New Advances in Alpha-Beta Searching

Jonathan Schaeffer
Dept. of Computing Science, University of Alberta,
615 General Services Building,
Edmonton, Alberta, Canada T6G 2H1
jonathan@cs.ualberta.ca

Aske Plaat
Dept. of Computer Science, Erasmus University,
Room H4-31, P.O. Box 1738,
3000 DR Rotterdam, The Netherlands
plaat@theory.lcs.mit.edu

Abstract

Alpha-Beta has been the algorithm of choice for game-tree search for over three decades. Its success is largely attributable to a variety of enhancements to the basic algorithm that can dramatically improve the search efficiency. Although state-of-the-art game-playing programs build trees that are close in size to the minimal Alpha-Beta search tree, this paper shows that there is still room for improvement. Three new enhancements are presented: best-first Alpha-Beta search, better use of transpositions, and improving aspiration search under real-time constraints. Measurements show that these improvements can reduce search effort by 35%.

Keywords: Alpha-Beta, best-first search, SSS*, heuristic search, computer chess.

1 Introduction

The Alpha-Beta search algorithm is at the heart of the programming strategy for many games. Although this simplistic depth-first, brute-force approach has not found favor in the artificial intelligence community, it is hard to argue with its success. Programs such as Deep Thought in chess (playing at Grandmaster strength [10]), Chinook at checkers (the Man-Machine World Champion [27]) and Logistello in Othello (much stronger than all humans [4]) have achieved spectacular success with this algorithm. Alternative search strategies are promising in theory but the results have yet to be demonstrated in practice (for example, BPIP [1], B* [2], Conspiracy Numbers [16]).

Three decades of research into Alpha-Beta has resulted in a large number of enhancements to the algorithm including:

1. *Transpositions*. Usually the search space is referred to as a tree. However, it is actually a directed acyclic graph. Recognizing previously visited nodes allows one to eliminate potentially large portions of the tree traversed by Alpha-Beta. Transposition tables are used to save information about visited nodes, in the event that these nodes are revisited [25].
2. *Move ordering*. The effectiveness of Alpha-Beta cutoffs is maximized if the best move is considered first at all interior nodes of the search tree. Hence research has concentrated on static (knowledge-based) and dynamic (information extracted from the search tree) schemes for ordering moves in a best-to-worst order. Techniques such as iterative deepening and the history heuristic have shown that it is possible to achieve excellent move ordering [25].
3. *Minimal windows*. Alpha-Beta searches with a lower bound (α) and an upper bound (β) on the range of useful search values, the so-called search window. Narrowing this window can increase the likelihood of a cutoff [25]. The narrowest, or minimal, window occurs when $\alpha+1 = \beta$ (for integer-valued leaf nodes).
4. *Variable search depth*. All moves are not equal. Some moves are weak and should be allocated few search resources. Other moves have potential and should be explored further. Instead of a fixed-depth search, many game-playing programs reduce the search depth for weak moves and increase it for strong moves.

The effects of points 1–3 can be easily quantified. All one has to do is build fixed-depth search trees with/without the enhancement and compare the tree size. Unfortunately, it is difficult to quantify the effects of variable search depths. Here not only the size of the search tree must be considered, but also the quality of the answer provided. We do not know of any fair way of doing this comparison.

In this paper we consider three new enhancements to the Alpha-Beta algorithm. The first one concerns aspiration searching. An Alpha-Beta search can be called with a lower bound of $-\infty$ and an upper bound of $+\infty$. Experience shows that narrowing this window can significantly reduce the search effort. Aspiration search makes the initial call to Alpha-Beta with a small search window centered around the expected value. No problems occur if the search value falls in the window (has an accurate value) or exceeds the window (fail high—a better value than expected). The problem occurs when the value of the first move is less than the search window (fail low).

In the fail low scenario, our search value expectations are seriously wrong. The usual resolution of this problem is to re-search the move with the correct search window to find its true value, and then continue to examine other moves looking for an improvement. In the critical period until the program finds the right move, it may not be able to play a reasonable move if forced to move because of (time) resource constraints. Instead, on a fail low we propose to restart the search and use the transposition table values to seed a new iterative-deepening search. In this way, the new best move is quickly found. Experiments in chess show that this not only finds an alternative best move quickly, but also does so with less search effort.

The second enhancement takes the idea of minimal windows to the extreme: all searches are performed with a minimal window. The result of the search is a bound on the true value. A series of searches allows one to converge on the minimax value. One way of using this is to start with an upper bound of $+\infty$ and search to successively lower that bound until the exact value is found. Surprisingly, this algorithm expands the same leaf nodes in the same order as the best-first algorithm SSS* [28]. In effect, SSS* is now a special case of depth-first Alpha-Beta.

Instead of starting at $+\infty$ and lowering the bound (SSS*), or starting at $-\infty$ and raising the bound (DUAL* [15]), we can start with an approximation of the bound and converge from there. Using the score from the previous iteration of an iterative-deepening search yields the new MTD(f) algorithm. For chess, experimental results show this to be a 9–16% improvement over the algorithm of choice by most chess programmers (aspiration window enhanced NegaScout [22]).

The third enhancement improves the benefit of transpositions. For each node, a transposition table typically stores the depth to which that node was searched, the score achieved and the best move. When a node is visited in the search, if the table entry's value does not cause a cutoff, then the best move saved in that entry is searched first before other moves are considered. However, what if another move transposes into a previously searched part of the tree and that entry has a value sufficient for a cutoff? Conventional implementations will miss this cheap cutoff.

The Enhanced Transposition Cutoff (ETC) attempts to maximize the benefits of the transposition table by doing additional lookups. By looking up *all* successors of a node in the table, additional transpositions to other parts of the tree can be detected. In particular, if one of these lookups produces a value sufficient for a cutoff, the search can be stopped at this node. For chess, experiments show that ETC can reduce the search tree by 28%.

The benefits of these enhancements are not independent of each other. Experimental results shows that the combination of MTD(f) and ETC results in a 35% reduction over conventional aspiration window NegaScout. Given that some researchers have speculated that there is little of interest left to explore in sequential fixed-depth Alpha-Beta searching [25], the magnitude of the improvement is both significant and surprising.

2 Current State of the Art

There are two popular criteria for assessing the search efficiency: the quality of move ordering and the closeness of the search tree to the theoretical minimal tree. To give an idea of the state-of-the-art, we present measurements for these two criteria for a tournament quality chess program, Phoenix [24].

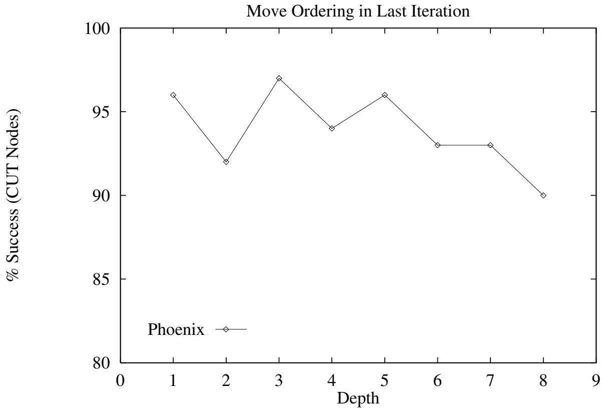

Figure 1: Quality of Move Ordering by Depth

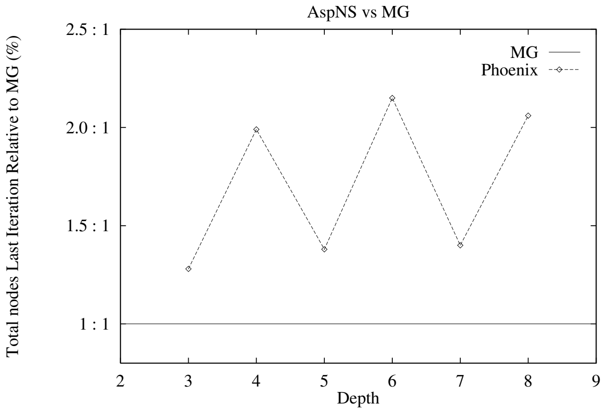

Figure 2: Efficiency Relative to Minimal Graph

2.1 Quality of 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 [25]). Ideally, only one move should be considered at nodes where a cutoff is expected.

Figure 1 shows how often the first move considered caused a cutoff at nodes where a cutoff occurred (note the vertical scale). The algorithm used is NegaScout enhanced with iterative deepening, transposition tables, aspiration windows and the history heuristic. The data points were averaged over twenty test positions. The quality of move ordering of the last ply is shown. For nodes that have been searched deeply, we see a success rate of over 90–95%, in line with results reported by others [9]. 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%. Here the program does not benefit from the transposition table and has to rely on move ordering heuristics. Unfortunately, the majority of the nodes in the search tree are at the deepest levels. Thus, there is still some room for improvement.

A phenomenon visible in the figure is an odd/even oscillation. At even levels in the 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 the formula for the minimal search tree, $w^{\lfloor d/2 \rfloor} + w^{\lceil d/2 \rceil} - 1$ leaf nodes (assuming fixed width w and depth d), 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.

2.2 The Minimal Tree

In a seminal paper in 1975, Knuth and Moore introduced the notion of the minimal tree [12]. Any algorithm that wants to find the minimax value has to search at least this tree. For actual games, where the game tree is nonuniform, the minimal tree is usually taken to be Alpha-Beta's best-case. Ebeling describes a procedure to compute the size of the search tree in relation to Alpha-Beta's best-case [8].

The minimal tree 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 [8], *Phoenix* within 1.4 (measured in 1985) [24], *Hitech* within 1.5 [8] and *Zugzwang* within 1.2 [9]. Using Ebeling's procedure, we measured the performance of the current version of Phoenix. The results of the comparison of NegaScout against this minimal tree are shown in figure 2 (based on all nodes searched in the last iteration). The figure confirms that the program can search close to the minimal tree.

An interesting feature is that Phoenix has a significantly worse performance for even depths. The reason for this can be found in the structure of the minimal tree. This 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; even-ply data is more representative of real program performance. In light of this, the *Hitech* result of 1.5 for 8-ply searches seems even more impressive [8].

3 Improvements

The measurements of the previous section indicate that the search efficiency of Alpha-Beta is at a high level. We present three new Alpha-Beta enhancements to further narrow the gap between trees built in practice and the minimal search tree.

3.1 Best-First versus Depth-First?

Alpha-Beta does a rigid depth-first, left-to-right traversal of the tree. In contrast, the best-first approach of SSS* seems more appealing. It was proven that SSS* will never examine more nodes than Alpha-Beta [5, 28], and numerous simulations showed it to build significantly smaller trees (recent publications include [3, 7, 10, 11, 23]).

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

Figure 3: SSS* as a Sequence of Alpha-Beta Searches with a Transposition Table

Despite the encouraging results, SSS* has been shunned in practice because of a number of perceived drawbacks:

- It is a complex algorithm that is difficult to understand or adapt. The algorithm is formulated as an implicit finite-state machine with six ingeniously interlocking state-space operators that manipulates a sorted OPEN list.
- It is slow, because of the overhead of maintaining a sorted OPEN list.
- It has memory requirements that are exponential in the search depth. It is widely believed that this makes SSS* unsuitable for practical use.

This unsatisfactory state of affairs has left many researchers in the field with a nagging feeling. Although Alpha-Beta-based programs achieve good results, it could be that depth-first strategies are missing out on some fundamental notion, and that best-first is a better way.[1]

Alpha-Beta Goes Best-First

The idea of minimal window search can be taken to its extreme: perform all Alpha-Beta searches with a minimal window. Since the result of a minimal window search is a lower or upper bound on the true value, a series of searches must be conducted to converge on the value. Doing extra searches sounds expensive, but a cache (the transposition table) can be used to prevent unnecessary re-searching.

Figure 3 shows one instance of minimal-windows-only search. Start with $+\infty$ as an upper bound on the search and then repeatedly decrease it until the true value is found. It has been formally proven that this code, called AB-SSS*, expands the same leaf nodes as SSS* [17, 20, 21]. Surprisingly, a best-first algorithm can be reformulated as a special case of a depth-first algorithm.

Other convergence schemes are possible. Starting with a lower bound of $-\infty$ and refining it upward yields the DUAL* algorithm (AB-DUAL*) [15, 22]. Another idea is to use the bounds that are returned by Alpha-Beta in a bisection scheme, yielding the C* algorithm [6].

Instead of starting with an extreme initial value, one can use a heuristic value to "guess" at an initial bound, and then converge either upward or downward towards the minimax value, depending on whether the bound was a lower

[1] There is potential for confusion between algorithms such as SSS*, which are principally fixed-depth best-first minimax searches, and a very different, variable-depth, best-first minimax algorithm by Korf and Chickering [14].

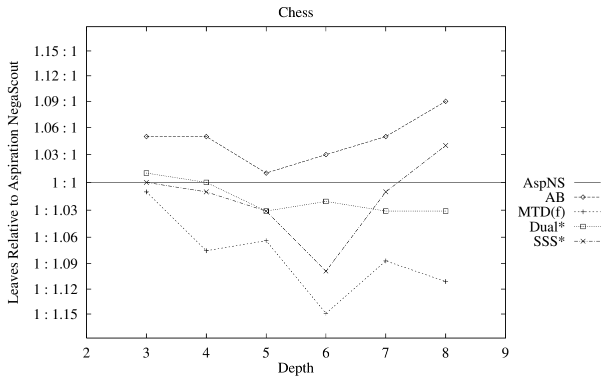

Figure 4: Leaf Node Count Chess

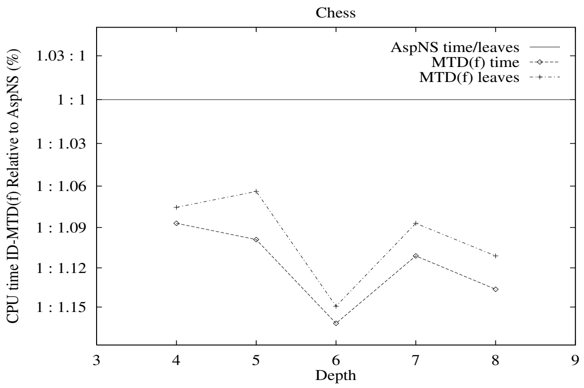

Figure 5: Execution Time Chess

bound or an upper bound, respectively. We call this variant MTD(f) (an explanation for the choice of name and the code, a minor modification to figure 3, can be found in [19, 20]). The intuition behind MTD(f) is that starting a sequence of minimal-window Alpha-Beta calls close to the minimax value is cheaper than using a start value of $+\infty$ or $-\infty$, as in SSS* or DUAL*. Empirical tests have established this assumption to be true, for simulations and for a number of games [19, 20, 21, 24]. SSS* and DUAL* consists of many searches that compute uninteresting bounds that are far away from the target value. MTD(f) consists of a few well-placed searches, yielding bounds close to the target. In an iterative deepening setting, a natural choice for the heuristic start value is the value of a previous iteration.

Test Results

Experiments have been conducted for a variety of algorithms using Phoenix. Experiments have also been performed for checkers and Othello with results similar to those reported here [20, 21]. All algorithms use the same base procedures with iterative deepening, transposition tables and the history heuristic. Forward pruning and selective search have been turned off to ensure comparable results. Results were obtained using 20 test positions that were chosen for providing reliable and representative results. The results were cross-checked with different positions and for deeper searches.

Figure 4 shows the number of leaf nodes evaluated by Alpha-Beta, Aspiration NegaScout (NegaScout enhanced with aspiration windows), AB-SSS*, AB-DUAL* and MTD(f) for different search depths. Aspiration NegaScout is the current algorithm of choice by most chess programmers, therefore we have chosen this algorithm as our baseline. Figure 5 shows that MTD(f)'s execution time performance is proportional to the leaf node count.

From these figures we see a number of interesting points. First, the performance of all algorithms is within a range of $\pm$ 15%, contradicting simulation results that claim SSS* can be significantly better than Alpha-Beta [15]. The tests differ because Alpha-Beta-enhancements have improved the performance of all algorithms, and because simulated trees lack essential properties of real trees [20, 21]. Given these results, we find that performance comparisons based on vanilla textbook versions of Alpha-Beta are of no value for the real world.

Second, MTD(f) performs better than all other tested algorithms. The intuition that starting close to the minimax value is efficient has been experimentally justified.

Third, the depth-first Aspiration NegaScout algorithm can outperform the best-first SSS* algorithm. NegaScout uses minimal-window search, like our reformulation of SSS*. Using a non-optimal start value of $+\infty$ leaves room for SSS* to be outperformed.

Fourth, a word of caution. Since the tested algorithms perform quite close together, the relative differences are quite sensitive to variations in input parameters, such as characteristics of test positions. In generalizing these results, one should keep this sensitivity in mind. Using these numbers as absolute predictors for other situations would not do justice to the complexities of real-life game trees. The experimental data is better suited to provide insight on, or guide and verify hypotheses about these complexities.

Other results borne out by experiments are that the memory requirements of all algorithms are perfectly acceptable for typical tournament play, since only a small subset of the visited nodes (the solution tree) has to be stored in memory. This means that the widely held belief that SSS* uses inordinate amounts of memory is not correct [19].

SSS*: A Footnote in the Game-Tree Search Literature?

For many years, SSS* has cast doubt on the effectiveness of depth-first minimax strategies, because a number of publications showed that best-first strategies had the potential to be better. We show that best-first can be reformulated as depth-first plus memory. This reformulation led us to the following conclusions, dispelling a number of myths:

- The A*-like OPEN list-based formulation of SSS* is unclear and inefficient. The AB-SSS* reformulation using a recursive depth-first procedure and a transposition table shows more clearly how the algorithm traverses its trees, and is easily and efficiently implemented.

- SSS* is not "better" than Alpha-Beta (contradicting, for example, [23, 28]). It is a special case of Alpha-Beta. Other variants of Alpha-Beta outperform AB-SSS*. Thus, the claim should be the other way around:

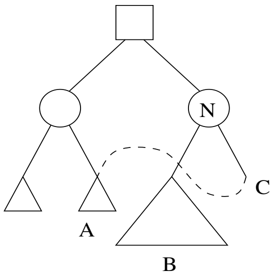

Figure 6: Enhanced Transposition Cutoff

Alpha-Beta-based algorithms are better than SSS*, both in clarity and performance.

- Best-first algorithms such as MTD(f), DUAL* and SSS* do not need too much memory in practice.
- There are many application-independent enhancements to Alpha-Beta. Ignoring them in simulated performance assessments leads to incorrect results.
- The boundary between best-first and depth-first algorithms in minimax search is fuzzy. If best-first is outperformed by depth-first, and if best-first can be reformulated as a special case of depth-first, perhaps we should look for a different criterion to classify search strategies. Interestingly, the literature on single-agent search shows this convergence of depth-first and best-first too, IDA* [13] being the best known example.
- MTD(f) performs better than the current algorithm of choice by chess programmers, and is just as easy (or hard) to implement.

In light of this, we believe that SSS* should now become a footnote in the history of game-tree search.

3.2 Effective Use of Transpositions

Transposition tables are one of the Alpha-Beta enhancements. Normally, transpositions are checked at each visit to a node. If no transposition table cutoff occurs, then the best move suggested by the table is expanded depth-first, before its brothers are considered. 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 (figure 6). 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 and can potentially produce an immediate cutoff. Before doing any search at an interior node, a quick check of all the positions arising from this node (nodes B and C) in the transposition table may result in finding a cutoff. We call this technique *Enhanced Transposition Cutoffs*, ETC. It performs transposition table lookups on all successors of a node, looking for transpositions into previously searched lines. In a left-to-right search, ETC encourages subtrees in the right part of the tree to transpose into the left.

Figure 7 shows the results of enhancing Phoenix with ETC. For search depth 8, ETC lowered the number of expanded

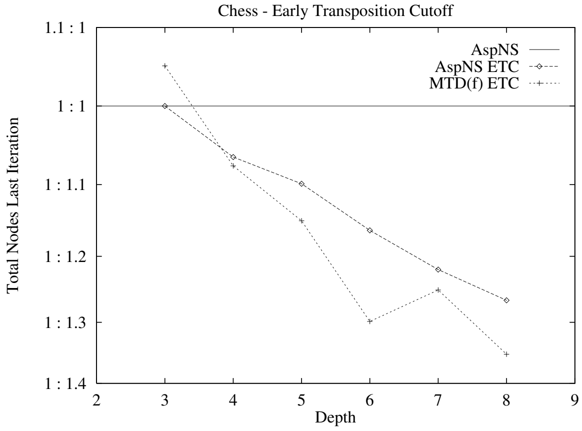

Figure 7: Effectiveness of ETC in Phoenix

total nodes by a factor of 1.28 for NegaScout enhanced with aspiration searching. The combination of MTD(f) and ETC yields a factor of 1.35 fewer total nodes as compared to Phoenix's original algorithm.

The reduction in search tree size offered by ETC is, in part, offset by the increased computation per node. For chess, 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. Thus, ETC is a practical enhancement to most Alpha-Beta search programs.

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

3.3 Failing Low at the Root

Aspiration searching anticipates where the value of a search lies, and selects a small search window that encompasses those expectations. If the search returns a result within the aspiration window, then the expectations have been realized. Exceeding the search window is usually not a problem; the search is more favorable than anticipated. However, under the real-time constraints of a tournament game, a first move that returns a result below the window (failing low) can cause serious problems. Typically, most Alpha-Beta implementations re-search the move to find its true score and then hunt for better moves. Until the right move is found, there is a danger that (time) resources will run out and the program will be forced to move before resolving the difficulty. The program needs to find an alternative best move quickly.

The solution is to restart the search. If at depth i the search fails low, restart the search back at depth 1. Information about the previous search is contained in the transposition table. When the search is restarted, the best move has a bad score allowing other moves to move ahead of it in the ordered move list. Typically, the second best move now becomes best and stays there until depth i is reached again. By restarting the search, an alternative best move is quickly generated, alleviating the problems of the real-time constraint.

This idea has been successful in the checkers program Chinook [26]. To test its effectiveness in chess, the games played by Phoenix in the recent World Computer Chess Championship (Hong Kong, 1995) were scrutinized. Three nontrivial fail low scenarios occurred in the five games. Phoenix was modified so that we could see the impact of the restart mechanism on these three positions. Note that in the following results, for compatibility with the results generated in the World Championship, Phoenix is using search extensions.

1. The fail low occurred after a search of 3.1 million nodes (8 ply). Phoenix was unable to find the correct move until 10.7 million nodes had been examined. Using restart, the program changed its mind several times before finally finding the right move at 8.4 million nodes.
2. The fail low occurred at 637,000 nodes (7 ply). Without restart, the correct move was found at 838,000 nodes. With restart, it is found at 638,000 (3 ply) and deeper search only confirms the move choice.
3. A fail low occurs at 4 ply and then again at 8 ply (1,200,000 nodes). The right move is found at 12,300,000 nodes. With restart, the failure at 4 ply restarts the search. This results in different move ordering and different search extensions. The correct move is found after only 203,000 nodes!

The above examples are necessarily anecdotal. Finding interesting fail low examples is difficult; there are no test suites of fail low positions readily available. However, these results are consistent with experience gathered from the Chinook program, indicating that the restart may be a significant improvement in the search. Not only does it quickly find better moves in critical positions, our experience is that in the presence of search extensions the search results are also more accurate.

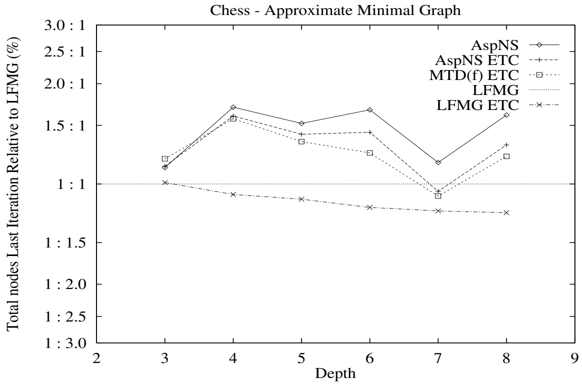

Figure 8: LFMG Is Not Minimal, Phoenix

4 Conclusions & Future Research

Even after 30 years of research, the Alpha-Beta algorithm continues to surprise. Despite many inventive alternatives, none appears poised to supplant Alpha-Beta as the algorithm of choice by practitioners. The exponential growth of the tree with the depth of search hasn't been an obstacle to achieving high performance in popular games such as chess, checkers and Othello.

This paper presents experimental results showing that in practice it is possible to build almost minimal Alpha-Beta trees. This is a surprising result, given that an oracle is required to achieve perfection. Further, this high performance is achieved without any explicit domain-specific knowledge. By taking advantage of search space properties (transpositions), minimizing unnecessary information (minimal windows) and using dynamic knowledge gained from the search itself (history heuristic), Alpha-Beta avoids the knowledge acquisition bottleneck which has been an obstacle for many AI applications.

Given the already high level of Alpha-Beta performance, it is surprising that there is still room for significant improvement. MTD(f) and ETC reduce search effort in chess by roughly 35%. Restarting fail low searches also improves the search, but in a way that is difficult to quantify. Figure 8 is a new version of figure 2, updated to reflect the MTD(f) and ETC enhancements. This figure also shows that in practice Alpha-Beta's best-case is not the smallest graph that proves the minimax value. What we have been using as the minimal tree is really the *left-most minimal graph* (LFMG), constructed by a left-to-right traversal of a graph. The *real minimal graph* (RMG) is smaller, requiring that if there is a choice of cutoff move, the one building the smallest tree be selected. Computing the RMG is computationally infeasible for interesting search depths. The figure shows a loose upper bound on the RMG that we have computed [18].

This leads to the obvious question: what other improvements are waiting to be discovered? The minimal graph discussion suggests that one might try to find the cheapest cutoff. Although we have a number of ideas here, none of them has yet translated into something usable in practice.

One area that has yet to be adequately explored is the role of memory. Additional memory can be used to increase the size of the transposition table, but this leads to diminishing returns [19]. If large endgame databases are used (as in checkers), then additional memory can be used for caching expensive disk I/O. Most chess programs fill all of available memory with a transposition table. Given the increased availability of cheap memory, we pose the question: how do you improve Alpha-Beta search when given a gigabyte of RAM?

5 Acknowledgments

Reformulating SSS* and creating MTD(f) was joint research with Wim Pijls and Arie de Bruin. We gratefully acknowledge their cooperation.

Some results of this research have appeared previously in [19].

References

[1] Eric Baum. How a bayesian approaches games like chess. In *Proceedings of the AAAI'93 Fall Symposium*, pages 48–50. American Association for Artificial Intelligence, AAAI Press, October 1993.

[2] Hans J. Berliner and Chris McConnell. B* probability based search. *Artificial Intelligence*, 1996. To appear.

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

[4] Michael Buro. *Techniken für die Bewertung von Spielsituation anhand von Beispielen*. PhD thesis, Universität-Gesamthochschule Paderborn, Germany, September 1994.

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

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

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

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

[9] Rainer Feldmann. *Spielbaumsuche mit massiv parallelen Systemen*. PhD thesis, Universität-Gesamthochschule Paderborn, Germany, May 1993.

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

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

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

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

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

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

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

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

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

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

[20] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. A minimax algorithm better than SSS*. *Artificial Intelligence*, 1995. Accepted for publication.

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

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

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

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

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

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

[27] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. Chinook: The world man-machine checkers champion. *AI Magazine*, 1996. To appear.

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