Best-First Fixed-Depth Minimax Algorithms

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

December 14, 1995

Abstract

This article has three main contributions to our understanding of minimax search:

First, a new formulation for Stockman's SSS* algorithm, based on Alpha-Beta, is presented. It solves all the perceived drawbacks of SSS*, finally transforming it into a practical algorithm. In effect, we show that SSS* = Alpha-Beta + transposition tables. The crucial step is the realization that transposition tables contain so-called solution trees, structures that are used in best-first search algorithms like SSS*. Having created a practical version, we present performance measurements with tournament game-playing programs for three different minimax games, yielding results that contradict a number of publications.

Second, based on the insights gained in our attempts at understanding SSS*, we present a framework that facilitates the construction of several best-first fixed-depth game-tree search algorithms, known and new. The framework is based on depth-first null-window Alpha-Beta search, enhanced with storage to allow for the refining of previous search results. It focuses attention on the essential differences between algorithms.

Third, a new instance of this framework is presented. It performs better than algorithms that are currently used in most state-of-the-art game-playing programs. We provide experimental evidence to explain why this new algorithm, MTD(f), performs better than other fixed-depth minimax algorithms.

Keywords: Game-tree search, Minimax search, Alpha-Beta, SSS*, Transposition tables, Null-window search, Solution trees.

1 Introduction

The Alpha-Beta tree-searching algorithm [18] has been in use since the 1960's. No other minimax search algorithm has achieved the wide-spread use in practical applications that Alpha-Beta has. Thirty years of research has found ways of improving the algorithm's efficiency, and variants such as NegaScout [41] and PVS [8] are quite popular. Interesting alternatives to depth-first searching, such as breadth-first and best-first strategies, have been largely ignored in practice.

In 1979 Stockman introduced SSS*, which looked like a radically different approach from Alpha-Beta for searching fixed-depth minimax trees [51]. It builds a tree in a so-called best-first fashion by visiting the most promising nodes first.[1] Alpha-Beta, in contrast, uses a depth-first, left-to-right traversal of the tree. Intuitively, it would seem that a best-first strategy should prevail over a rigidly ordered depth-first one. Stockman proved that SSS* dominated Alpha-Beta; it would never evaluate more leaf nodes than Alpha-Beta. Numerous simulations have shown that on average SSS* evaluates considerably fewer leaf nodes (for example, [17, 24, 26, 41, 43, 45]). Why, then, has the algorithm been shunned by practitioners?

SSS*, as formulated by Stockman, has several problems. First, it takes considerable effort to understand how the algorithm works, and still more to understand its relation to Alpha-Beta. Second, SSS* maintains a data structure known as the OPEN list, similar to that found in single-agent search algorithms like A* [30]. The size of this list grows exponentially with the depth of the search tree. This has led many authors to conclude that SSS* is effectively disqualified from being useful for real applications like game-playing programs [17, 26, 45, 51]. Third, the OPEN list must be kept in sorted order. Insert and (in particular) delete/purge operations on the OPEN list can dominate the execution time of any program using SSS*. Despite the promise of expanding fewer nodes, the disadvantages of SSS* have proven a significant deterrent in practice. The general view of SSS* then is that:

1. it is a complex algorithm that is difficult to understand,
2. it has large memory requirements that make the algorithm impractical for real applications,
3. it is "slow" because of the overhead of maintaining the sorted OPEN list,
4. it has been proven to dominate Alpha-Beta in terms of the number of leaf nodes evaluated, and
5. it evaluates significantly fewer leaf nodes than Alpha-Beta.

For a number of years, we have been trying to find out how and why SSS* works, and whether the drawbacks can be solved. In this article we report the following results:

- The obstacles to efficient SSS* implementations have been solved, making the algorithm a practical alternative to Alpha-Beta variants. By reformulating the algorithm, SSS* can be expressed simply and intuitively as a series of calls to Alpha-Beta enhanced with a transposition table (TT), yielding a new formulation called MT-SSS*. MT-SSS* does not need an expensive OPEN list; a familiar transposition table performs as well. In effect: SSS* = Alpha-Beta + TT.

[1] There is potential for confusion between SSS*, which selects the node offering the "best" information on bounds at the root in a fixed-depth search, and a new algorithm called Best-First Minimax Search, which expands the children of the "best" node in a variable-depth search [20].

- Inspired by the MT-SSS* reformulation, a new framework for minimax search is introduced. It is based on memory-enhanced null-window Alpha-Beta search. We call this procedure MT, after Pearl's Test procedure [30]. We present a simple framework of MT drivers (MTD) that make repeated calls to MT to home in on the minimax value. Search results from previous passes are stored in memory and re-used. MTD can be used to construct a variety of fixed-depth best-first search algorithms using depth-first search. It is easily incorporated into existing game-playing programs.

- Using our new framework, we were able to compare the performance of a number of best-first algorithms to some well-known depth-first algorithms, using three high performance game-playing programs. The results of these experiments were quite surprising, since they contradict the large body of published results based on simulations: best-first searches and depth-first searches have roughly comparable performance, with NegaScout, a depth-first algorithm, often out-performing SSS*, a best-first algorithm.

  In previously published experimental results, depth-first and best-first minimax search algorithms were allowed different memory requirements. To our knowledge, we present the first experiments that compare them using *identical* storage requirements.

- With dynamic move reordering schemes, like iterative deepening, SSS* (and its dual DUAL* [21, 24, 41]) are no longer guaranteed to expand fewer leaf nodes than Alpha-Beta. The conditions for Stockman's proof [51] are not met in practice.

- In analyzing why our results differ from simulations, we identify a number of differences between real and artificially generated game trees. Two important factors are transpositions and value interdependence between parent and child nodes. In game-playing programs these factors are commonly exploited by transposition tables and iterative deepening to yield large performance gains—making it possible for depth-first algorithms to out-perform best-first. Given that most simulations neglect to include important properties of trees built in practice, of what value are the previously published simulation results?

- We formulate a new algorithm, MTD(f). It out-performs our best Alpha-Beta variant, NegaScout enhanced with an aspiration window, on leaf nodes, total nodes, and execution time for our test programs. Since MTD(f) is an instance of the MT framework, it is easily implemented in existing programs: just add one loop to an Alpha-Beta-based program.

- In the past, much research effort has been devoted to understanding how SSS* works, and finding out what the pros and cons of SSS*'s best-first approach are for minimax search. In the new framework, SSS* is equivalent to a special case of Alpha-Beta and it is out-performed by other Alpha-Beta variants (both best-first and depth-first). In light of this, we believe that SSS* should now become a footnote in the history of game-tree search.

In section 2 we use an example to demonstrate how a best-first search uses its information to decide which node to select next. Specifically, this section introduces MT-SSS*, which is a reformulation of SSS* based on Alpha-Beta. Section 3 addresses one of the biggest drawbacks of SSS*: its memory requirements. We will show empirical evidence using our reformulation that this problem is effectively solved for our applications. In section 4 we introduce a framework for fixed-depth best-first minimax algorithms based on null-window Alpha-Beta searches enhanced with a transposition table. In section 5 we present the results of performance tests with three tournament-level game-playing programs. One algorithm, MTD(f), is on average consistently better. In explaining its behavior, we establish a relation between the start value of a series of null-window searches and performance. Section 6 addresses the reasons why our results contradict the literature: the difference between real and artificial game trees is significant. Given that high-performance game-playing programs are readily available, the case for simulations is weak. Section 7 gives the conclusions. Appendix A provides a more formal treatment of why MT-SSS* and SSS* are equivalent in the sense that they expand the same leaf nodes in the same order. Appendix B presents an example proving that when SSS* is used with dynamic move reordering, it no longer dominates Alpha-Beta.

To conclude this introduction, we make a remark on terminology. Sometimes the term Alpha-Beta is used to denote a single procedure which can be called with any search window, for example, as a building block for algorithms like MT-SSS* and MTD(f). At other times Alpha-Beta is meant as the algorithm Alpha-Beta($n, -\infty, +\infty$) that finds the minimax value of a tree rooted at n. Which of the two is meant should be clear from the context.

Preliminary results from this research have appeared in [38].

2 A Practical Version of SSS*

SSS* is a difficult algorithm to understand, as can be appreciated by looking at the code in figure 1. SSS* works by manipulating a list of nodes, the OPEN list, using six ingenious inter-locking cases of the so-called Γ operator. Throughout this paper, it is assumed that the root is of type MAX. The nodes have a status associated with them, either *live* (L) or *solved* (S), and a merit, denoted $\hat{h}$. The OPEN list is sorted in descending order, so that the entry with highest merit (the "best" node) is at the front and will be selected for expansion.

In this section we present a clearer formulation that has the added advantage of solving a number of obstacles that have hindered SSS*'s use in practice. The reformulation is based on the Alpha-Beta procedure. It examines the same leaf nodes in the same order as SSS*. It is called MT-SSS*, and the code is shown later in figure 8.

Figure 2 shows the pseudo-code for Alpha-Beta (enchanced with storage) [18, 23]. In contrast to SSS*, the code is a tight recursive formulation. The relative simplicity of the code has made it a popular choice for implementation by practitioners. In the code, *eval* returns the evaluation of a leaf node, *firstchild* and *nextbrother* are used to generate the successor nodes of a position, and storage is accessed using the *store* and *retrieve* routines. f denotes the minimax value of a node; f^+ is an upper bound on that value, while f^- is a lower bound. The code specifies the fail-soft variant of Alpha-Beta [14], where a return value outside the search window is a bound on the minimax value.

Stockman's SSS* (including Campbell's correction [7])

(1) Place the start state $\langle n = root, s = \text{LIVE}, \hat{h} = +\infty \rangle$ on a list called OPEN.

(2) Remove from OPEN state $p = \langle n, s, \hat{h} \rangle$ with largest merit $\hat{h}$. OPEN is a list kept in non-decreasing order of merit, so p will be the first in the list.

(3) If $n = root$ and $s = \text{SOLVED}$ then p is the goal state so terminate with $\hat{h} = f(root)$ as the minimax evaluation of the game tree. Otherwise continue.

(4) Expand state p by applying state space operator Γ and queuing all output states $\Gamma(p)$ on the list OPEN in merit order. Purge redundant states from OPEN if possible. The specific actions of Γ are given in the table below.

(5) Go to (2)

State space operations on state $\langle n, s, \hat{h} \rangle$ (just removed from top of OPEN list)

Case of operator Γ	Conditions satisfied by input state $\langle n, s, \hat{h} \rangle$	Actions of Γ in creating new output states
not applicable	$s = \text{SOLVED}$ $n = \text{ROOT}$	Final state reached, exit algorithm with $g(n) = \hat{h}$.
1	$s = \text{SOLVED}$ $n \neq \text{ROOT}$ $\text{type}(n) = \text{MIN}$	Stack $\langle m = \text{parent}(n), s, \hat{h} \rangle$ on OPEN list. Then purge OPEN of all states $(k, s, \hat{h})$ where m is an ancestor of k in the game tree.
2	$s = \text{SOLVED}$ $n \neq \text{ROOT}$ $\text{type}(n) = \text{MAX}$ $\text{next}(n) \neq \text{NIL}$	Stack $\langle \text{next}(n), \text{LIVE}, \hat{h} \rangle$ on OPEN list
3	$s = \text{SOLVED}$ $n \neq \text{ROOT}$ $\text{type}(n) = \text{MAX}$ $\text{next}(n) = \text{NIL}$	Stack $\langle \text{parent}(n), s, \hat{h} \rangle$ on OPEN list
4	$s = \text{LIVE}$ $\text{first}(n) = \text{NIL}$	Place $\langle n, \text{SOLVED}, \min(\hat{h}, f(n)) \rangle$ on OPEN list (interior) in front of all states of lesser merit. Ties are resolved left-first.
5	$s = \text{LIVE}$ $\text{first}(n) \neq \text{NIL}$ $\text{type}(\text{first}(n)) = \text{MAX}$	Stack $\langle \text{first}(n), s, \hat{h} \rangle$ on (top of) OPEN list.

6	$s = \text{LIVE}$
	$\text{first}(n) \neq \text{NIL}$
	$\text{type}(\text{first}(n)) = \text{MIN}$
		Reset n to $\text{first}(n)$.
		While $n \neq \text{NIL}$ do
		  queue $\langle n, s, \hat{h} \rangle$ on top of OPEN list
		  reset n to $\text{next}(n)$

Figure 1: Stockman's SSS* [30, 51]

```
function alphabeta(n, α, β) → g;
    if retrieve(n) = ok then
        if n.f⁻ ≥ β then return n.f⁻;
        if n.f⁺ ≤ α then return n.f⁺;
    if n is a leaf node then g := eval(n);
    else if n is a max node then
        g := −∞; a := α;
        c := firstchild(n);
        while g < β and c ≠ ⊥ do
            g := max(g, alphabeta(c, a, β));
            a := max(a, g);
            c := nextbrother(c);
    else /* n is a min node */
        g := +∞; b := β;
        c := firstchild(n);
        while g > α and c ≠ ⊥ do
            g := min(g, alphabeta(c, α, b));
            b := min(b, g);
            c := nextbrother(c);
    if g < β then n.f⁺ := g;
    if g > α then n.f⁻ := g;
    store n.f⁻, n.f⁺;
    return g;
```

Figure 2: The Alpha-Beta Function for use with Transposition Tables

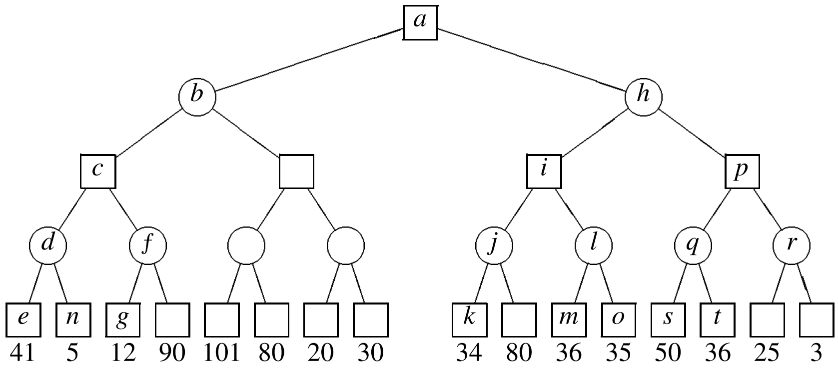

Figure 3: Example Tree for MT-SSS*

The relationship between SSS* and Alpha-Beta will be discussed using an example which concentrates on the higher-level concepts. Formality is deferred to appendix A.

The two key concepts in our explanation of the relationship between SSS* and Alpha-Beta are an *upper bound* on the minimax value, and a *max solution tree*, which is the minimal search tree that proves an upper bound.[2] We will explain max solution trees, and how SSS* constructs them, shortly.

2.1 Example

Figure 3 is used to illustrate how SSS* and MT-SSS* search for the minimax value. This section contains a detailed description of how MT-SSS* works. The example assumes some familiarity with SSS*. One of the reasons to create MT-SSS* was the sense of confusion that the complexity of SSS* brings about. By using standard concepts from the Alpha-Beta literature we try to alleviate this problem. Although instructive, going through the example step-by-step is not necessary to follow the rest of this article. For ease of reference, this tree is the same as used by Pearl in his explanation of SSS* [30].

A number of stages, or passes, can be distinguished in the traversal of this tree. At the end of each pass the OPEN list consists of *solved* nodes only. We will go to some depth examining how Alpha-Beta can be used to traverse this tree in a best-first fashion. For reasons of brevity we refer to to [30, 35, 39] for the details of how SSS* traverses the tree. In the figures the nodes are numbered *a* to *t* in the order in which SSS* first visits them.

First pass: (see figure 4)

In the first pass the left-most max solution tree is constructed, creating the first non-trivial upper bound on the minimax value of the root. $+\infty$ and $-\infty$ are used as the upper and lower bounds on the range of leaf values. In real implementations, these bounds are suitably large finite numbers.

As can be seen from the example in [30], SSS* starts by building the tree shown in figure 4, using cases 4, 5, and 6 of the Γ operator. At max nodes, all the children were expanded (case 6), while at min nodes only the first child was added to the OPEN list (case 5). Case 4 evaluated the leaf nodes of the tree. Sorting the list guaranteed that

[2] Stockman originally used min solution trees to explain his algorithm. We explain SSS* using upper bounds and max solution trees, since it improves the clarity of the arguments.

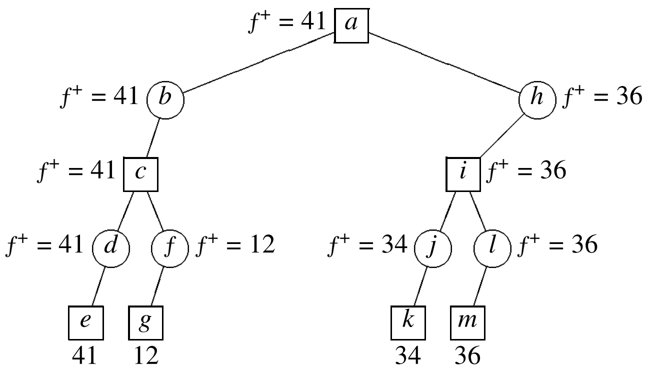

Figure 4: Pass 1

the entry with the highest upper bound was at the front. It is interesting to determine the minimax value of the sub-tree expanded thus far (see figure 4). Since only one child of a min node is included, minimaxing the leaf values simplifies to taking the maximum of all the leaves. The minimax value of this tree is 41, the maximum of the leaves, which is also the $\hat{h}$ value of the first entry of the OPEN list. The (left-most) leaf equal to the value at the root is called the *critical* leaf, while the path from the root to the critical leaf is the *principal variation*. A tree which includes one child at min nodes and all children at max nodes, is called a *max solution* tree (for example, figure 4). The term "solution tree" was originally used in the context of AND/OR trees, where it meant, in our terminology, a min solution tree (one child at max nodes and all children at min nodes). SSS* has shown that solution trees are a useful concept for understanding game-tree algorithms. Solution trees are discussed in [21, 22, 31, 51].

Instead of using Γ cases 4, 5 and 6 and a sorted OPEN list, there are other ways to compute the "left-first" upper bound on the minimax value of a. One way is suggested by the following post-condition of the Alpha-Beta procedure. Assume for node n with minimax value f, that g is the return value of an Alpha-Beta(n, α, β) call. There are three possible outcomes:

1. $\alpha < g < \beta$ (success). g is equal to the minimax value f of node n.
2. $g \leq \alpha$ (failing low). g is an upper bound on f, denoted f^+, or $f \leq g$.
3. $g \geq \beta$ (failing high). g is a lower bound on f, denoted f^-, or $f \geq g$.

Using outcome 2, we can force the Alpha-Beta procedure to return an upper bound (fail low) by calling it with a search window greater than any possible leaf node value. Since both Alpha-Beta and SSS* expand nodes in a left-to-right order, Alpha-Beta when called with this window will find the same upper bound, and expand the same max solution tree, as SSS*. Appendix A provides a more formal treatment of this claim.

In the special case where $\alpha = \beta - 1$, Alpha-Beta always returns a bound on the minimax value. This search window, the narrowest possible for integer-valued evaluations, is called a minimal or null window. The concept of a null-window search, or proof procedure, is well known [14, 29]. Many people have noted that null-window search is more efficient than wide-window search, because the tighter bounds cause

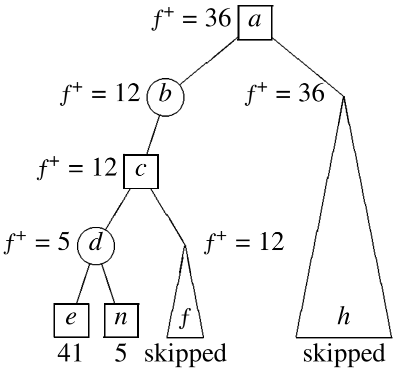

Figure 5: Pass 2

more cutoffs [1, 8, 9, 14, 28, 46]. Pearl introduced the procedure Test, part of his Scout algorithm [28, 29]. NegaScout [40, 41], an enhanced version of Scout, has become the algorithm of choice for many game-playing programs. We named our proof-procedure MT, for Memory-enhanced Test. MT returns a bound, not just a Boolean value. This procedure is sometimes called *fail-soft* Test. The name MT is just shorthand for a null-window call to Alpha-Beta enhanced with storage (such as a transposition table).

A call Alpha-Beta($a, \infty - 1, \infty$) will cause an alpha cutoff at all min nodes, since all internal calls return values $g \leq \alpha = \infty - 1$. No beta cutoffs at max nodes will occur, since all $g < \beta$. The call Alpha-Beta($a, \infty - 1, \infty$) on the tree in figure 3 will traverse the tree in figure 4. Due to the *store* operation in figure 2, this tree is saved in memory so that its backed-up values can be used in a later pass. The max solution tree stored at the end of this pass consists of the nodes $a, b, c, d, e, f, g, h, i, j, k, l$ and m, yielding an upper bound of 41. For Stockman's formulation, the leaves of this tree are stored in the OPEN list, which is $(\langle e, S, 41\rangle, \langle m, S, 36\rangle, \langle k, S, 34\rangle, \langle g, S, 12\rangle)$. Note that the entry at the head of the list is also 41, SSS*'s upper bound on the minimax value.

Second pass: (see figure 5)
This pass lowers the upper bound on f from 41 to 36 using Γ cases 2 and 4 (see [30]). The OPEN list becomes $(\langle m, S, 36\rangle, \langle k, S, 34\rangle, \langle g, S, 12\rangle, \langle n, S, 5\rangle)$. Only one new node has been expanded. The value of the upper bound is determined by a new (sharper) max solution tree, whose leaves are contained in the OPEN list.

How can we use Alpha-Beta to lower the upper bound of the first pass? Since the max solution tree defining the upper bound of 41 has been stored by the previous call, we can re-traverse the nodes on the principal variation (a, b, c, d, e) to find the critical leaf e, and see whether expanding its brother will yield a search tree with a lower minimax value. To give Alpha-Beta the task of returning a value lower than $f^+ = 41$, we give it a search window which will cause it to fail low. The old window of $\langle \infty - 1, \infty\rangle$ will not do, since the code in figure 2 will cause Alpha-Beta to return from both nodes b and h with a value of 41, lower than ∞. A better choice would be the search window $\langle f^+ - 1, f^+\rangle$, or $\langle 40, 41\rangle$, which prompts Alpha-Beta to descend the principal variation and return as soon as a lower f^+ on node a is found. Alpha-Beta will descend to nodes b, c, d, e and continue to search node n. It will back up value 5 to node d and cause a cutoff. The value of d is no longer determined by e but by n.

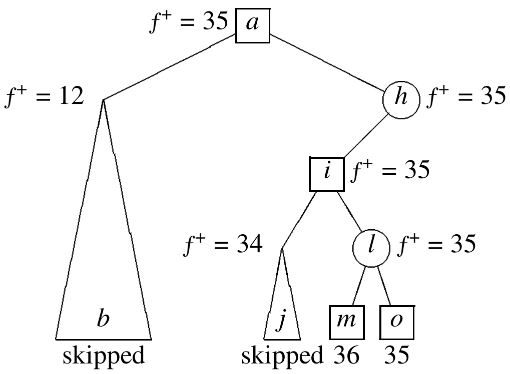

Figure 6: Pass 3

Node e is no longer part of the max solution tree that determines the sharpest upper bound. It has been proven that e can be erased from memory as long as we remember that n is the new best child (not shown in the Alpha-Beta code). The value 5 is backed up to c. No beta cutoff occurs at c, so f's bound is retrieved. Since $f^+ \leq \alpha$ at node f, it returns immediately with value 12. 12 is backed up to b, where it causes an alpha cutoff. Next, 12 is backed up to a. Since $g < \beta$, node h is entered, which returns immediately its value of 36. The call Alpha-Beta$(a, 40, 41)$ fails low with value 36, the sharper upper bound. The max solution tree defining this bound consists of nodes $a, b, c, d, n, f, g, h, i, j, k, l$ and m (that is, node e has been replaced with n).

By storing previously expanded nodes in memory, and calling Alpha-Beta with the right search window, we can make it traverse the principal variation. Alpha-Beta expands brothers of the critical leaf to get a better upper bound on the minimax value of the root, in exactly the same way as SSS* does.

Third Pass: (see figure 6)
In this pass, the upper bound is lowered from 36 to 35. Again, only one new node is expanded. The new OPEN list is $(\langle o, S, 35\rangle, \langle k, S, 34\rangle, \langle g, S, 12\rangle, \langle n, S, 5\rangle)$.

In the Alpha-Beta case, a call Alpha-Beta$(a, 35, 36)$ is performed. From the previous search, we know that b has an $f^+ \leq 35$ and h does not. The algorithm follows the principal variation leading to the leaf node with value 36 (h, i, l, m). The brother of m is expanded. The bound on the minimax value at the root has now been improved from 36 to 35. The max solution tree defining this bound consists of nodes $a, b, c, d, n, f, g, h, i, j, k, l$ and o.

Fourth Pass: (see figure 7)
This is the last pass of SSS*, in which the upper bound cannot be lowered. Γ cases 1 and 3 back up 35 to the root. Again, we refer to [30] for the details of the SSS* part of the example.

In the Alpha-Beta case, a call with window $\langle f^+ - 1, f^+\rangle$, or Alpha-Beta$(a, 34, 35)$, is performed. In this pass we will not find a fail low as usual, but a fail high with return value 35. The return value is now a lower bound, backed-up by a min solution tree.

How does Alpha-Beta traverse this min solution tree? The search follows the critical path a, h, i, l and o. At node l, both its children immediately return without

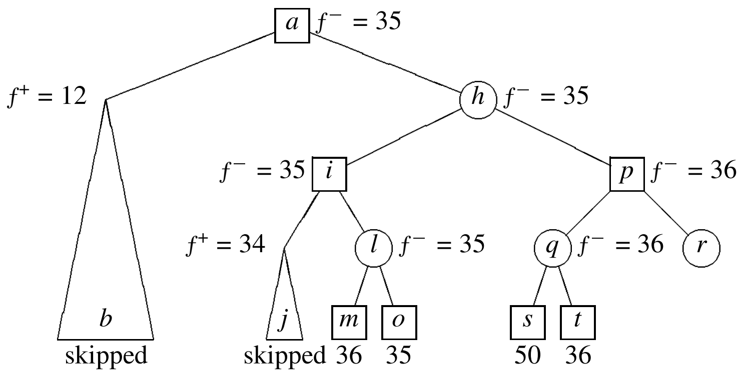

Figure 7: Pass 4

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

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

Figure 8: SSS* and DUAL* as a Sequence of Alpha-Beta Searches

having been evaluated; the value is retrieved from storage. Note that the previous pass stored an f^+ value for l, while this pass will store an f^-. The value of l does not change, j's bound of 34 precludes it from being searched, so i's value remains unchanged. Node i cannot lower h's value ($g > \alpha, 35 > 34$, no cutoff occurs), so the search explores p. Node p expands q which, in turn, searches s and t. Since p is a maximizing node, the value of q (36) causes a cutoff: $g \not< \beta$, node r is not searched. Both of h's children are ≥ 35. Node h returns 35, and so does a. Node a was searched attempting to show whether its value was $<$ or ≥ 35. h provides the answer: greater than or equal. This call to Alpha-Beta fails high, meaning we have a lower bound of 35 on the search. The previous call to Alpha-Beta established an upper bound of 35. Thus the minimax value of the tree is proven to be 35.

We see that nothing special is needed to have Alpha-Beta traverse the min solution tree $a, h, i, l, m, o, p, q, s$ and t. The ordinary cutoff decisions cause its traversal, when $\alpha = f^+(a) - 1$ and $\beta = f^+(a)$.

In the previous four passes we called Alpha-Beta with a special search window to have it emulate SSS*. This sequence of calls, creating a sequence of fail lows until the final fail high, can be captured in a single loop, given by the pseudo-code of figure 8. The reformulation is called MT-SSS*.

One of the problems with Stockman's original SSS* formulation is that it is hard to understand what is "really" going on. It is difficult to create an understanding in terms of concepts above the level of which Γ case happens when and does what. Part of the reason is the iterative nature of the algorithm. This has been the motivation behind the development of other algorithms, notably RecSSS* [4] and SSS-2 [31], which are recursive formulations of SSS*. Although clarity is a subjective issue, it seems simpler to express SSS* in terms of a well-understood algorithm (Alpha-Beta), rather than inventing a new formulation. We think that comparing the codes in figures 1 and 8 shows why we believe to have made the algorithm easier to understand. Furthermore, figure 8 also gives the code for our reformulation of DUAL*, called MT-DUAL*, showing the versatility of this formulation. In section 4 we will pursue this point by presenting a generalization of these two codes.

3 All About Storage

The literature portrays storage as the biggest problem with SSS*. The way it was dealt with in Stockman's original formulation gave rise to two points of criticism:

1. *SSS* is slow.* Some operations on the sorted OPEN list have non-polynomial time complexity. In particular, measurements show that the purge operation of Γ case 1 consumes about 90% of SSS*'s runtime [24].

2. *SSS* has unreasonable storage demands.* Stockman states that his OPEN list needs to store at most $w^{\lceil d/2 \rceil}$ entries for a game tree of uniform branching factor w and uniform depth d—the number of leaves of a max solution tree. In the example we also saw that a single max solution tree is manipulated. (In contrast, DUAL* requires $w^{\lfloor d/2 \rfloor}$ entries, the number of leaves of a min solution tree.) This is usually perceived as being unreasonably large storage requirements.

Several alternatives to the SSS* OPEN list have been proposed. One solution implements the storage as an unsorted array, alleviating the need for the costly purge operation by overwriting old entries (RecSSS* [3, 4, 43]). By organizing this data as an implicit tree, there is no need to do any explicit sorting, since the principal variation can be traversed to find the critical leaf. Another alternative is to use a pointer-based tree, the conventional implementation of a recursive data structure.

Our solution is to extend Alpha-Beta to include the well-known transposition table (see, for example, section 3.1 or [23]). As long as the transposition table is large enough to store at least the min or max solution trees[3] that are essential for the efficient operation of MT-SSS* and MT-DUAL*, it provides for fast access and efficient storage. MT-SSS* will operate when the table is too small, at the cost of extra re-expansions.

The flexibility of the transposition table allows experiments with different memory sizes. In section 3.3 we will see how big the transposition table should be for MT-SSS* to function efficiently. That section presents experimental data addressing the storage concerns of SSS*. Many single and double agent search programs include iterative deepening and transposition tables [19, 42]. They are also used in our experiments and are briefly described below.

3.1 Transposition Tables and Iterative Deepening

In many application domains of minimax search algorithms, the search space is a graph, whereas minimax-based algorithms are suited for *tree* search. Transposition

[3]This includes the direct children of nodes in the max solution tree. These can be skipped by optimizations in the Alpha-Beta code, in the spirit of what Reinefeld has done for Scout [40, 41].

tables (TT) are used to enhance the efficiency of tree-search algorithms by preventing the re-expansion of children with multiple parents [23, 47]. A transposition table is a hash table in which searched nodes (barring collisions, the search tree) are stored. The tree-search algorithm is modified to look in this table before it searches a node and, if it finds the node, uses the value instead of searching. In application domains where there are many paths leading to a node, this scheme leads to a substantial reduction of the search space. (Although technically incorrect, we will stick to the usual terminology and keep using terms like minimax *tree* search.)

A potential drawback of most transposition table implementations is that they do not handle hash-key collisions well. In [35, 39] it is shown that this is not a problem in practice.

Most game-playing programs use iterative deepening [23, 47, 50]. It is based on the assumption that a shallow search is a good approximation of a deeper search. It starts off by doing a depth one search, which terminates almost immediately. It then increases the search depth step by step, each time restarting the search over and over again. Due to the exponential growth of the tree the former iterations usually take a negligible amount of time compared to the last iteration. Among the benefits of iterative deepening (ID) in game-playing programs are better move ordering (explained in the next paragraph), and advantages for tournament time control information. (In the area of one player games it is mainly used as a way of reducing the space complexity of best-first searches [19].)

Transposition tables are often used in conjunction with iterative deepening to achieve a partial move ordering. The search value and the branch leading to the highest score (best move) are saved for each node. When iterative deepening searches one level deeper and revisits nodes, the move suggested by the transposition table (if available) is searched first. Since we assumed that a shallow search is a good approximation of a deeper search, this best move for depth d will often turn out to be the best move for depth $d + 1$ too. Good move ordering increases the pruning power of algorithms like Alpha-Beta and SSS*.

Transposition tables in conjunction with ID are typically used to enhance the performance of algorithms in two ways:

1. improve the quality of the move ordering and
2. detect when different paths through the search space transpose into the same state, to prevent the re-expansion of that node.

In the case of an algorithm in which each ID iteration performs multiple passes over the search tree, like MT-SSS* and MT-DUAL*, there is an additional use for the TT:

3. prevent the re-search of a node that has been searched in a previous pass, in the *current* ID iteration.

3.2 Experiment Design

In our reformulation, MT-SSS* uses a standard transposition table to store previous search results. If that table is too small, previous results will be overwritten, requiring occasional re-searches. A search using a small table will still yield the correct minimax value, although the number of leaf expansions may be high. To test the behavior of our algorithm, we experimented with different transposition table sizes for MT-SSS* and MT-DUAL*.

The questions we want to see answered are: "Does SSS* fit in memory in practical situations" and "How much memory is needed to out-perform Alpha-Beta?". We used iterative deepening versions of MT-SSS* and Alpha-Beta, since these are used in practical applications too. The experiments were conducted using game-playing programs of tournament quality. For generality, our data has been gathered from three programs: Chinook (checkers) [49], Keyano (Othello) [5] and Phoenix (chess) [46]. With these programs we cover the range from low to high branching factors. All three programs are well known in their respective domain. The only changes we made to the programs was to disable search extensions and forward pruning, to ensure consistent minimax values for the different algorithms. For our experiments we used the original program author's transposition table data structures and code, without modification. 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. Chinook and Phoenix use dynamic ordering based on the history heuristic [47], while Keyano uses static move ordering.

The Alpha-Beta code given in figure 2 differs from the one used in practice in that the latter usually includes two details, both of which are common practice in game-playing programs. The first is a search depth parameter. This parameter is initialized to the depth of the search tree. As Alpha-Beta descends the search tree, the depth is decremented. Leaf nodes are at depth zero. The second is the saving of the best move at each node. When a node is revisited, the best move from the previous search is always considered first.

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, each data point was averaged over 20 test 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 runs were performed on a larger test set to check that the test data did not cause anomalies (the data set size is consistent with [46]). All three programs ran to a depth so that all searched roughly for the same amount of time. The search depths reached by the programs vary greatly because of the differing branching factors. In checkers, the average branching factor is approximately 3 (there are typically 1.2 moves in a capture position while roughly 8 in a non-capture position), in Othello 10 and in chess 36. Because of the low branching factor Chinook was able to search to depth 17, iterating two ply at a time. Keyano searched to 10 ply and Phoenix to 8, both one ply at a time.

3.3 Results

Figures 9 and 10 show the number of leaf nodes expanded by ID MT-SSS* and ID MT-DUAL* relative to ID Alpha-Beta as a function of transposition table size (number of entries in powers of 2). The graphs show that for small transposition tables, Alpha-

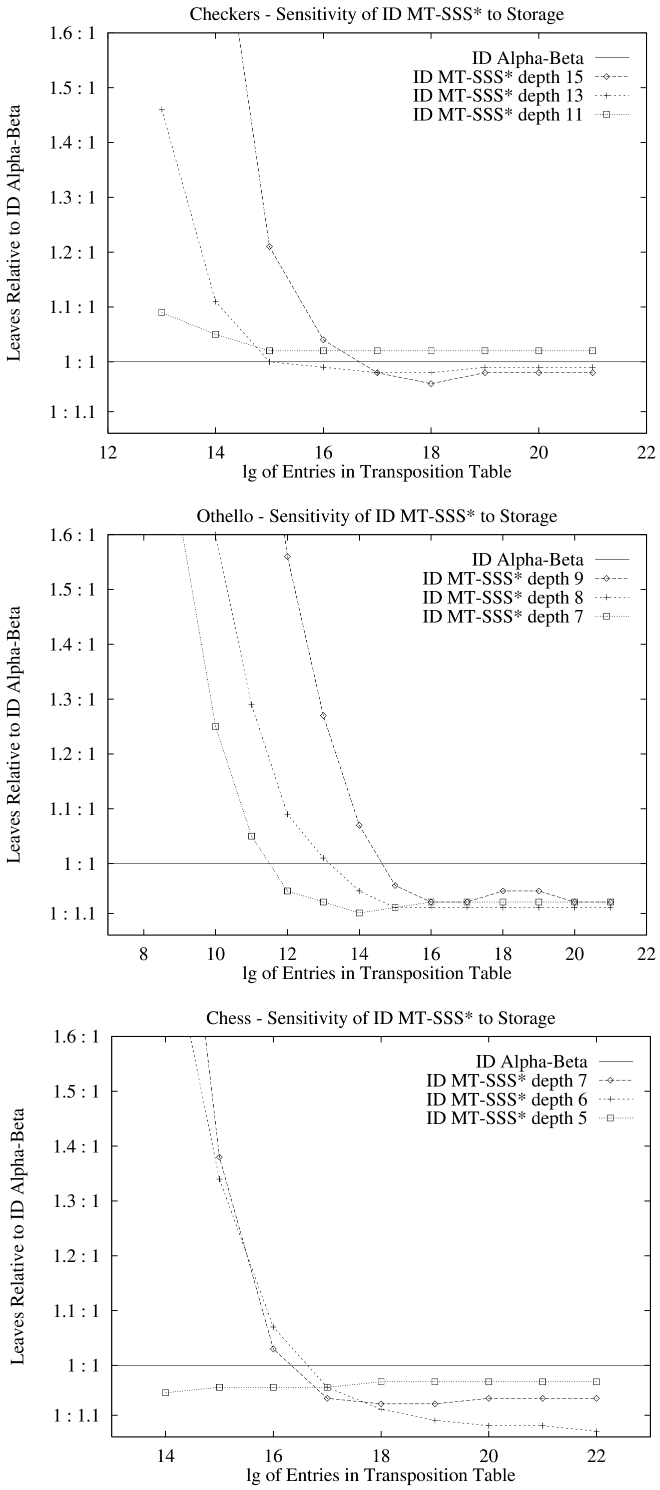

Figure 9: Leaf Node Count ID MT-SSS*

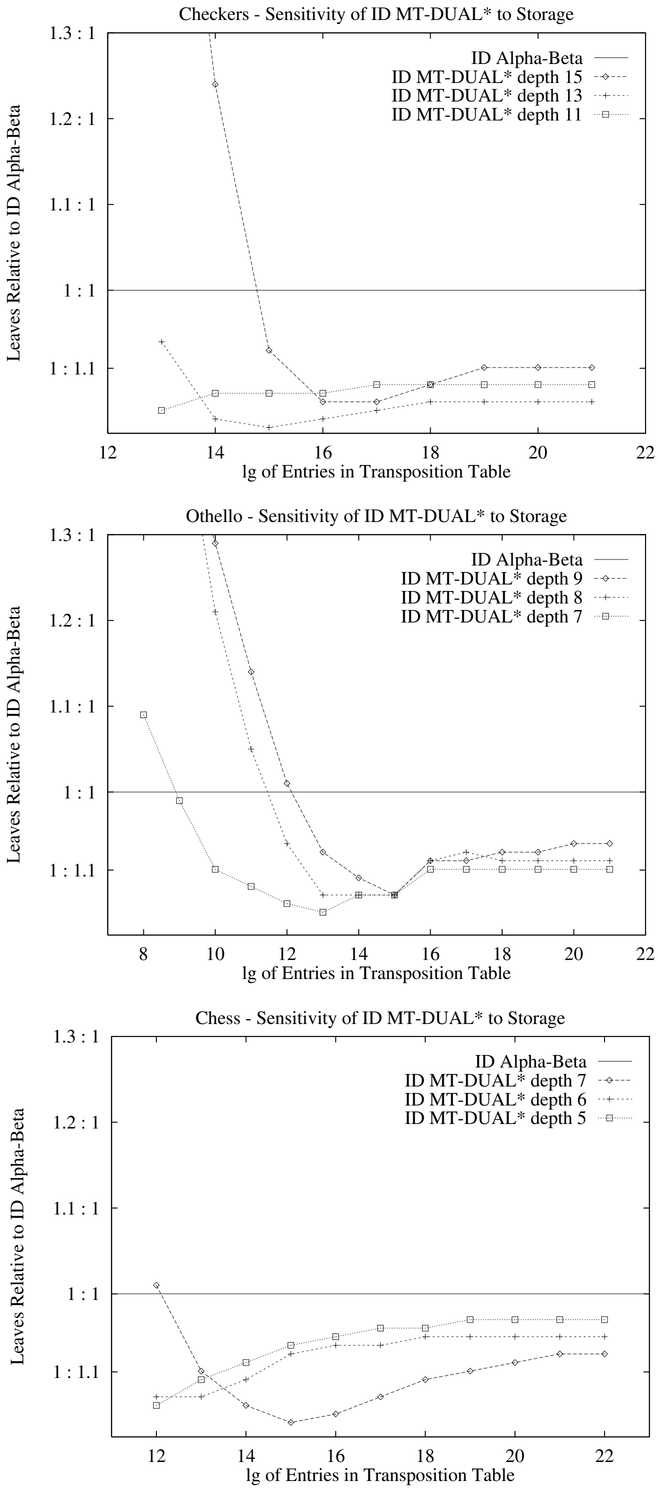

Figure 10: Leaf Node Count ID MT-DUAL*

Beta out-performs MT-SSS*, and for very small sizes it out-performs MT-DUAL* too. However, once the storage reaches a critical level, MT-SSS*'s performance levels off and is generally better than Alpha-Beta. The graphs for MT-DUAL* are similar to those of SSS*, except that the lines are shifted to the left.

Simple calculations and the empirical evidence leads us to disagree with authors stating that $O(w^{\lceil d/2 \rceil})$ is too much memory for practical purposes [17, 24, 26, 41, 45, 51]. Further, many applications have transpositions that can reduce the search effort by a large factor (9 for checkers at depth 15, 4 for chess at depth 9, and 2 for Othello at depth 9 [35]). For present-day search depths in applications like checkers, Othello and chess, using present-day memory sizes, we see that MT-SSS*'s search trees fit in the available memory. For most real-world game-playing programs, a transposition table size of less than 2^{20} entries will be more than adequate for MT-SSS* under tournament conditions.

The graphs provide a clear answer to the main question: SSS* fits in memory, for practical search depths in games with both narrow and wide branching factors. It out-performs Alpha-Beta when given a reasonable amount of storage. However, the original SSS* formulation does not work when there is insufficient storage to hold the OPEN list. The MT-SSS* reformulation benefits from the flexibility provided by the transposition tables, allowing the program to work correctly with any amount of memory. In other words, memory only affects efficiency, not correctness.

The graphs support the theory that says that MT-SSS* is constantly refining a single max solution tree. As soon as there is enough memory to store most of the max solution tree, MT-SSS* runs smoothly in that it does not have to re-expand parts of the tree that it has searched in previous passes. The graphs also support the notion that MT-DUAL* needs less memory, since it manipulates a (smaller) min solution tree ($O(w^{\lfloor d/2 \rfloor})$ versus $O(w^{\lceil d/2 \rceil})$ for max solution trees).

We can conclude from the experiments that MT-SSS* and MT-DUAL* are practical alternatives to Alpha-Beta, where the transposition table size is concerned. However, the experiments also made clear that there are a number of minor issues that are not yet fully understood. For brevity, these issues are discussed elsewhere [35].

3.4 MT-SSS is a Practical Algorithm*

The introduction cited two storage-related drawbacks of SSS*. The first is the excessive memory requirements. We have shown that this is solved in practice.

The second drawback, the inefficiencies incurred in maintaining the OPEN list, specifically the sort and purge operations, was addressed in the RecSSS* algorithm [4, 43]. Both MT-SSS* and RecSSS* store interior nodes and overwrite old entries to solve this. The difference is that RecSSS* uses a restrictive data structure to hold the OPEN list that has the disadvantages of requiring the search depth and width be known *a priori*, and having no support for transpositions. Programming effort (and ingenuity) are required to make RecSSS* usable for high-performance game-playing programs.

In contrast, since most game-playing programs already use Alpha-Beta and transposition tables, the effort to implement MT-SSS* consists only of adding a simple driver routine (figure 8). Implementing MT-SSS* is as simple (or hard) as implementing Alpha-Beta. All the familiar Alpha-Beta enhancements (such as iterative deepening, transpositions and dynamic move ordering) fit naturally into our new framework with no practical restrictions (variable branching factor, search extensions and forward pruning, for example, cause no difficulties).

In MT-SSS*, interior nodes are accessed by fast hash table lookups, to eliminate the slow OPEN list operations. Execution time measurements (not shown) confirm that in general the run time of MT-SSS* and MT-DUAL* are proportional to the leaf count, as shown in figure 9 and 10, showing that they are a few percent faster than Alpha-Beta. However, in some programs where interior node processing is slow, the high number of tree traversals by MT-SSS* and MT-DUAL* can have a noticeable adverse effect. For real applications, in addition to leaf node count, the total node count should also be checked (see section 5).

Keeping this point in mind, we conclude that SSS* and DUAL* have become practical, understandable, algorithms, when expressed in the new formulation.

4 Memory-enhanced Test: a Framework

This section introduces a generalization of the ideas behind MT-SSS*, in the form of a new framework for best-first minimax algorithms. To put it succinctly: this framework uses *depth-first* procedures to implement *best-first* algorithms. Memory is used to pass on previous search results to later passes, allowing selection of the "best" nodes based on the available information from previous passes.

We can construct a generalized driver routine to call MT repeatedly. Recall that MT(n, γ) is equivalent to Alpha-Beta($n, \gamma - 1, \gamma$) using storage. One idea for such a driver is to start at an upper bound for the minimax value, $f^+ = +\infty$. Subsequent calls to MT can lower this bound until the minimax value is reached, as shown in figure 8.

Having seen the two drivers for MT in figure 8, 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. The driver code can be parameterized so that one piece of code can construct a variety of algorithms. The three parameters needed are:

- ***n*** The root of the search tree.
- ***first*** The *first* starting bound for MT.
- ***next*** A search has been completed. Use its result to determine the *next* bound for MT.

Using these parameters, an algorithm using our MT driver, MTD, can be expressed as *MTD(n, first, next)*. The last parameter is not a value but a piece of code. The corresponding pseudocode can be found in figure 11. A number of interesting algorithms can easily be constructed using MTD, of which we present the following examples.

- MTD($n, +\infty, bound := g$)
  This is just MT-SSS*. For brevity we call this driver MTD($+\infty$).
- MTD($n, -\infty, bound := g + 1$)
  This is MT-DUAL*, which we refer to as MTD($-\infty$).
- MTD(n, approximation, **if** $g < bound$ **then** $bound := g$ **else** $bound := g + 1$)
  Rather than arbitrarily using an extreme value as a starting point, any information on where the value is likely to lie can be used as a better approximation.

```
function MTD(n, first, next) → f;
    f⁺ := +∞; f⁻ := −∞;
    bound := g := first;
    repeat
        g := MT(n, bound);
        if g < bound then f⁺ := g else f⁻ := g;
        /* The next operation must set bound */
        next;
    until f⁻ = f⁺;
    return g;
```

```
function MTD(n, f) → f;
f⁺ := +∞; f⁻ := −∞;
if f = −∞ then bound := f + 1 else bound := f;
repeat
    g := MT(n, bound);
    if g < bound then f⁺ := g else f⁻ := g;

    if g = f⁻ then bound := g + 1 else bound := g;
until f⁻ = f⁺;
return g;
```

Figure 11: A Framework for MT Drivers; The MTD(f) instance

(This assumes a relation between start value and search effort that is discussed in section 5.2.3.) Given that iterative deepening is used in many application domains, the obvious approximation for the minimax value is the result of the previous iteration. This algorithm, which we call MTD(f), can be viewed as starting close to f, and then doing either SSS* or DUAL*, skipping a large part of their search path. The right-hand side of Figure 11 shows the pseudo-code for MTD(f).

- $\text{MTD}\big(n, \lfloor \text{average}(+\infty, -\infty) \rfloor, bound := \lfloor \text{average}(f^+, f^-) \rfloor\big)$
  Since MT can be used to search from above (SSS*) as well as from below (DUAL*), an obvious try is to bisect the interval and start in the middle. Since each pass produces an upper or lower bound, we can take some pivot value in between as the next center for our search. This algorithm, called MTD(bi) for short, bisects the range of interest, reducing the number of MT calls. To reduce big swings in the pivot value, some kind of aspiration searching may be beneficial in many application domains [47].

  Coplan introduced an equivalent algorithm which he named *C** [9]. He does not state the link with best-first SSS*-like behavior, but does prove that C* dominates Alpha-Beta in the number of leaf nodes evaluated, provided there is enough storage. (This idea has also been discussed in [1, 52].)

- $\text{MTD}\big(n, +\infty, bound := \max(f_n^- + 1, g - stepsize)\big)$
  Instead of making tiny jumps from one bound to the next, as in all the above algorithms except MTD(bi), we could make bigger jumps. By adjusting the value of *stepsize* to some suitably large value, we can reduce the number of calls to MT. This algorithm is called MTD(step).

Other MTD variations are possible, such as searching for the best move (not the best value). This idea, put forward by Berliner in his B* algorithm [2], would require a different termination condition for the loop, but otherwise fits straightforwardly into the framework. In [37] we report on tests with this variant.

Note that while all the above algorithms use storage for bounds, not all of them need to save both f^+ and f^- values. MTD($+\infty$), MTD($-\infty$) and MTD(f) refine one solution tree. MTD(bi) and MTD(step) usually refine a union of two solution trees,

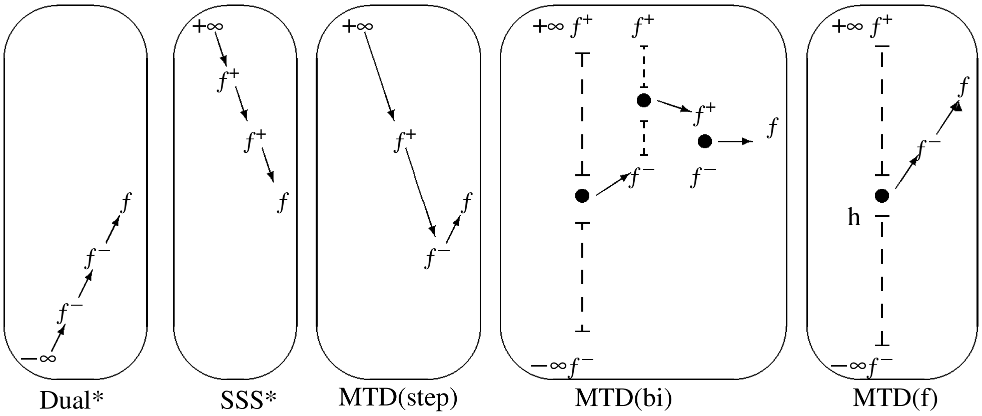

Figure 12: MT-based Algorithms

where nodes on the intersection (the principal variation) should store both an upper and lower bound at the same time (see also [33]). We refer to section 3 for data indicating that these memory requirements are acceptable in practice.

Some of the above instances are new, some are not, and some are small adaptations of known ideas. The value of this framework does not lie so much in the newness of the instances, but in the way how MT enables one to formulate the behavior of a number of algorithms. Formulating a seemingly diverse collection of algorithms into one unifying framework allows us to focus the attention on the fundamental differences in the algorithms. For each algorithm, figure 12 shows how the bounds converge from their start value to the minimax value. The framework allows the reader to see just how similar SSS* and DUAL* really are; they are just special cases of calling Alpha-Beta. 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 like search trees and solution trees.

All the algorithms presented are based on MT. Since MT is equivalent to a null-window Alpha-Beta call (plus storage), they search less nodes than the inferior one-pass Alpha-Beta($-\infty, +\infty$) algorithm. There have been other (less successful) attempts with algorithms that solely use null-window Alpha-Beta searches [27, 46]. Many people have noted that null-window searches have a great potential, since narrow windows usually generate more cutoffs than wider windows [1, 8, 9, 14, 28, 46]. However, it appears that the realization that the transposition table can be used to create algorithms that retain the efficiency of null-window searches by gluing them together without *any* re-expansions—and create an SSS*-like best-first expansion sequence—is new. The notion that the value of a bound on the minimax value of the root of a tree is determined by a solution tree was not widely known among researchers. In this light, it should not be too surprising that the idea of using depth-first null-window Alpha-Beta searches to model best-first algorithms like SSS* is new, despite their widespread use by the game-tree search community.

5 Performance

To assess the performance of the proposed algorithms, a series of experiments was performed. We present data for the comparison of Alpha-Beta, NegaScout, MT-SSS*/MTD($+\infty$), MT-DUAL*/MTD($-\infty$) and MTD(f). Results for MTD(bi) and MTD(step) are not shown; they are inferior to MTD(f).

5.1 Experiment Design

We will assess the performance of the algorithms by counting leaf nodes and total nodes (leaf nodes, interior nodes and nodes at which a transposition occurred). For two algorithms we also provide data for execution time. As before, experiments were conducted with three tournament-quality game-playing programs. All three programs use a transposition table with a maximum of 2^{21} entries. The tests from section 3 showed that the solution trees could comfortably fit in tables of this size for the depths used in our experiments, without any risk of noise due to collisions. We used the original program author's transposition table data structures and code without modification.[4]

Many papers in the literature use Alpha-Beta as the base-line for comparing the performance of other algorithms (for example, [8, 23]). 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 null-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 Chinook, Phoenix and Keyano), it seems more reasonable to use this as our base-line standard. The worse the base-line comparison algorithm chosen, the better other algorithms appear to be. By choosing NegaScout enhanced with aspiration searching (Aspiration NegaScout) as our performance metric, 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*—turn out to be inferior.

Because we implemented the MTD algorithms using MT 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 fixed-depth depth-first and best-first minimax search algorithms where all the algorithms are given identical resources. Through the use of large transposition tables, our base line, Aspiration NegaScout, becomes for all practical purposes as effective as Informed NegaScout [24, 44, 46].

5.2 Results

Figure 13 shows the performance of Chinook, Keyano and Phoenix, respectively, using the number of leaf evaluations as the performance metric. Figure 14 shows the performance of the programs using the total number of nodes in the search tree as the metric (note the different scale). The graphs show the cumulative number of

[4] 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 search extensions. We only had to introduce the MTD driver code.

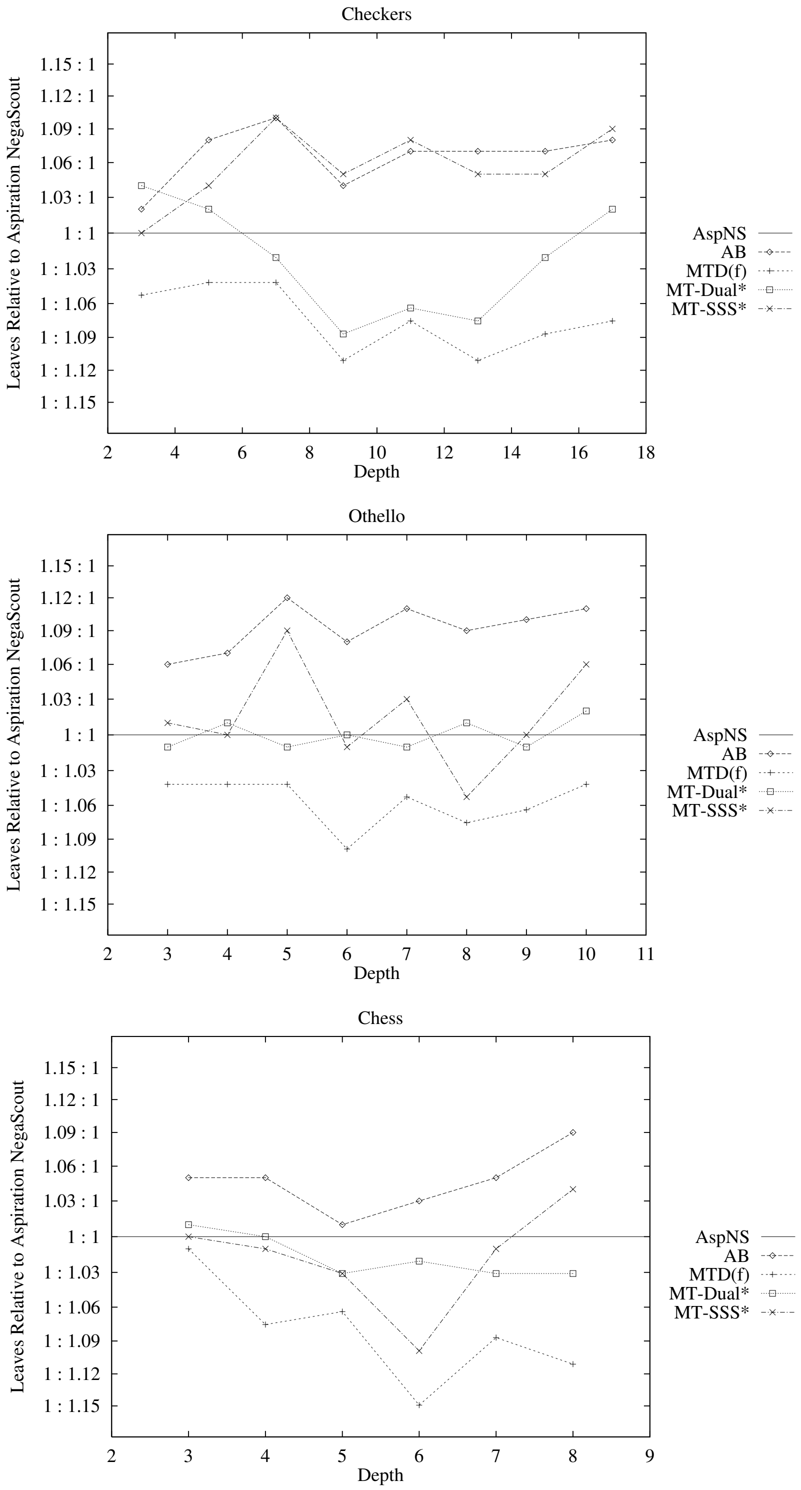

Figure 13: Leaf Node Count

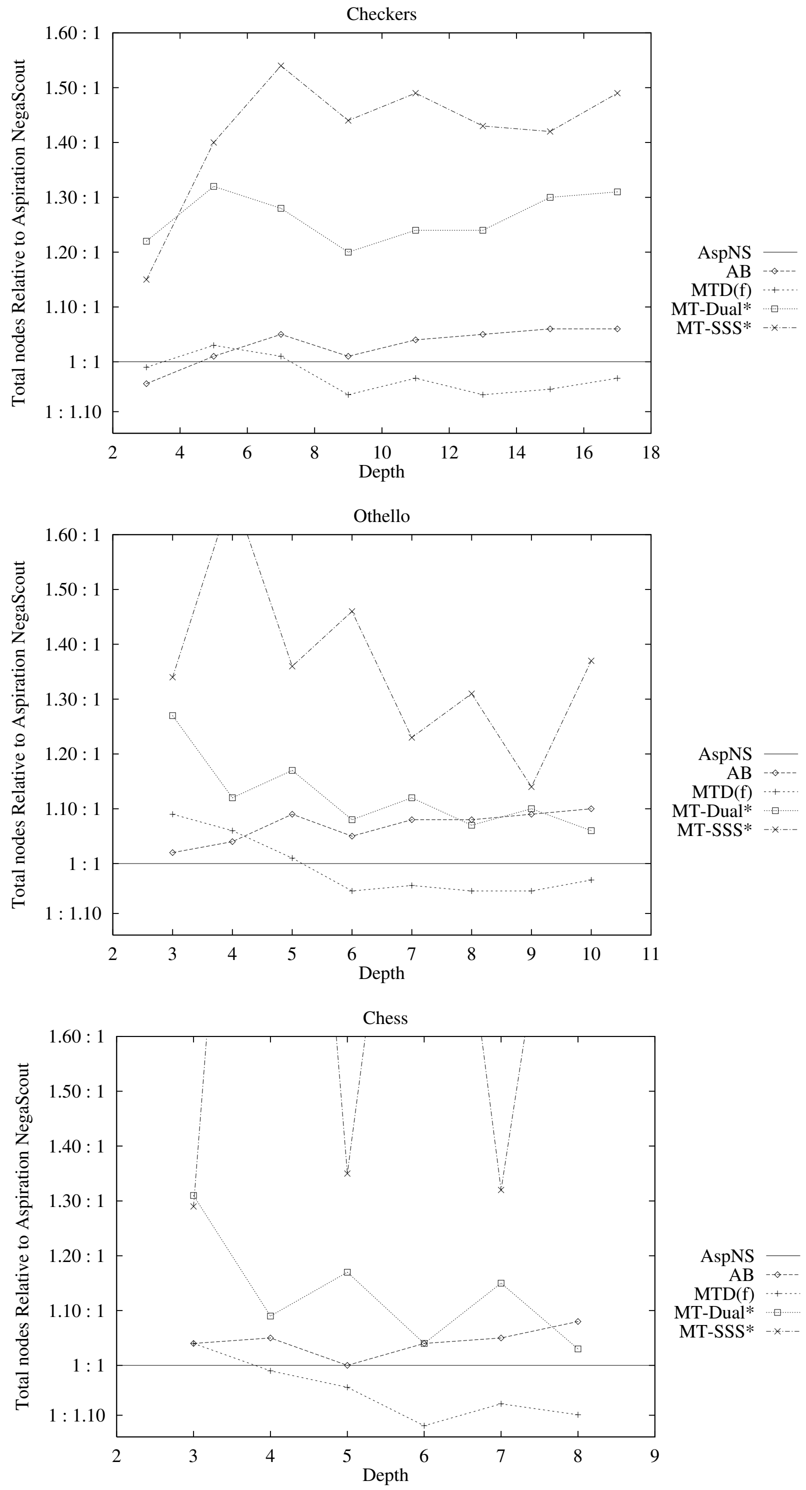

Figure 14: Total Node Count

nodes over all previous iterations for a certain depth (which is realistic since iterative deepening is used) relative to Aspiration NegaScout.

5.2.1 SSS and DUAL**

Looking at the graphs shows that SSS* examines substantially more *total* nodes than Alpha-Beta but, contrary to many simulations, the difference in the number of *leaf* nodes is relatively small. Since game-playing programs use many search enhancements that reduce the search effort—we used only iterative deepening, the history heuristic, and transposition tables—the potential benefits of a best-first search are greatly reduced (see section 6). In practice, SSS* is a small improvement on Alpha-Beta using the leaf node metric (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. In effect, the main advantage of SSS* (point 5 in the introduction) is wrong. Reasons for this will be discussed further in section 6.

Looking at the graphs for total nodes, we see a clear odd/even effect for MT-SSS* and MT-DUAL*. The reason is that the former refines a max solution tree, whereas the latter refines a min solution tree. At even depths the parents of the leaves are min nodes. With a wide branching factor, like in chess, there are many leaves that will initially cause cutoffs for a high bound, causing a return at their min parent (Alpha-Beta's cutoff condition at min nodes $g \leq \alpha$ is easily satisfied when α is close to $+\infty$). It is likely that MT-SSS* will quickly find a slightly better bound to end each pass, causing it to make many traversals through the tree, perform many hash table lookups, and make many calls to the move generator. These traversals show up in the total node count (figure 14) and interior node count (not shown). For MT-DUAL*, the reverse holds. At odd depths, many leaves cause a pass to end at the max parents of the leaves when the bound is close to $-\infty$. (There is room for improvement here by remembering which moves have already been searched. This will reduce the number of hash table lookups, but not the number of visits to interior and leaf nodes.)

As a last point concerning SSS*, we see that for certain depths the iterative deepening version of SSS* expands more leaf nodes than iterative deepening Alpha-Beta in the case of checkers. This result appears to run counter to Stockman's proof that Alpha-Beta is dominated by SSS*. How can this be? No one has questioned the assumptions under which this proof was made. In general, game-playing programs do not perform single fixed-depth searches. Typically, they use iterative deepening and dynamic move ordering to increase the likelihood that the best move is searched first. The SSS* proof implicitly assumes that every time a node is visited, its successor moves will *always* be considered in the same order (Coplan makes this assumption explicit in his proof of C*'s dominance over Alpha-Beta [9]). In appendix B, an example is given that proves the non-dominance of iterative deepening SSS* over iterative deepening Alpha-Beta. We conclude that an advantage of SSS*, its domination of Alpha-Beta (point 4 in the introduction) is wrong in practice.

5.2.2 Aspiration NegaScout and MTD(f)

The results show that Aspiration NegaScout is better than Alpha-Beta. This is consistent with [47] which showed Aspiration NegaScout to be a small improvement over

Alpha-Beta when transposition tables and iterative deepening were used.

Over all three games, the best results are from MTD(f). Not surprisingly, the current algorithm of choice by the game programming community, Aspiration NegaScout, performs well too. The averaged MTD(f) leaf node counts are consistently better than for Aspiration NegaScout, averaging a 5–10% improvement, depending on the game. More surprising is that MTD(f) outperforms Aspiration NegaScout on the total node measure as well. This suggests that MTD(f) is calling MT close to the minimum number of times. Measurements confirm that for all three programs, MTD(f) calls MT about 3 to 6 times per iteration on average. In contrast, the MT-SSS* and MT-DUAL* results are poor compared to Aspiration NegaScout when all nodes in the search tree are considered. Each of these algorithms usually performs hundreds of MT searches. The wider the range of leaf values, the smaller the steps with which they converge, and the more passes they need.

From section 3 we recall that the many MT calls of MT-SSS* and MT-DUAL* make those algorithms perform badly when the transposition table is too small to contain the nodes needed to refine the solution tree. Since MTD(f) performs significantly fewer calls, re-expansions due to insufficient storage are not as big a problem. Compared to one-pass/wide-window Alpha-Beta, the few-pass/null-window MTD(f) performs even better than Alpha-Beta when given less memory than needed for the solution tree. An explanation for this surprising behavior, a best-first algorithm using less memory than a depth-first algorithm, can be found in the literature on NegaScout [30, 41]. For NegaScout, the benefit of the cheaper null-window searches out-weighs a few re-searches, even if there is not enough memory to prevent the re-expansions [8, 24, 28]. This also holds for MTD(f)'s behavior in small-memory situations.

5.2.3 Start Value and Search Effort

This subsection investigates the relation between the size of the search tree, and the start value of a sequence of MT calls.

The biggest difference in the MTD algorithms is their first approximation of the minimax value: SSS*/MTD($+\infty$) is optimistic, DUAL*/MTD($-\infty$) is pessimistic and MTD(f) is realistic. It is clear that starting close to f, assuming integer-valued leaves, should result in faster convergence, simply because there are fewer discrete values in the range from the start value to f. If each MT call at the root expands roughly the same number of nodes, then doing less passes yields a better algorithm. However, MT calls generally do *not* expand the same number of nodes. Since we could not find an analytical solution to the question, we have conducted experiments to test the intuitively appealing idea that starting a search close to f is cheaper than starting far away.

Figure 15 validates the choice of a starting parameter close to the game value (only the results for chess are shown; the results for Othello and checkers are similar [35]). The figure shows the efficiency of the search as a function of the distance of the first guess from the correct minimax value. The data points are given as a percentage of the size of the search tree built by Aspiration NegaScout. To the left of the graph, MTD(f) is closer to DUAL*/MTD($-\infty$), to the right it is closer to SSS*/MTD($+\infty$).

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.

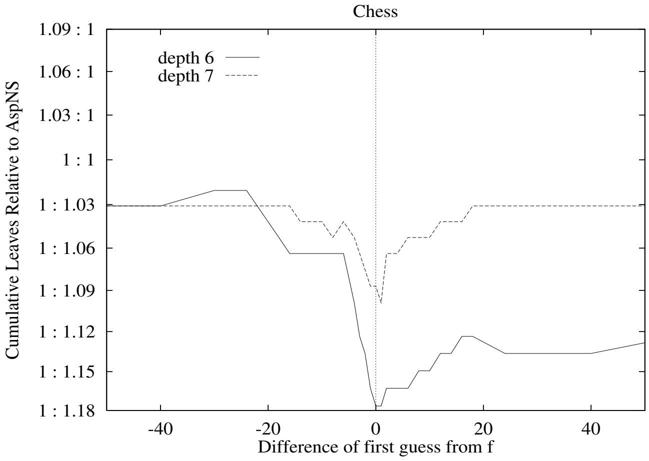

Figure 15: Tree Size Relative to the First Guess f

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.[5] Comparing the graphs in figures 13 and 15, we see that MTD(f) is not achieving its lowest point, so there is room for improvement. Indeed, we found that adjusting the first guess by ± 1 to 4 points for each iteration can improve the results for MTD(f) in terms of leaf count by two to three percentage points. This can be regarded as some form of application dependent fine tuning of the MTD(f) algorithm.

In doing these experiments, the diversity of real-life game trees became apparent. Just as it is not hard to construct a counter-example where a bad first guess expands *less* nodes than a good first guess [35], we encountered some test positions where Aspiration NegaScout performed better than MTD(f).

5.3 Execution Time

The bottom line for practitioners is execution time. This metric may vary considerably for different programs. It is nevertheless included, to give evidence of the potential of MTD(f). We only show the deeper searches, since the relatively fast shallower searches hamper accurate timings. The data shown is from typical runs 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 the program has a considerable impact as well.

The experiments show that for our test programs, the leaf node count is a good indicator of execution time. For Chinook and Keyano, MTD(f) was about 5% faster in execution time than Aspiration NegaScout; for Phoenix we found MTD(f) 9–16% faster. (As pointed out in the previous section, application-dependent tuning can

[5] For programs with a pronounced odd/even oscillation in their score, results are better if the score from two iterations ago is used.

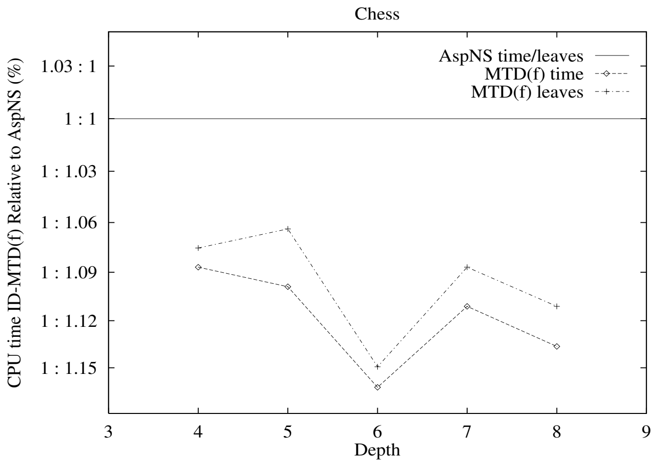

Figure 16: Execution Time

improve this a few percentage points.) For other programs and other machines these results will obviously differ, depending in part on the quality of the start value f and on the test positions used. For programs of lesser quality, the performance difference will be bigger, with MTD(f) out-performing Aspiration NegaScout by a wider margin. Also, since the tested algorithms perform quite 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. The experimental data is better suited to provide insight on, or guide and verify hypotheses about these complexities.

6 Performance Results in Perspective

The introduction summarized the general view on SSS* in five points. Three of these points were drawbacks that were solved in previous sections. The remaining two points were positive ones: SSS* provably dominates Alpha-Beta, and it expands significantly fewer leaf nodes. With the disadvantages of the algorithm solved, the question that remains is: what about the advantages in practice?

The first of the two advantages, theoretical domination, has disappeared. With dynamic move reordering, Stockman's dominance proof for SSS* does not apply. Consequently, experiments confirm that Alpha-Beta can out-search SSS*.

The second advantage was that SSS* and DUAL* expand significantly less leaf nodes than Alpha-Beta. However, modern game-playing programs do a nearly optimal job of move ordering, and employ other enhancements that are effective at improving the efficiency of the search, considerably reducing the advantage of null-window-based best-first strategies. The experiments show that SSS* offers some search tree size advantages over Alpha-Beta for chess and Othello, but not for checkers. These small advantages disappear when comparing to NegaScout. Both SSS* and DUAL* compare unfavorably to Alpha-Beta when all nodes in the search tree are considered.

All algorithms, including MTD(f), perform within a few percentage points of each other's leaf counts. For fixed-depth searches without transposition tables and iterative deepening, simulation results show that SSS*, DUAL* and NegaScout are major improvements over simple Alpha-Beta [17, 24, 26, 41]. For example, one study shows SSS* and DUAL* building trees that are about half the size of those built by Alpha-Beta [24]. This is in sharp contrast to the results reported here. Why is there such a disparity with the previously published work? The reason is the difference between real and artificial minimax trees.

The literature on minimax search abounds with investigations into the relative performance of algorithms. In many publications artificially-generated game trees are used to test these algorithms. We argue that artificial trees are too simple to form a realistic test environment.

Over the years researchers have uncovered a number of interesting features of minimax trees as they are generated in actual application domains like game-playing programs. The following four features of game trees can be exploited by application-independent techniques to increase the performance of search algorithms.

- Variable branching factor.
  The number of children of a node is often not a constant. Algorithms such as proof number and conspiracy number search use this fact to guide the search in a "least-work-first" manner [1, 25, 48].

- Value interdependence between parent and child nodes.
  A shallow search is often a good approximation of a deeper search. This notion is used in techniques like iterative deepening, which—in conjunction with storing previous best moves—greatly increases the quality of move ordering. Value interdependence also facilitates forward pruning based on shallow searches [6].

- Value independence of moves.
  In many domains there exists a global partial move ordering: moves that are good in one position tend to be good in another as well. This fact is used by the history heuristic and the killer heuristic [47].

- Transpositions.
  The fact that the search space is often a graph has lead to the use of transposition tables. In some games, notably chess and checkers, they lead to a substantial reduction of the search effort [36]. Of no less importance is the better move ordering, which dramatically improves the effectiveness of Alpha-Beta.

There are other features which we do not address for reasons of brevity.

The impact of the enhancements is significant: many state-of-the-art game-playing programs are reported to approach their theoretical lower bound, the minimal tree [12, 13, 36, 46]. Regrettably, this high level of performance does not imply that we have a clear understanding of the detailed structure of real-life game trees.

Many points influence the search space in certain ways, although it is not exactly known what the effect is. For example, transpositions, iterative deepening and the history heuristic all cause the tree to be dynamically re-ordered based on information that is gathered during the search. The effectiveness of iterative deepening depends on many factors, such as the strength of the value interdependence, number of cutoffs in the previous iteration, and quality of the evaluation function. The effectiveness of transposition tables depends on game-specific parameters, the size of the transposition table, the search depth, and possibly on move ordering and the phase of the game. The effectiveness of the history heuristic also depends on game-specific parameters, and on the quality of the evaluation function.

The consequence of this is that game trees remain highly complex and dynamic entities, whose structure is influenced by the techniques that make use of (some of) the four listed features. Acquiring data on these factors and the way they relate seems a formidable task. It poses many problems for researchers attempting to reliably model the behavior of algorithms on realistic minimax trees.

All of the simulations that we know of include at most one of the above four features in the trees that they simulate (for example, [3, 4, 8, 10, 15, 17, 24, 26, 41, 43, 51]). In the light of the highly complex nature of real-life game trees, simulations can only be regarded as approximations, whose results may not be accurate for real-life applications. We feel that simulations provide a feeble basis for conclusions on the relative merit of search algorithms as used in practice. The gap between the trees searched in practice and in simulations is large. Simulating search on artificial trees that have little relationship with real trees runs the danger of producing misleading or incorrect conclusions. It would take a considerable amount of work to build a program that can properly simulate real game trees. Since there are already a large number of quality game-playing programs available, we feel that the case for simulations of minimax search algorithms is weak.

An often used approach to have simulations approximate the efficiency of real applications is to increase the quality of move ordering. In the light of what has been said previously, just increasing the probability of first moves causing a cutoff to, say, 98% can only be viewed as a naive solution, that is not sufficient to yield realistic simulations. First, the move ordering is not uniform throughout the tree (in [35] this is further analyzed). Secondly, and more importantly, the good move ordering is not a cause but an effect. It is caused by techniques (like the history heuristic) making use of phenomena like a variable branching factor, value interdependence, value independence and transpositions. Causes and effects appear to be all interconnected, yielding a picture of great complexity that does not look very inviting to disentangle.

As an example of what the differences between real and artificial trees can lead to, let us look at some statements in the literature concerning SSS*. In the introduction we mentioned five points describing the general view on SSS*: it 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. The validity of these points has been examined by numerous researchers in the past [8, 17, 24, 26, 41, 45, 51]. All come to roughly the same conclusion, that the answer to all five points is “true:” SSS* searches less leaves than Alpha-Beta, but it is *not* a practical algorithm. However, two publications contend that points 2 and 3 may be false, indicating that SSS* not only builds smaller trees, but that the problem of the slow operations on the OPEN list may be solved [3, 43]. This paints a favorable picture for SSS*, since the negative points would be solved, while the positive ones would still stand. Probably due to the complexity of the SSS* algorithm the authors restricted themselves to simulations. With our reformulation we were able to use real programs to give the definitive answer on the five questions. In practice *all* five points are wrong, making it clear that, although SSS* is practical, in realistic programs it has *no* substantial advantage over Alpha-Beta, and is even worse than Alpha-Beta-variants like Aspiration NegaScout.

This example may serve to illustrate our point that it is hard to reliably model real trees. In the past we have performed simulations too [10, 24]. We were quite shocked when we found out how easy it is to draw wrong conclusions based on what appeared to be valid assumptions. We hope to have shown in this paper that the temptation of oversimplifying the structure of game trees can and should be resisted. Whether this problem only occurs in minimax search, or also in other domains of Artificial Intelligence, is a question that we leave open.

7 Conclusions

From the original formulation, it is hard to understand how and why SSS* works. It takes a considerable amount of effort to see through the six interlocking Γ cases. SSS* manipulates a single max solution tree and establishes a sequence of upper bounds on the minimax value. In our reformulation, MT-SSS*, we use the concepts of null-window Alpha-Beta search and transposition tables to create this behavior. Null-window searches cut off more nodes than wider-window searches. Just like for NegaScout, the domination of SSS* over Alpha-Beta can be explained by the pruning power of null-window searches.

Unlike NegaScout, MT-SSS* uses *only* null-window searches. At the root of the tree, many repeated calls to MT are performed. Consequently, some form of storage is needed to glue the calls together, preventing excessive node re-expansions. Transposition tables provide an efficient way to do this. They allow for the pruning power of null-window Alpha-Beta calls to be retained over a sequence of searches, and for subsequent Alpha-Beta calls to build on the work of previous ones, constructing a best-first expansion sequence.

We have formulated a framework for null-window-based best-first algorithms. One instance of this framework is MTD(f). It uses an approximation, such as the previous score in an iterative deepening setting, as the start value, instead of $+\infty$ or $-\infty$. In this way the number of null-window searches is dramatically reduced, making the algorithm much less dependent on storage of search results. The few re-expansions are more than offset by the efficiency of the null-window calls. Furthermore, a start value close to the minimax value creates a more efficient search. In our experiments, using three different game-playing programs, MTD(f) is consistently the most efficient search algorithm. The efficiency comes at no extra algorithmic complexity: just a standard Alpha-Beta-based program plus one control loop. By doing away with wider search windows altogether, and using a good start value, our experiments show that one can improve on NegaScout by a wider margin than NegaScout's use of null-windows allowed it to improve on Alpha-Beta.

The experiments allowed us to disspell a myth: none of the algorithms discussed in this article, not even SSS*, needs too much memory for use in practical applications. The solution trees that are traversed fit perfectly well in today's memory sizes.

One of the most interesting outcomes of our experiments is that the performance of all algorithms differs only by a few percentage points. The search enhancements used in high-performance game-playing programs improve the search efficiency to such a high degree that the question of which algorithm to use, be it Alpha-Beta, NegaScout, SSS* or MTD(f), is no longer of prime importance. (For programs of lesser quality, the performance difference will be bigger, with MTD(f) out-performing NegaScout by a wider margin.) A consequence of this is that in practice SSS* is not a significant improvement over Alpha-Beta, is regularly out-performed by NegaScout, and is dominated by MTD(f) in every respect. Hence we believe that SSS* should now become a footnote in the history of game-tree search.

The reason for the difference between our results and simulations is that the trees generated in actual applications are complex. It is hard to create reliable models for simulations. Using artificial trees runs the danger of producing misleading or incorrect results. The field of minimax search is fortunate to have a large number of game-playing programs available. These should be used in preference to artificially-constructed simulations. Future research should try to identify factors that are of importance in real game trees, and use them as a guide in the construction of better search algorithms, instead of artificial models with a weak link to reality.

A Equivalence of MT-SSS* and SSS*

In this appendix we will look deeper into the relation between MT-SSS* and SSS*. The full proof that both formulations are equivalent, in the sense that they expand the same leaf nodes in the same order, can be found in [34]. Here a sketch of the proof is given. The notion of an explicit search history, called the *search tree*, can be found in [16]. Theoretical work on algorithms refining this search tree can be found in [11, 31, 32, 34].

The idea is to insert into the Alpha-Beta code extra operations that insert triples into a LIST. In figure 17 the list-operations between **{*** and ***}** are inserted to show the equivalence of MT-SSS* and Stockman's SSS*. (In implementations of MT-SSS* they should not be included.) The call *List-op*(i, n) means that the operations of Γ case i in figure 1 have to be executed on LIST. The list operations in MT-SSS* cause the same Γ operations to be applied as in Stockman's original formulation.[6] These extra operations cause exactly the same triples to be inserted in the same order as SSS* does for its OPEN list.

In accessing storage, most Alpha-Beta implementations descend to a child node and retrieve any associated bounds, and check whether an immediate cutoff occurs. In our pseudo-code, we have taken a slightly different approach. MT checks whether a child bound will cause a cutoff before calling itself recursively. In this way we save a recursive call, and it simplifies the formal treatment in this appendix. However, there is no conceptual difference; other Alpha-Beta implementations (for example, figure 2) expand the same nodes, and can be used just as well.

In this appendix we will be less rigorous in some places, for reasons of brevity. By following the MT-SSS* code (see figures 8 and 17), one can easily get a feeling just how and where MT-SSS* and SSS* are interrelated.

In studying MT-SSS*, one can distinguish between a *new* call to Alpha-Beta$(n, \gamma - 1, \gamma)$ (equivalent to MT(n, γ)), where node n has never been searched before, and a

[6]For MT-SSS* to traverse the same leaf nodes as SSS*, one bound should be stored at interior nodes. Storing and updating two bounds yields an algorithm that will occasionally expand a few nodes less than SSS* [33].

```
/* MT: storage enhanced null-window Alpha-Beta(n, γ − 1, γ).                                    */
/* n is the node to be searched, γ − 1 is the α parameter, γ is the β parameter of the call.    */
/* 'Store' saves search bound information in memory; 'retrieve' accesses this information.      */
function MT(n, γ) → g;
if n is a leaf node then
    retrieve n.f⁻, n.f⁺; /* non-existing bounds are ± ∞ */
    if n.f⁻ = −∞ and n.f⁺ = +∞ then
        {* List-op(4, n); *}
        g := eval(n);
    else if n.f⁺ = +∞ then g := n.f⁻ else g := n.f⁺;
else if n is a max node then
    g := −∞;
    c := firstchild(n);
    {* retrieve n.f⁻, n.f⁺; if n.f⁺ = +∞ and n.f⁻ = −∞ then List-op(6, n); *}
    /* g ≥ γ causes a beta cutoff (β = γ) */
    while g < γ and c ≠ ⊥ do
        retrieve c.f⁺;
        if c.f⁺ ≥ γ then
            g′ := MT(c, γ);
            {* if g′ ≥ γ then List-op(1, c); *}
        else g′ := c.f⁺;
        g := max(g, g′);
        c := nextbrother(c);
else if n is a min node then
    g := +∞;
    c := firstchild(n);
    {* retrieve n.f⁻, n.f⁺; if n.f⁺ = +∞ and n.f⁻ = −∞ then List-op(5, n); *}
    /* g < γ causes an alpha cutoff (α = γ − 1) */
    while g ≥ γ and c ≠ ⊥ do
        retrieve c.f⁻;
        if c.f⁻ < γ then
            g′ := MT(c, γ);
            {* if g′ ≥ γ then
                if c < lastchild(n) then List-op(2, c); else List-op(3, c); *}
        else g′ := c.f⁻;
        g := min(g, g′)
        c := nextbrother(c);
/* Store one bound per node. */
if g ≥ γ then n.f⁻ := g; store n.f⁻;
    else n.f⁺ := g; store n.f⁺;
return g;
```

Figure 17: Null-window Alpha-Beta Including Storage for Search Results

call where n has been searched before. In the latter case, MT has previously created a "trail" of bounds, forming a max solution tree below n, as we saw in the example of section 2.1.

All but the last top-level call to MT fails low and builds a max solution tree. The last call to MT, which stops the search, fails high and builds a min solution tree. These two cases are used in the following pre-conditions.

Notation: $T(n)$ is a solution tree rooted at node n, $T^{+}(n)$ is a max solution tree and $T^{-}(n)$ a min solution tree. Sometimes these are abbreviated to T, T^{+} and T^{-} when the meaning is clear. The minimax value of a game tree rooted at node n is called $f(n)$, an upper bound on this value is denoted $f^{+}(n)$ and a lower bound is denoted by $f^{-}(n)$. We define $g = f(T(n))$. An entry in LIST consists of a node, state and merit (value) $\langle n, s, v\rangle$. The state is either *live* or *solved*. When a node is first visited, its children are still unexpanded. It is said to be *open*. When its children are generated, it is called *closed*.

The proof refers to the six Γ operators in figure 1.

In the context of MT-SSS*, we can identify a property in the search tree due to the post-condition of Alpha-Beta given in section 2.1. In the first pass, the *left-most solution tree* with finite g-value is constructed. For the next passes, the following propositions hold. Each follows from an extended version of the post-condition of Alpha-Beta, as can be found in [11, 34], and the fact that Alpha-Beta is called in the **repeat-until** loop of MT-SSS* (figure 8).

1. Before each pass, we have in the search tree the left-most max solution tree with $f^{+}(n) = g(T^{+}) = \gamma$, where the children c of min node n to the left of the current best child have already been searched and have $f^{-}(c) > \gamma$.

2. In each pass, every node n in the search tree that is revisited belongs to T^{+}, with f^{+}-value equal to γ; if n is a min node, children to the left of the only child of n in T^{+} have f^{-}-value $> \gamma$ and will never be revisited.

3. Each nested call MT(n, γ) generates a max solution tree when the search fails low, where the children of min nodes have the same properties as in case 1 above.

Theorem A.1 *During execution of MT-SSS*, the following conditions apply to the calls List-op(i, n) and to the call MT(n, γ):*

- *pre-condition of List-op(i, n):*
  LIST includes a triple $\langle n, state, \gamma\rangle$, being the leftmost triple with maximal merit; the restrictions in Γ case i of SSS are satisfied for this triple;*

- *pre-condition of MT(n, γ):*
  If n is open, then $\langle n, live, \gamma\rangle$ is in LIST and n is the leftmost node in LIST with maximal merit.
  If n is not open, then n is the root of a max solution tree T^{+} with $\gamma = g(T^{+}) = f^{+}(n)$ and every leaf x of T^{+} has status = solved *and merit = $f(x)$. One of the leaves of T^{+} is the leftmost node in LIST with maximal merit; no other descendants of n are included in LIST.*

- *post-condition of MT(n, γ):*
  If the return value of the MT call $< \gamma$, then n is the root of a max solution tree

with the return value of the MT call $= g(T^+) = f^+(n)$ *and every leaf* x *of* T^+ *has status* = solved *and merit* $= f(x)$*; no other descendants of n are included in* LIST*. If the return value of the MT call* $\geq \gamma$*, then* $\langle n, solved, \gamma\rangle$ *is in* LIST*; no other descendants of n are included in* LIST*.*

Proof
For the MT pre- and post-condition, we give a proof by induction. The pre-condition of List-op is proved as a side-effect, yielding the basis for the equivalence proof of MT-SSS* and SSS*.

Precondition of MT(n, γ)
At the start of the first MT call (on an open node n, the root), the pre-condition holds. Assume the pre-condition holds for a call MT(n, γ) with n an open node. By assumption, $\langle n, live, g\rangle$ is in LIST and n is the leftmost node in LIST with maximal merit.

First consider node n being a max node. The restrictions of Γ case 6 hold, and *List-op*$(6, n)$ replaces the triple including n by a series of triples, each including a child of n. A child c is expanded by MT, if the subcalls to brothers b to the left of c have resulted in $\gamma > g'$. By the induction hypothesis, after each call, b is the root of a max solution tree T' and each leaf x has merit $f(x)$. Since $g(T'(b)) = g' < \gamma$, each of these merits is $< \gamma$. It follows that when c has been expanded by MT, $\langle c, live, \gamma\rangle$ is still in LIST and c is the leftmost node with maximal merit. Hence the pre-condition holds for c.

Second, consider node n being a min node. The restrictions of Γ case 5 hold and *List-op*$(5, n)$ causes the pre-condition to be met for the left-most child c of n. As long as each subcall ends with $g' \geq \gamma$, the **while** loop in figure 17 is continued. Before each subcall, a triple $\langle c, live, \gamma\rangle$ is in LIST. After the subcall, the status of this triple is *solved*. For this triple, the left-most one with highest merit, Γ case 2 applies and the related operation replaces this triple by $\langle next(c),\ live,\ \gamma\rangle$. We conclude that the pre-condition also holds for the next children.

Now we treat the case where n is a closed node. The pre-condition of MT holds for the root n in the subsequent passes of MT-SSS* as a consequence of the post-condition of the preceding MT call.

Assume the pre-condition holds for an inner node n. If n is a max node, then n is the root of a max solution tree T^+ with $f^+(n) = g(T^+) = \gamma$ and every leaf x has $f(x) \leq f^+(n)$. When a child c is expanded by MT, $f^+(c) = \gamma$, and every brother b to the left of c has $f^+(b) < \gamma$. Therefore, c is an ancestor of the left-most node in LIST with highest merit. If n is a min node, the only child c of n in T^+ is the left-most child that is expanded by MT. The pre-condition for this child c follows immediately from the pre-condition of n. □

Postcondition of MT
The assumption is made that every subcall to MT satisfies the post-condition. We have three situations. First, consider n as a leaf node. The call *List-op*$(4, n)$ conforms to its pre-condition. On exit, either n is in LIST with status *solved* and *merit* $= f(n) = g < \gamma$, or *merit* $= \gamma \leq f(n) = g$. In both cases, the post-condition holds.

Second, assume n is an inner max node with children c. If every call to MT(c, γ) with return value g' ends with $g' < \gamma$, then $g < \gamma$ on exit and n is the root of a tree $T^+(n)$. Since the leaves of $T^+(c)$ occur in LIST for every c, also the leaves of $T^+(n)$ do so. If at least one subcall ends with $g' = \gamma$, then due to the operation *List-op*$(1, c)$, the post-condition holds for n.

Third, assume n is an inner min node with children c. If at least one call MT(c, γ) return a value g' with $g' < \gamma$, then by the induction hypothesis the leaves of $T^+(c)$ are in LIST, as are the leaves of $T^+(n)$. If all MT-calls end with $g' \geq \gamma$, then after all children have been searched, $\langle last(n), solved, \gamma\rangle$ is in LIST. Due to *List-op*(3, *last*(n)), the post-condition of MT holds. □

Theorem A.2 *MT-SSS* is equivalent to SSS*.*

Proof
Each list operation is always applied to the left-most node in LIST with highest merit. So the operations performed on LIST conform with those of SSS*. The notion *live* is equivalent to *open* in SSS*. Expanding in SSS* is performed by Γ case 4, 5, and 6. The MT-code shows that expanding an open node coincides with a call to list-op 4, 5 or 6. We conclude that MT-SSS* and SSS* expand open nodes in the same order. □

B Non-dominance of Iterative Deepening SSS*

This appendix presents an example to prove that SSS* with dynamic move reordering does not dominate Alpha-Beta. Iterative deepening and move reordering are part of all state-of-the-art game-playing programs. While building a tree to depth d, a node n might consider the moves in the order $1, 2, 3, \ldots, w$. Assume move 3 causes a cutoff. When the tree is re-searched to depth $d+1$, the transposition table can retrieve the results of the previous search. Since move 3 was successful at causing a cutoff previously, albeit for a shallower search depth, there is a high probability it will also work for the current depth. Now move 3 will be considered first and, if it fails to cause a cutoff, the remaining moves will be considered in the order $1, 2, 4, \ldots, w$ (depending on any other move ordering enhancements used). The result is that prior history is used to *change* the order in which moves are considered.

Any form of move ordering violates the implicit preconditions of Stockman's proof. In expanding more nodes than SSS* in a previous iteration, Alpha-Beta stores more information in the transposition table which may later be useful. 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 possible for Alpha-Beta to expand *fewer* leaf nodes than SSS*.

When used with iterative deepening, SSS* does not dominate Alpha-Beta. Figures 18 and 19 prove this point. In the figures, the smaller depth-2 search tree causes SSS* to miss information that would be useful for the search of the larger depth-3 tree. It searches a differently ordered depth-3 tree and, in this case, misses the cutoff at node o found by Alpha-Beta. If the branching factor at node d is increased, the improvement of Alpha-Beta over SSS* can be made arbitrarily large.

That SSS*'s dominance proof does not hold for dynamically ordered trees does not mean that Alpha-Beta is structurally better. If SSS* expands more nodes for depth d, it will probably have more information for the next depth, and it may well out-perform Alpha-Beta again at depth $d + 1$. All it means is that under dynamic reordering the theoretical superiority of SSS* over Alpha-Beta does not apply.

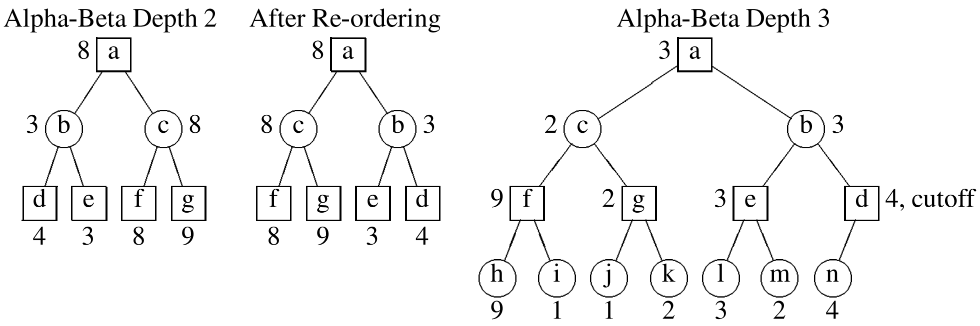

Figure 18: Iterative Deepening Alpha-Beta

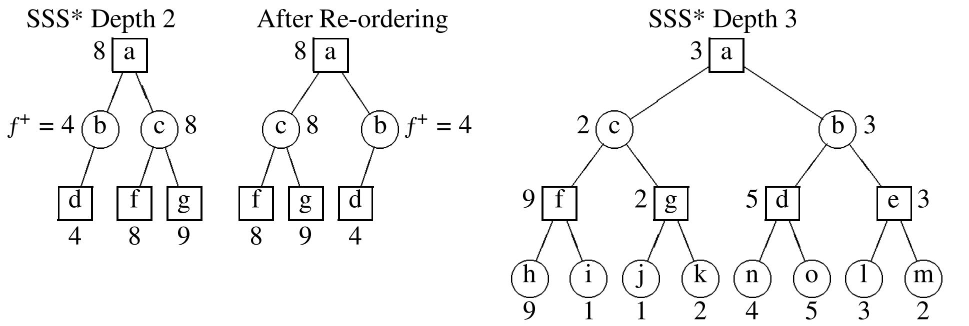

Figure 19: Iterative Deepening SSS*

The smaller the branching factor, the more likely this phenomenon is observed. The larger the branching factor, the more opportunity there is for best-first search to offset the benefits of increased information in the transposition table.

Acknowledgments

This work has benefited from discussions with Mark Brockington (author of Keyano), Yngvi Bjornsson and Andreas Junghanns. An anonymous referee provided valuable feedback. The support of Jaap van den Herik is appreciated. The financial support of the Netherlands Organization for Scientific Research (NWO), the Tinbergen Institute, the Natural Sciences and Engineering Research Council of Canada (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] Hans J. Berliner. The B* tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12:23–40, 1979.

[3] Subir Bhattacharya and A. Bagchi. Unified recursive schemes for search in game trees. Technical Report WPS-144, Indian Institute of Management, Calcutta, 1990.

[4] Subir Bhattacharya and A. Bagchi. A faster alternative to SSS* with extension to variable memory. *Information processing letters*, 47:209–214, September 1993.

[5] Mark Brockington. *Improvements to Parallel Alpha-Beta Algorithms*. PhD thesis, proposal, Department of Computing Science, University of Alberta, Edmonton, Canada, 1994.

[6] Michael Buro. ProbCut: A powerful selective extension of the $\alpha\beta$ algorithm. *ICCA Journal*, 18(2):71–81, June 1995.

[7] Murray Campbell. Algorithms for the parallel search of game trees. Master's thesis, Department of Computing Science, University of Alberta, Canada, August 1981.

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

[9] 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.

[10] Arie de Bruin, Wim Pijls, and Aske Plaat. Solution trees as a basis for game tree search. Technical Report EUR-CS-94-04, Department of Computer Science, Erasmus University Rotterdam, Rotterdam, The Netherlands, May 1994.

[11] 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.

[12] Carl Ebeling. *All the Right Moves*. MIT Press, Cambridge, Massachusetts, 1987.

[13] Rainer Feldmann, Peter Mysliwietz, and Burkhard Monien. *A Fully Distributed Chess Program*, pages 1–27. Ellis Horwood, 1990. Editor: Don Beal.

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

[15] Feng-Hsiung Hsu. *Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, February 1990.

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

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

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

[19] Richard E. Korf. Iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[20] Richard E. Korf and David W. Chickering. Best-first minimax search: Othello results. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, volume 2, pages 1365–1370. American Association for Artificial Intelligence, AAAI Press, August 1994.

[21] Vipin Kumar and Laveen N. Kanal. Parallel branch-and-bound formulations for AND/OR tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):768–778, November 1984.

[22] 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.

[23] T. Anthony Marsland and Murray Campbell. Parallel search of strongly ordered game trees. *Computing Surveys*, 14(4):533–551, December 1982.

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

[25] David Allen McAllester. Conspiracy numbers for min-max searching. *Artificial Intelligence*, 35:287–310, 1988.

[26] 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.

[27] Wolfgang Nagl. Best-move-proving: A fast game-tree searching program. In D.N.L. Levy and D.F. Beal, editors, *Heuristic Programming in Artificial Intelligence—The first computer olympiad*, pages 255–272. Ellis Horwood, 1989.

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

[29] 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.

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

[31] 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.

[32] Wim Pijls and Arie de Bruin. Searching informed game trees. Technical Report EUR-CS-92-02, Erasmus University Rotterdam, Rotterdam, NL, October 1992. Extended abstract in Proceedings CSN 92, pp. 246–256, and Algorithms and Computation, ISAAC 92 (T. Ibaraki, ed), pp. 332–341, LNCS 650.

[33] Wim Pijls and Arie de Bruin. SSS*-like algorithms in constrained memory. *ICCA Journal*, 16(1):18–30, March 1993.

[34] 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, Rotterdam, The Netherlands, April 1995.

[35] Aske Plaat. *Research Re: search & Re-search*. PhD thesis, Erasmus University, Rotterdam, The Netherlands, 1996. Forthcoming.

[36] 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.

[37] 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.

[38] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. Best-first fixed-depth game-tree search in practice. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, volume 1, pages 273–279, August 1995.

[39] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. A minimax algorithm better than Alpha-Beta? No and Yes. Technical Report 95-15, University of Alberta, Department of Computing Science, Edmonton, AB, Canada T6G 2H1, May 1995.

[40] Alexander Reinefeld. An improvement of the Scout tree-search algorithm. *ICCA Journal*, 6(4):4–14, 1983.

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

[42] Alexander Reinefeld and T. Anthony Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, July 1994.

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

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

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

[46] 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.

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

[48] Jonathan Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43:67–84, 1990.

[49] Jonathan Schaeffer, Joseph Culberson, Norman Treloar, Brent Knight, Paul Lu, and Duane Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2-3):273–290, 1992.

[50] David Slate and Larry Atkin. Chess 4.5 — The Northwestern University chess program. In P.W. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118, New York, 1977. Springer-Verlag.

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

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