better in constrained memory than SSS*, since there are much less re-expansions. Measurements confirm that MT typically gets called 3 to 6 times in MTD($f$). In contrast, the AB-SSS* and AB-DUAL* results are poor compared to Aspiration NegaScout when all nodes in the search tree are considered. Each of these algorithms performs dozens and often even hundreds of MT searches. The wider the range of leaf values, the smaller the steps with which they converge, and the more re-searches they need.

# 4 Performance

To assess the feasibility of the proposed algorithms, a series of experiments was performed. We present data for the comparison of Alpha-Beta, Aspiration NegaScout, AB-SSS*, AB-DUAL*, and MTD($f$).

### 4.1 Experiment Design

We will assess the performance of the algorithms by counting leaf nodes and total nodes. For two algorithms we also provide data for execution time. This metric may vary considerably for different programs. It is nevertheless included, to give evidence of the potential of MTD($f$).

We have tried to come as close to real life applications of our algorithms as possible by conducting the experiments with a tournament-quality game playing program.

All algorithms used iterative deepening. They are repeatedly called with successively deeper search depths. All three algorithms use a standard transposition table with a maximum of $2^{21}$ entries; the tests from section 3.1 showing that the solution trees could comfortably fit in tables of this size, without any risk of noise due to collisions. For our experiments we used the original program author's transposition table data structures and code without modification.[1] At an interior node, the move suggested by the transposition table is always searched first (if known), and the remaining moves are ordered before being searched. Phoenix uses dynamic ordering based on the history heuristic [28].

Conventional test sets in the literature proved to be inadequate to model real-life conditions. Positions in test sets are usually selected to test a particular characteristic or property of the game (such as tactical combinations in chess) and are not necessarily indicative of typical game conditions. For our experiments, the algorithms were tested using a set of 20 positions that corresponded to move sequences from tournament games. By selecting move sequences rather than isolated positions, we are attempting to create a test set that is representative of real game search properties (including positions with obvious moves, hard moves, positional moves, tactical moves, different game phases, etc.). A number of test runs was performed on a bigger test set and to a higher search depth to check that the 20 positions did not contain anomalies.

Many papers in the literature use Alpha-Beta as the base-line for comparing the performance of other algorithms (for example, [3, 11]). The implication is that this is the standard data point which everyone is trying to beat. However, game-playing programs have evolved beyond simple Alpha-Beta algorithms. Most use Alpha-Beta enhanced with minimal window search (NegaScout), iterative deepening, transposition tables, move ordering and an initial aspiration window. Since this is the typical search algorithm used in high-performance programs (such as Phoenix), it seems more reasonable to use this as our base-line.

The worse the base-line comparison algorithm chosen, the better other algorithms appear to be. By choosing NegaScout enhanced with aspiration searching [2] (Aspiration NegaScout) as our performance metric, and giving it a transposition table big enough to contain all re-search information, we are emphasizing that it is possible to do better than the "best" methods currently practiced and that, contrary to published simulation results, some methods—notably SSS*—will turn out to be inferior.

Because we implemented the MTD algorithms (like AB-SSS* and AB-DUAL*) using MT (null-window Alpha-Beta calls with a transposition table) we were able to compare a number of algorithms that were previously seen as very different. By using MT as a common proof-procedure, every algorithm benefited from the same enhancements concerning iterative deepening, transposition tables, and move ordering code. To our knowledge this is the first comparison of depth-first and best-first minimax search algorithms where all the algorithms are given identical resources. A consequence is that our base line, Aspiration NegaScout, becomes through the use of transposition tables big enough to store solution trees, in effect *informed* Aspiration NegaScout, cf. [25].

## 4.2 Results

Figure 7 shows the performance of Phoenix, using the number of leaf evaluations (NBP or Number of Bottom Positions) as the performance metric. Figure 8 shows the performance of the algorithms using the number of nodes in the search tree (interior and leaf, including

[1] As a matter of fact, since we implemented MT using null-window alpha-beta searches, we did not have to make any changes at all to the code other than the disabling of forward pruning and bound-dependent search extensions (to keep our search deterministic). We only had to introduce the MTD driver code.

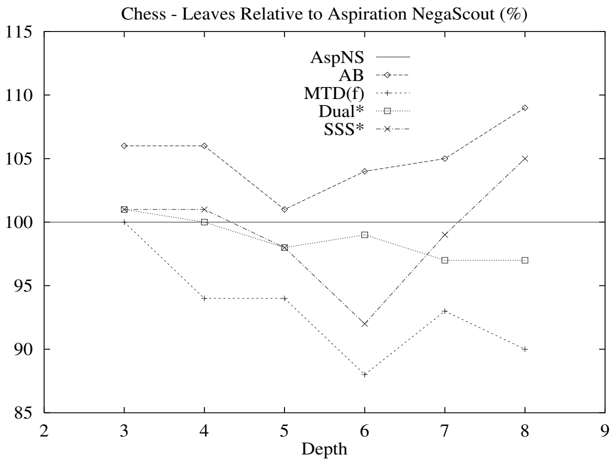


Figure 7: Leaf node count

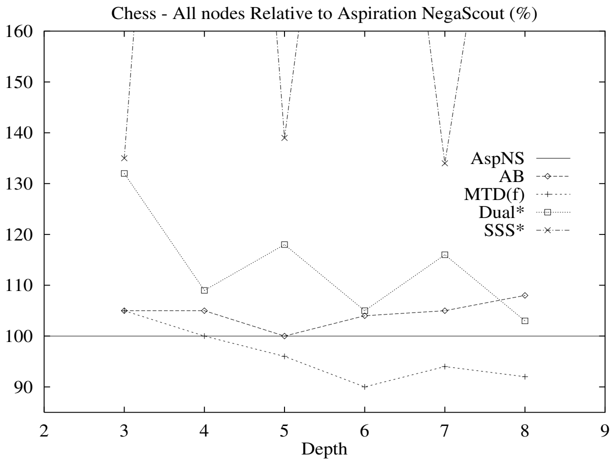


Figure 8: Total node count

nodes that caused transposition cutoffs) as the metric. The graphs show the cumulative number of nodes over all previous iterations for a certain depth (which is realistic since iterative deepening is used) relative to Aspiration NegaScout.

The bottom line for practitioners is execution time. We only show execution time graphs for ID MTD($f$) and ID Aspiration NegaScout (figure 9), the comparison that we think is the most interesting, since comparing results for the same machines we found that MTD($f$) is consistently the fastest algorithm. We only show the deeper searches, since the relatively fast shallower searches hamper accurate timings. The run shown is a typical example run on a Sun SPARC. We did experience different timings when running on different machines. It may well be that cache size plays an important role, and that tuning of the program can have a considerable impact.

In our experiments we found that for Phoenix MTD($f$) was about 9–13% faster in execution time than Informed Aspiration NegaScout. For other programs and other machines these results will obviously differ, depending in part on the quality of the score of the previous iteration, and on the test-positions used. Also, since the tested algorithms perform quite close together, the relative differences are quite sensitive to variations in input parameters.

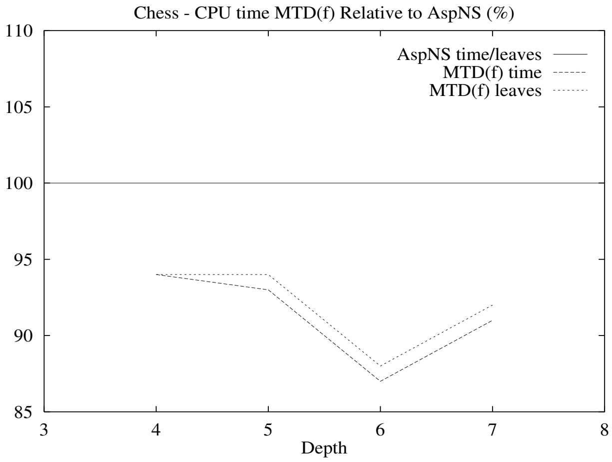


Figure 9: Execution time

In generalizing these results, one should keep this sensitivity in mind. Using these numbers as absolute predictors for other situations would not do justice to the complexities of real life game trees. The experimental data is better suited to provide insight on, or guide and verify hypotheses about these complexities.

### 4.2.1 Aspiration NegaScout and MTD($f$)

The results show that Aspiration NegaScout is better than Alpha-Beta. This result is consistent with [28] which showed Aspiration NegaScout to be a small improvement over Alpha-Beta when transposition tables and iterative deepening were used.

MTD($f$) consists solely of null window searches. In each pass, the previous search results is used to select the "best" node. The majority of the searches in NegaScout is also performed with a null window. An important difference is with which $\gamma$ this null window search is performed. NegaScout derives it from the tree itself, whereas MTD($f$) relies for the first guess on information from outside the tree. (In our experiments the minimax value from a previous iterative deepening iteration was used for this purpose.)

The best results are from MTD($f$), although the current algorithm of choice by the game programming community, Aspiration NegaScout, performs very well too. The averaged MTD($f$) leaf node counts are consistently better than for Aspiration NegaScout, averaging at around a 10% improvement for Phoenix. More surprisingly is that MTD($f$) outperforms Aspiration NegaScout on the total node measure as well. Since each iteration requires repeated calls to MT (at least two and possibly many more), one might expect MTD($f$) to perform badly by this measure because of the repeated traversals of the tree. This suggests that MTD($f$), on average, is calling MT close to the minimum number of times.

### 4.2.2 Start Value and Search Effort

Perhaps the biggest difference in the MTD algorithms is their first approximation of the minimax value: AB-SSS* is optimistic, AB-DUAL* is pessimistic and MTD($f$) is realistic. It is clear that starting close to $f$, assuming integer valued leaves, should generally result in converging in less steps, simply because there are less values in the range from the start value to $f$.

If each MT call expands roughly the same number of nodes, then doing less passes yields a better algorithm. Since we could not find by analytical means whether this is true, we have conducted experiments to test the intuitively appealing idea that starting a search close to $f$ is cheaper than starting far away.

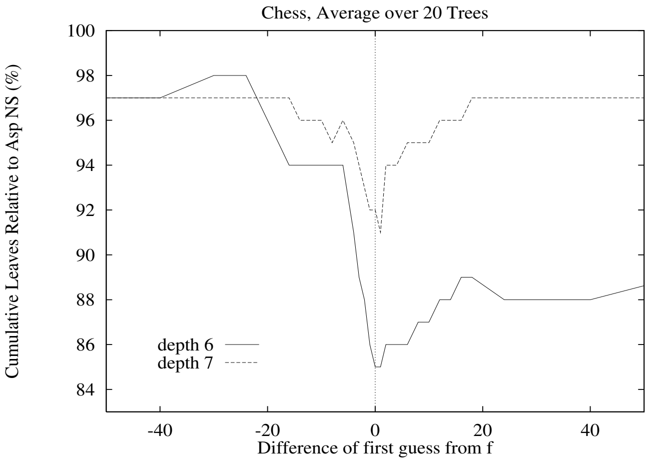


Figure 10: Tree size relative to the first guess $f$.

Figure 10 validates the choice of a starting parameter close to the game value. The figure shows the percentage of unique leaf evaluations of iteratively deepened calls to a special version of MTD($f$) where the first guess was deliberately distorted. It was set at a certain value some points off the minimax value for each search depth (plotted on the $x$-axis). In this way we obtain a picture of the search tree size for versions of MTD($f$) where the first guess is of different quality. The data points are given as a percentage of the size of the search tree built Aspiration NegaScout. To be better than Aspiration NegaScout, an algorithm must be less than the 100% baseline. (Since iterative deepening algorithms are used, the cumulative leaf count over all previous depths is shown for the depths.) Given an initial guess of $h$ and the minimax value of $f$, the graph plots the search effort expended for different values of $h - f$. To the left of the graph, MTD($f$) is closer to AB-DUAL*, to the right it is closer to AB-SSS*.

It appears that the smaller the distortion, the smaller the search tree is. Our intuition that starting close to the minimax value is a good idea, is justified by these experiments. A first guess close to $f$ makes MTD($f$) perform better than the 100% Aspiration NegaScout baseline. We also see that the guess must be quite close to $f$ for the effect to become significant. Thus, if MTD($f$) is to be effective, the $f$ obtained from the previous iteration must be a good indicator of the next iteration's value.

What is the minimum amount of search effort that must be done to establish the minimax value of the search tree? If we know the value of the search tree is $f$, then two searches are required: MT($f - 1, f$), which fails high establishing a lower bound on $f$, and MT($f, f + 1$), which fails low and establishes an upper bound on $f$. Of the algorithms presented, MTD($f$) has the greatest likelihood of doing this minimum amount of work, since it uses the previous iteration to provide a good guess for $f$. We see in the graphs in figure 7 that MTD($f$) is not performing as good as the lowest points in figure 10; there remains room for improving the performance through better first guesses for each depth.

As the graphs in figure 10 indicate, the performance of MTD($f$) is dependent on the quality of the score that is used as the first-guess. For programs with a pronounced odd/even oscillation in their score, results are better if not the score of the previous iterative deepening pass is used, but the one from 2 iterations ago. Considering all this, it is not a surprise that both DUAL* and SSS* come our poorly. Their initial bounds for the minimax value are generally poor ($-\infty$ and $+\infty$ respectively), meaning that the many calls to MT result in significantly more interior nodes.

#### 4.2.3 SSS* and DUAL*

Contrary to many simulations, our results show that the difference in the number of leaves expanded by SSS* and Alpha-Beta is relatively small. Since game-playing programs use many search enhancements, the benefits of a best-first search are greatly reduced. We conclude that in practice, AB-SSS* is a small improvement on Alpha-Beta (depending on the branching factor). Claims that SSS* and DUAL* evaluate significantly fewer leaf nodes than Alpha-Beta are based on simplifying assumptions that have little relation with what is used in practice.

Several simulations have pointed out the complementary behavior of SSS* and DUAL* for odd/even search depths. Both algorithms are said to significantly out-search Alpha-Beta, but which one is superior depends on the parity of the search depth. In our experiments, this effect is less pronounced. The graphs indicate that AB-DUAL* is slightly superior for small branching factors, probably since min solution trees are smaller than max solution trees. This difference decreases as the branching factor increases. For chess, AB-SSS* and AB-DUAL* perform comparably, contradicting the literature [12, 23, 24].

## 5 Conclusions

Over thirty years of research have been devoted to improving the efficiency of Alpha-Beta searching. The MT family of algorithms are comparatively new, without the benefit of intense investigations. Yet, MTD($f$) is already out-performing our best Alpha-Beta based implementations in real game-playing programs. MT is a simple and elegant paradigm for high performance game-tree search algorithms. It makes it possible to eliminate all perceived drawbacks of SSS* in practice.

The purpose of a simulation is to reliably model an algorithm to gain insight into its performance. Simulations are usually performed when it is too difficult or too expensive to construct the proper experimental environment. For game-tree searching, the case for simulations is weak. There is no need to do simulations when there are quality game-playing programs available for obtaining actual data. Further, as this paper has demonstrated, simulation parameters can be incorrect, resulting in large errors in the results that lead to misleading conclusions. In particular, the failure to include iterative deepening and transposition tables in many simulations are serious omissions.

Although MTD($f$)'s 10% improvement for Chess may not seem much, it comes at no extra algorithmic complexity: just a standard Alpha-Beta-based Chess program plus one while-loop. In addition, the improvement is bigger than the difference between Aspiration NegaScout and Alpha-Beta. In the light of the high level of search space reduction achieved in tournament quality chess programs, 10% is a sizable improvement.

Binary-valued searches enhanced with iterative deepening, transposition tables and the history heuristic is an efficient search method that uses no explicit knowledge of the application domain. It is remarkable that one can search almost perfectly without explicitly using application-dependent knowledge other than the evaluation function.

## Acknowledgements

This work has benefited from discussions with Mark Brockington, Yngvi Bjornsson and Andreas Junghanns. The financial support of the Netherlands Organization for Scientific Research (NWO), the Tinbergen Institute, the Natural Sciences and Engineering Research Council of Canada (NSERC grant OGP-5183) and the University of Alberta Central Research Fund are gratefully acknowledged.

## References


[1] L. Victor Allis, Maarten van der Meulen, and H. Jaap van den Herik. Proof-number search. *Artificial Intelligence*, 66:91–124, March 1994.

[2] Gérard M. Baudet. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1978.

[3] Murray S. Campbell and T. Anthony Marsland. A comparison of minimax tree search algorithms. *Artificial Intelligence*, 20:347–367, 1983.

[4] K. Coplan. A special-purpose machine for an improved search algorithm for deep chess combinations. In M.R.B. Clarke, editor, *Advances in Computer Chess 3, April 1981*, pages 25–43. Pergamon Press, Oxford, 1982.

[5] Arie de Bruin, Wim Pijls, and Aske Plaat. Solution trees as a basis for game-tree search. *ICCA Journal*, 17(4):207–219, December 1994.

[6] John P. Fishburn. *Analysis of Speedup in Distributed Algorithms*. PhD thesis, University of Wisconsin, Madison, 1981.

[7] Toshihide Ibaraki. Generalization of alpha-beta and SSS* search procedures. *Artificial Intelligence*, 29:73–117, 1986.

[8] Hermann Kaindl, Reza Shams, and Helmut Horacek. Minimax search algorithms with and without aspiration windows. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-13(12):1225–1235, December 1991.

[9] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[10] Vipin Kumar and Laveen N. Kanal. A general branch and bound formulation for and/or graph and game tree search. In *Search in Artificial Intelligence*. Springer Verlag, 1988.

[11] T. Anthony Marsland. A review of game-tree pruning. *ICCA Journal*, 9(1):3–19, March 1986.

[12] T. Anthony Marsland, Alexander Reinefeld, and Jonathan Schaeffer. Low overhead alternatives to SSS*. *Artificial Intelligence*, 31:185–199, 1987.

[13] Agata Muszycka and Rajjan Shinghal. An empirical comparison of pruning strategies in game trees. *IEEE Transactions on Systems, Man and Cybernetics*, 15(3):389–399, May/June 1985.

[14] Judea Pearl. Asymptotical properties of minimax trees and game searching procedures. *Artificial Intelligence*, 14(2):113–138, 1980.

[15] Judea Pearl. The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. *Communications of the ACM*, 25(8):559–564, August 1982.

[16] Judea Pearl. *Heuristics – Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Publishing Co., Reading, MA, 1984.

[17] Wim Pijls and Arie de Bruin. Another view on the SSS* algorithm. In T. Asano, T. Ibaraki, H. Imai, and T. Nishizeki, editors, *Algorithms, SIGAL '90, Tokyo*, volume 450 of *LNCS*, pages 211–220. Springer-Verlag, August 1990.

[18] Wim Pijls, Arie de Bruin, and Aske Plaat. Solution trees as a unifying concept for game tree algorithms. Technical Report EUR-CS-95-01, Erasmus University, Department of Computer Science, P.O. Box 1738, 3000 DR Rotterdam, The Netherlands, April 1995.

[19] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. Nearly optimal minimax tree search? Technical Report TR-CS-94-19, Department of Computing Science, University of Alberta, Edmonton, AB, Canada, December 1994.

[20] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. A new paradigm for minimax search. Technical Report TR-CS-94-18, Department of Computing Science, University of Alberta, Edmonton, AB, Canada, December 1994.

[21] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. SSS* = $\alpha$-$\beta$ + TT. Technical Report TR-CS-94-17, Department of Computing Science, University of Alberta, Edmonton, AB, Canada, December 1994.

[22] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. Best-first fixed-depth game-tree search in practice. In *IJCAI-95*. International Joint Conference on Artificial Intelligence, August 1995. To Appear.

[23] Alexander Reinefeld. *Spielbaum Suchverfahren*. Volume Informatik-Fachberichte 200. Springer Verlag, 1989.

[24] Alexander Reinefeld and Peter Ridinger. Time-efficient state space search. *Artificial Intelligence*, 71(2):397–408, 1994.

[25] Alexander Reinefeld, Jonathan Schaeffer, and T. Anthony Marsland. Information acquisition in minimal window search. In *Proceeding of the International Joint Conference on Artificial Intelligence (IJCAI-85)*, volume 2, pages 1040–1043, 1985.

[26] Igor Roizen and Judea Pearl. A minimax algorithm better than alpha-beta? Yes and no. *Artificial Intelligence*, 21:199–230, 1983.

[27] Jonathan Schaeffer. *Experiments in Search and Knowledge*. PhD thesis, Department of Computing Science, University of Waterloo, Canada, 1986. Available as University of Alberta technical report TR86-12.

[28] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-11(1):1203–1212, November 1989.

[29] George C. Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence*, 12(2):179–196, 1979.

[30] Jean-Christophe Weill. The NegaC* search. *ICCA Journal*, 15(1):3–7, March 1992.