Best-First Fixed-Depth Game-Tree Search in Practice

Aske Plaat,[1] Jonathan Schaeffer,[2] Wim Pijls,[1] Arie de Bruin[1]
plaat@cs.few.eur.nl, jonathan@cs.ualberta.ca, whlmp@cs.few.eur.nl, arie@cs.few.eur.nl

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

[2] University of Alberta,
Dept. of Computing Science,
615 General Services Building,
Edmonton, AB, Canada T6G 2H1

Abstract

We present a new paradigm for minimax search algorithms: MT, a memory-enhanced version of Pearl's *Test* procedure. By changing the way MT is called, a number of practical best-first search algorithms can be simply constructed. Reformulating SSS* as an instance of MT eliminates all its perceived implementation drawbacks. Most assessments of minimax search performance are based on simulations that do not address two key ingredients of high performance game-playing programs: iterative deepening and memory usage. Instead, we use experimental data gathered from tournament checkers, Othello and chess programs. The use of iterative deepening and memory makes our results differ significantly from the literature. One new instance of our framework, MTD(/), out-performs our best Alpha-Beta searcher on leaf nodes, total nodes and execution time. To our knowledge, these are the first reported results that compare both depth-first and best-first algorithms given the same amount of memory.

1 Introduction

For over 30 years, Alpha-Beta has been the algorithm of choice for searching game trees. Using a simple left-to-right depth-first traversal, it is able to efficiently search trees [Knuth and Moore, 1975; Pearl, 1982]. Several important enhancements were added to the basic Alpha-Beta framework, including iterative deepening, transposition tables, the history heuristic, and minimal search windows [Schaeffer, 1989]. The resulting algorithm is so efficient that other promising fixed-depth algorithms were largely ignored. In particular, although best-first search strategies seemed promising both analytically and in simulations, they are not used in practice.

This paper presents a number of contributions to our understanding of depth-first and best-first minimax search:

- MT, a memory enhanced version of Pearl's Test procedure, is introduced. MT yields a binary-valued decision. We present a simple framework of MT drivers (MTD) that make repeated calls to MT to home in on the minimax value. Surprisingly, MTD can be used to construct a variety of best-first search algorithms (such as SSS* [Stockman, 1979]) using depth-first search.
- SSS* (and its dual DUAL* [Marsland *et al.*, 1987]) has not been used in practice because of several perceived drawbacks [Campbell and Marsland, 1983; Kaindl *et al,* 1991; Marsland *et al.*, 1987; Roizen and Pearl, 1983]. When expressed in the MTD framework, these problems disappear. Furthermore, SSS* becomes easy to implement and integrate into a conventional Alpha-Beta game-playing program.
- Simulations of minimax search algorithms in the literature are misleading because they make simplifying assumptions (such as assuming no dependencies between leaf values, or excluding iterative deepening and transposition tables). Our approach was to gather experimental data from three real game-playing programs (chess, Othello and checkers), covering the range from high to low branching factors. Our results contradict published simulation results on the relative merit of a variety of minimax search algorithms [Kaindl *et al.*, 1991; Marsland *et al.*, 1987; Reinefeld and Ridinger, 1994].
- In previous work, depth-first and best-first minimax search algorithms were compared using different amounts of memory. These are the first experiments that compare them using *identical storage* requirements.
- With dynamic move reordering, SSS* is no longer guaranteed to expand fewer leaf nodes than Alpha-Beta (Stockman's proof [Stockman, 1979] does not hold in practice). In our experiments, SSS* performs as fast as any of our Alpha-Beta implementations, but visits too many *interior* nodes to be practical.
- A new instance of our framework, MTD(/), out performs our best Alpha-Beta searcher on leaf nodes, total nodes and execution time.

2 Memory-enhanced Test

Pearl introduced the concept of a proof procedure for game trees in his Scout algorithm [Pearl, 1982] (also known as *Null-Window Search*). Scout called Test, a proof procedure that takes a search assertion (for example, the minimax value of the tree is $\geq \omega$) and returns a binary value (assertion is true or false). It turns out that Scout-based algorithms will never consider more unique leaf nodes than would Alpha-Beta. For the special case of a perfectly ordered tree both Alpha-Beta and the Scout-variants search the so-called *minimal search tree* [Knuth and Moore, 1975]. Simulations have shown that, on average, Scout-variants (such as NegaScout) significantly out-perform Alpha-Beta [Kaindl *et al.*, 1991; Marsland *et al.*, 1987]. However, when used in practice with iterative deepening, aspiration searching and transposition tables, the performance of all algorithms greatly increases. As a result, the relative advantage of NegaScout significantly decreases [Schaeffer, 1989].

Test does an optimal job of answering a binary-valued question by maximizing the number of cutoffs. However, game-tree searches are usually over a range of values. We would like to use the efficiency of Test to find the minimax value of a search tree. Repeated calls to Test will be inefficient, unless Test is modified to reuse the results of previous searches. Enhancing Test with memory yields a new algorithm which we call MT (short for *Memory-enhanced Test*). The storage can be organized as a familiar transposition table [Campbell and Marsland, 1983]. Before a node is expanded in a search, a check is made to see if the value of the node is available in memory, the result of a previous search. Later on we will see that adding storage has some other benefits that are crucial for the algorithm's efficiency: it makes it possible to have the algorithm select nodes in a best-first order.

We would like our formulation of Test to be as clear and concise as possible. In figure 1 MT is shown as a procedure that takes a node n and a single value γ to test on. For MT to function correctly, γ must be unequal to any possible leaf value, since we have deleted the "=" part of the cutoff test [Plaat *et al.*, 1994a]. We can set $\gamma = \omega + \varepsilon$, where ε is a value smaller than the difference between any two leaf node evaluations. An alternative would be to formulate MT as a null-window Alpha-Beta search ($\alpha = \lfloor\gamma\rfloor, \beta = \lceil\gamma\rceil$) with a transposition table. We prefer our one-bound version because it focuses attention on the value to be tested, although the null window search version may be better suited for implementation in existing game playing programs.

There are three differences between MT and Test. One is that MT is called with $\omega - \varepsilon$ instead of ω, so that there is no need for the "=" part in the cutoff test, obviating extra tests in the code of MT. The second is that MT returns a bound on the minimax value, and not just a Boolean result. The third (and more fundamental) difference is that MT uses storage to pass on search results from one pass to the next, making efficient multi-pass searches possible. [Ibaraki, 1986; De Bruin *et al.*, 1994] provide a theoretical background.

Figure 1 shows the pseudo-code for MT using Negamax. The routine assumes an *evaluate* routine that assigns a value to a node. Determining when to call *evaluate* is application-dependent and is hidden in the definition of the condition n = *leaf*. For a depth *d* fixed-depth search, a leaf is any node that is *d* moves from the root of the tree. The search returns an upper or lower bound on the search value at each node, denoted by f^+ and F- respectively. Before searching a node, the transposition table information is *retrieved* and, if it has been previously searched deep enough, the search is cutoff. At the completion of a node, the bound on the value is *stored* in the table. The bounds stored with each node are denoted using Pascal's *dot* notation.

```
function MT($n, \gamma$) → $g$;
{ precondition: $\gamma \neq$ any leaf-evaluation; MT must be
   called with $\gamma = \omega - \varepsilon$ to prove $g < \omega$ or $g \geq \omega$ }
   if retrieve($n$) = found then
      if $n.f^- > \gamma$ then return $n.f^-$;
      if $n.f^+ < \gamma$ then return $n.f^+$;
   if $n$ = leaf then
      $n.f^+ := n.f^- := g :=$ evaluate($n$);
   else
      $g := -\infty$;
      $c :=$ firstchild($n$);
      while $g < \gamma$ and $c \neq \bot$ do
         $g := \max(g, -$MT$(c, -\gamma))$;
         $c :=$ nextbrother($c$);
      if $g < \gamma$ then $n.f^+ := g$ else $n.f^- := g$;
   store($n$);
   return $g$;
```

Figure 1: MT

```
function MTD($+\infty$) → $f$;
   $g := +\infty$;
   $n := root$;
   repeat
      $bound := g$;
      $g :=$ MT$(n, bound - \varepsilon)$;
   until $g = bound$;
   return $g$;
```

Figure 2: MTD($+\infty$)

In answering the binary-valued question, MT returns a bound on the minimax value. If MT's return-value *g* > 7 then it is a lower bound, while if *g* < 7, it is an upper bound. Usually we want to know more than just a bound. Using repeated calls to MT, the search can home in on the minimax value /. To achieve this, MT must be called from a driver routine. One idea for such a driver would be to start at an upper bound for the search value, /+ = + 00. Subsequent calls to MT can lower this bound until the minimax value is reached. Figure 2 shows the pseudo-code for such a driver, called MTD(+oo). The variable *g* is at all times an upper bound f^+ on the minimax value of the root of the game tree [Plaat *et 0/.*, 1994c]. Surprisingly, MTD(+oo) expands the same leaf nodes in the same order as SSS*, provided that the transposition table is big enough and no information is lost through table collisions (see section 3).

Storage is critical to the performance of a multi-pass MT algorithm. Without it, the program would revisit interior nodes without the benefit of information gained from the previous search and expand them. Instead, MT can retrieve an upper and/or lower bound for a node, using a relatively cheap table lookup. The storage table provides two benefits: (1) preventing unnecessary node re-expansion, and (2) facilitating best-first node selection (see sections 3 and 4). Both are necessary for the efficiency of the algorithm.

One could ask the question whether a simple one-pass Alpha-Beta search with a wide search window would not be as efficient. Various papers point out that a tighter Alpha-Beta window causes more cutoffs than a wider window, all other things being equal (for example, [Campbell and Marsland, 1983; Marsland *et a*/., 1987; Plaat *et a*/., 1994c]). Since MT does not re-expand nodes from a previous pass, it cannot have fewer cutoffs than wide-windowed Alpha-Beta for new nodes. (Nodes expanded in a previous pass are not re-expanded but looked-up in memory.) This implies that any sequence of MT calls will be more efficient (it will never evaluate more leaf nodes and usually significantly less) than a call to Alpha-Beta with window (—oo, +oo), for non-iterative deepening searches.

3 Four Misconceptions concerning SSS*

MTD(+oo) causes MT to expand the same leaf nodes in the same order as SSS* (see [Plaat *et al.,* 1994c] for a substantiation of this claim). The surprising result that a depth-first search procedure can be used to examine nodes in a best-first manner can be explained as follows. The value of *g - e* (where $g = f^+$) causes MT to explore only nodes that can lower the upper bound at the root; this is the best-first expansion order of SSS*. Only children that can influence the value of a node are traversed: the highest-valued child of a max node, and the lowest of a min node. Expanding brothers of these so-called critical children gives a best-first expansion. It is instructive to mentally execute the MTD(+oo) algorithm of figure 1 on an example tree such as the one in [Stockman, 1979], as is done in [Plaat *et a*/., 1994b].

An important issue concerning the efficiency of MT-based algorithms is memory usage. SSS* can be regarded as manipulating one max solution tree in place. A max solution tree has only one successor at each min node and all successors at max nodes, while the converse is true for min solution trees [Pijls and De Bruin, 1990; Stockman, 1979]. Whenever the upper bound is lowered, a new (better) subtree has been expanded. MTD(+oo) has to keep only this best max solution tree in memory. Given a branching factor of *w* and a tree of depth d, the space complexity of a driver causing MT to construct and refine one max solution tree is therefore of the order $0(w^{[d/2]}$^ and a driver manipulating one min solution tree is of order $0(w^{[d/2]}$ (as required for DUAL*). A simple calculation and empirical evidence show this to be realistic storage requirements. (Due to lack of space we refer to [Plaat *et a*/., 1994c] for an in-depth treatment of these issues.)

A transposition table provides a flexible way of storing solution trees. While at any time entries from old (inferior) solution trees may be resident, they will be overwritten by newer entries when their space is needed. There is no need for a time-consuming SSS* *purge* operation. As long as the table is big enough to store the min and/or max solution trees that are essential for the efficient operation of the algorithm, it provides for fast access and efficient storage.

The literature cites four issues concerning SSS* [Kaindl *et a*/., 1991; Roizen and Pearl, 1983]. The first is the complicated algorithm. Comparing the code for Alpha-Beta [Knuth and Moore, 1975] and SSS* [Stockman, 1979], one cannot help getting the feeling of being overwhelmed by its complexity. Looking at the code in figure 2 we think this is solved. The second is SSS*'s exponential storage demands. A counter-argument is that Alpha-Beta in game-playing programs also has exponential storage needs to achieve good performance; the transposition table must be large enough to store the move ordering information of the previous iteration. In other words, both Alpha-Beta and SSS* perform best with a minimum of storage of the order of the size of min/max solution tree(s). The third is that SSS* uses expensive operations on the sorted OPEN list. In our MT reformulation, no such operations are used. There is no explicit OPEN list, only an implicit search tree stored in a transposition table. The store and retrieve operations are just as fast for Alpha-Beta as for SSS*. In summary, the arguments against SSS* are eliminated using an MT representation. SSS* is no longer an impractical algorithm [Plaat *et al.,* 1994c].

The fourth issue in the literature is that SSS* will provably never expand more leaf nodes than Alpha-Beta [Stockman, 1979]. However, our experiments used iterative deepening and move reordering, which violates the implied preconditions of the proof. In expanding more nodes than SSS* in a previous iteration, Alpha-Beta reorders more nodes. Consequently, in a subsequent iteration SSS* may have to consider a node for which it has no move ordering information whereas Alpha-Beta does. Thus, Alpha-Beta's inefficiency in a previous iteration can actually benefit it later in the search. With iterative deepening, it is now possible for Alpha-Beta to expand *fewer* leaf nodes than SSS* (a short example proving this can be found in [Plaat *et a*/., 1994c]).

MTD(+ oo) shows up poorly if all nodes visited in the search is used as the performance metric. MTD(-l-oo) retraverses internal nodes to find the best node to expand next, whereas Alpha-Beta does not.

We conclude that our reformulation together with the results of section 5 contradict the literature on all four points.

4 Drivers for MT

Having seen one driver for MT, the ideas can be encompassed in a generalized driver routine. The driver can be regarded as providing a series of calls to MT to successively refine bounds on the minimax value. By parameterizing the driver code, a variety of algorithms can be constructed. The parameter needed is the first starting bound for MT. Using this parameter, an algorithm using our MT driver, MTD, can be expressed as *MTD (first)* (see figure 3). (In [Plaat *et a*/., 1994b] a more general version of MTD is presented, facilitating the construction of more algorithms.) A number of interesting algorithms can easily be constructed using MTD. Some interesting MTD formulations include:

```
function MTD(first) → $f$;
    $f^+ := +\infty$; $f^- := -\infty$;
    $g$ := first;
    $n$ := root;
    repeat
        if $g = f^+$ then $\gamma := g - \epsilon$ else $\gamma := g + \epsilon$;
        $g$ := MT($n, \gamma$);
        if $g < \gamma$ then $f^+ := g$ else $f^- := g$;
    until $f^- = f^+$;
    return $g$;
```

Figure 3: A framework for MT drivers

SSS*. SSS* can be described as MTD($+\infty$).

DUAL*. In the dual version of SSS* minimization is replaced by maximization, the OPEN list is kept in reverse order, and the starting value is -oo. This algorithm becomes MTD($-\infty$). The advantage of DUAL* over SSS* lies in the search of odd-depth search trees [Marsland *et* a/., 1987].

MTD(/). Rather than using $+\infty$ or $-\infty$ as a first bound, we can start at a value which might be closer to /. Given that iterative deepening is used in many application domains, the obvious approximation for the minimax value is the result of the previous iteration. In MTD terms this algorithm becomes MTD(heuristic-guess). If the initial guess is below the minimax value, MTD(/) can be viewed as a version of DUAL* that started closer to /, otherwise it becomes a version of SSS* that started closer to /.

Other MTD variations possible are: bisecting the interval $[f^+, f^-]$ in each pass, using larger step sizes, and searching for the best move (not the best value) [Plaat *et* a/., 1994b].

Formulating a seemingly diverse collection of algorithms into one unifying framework focuses attention on the fundamental differences. For example, the framework allows the reader to see just how similar SSS* and DUAL* really are, and that these are just special cases of calling Pearl's Test (or rather MT). The drivers concisely capture the algorithm differences. MTD offers us a high-level paradigm that facilitates the reasoning about important issues like algorithm efficiency and memory usage, without the need for low-level details.

By using MT, all MTD algorithms benefit from the maximum number of cutoffs a single bound can generate. Each MTD makes a different choice for this bound, which influences the number of cutoffs. Tests show that on average, there is a relationship between the starting bound and the size of the search trees generated: a sequence of MT searches to find the game value benefits from a start value close to the game value. Starting bounds such as $+\infty$ or $-\infty$ are in a sense the worst possible choices.

Figure 4 validates the choice of a starting bound close to the game value. The figure shows the percentage of unique leaf evaluations of MTD(/), for Othello; similar results were obtained using chess and checkers. The data points are given as a percentage of the size of the search tree built by our best Alpha-Beta searcher (Aspiration NegaScout). (Since iterative deepening algorithms are used, the cumulative leaf count over all previous depths

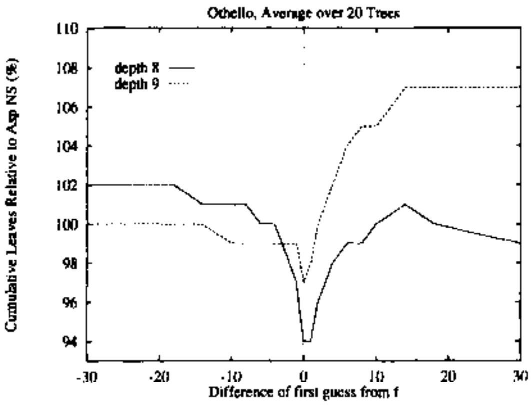

Figure 4: Tree size relative to the first guess f

is shown for depth 8 and 9.) Given an initial guess of *h* and the minimax value of /, the graph plots the search effort expended for different values of *h — f.* For each depth the first guess is distorted by the same amount. To the left of the graph, MTD(/) is closer to DUAL*, to the right it is closer to SSS*. A first guess close to / makes MTD(/) perform better than the 100% Aspiration NegaScout baseline. The guess must be close to / for the effect to become significant (between —15 and + 10 of / for Othello, given that values lie in the range $[-64, +64]$). Thus, if MTD(/) is to be effective, the / obtained from the previous iteration must be a good indicator of the next iteration's value.

5 Experiments

There are three ways to evaluate a new algorithm: analysis, simulation or empirical testing. The emphasis in the literature has been on analysis and simulation. This is surprising given the large number of game-playing programs in existence.

The mathematical analyses of minimax search algorithms do a good job of increasing our understanding of the algorithms, but fail to give reliable predictions of their performance. The problem is that the game trees are analyzed using simplifying assumptions; the trees differ from those generated by real game-playing programs. To overcome this deficiency, a number of authors have conducted simulations (for example, [Kaindl *et* a/., 1991; Marsland *et* a/., 1987; Muszycka and Shinghal, 1985]). In our opinion, the simulations did not capture the behavior of realistic search algorithms as they are used in game-playing programs. Instead, we decided to conduct experiments in a setting that was to be as realistic as possible. Our experiments attempt to address the concerns we have with the parameters chosen in many of the simulations:

- High degree of ordering: most simulations have the quality of their move ordering below what is seen in real game-playing programs.
- Dynamic move re-ordering: simulations use fixed-depth searching. Game-playing programs use iterative deepening to seed memory (transposition table)

with best moves to improve the move ordering. This adds overhead to the search, which is more than offset by the improved move ordering. Also, transpositions and the history heuristic dynamically re-order the game tree during the search. Proofs that SSS* does not expand more leaf nodes than Alpha-Beta do *not* hold for the iterative deepened versions of these algorithms.

- Memory: simulations assume either no storage of previously computed results, or unfairly bias their experiments by not giving all the algorithms the same storage. For iterative deepening to be effective, best move information from previous iterations must be saved in memory. In game-playing programs a transposition table is used. Simulations often use an inconsistent standard for counting leaf nodes. In conventional simulations (for example, [Marsland *et* a/., 1987]) each visit to a leaf node is counted for depth-first algorithms like NegaScout, whereas the leaf is counted only once for best-first algorithms like SSS* (because it was stored in memory, no re-expansion occurs).
- Value dependence: some simulations generate the value of a child independent of the value of the parent. However, there is usually a high correlation between the values of these two nodes in real games.

The net result is that iterative deepening and memory improve the move ordering beyond what has been used in most simulations. Besides move ordering the other three differences between artificial and real trees can cause problems in simulations. Just increasing the move ordering to 98% is not sufficient to yield realistic simulations. As well, simulations are concerned with tree size, but practitioners are concerned with execution time. Simulation results do not necessarily correlate well with execution time. For example, there are many papers showing SSS* expands fewer leaf nodes than Alpha-Beta. However, SSS* implementations using Stockman's original formulation have too much execution overhead to be competitive with Alpha-Beta [Roizen and Pearl, 1983],

5.1 Experiment Design

To assess the feasibility of the proposed algorithms, a series of experiments was performed to compare Alpha-Beta, NegaScout, SSS* (MTD(+oo)), DUAL* (MTD(-oo)), MTD(/) (and other variants, see [Plaat *et* al., 1994b]).

Rather than use simulations, our data has been gathered from three game-playing programs: Chinook (checkers), Keyano (Othello), and Phoenix (chess). All three programs are well-known in their respective domain. For our experiments, we used the program author's search algorithm which, presumably, has been highly tuned to the application. The only change we made was to disable search extensions and forward pruning. All programs used iterative deepening. The MTD algorithms would be repeatedly called with successively deeper search depths. All three programs used a standard transposition table with 2^{21} entries. For our experiments we used the program author's original transposition table data structures and table manipulation code.

Conventional test sets in the literature proved to be poor predictors of performance. Test set positions are selected, usually, to test a particular characteristic or property of the game and are not indicative of typical game conditions. By using a sequences of moves from real games as the test positions, we are attempting to create a test set that is representative of real game search properties.

All three programs were run on 20 balanced test positions, searching to a depth so that all searched roughly the same amount of time. (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 cause anomalies.) In checkers, the average branching factor is approximately 3 (1.2 moves in a capture position; 8 in a non-capture position), in Othello it is 10 and in chess it is 36. The branching factor determined the maximum search depth for our experiments: 17 ply for Chinook, 10 ply for Keyano, and 8 ply for Phoenix.

Many papers in the literature use Alpha-Beta as the base line for comparing the performance of other algorithms (for example, [Campbell and Marsland, 1983]). 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. Therefore, we have chosen to use the current algorithm of choice as our base line: aspiration window enhanced NegaScout [Campbell and Marsland, 1983]. The graphs in figure 4 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.

To our knowledge this is the first comparison of algorithms like Alpha-Beta, NegaScout, SSS* and DUAL* where all algorithms are given the exact same resources.

5.2 Experiment Results

Figure 5 shows the performance of Phoenix for (unique) leaf evaluations (NBP or number of bottom positions), and Figure 6 shows the total node count (leaf, interior, *and* transposition nodes). The total node count includes all revisits to previously searched nodes. Although most simulations only report NBP, we find that the total node count has a higher correlation with execution time for some algorithms. Detailed results for all the games can be found in [Plaat *et* a/., 1994b].

Over all three games, the best results are from MTD(/). Its leaf node counts are consistently better than Aspiration NegaScout, averaging at least a 5% improvement. More surprisingly is that MTD(/) 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(/) to perform badly by this measure because of the repeated traversals of the tree. This suggests that MTD(/), on average, is calling MT close to the minimum number of times. For all three programs, MT gets called between 3 and 4 times on average. In contrast, the SSS* and DUAL* results are poor compared

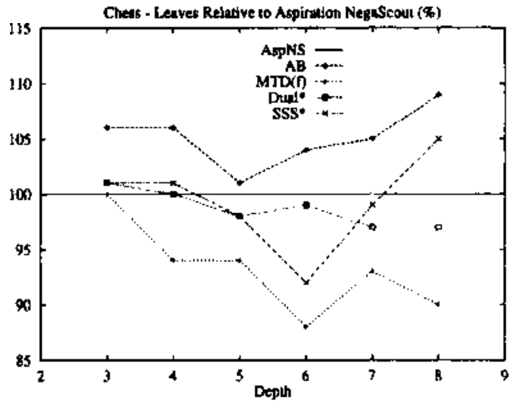

Figure 5: Cumulative leaf node count

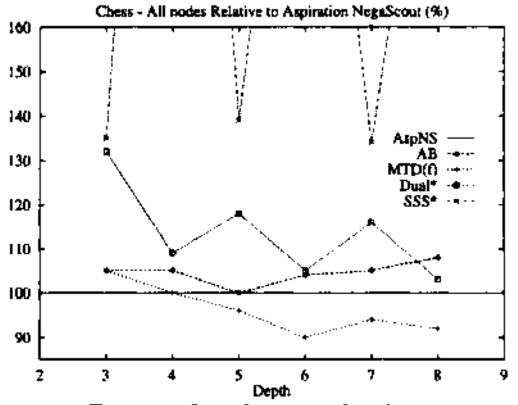

Figure 6: Cumulative total node count

to NegaScout when all nodes in the search tree are considered. Each of these algorithms performs dozens and sometimes even hundreds of MT searches, depending on how wide the range of leaf values is.

Implementing SSS* as an instance of MTD yields results that run counter to the literature. SSS* is now as easy to implement as Aspiration NegaScout, uses as much storage and has no additional execution overhead, but performs generally worse when viewed in the context of iterative deepening and transposition tables. DUAL* is more efficient than SSS* but still comes out poorly in all the graphs measuring total node count. Sometimes SSS* expands more leaf nodes than Alpha-Beta (as discussed in section 3), contradicting both the analytic and simulation results for fixed-depth SSS* and Alpha-Beta. An interesting observation is that the effectiveness of SSS* appears to be a function of the branching factor; the larger the branching factor, the better it performs.

Given these results, some of the algorithmic differences can be explained. If we know the value of the search tree is /, then two searches are required: $\mathrm{MT}(f - \varepsilon)$, which fails high establishing a lower bound on /, and $\mathrm{MT}(f + \varepsilon)$., which fails low and establishes an upper bound on /. The closer the approximation to /, the less the work that has to be done (according to figure 4). As that figure indicated, the performance of MTD(/) is

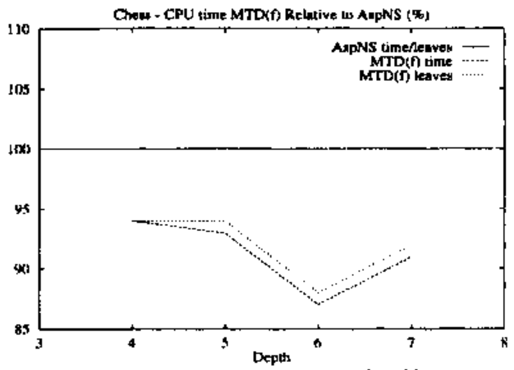

Figure 7: Execution time for Chess

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 passes ago. Considering this, it is not a surprise that both DUAL* and SSS* come out poorly. Their initial bounds for the minimax value are poor, meaning that the many calls to MT result in significantly more interior as well as leaf nodes. NegaScout used a wide window for the principal variation (PV) and all re-searches. The wide-window search of the PV gives a good first approximation to the minimax value. That approximation is then used to search the rest of the tree with minimal window searches—which are equivalent to MT calls. If these refutation searches are successful (no re-search is needed), then NegaScout deviates from MTD(/) only in the way it searches the PV for a value, the wider window causing less cutoffs. MTD(/) uses MT for searching all nodes, including the PV.

The bottom line for practitioners is execution time. Since we did not have the resources to run all our experiments on identical and otherwise idle machines, we only show execution time graphs for MTD(/) in figure 7. Comparing results for the same machines we found that MTD(/) is on average consistently the fastest algorithm. In our experiments we found that for Chinook and Keyano MTD(/) was about 5% faster in execution time than Aspiration NegaScout, for Phoenix we found MTD(/) 9-13% faster. 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 relatively close together, the relative differences are quite sensitive to variations in input parameters. 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. We refer to [Plaat *et al.,* 1994b] for the remainder of our experimental data and explanations.

Basing one's conclusions only on simulations can be hazardous. For example, the general view is that SSS* is (1) difficult to understand, (2) has unreasonable memory requirements, (3) is slow, (4) provably dominates Alpha-Beta in expanded leaves, and (5) that it expands significantly fewer leaf nodes than Alpha-Beta. A recent paper used simulations to show that point 2 and 3 could be wrong [Reinefeld and Ridinger, 1994], painting an altogether favorable picture for SSS*. Using real programs, we showed that *all* five points are wrong, making it clear that, although SSS* is practical, in realistic programs it has *no* substantial advantage over Alpha-Beta-variants like Aspiration NegaScout. We think that only real programs provide a valid basis for conclusions.

6 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(/) 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 eliminates all the 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, transposition tables, and almost perfectly ordered trees in many simulations are serious omissions.

Although a 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. 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 (author of Keyano), Yngvi Bjornsson and Andreas Junghanns. The financial support of the Dutch Organization for Scientific Research (NWO), the Natural Sciences and Engineering Research Council of Canada (grant OGP-5183) and the University of Alberta Central Research Fund are gratefully acknowledged.

References

[De Bruin *et ai,* 1994] A. de Bruin, W. Pijls, and A. Plaat. Solution trees as a basis for game-tree search. *ICCA Journal,* 17(4):207-219, December 1994.

[Campbell and Marsland, 1983] M. Campbell and T.A. Marsland. A comparison of minimax tree search algorithms. *Artificial Intelligence,* 20:347-367, 1983.

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

[Kaindl *et ai,* 1991] H. Kaindl, R. Shams, and H. Horacek. Minimax search algorithms with and without aspiration windows. *IEEE PAMI,* 13(12):1225-1235, 1991.

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

[Marsland *et ai,* 1987] T.A. Marsland, A. Reinefeld, and J. Schaeffer. Low overhead alternatives to SSS*. *Artificial Intelligence,* 31:185-199, 1987.

[Muszycka and Shinghal, 1985] A. Muszycka and R. Shinghal. An empirical comparison of pruning strategies in game trees. *IEEE SMC,* 15(3):389-399, 1985.

[Pearl, 1982] J. Pearl. The solution for the branching factor of the alpha-beta pruning algorithm and its optimally. *CACM,* 25(8):559-564, 1982.

[Pijls and De Bruin, 1990] W. Pijls and A. de Bruin. Another view on the SSS* algorithm. In T. Asano, editor, *Algorithms, SIGAL '90, Tokyo,* volume 450 of *LNCS,* pages 211-220. Springer-Verlag, August 1990.

[Plaat *et ai,* 1994a] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Nearly optimal minimax tree search? Technical Report CS-94-19, Dept. of Computing Science, Univ. of Alberta, 1994.

[Plaat *et al,* 1994b] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. A new paradigm for minimax search. Technical Report CS-94-18, Dept. of Computing Science, Univ. of Alberta, 1994.

[Plaat *et ai,* 1994c] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. SSS* = *aB* + TT. Technical Report CS-94-17, Dept. of Computing Science, Univ. of Alberta, 1994.

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

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

[Schaeffer, 1989) J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE PAMI,* 11(1):1203 1212, 1989.

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