Chapter 5

The Minimal Tree?

The experiments showed that all tested algorithms perform quite close together. This raises the question whether the enhancements are causing the programs to approach their theoretical limit on the performance, the minimal tree. In this chapter we re-examine the concept of the minimal tree in the light of the experiments. Results from this chapter have been published in [114].

The search tree built by Alpha-Beta is exponential in the depth of the tree. With a constant branching factor w and search depth d, Alpha-Beta builds a search tree ranging in size from $O(w^{\lceil d/2 \rceil})$ to $O(w^d)$. Given the large gap between the best and worst case scenarios, the research effort has concentrated on methods to ensure that the search trees built in practice come as close to the best case as possible. Alpha-Beta enhancements such as minimal window searching, move ordering and transposition tables have been successful at achieving this. Numerous authors have reported programs that build search trees within 50% of the optimal size [40, 41, 125]. This is quite a remarkable result, given that a small error in the search can lead to large search inefficiencies.

This section examines the notion of the minimal Alpha-Beta search tree. The notion *minimal tree* arises from Knuth and Moore's pioneering work on search trees with a constant branching factor, constant search depth and no transpositions [65]. In practice, real game trees have variable branching factor and are usually searched to variable depth. Since two search paths can transpose into each other, nodes in the tree can have more than one parent, implying that the search tree is more precisely referred to as a search *graph*. For a real game, what is the minimal search graph?

We introduce the notion of the *left-most* minimal graph, the minimal graph that a left-to-right Alpha-Beta traversal of the tree would generate. The *real* minimal graph is too difficult to calculate, but upper bounds on its size show it to be significantly smaller than the left-most minimal graph. The insights gained from these constructions lead to ideas for several Alpha-Beta enhancements. One of them, *enhanced transposition cutoffs*, results in significant search reductions that translate into tangible program execution time savings.

5.1 Factors Influencing Search Efficiency

Several authors have attempted to approximate the minimal graph for real applications (for example, [40]). In fact, what they have been measuring is a minimal graph as generated by a left-to-right, depth-first search algorithm. Conventional Alpha-Beta search considers nodes in a left-to-right manner, and stops work on a sub-tree once a cutoff occurs. However, there may be another move at that node capable of causing the same cutoff, possibly achieving that result by building a smaller search tree. A cutoff caused by move *A* may build a larger search tree than a cutoff caused by move *B* because of three properties of search trees:

1. *Move ordering*
   Move *B*'s search tree may be smaller because of better move ordering. Finding moves that cause a cutoff early will significantly reduce the tree size.

2. *Smaller branching factor*
   Move *B* may lead to a search tree with a smaller average branching factor. For example, in chess, a cutoff might be achieved with a forced series of checking moves. Since there are usually few moves out of check, the average branching factor will be smaller.

3. *Transpositions*
   Some moves may do a better job of maximizing the number of transpositions encountered. Searching move *B*, for example, may cause transpositions into previously encountered sub-trees, thereby reusing available results.

Note that while the last two points are properties of search trees built in practice, most search-tree models and simulations do not take them into consideration (for example, [57, 63, 85, 88, 99, 101, 116, 118, 121]).

Move Ordering

Considerable research effort has been devoted to improving the move ordering, so that cutoffs will be found as soon as possible (for example, the history heuristic, killer heuristic, iterative deepening and transposition tables [128]). Ideally, only one move should be considered at nodes where a cutoff is expected.

To see how effective this research has been, we conducted measurements using the three programs, Chinook, Keyano and Phoenix, covering the range of high (36 in chess) to low (3 in checkers) branching factors (Othello has 10). Data points were averaged over 20 test positions. To be able to build reasonably sized trees, all tests used iterative deepening. The tests in chapter 4 were concerned with the performance of algorithms, and reported cumulative node counts over all iterations. Here we are interested in comparing sizes of trees. Therefore we only report the tree size of the last iteration. Including nodes of previous iterations could create a disturbance.

The different branching factors of the three games affect the depth of the search trees built within a reasonable amount of time. For a d-ply search, the deepest nodes

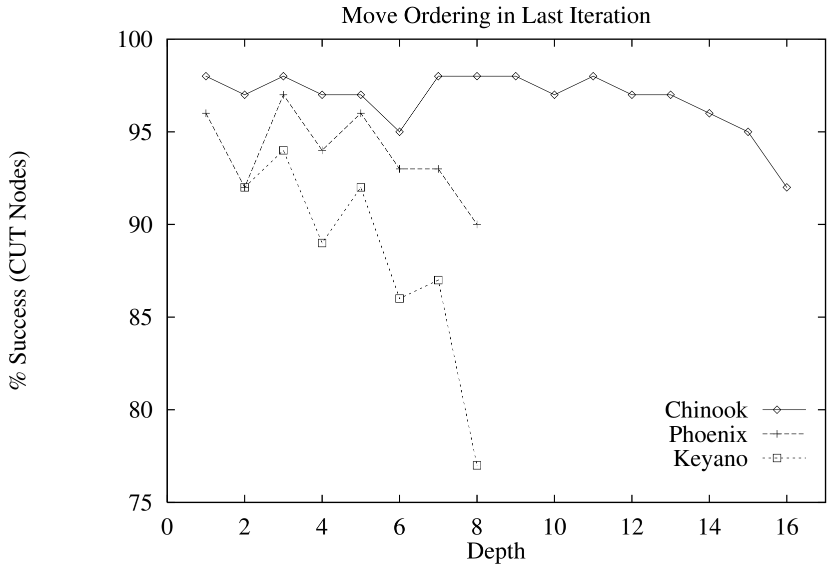

Figure 5.1: Level of Move Ordering by Depth

with move ordering information are at depth $d - 1$. Leaves have no move ordering information.

Figure 5.1 shows how often, during the last iteration of an Aspiration NegaScout search, the first move considered caused a cutoff at nodes where a cutoff occurred (note the vertical scale). For nodes that have been searched deeply, we see a success rate of over 90–95%, in line with results reported by others [41]. Since the searches used iterative deepening, all but the deepest nodes benefited from the presence of the best move of the previous iteration in the transposition table. Near the leaf nodes, the quality of move ordering decreases to roughly 90% (75% for Keyano). Here the programs do not benefit from the transposition table and have to rely on their move-ordering heuristics (dynamic history heuristic for Chinook; static knowledge for Keyano). Unfortunately, the majority of the nodes in the search tree are at the deepest levels. Thus, there is still some room for improvement.

Of the three programs, Chinook consistently has the best move ordering results. The graph is misleading to some extent, since the high performance of Chinook is partially attributable to the low branching factor. The worst case is that a program has no knowledge about a position and effectively guesses its choice of first move to consider. With a lower branching factor (roughly 8 in non-capture positions), Chinook has a much better chance of correctly guessing than does Phoenix (branching factor of 36).

A phenomenon visible in the figure is an odd/even oscillation. At even levels in the

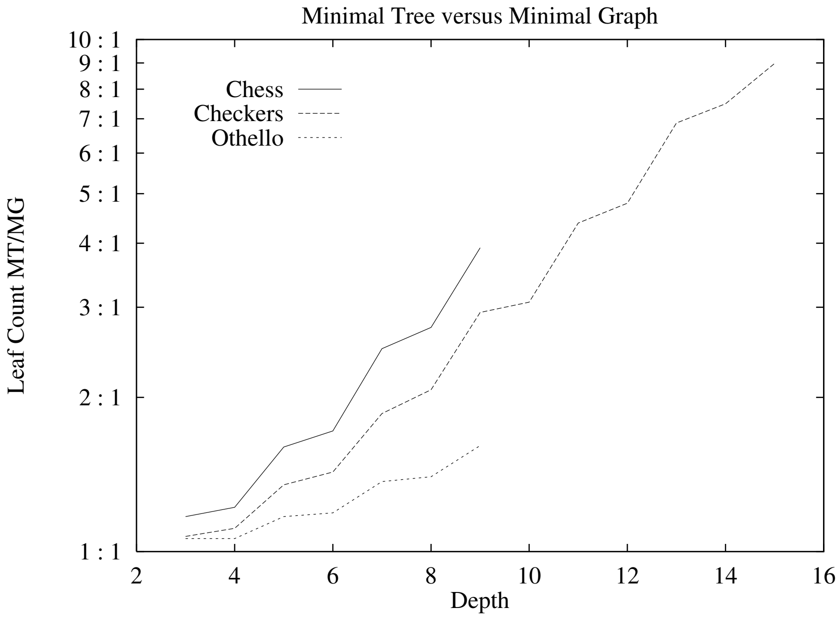

Figure 5.2: Comparing the Minimal Tree and Minimal Graph

tree, the move ordering appears to be less effective than at odd levels. This is caused by the asymmetric nature of the search tree, where nodes along a line alternate between those with cutoffs (one child examined) and those where all children must be examined. This is clearly illustrated by Knuth and Moore's formula for the minimal search tree, $w^{\lfloor d/2 \rfloor} + w^{\lceil d/2 \rceil} - 1$ leaf nodes, whose growth ratio depends on whether d is even or odd.

The evidence suggests that the research on move-ordering techniques for Alpha-Beta search has been very successful.

Variable Branching Factor

Analyses of Alpha-Beta often use the simplifying assumptions of a fixed branching factor and depth to the search tree. In practice, minimax trees have a less regular structure with a variable branching factor and depth. Algorithms like Conspiracy Number search [87, 129] and Proof Number search [2] exploit this irregularity by using a "least-work-first" strategy. For a number of application domains with a highly irregular tree structure, such as chess mating problems or the game of qubic, these algorithms search more efficiently than Alpha-Beta-like algorithms [3].

Transpositions

In many application domains the search space is a graph: nodes can have multiple parents. To search this graph efficiently with a tree-search algorithm like Alpha-Beta, nodes are stored in a transposition table. For most games, transposition tables can be used to make significant reductions in the search tree.

For our test programs, we examined the size of the minimal tree with and without transpositions. The result is shown in figure 5.2 (the method used to compute this graph will be explained in section 5.2). Note the logarithmic vertical scale. Identifying transpositions can reduce the size of the minimal tree for a chess program searching to depth 9 by a factor of 4. In checkers searching to depth 15 yields a difference of a factor of 9. Othello has less transpositions, although there still is a clear advantage to identifying transpositions in the search space. In chess and checkers, a move affects few squares on the board, meaning that a move sequence A, B, C often yields the same position as the sequence C, B, A. This is usually not true in Othello, where moves can affect many squares, reducing the likelihood of transpositions occurring.

It is interesting to note that this figure also shows an odd/even effect, for the same reasons as discussed previously.

To understand the nature of transpositions better we have gathered some statistics for chess and checkers. It turns out that roughly 99% of the transpositions occur between the nodes at the same depth in the tree. Relatively few transposition nodes have parents of differing search depths. (Although 1% of 1,000,000 is not negligible, especially since the number of transpositions does not indicate how big the sub-trees were whose search was prevented.) Another interesting observation is that the number of transpositions is roughly linear to the number leaf nodes. In checkers and chess, identifying transpositions reduces the effective width of nodes in the search tree by about 10 to 20%, depending primarily on characteristics of the test position. In endgame positions, characterized by having only a few pieces on the board, the savings can be much more dramatic.

Conclusion

Having seen the impact of three factors on the efficiency of minimax search algorithms, we conclude that the often-used uniform game tree is not suitable for predicting the performance of minimax algorithms in real applications [108, 111]. The minimal tree for fixed w and d must be an inaccurate upper bound on the performance of minimax search algorithms. In the next section we will discuss other ways to perform a best-case analysis.

5.2 The Left-First Minimal Graph

Many simulations of minimax search algorithms have been performed using a comparison with the size of the minimal tree as the performance metric (for example, [63, 85]). They conclude that some Alpha-Beta variant is performing almost perfectly, since the size of trees built is close to the size of the minimal search tree. Unfortunately, as pointed out previously, simulated trees have little relation to those built in practice.

For most games, the search "tree" is a directed graph. The presence of transpositions, nodes with more than one parent, makes it difficult to calculate the size of the minimal graph accurately. However, by using the following procedure, it is possible to compute the size of the graph traversed by a left-to-right, depth-first search algorithm like Alpha-Beta [40]. In the following, the transposition table is used to store intermediate search results. Trees are searched to a fixed depth.

1. *Alpha-Beta:* Compute the minimax value f of the search using any Alpha-Beta-based algorithm, such as NegaScout. At each node the best move (the one causing a cutoff or, failing that, the one leading to the highest minimax value) is saved in the transposition table.

2. *Minimal Tree:* Clear the transposition table so that only the positions and their best moves remain (other information, like search depth or value is removed). Repeat the previous search using the transposition table to provide only the best move (first move to search) at each node (no transpositions are allowed). Alpha-Beta will now traverse the minimal tree, using the transposition table as an oracle to select the correct move at cutoff nodes always. Since our transposition table was implemented as a hash table, a possibility of error comes from table collisions (no rehashing is done). In the event of a collision, searching with a window of $\alpha = f-1$ and $\beta = f+1$ will reduce the impact of these errors. Alternatively, a more elaborate collision-resolution scheme can be used to eliminate this possibility.

3. *Minimal Graph:* Clear the transposition table again (except for best moves). Do another search, using the best-move information in the transposition table. Allow transpositions, so that if a line of play transposes into a previously seen position, the search can re-use the previous result (assuming it is accurate enough). Again, a minimal search window ($\alpha = f - 1$, $\beta = f + 1$) is used. The minimal tree is searched with transpositions, resulting in a minimal graph.

Of course, for this procedure to generate meaningful numbers, the transposition table must be large enough to hold at least the minimal graph. Our table size was chosen to be consistent with the results in section 4.1.2.

The minimal graph has been used by many authors as a yardstick to compare the performance of their search algorithms in practice. For example, in chess, *Belle* is reported to be within a factor of 2.2 of the minimal Alpha-Beta tree [40], *Phoenix* within 1.4 [125], *Hitech* within 1.5 [40] and *Zugzwang* within 1.2 [41]. Using the three-step procedure, we have measured the performance of Chinook, Keyano and Phoenix. The results of the comparison of NegaScout against this minimal graph are shown in figure 5.3 (based on all nodes searched in the last iteration). The figure confirms that the best programs are searching close to the minimal graph (within a small factor).

An interesting feature is that all three games, Othello and chess in particular, have significantly worse performance for even depths. The reason for this can be found in the structure of the minimal tree. In going from an odd to an even ply, most of the new nodes are nodes where a cutoff is expected to occur. For the minimal graph, their

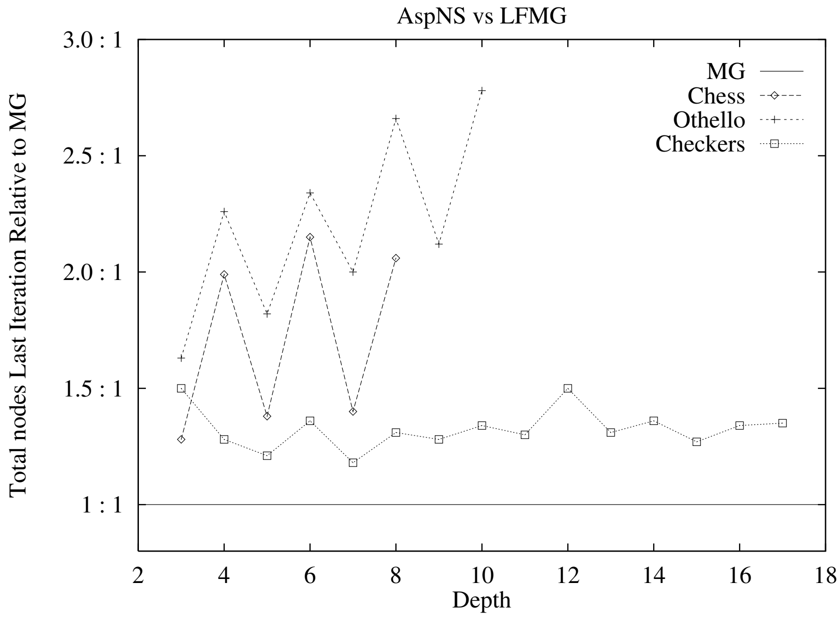

Figure 5.3: Efficiency of Programs Relative to the Minimal Graph

children count as just one node access. However, the search algorithm may have to consider a number of alternatives before it finds one that causes the cutoff. Therefore, at even plies, move ordering is critical to performance. On the other hand, in going from an even to an odd ply, most of these leaves are children of nodes where no cutoff is expected. All of the leaves are part of the minimal graph. Hence, at these nodes move ordering has no effect since all children have to be searched anyway.

The preceding leads to an important point: reporting the efficiency of a fixed-depth search algorithm based on odd-ply data is misleading. The odd-ply iterations give an inflated view of the search efficiency. For odd-ply searches, all three programs are searching with an efficiency similar to the results reported for other programs. However, the even-ply data is more representative of real program performance and, on this measure, it appears that there is still room for improvement. In light of this, the *Hitech* results of 1.5 for 8-ply searches seem even more impressive [40].

The Left-First Minimal Graph and The Real Minimal Graph

The previous section discussed a minimal graph for the comparison of algorithms that search trees in a left-to-right manner, such as Alpha-Beta, NegaScout, SSS*, DUAL* and MTD(f). Although the last three are usually called “best-first”, they expand new children at each node in a left-to-right order. On a perfectly ordered tree, all algorithms expand the same tree.

This minimal graph is not necessarily the smallest possible. Consider the following scenario. At an interior node N, there are two moves to consider, A and B. Searching A causes a cutoff, meaning move B is not considered. Using iterative deepening and transposition tables, every time N is visited only move A is searched, as long as it continues to cause a cutoff. However, move B, if it had been searched, was also sufficient to cause a cutoff. Furthermore, what if B can produce a cutoff by building a smaller search tree than for move A? For example, in chess, B might start a sequence of checking moves that leads to the win of a queen. The smaller branching factor (because of the check positions) and the magnitude of the search score will help reduce the tree size. In contrast, A might lead to a series of non-checking moves that culminates in a small positional advantage. The larger branching factor (no checking moves) and smaller score can lead to a larger search tree. Most minimax search algorithms stop when they find a cutoff move, even though there might be an alternative cutoff move that can achieve the same result with less search effort.

In real applications, where w and d are not uniform, the minimal graph defined in the previous section is not really minimal, because at cutoff nodes no attempt has been made to achieve the cutoff with the smallest search effort. The "minimal graph" in the literature [40, 41, 125] is really a *left-first* minimal graph (LFMG), since only the left-most move causing a cutoff is investigated. The *real* minimal graph (RMG) must select the cutoff move leading to the smallest search tree.

The preceding suggests a simple way of building the RMG, by enhancing part 1 of the minimal graph construction algorithm:

1. Search all moves at a cutoff node, counting the number of nodes in the sub-trees generated. The move leading to a cutoff with the smallest number of nodes in its search tree is designated "best".

In other words, explore the entire minimax tree, looking for the smallest minimal tree.

Obviously, this adds considerably to the cost of computing the minimal graph. An optimization is to stop the search of a cutoff candidate as soon as its sub-tree size exceeds the size of the current cheapest cutoff.

Unfortunately, finding the size of the RMG is not that simple. This solution would only work if there were no transpositions. In the presence of transpositions, the size of a search can be largely influenced by the frequency of transpositions. Consider interior node N again. Child A builds a tree of 100 nodes to generate the cutoff, while child B requires 200 nodes. Clearly, A should be part of the minimal graph. Interior node M has two children, C and D, that cause cutoffs. C requires 100 nodes to find the cutoff, while D needs 50. Obviously, D is the optimal choice. However, these trees may not be independent. The size of D's tree may have been influenced by transpositions into work done to search B. If B is not part of the minimal graph, then D cannot benefit from the transpositions. In other words, minimizing the tree also implies maximizing the benefits of transpositions. Since there is no known method to predict the occurrence of transpositions, finding the minimal graph involves enumerating all possible sub-graphs that prove the minimax value.

Computing the real minimal graph is a computationally infeasible problem for non-trivial search depths. The number of possible minimal trees is exponential in the size of the search tree. Transpositions increase the complexity of finding the RMG by making the size of sub-trees interdependent. Choosing a smaller sub-tree at one point may increase the size of the total solution. We have not found a solution for finding the *optimal* RMG. The following describes a method to approximate its size.

5.3 Approximating the Real Minimal Graph

We have found two methods to approximate the RMG that find an upper bound on its size that is smaller than the LFMG. The first approach involves trying to maximize the number of transpositions in the tree. The second approach is to exploit the variable branching factor of some games, to select cutoff moves that lead to smaller search trees. We will call the graph generated using these ideas an approximate RMG (ARMG).

Maximizing Transpositions

A simple and relatively cheap enhancement to improve search efficiency is to try and make more effective use of the transposition table. Consider interior node N with children B and C. The transposition table suggests move B and as long as it produces a cutoff, move C will never be explored. However, node C might transpose into a part of the tree, node A, that has already been analyzed (figure 5.5). Before doing any search at an interior node, a quick check of all the positions arising from this node in the transposition table may result in finding a cutoff. The technique to achieve *Enhanced Transposition Cutoffs*, ETC, performs transposition table lookups on successors of a node, looking for transpositions into previously searched lines. Figure 5.4 shows the standard Alpha-Beta-with-transposition-table pseudo code, with the ETC code marked by **. In a left-to-right search, ETC encourages sub-trees in the right part of the tree to transpose into the left.

Figure 5.6 shows the results of enhancing Phoenix with ETC. For search depth 9, ETC lowered the number of expanded leaf nodes by a factor of 1.28 for NegaScout enhanced with aspiration searching. Using MTD(f), the cumulative effect is a factor of 1.33 fewer leaf nodes as compared to Phoenix' original algorithm (not shown).

Figure 5.7 shows that the effect of ETC in Chinook is of a comparable magnitude. The Othello results are not shown. There are relatively few transpositions in the game and, hence, the effect of ETC is small (roughly 4% for depth 9).

The reduction in search tree size offered by ETC is, in part, offset by the increased computation per node. For chess and checkers, it appears that performing ETC at all interior nodes is not optimal. A compromise, performing ETC at all interior nodes that are more than 2 ply away from the leaves, results in most of the ETC benefits with only a small computational overhead, making it well suited for use in both on-line and off-line algorithms. Thus, ETC is a practical enhancement to most Alpha-Beta search programs.

In addition, we have experimented with more elaborate lookahead schemes involving shallow searches. For example, ETC can be enhanced to transpose also from left to

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

Figure 5.4: ETC pseudo code

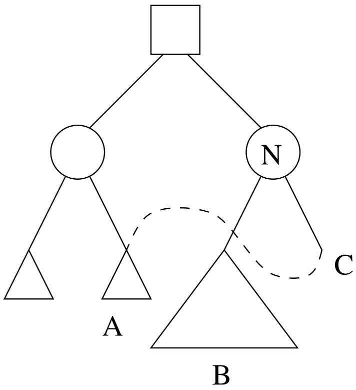

Figure 5.5: Enhanced Transposition Cutoff

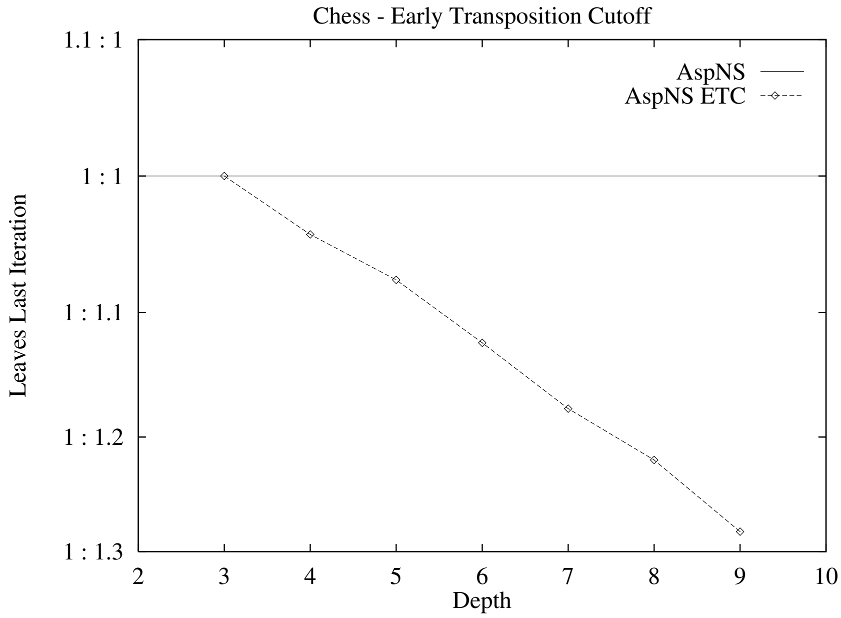

Figure 5.6: Effectiveness of ETC in Chess

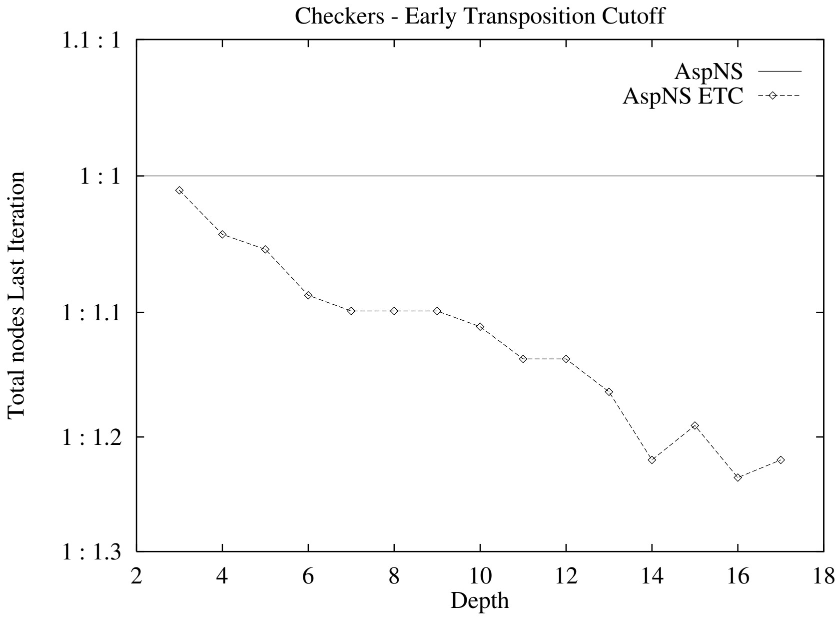

Figure 5.7: Effectiveness of ETC in Checkers

right. At an interior node, all the children's positions are looked up in the transposition table. If no cutoff occurs, then check to see if one of the children leads to a position with a cutoff score that has not been searched deep enough. If so, then use the move leading to this score as the first move to try in this position. Unfortunately, several variations on this idea have failed to yield a tangible improvement.

Minimaxing to Exploit a Variable Branching Factor

The ARMG can be further reduced by recognizing that all cutoffs are not equal; some moves may require less search effort. Ideally, at all interior nodes the move leading to the cutoff that builds the smallest search tree should be used. Unfortunately, without an oracle, it is expensive to calculate the right move. In this section, we present a method for finding some of the cheaper cutoffs, allowing us to obtain a tighter upper bound on the ARMG.

Instead of performing a full minimax search to find the cheapest cutoff, we perform a minimax search at the lowest plies in the tree only. The best moves at higher plies in the tree have already been optimized by previous iterative deepening searches. Whenever a cutoff occurs, we record the size of the sub-tree that causes it. Then we continue searching at that node, looking for cheaper cutoffs. The cutoff move leading to the smallest sub-tree is added to the transposition table. A problem with this approach is that in discarding a sub-tree because it was too big, we may also be throwing away some useful transpositions. Therefore, an extra Alpha-Beta pass must traverse the best

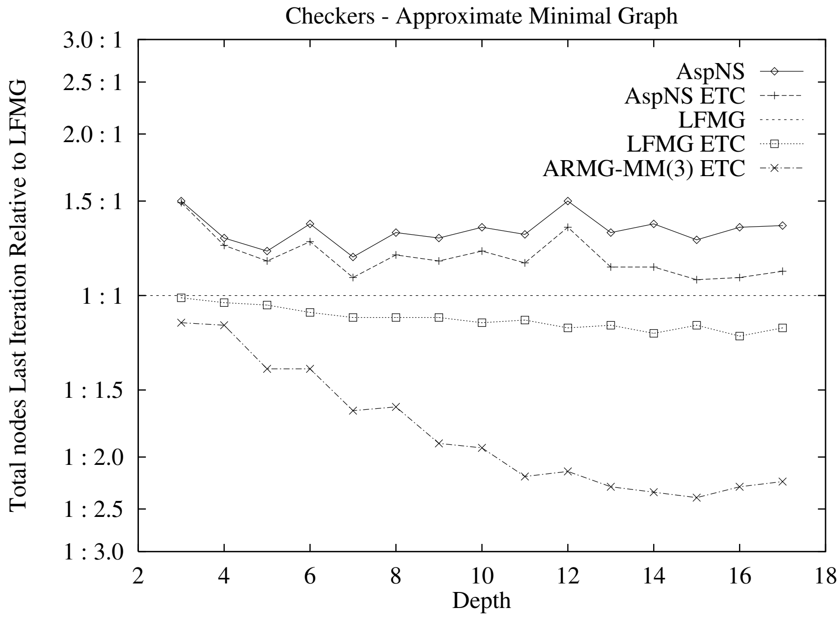

Figure 5.8: LFMG Is Not Minimal in Checkers

moves again, to count the real size of the approximated minimal graph. (For the best result every transposition should be counted for the full size of the sub-tree that it represents. We assume as it were, that it will be removed and that the algorithm has to search the sub-tree itself.)

The results for Chinook and Keyano are shown in figures 5.8 and 5.9. ARMG-MM(d) means that the last d ply of the search tree were minimaxed for the cheapest cutoff. Chinook used MM(3), while Keyano used MM(2). Othello has a larger branching factor than checkers, resulting in MM(3) taking too long to compute. The chess results are not reported since the branching factor in the search tree is relatively uniform (except for replies to check), meaning that this technique cannot improve the ARMG significantly (as has been borne out by experiments). We do show in figure 5.10 the possible savings of ETC on the LFMG.

In checkers the forced capture rule creates trees with a diverse branching factor. The ARMG can take advantage of this. The savings of ARMG-MM(3) are around a factor of 2. Minimaxing a bigger part of the graph (such as MM(4) or greater) will undoubtedly create an even smaller "minimal" graph.

Othello's branching factor can vary, but tends to be less volatile than for checkers, accounting for the lower savings (a factor of 1.5–1.6). In addition, since there are fewer transpositions possible in Othello, there is less risk of throwing away valuable transpositions.

Chess has a fairly uniform branching factor except for moving out of check. Conse-

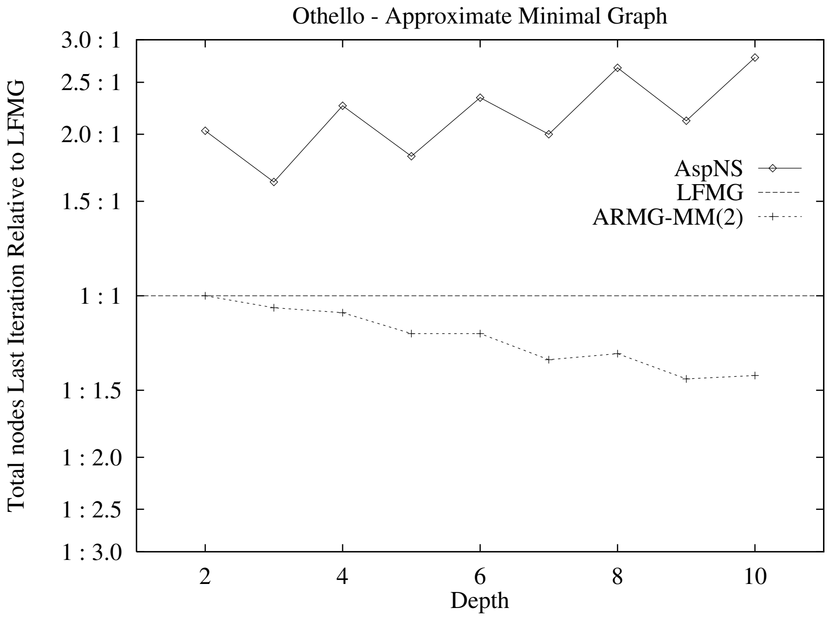

Figure 5.9: LFMG Is Not Minimal in Othello

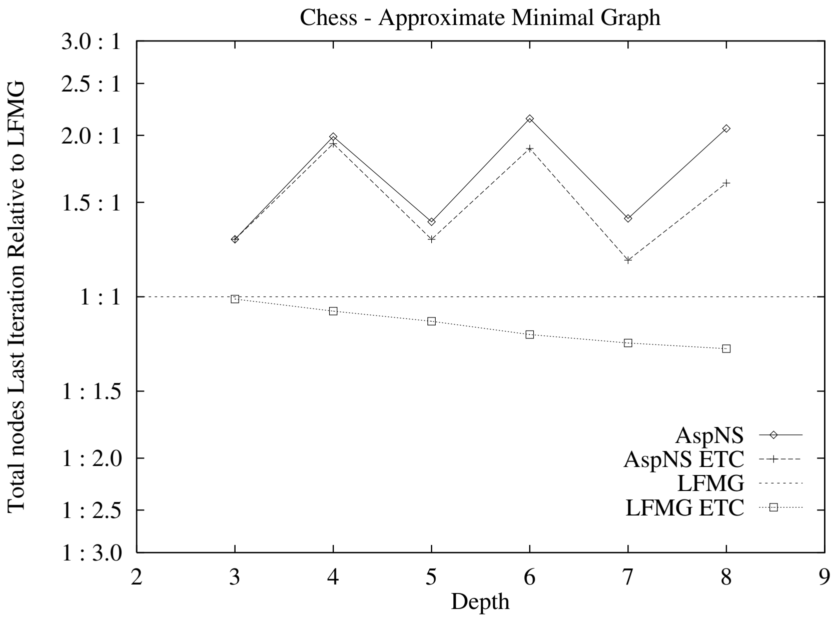

Figure 5.10: LFMG Is Not Minimal in Chess

quently, our test positions failed to show significant reductions in the ARMG using our approach. However, figure 5.10 shows that ETC still reduces the LFMG substantially. More research is required to get a tighter bound for chess.

Thus minimaxing can find, in an off-line computation, a smaller minimal graph. The overhead involved in minimaxing a few plies of the tree makes this method unsuited for use in on-line, real-time, algorithms. We tried many ideas for exploiting transpositions and non-uniform branching factors in real-time search. All the ideas are interesting and show potential on individual positions. However, every one of our ideas (except ETC) fails to yield a consistent improvement when averaged over a test set of 20 positions.

Seeing the Forest through the Trees

Figure 5.11 gives a road map of the relations between all the different kinds of approximations of the real minimal graph. In the left bottom corner the worst case of minimax search can be found. In the top right corner is the optimal case for real trees (those with variable branching factor and transpositions). From left to right in the diagram, the effectiveness of transpositions is improved. From bottom to top in the diagram, the quality of move ordering is improved. The "X"s represent data points in the continuum which are either too difficult to calculate or are of no practical interest. The top-right entry, the Real Minimal Graph, represents the theoretically important, but unattainable, perfect search. Abbreviations used include TT (transposition table), ID (iterative deepening), HH (history heuristic, or some equally good ordering scheme), ARMG (approximate real minimal graph), LFMG (left-first minimal graph), ETC (enhanced transposition cutoffs) and MM(d) (minimax d-ply searches for finding cheapest cutoffs).

The figure illustrates how far game-tree searching has evolved since the invention of minimax search. The entry for Alpha-Beta enhanced with TT, ID, HH and ETC data point is almost the new state-of-the-art performance standard. To be complete, the figure should contain extra entries, for null-window enhancements such as NegaScout and MTD(f). The new state-of-the-art algorithm is the enhanced version of MTD(f). As this section shows, the gap between what can be achieved in practice and the real minimal graph is larger than previously suspected. Thus, there is still room for extending the road map to narrow the distance between where we are and where we want to be.

5.4 Summary and Conclusions

The notion of the minimal search tree is a powerful tool to increase our understanding of how minimax search algorithms work, and how they can be improved. One use of the minimal tree is as a yardstick for the performance of minimax search algorithms and their enhancements. However, trees as they are built by real applications, such as game-playing programs, are neither uniform, nor trees. Therefore, we have to be more precise in our definition of the minimal search tree and graph. This section defined two

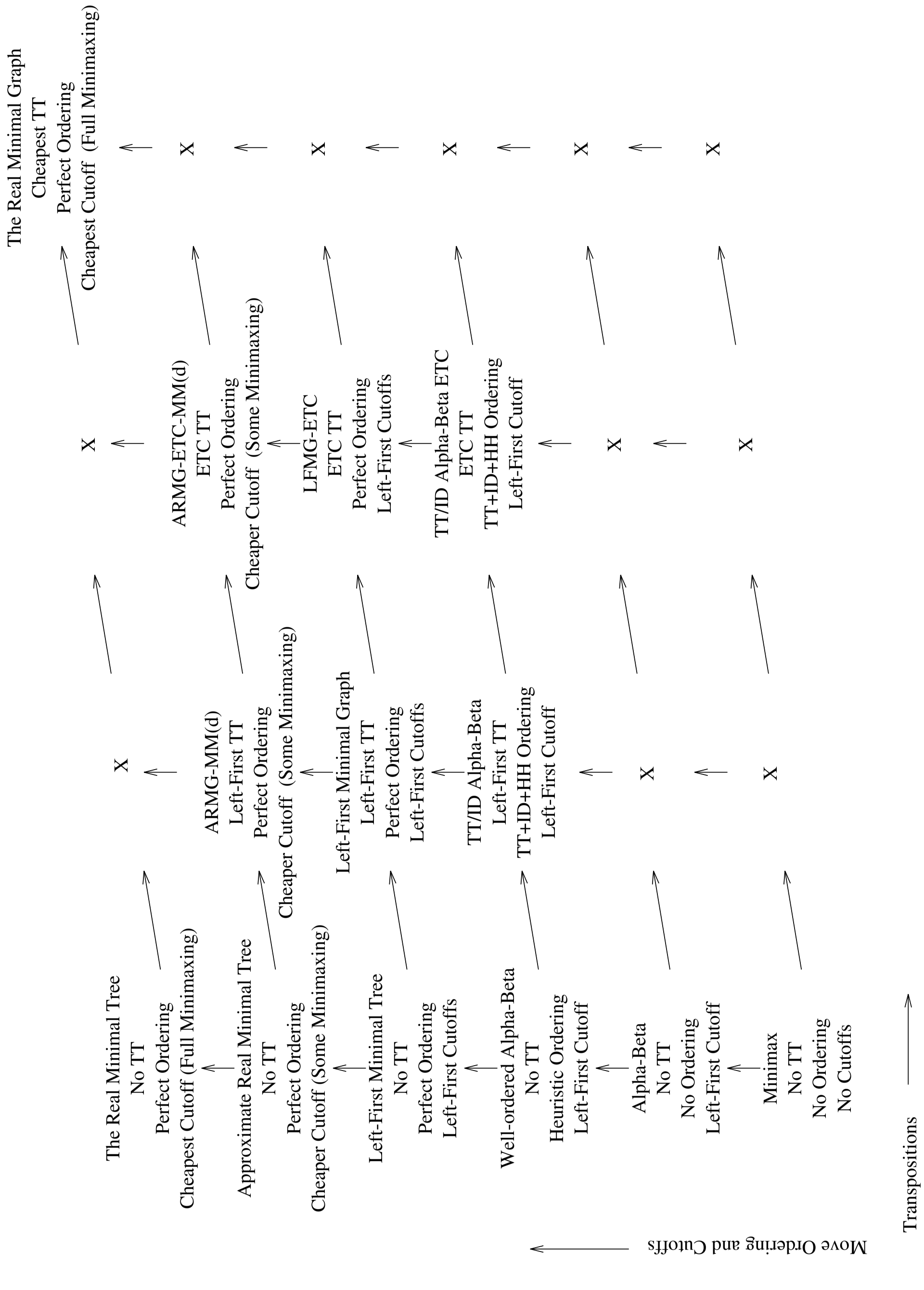

Figure 5.11: A Roadmap towards the Real Minimal Graph

types of minimal graphs:

1. The *Left-First Minimal Graph* is constructed by a left-to-right, depth-first traversal of the search tree (using, for example, Alpha-Beta). The use of a transposition table allows for the possibility of transpositions, making the search tree into a search graph.

2. *The Real Minimal Graph* is the minimum effort required for a search. However, this search requires an oracle so that at cutoff nodes the branch leading to the cutoff requiring the least amount of search effort is selected. Finding the size of the real minimal graph is difficult because the utility of transpositions has to be maximized. This involves enumerating all possible sub-graphs that prove the minimax value. Lacking an ordering in the set of transpositions, this is a computationally intractable problem.

We arrive at the following conclusions:

- For performance assessments of minimax search algorithms, the minimal tree is an inadequate measure. Some form of minimal graph that takes variable width and transpositions into account should be used.

- Alpha-Beta-based search algorithms perform much closer to the left-first minimal graph for odd depths than for even depths. In performance comparisons, even ply results should be reported. For odd search depths, the tree size is dominated by the last ply. This ply consists largely of nodes where no cutoff occurs and, hence, the move ordering is of no importance. In effect, results for odd-ply search depths hide the results at nodes where move ordering is important.

- The real minimal graph is significantly smaller than the left-first minimal graph, the usual metric for search efficiency of minimax algorithms. Therefore, these algorithms are not as efficient as is generally believed.

- Enhanced Transposition Cutoffs improve Alpha-Beta searching by trying to maximize the number of transpositions in the search. The results indicate this to be a significant improvement.

Various publications indicate that game-playing programs are almost perfect in their search efficiency. Our results show that there remains room for improvement.

Background

Many of the analyses in the last two chapters were done in discussions with Jonathan Schaeffer. The initial reason for the work on minimal graphs was our disappointment with the relatively small gains that MTD(f) yielded—only a few percentage points. We assumed that the three programs were all close to the minimal tree. As it turned out, there was much to be learned. Our discussions and numerous failed attempts at improving the algorithms helped creating much of the insight behind the work in this chapter. Some ideas worked, notably the ETC, which was suggested in an early stage by Jonathan Schaeffer in a discussion on shallow searches, “just to try as a first cut.” As it turned out, all the more elaborate ideas on shallow searches, as well as many improvements to the history heuristic, did not yield a consistent improvement. The simple first cut, ETC, did.

Chapter 6

Concluding Remarks

This chapter concludes the research by briefly reviewing the main issues. At the end, some directions for future research are mentioned.

6.1 Conclusions

Null-window Alpha-Beta Search

A widely-used enhancement to the Alpha-Beta algorithm is the use of a search window of reduced size. Taking this idea to the limit is the reduction of the Alpha-Beta search window to a null window. A null-window Alpha-Beta call never finds the minimax value; it returns either an upper or a lower bound on it. An algorithm consisting solely of null-window Alpha-Beta calls would have to perform many re-searches to home in on the minimax value. This would reduce the efficiency of the null windows.

Chapter 3 discusses a solution to this problem. For all the nodes that the null-window Alpha-Beta call traverses, the return bounds are stored in memory. Transposition tables (TT) provide an efficient way to do this. They allow for the pruning power of null-window Alpha-Beta calls to be retained over a sequence of searches. In this way subsequent Alpha-Beta calls build on the work of previous ones, using the information to find the node to expand next. Interestingly, this results in a best-first expansion sequence. The idea is formalized in the MT framework, a framework for null-window-only best-first minimax algorithms. The memory needed to store the tree that defines a bound is $O(w^{d/2})$, for a minimax tree of uniform branching factor w and uniform depth d.

The framework allows the formulation of a number of algorithms, existing ones, such as SSS* [140], and new ones, such as MTD(f). It focuses attention on the fundamental differences between algorithms. The details of how to traverse trees are left to Alpha-Beta.

SSS*

An instance of MT called MT-SSS* evaluates the same nodes (in the same order) as SSS*. SSS* is an important algorithm because of its claim to be "better" than Alpha-Beta, since it provably never expands more nodes than Alpha-Beta. Quite a number of journal publications try to decide whether SSS* is really "better," using both analysis and simulations of the algorithm [33, 57, 72, 85, 118, 121, 140]. The general view is mixed:

1. SSS* is hard to understand.
2. SSS* stores nodes in a sorted OPEN list. The operations on this list make the algorithm slow.
3. Being a best-first algorithm, SSS*'s exponential memory usage makes it unsuited for practical use in game-playing programs.
4. SSS* provably never expands more leaf nodes than Alpha-Beta.
5. SSS* expands on average significantly less nodes than Alpha-Beta.

The first three items are the disadvantages of SSS*, while the last two are the advantages. Because of the three disadvantages, SSS* is not used in practice, despite the promise of expanding fewer nodes. In our framework, SSS* can be re-expressed as a single loop of null-window Alpha-Beta (+TT) calls. This makes SSS* easy to understand, solving the first problem of the list. Implementing the algorithm is no longer a problem. In the original publication SSS* was compared against the standard text-book version of the Alpha-Beta algorithm. Actual implementations of Alpha-Beta use many enhancements. For practical purposes, a comparison of SSS* against enhanced versions of Alpha-Beta is much more relevant. Our experiments were performed with tournament game-playing programs, for chess, Othello and checkers. These games cover the range from high to low branching factor. Using multiple programs, the chance of unreliable answers, due to peculiarities of a single game or program, is reduced. All three programs are based on Alpha-Beta enhanced with iterative deepening, transposition tables, and move ordering techniques.

MT-SSS* does not have a sorted OPEN list. The slow manipulation of the OPEN list is gone. Researchers trying to parallelize SSS* do no longer have to try and find an efficient way to parallelize the OPEN list. Since SSS* has been reformulated as a special case of Alpha-Beta, the research on parallel Alpha-Beta (see [26] for a detailed overview) is now directly applicable; the framework may lead to some new ideas for parallel best-first Alpha-Beta versions. In short, the second problem of the list is now solved.

The experiments show that the memory needs of SSS* in current game-playing programs are reasonable. Instances of the MT framework need $O(w^{d/2})$ transposition table entries to store the intermediate search results. Contrary to what the literature says, for game-playing programs under tournament conditions, this is a practical size.

An additional argument is that an analysis of Alpha-Beta's memory needs shows that to achieve high performance Alpha-Beta needs memory of about this size too. Thus, the third problem is solved too.

Having shown the first three points wrong, the reformulation of SSS* is now a practical Alpha-Beta variant. However, we have also examined the remaining two positive points. As it turns out they are wrong as well. Game-playing programs that use Alpha-Beta enhance it with forms of dynamic move reordering, such as iterative deepening (ID). This violates the assumptions of SSS*'s dominance proof, making it possible for ID-Alpha-Beta to search *less* leaf nodes than ID-SSS*, which has actually happened in our experiments.

The last point states that SSS* expands significantly less nodes than Alpha-Beta. For simulations this may be true, but not for real applications. In game-playing programs SSS* expands on average a few percent less leaf nodes than Alpha-Beta. Furthermore, these programs generally use an enhanced version of Alpha-Beta called NegaScout. Using this algorithm as the base line, even these few percents disappear (with NegaScout getting a clear advantage when interior nodes are counted as well).

Thus *all* five points are wrong: SSS* is a practical algorithm, but there is no point in using it, since practical Alpha-Beta versions out-perform it.

MTD(f)

MT-SSS* is an instance of the MT framework that starts the sequence of null-window Alpha-Beta searches at $+\infty$, descending down to the minimax value. Another instance is MT-DUAL*, which starts at $-\infty$, going up to the minimax value. Intuition says that a start value closer to the minimax value should perform better. It gets a head start on the algorithms starting at $\pm\infty$. Experiments show that this intuition is true. Creating bounds for an uninteresting value is wasteful. On average it is more efficient to start establishing bounds that are closer to the target.

MTD(f) is a new algorithm embodying this idea. It performs better than NegaScout, the current algorithm of choice for game-playing programs. In our tests MTD(f)'s improvement over NegaScout is slightly bigger than NegaScout's improvement over Alpha-Beta. The efficiency comes at no extra algorithmic complexity: just add a single control loop to a standard Alpha-Beta-based program.

One of the most interesting outcomes of our experiments is that the performance of all algorithms differs only by a few percentage points. The search enhancements used in high-performance game-playing programs improve the search efficiency to such a high degree, that the question of which algorithm to use, be it Alpha-Beta, NegaScout, SSS* or MTD(f), is no longer of prime importance. (For programs of lesser quality, the performance difference will be bigger, with MTD(f) out-performing NegaScout by a wider margin. There the best-first nature of MTD(f) will make more of a difference. Also, in some cases SSS* does not perform very well.) Judging by the performance, the enhancements have become more important than the base algorithm. Furthermore, seen from an algorithmic viewpoint, in the new framework a number of best-first algorithms become enhanced versions of Alpha-Beta.

The Alpha-Beta paradigm is versatile enough to be used for creating a best-first node selection sequence. Given that the new formulation for best-first full-width minimax search is more general, clearer, and allows a better performance, we believe that the old SSS* should become a footnote in the history of minimax search.

Traditionally depth-first and best-first search are considered to be two fundamentally different search strategies. Depth-first search uses little memory, but has a simplistic, rigid, left-to-right node expansion strategy. Best-first search, on the other hand, uses (too) much memory, but has a smart, flexible, node expansion strategy. Best-first algorithms typically recalculate after each node expansion which node appears the most promising to expand next. They jump from one part of the tree to another, a pattern that is markedly different from the fixed left-to-right sweep through the tree of a depth-first search.

Our work shows that in reality the picture is less clear cut. Simple enhancements to depth-first search create a node selection sequence that is identical to that of best-first algorithms. It is not hard to make depth-first search transcend the rigid left-to-right strategy. The view of best-first search as a memory-hungry monster with an inefficient OPEN list needs refinement too. The new MT framework solves the OPEN list problem and shows that memory requirements are not a problem for real-time search.

Thus, positioning depth-first and best-first as two irreconcilable, fundamentally different strategies, is an oversimplification. In reality, they are not that different at all. A few simple enhancements show how close they really are—at least in minimax search.

The Minimal Tree

Knuth and Moore reported a limit on the performance of any algorithm that finds the minimax value by searching a tree of uniform w and d [65]. Any such algorithm has to search at least the tree that proves the minimax value. Many authors of game-playing programs compare the size of their search tree to the size of the minimal tree, to see how much improvement of their search algorithm is still possible.

However, game playing programs do not search uniform trees. And since transpositions occur, the search *tree* is really a *graph*. One solution, adopted by several authors, is to redefine the minimal "tree" as the graph that is searched by Alpha-Beta when all cutoffs are caused by the first child at a node. We call this the Left-First Minimal Graph or LFMG. However, the LFMG is not the smallest graph that proves the minimax value. Because of transpositions and variances in w and d alternative children may cause cutoffs that are cheaper to compute. The entity where each cutoff is backed-up by the smallest sub tree is called the Real Minimal Graph, or RMG.

Finding the RMG is a computationally infeasible task for non-trivial search depths, due to transpositions. Using approximation techniques that exploit irregular branching factors or transpositions, we have found that the RMG is at least a factor of 1.25 (for chess) to 2 (for checkers) smaller than the LFMG.

Current game-playing programs are further from the minimal graph that proves the minimax value than is generally assumed. There is more room for improvement. One such improvement is the Enhanced Transposition Cutoff, or ETC. For games with many transpositions this technique reduces the search effort significantly.

6.2 Future Work

The research described in the previous chapters has uncovered a number of interesting avenues for further research. We list the following:

- *Node Expansion Criteria for MTD(f)*
  The literature describes static node expansion criteria for Alpha-Beta and SSS* [101, 116]. Finding these criteria for MTD(f) can give additional insight in the relation between the start value of a search and the size of the search tree, supplementing the experimental evidence in section 4.3.1.

- *Value/Size Experiments*
  In addition to more analysis, more experiments are needed to gain a deeper insight into the effect of the size and value of the Alpha-Beta window on the size of the search tree, extending the work on the minimax wall [85] and section 4.3.1.

- *NegaScout and MTD(f)*
  The same—more analysis, more experiments—is needed to gain a better understanding of the relation between NegaScout and MTD(f). Section 4.3.2 only scratches the surface of this relation.

- *Variable Depth MTD(f)*
  All experiments in this work were performed for fixed-depth searches, to make sure that different algorithms searched the same tree. We believe that since both algorithms use search windows that are comparable in size and value, the reaction of MTD(f) will not be very different from that of NegaScout when search extensions and forward pruning are turned on again. However, experiments are needed to show whether this is indeed the case.

- *Effect of Individual Enhancements*
  Section 4.4 describes the hazards of simulating minimax algorithm performance for real games. Although we believe that real trees are too hard to model realistically, it might be worthwhile to try to gain insight into the effect of individual aspects of real trees, such as iterative deepening, the history heuristic, search extensions, transpositions, narrow search windows, the distribution of leaf values, and the correlation between parent and child positions.

- *Exploiting Irregularity of the Branching Factor*
  Chapter 5 showed that the ARMG is significantly smaller than the LFMG. The chapter suggests that there is room (at least in some games) to exploit the irregularity of the branching factor, by devising a suitable Alpha-Beta enhancement. (See [114] for some early results.)

- *Replacement Schemes*
  The work on the relation between Alpha-Beta and SSS* (chapters 3 and 4) makes heavy use of the transposition table to store information from previous search passes. Conventional transposition table implementation choices [125] appear to be well-suited for this task. However, analysis of and experiments with storing additional information and applying different replacement schemes (see [24, 64]) can be fruitful.

- *Search Inconsistencies*
  On page 29 the problem of search inconsistencies is mentioned. Search inconsistencies occur when, using variable deepening or transposition tables, search results of different search depths (different accuracies) are compared. Every now and then a game is lost due to this problem and programmers are painfully reminded of it. However, although it is a well-known problem, it is, to the best of our knowledge, still unsolved.

- *Parallelism*
  Much research effort has been devoted to finding ways to parallelize minimax algorithms efficiently. Parallelization of the MTD family of algorithms seems viable along two lines. First of all, the conventional tree-splitting/work-stealing parallelization techniques used for Alpha-Beta (see, for example, [23, 26, 41, 75, 82]) are obvious candidates to try, since the Alpha-Beta procedure forms the heart of the MTD algorithms. Second, the calls to Alpha-Beta in the main loop of MTD can be run in parallel, each with different values for the null-window. This creates a Baudet-like scheme [8], with the exceptions that now the parallel aspiration windows have become null windows, and that in MTD the parallel Alpha-Beta-instances share information, through their transposition table. However, there is evidence that such a parallelization is not well-suited for strongly ordered search spaces (see, for example, [8, 82]).

  Passing information from one Alpha-Beta-pass to another via the transposition table is a cornerstone in the design of the MTD algorithms. Therefore, we believe that an efficient implementation of a (logically) shared transposition is a necessary condition for a succesful parallelization of MTD.

Understanding Trees

This work was driven by a desire to understand better what is going on in search trees as they are being searched by full-width minimax algorithms. For algorithms like Alpha-Beta, NegaScout, SSS* and MTD(f), we have found two notions central to our understanding: bounds and solution trees. Realizing that all these algorithms—and the minimal tree—could be understood in these terms created a new perspective on best-first and depth-first full-width minimax search. They have been a guide throughout this research.

Solution trees and the structure of the minimal tree can help explain many interesting phenomena witnessed in the experiments. Our experiments also show the limitations of our model and the danger of putting too much trust in models of reality. For example, although the minimal tree may be a good model to understand certain phenomena in search trees, it is not an accurate limit on the performance of game-playing programs. Analyses and simulations of SSS* and Alpha-Beta turned out to mispredict their relative performance in real applications severly. The reason for the difference between our results and that of simulations is that the trees generated in actual applications are complex.

It is hard to create reliable models for simulations. The field of minimax search is fortunate to have a large number of game-playing programs available. These should be used in preference to artificially-constructed simulations. Future research should try to identify factors that are of importance in real game trees, and use them as a guide in the construction of better search algorithms, instead of artificial models with a weak link to reality. For example, in pursuing the real minimal graph existing notions like bounds and solution trees proved inadequate to explain many results. Better concepts to help in reasoning about irregularity and transpositions are dearly needed. Finding them would be very useful for research on the minimal graph and on improving full-width algorithms further.