Chapter 3

The MT Framework

The previous chapter showed how the null-window idea is used in NegaScout. This chapter takes the idea further. In section 3.1 we present a reformulation of SSS* that uses null-window Alpha-Beta calls. It has the advantage of solving a number of obstacles that have hindered SSS*'s use in game-playing programs. 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 in figure 3.1.

In section 3.2 we will generalize the ideas behind MT-SSS* into a new framework that elegantly ties together a number of algorithms that are perceived to be dissimilar.

Results from this chapter have been published in [111, 113].

3.1 MT-SSS*

SSS* finds the minimax value through a successive lowering of an upper bound. In each of a number of passes it selects nodes in a best-first order. The upper bound of each pass is defined by a max solution tree, whose minimax value is the maximum of its leaves. The "best" node to expand next is a brother of the left-most leaf defining this value—the critical leaf at the end of the principal variation. (See also figure 2.18.)

Here we reformulate SSS* as a sequence of upper bounds, generated by null-window Alpha-Beta calls. The basis of our reformulation of SSS* is the realization that a conventional version of Alpha-Beta that uses transposition tables, such as the one in figure 2.15 or in [81, 82], can be used to traverse the principal variation. The node that it will expand next (if called on the same max solution tree with the right search window) is precisely the one that is "SSS*-best," the one that SSS* would select.

Alpha-Beta's postcondition on page 18 shows that for a fail low, a max solution tree is constructed and an upper bound is returned. Assuming that the solution trees that Alpha-Beta and SSS* generate are the same, then all the ingredients for a reformulation of SSS* are available. The code in figure 3.1 implements the idea of calling Alpha-Beta to generate a sequence of upper bounds. By using a window of $\langle \beta - 1, \beta \rangle$, where β is the previous upper bound, a return value of $g \leq \beta - 1$ is a fail low where g equals the new upper bound, and a return value of $g \geq \beta$ is a fail high where g equals a lower bound.

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

Figure 3.1: SSS* as a Sequence of Memory-Enhanced Alpha-Beta Searches

(Since β is always an upper bound, the $g > \beta$ part of $g \geq \beta$ will never occur, assuming there are no search inconsistencies, see section 2.2.3.). The example in appendix A.3 shows how Alpha-Beta constructs max solution trees by selecting nodes best-first. A full formal proof of the equivalence of MT-SSS* and SSS* can be found in [106], an extended outline in [113, 112]. An "informal proof" describing in detail the link between the six SSS* Γ cases and the Alpha-Beta code can be found in appendix B.

MT-SSS* only uses null-windows, so there really is no need to carry around the two α and β bounds. Figure 3.2 shows a null-window-only version of Alpha-Beta, enhanced with memory. Pearl introduced the procedure Test [96, 97], a routine that tests whether the value of a sub tree lies above or below a certain threshold. We have named our procedure MT, for Memory-enhanced Test. MT returns a bound, not just a Boolean value. It is a *fail-soft* Test.

In accessing storage, most Alpha-Beta implementations descend to a child node, retrieve the bounds stored previously in memory, and check whether an immediate cutoff occurs (see for example the code in figure 2.15). In our pseudo code for MT, we have taken a 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 showing that SSS* and MT-SSS* are equivalent. In figure B.1 a version of MT is shown where the six SSS* list operations are inserted, which is used to show the equivalence. However, there is no conceptual difference. Other Alpha-Beta implementations expand the same nodes, and can be used just as well, as long as values of stored nodes are treated as in figure 2.15.

One of the problems with Stockman's original SSS* formulation is that we found it very hard to understand what is "really" going on. Part of the reason of the problem is the iterative nature of the algorithm. This has been the motivation behind the development of other algorithms, notably RecSSS* [21] and SSS-2 [102], 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. Comparing the figures 2.17 and 3.1 shows why we believe to have solved this point. Furthermore, figure 3.3, which gives the code for our reformulation of DUAL*, shows the versatility of this formulation. In section 3.2 we

```
/* MT: storage enhanced null-window Alpha-Beta(n, γ − 1, γ).          */
/* n is the node to be searched, γ − 1 is α and γ is β in the call.   */
/* 'Store' saves search bound information in memory;                  */
/* 'Retrieve' accesses these bounds.                                  */
function MT(n, γ) → g;
        if n = leaf then
            retrieve n.f⁻, n.f⁺; /* non-existing bounds are ± ∞ */
            if n.f⁻ = −∞ and n.f⁺ = +∞ then
                g := eval(n);
            else if n.f⁺ = +∞ then g := n.f⁻; else g := n.f⁺;
        else if n = max then
            g := −∞;
            c := firstchild(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);
                else g′ := c.f⁺;
                g := max(g, g′);
                c := nextbrother(c);
        else /* n is a min node */
            g := +∞;
            c := firstchild(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);
                else g′ := c.f⁻;
                g := min(g, g′);
                c := nextbrother(c);
        /* Store one bound per node. Delete any old bound (see page 53). */
        if g ≥ γ then n.f⁻ := g; store n.f⁻;
                else n.f⁺ := g; store n.f⁺;
        return g;
```

Figure 3.2: MT: Null-window Alpha-Beta With Storage for Search Results

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

Figure 3.3: DUAL* as a Sequence of Memory-Enhanced Alpha-Beta Searches

pursue this point further by presenting a generalization of these codes.

3.2 Memory-enhanced Test: A Framework

The relatively simple and well-known concept of null-window Alpha-Beta search is powerful or versatile enough to create the best-first behavior of a complicated algorithm. 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.

The previous section showed how null-window Alpha-Beta searches can be used as an efficient method to compute bounds, and thus form the core of our SSS* reformulation. So far, we have discussed the following two mechanisms to be used in building efficient algorithms:

1. null-window searches cut off more nodes than wide search windows, and

2. we can use storage to glue multiple passes of null-window calls together, so that they can be used to home in on the minimax value, without re-expanding nodes searched in previous passes, creating a best-first expansion sequence.

A general driver routine to call MT repeatedly can be constructed. One idea, SSS*'s 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 3.1.

Having seen the two drivers for MT in figure 3.1 and 3.3, the ideas can be encompassed in a generalized driver routine. The driver provides a series of calls to MT to refine bounds on the minimax value succesively. The driver code can be parameterized so that one piece of code can construct a variety of algorithms. The three parameters needed are:

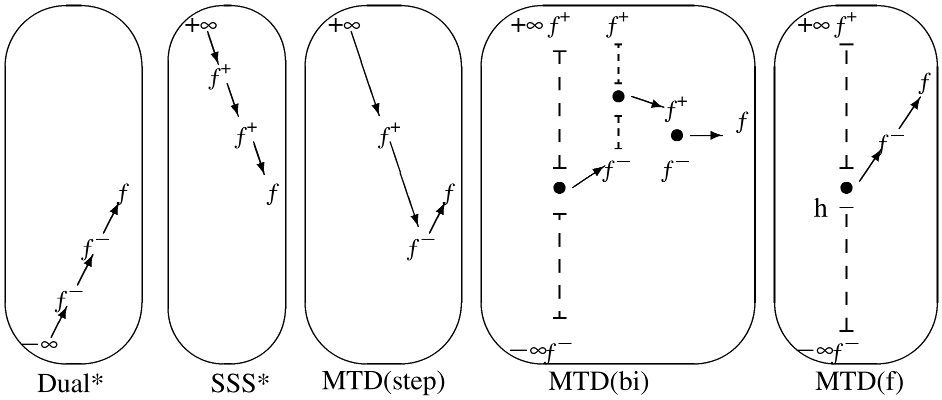

Figure 3.4: MT-based Algorithms

function MTD(n, first, next) $\rightarrow f$;
  $f^+ := +\infty$; $f^- := -\infty$;
  bound := first;
  repeat
    $g := \mathrm{MT}(n, bound)$;
    if $g < bound$ **then** $f^+ := g$ **else** $f^- := g$;
    /* The *next* operation must set the variable *bound* */
    next;
  until $f^- = f^+$;
  return g;

Figure 3.5: A Framework for MT Drivers

- *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 pseudo code of the driver can be found in figure 3.5. A number of interesting algorithms can be constructed using MTD, of which we present the following examples (see also figure 3.4):

function MTD$(n, f) \rightarrow f$;
  $g := f$;
  $f^+ := +\infty$; $f^- := -\infty$;
  repeat
    if $g = f^-$ **then** $\gamma := g + 1$ **else** $\gamma := g$;
    $g :=$ MT(n, γ);
    /* $g :=$ Alpha-Beta$(n, \gamma - 1, \gamma)$ is equivalent */
    if $g < \gamma$ **then** $f^+ := g$ **else** $f^- := g$;
  until $f^- = f^+$;
  return g;

Figure 3.6: MTD(f)

- MTD$(n, +\infty, \textit{bound} := g)$
  This is just MT-SSS*. For brevity we call this driver MTD(+∞).

- MTD$(n, -\infty, \textit{bound} := g + 1)$
  This is MT-DUAL*, which we refer to as MTD(−∞).

- MTD$\big(n, \text{approximation}, \textbf{if } g < \textit{bound} \textbf{ then } \textit{bound} := g \textbf{ else } \textit{bound} := g + 1\big)$
  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. (This assumes a relation between start value and search effort that is discussed in section 4.3.1.) 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. Since the generic MTD code may be confusing at first sight, we give MTD(f)'s pseudo code (slightly reorganized to make it look better) in figure 3.6. This figure also facilitates a direct comparison with figures 3.1 and 3.3.

- MTD$\big(n, \lceil \text{average}(+\infty, -\infty) \rceil, \textit{bound} := \lceil \text{average}(f^+, f^-) \rceil\big)$
  Since MT can be used to search from above (SSS*) as well as from below (DUAL*), an other 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 [128]. Looking at Alpha-Beta's postcondition the bisection idea comes to mind easily. It is further discussed in section 3.3.1.

- MTD$\big(n, +\infty, \textit{bound} := \max(f_n^- + 1, g - \textit{stepsize})\big)$
  Instead of making tiny jumps from one bound to the next, as in MTD(+∞),

MTD($-\infty$) and MTD(f), 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).

- MTD(best)
  If we are not interested in the game value itself, but only in the best move, then a stop criterion suggested by Hans Berliner in the B* algorithm can be used [13]. Whenever the lower bound of one move is not lower than the upper bounds of all other moves, it is certain that this must be the best move. To prove this, we have to do less work than when we try to determine f, since no upper bound on the value of the best move has to be computed. We can use either a disprove-rest strategy (establish a lower bound on one move and then try to create an upper bound on the others) or prove-best (establish an upper bound on all moves thought to be inferior and try to find a lower bound on the remaining move). A major difference between B* and MTD(best) is that we use fixed-depth search, and the bounds are backed-up leaf values, instead of two separate heuristic bounds. The stop criterion in the ninth line of figure 3.5 must be changed to $f^-_{bestmove} \geq f^+_{othermoves}$. Note that this strategy has the potential to build search trees *smaller* than the minimal search tree, because it does not prove the minimax value.

  The indication which move should be regarded as best, and a suitable value for a first guess, can be obtained from a previous iteration in an iterative deepening scheme. This notion can change during the search, which makes for a more complicated implementation. Section 3.3.2 contains a short analysis of the best-move cutoff. In [108] 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 a single solution tree at a time. MTD(bi) and MTD(step) usually refine a union of two solution trees, where nodes on the intersection (the principal variation) should store both an upper and lower bound at the same time (see also a remark on page 53 and [104]). We refer to section 4.1 for data indicating that these memory requirements are acceptable.

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 some of the instances, but in the way how it 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 attention on the fundamental differences in the algorithms (see figure 3.4). For example, the framework allows the reader to see just how similar SSS* and DUAL* really are, that these are just special cases of calling Alpha-Beta. The drivers concisely capture the algorithm differences. MTD offers a high-level paradigm that facilitates the reasoning about issues like algorithm efficiency and memory usage, without the need for low-level details like search trees, solution trees and which node to expand next.

3.2.1 Related Work

Other Work on Null-Windows

All the MTD algorithms 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/wide-window Alpha-Beta($n, -\infty, +\infty$) algorithm.

There have been other attempts with algorithms that solely use null-window Alpha-Beta searches [90, 125]. Many people have noted that null-window searches have a great potential, since tight windows usually generate more cutoffs than wider windows [3, 33, 35, 44, 96, 125]. 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. This idea is supported by the fact that previous algorithms all tried to minimize the number of small-window Alpha-Beta calls in some way, causing those algorithms to make sub-optimal decisions (see section 3.3.2). 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, let alone that this sub-tree fits in memory. When seen in this light, it is not 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.

Other Frameworks

The literature describes two frameworks that generalize best-first and depth-first minimax search—or rather, SSS* and Alpha-Beta. Ibaraki created one such framework, called Gsearch [57]. He concentrated on the informed model where, inspired by the ideas of B*, heuristic upper and lower bounds are available for every node. In the uninformed version—disregarding the heuristic bounds—it is possible to instantiate the framework to both Alpha-Beta and an SSS* variant by choosing an appropriate function for the so-called select rule. Ibaraki has used Gsearch for analyzing domination of algorithms [58]. Pijls and De Bruin [104] show that the SSS*-like Gsearch instance differs slightly from Stockman's SSS*. The Gsearch instance, which they call Maxsearch, is a bit more efficient because it stores both upper *and* lower bounds for every node, wheras SSS* works with only one. This makes that SSS* misses cutoffs in certain special cases [104]. A recursive version of Gsearch, called Rsearch, allows restricted-memory versions of SSS* (or rather, Maxsearch) to be formulated [59]. Rsearch instances typically restrict the search trees to be built to a certain depth, so that a best-first select rule can only work in a part of the tree. Pijls and De Bruin [105] show that when the size of the resident tree is reduced to zero, the resulting Rsearch instance selects the same nodes as Alpha-Beta. Thus, in Rsearch one can instantiate Alpha-Beta in two ways: by using a left-first select rule, or by reducing the size of the stored search tree.

Another framework for Alpha-Beta and SSS* is proposed by Bhattacharya and

Bagchi [22]. This framework is called GenGame. It is based on their work on RecSSS* [21]. GenGame suffers from this background, in that it is a relatively complex framework. Also, it uses an SSS*-like OPEN list. Just like the Gsearch/Rsearch duo, instances of GenGame are created by changing certain rules, and instances vary in the amount of memory that is needed. The authors describe how the framework can be adapted to yield two variations (QuickGame and MGame) that both have reduced memory requirements, at the cost of expanding more nodes than SSS*. The QuickGame variation is not dominated by Alpha-Beta, it misses some of Alpha-Beta's deep cutoffs, although Bhattacharya reports that on average QuickGame tends to expand less nodes on artificial random trees [18]. The other variation, MGame, appears to be using the same idea as Rsearch to achieve a compromise between memory efficiency and search efficiency. On his part, Ibaraki, the inventor of Rsearch, makes no mention of the work on Staged SSS* by Campbell [32, 33] which also describes the idea of applying SSS* recursively. The idea of using stages to control memory requirements of best-first algorithms goes back to standard texts on search such as Pearl [99, p. 68] and Nilsson [93, p. 71].

The strength of both frameworks is that they isolate the differences between best-first and depth-first search in certain parts of the framework. Furthermore, they provide a model that helps in reasoning about Alpha-Beta and SSS*-like algorithms. This applies especially to Gsearch, which is a rather abstract, but clear, framework.

A drawback of Gsearch and GenGame is that they are not easily amenable to an implementation. They are both high-level frameworks that require a non-trivial amount of detail to be filled in to yield an efficient design that can be implemented in an Alpha-Beta-based game-playing program.

Compared to Gsearch and GenGame, MT is a "focused" framework. It is only concerned with the manipulation of bounds on the value of the root, and only with best-first algorithms. The details of how to traverse a tree are performed by a transposition table-enhanced Alpha-Beta procedure that acts as a black-box for the calculation of bounds. This makes MT a simpler framework that nevertheless has the power to model a complex algorithm such as SSS*. Since MT is based on Alpha-Beta, it is a very practical framework. Combined with its simplicity, MT is well-suited for experimental validation of ideas for new best-first algorithms. MTD(f) shows that this process can be successful (see section 4.2).

Gsearch, and especially GenGame, are designed to capture SSS*'s behavior. The relatively complex best-first selection strategy of SSS* is their basis. Modeling the simple left-to-right Alpha-Beta behavior is achieved by disabling some of the features of the framework. In effect, they use advanced, complex machinery to model simple behavior. They start at the complex side, and make modifications to end up at the simple side. MT, on the other hand, does not attempt to model Alpha-Beta, but uses it as a building block to model best-first algorithms. It is based on a few items of simple machinery that are used to model complex behavior. MT starts at the simple side, adds memory and a loop, and ends up explaining the complex side.

A disadvantage of MT is that it is less general than Gsearch or GenGame because it does not model the depth-first wide-window Alpha-Beta algorithm. The essence of MT is that it performs null-window Alpha-Beta calls. Of course, it is straightforward to extend MT to use wider windows, or even to become wide-window Alpha-Beta [106]. However, this complicates reasoning about its tree-traversing behavior.

3.3 Null-Window Alpha-Beta Search Algorithms

In this section two ideas for MT instances will be analyzed more deeply by looking at previous work: the bisection idea and searching for the best move.

3.3.1 Bisection

In addition to the widely used Scout family, there have been experiments with other null-window Alpha-Beta based algorithms. Coplan's C* uses bisection to find the minimax value with a relatively small number of re-searches [35, 146]. Although C* was introduced in a rather specialized context, that of a hardware implementation of a chess-endgame solver, and later for Othello endgames, C* is conceptually equivalent to MTD(bi).

Coplan proposes a pointer based structure to store visited nodes, which incurs some storage management overhead. It seems more attractive to use a hash table, as is normally used for storing transpositions by many Alpha-Beta implementations [82].

Bisecting a wide interval of interest usually gives big swings in the value of γ between successive Alpha-Beta calls. Section 4.3.1 will show that it is efficient to keep the null-window as close to the minimax value as possible. The swings should therefore be reduced as much as possible, for example by the use of an aspiration window.

In a specialized endgame search the range of values is usually much smaller than in ordinary mid-game search, so that the number of re-searches will be small. From the perspective of a reduction of the number of Alpha-Beta calls, C*'s bisection method makes sense. When C* was created it was generally assumed that search trees were too big to be stored in memory (in section 4.1 we will show that they do fit in memory). In retrospect, the reduction of the number of Alpha-Beta calls should not have been such a high priority and should not be done at the cost of using inefficient values for the Alpha-Beta calls.

3.3.2 Searching for the Best Move

Berliner has suggested that the search can be stopped as soon as a lower bound on the value of the best move is equal to an upper bound on the value of the other moves ($f^-_{best} \geq f^+_{rest}$) [13]. We will analyze the possible savings in the case the bounds are non-heuristic bounds, whose value is defined by a max or min solution tree, as is the case for MTD(best). First we will look at the best case, using the structure of the minimal tree to see how much savings can be achieved.

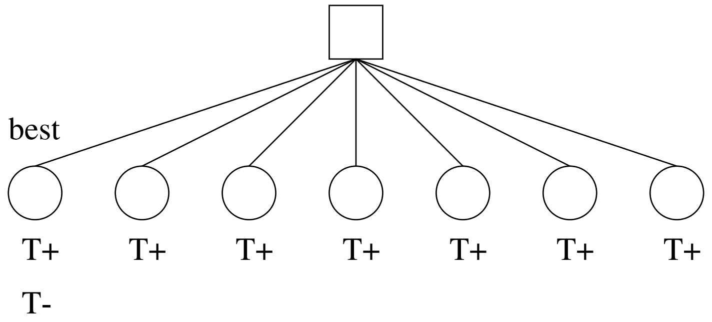

Figure 3.7: Best Move

With $f^-_{root} = f^+_{root}$ as the termination condition, a max and a min solution tree have to be constructed at the root. Figure 3.7 shows the root position and its moves. The picture shows where the max and min solution trees at these moves are created: a max solution tree (T^+) at all of the moves, and a min solution tree (T^-) at the best move.

With the new termination condition $f^-_{best} \geq f^+_{rest}$ the T^+_{best} is no longer necessary. Thus, searching for the best move saves in the best case the construction of a max solution tree below a single move. For all the other moves the max solution trees still have to be constructed. We conclude that, although there are definitely some savings, the order of the size of the search tree is unchanged, for all but the smallest branching factors.

Next, we look at the average case. Now some effort has to be expended in order to find the solution trees that make up the minimal tree of the best case. Although the search effort is bigger, the reasoning that the solution trees of the rest of the moves determine the order of the size of the search tree still holds. This analysis suggests that searching for the best move using backed-up leaf values as bounds will lead to small improvements. Tests with MTD(best) show improvements of a few percentage points [108]. It performs relatively better in "quiet" test positions, where the principal variation does not change between iterations, so MTD(best) predicts the best move correctly, and the value does not change much between iterations.

Another algorithm implementing this idea is Alpha Bounding, introduced by Schaeffer [125]. (Nagl has independently re-invented this algorithm three years later [90].) Alpha Bounding tries to be as efficient as possible by using two ideas: (a) null windows and (b) best-move searching.

The start value of the null-window searches should stay on the low side of the minimax value as much as possible, since a lower bound on the best move has to be created. Starting with the supposedly best move from a previous iteration, Alpha Bounding searches all moves at the root with a value that will make a fail high likely below the best move, so that a lower bound will be found. If this is successful, a min solution tree is constructed below the best move.

Unlike the MT algorithms, Alpha Bounding may sometimes evaluate nodes that are

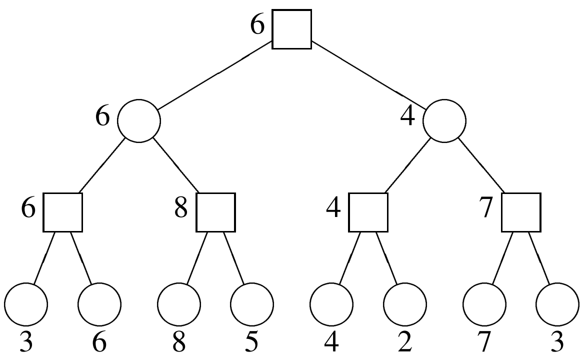

Figure 3.8: Alpha Bounding Expands 7

not visited by Alpha-Beta($n, -\infty, +\infty$). Figure 3.8 gives an example of this situation. Assuming that the start value of the search is below 3, Alpha Bounding would expand the leaf with value 7 in the first search pass (α bound = 3). In general, with a low start value, it could construct min solution trees at *each* move at the root. However, as we can see in figure 3.7, to find the best move, an algorithm has to create *max* solution trees below the inferior moves. All the min solution trees, except the single one below the best move, are overhead. Thus, a start value that is too low can result in overhead. Likewise, if the start value was chosen too high, then all moves will fail low and will be re-searched with a lower start value. Tests showed this algorithm to perform not too bad, although NegaScout was 18% better [125].

As was the case with C*, Alpha Bounding uses ideas that the MT framework also has. And as with C*, a better model of the structure of search trees would probably have resulted in an improved algorithm. One such model is the solution tree, the notion behind MT.

Background

Wim Pijls and Arie de Bruin have been investigating SSS* for some time [101, 102, 103]. In 1989 they found that there is another view on SSS*—the max-solution-tree view. They introduced a two-procedure algorithm embodying these ideas called SSS-2 [102]. In May 1993 I joined this research, trying to find my way in solution trees, Alpha-Beta, and the critical tree. After some time our discussions also turned to NegaScout and null-window search. In the spring of 1994 we found out about the link between solution trees and null-window Alpha-Beta searches, and that there are other ways of using sequences of null-windows searches, which showed up favorably in simulations [28].

The elegant idea of proving the equivalence of SSS* and MT-SSS* by inserting the list operations into the Alpha-Beta code (appendix B) was suggested by Wim Pijls, based on earlier work by him and Arie de Bruin. He created an SSS-2 like two-procedure version and helped by discussing and pointing out errors in earlier versions of the subsequent one-procedure MT code. The full MT-SSS*/SSS* equivalence proof in [106] was primarily worked out by him. Work on an alternative equivalence proof, based on static features of the search trees of SSS* and MT-SSS*, was also primarily done by him [106].

Chapter 4

Experiments

The prevailing view in the literature is that SSS* is not practical, because it uses too much memory, is too slow, and too complicated. With MT-SSS*, we oppose this view on all points. There is nothing complex about implementing MT-SSS*, it is as simple (or hard) as implementing Alpha-Beta. This enabled us to test MT-SSS* in real game-playing programs, and see whether the other points were true. In this chapter we report on these experiments. First we look at how much storage is needed for MT-based algorithms in typical applications. Next we look at their performance.

Results from this chapter have been published in [111, 113].

4.1 All About Storage

The literature portrays storage as the biggest problem with SSS*. The way it was dealt with in Stockman's original formulation [140] gave rise to two points of criticism:

1. *SSS* is slow.* Some operations on the sorted OPEN list have a time complexity that is non-polynomial in the search depth. As a results, measurements show that the purge operation of Γ case 1 consumes about 90% of SSS*'s runtime [85].

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. The example in appendix A.2 illustrates 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* [20, 21, 118]). 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 text-book implementation of a recursive data structure.

Our solution is to extend Alpha-Beta to include transposition tables (see section 2.2.2). As long as the transposition table is large enough to store at least the min or max solution trees that are essential for the efficient operation of MT-SSS* and MT-DUAL*, it provides for fast access and efficient storage. (Including 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 [115, 116, 119].) MT-SSS* will in principle 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 4.1.2 we will see how big the transposition table should be for MT-SSS* to function efficiently. SSS* stores the leaf nodes of a max solution tree. MT-SSS* stores the interior nodes too. In this way the PV can be traversed to the critical leaf, without the need for time-consuming sorting.

A potential drawback of most transposition-table implementations is that they do not handle hash-key collisions well. Collisions occur since the hash function maps more than one position to one table entry (since there are many more possible positions than entries). For the sake of speed many implementations just overwrite older entries when a collision occurs. If such a transposition table is used, then re-searches of previously expanded nodes are highly likely. Only in the case that no relevant information is lost from the table does MT-SSS* search exactly the same leaf nodes as SSS*. In section 4.1.2 we discuss why collisions are not a significant problem in practice.

The transposition table has a number of important advantages:

- It facilitates the identification of transpositions in the search space, making it possible for tree-search algorithms to search a graph efficiently. Other SSS* data structures do not readily support transpositions.

- It takes a small constant time to add an entry to the table, and effectively zero time to delete an entry. There is no need for costly purges; old entries get overwritten with new ones. While at any time entries from old (inferior) solution trees may be resident, they will be overwritten by newer entries when their space is needed. Since internal nodes are stored in the table, the Alpha-Beta procedure has no problem of finding the critical leaf; it can traverse the principal variation.

- The larger the table, the more efficient the search (because more information can be stored, the number of hask-key collisions diminishes). Unlike other storage proposals, the transposition-table size is easily adaptable to the available memory resources.

- There are no constraints on the branching factor or depth of the search tree. Implementations that use an array as an implicit data structure for the OPEN list are constrained to fixed-width, fixed-depth trees.

- Most high-performance game-playing programs already use the Alpha-Beta procedure with a transposition table. Consequently, no additional programming effort is required to implement it.

We need to make one last remark on MT-SSS*. In this algorithm (and in all the tests that we will discuss further on) one value was associated with each node, either an f^+, an f^- or an f. MT-SSS* manipulates one solution tree at a time. The pseudo code for MT stores only one bound per node. Other algorithms may manipulate two solution trees at the same time (for example, MTD(bi) and MTD(step)) and may need to store two bounds for certain nodes. In addition to this, there are some rare cases where SSS* (and MT-SSS*) unnecessarily expand some nodes. This can be prevented by storing the value of both an upper and a lower bound at the nodes, and using the value of all the children to update them. This is the difference between Stockman's SSS* and Ibaraki's SSS* (or Maxsearch) from section 3.2.1. The difference is further analyzed in [101, 106, 109].

4.1.1 Experiment Design

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 small table will still provide 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. Our data has been gathered from three programs: Chinook (checkers) [131], Keyano (Othello) [25] and Phoenix (chess) [125]. With these programs we cover the range from low to high branching factors. Using three programs we ensure a higher degree of generality and reliability of the results. All three programs are well known in their respective domains. The only change we made to the programs was to disable search extensions and forward pruning, to ensure consistent minimax values for the different algorithms. For the same reason we used the sequential versions of the programs. 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, while our version of Keyano uses static move ordering.

The Alpha-Beta code given in figure 3.2 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.

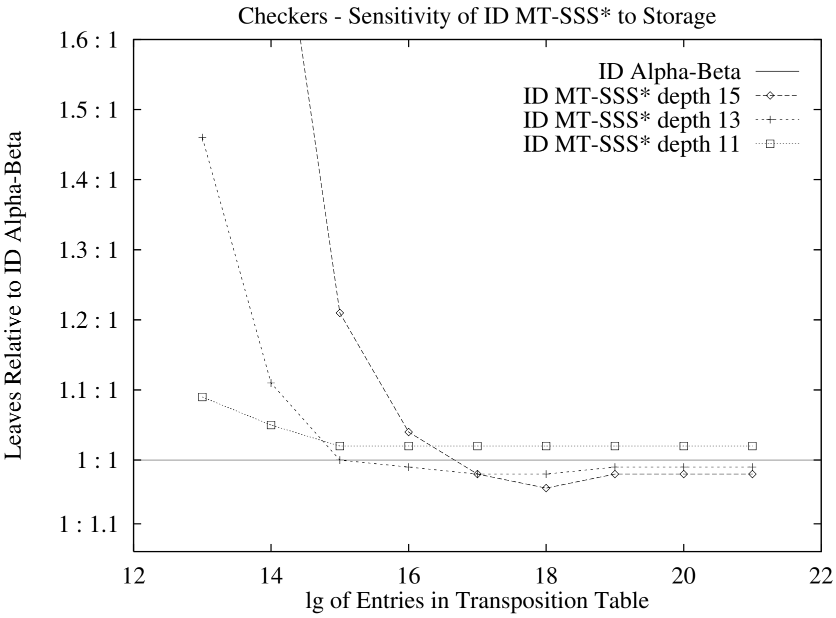

Figure 4.1: Memory Sensitivity ID MT-SSS* Checkers

Conventional test sets in the literature (such as [66]) proved to be inadequate to model real-life conditions. Positions in test sets are usually selected to test a particular characteristic or property of the game (such as tactical combinations in chess) and are not necessarily indicative of typical game conditions. For our experiments, the programs were tested using a set of 20 positions that mostly corresponded to move sequences from tournament games (see appendix C). 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.). Test runs were performed on a bigger test set and to a higher search depth to check that the 20 positions did not cause anomalies. 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 15 to 17, iterating two ply at a time. Keyano searched to 9–10 ply and Phoenix to 7–8, both one ply at a time.

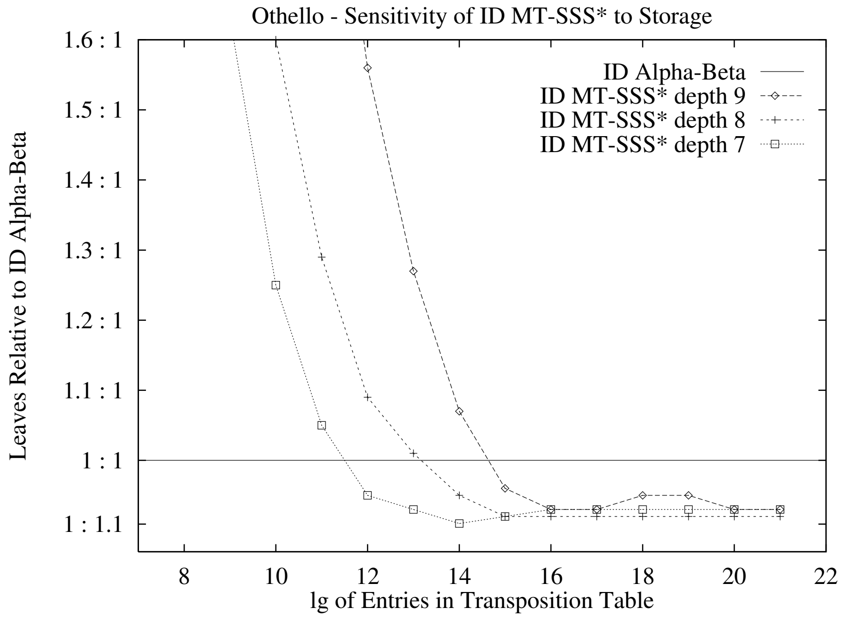

Figure 4.2: Memory Sensitivity ID MT-SSS* Othello

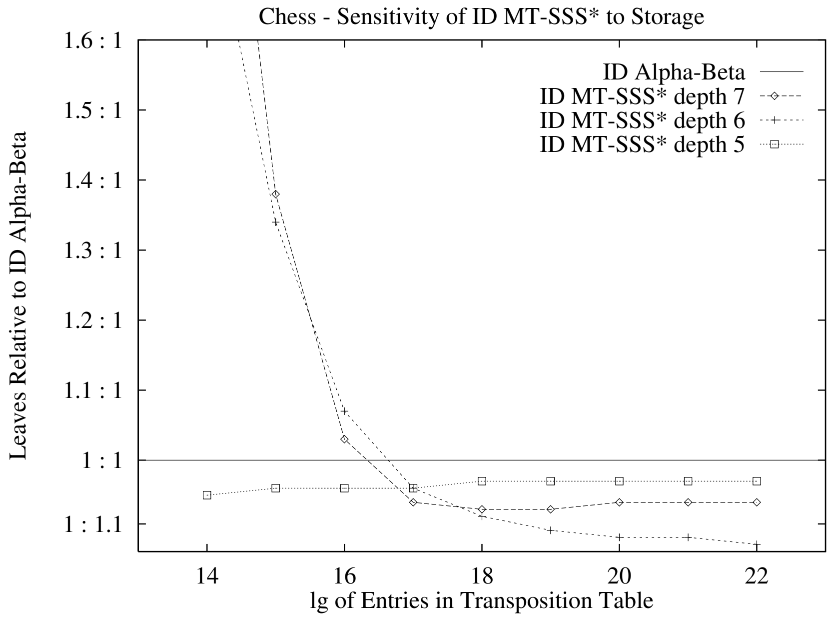

Figure 4.3: Memory Sensitivity ID MT-SSS* Chess

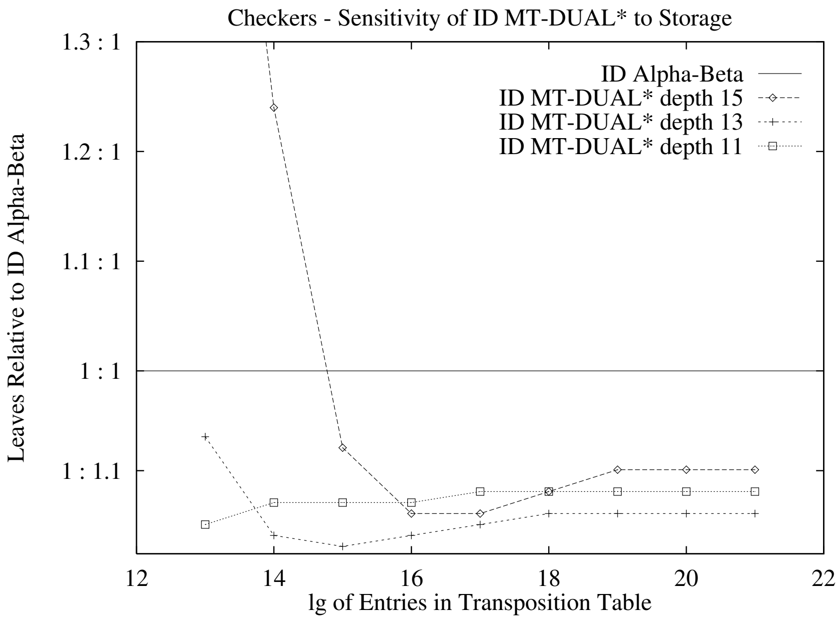

Figure 4.4: Memory Sensitivity ID MT-DUAL* Checkers

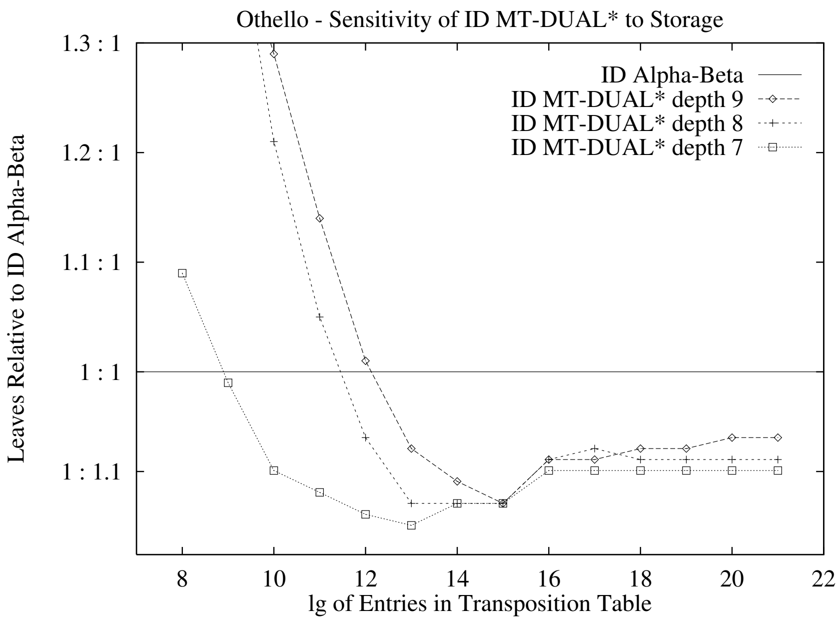

Figure 4.5: Memory Sensitivity ID MT-DUAL* Othello

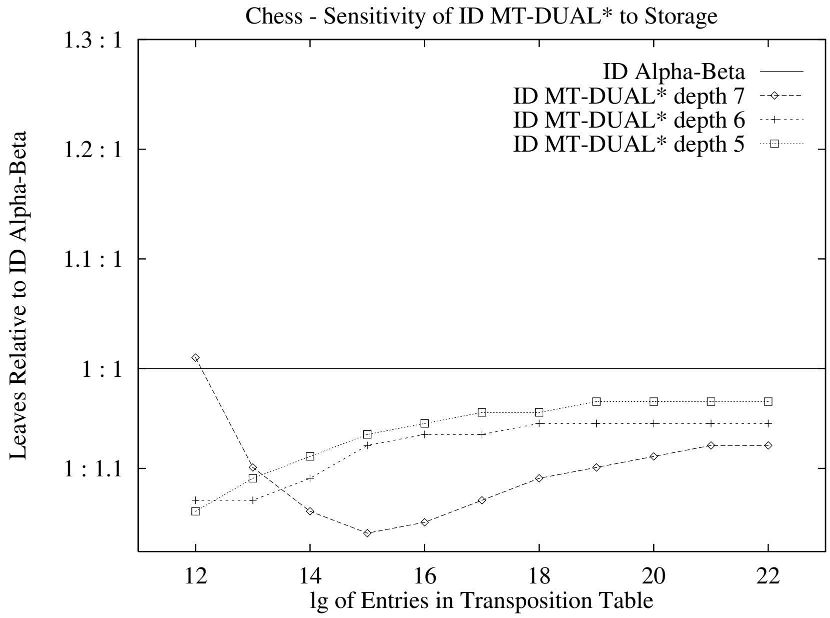

Figure 4.6: Memory Sensitivity ID MT-DUAL* Chess

4.1.2 Results

Figures 4.1–4.3 and 4.4–4.6 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, lg denotes log base 2). The graphs show that for small transposition tables, Alpha-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 MT-SSS*, except that the lines are shifted to the left.

Simple calculations and the empirical evidence lead us to disagree with authors stating that $O(w^{\lceil d/2 \rceil})$ is too much memory for practical purposes [63, 85, 88, 116, 121, 140]. 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. The graphs in figures 4.1–4.3 show that MT-SSS* needs about 2^{17} table entries for the tested search depths. Assuming that each entry is 16 bytes, a transposition table of 2 Megabyte is large enough. For deeper searches a transposition table size of 10 Megabyte will be more than adequate for MT-SSS* under tournament conditions.

The literature contains a number of proposals for reducing the memory requirements of SSS*. This is done either through having Alpha-Beta search the top or bottom part of the tree in Alpha-Beta/SSS* hybrids [78, 32, 33] or by phased or recursive versions of SSS* such as Phased SSS* [86], or Staged SSS* (or Rsearch or GenGame) [22, 32, 33, 59, 104]. Our tests show that neither hybrids nor staged/phased versions of SSS* are necessary in practice, especially because they achieve the memory reduction at the cost of a lower performance, since the best-first selection mechanism is not applied to the full tree.

There is an important difference between Alpha-Beta as found in text books and as it is used in game-playing programs. To achieve high performance, Alpha-Beta is enhanced with iterative deepening, transposition tables, and various move-ordering enhancements. These enhancements enlarge the memory requirements of the basic algorithms significantly. Here we will analyze how much memory is needed to achieve high performance. We will do this by comparing Alpha-Beta to MT-SSS* and MT-DUAL*.

The graphs provide a clear answer to the main question: MT-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.

The shape of the graphs supports the idea that there is a table size where MT-SSS* does not have to re-expand previously expanded nodes. This point lies roughly at the size of a max solution tree, which agrees with the statement that MT-SSS* needs memory to manipulate a single solution max solution tree. As soon as there is enough memory to store essentially the max solution tree, MT-SSS* does not have to re-expand nodes in each pass. The graphs also support the notion that MT-DUAL* needs less memory since it manipulates a (smaller) min solution tree.

Alpha-Beta

The lines in the graphs show the leaf count of MT-SSS* relative to that of Alpha-Beta. This poses the question as to what degree the shape is influenced by the sensitivity of Alpha-Beta to the table size. If the table is too small, then collisions will occur, causing deeper entries to be erased in favor of nodes closer to the root. Thus, the move ordering and transposition identification close to the leaves diminishes. The denominator of the lines in the graphs is not free from memory effects. To answer the question, we have to look at the absolute values. These show that Alpha-Beta has in principle the same curve as MT-SSS*, only the curve is not as steep for small table sizes. Interestingly, at roughly the same point as MT-SSS*, does Alpha-Beta's line stabilize, indicating that both need roughly the same amount of memory. The numbers indicate that Alpha-Beta needs about as much memory to achieve high performance as MT-SSS*.

To understand how much memory Alpha-Beta needs for optimal performance, we recall from section 2.2.2 that Alpha-Beta uses the transposition table for two purposes:

1. *Identification of transpositions*
   To store the nodes in a search to depth d, the current search tree must be stored. For high-performance programs this is close to the minimal tree, whose size is $O\left(w^{\lceil d/2 \rceil}\right)$.

2. *Storing best-move information*
   To store the best-move information in a search to depth d, for use in the next iteration $d+1$, the minimal tree *minus the leaf nodes* for that depth must fit in memory, or size $O\left(w^{\lceil (d-1)/2 \rceil}\right)$.

Of these two numbers the transposition information is the biggest, $O\left(w^{\lceil d/2 \rceil}\right)$. To store best-move information of the *previous* iteration only $O\left(w^{\lceil (d-2)/2 \rceil}\right)$ is needed. Of these two factors, move ordering generally has the biggest impact on the search effort. We conclude that Alpha-Beta needs between $O\left(w^{\lceil d/2 \rceil}\right)$ and $O\left(w^{\lceil (d-2)/2 \rceil}\right)$ transposition-table entries.

MT-SSS*

From section 2.2.2 we recall that MT-SSS* benefits from the transposition table in a third way: prevention of re-expansion of nodes searched in previous passes. The graphs show that this last aspect has the biggest impact on MT-SSS*'s memory sensitivity. For small table sizes collisions cause nodes near the leaves of the tree to be overwritten constantly. We could ask ourselves whether collisions remain more of a problem for MT-SSS* than for Alpha-Beta when more memory is available.

If the transposition table never loses any information except nodes outside the max solution tree plus its direct descendants, then MT-SSS* builds exactly the same search tree as SSS*. Conventional transposition tables, however, are implemented as hash tables that resolve collisions by over-writing entries. Usually, entries further away from the root are not allowed to overwrite entries closer to the root, since these entries are thought to prevent the search of more nodes. In the case of MT-SSS* some of these nodes could be useless—not belonging to the max solution tree—while some nodes that were searched to a shallow depth could be part of the principal variation, and are thus needed for the next pass.

When information is lost, how does this affect MT-SSS*'s performance? From our experiments with "imperfect" transposition tables we conclude that MT-SSS*'s performance does not appear to be negatively affected. Inspection of the MT-SSS* test results shows that after a certain critical table size is reached, the lines stay relatively flat, just as in figure 4.1–4.3. If collisions were having a significant impact, then we would expect a downward slope, since in a bigger table the number of collisions would drop. (Maybe this happens close to the point where the lines become horizontal, implying that choosing a slightly bigger size for the table removes the need for additional collision resolution mechanisms.) We conclude that in practice the absence of elaborate collision resolution mechanisms in transposition tables, such as chaining or rehashing, is not an issue where MT-SSS* is concerned.

How much memory does MT-SSS* need for the third function: prevention of re-searches of previous passes? In section 3.1 we noted that MT-SSS* manipulates a max solution tree. The size of this tree is $O\left(w^{\lceil d/2 \rceil}\right)$. However, the benefit of storing leaf nodes is small. The best-move information of their parents causes just one call to the evaluation function, which will cause a cutoff. Most benefits from the transposition table are already achieved if it is of size $O\big(w^{\lceil (d-1)/2 \rceil}\big)$. So, for high performance in MT-SSS* we need $O\big(w^{\lceil d/2 \rceil}\big)$ for the transpositions, $O\big(w^{\lceil (d-1)/2 \rceil}\big)$ for the multiple passes, and $O\big(w^{\lceil (d-2)/2 \rceil}\big)$ for the best moves of the previous iteration.

Assuming that the impact of transpositions is less than that of move ordering and multiple-pass re-searches, it seems that MT-SSS* needs a bit more memory than Alpha-Beta, for high performance. However, MT-SSS* uses null-window Alpha-Beta calls, that generate more cutoffs than the standard wide-window Alpha-Beta algorithm. The null windows cause MT-SSS* to have, in effect, a smaller w. Since both algorithms stabilize at roughly the same table size, it appears that these effects compensate each other.

MT-DUAL*

The preceding reasoning is supported by the graphs for MT-DUAL*. Here we see an interesting phenomenon: compared to Alpha-Beta first the node count drops sharply, and then increases again slightly with growing table sizes. It appears there is a point where the best-first, "memory hungry" algorithm performs better when given *less* memory. Again, the explanation is that the lines in the graph show MT-DUAL* *in relation to* Alpha-Beta. Inspection of the MT-DUAL* test results reveals that the effect is caused by the fact that the MT-DUAL* curve stabilizes earlier than the Alpha-Beta curve (which stabilizes at roughly the same point as MT-SSS*). So, the increase at the end of the graph is not caused by MT-DUAL*, but by Alpha-Beta.

The reason that MT-DUAL* needs less memory than MT-SSS* is that it manipulates min solution trees, which are of size $O\big(w^{\lfloor d/2 \rfloor}\big)$. For high performance MT-DUAL* needs $O\big(w^{\lfloor d/2 \rfloor}\big)$ for the transpositions, $O\big(w^{\lfloor (d-1)/2 \rfloor}\big)$ for the multiple passes, and $O\big(w^{\lceil (d-2)/2 \rceil}\big)$—*not* smaller—for the best-moves of the previous iteration (the size of the minimal tree is the same for MT-DUAL*). Since most of these figures are smaller than for Alpha-Beta and MT-SSS*, this analysis provides an explanation why MT-DUAL* could perform better with less memory. Since we are using iterative deepening versions of the algorithms, this advantage can also shine through in *even* search depths, where the floor or ceiling operators do not make a difference.

By examining the trees that are stored in the transposition table by iterative deepening versions of Alpha-Beta and MT-SSS*, and approximating the amount of memory that is needed, we were able to find an explanation why both algorithms need roughly the same amount of memory to achieve high performance. More research can provide further insight in this matter. We conclude from the experiments that MT-SSS* and MT-DUAL* are practical alternatives to Alpha-Beta, as far as the transposition-table size is concerned.

4.1.3 MT-SSS* is a Practical Algorithm

Section 4.1 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, were addressed in the RecSSS* algorithm [21, 118]. 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 disadvantage 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 the Alpha-Beta procedure and transposition tables, the effort to implement MT-SSS* consists only of adding a simple driver routine (figure 3.1). 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* and MT-DUAL*, interior nodes are accessed by fast hash-table lookups, to eliminate the slow 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 can be seen in figure 4.1–4.3 and 4.4–4.6, indicating 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 traversal by MT-SSS* and MT-DUAL* can have a noticeable effect. For real applications, in addition to leaf node count, the total node count should also be checked (see section 4.2).

We conclude that SSS* and DUAL* have become practical, understandable, algorithms, when expressed in the new formulation.

4.2 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; their results were inferior to MTD(f). An in-depth treatment of the other algorithms is deemed more interesting.

4.2.1 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. Tests like the ones in section 4.1 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. 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.

Many papers in the literature use Alpha-Beta as the base-line for comparing the performance of other algorithms (for example, [33, 82]). 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, Keyano, and Phoenix), it seems more reasonable to use the enhanced programs 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 algorithms—notably SSS*—turn out to be inferior. To achieve high performance, the programs use the following enhancements: all three programs use iterative deepening and a transposition table, Chinook and Phoenix use the history heuristic and quiescence search, Chinook uses the ETC (see chapter 5), and Phoenix and Keyano use static move ordering.

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 [119].

4.2.2 Results

Figures 4.7–4.9 show the performance of Chinook, Keyano and Phoenix, respectively, using the number of leaf evaluations as the performance metric. Figures 4.10–4.12 show 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 geometric mean of the cumulative number of nodes over all previous iterations for a certain depth normalized to Aspiration NegaScout (which is realistic since iterative deepening is used).

The lines in the graphs are generally quite jagged. Many lines show an odd/even oscillation depending on the search depth, although the effect is not consistent. Also it appears that generally DUAL* builds slightly smaller trees than SSS*, although here too there are exceptions. These effects can to some extent be explained out of the basic asymmetry of max and min solution trees: in a uniform max-rooted minimax tree of odd depth, a min solution tree has less leaf nodes than a max solution tree. The fact that the lines appear to fluctuate wildly is partly due to the fine scale of the graphs and

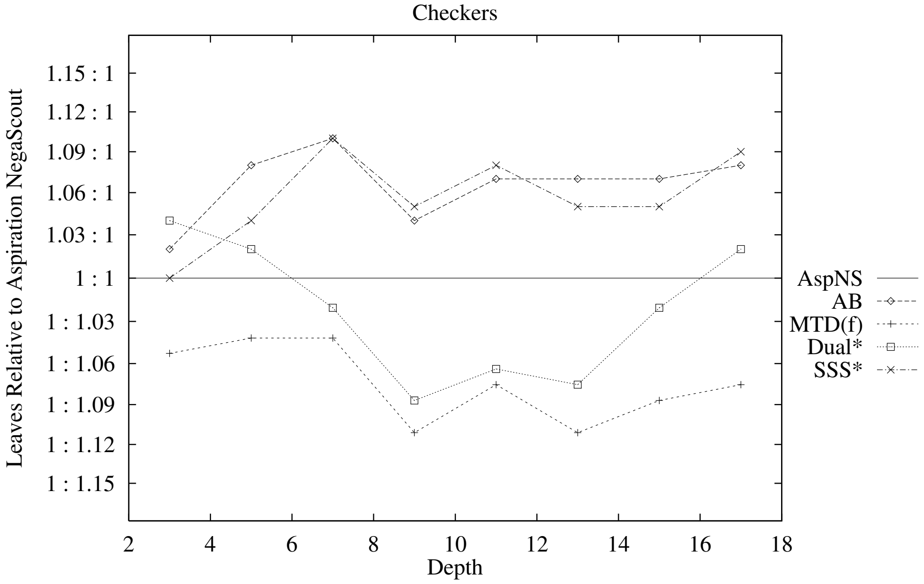

Figure 4.7: Leaf Node Count Checkers

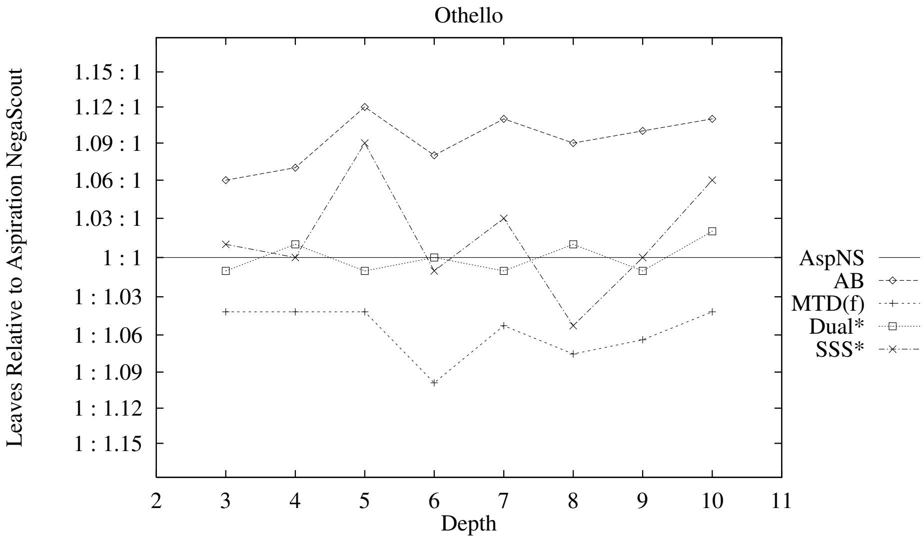

Figure 4.8: Leaf Node Count Othello

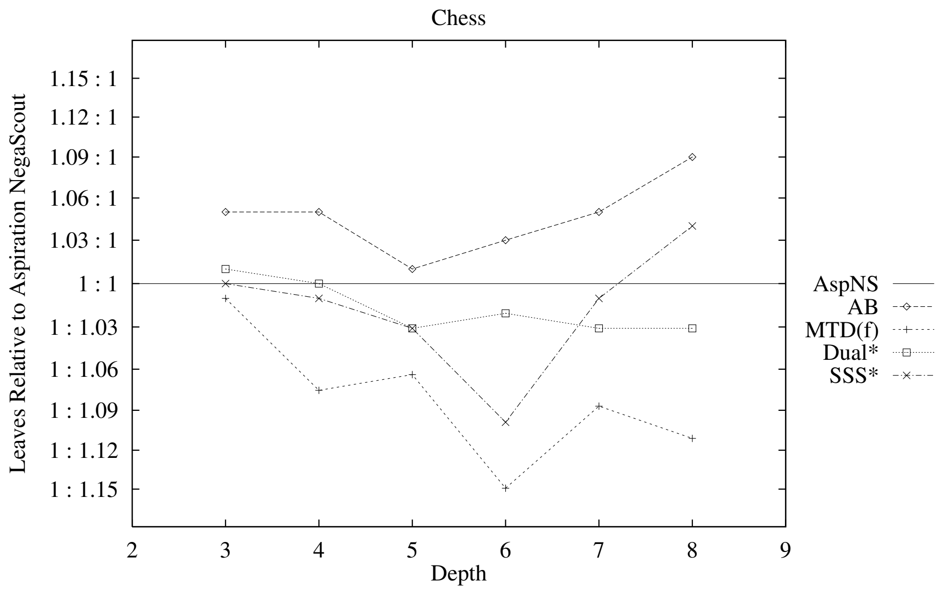

Figure 4.9: Leaf Node Count Chess

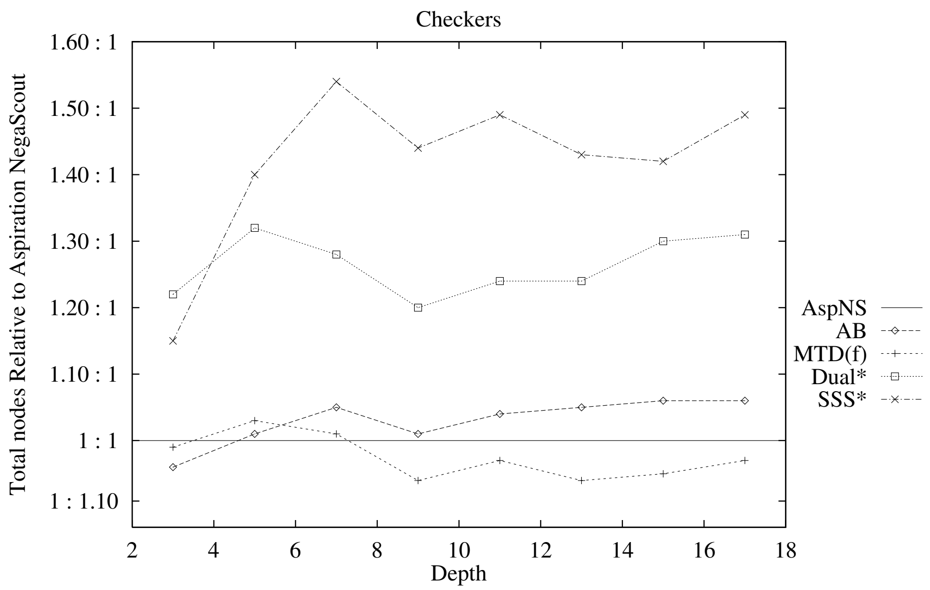

Figure 4.10: Total Node Count Checkers

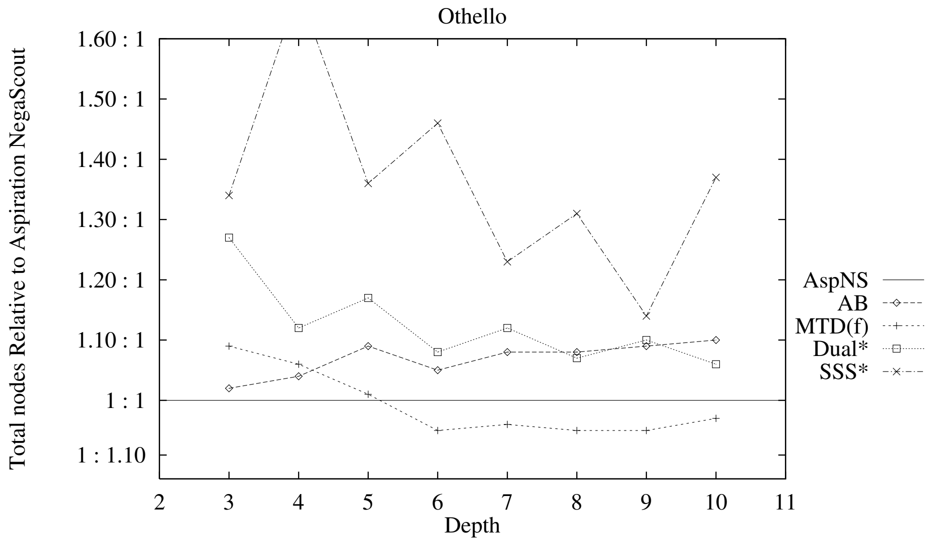

Figure 4.11: Total Node Count Othello

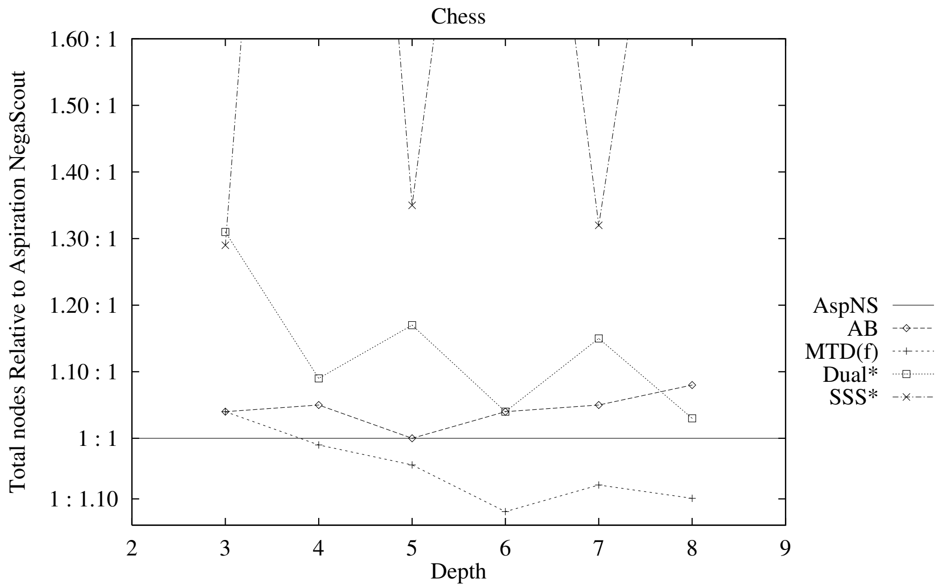

Figure 4.12: Total Node Count Chess

the irregularity of minimax trees that are generated in actual applications.

SSS* and DUAL*

Contrary to many simulations, our results show that the difference in the number of leaves expanded by SSS* and Alpha-Beta is relatively small. Since game-playing programs use many search enhancements 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. In practice, SSS* is a small improvement on Alpha-Beta (depending to some extent 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 section 2.3.2 on page 33) has disappeared. Reasons for this will be discussed further in section 4.4.

Odd/Even Effect in MT-SSS* and MT-DUAL*

Looking at the graphs for total nodes (figures 4.10–4.12), 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 $g \leq \gamma$ is easily satisfied when γ is close to $+\infty$). Especially since the move ordering near the leaves gets worse, 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 and interior node count (not shown separately). 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 here for improvement, 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.)

Dominance Under Dynamic Move Reordering

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* [140]. 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 other dynamic move ordering schemes 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*

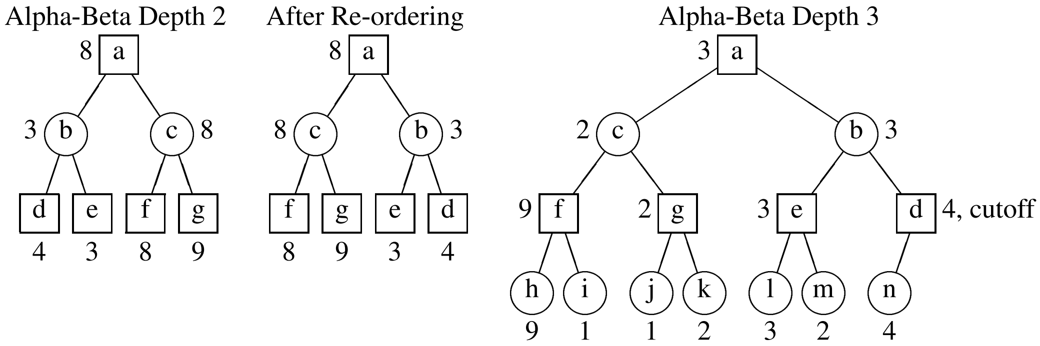

Figure 4.13: Iterative Deepening Alpha-Beta

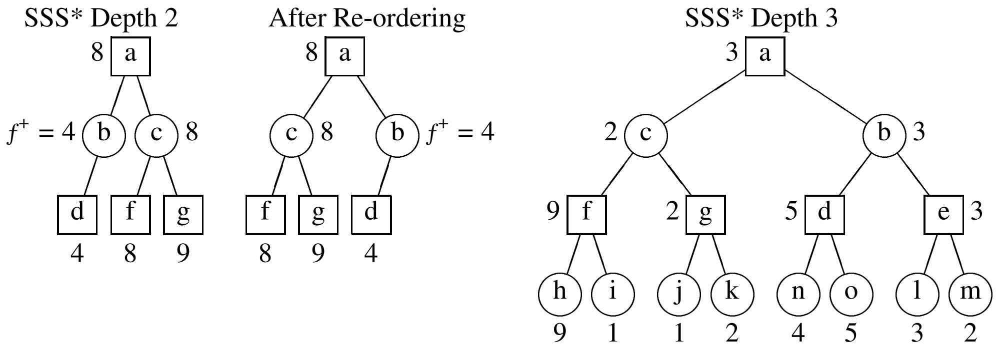

Figure 4.14: Iterative Deepening SSS*

be considered in the same order (Coplan makes this assumption explicit in his proof of C*'s dominance over Alpha-Beta [35]).

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 is best. 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 implied 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 certain 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 4.13 and 4.14 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.

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.

We conclude that an advantage of SSS*, its domination over Alpha-Beta (point 4 in section 2.3.2), is wrong in practice.

Aspiration NegaScout and MTD(f)

The results show that Aspiration NegaScout is better than Alpha-Beta. This is consistent with [128] 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 to 10% improvement, depending on the game. More surprising is that MTD(f) out-performs Aspiration NegaScout on the total node measure as well. This suggests that MTD(f) is calling MT close to the minimum number of times (which is 2—one for the upper bound, one for the lower bound). 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.

4.2.3 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) (figures 4.15–4.17). We only show the deeper searches, since the relatively fast shallower searches hamper accurate timings. The runs shown are typical example runs on a Sun SPARC. We did experience different timings when running on different

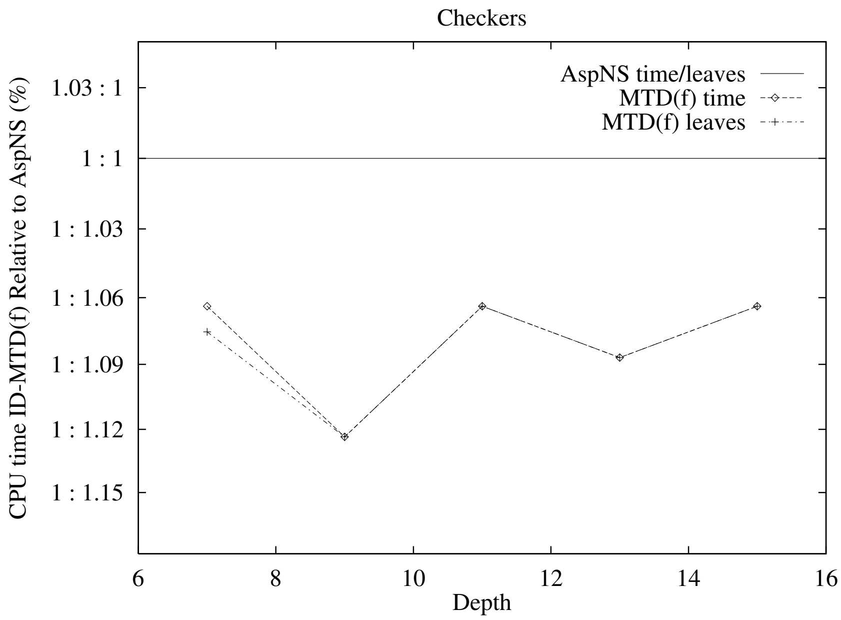

Figure 4.15: Execution Time Checkers

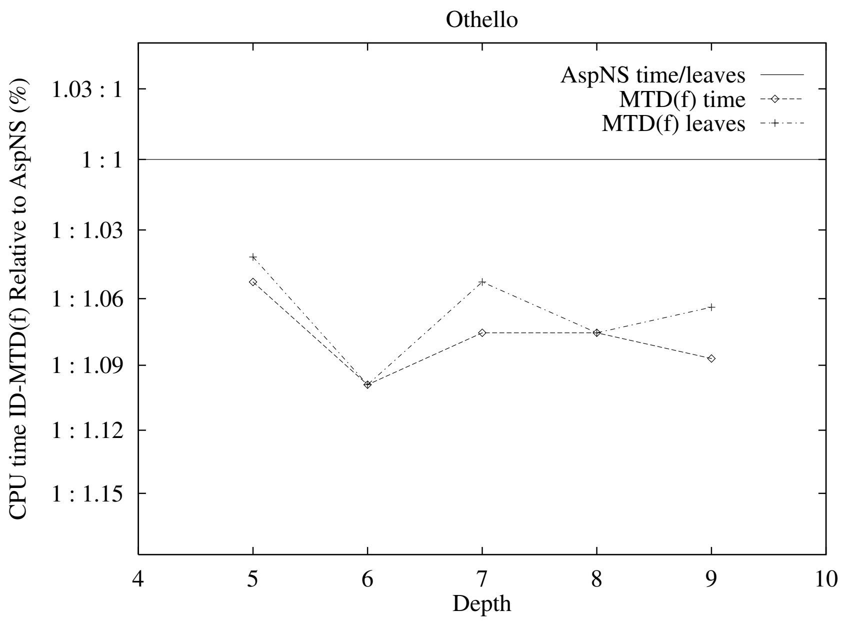

Figure 4.16: Execution Time Othello

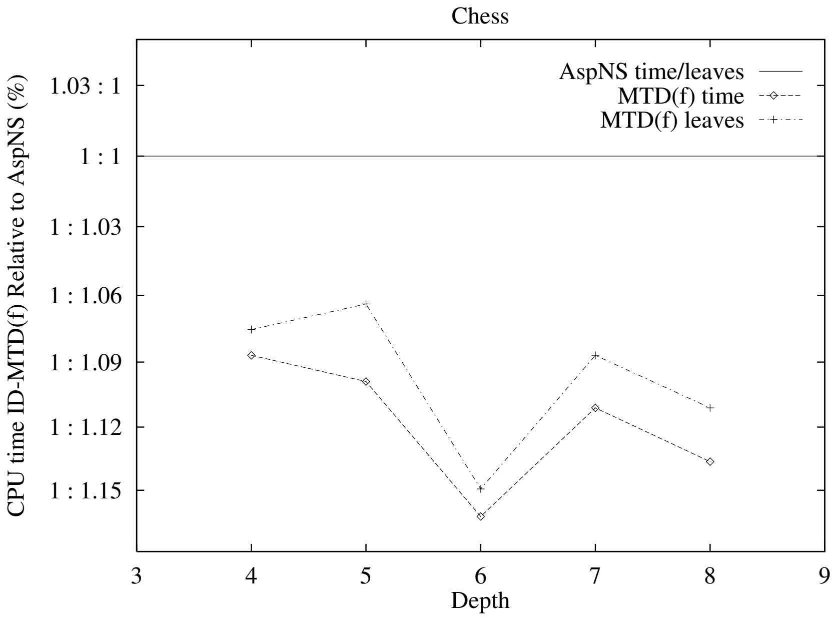

Figure 4.17: Execution Time Chess

machines. It may well be that cache size plays an important role, and that tuning the program has an impact as well.

The experiments showed that for Chinook and Keyano, MTD(f) was about 5% faster in execution time than Aspiration NegaScout; for Phoenix we found MTD(f) 9 to 16% faster. (Application dependent tuning of MTD(f) can improve this a few percentage points, see section 4.3.1.) For other programs and other machines these results will obviously differ, depending in part on the quality of 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 into, or guide and verify hypotheses about these complexities, as done, for example, in chapter 5.

4.3 Null-Windows and Performance

This section looks at some relations between the value of bounds and the search effort that must be expended to compute them with a null-window Alpha-Beta search.

4.3.1 Start Value and Search Effort

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 convergence in less steps, 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, this is not the case. Generally an MT call with a loose bound, like $+\infty$, is cheaper than an MT call with a tight bound, like $f + 4$. For a loose bound the left-first solution tree suffices. For tighter bounds it takes more work to get a cutoff, and hence the work to find the solution tree for the bound is greater. Also, in well-ordered seach spaces, the construction of the first solution tree is by far the most expensive. Refining it to yield a slightly sharper bound costs only a few node expansions. Furthermore, max solution trees contain $w^{\lceil d/2 \rceil}$ leaf nodes, while min solution trees contain $w^{\lfloor d/2 \rfloor}$ leaf nodes (if trees are of uniform width and depth).

Schaeffer has looked at the size of the search tree of an isolated null-window call, for different values [85, 125]. The results pointed out that not all null-windows are equal. Using Phoenix on the 24 Bratko-Kopec positions [66], a search with a null-window on the low side of f was significantly cheaper than a search on the high side, for both odd and even search depths. At f the search effort jumped to a higher level. This sharp increase in search effort has been called the *minimax wall*. In a program, such as Phoenix, that searches well-ordered trees, a fail high at the root (f^-, T^-) will occur before all children have been searched. An Alpha-Beta call resulting in a fail low (f^+, T^+) will have expanded all children at the root.

Thus, MT calls generally do *not* expand the same number of nodes. Since we could not find an analytical solution we have conducted experiments to test the intuitively appealing idea that starting a search close to f is cheaper than starting far away.

Figures 4.18–4.20 validate the choice of a starting parameter close to the game value. The figures show the efficiency of an iterative deepening search as a function of the distance of the first guess from the correct minimax value for each search depth. 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 is instructive to compare these figures with figure 4.21, which is based on simulated trees—no iterative deepening or transposition tables there. Now the curves look reassuringly smooth. The graph is taken from [28]. Each line in the graph is the average of 20 artificial trees of width 5 and depth 9. The top line shows unordered trees, where the first move has a $\frac{1}{w}$ = 20% probability of being best. The lower lines are progressively better ordered. The bottom line represents a perfectly-ordered tree. Here all algorithms search the minimal tree, independent of the start value of the search. Figure 4.21 shows that the closer a search starts to the minimax value of a game tree, the less nodes are expanded, on average. The gain in performance is less in ordered trees.

The graphs in figures 4.18–4.20 show that the smaller the distortion, the smaller the

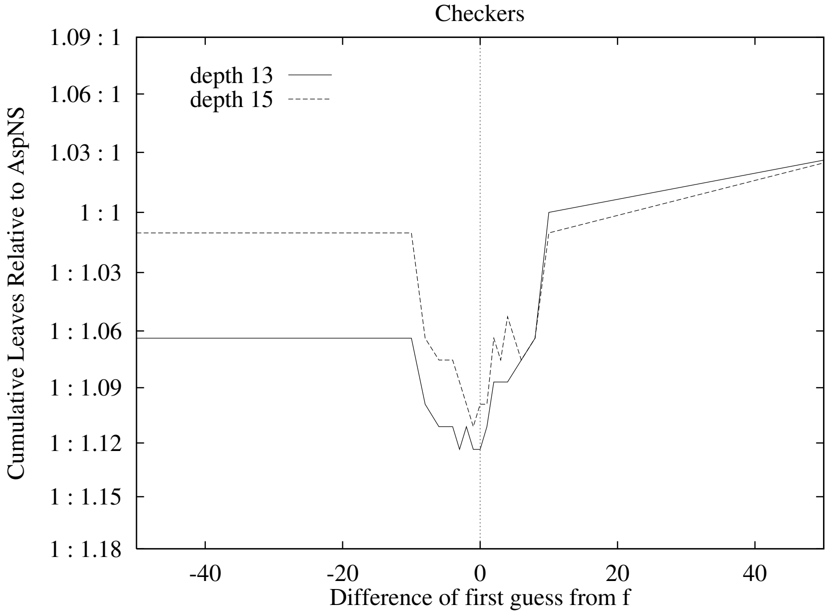

Figure 4.18: Tree Size Relative to the First Guess f in Checkers

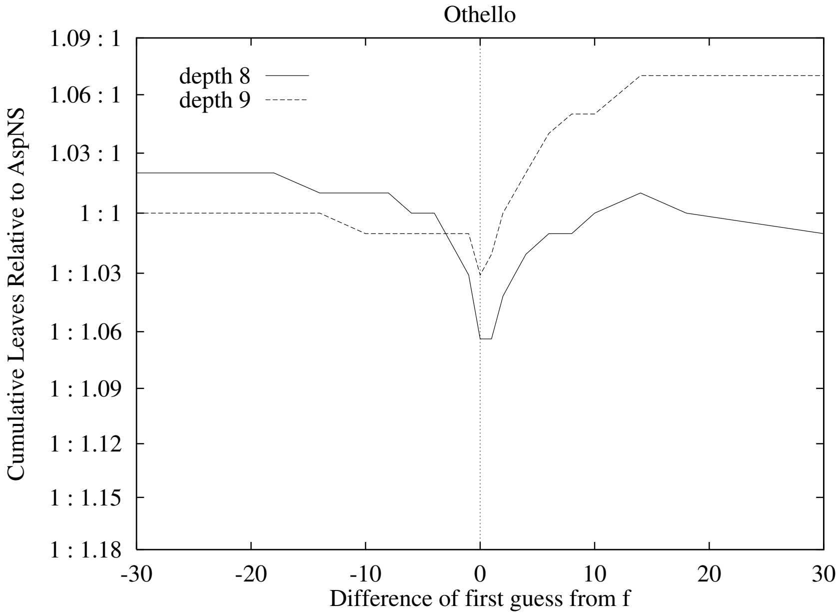

Figure 4.19: Tree Size Relative to the First Guess f in Othello

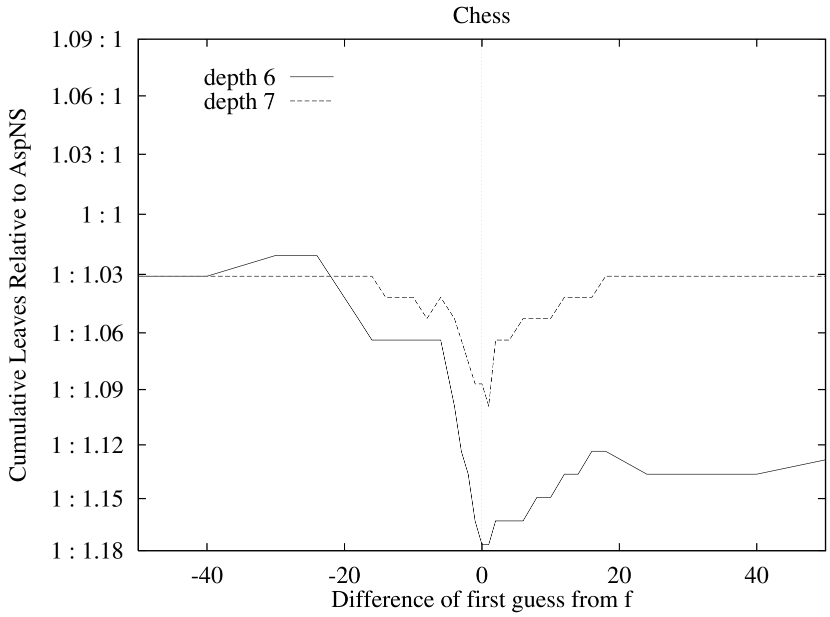

Figure 4.20: Tree Size Relative to the First Guess f in Chess

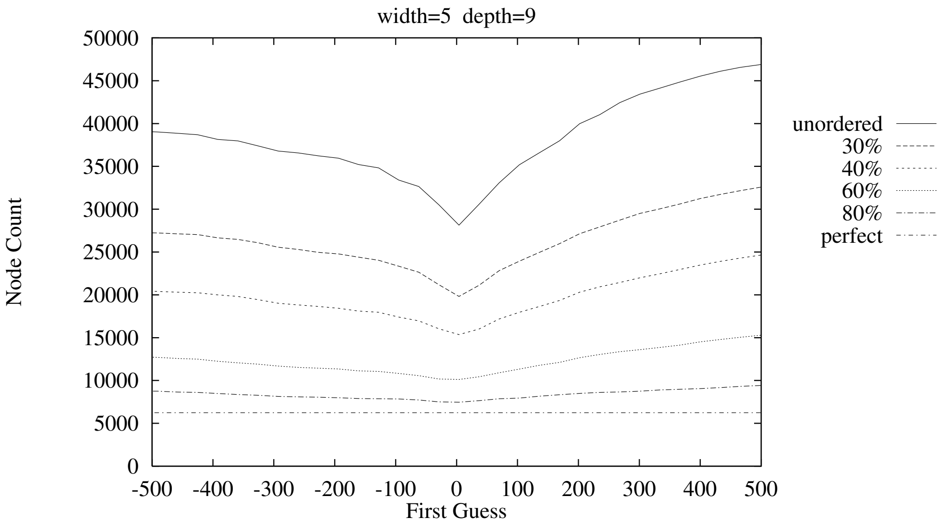

Figure 4.21: Effect of First Guess in Simulated Trees

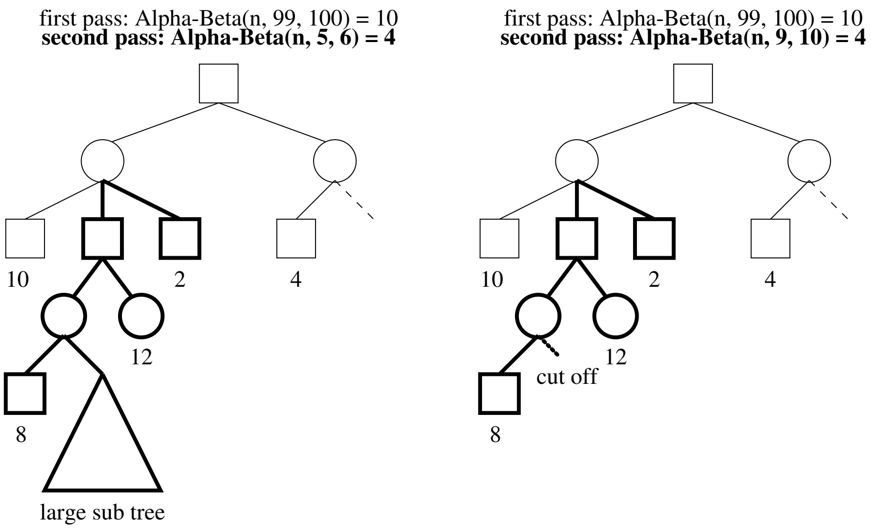

Figure 4.22: Two Counter Intuitive Sequences of MT Calls

search tree is. Our intuition that starting close to the minimax value is a good idea is justified by these experiments. A first guess close to f makes MTD(f) perform better than the 100% Aspiration NegaScout baseline. We also see that the guess must be quite close to f for the effect to become significant. Thus, if MTD(f) is to be effective, the f obtained from the previous iteration must be a good indicator of the next iteration's value. For programs with a pronounced odd/even oscillation in their score, results are better if the score from two iterations ago is used. Comparing the graphs in figures 4.7–4.9 and 4.18–4.20, 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 a form of application-dependent fine tuning of the MTD(f) algorithm.

When a *single* null-window Alpha-Beta search for a bound is performed, a search for a loose bound generally takes less effort than a search for a tight bound. The figures 4.18–4.21, however, show the effort of *a number* of null-window searches, of all the searches that are needed to prove the minimax value.

In doing these experiments, the diversity of real-life game trees became apparent. It is not hard to construct a counter-example where a bad null-window expands *less* nodes than a good first guess. For example, figure 4.22 shows that Alpha-Beta($root$, 99, 100) followed by the re-search Alpha-Beta($root$, 9, 10) (in bold) skips the large sub-tree, whereas the call Alpha-Beta($root$, 99, 100) followed by Alpha-Beta($root$, 5, 6) (in bold) expands it, where the β of 6 is closer to $2 \leq f \leq 4$ than the β of 10. In the tests we also encountered some positions where Aspiration NegaScout performed better than MTD(f).

	best-first	**not** best-first
good start value	MTD(f)	NegaScout
not good start value	SSS*	Alpha-Beta

Figure 4.23: Four Algorithms, Two Factors

4.3.2 Start Value and Best-First

One could ask the question how it is possible at all for a depth-first algorithm like NegaScout to out-perform a best-first algorithm like SSS*. The answer is given by the influence of the start value of an MT sequence. NegaScout derives its start value for later recursive null-window calls from the tree it is searching. If that tree has become relatively well-ordered through the use of enhancements, such as iterative deepening and the history heuristic, then the start value will become a reasonable guess. SSS* does not use this idea; instead it uses best-first node selection based on the information from a previous pass. In our tests, with programs using many search enhancements, a good start value seems to be a bit more effective. Best-first schemes tend to have an advantage on trees with a lower quality of move ordering and a wider branching factor. The table in figure 4.23 shows the two ideas:

1. *Best-First Selection*
   The best-first node selection scheme that SSS* and other MT instances use, is essentially based on a traversal of solution trees from previous search passes to descend the principal variation to the critical leaf, and then expand an open brother of this leaf—the *best* node to expand in the SSS* sense (see figure 2.18).

2. *A Good Start Value*
   Section 4.3.1 showed that generally a start value closer to the final minimax value yields a more efficient search for an MT sequence.

The table in figure 4.23 suggests that MTD(f) combines the good parts of two existing algorithms: best-first expansion from SSS* and a good start value from NegaScout.

NegaScout and MTD(f) are the two algorithms that perform best in the experiments. One is categorized as depth-first, the other as best-first. Otherwise, they have much in common. By looking at their behavior in very small memory situations, we will be able to see some similarities and differences more clearly, since the best-first behavior of MTD(f) depends on the information of previous passes to guide it through the tree.

From section 4.1.2 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 previously expanded solution tree. Since MTD(f) performs significantly fewer calls, re-expansions due to insufficient storage are not as big a problem.

This point is illustrated in figures 4.24–4.28. In these graphs the size of the transposition table has been reduced gradually to as low as $2^6 = 64$ entries. The graphs show

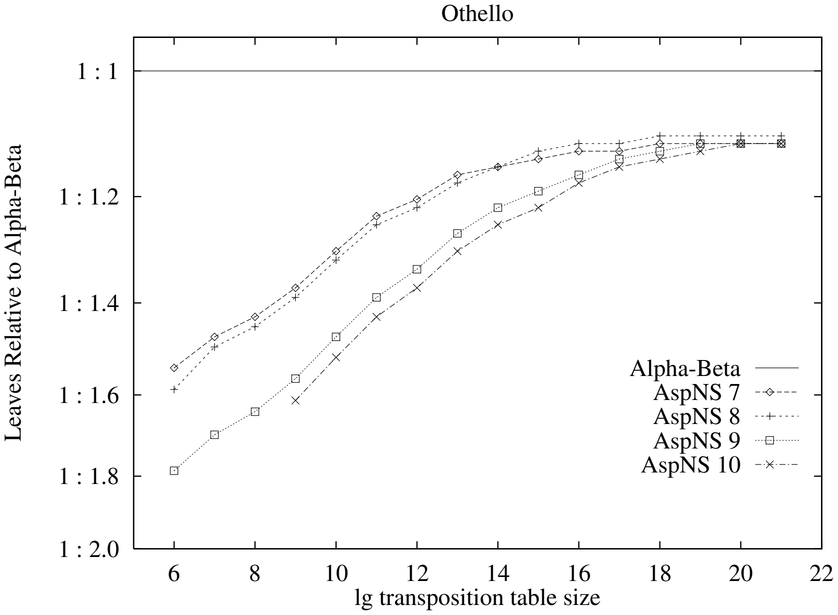

Figure 4.24: Aspiration NegaScout in Small Memory in Othello

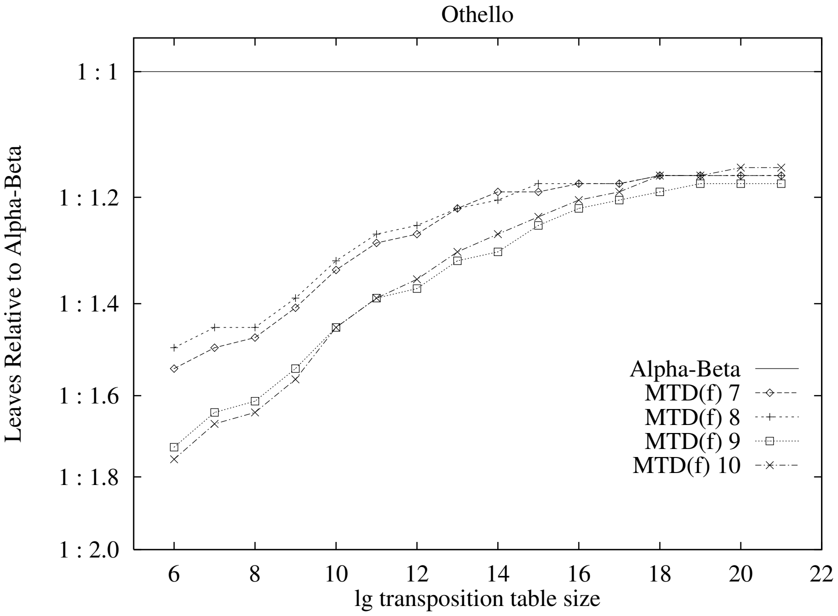

Figure 4.25: MTD(f) in Small Memory in Othello

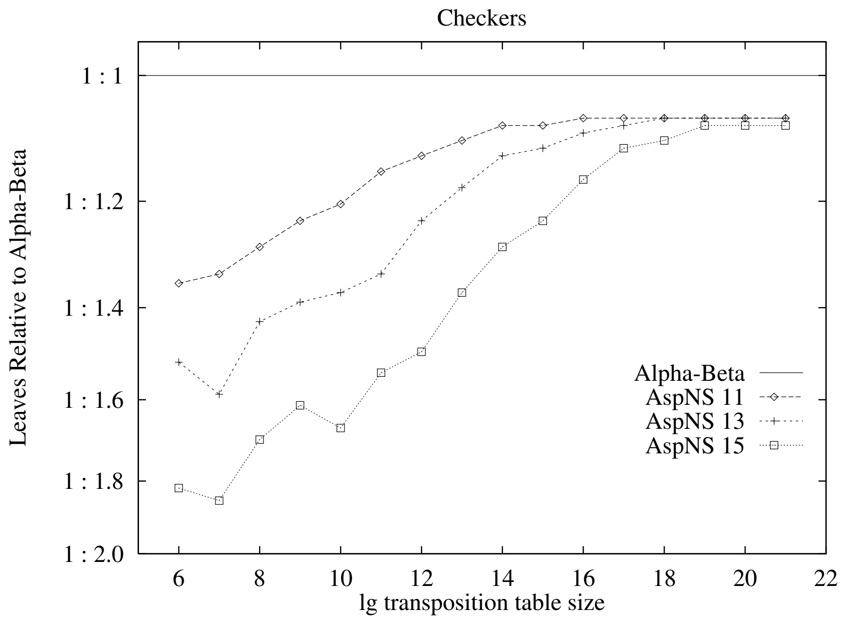

Figure 4.26: Aspiration NegaScout in Small Memory in Checkers

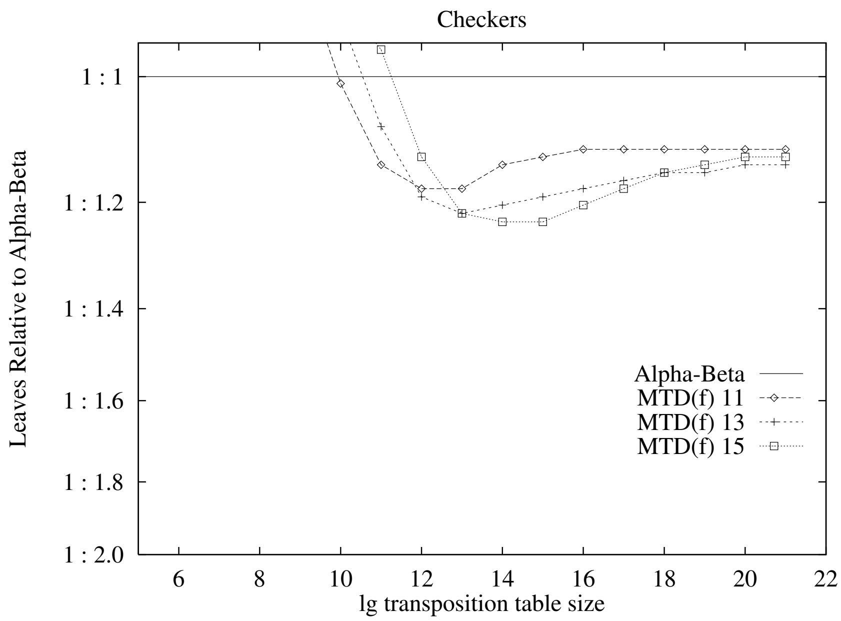

Figure 4.27: MTD(f) in Small Memory in Checkers

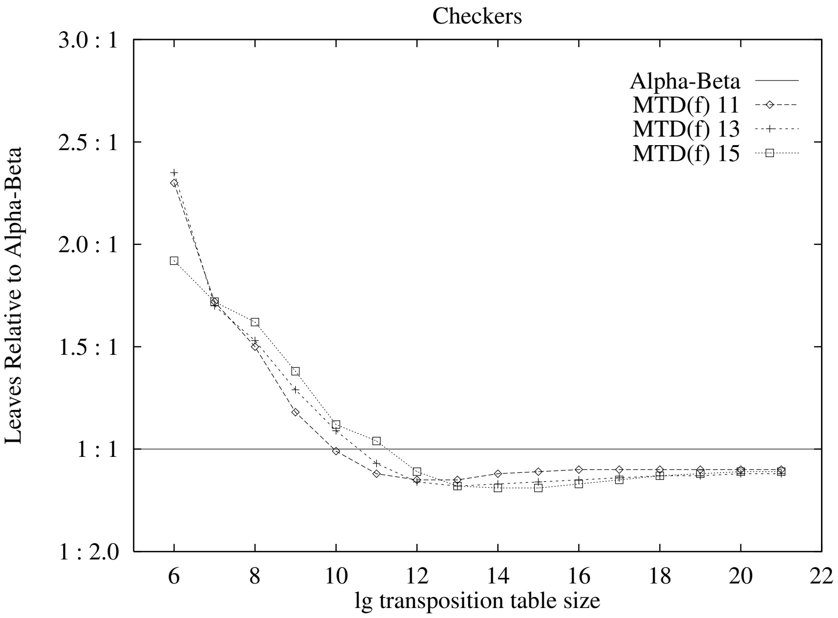

Figure 4.28: MTD(f) in Small Memory (Chinook) II

test results (leaf count) for Othello (Keyano) and checkers (Chinook), with iterative deepening versions of Alpha-Beta, NegaScout and MTD(f). For Keyano depth 7–10 is shown, for Chinook depths 11, 13 and 15. Note for Keyano the odd/even effect: for small memory the results for the depths are paired. For clarity, the Chinook result for MTD(f) is shown again in figure 4.28 with a different scale on the y axis. (The size of the transposition table in Phoenix cannot be reduced easily beyond 2^{12}. In the range from 2^{12}–2^{21} the results resemble Keyano's graphs.)

Aspiration NegaScout

We see that Aspiration NegaScout out-performs Alpha-Beta in small memory by a wide margin, like the literature on NegaScout predicted [85, 99, 116]. As the transposition table gets smaller, the margin grows wider. Apparently, the fact that the move ordering information gradually disappears, hurts Alpha-Beta more than NegaScout. NegaScout still benefits from the tight bounds of null-window calls, where Alpha-Beta's search window converges more slowly. These graphs support the reasoning in section 4.1.2 that for high performance Alpha-Beta has in fact exponential memory requirements.

MTD(f)

For MTD(f) the figures need more explanation. For Keyano, MTD(f)'s sensitivity to memory is comparable to that of Aspiration NegaScout. The graph for Chinook is different. As long as the transposition table has more than 2^{12} = 4096 entries, a smaller transposition table hurts Alpha-Beta more than MTD(f). As the table gets extremely small, MTD(f)'s relative performance deteriorates rapidly, although it remains much better than for MT-SSS* (figure 4.1). The different memory sensitivity of MTD(f) in Chinook and Keyano is caused by a different range of the evaluation function. Chinook's evaluation function range is ± 9000, Keyano's range is ± 64. The small range for Keyano causes the value of the previous iteration to be quite a good estimate. MTD(f) rarely calls MT more than 3 times. The wider range in Chinook causes MTD(f) to call MT more often, in some positions 2 or 3 times, in some other 12 to 20 times. The absence of enough memory makes these MT re-searches much more expensive, causing the number of leaf evaluations to rise to 2–2.5 times that of Alpha-Beta, for a table with 64 entries.

This points to a difference between NegaScout and MTD(f). NegaScout is fully recursive, in contrast to MTD(f), which restarts the search a few times at the root. By starting the re-searches at the root, MTD(f) takes the risk of having to re-expand bigger trees than NegaScout, the impact of which is normally reduced by the presence of memory to the extent that NegaScout is out-performed. As a consequence, the absence of memory removes the best-first nature of MTD(f)'s node selection scheme. With limited memory, there is not enough information in the transposition table to guide MT towards the best node to select next. All the previous search information has to be re-expanded in each pass. The tests show that for depth 15 at least 2^{12} transposition-table entries should be present for Chinook, because of its wide evaluation-function range (assuming 16 byte entries, this amounts to 64 kilobyte of memory). If there is almost no memory at all, then NegaScout performs better.

It is possible to improve the low-memory behavior of MT instances by changing the replacement strategy of the transposition table. Currently nodes close to the root are favored. For MT a preference for the last traversed solution tree would be better. However, the test results indicate that there still is a wide margin before tournament game-playing programs would benefit from these changes. The problems of MTD(f) only occur with extremely small amounts of memory. Otherwise MTD(f) out-performs all tested algorithms in all three games.

Figure 4.29 shows a performance picture of algorithms as they are used in a practical setting, with a transposition table, iterative deepening, and other move ordering enhancements, in contrast to figures 2.10, 2.14 and 2.19, that showed the un-enhanced, standard text-book versions of the algorithms. All these enhancements reduce the number of nodes considered before finding the cutoff to the extent that all algorithms perform almost the same. The enhancements also determine the memory needs for high performance. The picture shows that more memory improves the effectiveness of the enhancements. Using the same amount of storage as Alpha-Beta and MT-SSS*, NegaScout and MTD(f) generally perform better than the other two. The null-windows cause more cutoffs to occur than with Alpha-Beta($n, -\infty, +\infty$). For the highest performance they need $O(w^{d/2})$ as well, although their performance is less sensitive to smaller transposition tables than Alpha-Beta and MT-SSS*. Their good start value for the null-window Alpha-Beta searches makes them perform less re-searches than

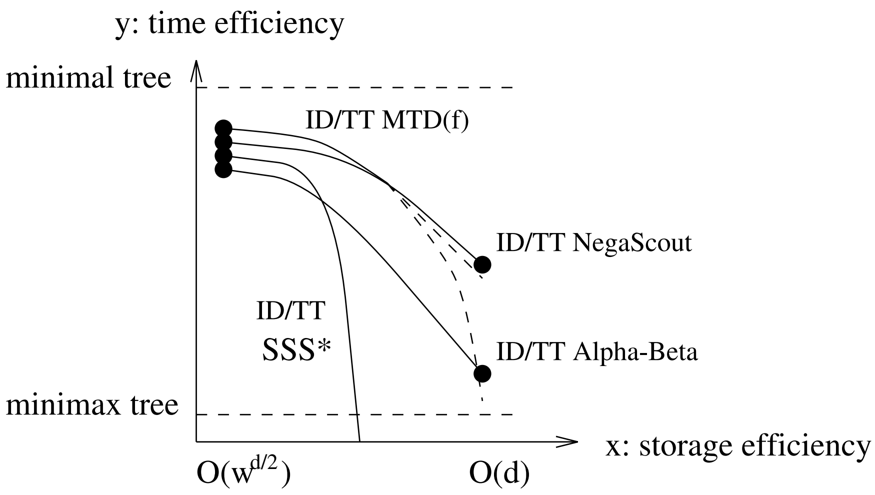

Figure 4.29: Performance Picture of Practical Algorithms

MT-SSS*. (The two dotted lines indicate that MTD(f)'s performance in low memory situations varies. It depends on the number of re-searches, as with MT-SSS*. A narrow evaluation-function range causes less re-searches and a better performance, see figures 4.25 and 4.27, and their explanation.) Again, the experiments showed that memory requirements of $O(w^{d/2})$ are perfectly reasonable.

4.4 SSS* and Simulations

The list in section 2.3.2 summarized the general view on SSS* in five points. Three of these points were drawbacks that were declared "solved" in section 4.1.3. 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. Experiments confirm that Alpha-Beta can out-search SSS*.

The second advantage was that SSS* and DUAL* expand significantly less leaf nodes. 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 MT-SSS* and MT-DUAL* compare unfavorably to Alpha-Beta and NegaScout 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. Simulation results show that for fixed-depth searches, without transposition tables and iterative deepening, SSS*, DUAL* and NegaScout are major improvements over simple Alpha-Beta [63, 85, 88, 116]. For example, one study shows SSS* and DUAL* building trees that are about half the size of those built by Alpha-Beta [85]. This is in sharp contrast to the results reported here. The reason for this disparity with previously published work 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 real 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 [3, 87, 129].

- *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 and move ordering based on shallow searches [31].

- *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 [128].

- *Transpositions*
  The fact that the search space is most 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 [107]. Of no less importance is the better move ordering, which drastically improves the effectiveness of Alpha-Beta.

Furthermore, in many simulation experiments the nodes are counted in an inconsistent manner—for example, re-expansions are not counted in SSS*, but in NegaScout they are [85, 116]. The algorithms used in simulations are often significantly different from those used in applications, making reported node counts a bad predictor of execution time in practice. The question of finding a set of representative test positions is a problem for simulations as well. A large number of different artificial trees is not necessarily a realistic test set.

The impact of the Alpha-Beta enhancements is significant: many state-of-the-art game-playing programs are reported to approach their theoretical lower bound, the minimal tree [40, 43, 107, 125]. 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 different 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 on the strength of the value interdependence, on the number of cutoffs in the previous iteration, and on the 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 is that trees that are generated in practice are 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 model the behavior of algorithms on realistic minimax trees reliably.

All of the simulations that we know of include at most one of the above four features [20, 21, 33, 28, 54, 63, 85, 88, 116, 118, 140]. 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 of all, the move ordering is not uniform throughout the tree (see figure 5.1). Second, and more important, the high level of move ordering is not a cause but an effect. It is caused by techniques (like the history heuristic) that make use of phenomena like a variable branching factor, value interdependence, value independence and transpositions. These causes and effects are 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 section 2.3.2 we mentioned five points describing the general view on SSS*: it (1) is difficult to understand, (2) has unreasonable memory requirements, (3) is slow, (4) provably dominates Alpha-Beta in expanded leaves, and (5) expands significantly fewer leaf nodes than Alpha-Beta. The validity of these points has been examined by numerous researchers in the past [33, 63, 85, 88, 116, 121, 140]. 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 [20, 118]. 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 have restricted themselves to simulations. With our reformulation we were finally able to use real programs to answer 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 model real trees reliably. In the past we have performed simulations too [28]. 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 section 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 unanswered.

Background

Seeing that the null-window Alpha-Beta ideas stood up in practice, working through the intricacies of solution trees and transposition tables, and finding that solution trees do fit in memory, has been joint work with Jonathan Schaeffer.

It all started on a day in early October 1994 in Edmonton, just before the first snow would fall, when we were having a quick lunch in a Korean restaurant on campus. After talking a bit about SSS*, solution trees, and transposition tables, we tried to estimate how much memory a max solution tree would take up in chess. To our surprise, the number seemed entirely reasonable. Since this contradicted all the papers that we knew of, we did some experiments, in the hope that they would show us where our quick calculation had gone wrong. As has been shown in section 4.1, they didn't; the estimates were correct. The theoretical work on SSS* from chapter 3 turned out to have a direct relevance to practice.

Jonathan Schaeffer's insight and experience were a big asset in designing the experiments and explaining the results. Finding out that SSS* turned out to be easily implementable in real programs, and then finding that it did not bring significant gains over NegaScout, caused much excitement as well as long discussions. Some of the ideas born in those discussions bore fruit, as the next chapter will show.