RESEARCH RE: SEARCH & RE-SEARCH

Aske Plaat

plaat@theory.lcs.mit.edu

RESEARCH
RE: SEARCH &
RE-SEARCH

This book is number 117 of the Tinbergen Institute Research Series. This series is established through cooperation between Thesis Publishers and the Tinbergen Institute. A list of books which already appeared in the series can be found in the back.

RESEARCH
RE: SEARCH &
RE-SEARCH

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR AAN DE
ERASMUS UNIVERSITEIT ROTTERDAM OP GEZAG VAN DE
RECTOR MAGNIFICUS

PROF.DR. P.W.C. AKKERMANS M.A.

EN VOLGENS BESLUIT VAN HET COLLEGE VOOR PROMOTIES.

DE OPENBARE VERDEDIGING ZAL PLAATSVINDEN OP
DONDERDAG 20 JUNI 1996 OM 13.30 UUR

DOOR

ASKE PLAAT
GEBOREN TE ODOORN.

Promotiecommissie

Promotores:	prof.dr. A. de Bruin
	prof.dr. J. Schaeffer
Overige leden:	dr. J.C. Bioch
	prof.dr.ir. R. Dekker
	prof.dr. H.J. van den Herik
	dr. W.H.L.M. Pijls (tevens co-promotor)

RESEARCH RE:
SEARCH & RE-SEARCH

Aske Plaat

ABSTRACT

Search algorithms are often categorized by their node expansion strategy. One option is the depth-first strategy, a simple backtracking strategy that traverses the search space in the order in which successor nodes are generated. An alternative is the best-first strategy, which was designed to make it possible to use domain-specific heuristic information. By exploring promising parts of the search space first, best-first algorithms are usually more efficient than depth-first algorithms.

In programs that play minimax games such as chess and checkers, the efficiency of the search is of crucial importance. Given the success of best-first algorithms in other domains, one would expect them to be used for minimax games too. However, all high-performance game-playing programs are based on a depth-first algorithm.

This study takes a closer look at a depth-first algorithm, Alpha-Beta, and a best-first algorithm, SSS*. The prevailing opinion on these algorithms is that SSS* offers the potential for a more efficient search, but that its complicated formulation and exponential memory requirements render it impractical. The theoretical part of this work shows that there is a surprisingly straightforward link between the two algorithms—for all practical purposes, SSS* is a special case of Alpha-Beta. Subsequent empirical evidence proves the prevailing opinion on SSS* to be wrong: it is not a complicated algorithm, it does not need too much memory, and it is also *not* more efficient than depth-first search.

Over the years, research on Alpha-Beta has yielded many enhancements, such as transposition tables and minimal windows with re-searches, that are responsible for the success of depth-first minimax search. The enhancements have made it possible to use a depth-first procedure to expand nodes in a best-first sequence. Based on these insights, a new algorithm is presented, MTD(f), which out-performs both SSS* and NegaScout, the Alpha-Beta variant of choice by practitioners.

In addition to best-first search, other ways for improvement of minimax search algorithms are investigated. The tree searched in Alpha-Beta's best case is usually considered to be equal to the minimal tree that has to be searched by any algorithm in order to find and prove the minimax value. We show that in practice this assumption is not valid. For non-uniform trees, the real minimal tree—or rather, graph—that proves the minimax value is shown to be significantly smaller than Alpha-Beta's best case. Thus, there is more room for improvement of full-width minimax search than is generally assumed.

Preface

I would like to thank my advisors Arie de Bruin, Wim Pijls, and Jonathan Schaeffer, for being just as eager as I to understand minimax trees. We have spent countless hours discussing trees and algorithms, trying to find the hidden essence. Their experience with minimax search, as well as their different backgrounds, were of great value. They know how much this research owes to their willingness to spend so much time discussing ideas, combining theory and practice, trying to understand trees. It has been a great time, for which I thank you three.

This work started in May 1993 in Rotterdam, the Netherlands, with Arie and Wim, at the department of Computer Science of the Erasmus University. I thank my colleagues then at the department of Computer Science for creating an interesting environment to work in. I thank Jan van den Berg for helping me experience how enjoyable research can be, which made me decide to pursue a PhD. It has been good talking to Roel van der Goot on subjects ranging from parsing to politics. Maarten van Steen answered many questions on parallel computing. Harry Trienekens and Gerard Kindervater know much about parallel branch and bound as well as operating systems. Reino de Boer knows LaTeX inside out.

From October 1994 to October 1995 I spent most of my time at the University of Alberta in Edmonton, Canada, with Jonathan Schaeffer. Working together with someone with so much experience and energy, and so many ideas, has been an extraordinary experience. Jonathan, I thank you for giving me the opportunity to do research with you and to learn from you; but most of all, I thank you for good times. In Edmonton many people did their best to make my stay as pleasant as possible. Jonathan and Stephanie Schaeffer made sure that I felt at home, helping me wherever and whenever they could. I thank the people of the department of Computing Science of the University of Alberta for creating a good working environment. I have had many discussions on minimax search with Mark Brockington, author of Keyano, one of the top Othello programs. Yngvi Björnsson and Andreas Junghanns, who started building a new chess program, were always keen to discuss new ideas too. Rob Lake answered my troff questions. I thank all of the members of the Games research group for making both the scheduled and the informal meetings interesting and enjoyable.

A number of people took the time to comment on or discuss various parts of this research. I thank Victor Allis, Don Beal, Hans Berliner, Murray Campbell, Rainer Feldmann, Jaap van den Herik, Hermann Kaindl, Laveen Kanal, Rich Korf, Patrick van der Laag, Tony Marsland, Chris McConnell, Alexander Reinefeld, and especially George Stockman, the inventor of SSS*, for their helpful comments.

I thank Victor Allis for teaching a class at the Free University in Amsterdam, in September 1993, in which he explained the concept of the minimal tree in a way that (to me) was strangely reminiscent to solution trees, planting a seed of the SSS*–Alpha-Beta relation.

In addition, research meetings with Victor Allis, Henri Bal, Dennis Breuker, Arie de Bruin, Dick Grune, Jaap van den Herik, Gerard Kindervater, Wim Pijls, John Romein, Harry Trienekens, and Jos Uiterwijk on search, combinatorial optimization and parallelism provided a valuable broader perspective for this work.

I thank Mark Brockington, Arie de Bruin, Jaap van den Herik, Andreas Junghanns, Wim Pijls, Jonathan Schaeffer, and Manuela Schöne for their careful and detailed reading of drafts of this thesis. Their many comments and suggestions have greatly enhanced its quality. Furthermore, Peter van Beek and Rommert Dekker made valuable suggestions for improvement of parts of this work. Thanks to Gerard van Ewijk for reporting an especially nasty bug.

Through all of this, Saskia Kaaks has been a great friend. I thank you for everything, with love.

Rotterdam,
March 31, 1996.

Contents

List of Figures

Chapter 1

Introduction

1.1 Games

Game playing is one of the classic problems of artificial intelligence. The idea of creating a machine that can beat humans at chess has a continued fascination for many people. Ever since computers were built, people have tried to create game-playing programs.

Over the years researchers have put much effort into improving the quality of play. They have been quite successful. Already, non-trivial games like nine-men's-morris, connect-four and qubic are solved; they have been searched to the end and their game-theoretic value has been determined [2, 50]. In backgammon the BKG program received some fame in 1979 for defeating the World Champion in a short match [14, 15]. Today, computer programs are among the strongest backgammon players [142]. In Othello computer programs are widely considered to be superior to the strongest human players [77], the best program of the moment being Logistello [30]. In checkers the current World Champion is the program Chinook [131, 132]. In chess, computer programs play better than all but the strongest human players and have occasionally beaten players such as the World Champion in isolated games [48, 49, 144]. Indeed, many believe that it is only a matter of time until a computer defeats the human world champion in a match [53, 56, 144].

For other games, such as bridge, go, shogi, and chinese chess, humans outplay existing computer programs by a wide margin. Current research has not been able to show that the successes can be repeated for these games. An overview of some of this research can be found in a study by Allis [2].

Computer game playing has a natural appeal to both amateurs and professionals, both of an intellectual and a competitive nature. Seen from a scientific perspective, games provide a conveniently closed domain, that reflects many real-world properties, making it well-suited for experiments with new ideas in problem solving. According to Fraenkel [46], applications of the study of two-player games include areas within mathematics, computer science and economics such as game theory, combinatorial games, complexity theory, combinatorial optimization, logic programming, theorem proving, constraint satisfaction, parsing, pattern recognition, connectionist networks and parallel computing. See for example the bibliographies by Stewart, Liaw and White [138], by Fraenkel [46] and Levy's computer chess compendium [79]. With this long list of applications one would expect results in game playing to have a profound influence on artificial intelligence and operations research. For a number of reasons, not all scientific, this is not the case [39]. Ideas generated in game playing find their way into main stream research at a slow pace.

1.2 Minimax Search

Of central importance in most game-playing programs is the search algorithm. In trying to find the move to make, a human player would typically try to look ahead a few moves, predicting the replies of the opponent to each move (and the responses to these replies, and so on) and select the move that looks most promising. In other words, the space of possible moves is searched trying to find the best line of play. Game-playing programs mimic this behavior. They search each line of play to a certain depth and evaluate the position. Assuming that both players choose the move with the highest probability of winning for them, in each position the value of the best move is returned to the parent position. Player A tries to maximize the chance of winning the game; player B tries to maximize B's chance, which is equivalent to minimizing A's chances. Therefore, the process of backing up the value of the best move for alternating sides is called minimaxing; two-player search algorithms are said to perform a *minimax search*.

Searching deeper generally improves the quality of the decision [54, 133, 143]. Quite a number of researchers have studied minimax search algorithms to improve their efficiency, effectively allowing them to search more deeply within a given real-time constraint. In game playing a move typically has to be made every few minutes. For many games this constraint is too tight to allow optimizing strategies—a look-ahead search to the end of the game is infeasible. Minimax search is a typical example of satisficing, heuristic search. Game-playing programs are real-time systems where the utility of actions is strongly time-dependent. The recent interest in anytime algorithms acknowledges the practical importance of this class of decision making agents [37, 148].

Real-time search is central to many areas of computer science, so it is fortunate that a number of the ideas created in minimax search have proved useful in other search domains. For example, iterative deepening (IDA*) [67], transposition tables [62, 117], pattern databases [36], real time search (RTA*) [68] and bi-directional search (BIDA*) [76, 80] have been applied in both two-agent and single-agent search. These are all examples of a successful technology transfer from game playing to other domains, supporting the view of games as a fertile environment for new ideas. However, the length of Fraenkel's [46] list of applications of game playing suggests that there ought to be more examples.

1.2.1 A Tale of Two Questions

The main theme of the research behind this thesis has been to find ways to improve the performance of algorithms that search for the minimax value of a (depth-limited) tree in real time, through a better understanding of the search trees that these algorithms generate. This goal is pursued along two lines: by looking into the best-first/depth-first issue, and by examining the concept of the minimal tree.

Best-First and Depth-First

Most successful game-playing programs are based on the Alpha-Beta algorithm, a simple recursive depth-first minimax procedure invented in the late 1950's. In its basic form a depth-first algorithm traverses nodes using a rigid left-to-right expansion sequence [99]. There is an exponential gap in the size of trees built by best-case and worst-case Alpha-Beta. This led to numerous enhancements to the basic algorithm, including iterative deepening, transposition tables, the history heuristic, and narrow search windows (see for example [128] for an assessment). These enhancements have improved the performance of depth-first minimax search considerably.

An alternative to depth-first search is best-first search. This expansion strategy makes use of extra heuristic information, which is used to select nodes that appear more promising first, increasing the likelihood of reaching the goal sooner. Although best-first approaches have been successful in other search domains, minimax search in practice has been almost exclusively based on depth-first strategies. (One could argue that the enhancements to Alpha-Beta have transformed it into best-first search. However, the designation "best-first" is normally reserved for algorithms that deviate from Alpha-Beta's left-to-right strategy.)

In 1979 SSS* was introduced, a best-first algorithm for searching AND/OR trees [140]. For minimax trees, SSS* was proved never to build larger trees than Alpha-Beta, and simulations showed that it had the potential to build significantly smaller trees. Since its introduction, many researchers have analyzed SSS*. Understanding this complicated algorithm turned out to be challenging. The premier artificial intelligence journal, *Artificial Intelligence*, has published six articles in which SSS* plays a major role [33, 57, 72, 85, 118, 121], in addition to Stockman's original publication [140], and our own forthcoming paper [113]. Most authors conclude that SSS* does indeed evaluate less nodes, but that its complex formulation and exponential memory requirements are serious problems for practical use. Despite the potential, the algorithm remains largely ignored in practice. However, the fact that it searches smaller trees casts doubts on the effectiveness of Alpha-Beta-based approaches, stimulating the search for alternatives.

This brings us to the first theme of our research: the relation between SSS* and Alpha-Beta, between best-first and depth-first. Simply put, in this work we try to find out which is best.

The Minimal Tree

The second issue concerns the notion of the minimal tree. In a seminal paper in 1975 Knuth and Moore proved that to find the minimax value, any algorithm has to search at least the *minimal tree* [65]. The concept of the minimal tree has had a profound impact on our understanding of minimax algorithms and the structure of their search trees.

The minimal tree can be used as a standard for algorithm performance. Since any algorithm has to expand at least the minimal tree, it is a limit on the performance of all minimax algorithms. Many authors of game-playing programs use it as a standard benchmark for the quality of their algorithms. The size of the minimal tree is defined for uniform trees. For use with real minimax trees, with irregular branching factor and depth, the minimal tree is usually defined as Alpha-Beta's best case. However, this is not necessarily the smallest possible tree that defines the minimax value. Due to transpositions and the irregularity of the branching factor it might be significantly smaller.

The second theme of this research is to find out by how much. A minimal tree that is significantly smaller would have two consequences. First, it would mean that the relative performance of game-playing programs has dropped. Second, and more important, it would show where possible improvements of minimax algorithms could be found.

1.3 Contributions

1.3.1 Best-First and Depth-First

The first main result of our research is that SSS* can be reformulated as a special case of Alpha-Beta. Given that best-first SSS* is normally portrayed as an entirely different approach to minimax search than depth-first Alpha-Beta, this is a surprising result. SSS* is now easily implemented in existing Alpha-Beta-based game-playing programs. The reformulation solves all of the perceived drawbacks of the SSS* algorithm. Experiments conducted with three tournament-quality game-playing programs show that in practice SSS* requires as much memory as Alpha-Beta. When given identical memory resources, SSS* does not evaluate significantly less nodes than Alpha-Beta. It is typically out-performed by NegaScout [44, 99, 119], the current depth-first Alpha-Beta variant of choice. In effect, the reasons for ignoring SSS* have been eliminated, but the reasons for using it are gone too.

The ideas at the basis of the SSS* reformulation are generalized to create a framework for best-first fixed-depth minimax search that is based on depth-first null-window Alpha-Beta calls. The framework is called MT, for Memory-enhanced Test (see page 38). A number of existing algorithms, including SSS*, DUAL* [73, 85, 116] and C* [35], are special cases of this framework. In addition to reformulations, we introduce new instances of this framework. One of the instances, called MTD(f), out-performs all other minimax search algorithms that we tested, both on tree size and on execution time.

In the new formulation, SSS* is equivalent to a special case of Alpha-Beta; tests show that it is out-performed by other Alpha-Beta variants. In light of this, we believe that SSS* should from now on be regarded as a footnote in the history of game-tree search.

The results contradict the prevailing view in the literature on Alpha-Beta and SSS*. How can it be that the conclusions in the literature are so different from what we see in practice? Probably due to the complex original formulation of SSS*, previous work mainly used theoretical analyses and simulations to predict the performance of SSS*. However, there are many differences between simulated algorithms and trees, and those found in practice. Algorithmic enhancements have improved performance considerably. Artificial trees lack essential properties of trees as they are searched by actual game-playing programs. Given the fact that there are numerous high-quality game-playing programs available, there is no valid reason to use simulations for performance assessments of minimax algorithms. Their unreliability can lead to conclusions that are the opposite of what is seen in practice.

1.3.2 The Minimal Tree

Concerning the second issue, we have found that in practice the minimal tree can indeed be improved upon. The irregular branching factor and transpositions mean that the *real* minimal tree (which should really be called minimal *graph*) is significantly smaller. Our approximations show the difference to be at least a factor of 1.25 for chess to 2 for checkers. This means that there is more room for improvement of minimax algorithms than is generally assumed [40, 41, 125].

We present one such improvement, called Enhanced Transposition Cutoff (ETC), a simple way to make better use of transpositions that are found during a search.

1.3.3 List of Contributions

The contributions of this research can be summarized as follows:

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

- *A framework for best-first minimax search based on depth-first search*
  Inspired by the MT-SSS* reformulation, a new framework for minimax search is introduced. It is based on the procedure MT, which is a memory-enhanced version of Pearl's Test procedure [99], also known as null-window Alpha-Beta search. We present a simple framework of MT drivers (MTD) that make repeated

calls to MT to home in on the minimax value. Search results from previous passes are stored in memory and re-used. MTD can be used to construct a variety of best-first search algorithms using depth-first search. Since MT can be implemented using Alpha-Beta with transposition tables, the instances of this framework are readily incorporated into existing game-playing programs.

- *Depth-first search can out-perform best-first*
  Using our new framework, we were able to compare the performance of a number of best-first algorithms to some well-known depth-first algorithms. Three high performance game-playing programs were used to ensure the generality and reliability of the outcome. The results of these experiments were quite surprising, since they contradict the large body of published results based on simulations. Best-first searches and depth-first searches are roughly comparable in performance, with NegaScout, a depth-first algorithm, out-performing SSS*, a best-first algorithm.

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

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

- *Memory size*
  In the MT framework the essential part of the search tree is formed by a max and/or a min solution tree of size $O(w^{d/2})$ for trees with branching factor w and depth d. Our experiments show that for game-playing programs under tournament conditions, these trees fit in memory without problems. The reason that the exponential space complexity is not a problem under tournament conditions is that the time complexity of a search is at least $O(w^{d/2})$; time runs out before the memory is exhausted.

- *Domination and dynamic move re-ordering*
  With dynamic move reordering schemes like iterative deepening, SSS* and its dual DUAL* [72, 116] are no longer guaranteed to expand fewer leaf nodes than Alpha-Beta. The conditions for Stockman's proof [140] are not met in practice.

- *MTD(f)*
  We formulate a new algorithm, MTD(f). It out-performs our best Alpha-Beta

variant, NegaScout, on leaf nodes, total nodes, and execution time for our test-programs. The improvement was bigger than the improvement of NegaScout over Alpha-Beta. Since MTD(f) is an instance of the MT framework, it is easily implemented in existing programs: just add one loop to an Alpha-Beta-based program.

- *The real minimal graph*
  The minimal tree is a limit on the performance of minimax algorithms. For search spaces with transpositions and irregular w and d, most researchers redefine the minimal "tree" as the minimal graph searched by Alpha-Beta. However, this is not the smallest graph that proves the minimax value. Because of transpositions and variances in w and d, cutoffs may exist that are cheaper to compute. The size of the real minimal graph is shown to be significantly smaller. This implies that algorithms are not as close to the optimum, leaving more room for improvement than is generally assumed.

- *ETC*
  We introduce a technique to take better advantage of available transpositions. The technique is called Enhanced Transposition Cutoff (ETC). It reduces the search tree size for checkers and chess between 20%–30%, while incuring only a small computational overhead.

- *Solution trees*
  Theoretical abstractions of actual search trees have been useful in building models to reason about the behavior of various minimax algorithms, explaining experimental evidence and facilitating the construction of a new framework. The key abstractions have been the dual notions of the max and min solution tree, defining an upper and a lower bound on the minimax value of a tree.

One of the most striking results, besides the simulation versus practice issue, is perhaps that best-first algorithms can be expressed in a depth-first framework. (In contrast to the IDA*/A* case [67], the algorithms in this work are reformulations of the best-first originals. They evaluate exactly the same nodes and the space complexity does not change.) Furthermore, it is surprising that despite the exponential storage requirements of best-first algorithms their memory consumption does not render them impractical.

1.4 Overview

This section gives a short overview of the rest of the thesis. Chapter 2 provides some background on minimax algorithms. Alpha-Beta and SSS* are discussed. The chapter includes an explanation of bounds, solution trees, the minimal tree, and narrow-window calls, notions that form the basis of the next chapters.

Chapter 3 introduces MT, a framework for best-first full-width minimax search, based on null-window Alpha-Beta search enhanced with memory. The main benefit of the framework is its simplicity, which makes it possible to use a number of real applications to test the instances.

Chapter 4 reports on results of tests with Alpha-Beta, NegaScout, and MT instances. We have tested both performance and storage requirements of the algorithms, using three tournament game-playing programs. As stated, the test results paint an entirely different picture of Alpha-Beta, NegaScout, and SSS* than the literature does. Section 4.4 discusses the reason: differences between real and artificial search trees.

Chapter 5 discusses the minimal tree in the light of the test results. This chapter discusses how the size of a more realistic minimal tree can be computed.

Chapter 6 presents the conclusions of this work.

In appendix A we give examples of how Alpha-Beta, SSS*, and MT-SSS* traverse a tree.

Appendix B is concerned with the equivalence of SSS* and MT-SSS*. It contains a detailed technical treatment indicating why our reformulation is equivalent to the original.

Appendix C lists test positions that were used for the experiments as well as some numerical results of these tests.

The ideas of chapter 3 and 4 have appeared in [111, 113], and also in [108, 109, 110, 112, 134]. The ideas on solution trees in chapter 2 have appeared in [28, 29]. The work on the real minimal graph in chapter 5 has appeared in [114], and also in [107, 134].

Chapter 2

Background—Minimax Search

2.1 Minimax Trees and Alpha-Beta

This chapter provides some background on minimax algorithms. We briefly introduce the minimax function, and the concept of a cutoff. To find the value of the minimax function, one does not have to search the entire problem space. Some parts can be pruned; they are said to be *cut off*. The basic algorithm of minimax search, Alpha-Beta, is discussed. Next we discuss common enhancements to Alpha-Beta, as well as alternatives, notably SSS*.

Other general introductions into this matter can be found in, for example, [47, 93, 99].

2.1.1 Minimax Games

This research is concerned with (satisficing) search algorithms for minimax games. In a zero-sum game the loss of one player is the gain of the other. Seen from the viewpoint of a single player, one player tries to make moves that maximize the likelihood of an outcome that is positive for this person, while the other tries to minimize this likelihood. The pattern of alternating turns by the maximizer and minimizer has caused these games to be called minimax games. Likewise, the function describing the outcome of a minimax game is called the minimax function, denoted by f.

As an example, let us look at tic-tac-toe. Figure 2.1 gives the possible moves from a certain position. A graph representation is shown in figure 2.2. This graph is called the minimax tree. For reasons that will become clear soon, player X is called "Max" and player O is called "Min." Board positions are represented by square or circle nodes. Nodes where the Max player moves are shown as squares, nodes where the Min player moves are shown as circles. The possible moves from a position are represented by unlabeled links in the graph. The node at the top represents the actual start position. In the start position, Max has three possible moves, leading to nodes b, c and g. By considering the options of Min in each of these nodes, and the responses by Max, the tree in figure 2.2 is constructed.

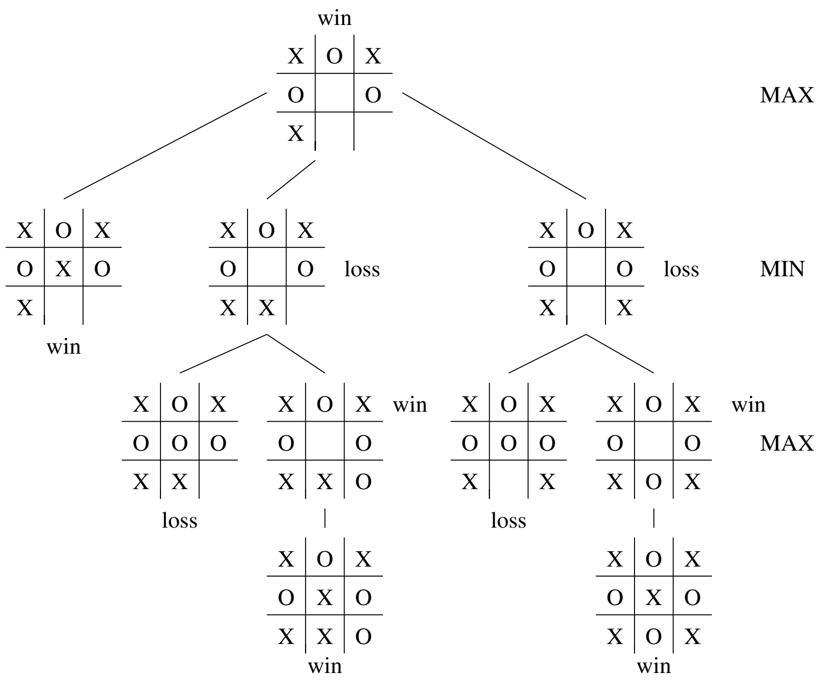


Figure 2.1: Possible Lines of Play for Tic-tac-toe Position

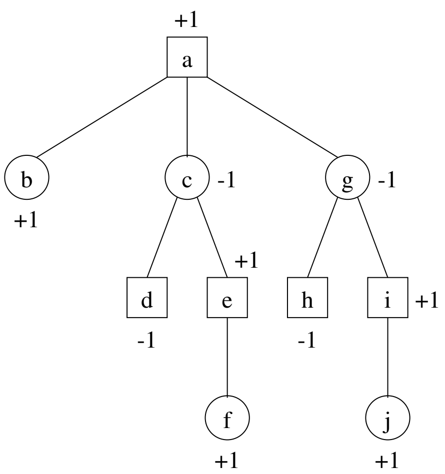


Figure 2.2: Minimax Tree of the Possible Lines of Play

function minimax(n) $\rightarrow f$;
  if n = leaf **then return** eval(n);
  else if n = max **then**
    $g := -\infty$;
    c := firstchild(n);
    while $c \neq \perp$ **do**
      $g := \max\big(g, \text{minimax}(c)\big)$;
      c := nextbrother(c);
  else /* n is a min node */
    $g := +\infty$;
    c := firstchild(n);
    while $c \neq \perp$ **do**
      $g := \min\big(g, \text{minimax}(c)\big)$;
      c := nextbrother(c);
  return g;

Figure 2.3: The Minimax Function

The start position (node a) is called the *root* of the tree. The other nodes represent the possible lines of play. Nodes a, c, e, g and i are called interior nodes. Nodes b, d, f, h and j are the leaf nodes of this tree. Considering the possible lines of play to find the best one is usually referred to as *searching the tree*. A move by one player is often called a half-move or a *ply*.

At the leaf nodes in the tree the game ends, and the value of the minimax function f can be determined. A win for max is denoted by +1, a -1 denotes a loss. The value 0 would represent a draw. The root of the tree is at level 0. At even levels player Max will make a move to maximize the outcome f, while at odd levels player Min will move to minimize f. Therefore, at even levels f equals the maximum of the value of its children and at odd levels f equals the minimum. In this way f is recursively defined for all positions. In the example position its value is +1, Max wins.

Figure 2.3 gives the recursive minimax function in a Pascal-like pseudo code. The code takes a node n as input parameter and returns f_n, the minimax value for node n. The code contains a number of abstractions. First, every node is either a leaf, a min, or a max node. Second, an evaluation function, *eval*, exists that returns the minimax value (win, loss, or draw) for each board position at a leaf node. Third, functions *firstchild* and *nextbrother* exist, returning the child node and brothers. (They return the value $\perp$ if no child or brother exist.) Note that the given minimax function traverses the tree in a depth-first order. The min and max operations implement the backing-up of the scores of nodes from a deeper level. The value of g represents an intermediate value of a node. When all children have been searched g becomes f, the final minimax value.

Strictly speaking, a game-playing program does not need to know the minimax value of the root. All it needs is the best move. By searching for the minimax value, the best move is found too. Later on (in section 3.2) we will discuss algorithms that stop searching as soon as the best move is known, giving a more efficient search.

In tic-tac-toe, the tree is small enough to traverse all possible paths to the end of the game within a reasonable amount of time. The evaluation function is simple, all it has to do is determine whether an end position is a win, loss, or draw. In many games it is not feasible to search all paths to the end of the game, because the complete minimax tree would be too big. For these games (such as chess, checkers and Othello) a different approach is taken. The evaluation function is changed to return a heuristic assessment. Now it can be called on any position, not just where the game has ended. To facilitate a better discrimination between moves, the range of return values is usually substantially wider than just win, loss, or draw. From the perspective of the search algorithm, the change is minor. The minimax function does not change at all, only the evaluation function is changed (and the definition of a leaf node). In its simplest form, the search of the minimax tree is stopped at a fixed depth from the root and the position is evaluated.

Evaluation functions are necessarily application dependent. In chess they typically include features such as material, mobility, center control, king safety and pawn structure. In Othello the predominant feature is mobility. Positional features are often assessed using patterns. A heuristic evaluation function has to collapse a part of the look-ahead search into a single numeric value—a task that is much harder to do well than recognizing a win, loss, or draw at the end of the game (see section 2.3.3 for a discussion of alternatives to the single numeric back-up value). The reason that the scheme works so well in practice, is that the value of the evaluation of a child node is usually related to that of its parent. If no strong correlation between parent and child values exists, searching deeper using the minimax back-up rule does not work very well. More on this phenomenon, called pathology, can be found in [91, 98].

2.1.2 The Minimal Tree

This section discusses bounds, solution trees and the minimal tree, to show why algorithms like Alpha-Beta can cut off certain parts of the minimax tree.

In increasing the search depth of a minimax tree, the number of nodes grows exponentially. For a minimax tree of uniform search depth d and uniform branching factor w (the number of children of each interior node), the number of leaf nodes is w^d. The exponential rate of growth would limit the search depths in game-playing programs to small numbers. For chess, given a rate of a million positional evaluations per second (which is relatively fast by today's standards), a branching factor of about 35, no transpositions (see chapter 5), and the requirement to compute 40 moves in two hours (the standard tournament rate), it would mean that under tournament conditions the search would be limited to depths of 5 or 6. (Such a machine could evaluate on average 180 million positions for each of the 40 moves, which lies between $35^5 \approx 53$ million and $35^6 \approx 1838$ million.) The search depth would have been limited to 3 or at most 4 in the 1960's and 1970's. A number of studies have shown a strong positive

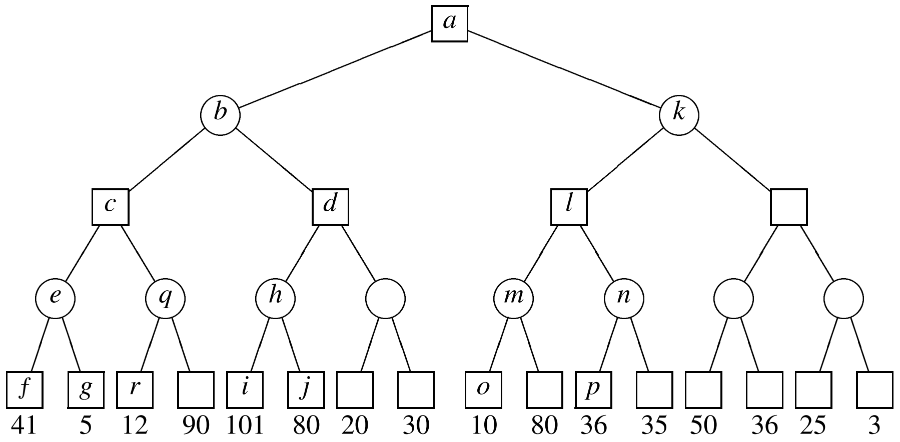


Figure 2.4: Example Tree for Bounds

correlation between deeper search and better play [54, 133, 143]. The studies show that a program that is to play chess at grand-master level has to look significantly further ahead than 5 ply. If it were not for the early introduction of a powerful pruning technique that made this possible, game-playing programs would not have been so successful.

The minimax back-up rule alternates taking the maximum and the minimum of a set of numbers. Suppose we are at the root of the tree in figure 2.1. The first child returns +1. The root is a max node, so its f can only increase by further expanding nodes, $f_{root} \geq +1$. We know that the range of possible values for f_{root} is limited to $\{-1, 0, +1\}$. Thus +1 is an upper bound on f_{root}. Now we have $f_{root} \geq +1$ and $f_{root} \leq +1$, or $f_{root} = +1$. The interpretation is that further search can never change the minimax value of the root. The first win returned is as good as any other win. In figure 2.1 only one child of the root has to be expanded to determine the minimax value. The rest can be eliminated, or carrying the tree analogy further, pruned.

In the more common situation where the range of values for f is much wider, say $[-1000, +1000]$, the probability of one of the moves returning exactly a win of +1000 is small. However, the fact that the output value of the maximum function never decreases can still be exploited. In the pseudo code of the minimax function in figure 2.3, the variable g is equal to the value of the highest child seen so far (for a max node). As return values of child nodes come in, g is never decreased. At any point in time g represents a lower bound on the return value of the max node, $f_{max} \geq g_{max}$. If no child has yet returned, it is $-\infty$, a trivial lower bound. Likewise, at min nodes g is an upper bound, $f_{min} \leq g_{min}$.

Up to now we have looked at single nodes. Things get more interesting if we try to extend the concept of bounds to the rest of the minimax tree. For this we will use a bigger example tree, as shown in figure 2.4. To show a sufficiently interesting depth, the branching factor (or width) of the nodes has been restricted to 2. (The tree is based on one in [99]. It is also used in the Alpha-Beta and SSS* examples in the appendices.

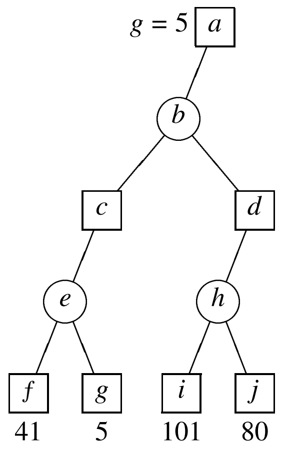


Figure 2.5: A Tree for a Lower Bound, a Min Solution Tree

To illustrate the concept of a deep cutoff in Alpha-Beta, the value of node o has been changed to 10.)

As we saw in the tic-tac-toe case, the search of a node n can be stopped as soon as an upper and lower bound of equal value for n exist (where node n is any node in a minimax tree, not just the one with the same name in figure 2.4). Therefore we will examine what part of the minimax tree has to be searched by the minimax function to find such bounds. Suppose we wish to find a better lower bound (other than $-\infty$) for the root, node a. This amounts to finding non-trivial lower bounds ($f \geq g > -\infty$) for all nodes in a sub-tree below a. At node a, a max node, all we need is to have *one* child return a lower bound $> -\infty$, to have the variable g_a change its value to $> -\infty$. Subsequent children cannot decrease the value of g, so we will then have that $f_a \geq g_a > -\infty$. We turn to finding a lower bound $> -\infty$ for this first child of node a. Node b is a min node. We are not sure that g_b is a lower bound unless all children have been expanded. So, the values of the children c and d have to be determined. These are max nodes, and one child suffices to have their return value conform to $f \geq g > -\infty$. Thus, the value of nodes e and h has to be determined. These are min nodes, so the value of all children f, g and i, j is needed. These are leaf nodes, for which the relation $f \geq g > -\infty$ always holds. We have come to the end of our recursive search for a lower bound. The sub-tree that we had to expand below node a is shown in figure 2.5.

If we apply the minimax back-up rule to figure 2.5, we find the value of the lower bound: 5. This is the lowest of the leaves of this sub tree. The tree has the special property that only one child of each max node is part of it, which renders the max-part of the minimax rule redundant. For this reason these trees are called *min solution trees*, denoted by T^-, their minimax value, denotred by f^-, is the *minimum* of their leaves. (Min solution trees originated from the study of AND/OR trees, where they were called simply solution trees. The term *solution* refers to the fact that they represent the solution to a problem [72, 93, 100, 140].)

An intuitive interpretation of min solution trees in the area of game playing is that

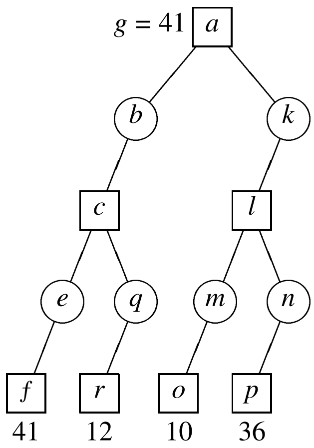


Figure 2.6: A Tree for an Upper Bound, a Max Solution Tree

of a *strategy*. A min solution tree represents all responses from the Min player to one of the moves by the Max player. It is like reasoning: "If I do this move, then Min has all those options, to which my response will be ..." A confusing artefact of this terminology is that a *min* solution tree is a strategy for the *Max* player.

A min solution tree at node n determines the value of a lower bound on f_n. The sub-tree that determines an *upper* bound (f^+) is called, not surprisingly, a max solution tree (T^+). It consists of all children at max nodes, and one at min nodes. An example is shown in figure 2.6. Its value is the highest of the leaves of the max solution tree: 41. The interpretation of a max solution tree is that of a strategy for the Min player.

Now we have the tools to construct upper and lower bounds at the root, and thus to find (prove) the minimax value. This can be done cheaper than by traversing the entire minimax tree, since solution trees are much smaller. In a solution tree, at half of the nodes only one child is traversed. Therefore, the exponent determining the number of leaf nodes is halved. To find an upper bound we have to examine only $w^{\lceil d/2 \rceil}$ leaf nodes. Since the root is a max node, for odd search depths lower bounds are even cheaper, requiring $w^{\lfloor d/2 \rfloor}$ leaves to be searched. With this, we can determine the best case of any algorithm that wants to prove the minimax value, for minimax trees of uniform w and uniform d. If the minimax tree is ordered so that the first child of a max node is its highest and the first child of a min node is its lowest, then the first max solution tree and the first min solution tree that are traversed prove the minimax value of the root. We note that these two solution trees overlap in one leaf, leaf f in figure 2.5 and 2.6 (in the figures the upper bound is not equal to the lower bound, so although these are the left-most solution trees, they do not prove the minimax value). Thus, the number of leaves of the minimal tree that proves the value of f is $w^{\lfloor d/2 \rfloor} + w^{\lceil d/2 \rceil} - 1$. This is a big improvement over the number of leaves of the minimax tree w^d. It means that with pruning, programs can search up to twice the search depth of full minimax.

The tree that is actually traversed by an algorithm is called the *search tree*. With pruning, the search tree has become a sub-set of the minimax tree.

This concept of a minimal tree (or critical tree, or proof tree) was introduced by Knuth and Moore [65]. They introduced it as the best case of the Alpha-Beta algorithm, using a categorization of three different types of nodes, not in terms of solution trees (see figure 2.12 for a minimal tree with node types). The treatment of the minimal tree in terms of bounds and solution trees, is based on [28, 29, 106], analogous to the use of strategies by Pearl [99, p. 222–226] and by Nilsson [93, p. 110]. (Regrettably, they use T^+ to denote a max strategy, which we refer to as a min solution tree, denoted by T^-.) Other works on solution trees are [57, 73, 100, 102, 116, 140].

2.1.3 Alpha-Beta

Pruning can yield sizable improvements, potentially reducing the complexity of finding the minimax value to the square root. The Alpha-Beta algorithm enhances the minimax function with pruning. Alpha-Beta has been in use by the computer-game-playing community since the end of the 1950's. It seems to have been conceived of independently by several people. The first publication describing a form of pruning is by Newell, Shaw and Simon [92]. John McCarthy is said to have had the original idea, and also to have coined the term Alpha-Beta [65]. However, according to Knuth and Moore [65], Samuel has stated that the idea was already present in his checker-playing programs of the late 1950's [123], but he did not mention this because he considered other aspects of his program to be more significant. The first accounts of the full algorithm in Western literature appear at the end of the 1960's, by Slagle and Dixon [136], and by Samuel [124]. However, Brudno already described an algorithm identical to Alpha-Beta in Russian in 1963 [27]. A comprehensive analysis of the algorithm, introducing the concept of the minimal tree, has been published by Knuth and Moore in 1975 [65]. This classical work also contains a brief historic account, which has been summarized here. Other works analyzing Alpha-Beta are [8, 29, 97, 105, 106]. In the following we pursue a description in intuitive terms.

Figure 2.7 gives the pseudo code for Alpha-Beta. It consists of the minimax function, plus two extra input parameters and cutoff tests. The α and β parameters together are called the *search window*. At max nodes, g is a lower bound on the return value. This lower bound is passed to children as the α parameter. Whenever any of these children finds it can no longer return a value above that lower bound, further searching is useless and is stopped. This can happen in children of type min, since there g is an upper bound on the return value. Therefore, figure 2.7 contains for min nodes the line "**while** $g > \alpha$." At min nodes g is an upper bound. Parameter β passes the bound on so that any max children with a lower bound $\geq \beta$ can stop searching when necessary. Together, α and β form a search window which can be regarded as a task for a node to return a value that lies inside the window. If a node finds that its return value is proven to lie not within the search window, the search is stopped. This is illustrated in figure 2.8. Assume that Alpha-Beta is called with an initial window of $\langle -\infty, +\infty \rangle$. Node e is searched with a window of $\langle 12, +\infty \rangle$. After node f returns, the return value of node e is ≤ 10, which lies outside the search window, so the search is stopped before

function alphabeta$(n, \alpha, \beta) \to g$;
    if n = leaf **then return** eval(n);
    else if n = max **then**
        $g := -\infty$;
        $c :=$ firstchild(n);
        while $g < \beta$ **and** $c \neq \perp$ **do**
            $g := \max\big(g, \text{alphabeta}(c, \alpha, \beta)\big)$;
            $\alpha := \max(\alpha, g)$;
            $c :=$ nextbrother(c);
    else /* n is a min node */
        $g := +\infty$;
        $c :=$ firstchild(n);
        while $g > \alpha$ **and** $c \neq \perp$ **do**
            $g := \min\big(g, \text{alphabeta}(c, \alpha, \beta)\big)$;
            $\beta := \min(\beta, g)$;
            $c :=$ nextbrother(c);
    return g;

Figure 2.7: The Alpha-Beta Function

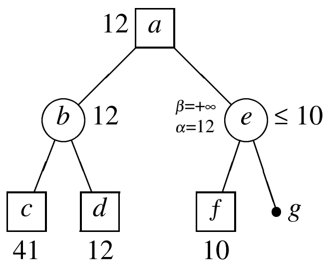


Figure 2.8: Node g is Cut Off

node g is traversed.

As children of a node are expanded, the g-value in that node is a bound. The bound is established because Alpha-Beta has traversed a solution tree that defines its value. As more nodes are expanded, the bounds become tighter, until finally a min and a max solution tree of equal value prove the minimax value of the root. In the following postcondition of Alpha-Beta these solution trees are explicitly present, following [29, 106]. The precondition of a call Alpha-Beta(n, α, β) is $\alpha < \beta$. As before, g denotes the return value of the call, f_n denotes the minimax value of node n, f_n^+ denotes the minimax value of a max solution tree T_n^+, which is an upper bound on f_n, and f_n^- denotes the minimax value of a min solution tree T_n^-, which is a lower bound on f_n. The postcondition has three cases [29, 65, 106]:

1. $\alpha < g_n < \beta$ (success). g_n is equal to f_n. Alpha-Beta has traversed at least a T_n^+ and a T_n^- with $f(T_n^+) = f(T_n^-) = f_n$.

2. $g_n \leq \alpha$ (failing low). g_n is an upper bound f_n^+, or $f_n \leq g_n$. Alpha-Beta has traversed at least a T_n^+ with $f(T_n^+) = f_n^+$.

3. $g \geq \beta$ (failing high). g_n is a lower bound f_n^-, or $f_n \geq g_n$. Alpha-Beta has traversed at least a T_n^- with $f(T_n^-) = f_n^-$.

If the return value of a node lies in the search window, then its minimax value has been found. Otherwise the return value represents a bound on it. From these cases we can infer that in order to be sure to find the game value Alpha-Beta must be called as Alpha-Beta($n, -\infty, +\infty$). (Older versions of Alpha-Beta returned α or β at a fail low or fail high. The version returning a bound is called *fail-soft* Alpha-Beta in some publications [33, 45, 116], because a fail high or fail low still returns useful information. We use the term Alpha-Beta to denote the fail-soft version. Furthermore, implementations of Alpha-Beta generally use the negamax formulation since it is more compact [65]. For reasons of clarity, we use the minimax view.)

Appendix A.1 contains a detailed example of how a tree is searched by Alpha-Beta.

2.1.4 Plotting Algorithm Performance

The purpose of this thesis is to find better minimax search algorithms. In comparing algorithms, we will use a grid that shows two key performance parameters, allowing us to position an algorithm at a glance.

For game-playing programs the quality of an algorithm is determined (a) by the speed with which it finds the best move for a given position and (b) by the amount of storage it needs in doing that. The speed is largely influenced by how many cutoffs it finds. Often there is a trade-off between speed and storage: more memory gives a faster search. In performance comparisons this trade-off introduces the danger of unfair biases. Luckily, for the algorithms that are studied in this work, this relation stabilizes at some point to the extent that adding more memory does not improve performance

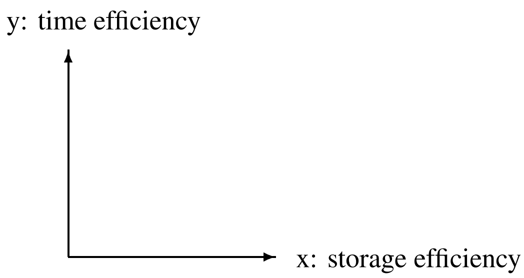


Figure 2.9: Two Performance Dimensions

measurably. To avoid this bias, all algorithms were tested with the same amount of storage, which was known to be large enough for all to achieve stability.

Execution time is generally considered to be a difficult performance metric, since it can vary widely for different programs and hardware. Instead, the size of the tree that is searched by an algorithm is often used in comparisons. Though better than execution time, this metric is still not without problems, since a program usually spends a different amount of time in processing leaf nodes and interior nodes. Many publications only report the number of leaf nodes searched by an algorithm. This introduces a bias towards algorithms that revisit interior nodes frequently. In many programs leaf evaluations are slowest, followed by making and generating moves, which occur at interior nodes. However, in some programs the reverse holds. Compared to the time spent on these actions, transposition nodes are fast (see section 2.2.2).

The algorithms in this thesis are judged on their performance. In the experiments we have counted leaf nodes, interior nodes, and transpositions. The most important parameter is the size of the search tree generated by the algorithm. This parameter will be used as an indicator of time efficiency. By finding more cutoffs, an algorithm performs better on this parameter. A second parameter is the amount of memory an algorithm needs. Though not a limiting factor in their practical applicability, this remains a factor of importance. Thus we have two parameters to judge algorithms on:

1. *Cutoffs*
   The degree to which an algorithm finds useful cutoffs to reduce its node count is influenced by the quality of move ordering. A higher quality of move ordering causes more cutoffs and thus a smaller search tree.

2. *Storage efficiency*
   The amount of storage an algorithm needs to achieve high performance.

Figure 2.9 shows a grid of these two dimensions. It will be used to summarize the behavior of conventional implementations of a number of algorithms (and not more than that; the graphs do not explain behavior, they only picture it). Algorithms should

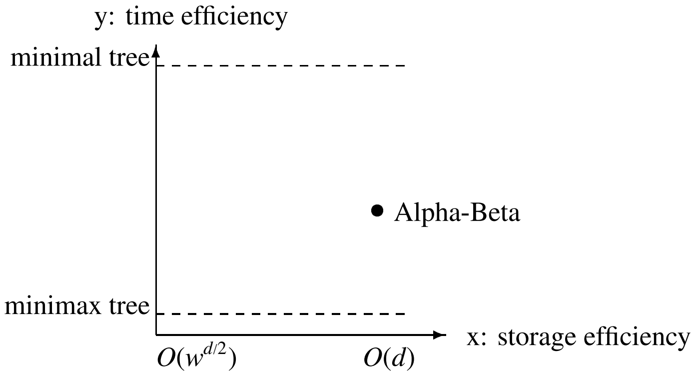


Figure 2.10: Alpha-Beta's Performance Picture

go as far as possible on the x and y axis, to achieve high performance; slow algorithms that use a lot of memory are close to the origin.

2.2 Alpha-Beta Enhancements

The example in appendix A.1 illustrates that Alpha-Beta can miss some cutoffs. Figure 2.10 illustrates that Alpha-Beta's performance picture lies between that of minimax and the minimal tree. Its storage needs are excellent, only $O(d)$, the recursion depth. (The y axis is not drawn to scale.)

The performance of Alpha-Beta depends on the ordering of child nodes in the tree. This brings us to the question of whether there are ways to enhance Alpha-Beta's performance to bring it closer to the theoretical best case, or whether there are alternative ways to implement pruning. The remainder of this section discusses the former, Alpha-Beta enhancements. Section 2.3 will discuss the latter, alternatives to Alpha-Beta pruning.

2.2.1 Smaller Search Windows

Alpha-Beta cuts off a sub-tree when the value of a node falls outside the search window. One idea to increase tree pruning is to search with a smaller search window. Assuming $c \leq a$ and $b \leq d$, a search with (wider) window $\langle c, d \rangle$ will visit at least every single node that the search with (smaller) window $\langle a, b \rangle$ will visit (if both search the same minimax tree). Normally the wider search will visit extra nodes [33, 99]. However, Alpha-Beta already uses all return values of the depth-first search to reduce the window as much as possible. Additional search window reductions run the risk that Alpha-Beta may not be able to find the minimax value. According to the postcondition in section 2.1.3, one can only be sure that the minimax value is found if $\alpha < g < \beta$. If the return value lies outside the window, then all we are told is that a bound on the minimax value is found. To

```
function aspwin(n, estimate, delta) → f;
    α := estimate − delta;
    β := estimate + delta;
    g := alphabeta(n, α, β);
    if g ≤ α then
        g := alphabeta(n, −∞, g);
    else if g ≥ β then
        g := alphabeta(n, g, +∞);
    return g;
```

Figure 2.11: Aspiration Window Searching

find the true minimax value in that case, a re-search with the right window is necessary. In practice the savings of the tighter window out-weigh the overhead of additional re-searches, as has been reported in many studies (see for example [33, 99, 116]). In addition, the use of storage can reduce the re-search overhead (see section 2.2.2 and chapter 3).

Here we will describe two widely used techniques to benefit from the extra cutoffs that artificially-narrowed search windows yield.

Aspiration Window

In many games the values of parent and child nodes are correlated. Therefore we can obtain cheap estimates of the result that a search to a certain depth will return. (We can do a relatively cheap search to a shallow depth to obtain this estimate.) This estimate can be used to create a small search window, in chess typically ± the value of a pawn. This window is known as an *aspiration window*, since we aspire that the result will be within the bounds of the window. With this window an Alpha-Beta search is performed. If it "succeeds" (case 1 of the postcondition on page 18), then we have found the minimax value cheaply. If it "fails," then a re-search must be performed. Since the failed search has returned a bound, this re-search can also benefit from a window smaller than $\langle -\infty, +\infty \rangle$.

Aspiration window searching is commonly used at the root of the tree. Figure 2.11 gives the pseudo code for this standard technique. One option for the estimate is to evaluate the current position. Assuming that a pawn is given the value of 100, the call "aspwin(n, eval(n), 100)" would find us the minimax value and usually do so more efficiently than the call "Alpha-Beta($n, -\infty, +\infty$)." Some references to this technique are [8, 33, 47].

Null-Window Search and Scout

Pushing the idea of a small-search-window-plus-re-search to the limit is the use of a null-window. The values for α and β are now chosen so that Alpha-Beta will always

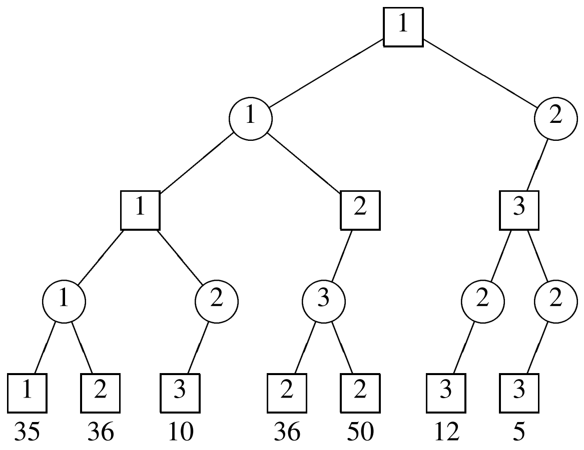


Figure 2.12: Minimal Tree (with node types)

return a bound. Case 1 of the postcondition in section 2.1.3 can be eliminated by choosing $\alpha = \beta - 1$, assuming integer-valued leaves, or $\alpha = \beta - \varepsilon$ in general, where ε is a number less than the smallest difference between any two leaf values. (The precondition of Alpha-Beta demands $\alpha < \beta$, so $\alpha = \beta$ won't work.)

An Alpha-Beta search window of $\alpha = \beta - 1$ ensures the highest number of cutoffs. On the downside, it also guarantees that a re-search is necessary to find the minimax value. Since we usually want a minimax value at the root, it makes more sense to use a wider aspiration window there, with a low probability of the need for a re-search. However, if we look at the structure of the minimal tree, we see that for most interior nodes a bound is all that is needed. Figure 2.12 shows an ordered version of the minimal tree of the Alpha-Beta example, with the nodes labeled with the three node types as defined by Knuth and Moore in [65]. Nodes that have only one child in the minimal tree are type 2, nodes with all children included are type 3, and nodes that are part of both the max and the min solution tree are type 1. The type 1 nodes form the path from the root to the best leaf. This intersection of the two solution trees is also known as the critical path or the principal variation (PV). Its interpretation is that of the line of play that the search predicts. For the type 1 nodes the minimax value is computed. In a depth d tree, there are only $d + 1$ type 1 nodes. The only task of the type 2 and type 3 nodes is to prove that it does not make sense to search them any further, because they are worse than their type 1 relative.

For the nodes off the PV it makes sense to use a null-window search, because a bound is all that is needed. In real trees that are not perfectly ordered, the PV node is not known *a priori* so there is a danger of having to re-search. Once again, in practice the small-window savings out-weigh the re-search overhead (see section 4.3.2 and [33, 82, 99]).

Thus we come to an algorithm that uses a wide search window for the first child, hoping it will turn out to be part of the PV, and a null-window for the other children. At a max node the first node should be the highest. If one of the null-window searches

```
function NegaScout(n, α, β) → g;
    if n = leaf then return eval(n);
    c := firstchild(n);
    g := NegaScout(c, α, β);
    c := nextbrother(c);
    if n = max then
        g := max(g, α);
        while g < β and c ≠ ⊥ do
            t := NegaScout(c, g, g + 1);
            /* the last two ply of the tree return an accurate value */
            if c = leaf or firstchild(c) = leaf then g := t;
            if t > g and t < β then g := NegaScout(c, t, β);
            g := max(g, t); c := nextbrother(c);
    else /* n is a min node */
        g := min(g, β);
        while g > α and c ≠ ⊥ do
            t := NegaScout(c, g − 1, g);
            if c = leaf or firstchild(c) = leaf then g := t;
            if t < g and t > α then g := NegaScout(c, α, t);
            g := min(g, t); c := nextbrother(c);
    return g;
```

Figure 2.13: NegaScout

returns a bound that is higher, then this child becomes the new PV candidate and should be re-searched with a wide window to determine its value. For a min node the first node should remain the lowest. If the null-window searches show one of the brothers to be lower, then that one replaces the PV candidate.

Early algorithms that use the idea of null-windows are P-alphabeta [33, 45], Scout [96, 99] and PVS [33, 85, 84]. Reinefeld has studied these algorithms as well as a number of other improvements [85, 116, 119]. This has resulted in NegaScout, the algorithm that is used by most high-performance game-playing programs. Figure 2.13 shows a minimax version of NegaScout. The choice of the name NegaScout is a bit unfortunate, since it suggests that this is just a negamax formulation of the Scout algorithm, instead of a new, cleaner, and more efficient algorithm. An extra enhancement in NegaScout is that it is observed that a fail-soft search of the last two ply of a tree always returns an exact value, even in cases 2 and 3 of the postcondition of section 2.1.3, so no re-search is necessary (figure 2.13 assumes a fixed-depth tree).

Empirical evidence has shown that, even without the use of extra memory, null-window algorithm NegaScout finds more cutoffs than wide-window algorithm Alpha-Beta($n, -\infty, +\infty$). Figure 2.14 illustrates this point. (The y axis is not drawn to scale.)

Most successful game-playing programs use a combination of NegaScout with aspiration window searching at the root. We will call this combination Aspiration

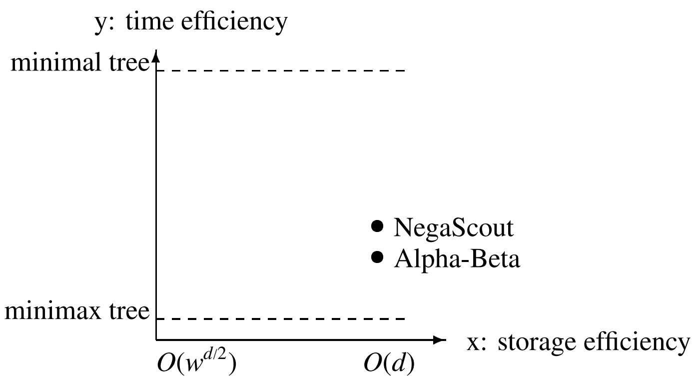


Figure 2.14: NegaScout's Performance Picture

NegaScout.

Another idea is to do away with wide-windowed re-searches altogether, and use null-windows only. This idea is further analyzed in chapter 3.

2.2.2 Move Ordering

A different way to improve the effectiveness of Alpha-Beta pruning is to improve the order in which child positions are examined. On a perfectly ordered uniform tree Alpha-Beta will cut off the maximum number of nodes. A first approach is to use application-dependent knowledge to order the moves. For example, in chess it is often wise to try moves that will capture an opponent's piece first, and in Othello certain moves near the corners are often better. However, there exist also a number of techniques that work independently of the application at hand.

The History Heuristic

We start with a technique that is in spirit still close to the use of application-dependent knowledge. In most games there are moves that are good in many positions, for example, the two types of moves just mentioned for chess and Othello. Schaeffer introduced the history heuristic, a technique to identify moves that were repeatedly good automatically [125, 128]. It maintains a table in which for each move a score is increased whenever that move turns out to be the best move or to cause a cutoff. At a node, moves are searched in order of history heuristic scores. In this way the program learns which moves are good in a certain position in a certain game and which are not, in an application-independent fashion. There exist a number of older techniques, such as killer moves and refutation tables, which the history heuristic generalizes and improves upon. Schaeffer has published an extensive study on the relative performance of these search enhancements [128].

Transposition Tables and Iterative Deepening

In many application domains of minimax search algorithms, the search space is a graph, whereas minimax-based algorithms are suited for *tree* search. Transposition tables (TT) are used to enhance the efficiency of tree-search algorithms by preventing the re-expansion of children with multiple parents (transpositions) [82, 128]. A transposition table is usually implemented as a hash table in which searched nodes are stored (barring collisions, the search tree). The tree-search algorithm is modified to look in this table before it searches a node and, if it finds the node, uses the value instead of searching. In application domains where there are many paths leading to a node, this scheme leads to a substantial reduction of the search space. (Although technically incorrect, we will stick to the usual terminology and keep using terms like minimax *tree* search.) Transposition tables are becoming a popular way to enhance the performance of single agent search programs as well [67, 117].

Most game-playing programs use iterative deepening [82, 128, 137]. It is based on the assumption that a shallow search is a good approximation of a deeper search. It starts off by doing a depth one search, which terminates almost immediately. It then increases the search depth step by step, each time restarting the search over and over again. Due to the exponential growth of the tree the former iterations usually take a negligible amount of time compared to the last iteration. Among the benefits of iterative deepening (ID) in game-playing programs are better move ordering (if used with transposition tables), and advantages for tournament time control information. (In the area of one-player games it is used as a way of reducing the space complexity of best-first searches [67].)

Transposition tables are often used in conjunction with iterative deepening to achieve a partial move ordering. The search value and the branch leading to the highest score (the best move) are saved for each node. When iterative deepening searches one level deeper and revisits nodes, this information is used to search the previously best move first. Since we assumed that a shallow search is a good approximation of a deeper search, this best move for depth d will often turn out to be the best move for depth $d+1$ too.

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

1. improve the quality of the move ordering, and

2. detect when different paths through the search space transpose into the same state, to prevent the re-expansion of that node.

In the case of an algorithm in which each ID iteration may perform re-searches, like NegaScout and aspiration window searching, there is an additional use for the TT:

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

```
function alphabeta(n, α, β) → g;
    if retrieve(n) = ok then
        if n.f⁻ ≥ β then return n.f⁻;
        if n.f⁺ ≤ α then return n.f⁺;
        α := max(α, n.f⁻);
        β := min(β, n.f⁺);
    if n = leaf then g := eval(n);
    else if n = max then
        g := −∞;
        c := firstchild(n);
        while g < β and c ≠ ⊥ do
            g := max(g, alphabeta(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(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 2.15: The Alpha-Beta Function for Use with Transposition Tables

In game-playing programs NegaScout is almost always used in combination with a transposition table. Figure 2.15 shows a version of Alpha-Beta for use with transposition tables [81, 82, 125]. Not shown is the code for retrieving and storing the best move, nor the fact that in most implementations usually one bound is stored, instead of both $n.f^+$ and $n.f^-$ (see also the remark on page 53).

Figure 2.16 shows that a better move ordering enables an algorithm to find more cutoffs. Usually a better move ordering comes at the cost of storing some part of the search tree, which is shown in the picture. Chapter 4 determines the storage needs of algorithms in game-playing programs that typically use quite a number of enhancements.

Originally transposition tables were introduced to prevent the search of transpositions in the search space—hence their name. However, as algorithms grew more sophisticated, the role of the transposition table evolved to that of storing the search tree. It has become a cache of nodes that may or may not be of use in future iterations or re-searches. Calling it a transposition table does not do justice to the central role that it

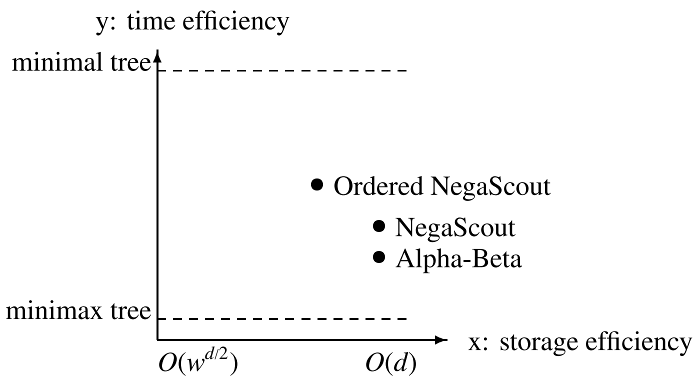


Figure 2.16: Performance Picture with Better Ordering

plays in a modern algorithm like ID Aspiration NegaScout—NegaScout enhanced with an initial aspiration window, and called in an iterative deepening framework. However, to prevent confusion we will keep using the term "transposition table."

Noting that current game-playing programs store (part of) the search tree in memory is a point that has to be taken into account when reasoning about the behavior of search algorithms. Many analyses and performance simulations of Alpha-Beta and its variants do not use a transposition table (for example, [8, 33, 63, 85, 99, 119]). Since a transposition table makes the overhead of doing re-searches negligible, this is a serious omission. The consequences of this point will be discussed in a broader scope in section 4.4. The fact that the search tree is effectively stored in memory also means that many theoretical schemes in which an explicit search tree plays a central role are closer to practice than one would generally assume [57, 96, 102, 104].

Of central importance for the validity of the *TT = Search Tree* assumption is the question of how big the search tree is that is generated by an algorithm, whether it fits in the available memory. This question is answered in section 4.1.

2.2.3 Selective Search

Up to now we have been dealing exclusively with fixed-depth, full-width algorithms. The pruning method used is sometimes called *backward* pruning, since the backed-up return values of Alpha-Beta are used for the cutoff decisions. Studying this class of algorithms has the advantage that performance improvements are easily measurable; one has only to look at the size of the tree. Although our research is concerned with fixed-depth full-width search, a short overview of selective search is useful to put things in perspective. (In practice a fixed-depth search may not always search all moves equally deep, for example by finding that a move leads to an early end of the game.)

In this section we will discuss algorithms that do not necessarily search full-width. It has long been known that a flaw of the Alpha-Beta algorithm is that it searches all nodes to the same depth. No matter how bad a move is, it gets searched as deep as the most promising move [135]. Of course, backward pruning will probably make sure that most of the nodes in the subtree of the bad move get pruned, but a more selective search strategy could perhaps make sure that really bad moves are not considered at all. These strategies are said to use selective searching (in contrast to full-width), variable depth (in contrast to fixed-depth), or forward pruning (in contrast to backward pruning). An algorithm that uses selective deepening can search a line of play more deeply in one iteration than in another. Forward pruning is a bit more aggressive. An algorithm that uses forward pruning will not select a pruned node again in later iterations, unless the score drops drastically. The problem with these ideas is to find a method to decide reliably which move is bad. Some moves may look bad at first sight, but turn out to be a winning move after a deeper search.

Comparing the quality of selective deepening and forward pruning enhancements is hard. Tree size is no longer the only parameter. With these schemes the quality of the decision becomes important as well, since one could say that search extensions try to mend deficiencies in the evaluation function by searching deeper. Selective deepening adds a second dimension to performance comparisons. In fixed-depth searching, improvements mean more cutoffs in the search; algorithm performance measurements are only a matter of counting nodes.

One way to compare selective deepening algorithms is by having different versions of the program play matches. A problem with this approach is that deficiencies in both versions can distort the outcome. Another problem is that these matches take up a large amount of time, especially if they are to be played with regular tournament time controls.

Alpha-Beta Search Extensions

Most game-playing programs are still based on the Alpha-Beta algorithm. In this setting search extensions are often introduced to make up for features that a static evaluation cannot uncover. A well-known problem of fixed-depth searching is the horizon effect [12]. Certain positions are more dynamic than others, which makes them hard to assess correctly by a static evaluation. For example, in chess re-captures and check evasions can change the static assessment of a board position drastically. The inability of an evaluation function to assess tactical features in a position is called the horizon effect. To reduce it, most programs use *quiescence search* [10, 47]. A quiescence search extends the search at a leaf position until a quiet position is reached. In chess "quiet" is usually defined as no captures present and not in check.

Going a step further are techniques such as singular extensions and null-moves. Singular extensions try to find, through shallow searches, a single move in a position that is more promising than all others. This node is then searched more deeply (details can be found in [5, 6]). A null-move is a move in which the side to play passes (the other side gets two moves in a row). In most positions passing is worse than any other move. The idea behind null-moves is to get a lower bound on a position in a cheap way, because all other moves are presumably better. (This is not true for all games. For example, in checkers and Othello null-moves do not work. Also, when in zugzwang in chess (a series of positions with no good move), passing can be beneficial.) If this lower bound causes a cutoff, then it has been found in a cheap way. If it does not, then little effort is wasted. Many people have experimented with different uses of the null-move. More can be found in [1, 10, 11, 51, 95, 126].

Another source of information on which nodes to prune is the result of the shallow search itself. A successful example of this idea is the ProbCut algorithm, formulated by Buro [31]. ProbCut performs a shallow null-window Alpha-Beta search to find nodes whose value will fall with a certain probability outside the search window. These nodes can be left out of the full-depth search. Although many researchers have used similar ideas—sometimes for move ordering, sometimes for selective search, sometimes for forward pruning—Buro was the first to use statistical analysis to determine the most effective values for the search windows.

An open problem in the use of selective search methods and transposition tables are *search inconsistencies*, mentioned by Schaeffer [130]. Searching a position to different depths can yield different values. Transpositions can cause nodes to return results for a deeper search. This can cause cutoff and score problems. Other than not using the deeper search result (which is wasteful) we do not know of a method to solve this problem.

A related problem of selective deepening is that one can use too much of it [130]. Selectively extending certain lines causes the search to become uneven. As an algorithm is made more selective, comparing scores for different depths gives more and more inconsistencies. Furthermore, moves that look less promising at first sight are not given enough resources, causing the program to miss good moves that a full-width search (or at least less selectivity) with the same resources would have found. Thus, selective search should be used with care.

There has been some debate over the pros and cons of selective searching, ever since Shannon's original article [135]. An interesting historical account is given by Beal [11], which we summarize in the following. In the early days of computer chess the lack of power of computers almost forced researchers to use forward pruning to achieve reasonable results. For example, Bernstein's program of around 1957 [17] looked at the "best 7" moves at every node. Ten years later Greenblatt's program [52], using carefully chosen quiescence rules, became the best of its time. By 1973 computers had grown powerful enough to make fixed-depth full-width searching the winning technique in the program Chess 4.0 [137], although a quiescent search at the leaves of the fixed-depth tree remained an important component.

The 1970's and 1980's showed interesting research into selective search algorithms [1, 13, 87, 95], as well as the drive towards using more powerful computers, such as parallel computers and special hardware [34, 40, 55, 83, 127]. The strongest programs emphasized full-width searching based on powerful hardware. These two trends have continued into the 1990's. Programs running on massively parallel computers, such as Deep Thought [54], *Socrates [60, 75], Zugzwang [41, 42] and Frenchess [147] compete against commercially available programs running on personal computers, such as Fritz, Chess Genius, and WChess. These programs use highly tuned aggressive selective deepening and forward pruning techniques, often based on null-moves and shallow searches, combined with lightweight evaluation functions. After the successes of full-width search in the 1970's and 1980's, the fact that personal computer programs are not crushed by the parallel power is an indication of the viability of selective search techniques, although we should not forget that personal computers have become quite fast as well, over the years. Also, there is some evidence that the utility of searching an extra ply deeper is less for deeper searches. For deep searches this increases the relative importance of a good, highly tuned, evaluation function [133].

2.3 Alternative Algorithms

The Alpha-Beta tree-searching algorithm has been in use since the end of the 1950's. No other minimax search algorithm has achieved the wide-spread use in practical applications that Alpha-Beta has. Thirty years of research has found ways of improving the algorithm's efficiency, and variants such as NegaScout and PVS are quite popular. Interesting alternatives to depth-first searching, such as breadth-first and best-first strategies, have been largely ignored.

In this section we will look at these alternative algorithms.

2.3.1 Best-First Fixed-Depth: SSS*

In 1979 Stockman introduced SSS*, a fixed-depth full-width algorithm which looked like a radically different approach from Alpha-Beta for searching minimax trees [140]. It builds a tree in a best-first fashion by visiting the most promising nodes first. Alpha-Beta, in contrast, uses a depth-first, left-to-right traversal of the tree. Intuitively, it would seem that a best-first strategy should prevail over a rigidly ordered depth-first one. Stockman proved this intuition true: SSS* dominates Alpha-Beta; it never evaluates more leaf nodes than Alpha-Beta. Consequently, SSS* has drawn considerable attention in the literature (for example, [21, 32, 33, 72, 73, 74, 78, 82, 85, 101, 102, 116, 118, 121]). On average SSS* evaluates considerably fewer leaf nodes. This has been repeatedly demonstrated in the literature by numerous simulations (for example, [63, 85, 88, 116, 118, 121]).

The code of SSS* is shown in figure 2.17 (taken from [32, 99, 140]). SSS* works by manipulating a list of nodes, the OPEN list, using six ingeniously inter-locking cases of the so-called Γ operator. The nodes have a status associated with them, either live or solved, and a merit, denoted $\hat{h}$. The OPEN list is sorted in descending order, so that the entry with highest merit (the "best" node) is at the front, to be selected for expansion. SSS* is a variation of AO*, an algorithm for searching AND/OR graphs, or problem-reduction-spaces. The more widely known A* algorithm searches plain OR graphs, or state-spaces [93, 94, 99].

At first sight the code in figure 2.17 may look overwhelmingly complex. However, once we see through the veil of the code, we find that SSS*'s idea of best-first node

Stockman's SSS* (including Campbell's correction [32])

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

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

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

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

(5) Go to (2)

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

Γ	Conditions satisfied by input state $\langle n, s, \hat{h}\rangle$	Actions of Γ in creating new output states
–	s = SOLVED n = ROOT	Final state reached, exit algorithm with $g(n) = \hat{h}$.
1	s = SOLVED $n \neq$ ROOT type(n) = MIN	Stack $\langle m = \text{parent}(n), s, \hat{h}\rangle$ on OPEN list. Then purge OPEN of all states $(k, s, \hat{h})$ where m is an ancestor of k in the game tree.
2	s = SOLVED $n \neq$ ROOT type(n) = MAX next(n) $\neq$ NIL	Stack $\langle$next(n), LIVE, $\hat{h}\rangle$ on OPEN list
3	s = SOLVED $n \neq$ ROOT type(n) = MAX next(n) = NIL	Stack $\langle \text{parent}(n), s, \hat{h}\rangle$ on OPEN list
4	s = LIVE first(n) = NIL	Place $\langle n$, SOLVED, $\min(\hat{h}, f(n))\rangle$ on OPEN list (interior) in front of all states of lesser merit. Ties are resolved left-first.
5	s = LIVE first(n) $\neq$ NIL type(first(n)) = MAX	Stack $\langle \text{first}(n), s, \hat{h}\rangle$ on (top of) OPEN list.

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

Figure 2.17: Stockman's SSS* [99, 140]

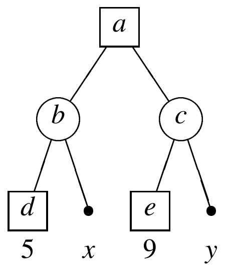


Figure 2.18: Which is “Best:” x or y?

selection is quite straightforward. SSS* finds the minimax value through a successive lowering of an upper bound on it. As the example of appendix A.2 illustrates, a number of passes can be distinguished in this process. In each pass the value of a new (better) upper bound is computed. What makes SSS* such an interesting algorithm, is that in each of these passes it selects nodes in a best-first order.

“Best” in the SSS* sense is defined in a simple manner. The upper bound of each pass is defined by a max solution tree (possibly under construction), whose minimax value is the maximum of its leaves. The leaf (or leaves) defining this value is the critical leaf at the end of the principal variation. See figure 2.18 for an example. In the figure the PV consists of nodes a, c, e. The value at the root is an upper bound of 9. Expanding node x will not change the minimax value of the tree. Only expanding the brother of the critical leaf e might give a better (sharper) upper bound, if its value happens to be less than 9. Thus, the SSS*-best node to expand in figure 2.18 is node y. (If there is more than one critical leaf, SSS* selects the left-most.)

The “best” node in SSS* terms is the left-most and deepest node whose expansion can possibly lower the upper bound. Intuitively, the fact that SSS* dominates Alpha-Beta is based on the property of the best node that it cannot be cut off by some others: node x can possibly be cut off by node y (for example, if y would get value 8) however, node y cannot be cut off by expansion of node x. Any node that Alpha-Beta could cutoff is not “best” in the SSS* sense.

Appendix A.2 contains a detailed example of how SSS* finds the minimax value of a tree in a best-first manner.

2.3.2 The Conventional View on SSS*

The literature lists a number of problems with SSS*. First, it takes considerable effort to understand how the algorithm works from a minimax point of view and still more to see its relation to Alpha-Beta, as a glance at figure 2.17 and the examples may have suggested. Although it is possible to follow the individual steps in the example, one of the problems with Stockman’s formulation is that it is hard to see what is “really” going on. The high level explanation provided with figure 2.18 is not obtained easily. An understanding at a higher level of abstraction is essential for those who wish to work with this algorithm. For example, parallelizing the original Stockman formulation is hard, and getting good results is even harder, as is shown by a number of (simulation) attempts [4, 19, 38, 71, 73, 145].

Another problem caused by the complexity of the algorithm is that to our knowledge nobody has reported measurements on how good SSS* performs in actual high performance game-playing programs; all published performance assessments are based on simulations. (One could argue that work published in 1987 by Vornberger and Monien is an exception [145], since chess positions were used in these measurements. However, one of the authors of the chess program reports that the work was performed with a less-than-state-of-the-art chess program, without any enhancements such as minimal windows, transposition tables, the history heuristic, or iterative deepening [89]. As can be expected, the results showed that SSS* searched substantially less leaf nodes than Alpha-Beta, in line with the simulation literature. Since 1987 the quality of their program has increased considerably [41]. Regrettably, they have not reported tests with SSS* since then.)

Second, a drawback of SSS* is its memory usage. SSS* maintains an OPEN list, similar to that found in single-agent best-first search algorithms like A* [99]. The size of this list grows exponentially with the depth of the search tree. This has led many authors to conclude that SSS* is effectively disqualified from being useful for real applications like game-playing programs [63, 88, 121, 140].

The OPEN list must be kept in sorted order. Insert and (in particular) delete/purge operations on the OPEN list are slow. They can dominate the execution time of any program using SSS*. Despite the promise of expanding fewer nodes, the disadvantages of SSS* have proven a significant deterrent in practice. The general view of SSS* then is that:

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

Figure 2.19 illustrates the point that at the cost of storing a solution tree of size $O(w^{d/2})$ in memory, the number of cutoffs can be improved through a best-first expansion sequence. This picture shows the conventional view in the literature of SSS*, where it is compared against un-enhanced algorithms (note that here we call NegaScout an un-enhanced algorithm, while elsewhere we consider it to be an Alpha-Beta variant enhanced with null-windows). In chapter 4 an enhanced version of SSS* will be compared against enhanced versions of Alpha-Beta and NegaScout.

In the next chapter we present results of our attempts to understand how and why SSS* works and see whether its drawbacks can be solved.

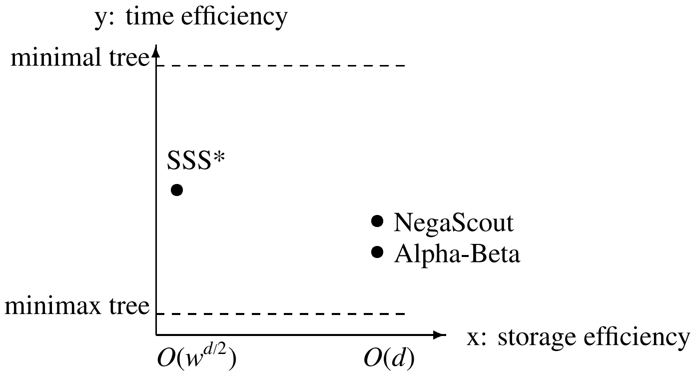


Figure 2.19: Performance Picture of Best-First Search Without Enhancements

Not everybody agrees with our describing SSS* as difficult to understand. Stockman's original application had little to do with minimax search or game-playing. His application was an analyzer for medical data related to EKG's (electrocardiograms). SSS* was used in parsing syntactic descriptions (AND/OR graphs) of waveforms to recognize certain types of waves [139]. Commenting on our (game-tree) view on SSS*, Stockman remarks: "So, you have yet another way and think that my reasoning was opaque—it just seemed to lay out that way naturally! Actually, I was competing multiple parse trees against each other in a waveform parsing application. I wanted an algorithm where the currently best parse tree was the one extended and this got me into SSS*. [...] The version of SSS* that I actually implemented in the thesis could develop trees bottom-up or even middle out and not just top-down as game-tree searches do." [141].

Stockman views SSS* as a problem-reduction search method such as AO*, in which one searches for the best solution tree. This view is in line with Nilsson's explanation of "an ordered search algorithm for AND/OR trees" which was later called AO* [93, 94]. Nilsson stresses the difference between the state-space and problem-reduction-space approach as follows: "The appropriate ordering technique involves asking 'Which is the most promising potential solution tree to extend?' rather than asking 'Which is the most promising node to expand next?' " [93, p. 129]. In chapter 3 we will go against this advice and look at SSS* asking which *node* to expand next. Applying a state-space view leads in this case to a reformulation that is much clearer.

2.3.3 Variable Depth

Sections 2.3.1 and 2.3.2 presented an alternative to Alpha-Beta in the form of the best-first fixed-depth full-width SSS* algorithm. Variable search depth algorithms provide another alternative to Alpha-Beta. It has always been felt that one of the biggest drawbacks of Alpha-Beta is its rigid search depth. Section 2.2.3 discussed Alpha-Beta search extensions—enhancements to reduce this drawback while staying within the Alpha-Beta framework. In this section we briefly review algorithms that do away with Alpha-Beta completely.

The frustration with Alpha-Beta has been well described in an article by Hans Berliner [13]. He proposes a more human-like search method, B*, which manipulates two heuristic bounds, an optimistic bound and a pessimistic bound. Lines of play that are easily refuted are not searched deeply, whereas more promising lines are searched more deeply. Furthermore, B* stops searching as soon as the backed-up value of the pessimistic bound of the best move is greater or equal to the backed-up value of the optimistic bound. By stopping the search before the minimax value is known, less nodes have to be expanded. A problem with B* is to find reliable heuristic bounds. This has turned out to be an obstacle.

A second problem of the Alpha-Beta framework is that the evaluation function returns a single value, without an assessment of the quality of the evaluation. In certain types of positions the evaluation function may be quite good at assessing the merit, whereas in other positions the return value is less trustworthy. B*'s optimistic and pessimistic bounds are better, in the sense that a narrow gap between the bounds can be interpreted as a high confidence, and a wide gap signals a low confidence. The work on B* was extended by Palay to include probability distributions, leading to an algorithm called PB* [95]. In a recent paper, Berliner and McConnell showed the performance of a new version of B* to approach that of a high-quality Alpha-Beta-based program [16]. They argue that there is still room for improvement, although it is also clear that the current implementation of the B* idea has become quite complex, when compared to the original idea in [13].

A potential disadvantage of more elaborate back-up schemes is that the extra information gained by having a backed-up probability distribution can be offset by the longer time it takes to compute it. However, the idea of backing up more information than just a single value is quite appealing, and this is an area of active research. See for example the work of Russell and Wefald [122], Ballard [7], Rivest [120], and Baum [9]. Junghanns describes related experiments using fuzzy numbers [61]. Some results show an algorithm beating unenhanced text-book versions of Alpha-Beta, for example [9, 122] (and even [140]). However, despite all efforts, we do not know of a result that proves the superiority of backing up probability distributions over programs based on Alpha-Beta. Beating Alpha-Beta is not an easy task, because the yardstick that has to be beaten has been considerably enhanced, amongst others by selective deepening techniques (section 2.2.3).

Another idea for variable depth search is Conspiracy Numbers [87, 129], and its Boolean variant, Proof Numbers [3]. These algorithms can be seen as a "least-work-first" approach. For every possible outcome of a node n these algorithms compute how many nodes in n's sub-tree must change their value to have n take on that value. In Proof Number search the value is either a 1 or a 0. These ideas have proven to be useful for certain problems. For minimax trees that are highly irregular—such as chess mating problems, the game of Awari, and for solving certain games—Proof Numbers can build smaller trees than Alpha-Beta [2]. If there is no clear solution to be found, then Proof Numbers does not perform well. In [29] a relation between proof trees and solution trees is discussed.

Recently another algorithm for performing variable depth minimax search has been proposed by Korf and Chickering [69, 70]. This algorithm, called Best-First Minimax Search (BFMMS), starts by expanding the root position and evaluating the children. Then it selects the best child and expands it, after which it evaluates its children, and so on and so forth. Although both BFMMS and SSS* are "best-first" algorithms, they traverse the search space in quite different ways. SSS* is in principal a fixed-depth algorithm, BFMMS is inherently variable-depth. SSS* expands the *brother* of the critical leaf (figure 2.18), BFMMS expands its *children*. BFMMS appplies a single-agent, A*-like, expansion strategy to the domain of two-agent search.

Korf and Chickering report first results with this algorithm that show it to perform well on moderate depth simulated Othello-like trees. For deeper searches the trees become too unbalanced. Here a hybrid algorithm with Alpha-Beta, using BFMMS as a kind of Alpha-Beta search extension, seems more effective.

2.4 Summary

Despite the considerable interest in new algorithms for minimax search, most successful game-playing programs are still based on the Alpha-Beta algorithm with enhancements like iterative deepening, transposition tables, narrow windows, the history heuristic, and search extensions. Many other algorithms seemed or seem promising. One of them is SSS*. It is a relatively conservative algorithm in that it does not rely on selective search, just like the original Alpha-Beta algorithm. This makes performance comparisons between Alpha-Beta and SSS* relatively easy, since one can compare tree sizes. Furthermore, there exist theoretical models—notably solution trees—to guide one's thoughts in trying to understand their search trees.

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:

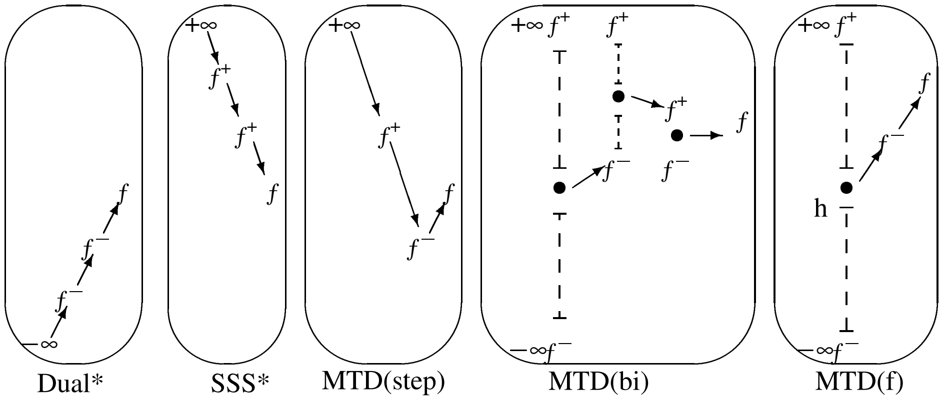


Figure 3.4: MT-based Algorithms

```
function MTD(n, first, next) → f;
    f⁺ := +∞; f⁻ := −∞;
    bound := first;
    repeat
        g := 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$(n, +\infty, \mathit{bound} := g)$
  This is just MT-SSS*. For brevity we call this driver MTD($+\infty$).

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

- MTD$\big(n$, approximation, **if** $g < \mathit{bound}$ **then** $\mathit{bound} := g$ **else** $\mathit{bound} := g + 1\big)$
  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$\big(n, \lceil \text{average}(+\infty, -\infty) \rceil, \mathit{bound} := \lceil \text{average}(f^+, f^-) \rceil\big)$
  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$\big(n, +\infty, \mathit{bound} := \max(f_n^- + 1, g - \mathit{stepsize})\big)$
  Instead of making tiny jumps from one bound to the next, as in MTD($+\infty$),

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.

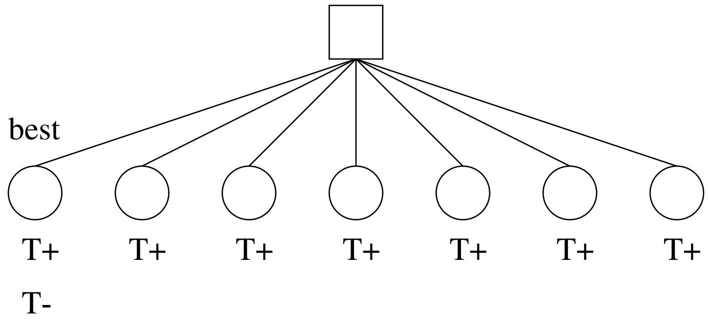


Figure 3.7: Best Move

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

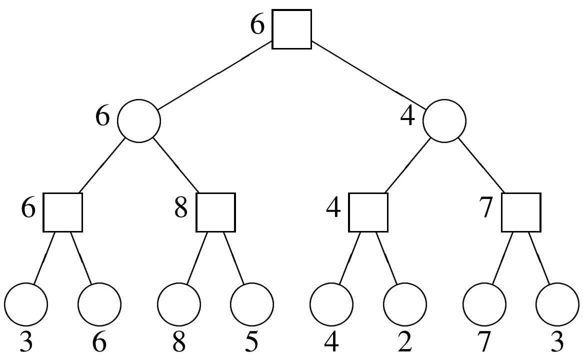


Figure 3.8: Alpha Bounding Expands 7

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.

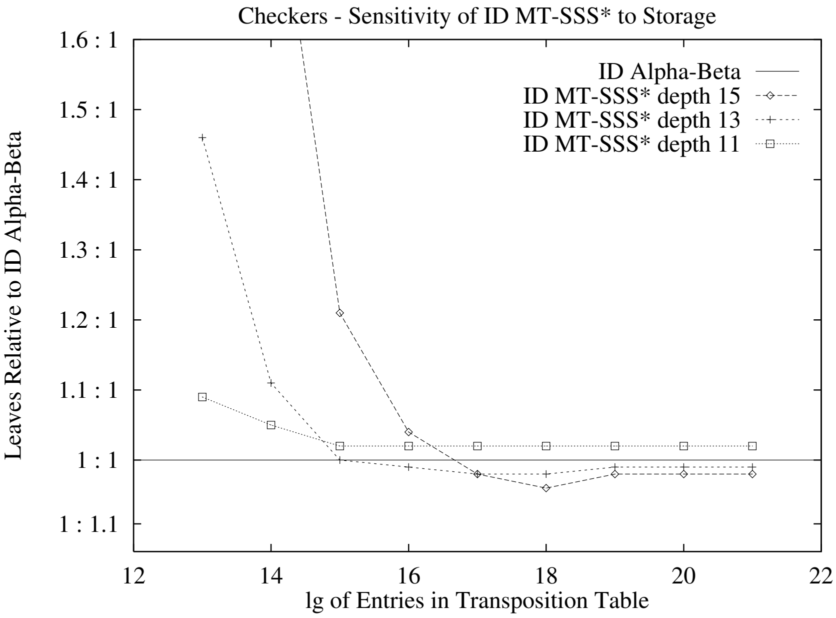


Figure 4.1: Memory Sensitivity ID MT-SSS* Checkers

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.

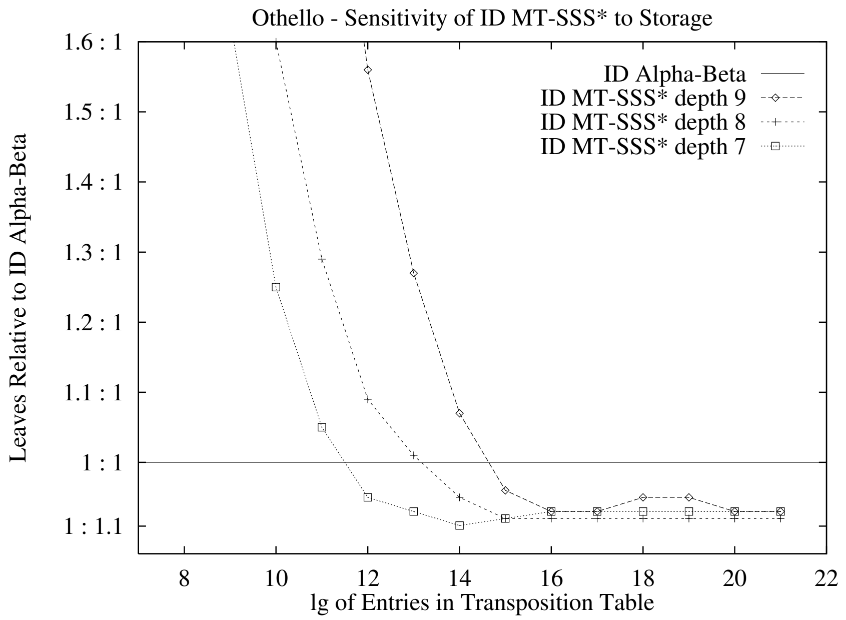


Figure 4.2: Memory Sensitivity ID MT-SSS* Othello

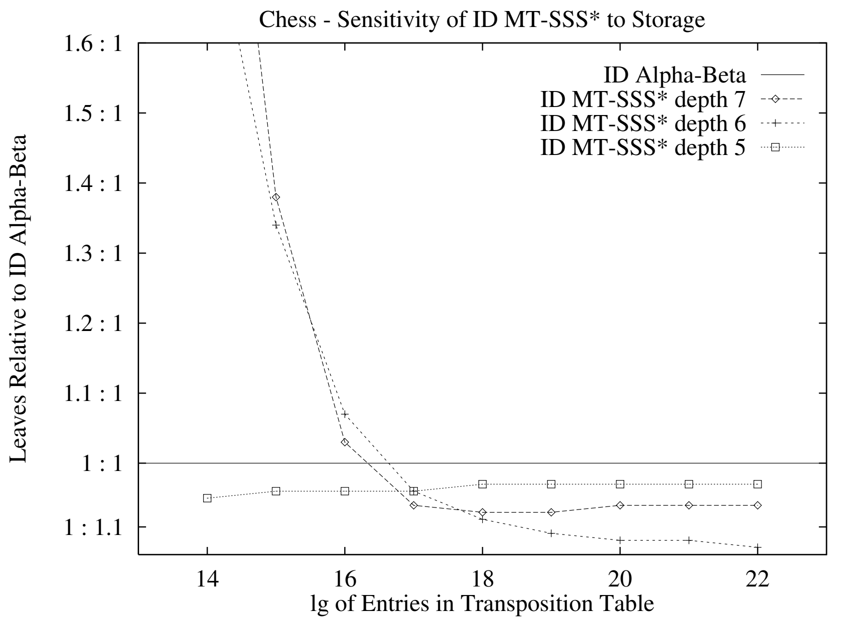


Figure 4.3: Memory Sensitivity ID MT-SSS* Chess

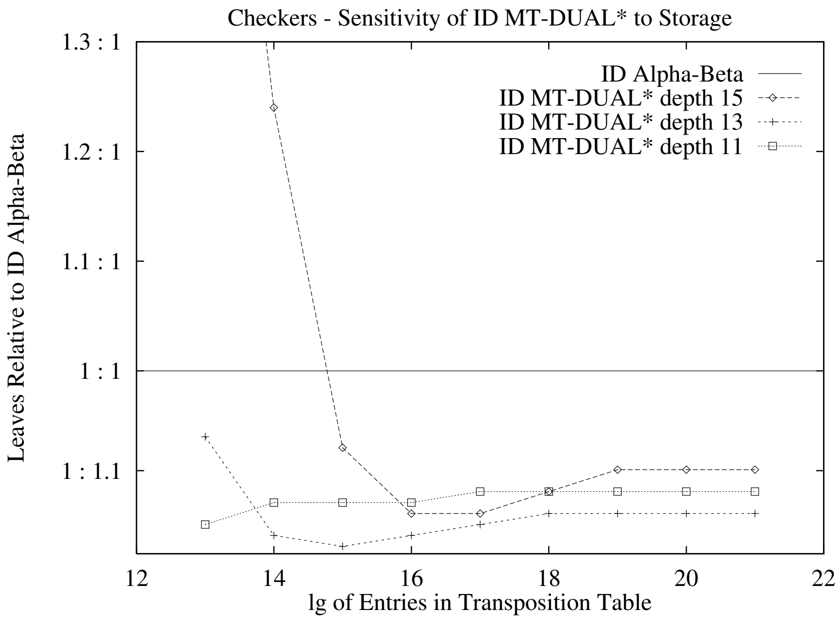


Figure 4.4: Memory Sensitivity ID MT-DUAL* Checkers

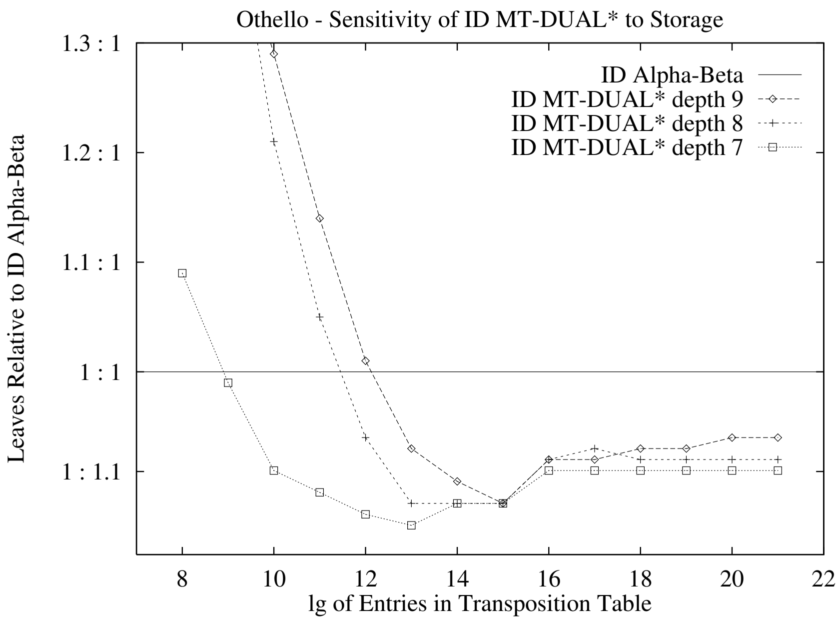


Figure 4.5: Memory Sensitivity ID MT-DUAL* Othello

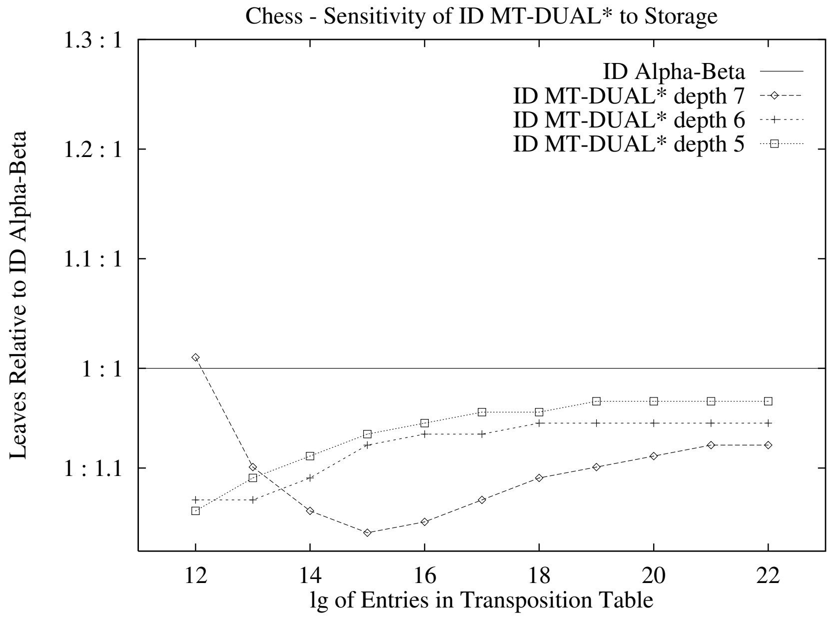


Figure 4.6: Memory Sensitivity ID MT-DUAL* Chess

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 $O\left(w^{\lceil (d-1)/2\rceil}\right)$. So, for high performance in MT-SSS* we need $O\left(w^{\lceil d/2\rceil}\right)$ for the transpositions, $O\left(w^{\lceil (d-1)/2\rceil}\right)$ for the multiple passes, and $O\left(w^{\lceil (d-2)/2\rceil}\right)$ 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 $O\left(w^{\lfloor d/2\rfloor}\right)$. For high performance MT-DUAL* needs $O\left(w^{\lfloor d/2\rfloor}\right)$ for the transpositions, $O\left(w^{\lfloor (d-1)/2\rfloor}\right)$ for the multiple passes, and $O\left(w^{\lceil (d-2)/2\rceil}\right)$—*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

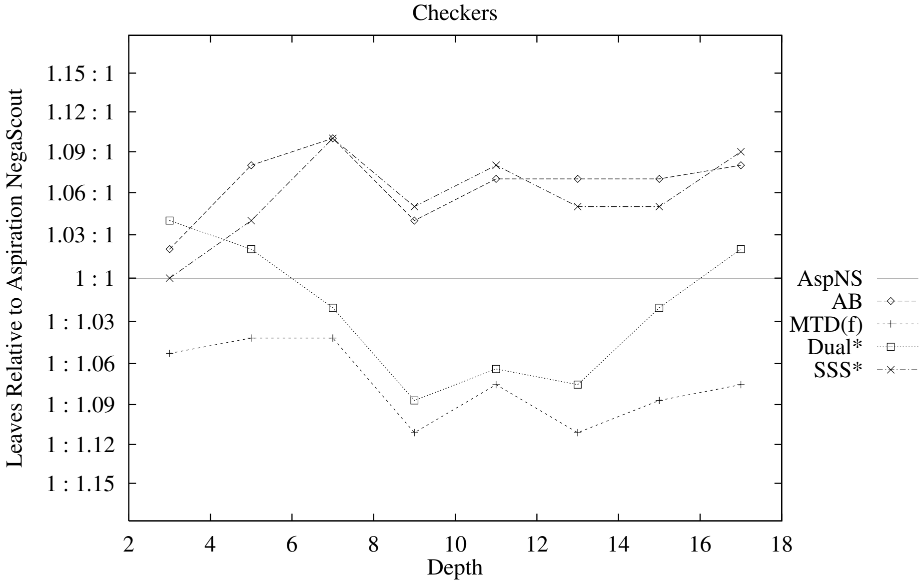


Figure 4.7: Leaf Node Count Checkers

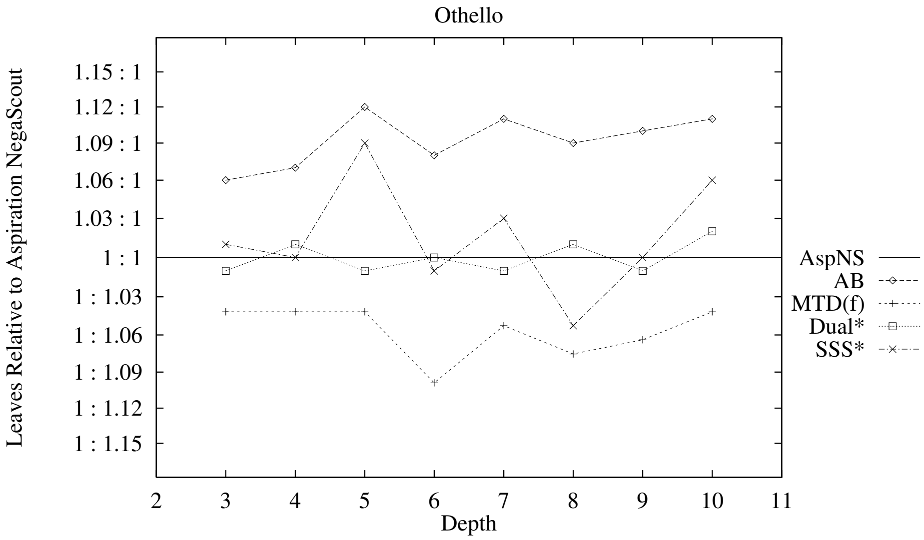


Figure 4.8: Leaf Node Count Othello

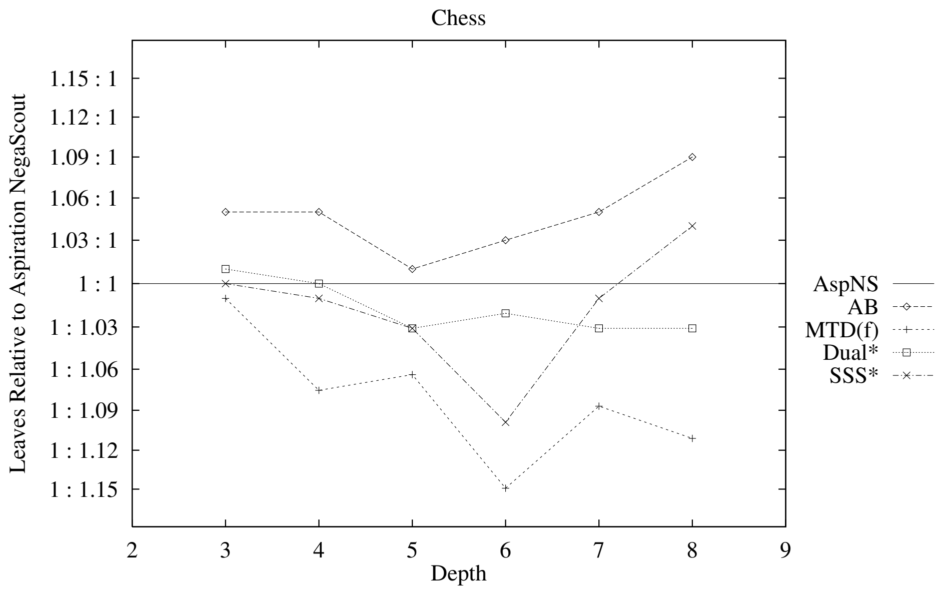


Figure 4.9: Leaf Node Count Chess

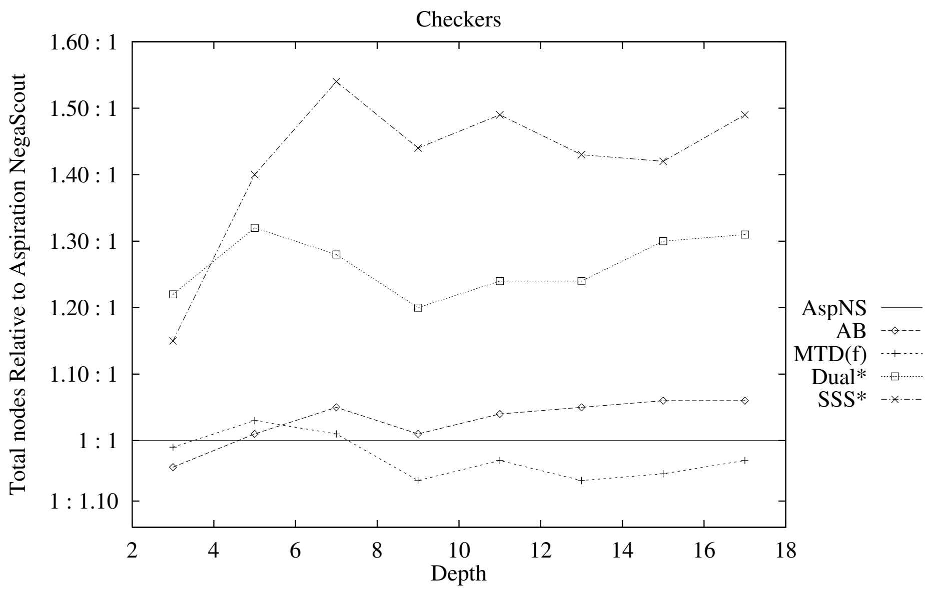


Figure 4.10: Total Node Count Checkers

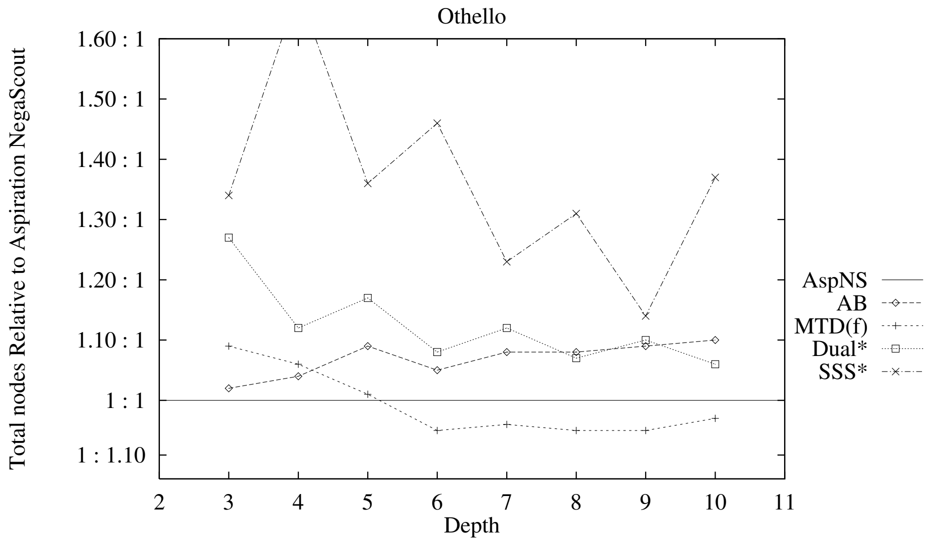


Figure 4.11: Total Node Count Othello

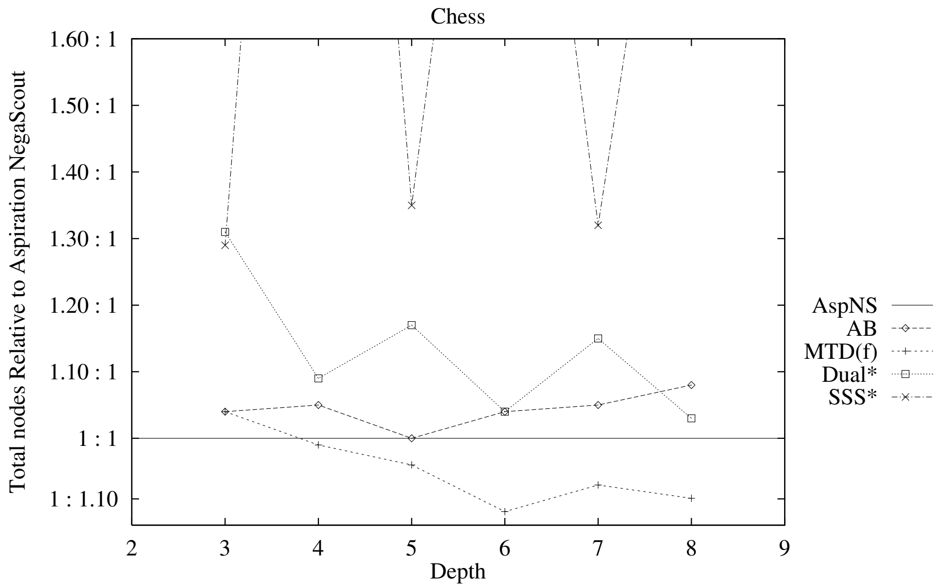


Figure 4.12: Total Node Count Chess

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*

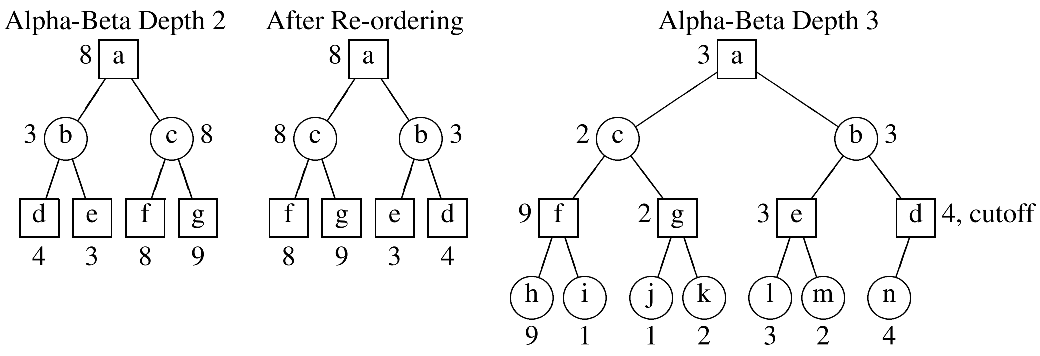


Figure 4.13: Iterative Deepening Alpha-Beta

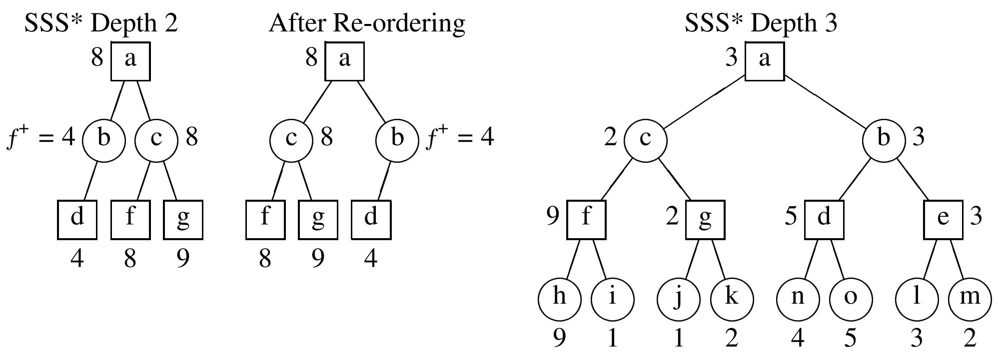


Figure 4.14: Iterative Deepening SSS*

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

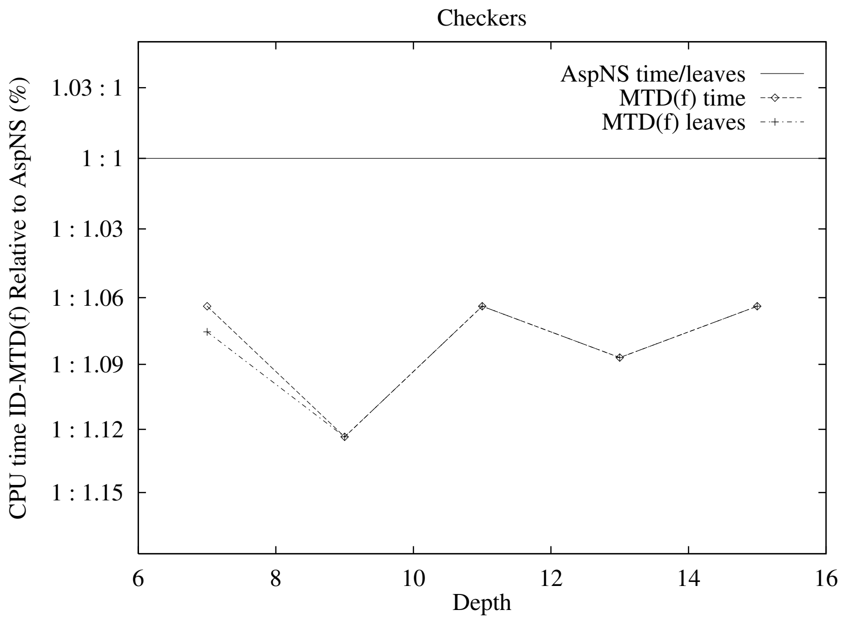


Figure 4.15: Execution Time Checkers

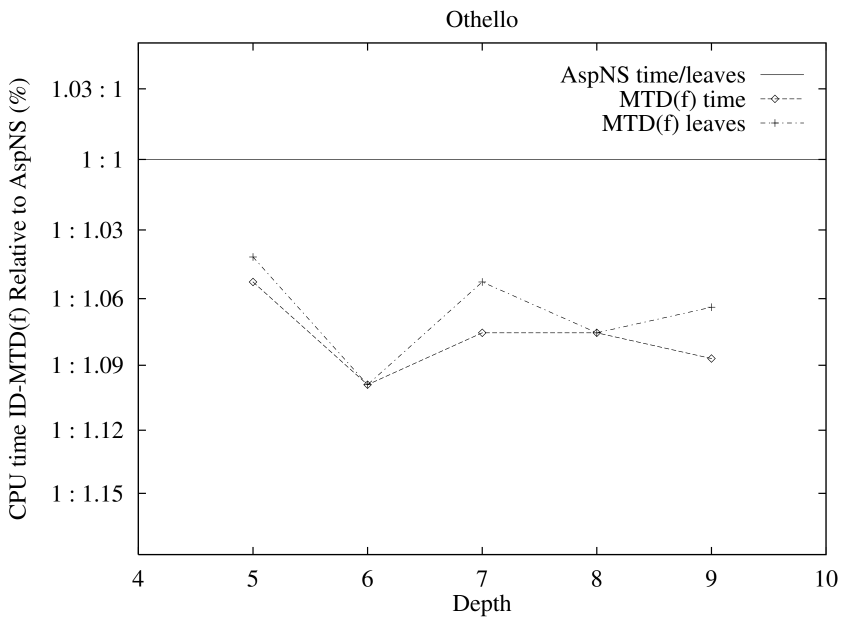


Figure 4.16: Execution Time Othello

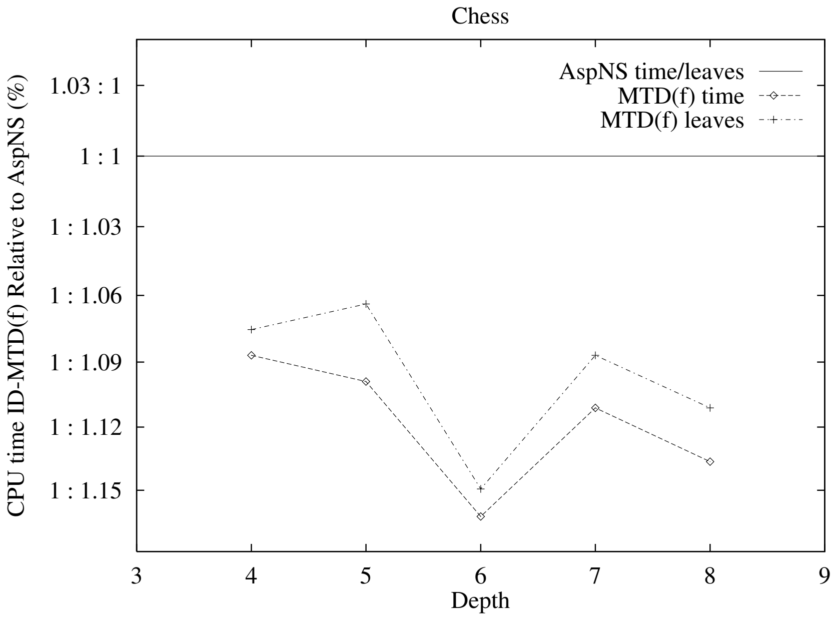


Figure 4.17: Execution Time Chess

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

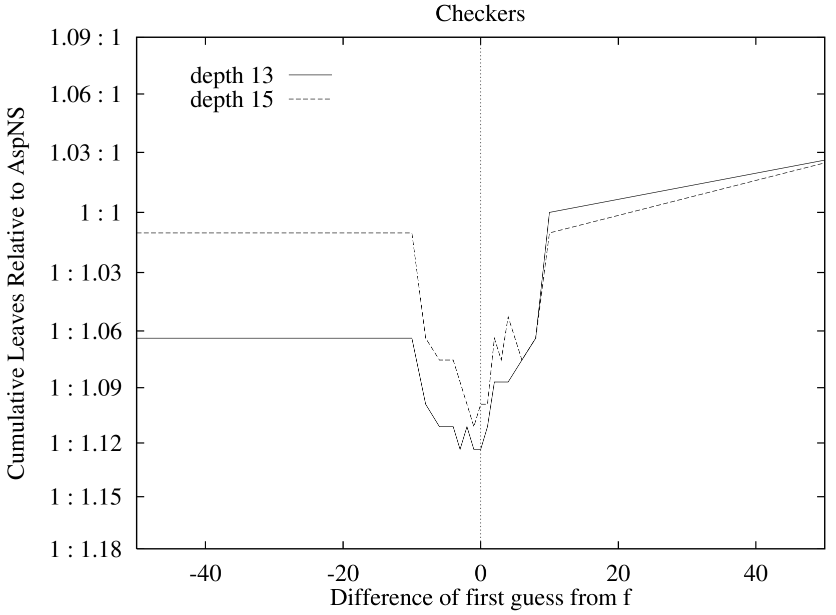


Figure 4.18: Tree Size Relative to the First Guess f in Checkers

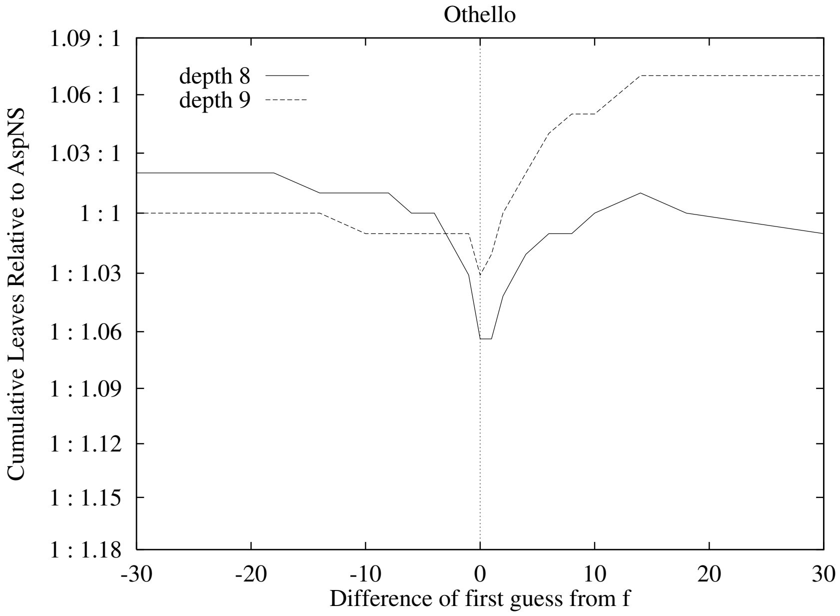


Figure 4.19: Tree Size Relative to the First Guess f in Othello

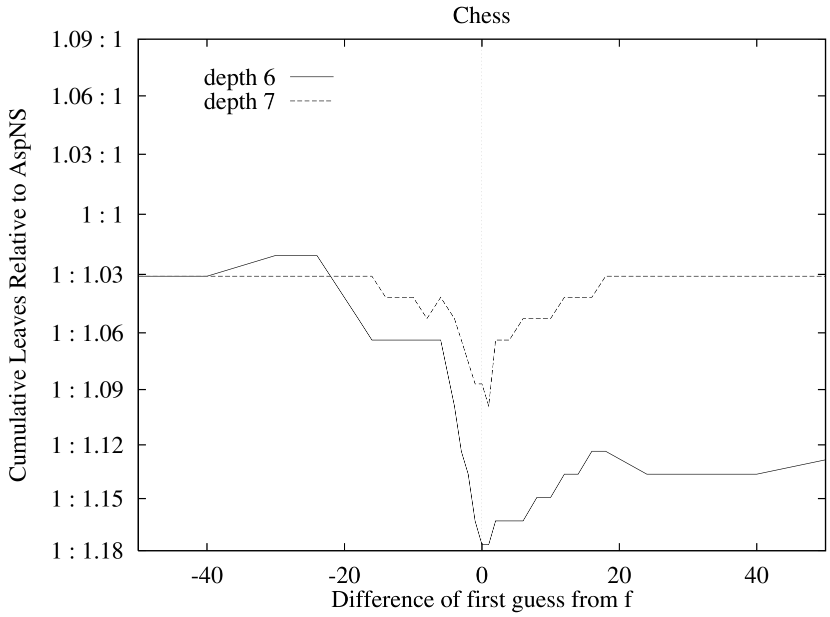


Figure 4.20: Tree Size Relative to the First Guess f in Chess

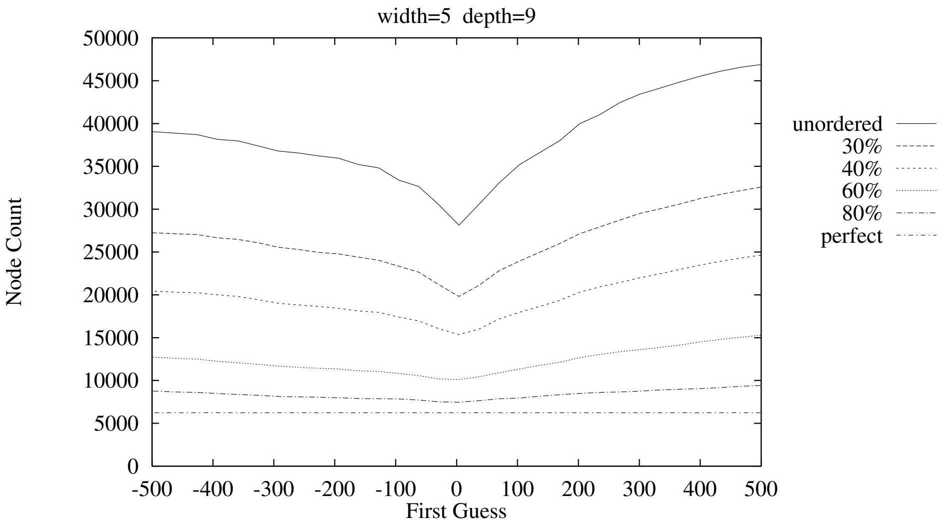


Figure 4.21: Effect of First Guess in Simulated Trees

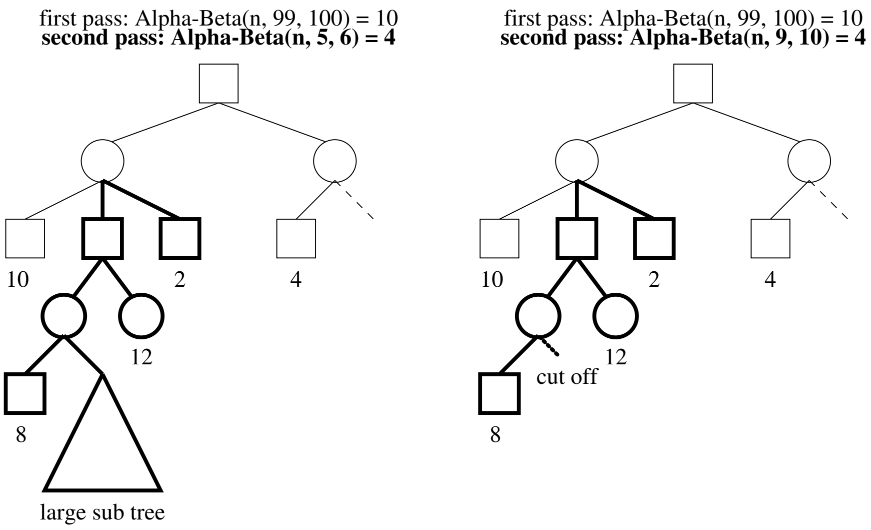


Figure 4.22: Two Counter Intuitive Sequences of MT Calls

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

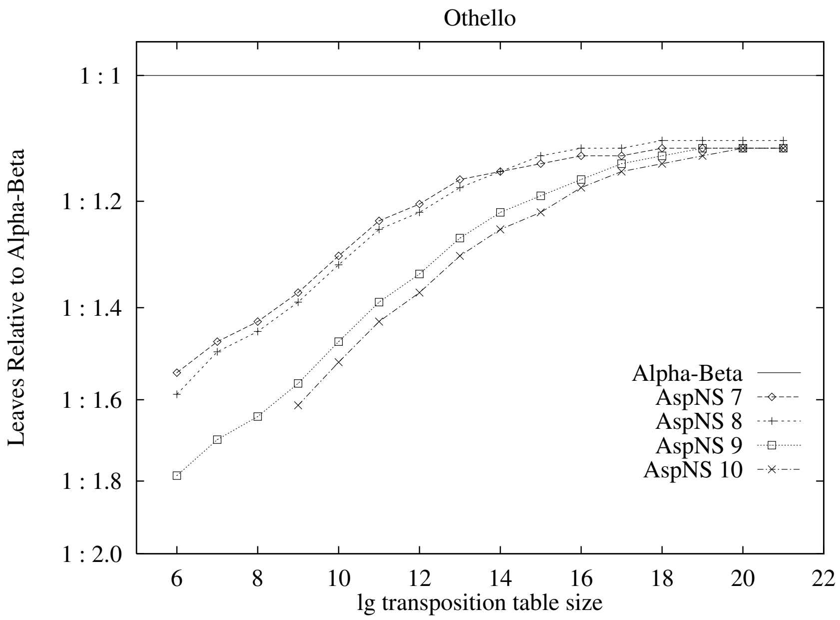


Figure 4.24: Aspiration NegaScout in Small Memory in Othello

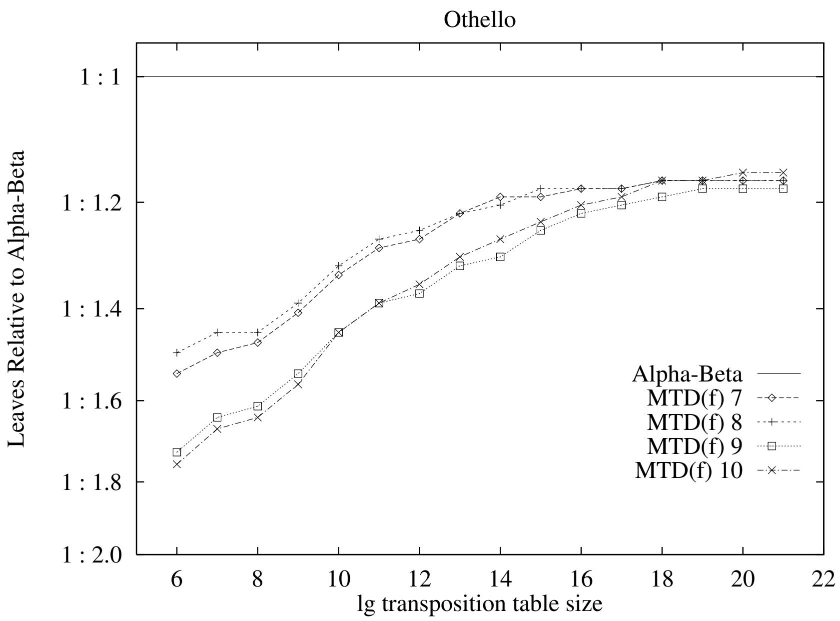


Figure 4.25: MTD(f) in Small Memory in Othello

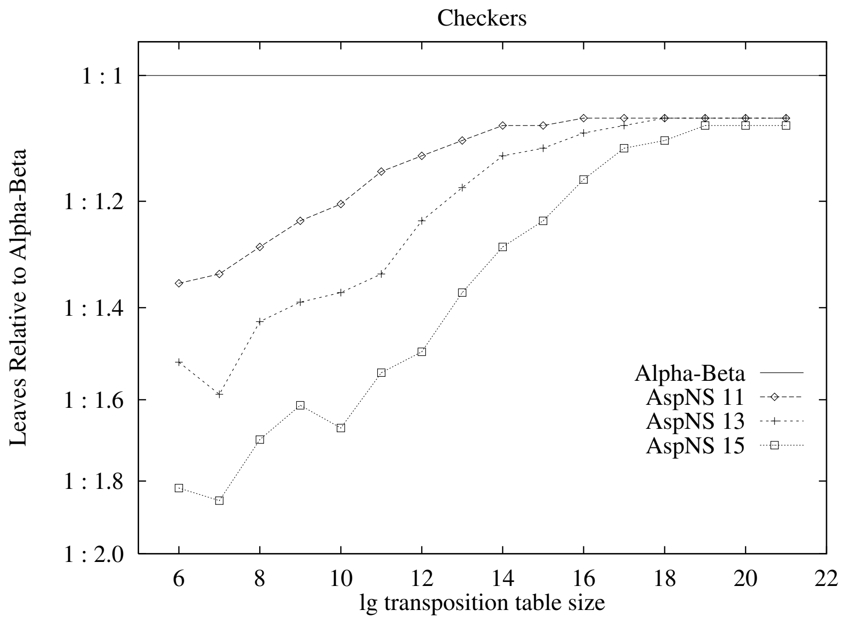


Figure 4.26: Aspiration NegaScout in Small Memory in Checkers

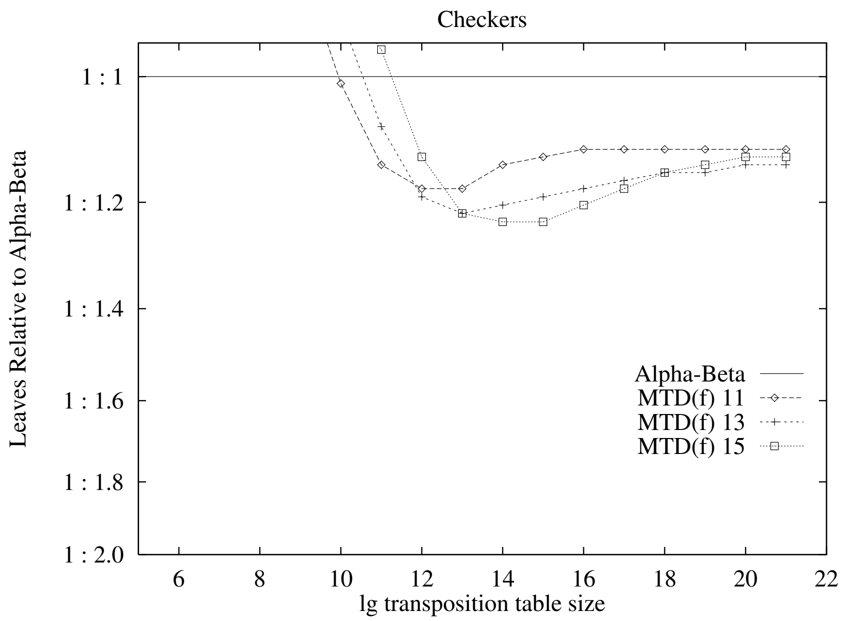


Figure 4.27: MTD(f) in Small Memory in Checkers

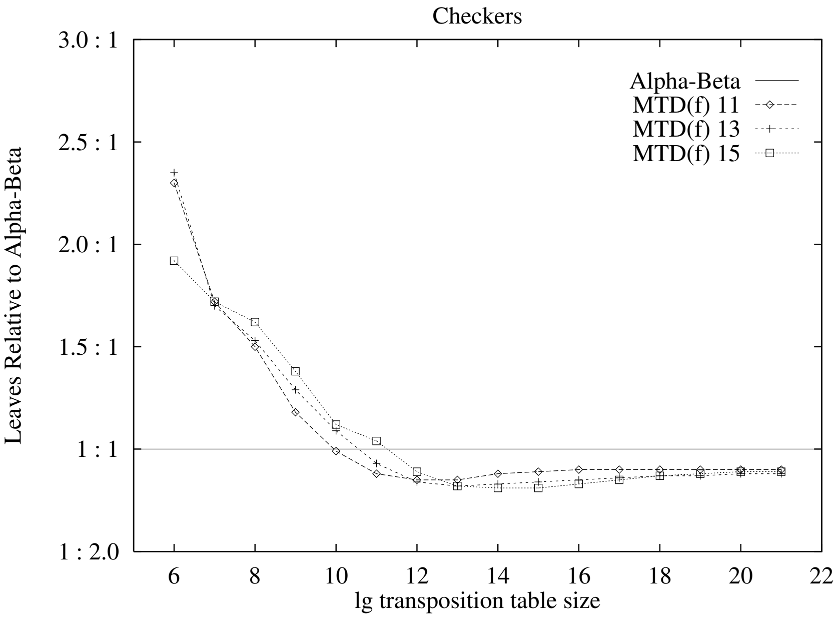


Figure 4.28: MTD(f) in Small Memory (Chinook) II

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

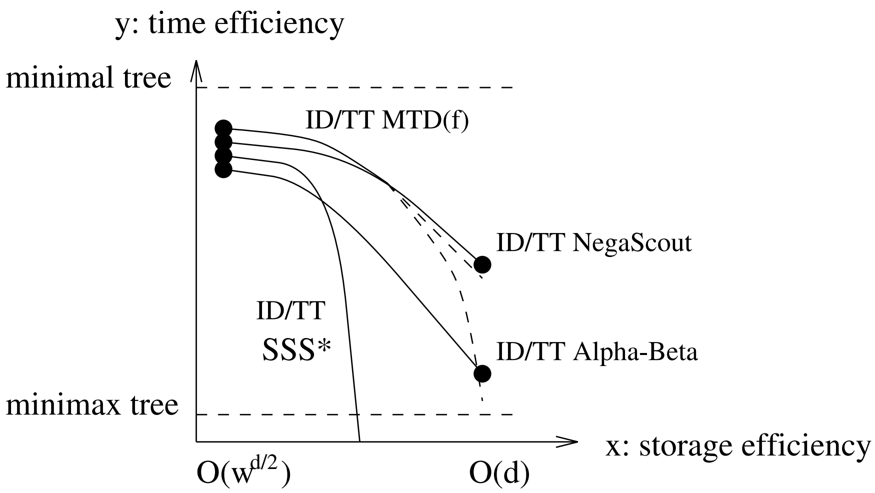


Figure 4.29: Performance Picture of Practical Algorithms

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.

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

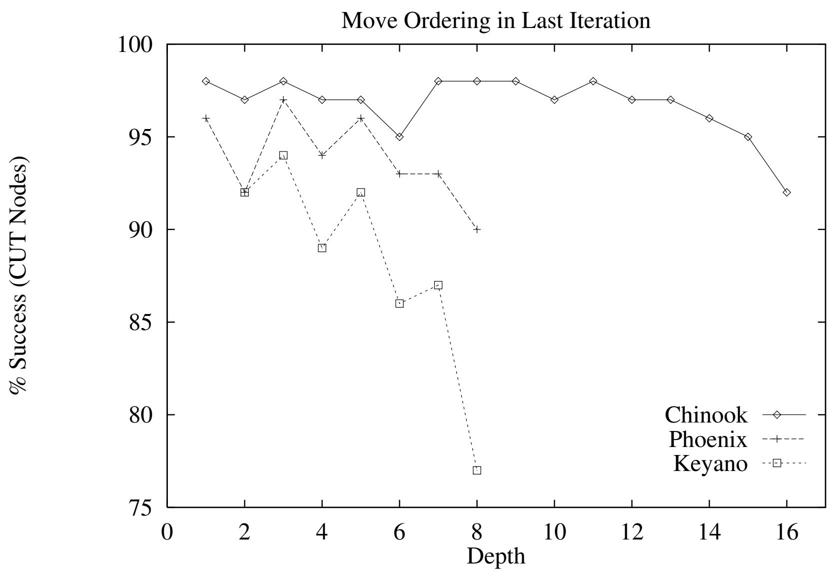


Figure 5.1: Level of Move Ordering by Depth

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

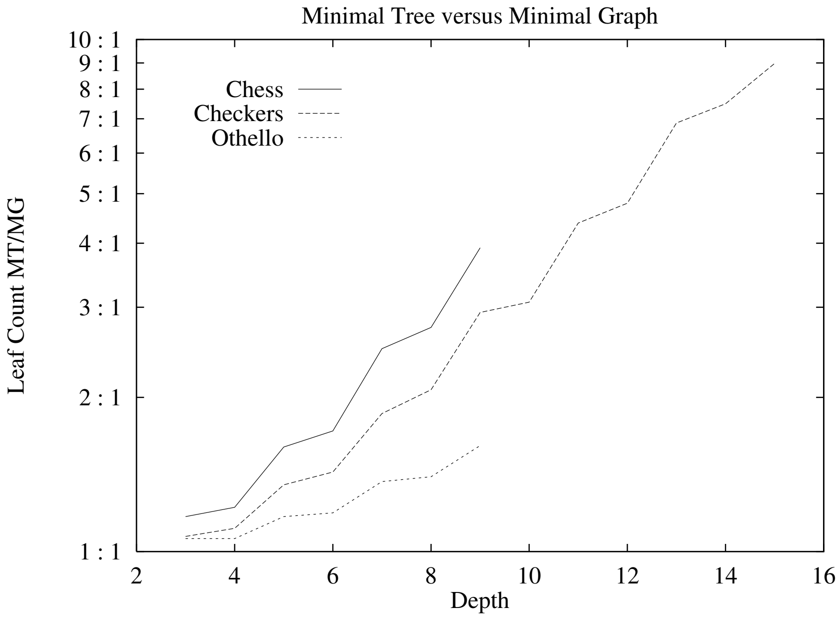


Figure 5.2: Comparing the Minimal Tree and Minimal Graph

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

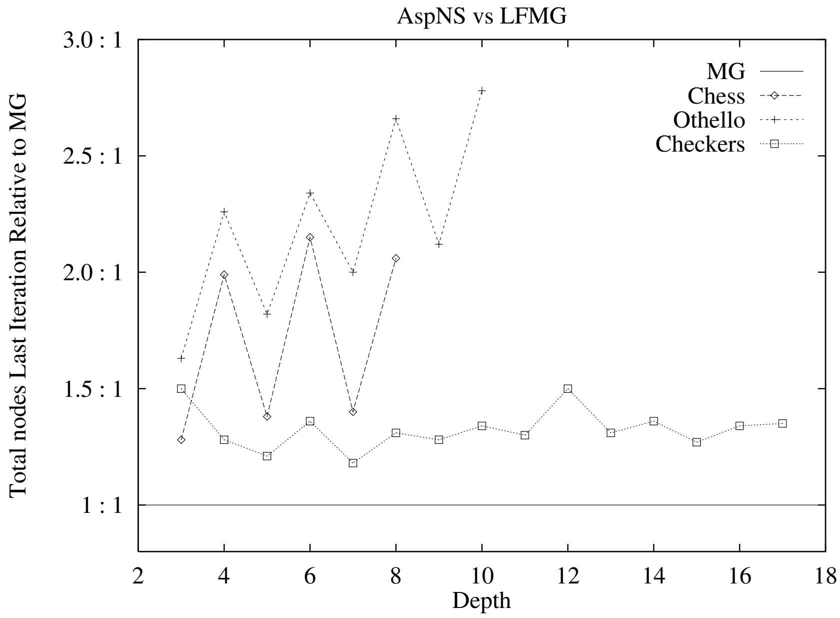


Figure 5.3: Efficiency of Programs Relative to the Minimal Graph

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

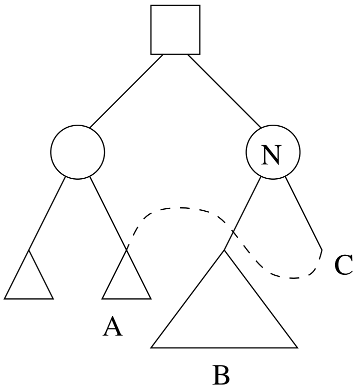


Figure 5.5: Enhanced Transposition Cutoff

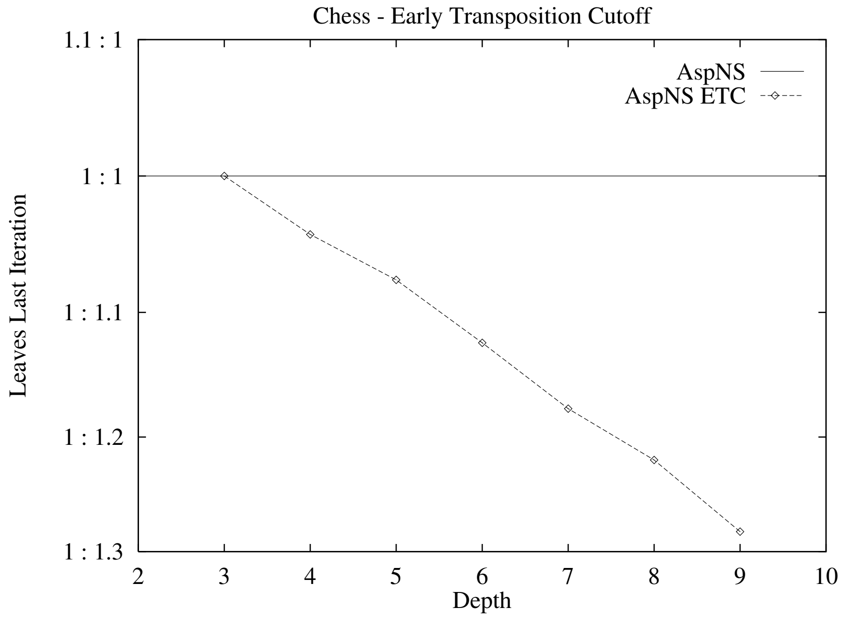


Figure 5.6: Effectiveness of ETC in Chess

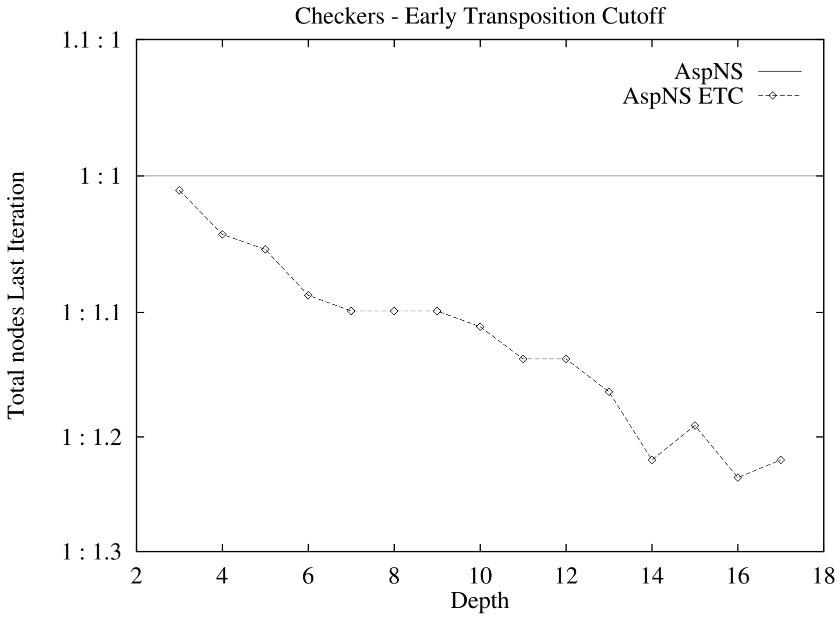


Figure 5.7: Effectiveness of ETC in Checkers

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

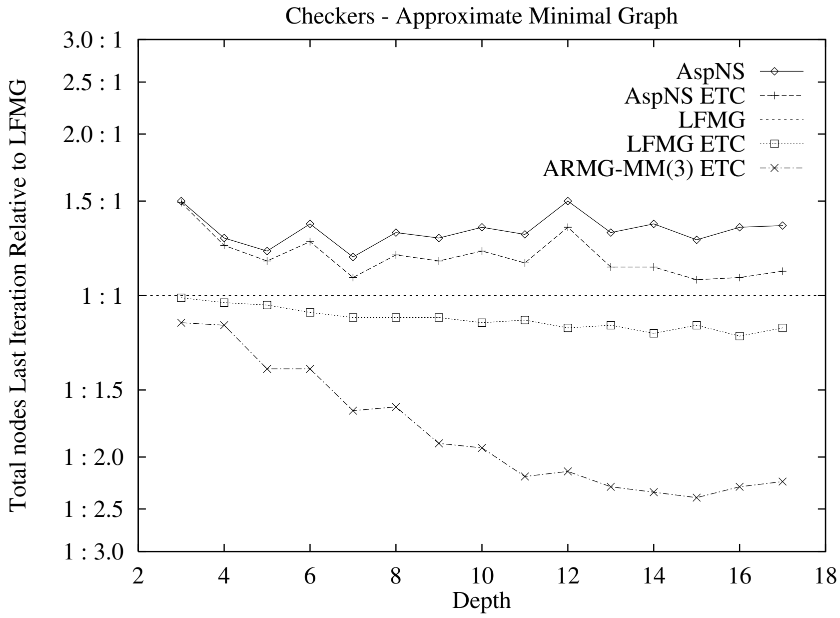


Figure 5.8: LFMG Is Not Minimal in Checkers

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-

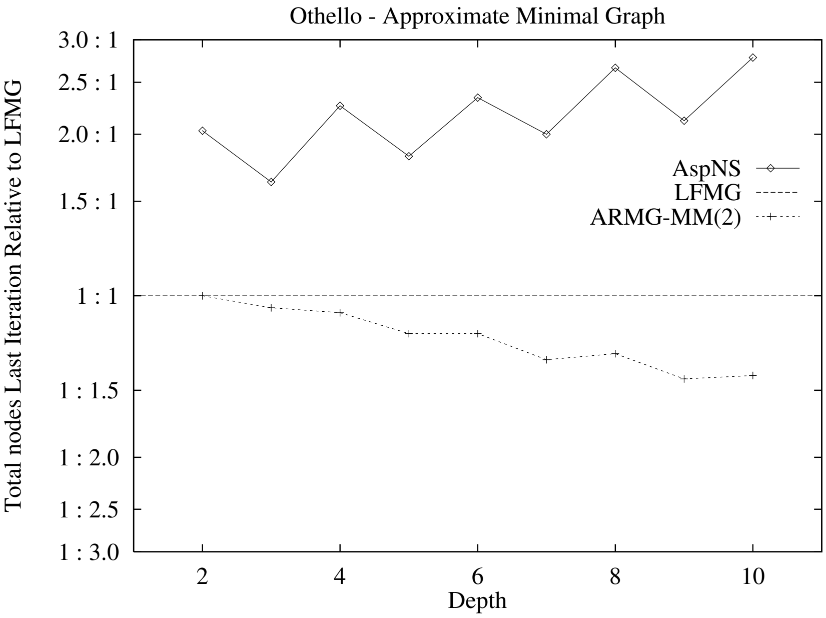


Figure 5.9: LFMG Is Not Minimal in Othello

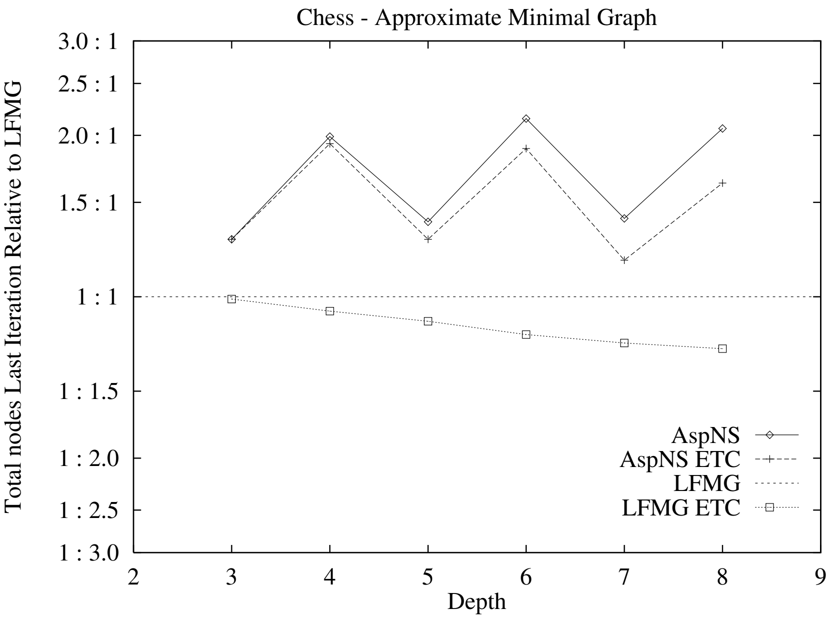


Figure 5.10: LFMG Is Not Minimal in Chess

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

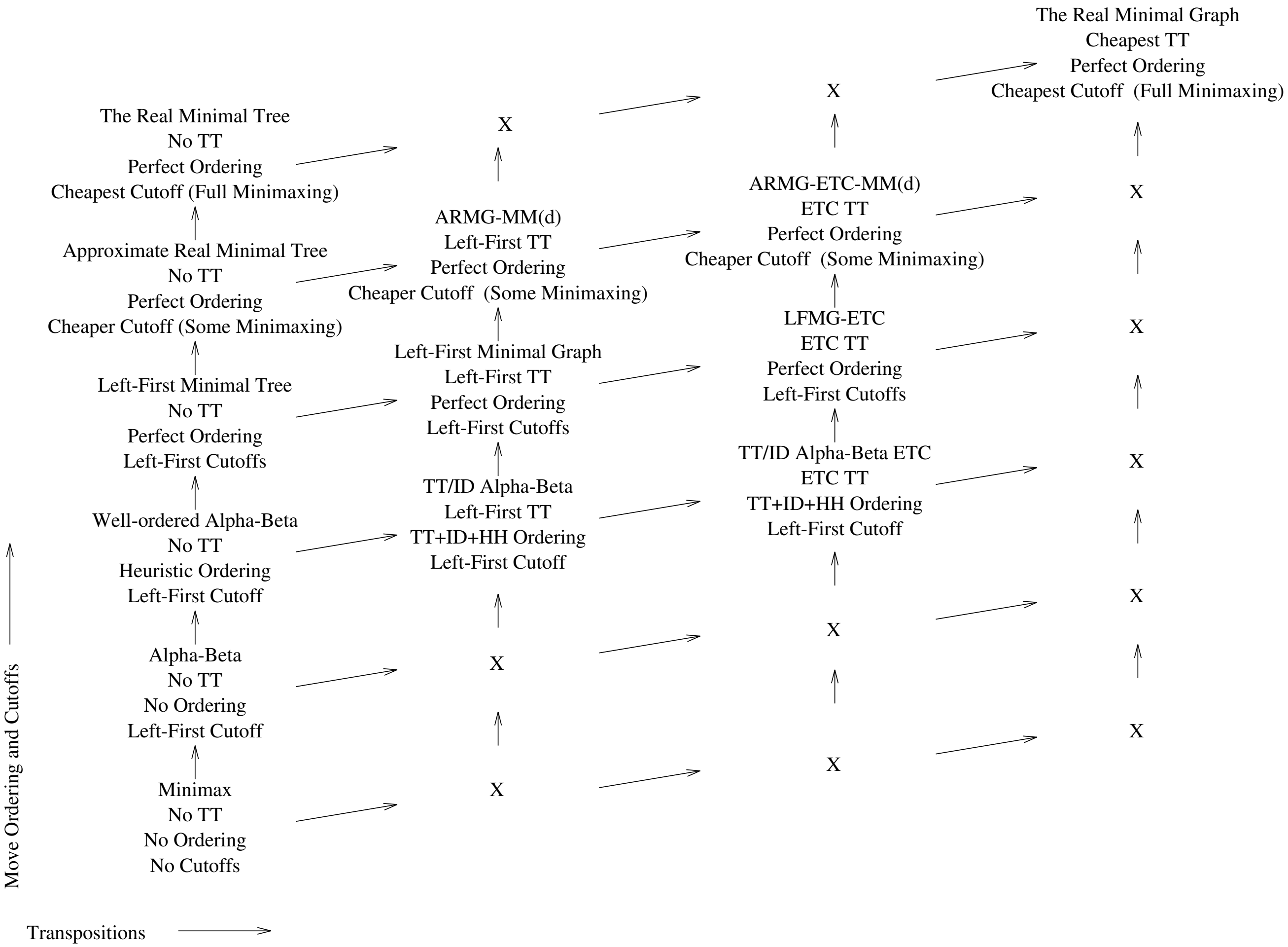


Figure 5.11: A Roadmap towards the Real Minimal Graph

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.

Appendix A

Examples

A.1 Alpha-Beta Example

To give an idea of how Alpha-Beta works, this appendix illustrates how it traverses the tree of figure 2.4 in detail and concept. For convenience, figure A.1 shows the code of the Alpha-Beta function again. The two items of most interest are how and why cutoffs are performed, and seeing how Alpha-Beta constructs the minimal tree that proves f_{root}. In this example Alpha-Beta will find some cutoffs, but it will traverse more than the minimal tree, since the children of some nodes are not ordered best first. (For example, in max node a the left-most child b is not the highest and in min node k the left-most child is not the lowest.) The values for α, β and g as Alpha-Beta traverses them are shown next to the nodes in figure A.2. Children that are cut off are shown as a small black dot.

The table in figure A.3 gives a step-by-step account of Alpha-Beta's progress. Cutoffs are indicated in the table, and explained in the text below. Since Alpha-Beta works towards finding a max and a min solution tree of equal value, we have shown these in the table as well. Recall from the postcondition of Alpha-Beta that if a node returns a g value that lies within the search window, then both a T^+ and a T^- have been traversed. This case applies to most of the nodes. Only nodes j, p and u return a value outside their window. Their value is determined by a single solution tree. Upper bounds for a node are indicated as f^+, lower bounds as f^-. Max solution trees are shown as T^+, min solution trees as T^-. Figure A.4 gives the final minimal tree that proves the minimax value.

The root is called with $\alpha = -\infty$ and $\beta = +\infty$. Its first child b is expanded with the same parameters. The same holds for node c, d and e. Node e is a leaf, which calls the evaluation function. It returns its minimax value of 41 to its parent. Here, at node d, the values of g and β are updated to 41. Node d performs a cutoff check; since $g > \alpha$ (because $41 > -\infty$) the search continues to the next child, f. Since β was updated in its parent, this node is searched with a window of $\langle -\infty, 41 \rangle$. Node f returns its minimax value.

Parent d returns 5, the minimum of 41 and 5. Parent c updates g and α to 5. Node c

```
function alphabeta(n, α, β) → g;
    if n = leaf then return eval(n);
    else if n = max then
        g := −∞;
        c := firstchild(n);
        while g < β and c ≠ ⊥ do
            g := max(g, alphabeta(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(c, α, β));
            β := min(β, g);
            c := nextbrother(c);
    return g;
```

Figure A.1: The Alpha-Beta Function

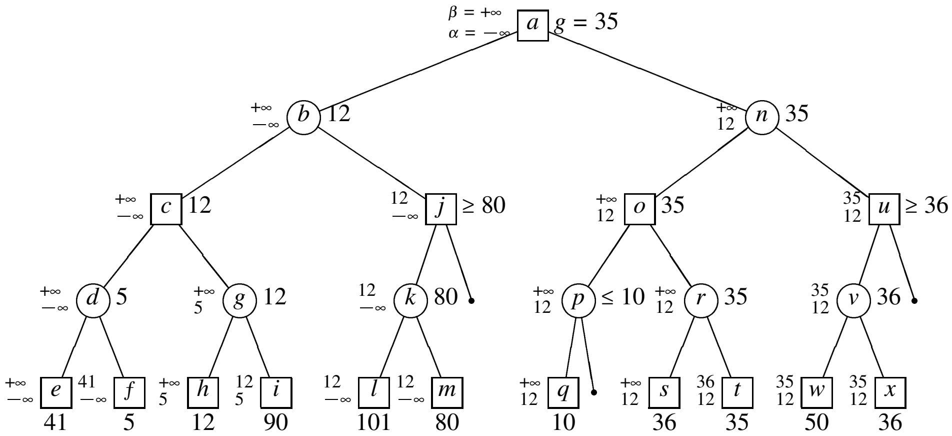


Figure A.2: Example Tree for Alpha-Beta

#	n	α_n	β_n	g_n	cutoff?	f_n^-	T_n^-	f_n^+	T_n^+
1	a	$-\infty$	$+\infty$	$-\infty$		$-\infty$		$+\infty$	
2	b	$-\infty$	$+\infty$	$+\infty$		$-\infty$		$+\infty$	
3	c	$-\infty$	$+\infty$	$-\infty$		$-\infty$		$+\infty$	
4	d	$-\infty$	$+\infty$	$+\infty$		$-\infty$		$+\infty$	
5	e	$-\infty$	$+\infty$	41		41	e	41	e
6	d	$-\infty$	41	41	$41 > -\infty$	$-\infty$		41	d, e
7	f	$-\infty$	41	5		5	f	f	f
8	d	$-\infty$	5	5	$\perp$	5	d, e, f	5	d, f
9	c	5	$+\infty$	5	$5 < +\infty$	5	c, d, e, f	$+\infty$	
10	g	5	$+\infty$	$+\infty$		$-\infty$		$+\infty$	
11	h	5	$+\infty$	12		12	h	12	h
12	g	5	12	12	$12 > 5$	$-\infty$		12	g, h
13	i	5	12	90		90	i	90	i
14	g	5	12	12	$\perp$	12	g, h, i	12	g, h
15	c	12	$+\infty$	12	$\perp$	12	c, g, h, i	12	c, d, f, g, h
16	b	$-\infty$	12	12	$12 > -\infty$	$-\infty$		12	b, c, d, f, g, h
17	j	$-\infty$	12	$-\infty$		$-\infty$		$+\infty$	
18	k	$-\infty$	12	$+\infty$		$-\infty$		$+\infty$	
19	l	$-\infty$	12	101		101	l	101	l
20	k	$-\infty$	12	101	$101 > -\infty$	$-\infty$		101	k, l
21	m	$-\infty$	12	80		80	m	80	m
22	k	$-\infty$	12	80	$\perp$	80	k, l, m	80	k, m
23	j	80	12	80	$80 \not< 12$	80	j, k, l, m	$+\infty$	
24	b	$-\infty$	12	12	$\perp$	12	$b, c, g, h,$ i, j, k, l, m	12	b, c, d, f, g, h
25	a	12	$+\infty$	12	$12 < +\infty$	12	$a, b, c, g,$ h, i, j, k, l, m	$+\infty$	
26	n	12	$+\infty$	$+\infty$		$-\infty$		$+\infty$	
27	o	12	$+\infty$	$-\infty$		$-\infty$		$+\infty$	
28	p	12	$+\infty$	$+\infty$		$-\infty$		$+\infty$	
29	q	12	$+\infty$	10		10	q	10	q
30	p	12	10	10	$10 \not> 12$	$-\infty$		10	p, q
31	o	12	$+\infty$	10	$10 < +\infty$	$-\infty$		$+\infty$	
32	r	12	$+\infty$	$+\infty$		$-\infty$		$+\infty$	
33	s	12	$+\infty$	36		36	s	36	s
34	r	12	36	36	$36 > 12$	$-\infty$		36	r, s
35	t	12	36	35		35	t	35	t
36	r	12	35	35	$\perp$	35	r, s, t	35	r, t
37	o	35	$+\infty$	35	$\perp$	35	o, r, s, t	35	o, p, q, r, t
38	n	12	35	35	$35 > 12$	$-\infty$		35	n, o, p, q, r, t
39	u	12	35	$-\infty$		$-\infty$		$+\infty$	
40	v	12	35	$+\infty$		$-\infty$		$+\infty$	
41	w	12	35	50		50	w	50	w
42	v	12	35	50	$50 > 12$	$-\infty$		50	v, w
43	x	12	35	36		36	x	36	x
44	v	12	35	36	$\perp$	36	v, w, x	36	v, x
45	u	36	35	36	$36 \not< 35$	36	u, v, w, x	$+\infty$	
46	n	12	35	35	$\perp$	35	$n, o, r, s,$ t, u, v, w, x	35	n, o, p, q, r, t
47	a	35	$+\infty$	35	$\perp$	35	$a, n, o, r,$ s, t, u, v, w, x	35	$a, b, c, d, f, g,$ h, n, o, p, q, r, t

Figure A.3: Alpha-Beta Example

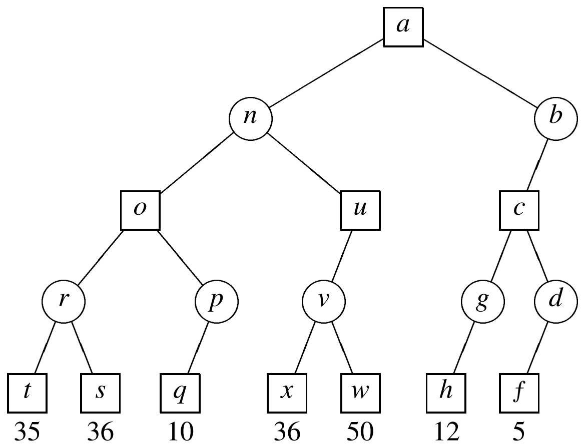


Figure A.4: Minimal Alpha-Beta Tree

continues to search node g, since $5 < +\infty$. The lower bound is used to search node g and its children with the smaller search window of $\langle 5, +\infty\rangle$. Node g returns the minimum of 12 and 90 to c, which returns the maximum of 5 and 12 to b. In node b the search is continued to expand j. Node b is a min node. The g-value 12 is an upper bound, substantiated by a max solution tree containing node c, d, f, g and h. See line 16 in the table. The search window for node j is reduced to $\langle -\infty, 12\rangle$, indicating that parent b already has an upper bound of 12, so that if in any of the children of b a lower bound ≥ 12 appears, the search can be stopped. Node j expands the sub-tree rooted in its child k, which returns 80. This causes a cutoff of its brother in node j, since $80 \not< 12$. The search of node j, whose value, being a max node, is a lower bound of 80 that can only increase, is no longer useful. The value of parent b is already as low as 12, and since it is a min node, it will never increase. Because b has no other children to lower his value further, it returns his value too. The return value of b is defined by min solution tree $b, c, g, h, i, j, k, l, m$ and max solution tree b, c, d, f, g, h.

At the root the value g is updated to the new lower bound of 12. Consequently, α is updated to 12. Since $12 < +\infty$, no cutoff occurs. Searching the sub-tree below n can still increase the g value of the root. Node n is expanded. Node o, p and q are searched with window $\langle 12, +\infty\rangle$. Node q returns 10. At its parent p this causes a cutoff, since $10 \not> 12$. In contrast to the previous cutoff, the reason for it cannot be found by looking at the value of the direct parent, since o does not have a g value yet other than $-\infty$. This kind of cutoff is known as a *deep* cutoff (in contrast to the previous cutoff, which is said to be shallow). It is called deep because the reason for the cutoff lies further away than the direct parent of the node. The sub-tree below b has returned 12, so the value of a, which is a max node, will be at least 12. Node p, a min node, can at most return a value of 10. The value of node q ensures that none of its brothers can ever influence the value of the root. Node p returns 10.

Next nodes r, s, t, u, v, w and x are searched. The sub-tree below v returns 36. This causes a cutoff (one of the shallow kind) in node u, since $36 \not< 35$. Node u returns 36,

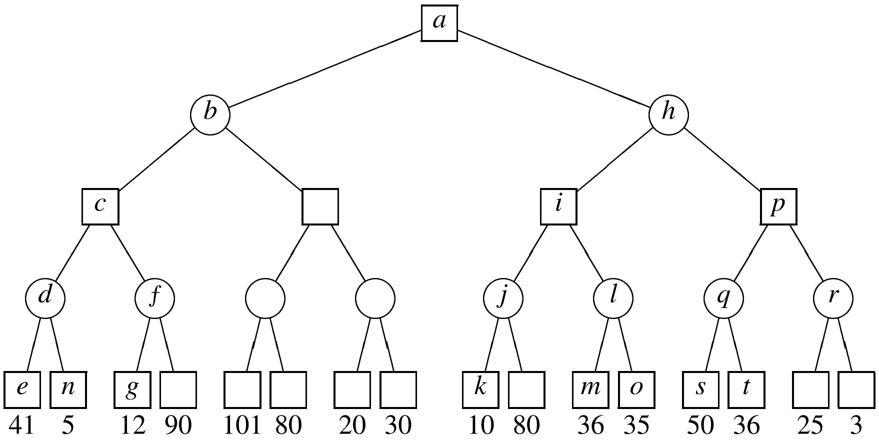


Figure A.5: Example Tree for SSS*

node *n* returns the minimum of 35 and 36, and the root returns the maximum of 12 and 35. The minimax value of the tree has been found, it is 35.

The example illustrates that Alpha-Beta can miss some cutoffs. Although pruning occurs in the example, Alpha-Beta builds a tree that is larger than the minimal tree, as can be seen by comparing figures A.2 and A.4. Alpha-Beta expands 11 leaves, less than the minimax tree of 16, but more than the minimal tree of 7.

A.2 SSS* Example

In the next example we will give an idea of how SSS* works.

Stockman originally used min solution trees to explain his algorithm. Here SSS* is explained using upper bounds and max solution trees, since we think that the algorithm is easier to understand that way and it makes the connection to MT-SSS* easier to see. The two key concepts in the explanation are an *upper bound* on the minimax value, and a *max solution tree*. SSS* starts off with an upper bound of $+\infty$, and works by successively lowering this upper bound until it is equal to the minimax value. The max solution trees are constructed to compute the value of each upper bound.

SSS* works by manipulating a list of nodes, the OPEN list. The nodes have a status associated with them, either *live* (L) or *solved* (S), and a merit, denoted $\hat{h}$. The OPEN list is sorted in descending order, so that the entry with highest merit (the "best" node) is at the front and will be selected for expansion.

We will examine how SSS* searches the tree in figure A.5 for its minimax value (the same tree as in the Alpha-Beta example). Since the tree is almost the same as the one used by Pearl in his explanation of SSS* [99], it may be instructive to compare his min with our max solution tree explanation. In the trees in figures A.6–A.12 the nodes are numbered *a* to *t* in the order in which SSS* visits them first. A number of stages, or passes, can be distinguished in the traversal of this tree. At the end of each pass the OPEN list consists of *solved* nodes only. In appendix B we will see that after

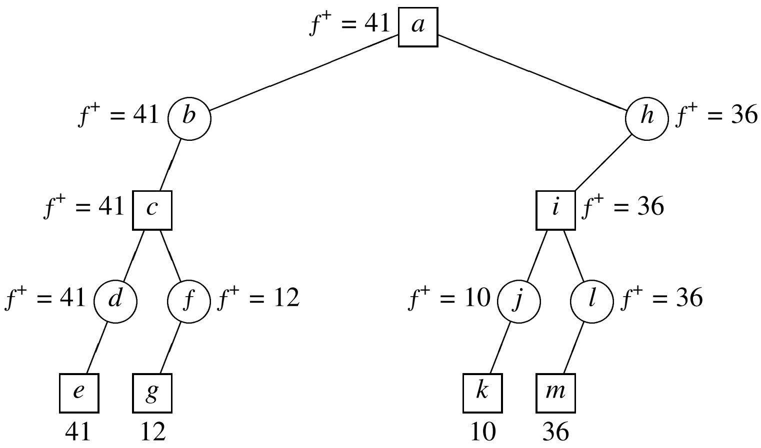


Figure A.6: SSS* Pass 1

Γ operations 1 and 3 SSS* has constructed a min solution tree for the solved node. Appendix B also stresses the fact that at any time the nodes in the OPEN list are the leaves of a (partial) max solution tree rooted at the root, defining an upper bound on it. These two solution trees are shown in the tables. We will start by examining how SSS* traverses the first pass.

First pass: (see figures A.6 and A.7)
In the first pass the left-most max solution tree is expanded to create the first non-trivial upper bound on the minimax value of the root. The code in figure 2.17 places an entry in the OPEN list containing the root node a, with a trivial upper bound: $+\infty$. This entry, containing an internal max node which has as first child a min node, matches Γ case 6, causing a to be replaced by its children (in left-first order). The OPEN list now contains nodes b and h, with a value of $+\infty$. The left-most of these children, b, is at the front of the list. It matches Γ case 5, causing it to be replaced by its left-most child, c. The list now contains nodes c and h. Next, case 6 replaces c by d and f, and case 5 replaces d by e, giving as list nodes e, f and h. Now a new Γ-case comes into play, since node e has no children. Node e does not match case 6, but case 4, causing its state to change from *live* to *solved*, and its $\hat{h}$ value to go from $+\infty$ to 41. Since the list is kept sorted in descending order, the next entry on the list appears at the front, f, the left-most entry with highest $\hat{h}$ value. It matches case 5, g is inserted, which matches case 4, causing it to enter the list with value 12. The list is now $\big(\langle h, L, \infty\rangle, \langle e, S, 41\rangle, \langle g, S, 12\rangle\big)$. Next the right subtree below h is expanded in the same way, through a sequence of Γ cases 5 (node i is visited), 6 (j and l enter the list), 5 (k replaces j), 4 (k gets value 10), 5 (l is replaced by m), and 4 (l gets value 36). The OPEN list is now $\big(\langle e, S, 41\rangle, \langle m, S, 36\rangle, \langle g, S, 12\rangle, \langle k, S, 10\rangle\big)$.

We have seen so far that at max nodes all children were expanded (case 6), while at min nodes only the first child was added to the OPEN list (case 5). Case 4 evaluated the leaf nodes of the tree. Maintaining the list in sorted order guaranteed that the entry with the highest upper bound was at front. Note that the sub-tree expanded thus far is

#	n	Γ	OPEN list after Γ	f^-_{top}	T^-_{top}	f^+_{root}	T^+_{root}
0			$\big(\langle a, L, +\infty\rangle\big)$			$+\infty$	a
1	a	6	$\big(\langle b, L, +\infty\rangle, \langle h, L, +\infty\rangle\big)$	$-\infty$		$+\infty$	a, b, h
2	b	5	$\big(\langle c, L, +\infty\rangle, \langle h, L, +\infty\rangle\big)$	$-\infty$		$+\infty$	a, b, c, h
3	c	6	$\big(\langle d, L, +\infty\rangle, \langle f, L, +\infty\rangle, \langle h, L, +\infty\rangle\big)$	$-\infty$		$+\infty$	a, b, c, d, f, h
4	d	5	$\big(\langle e, L, +\infty\rangle, \langle f, L, +\infty\rangle, \langle h, L, +\infty\rangle\big)$	$-\infty$		$+\infty$	a, b, c, d, e, f, h
5	e	4	$\big(\langle f, L, +\infty\rangle, \langle h, L, +\infty\rangle, \langle e, S, 41\rangle\big)$	$-\infty$		$+\infty$	a, b, c, d, e, f, h
6	f	5	$\big(\langle g, L, +\infty\rangle, \langle h, L, +\infty\rangle, \langle e, S, 41\rangle\big)$	$-\infty$		$+\infty$	a, b, c, d, e, f, g, h
7	g	4	$\big(\langle h, L, +\infty\rangle, \langle e, S, 41\rangle, \langle g, S, 12\rangle\big)$	$-\infty$		$+\infty$	a, b, c, d, e, f, g, h
8	h	5	$\big(\langle i, L, +\infty\rangle, \langle e, S, 41\rangle, \langle g, S, 12\rangle\big)$	$-\infty$		$+\infty$	$a, b, c, d,$ e, f, g, h, i
9	i	6	$\big(\langle j, L, +\infty\rangle, \langle l, L, +\infty\rangle,$ $\langle e, S, 41\rangle, \langle g, S, 12\rangle\big)$	$-\infty$		$+\infty$	$a, b, c, d, e,$ f, g, h, i, j, l
10	j	5	$\big(\langle k, L, +\infty\rangle, \langle l, L, +\infty\rangle,$ $\langle e, S, 41\rangle, \langle g, S, 12\rangle\big)$	$-\infty$		$+\infty$	$a, b, c, d, e,$ f, g, h, i, j, k, l
11	k	4	$\big(\langle l, L, +\infty\rangle, \langle e, S, 41\rangle,$ $\langle g, S, 12\rangle, \langle k, S, 10\rangle\big)$	$-\infty$		$+\infty$	$a, b, c, d, e,$ f, g, h, i, j, k, l
12	l	5	$\big(\langle m, L, +\infty\rangle, \langle e, S, 41\rangle,$ $\langle g, S, 12\rangle, \langle k, S, 10\rangle\big)$	$-\infty$		$+\infty$	$a, b, c, d, e,$ f, g, h, i, j, k, l, m
13	m	4	$\big(\langle e, S, 41\rangle, \langle m, S, 36\rangle,$ $\langle g, S, 12\rangle, \langle k, S, 10\rangle\big)$	41	e	41	$a, b, c, d, e,$ f, g, h, i, j, k, l, m

Figure A.7: SSS* Table Pass 1

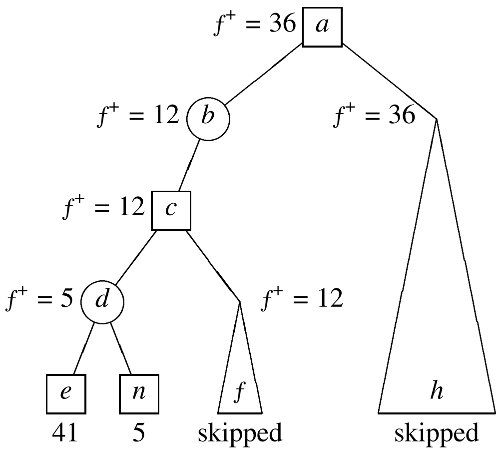


Figure A.8: SSS* Pass 2

#	n	Γ	OPEN list after Γ	f^-_{top}	T^-_{top}	f^+_{root}	T^+_{root}
14	e	2	$\big(\langle n, L, 41\rangle, \langle m, S, 36\rangle, \langle g, S, 12\rangle, \langle k, S, 10\rangle\big)$	$-\infty$		41	$a, b, c, d, e, f, g, h, i, j, k, l, m$
15	n	4	$\big(\langle m, S, 36\rangle, \langle g, S, 12\rangle, \langle k, S, 10\rangle, \langle n, S, 5\rangle\big)$	36	m	36	$a, b, c, d, e, n, f, g, h, i, j, k, l, m$

Figure A.9: SSS* Table Pass 2

a max solution tree (compare figure A.6 to figure 2.6). The minimax value of this tree is 41, which is also the $\hat{h}$ value of the first entry of the OPEN list.

Second pass: (see figures A.8 and A.9)
In the second pass, SSS* will try to lower the upper bound of 41 to come closer to f. The next upper bound will be computed by expanding a brother of the critical leaf e. The critical leaf has a min parent, node d, so expanding this brother can lower d's value, which will, in turn, lower the minimax value at the root of the max solution tree. Since this value is the maximum of its leaves, there is no point in expanding brothers of non-critical leaves, since then node e will keep the value of the root at 41. Thus, node n is in a sense the best node to expand. The entry for node e matches Γ case 2, which replaces e by the brother n, giving it state *live*, and the value 41, the sharpest (lowest) upper bound of the previous pass. The n entry matches Γ case 4. Case 4 evaluates the leaf, and assigns to $\hat{h}$ either this value (5), or the sharpest upper bound so far, if that happens to be lower. Node n gets value 5. In general, Γ case 4 performs the minimizing operation of the minimax function, ensuring that the $\hat{h}$ of the first (highest) entry of the OPEN list will always be the sharpest upper bound on the minimax value of the root, based on the previously expanded nodes. The OPEN list has become $\big(\langle m, S, 36\rangle, \langle g, S, 12\rangle, \langle k, S, 10\rangle, \langle n, S, 5\rangle\big)$. Thus, the upper bound on the root has been lowered to 36. Its value is determined by a new (sharper) max solution tree,

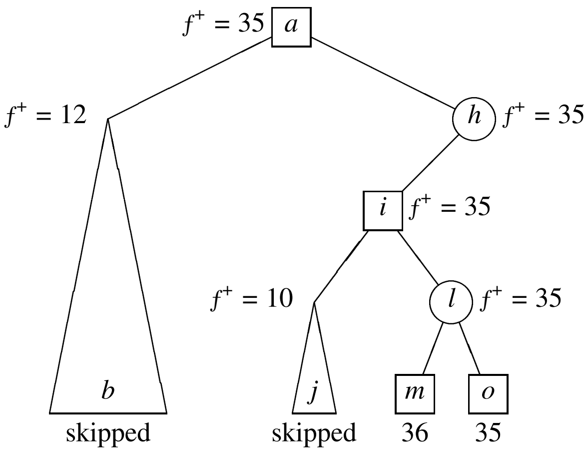


Figure A.10: SSS* Pass 3

#	n	Γ	OPEN list after Γ	f^-_{top}	T^-_{top}	f^+_{root}	T^+_{root}
16	m	2	$\big(\langle o,L,36\rangle, \langle g,S,12\rangle,$	$-\infty$		36	$a,b,c,d,e,n,f,g,$
			$\langle k,S,10\rangle, \langle n,S,5\rangle\big)$				h,i,j,k,l,m,o
17	o	4	$\big(\langle o,S,35\rangle, \langle g,S,12\rangle,$	35	o	35	$a,b,c,d,e,n,f,g,$
			$\langle k,S,10\rangle, \langle n,S,5\rangle\big)$				h,i,j,k,l,m,o

Figure A.11: SSS* Table Pass 3

whose leaves are contained in the OPEN list.

Third Pass: (see figures A.10 and A.11)
In the third pass, the goal of the search is to get the upper bound below 36. Just as in the second pass, the first entry of the OPEN list, m, matches Γ case 2, and its brother is inserted. It matches Γ case 4, so it gets evaluated. The new brother is node o, with value 35. Again, a sharper upper bound has been found. The new OPEN list is $\big(\langle o,S,35\rangle, \langle g,S,12\rangle, \langle k,S,10\rangle, \langle n,S,5\rangle\big)$.

Fourth Pass: (see figures A.12 and A.13)
The previous search lowered the bound from 36 to 35. In the fourth pass the first entry has no immediate brother. It matches a Γ case that is used to backtrack, case 3, which replaces node o by its parent l. Case 3 is always followed by case 1, which replaces l by its parent i and, in addition, deletes all child-entries from the list—only node k in this case. Each time case 1 applies, all children of a min node and its max parent have been expanded and the search of the subtree has been completed. To avoid having old nodes interfere with the remainder of the search, they must be removed from the OPEN list. The list now contains: $\big(\langle i,S,35\rangle, \langle g,S,12\rangle, \langle n,S,5\rangle\big)$. Next, case 2 matches entry i, and expansion of the brother of i commences. Node p is inserted into the list with state *live*. It matches case 6, which inserts q and r into the list. Node q matches case 5,

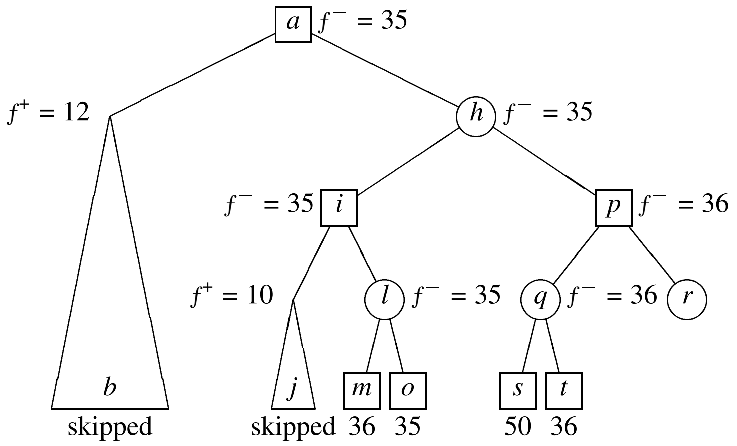


Figure A.12: SSS* Pass 4

#	n	Γ	OPEN list after Γ	f^-_{top}	T^-_{top}	f^+_{root}	T^+_{root}
18	o	3	$(\langle l,S,35\rangle, \langle g,S,12\rangle, \langle k,S,10\rangle, \langle n,S,5\rangle)$	35	l,m,o	35	$a,b,c,d,e,n,f,g,h,i,j,k,l,m,o$
19	l	1	$(\langle i,S,35\rangle, \langle g,S,12\rangle, \langle n,S,5\rangle)$	35	i,l,m,o	35	$a,b,c,d,e,n,f,g,h,i,j,k,l,m,o$
20	i	2	$(\langle p,L,35\rangle, \langle g,S,12\rangle, \langle n,S,5\rangle)$	$-\infty$		35	$a,b,c,d,e,n,f,g,h,i,j,k,l,m,o,p$
21	p	6	$(\langle q,L,35\rangle, \langle r,L,35\rangle, \langle g,S,12\rangle, \langle n,S,5\rangle)$	$-\infty$		35	$a,b,c,d,e,n,f,g,h,i,j,k,l,m,o,p,q,r$
22	q	5	$(\langle s,L,35\rangle, \langle r,L,35\rangle, \langle g,S,12\rangle, \langle n,S,5\rangle)$	$-\infty$		35	$a,b,c,d,e,n,f,g,h,i,j,k,l,m,o,p,q,r,s$
23	s	4	$(\langle s,S,35\rangle, \langle r,L,35\rangle, \langle g,S,12\rangle, \langle n,S,5\rangle)$	50	s	35	$a,b,c,d,e,n,f,g,h,i,j,k,l,m,o,p,q,r,s$
24	s	2	$(\langle t,L,35\rangle, \langle r,L,35\rangle, \langle g,S,12\rangle, \langle n,S,5\rangle)$	$-\infty$		35	$a,b,c,d,e,n,f,g,h,i,j,k,l,m,o,p,q,r,s,t$
25	t	4	$(\langle t,S,35\rangle, \langle r,L,35\rangle, \langle g,S,12\rangle, \langle n,S,5\rangle)$	36	t	35	$a,b,c,d,e,n,f,g,h,i,j,k,l,m,o,p,q,r,s,t$
26	t	3	$(\langle q,S,35\rangle, \langle r,L,35\rangle, \langle g,S,12\rangle, \langle n,S,5\rangle)$	36	q,s,t	35	$a,b,c,d,e,n,f,g,h,i,j,k,l,m,o,p,q,r,s,t$
27	q	1	$(\langle p,S,35\rangle, \langle g,S,12\rangle, \langle n,S,5\rangle)$	35	p,q,s,t	35	$a,b,c,d,e,n,f,g,h,i,j,k,l,m,o,p,q,r,s,t$
28	p	3	$(\langle h,S,35\rangle, \langle g,S,12\rangle, \langle n,S,5\rangle)$	35	h,i,l,m,o,p,q,s,t	35	$a,b,c,d,e,n,f,g,h,i,j,k,l,m,o,p,q,r,s,t$
29	h	1	$(\langle a,S,35\rangle)$	35	a,h,i,l,m,o,p,q,s,t	35	$a,b,c,d,e,n,f,g,h,i,j,k,l,m,o,p,q,r,s,t$

Figure A.13: SSS* Table Pass 4

which inserts its left-most child s, still with $\hat{h}$ value 35. This leaf is then evaluated by case 4. The evaluation of 50 is not less than the sharpest upper bound of 35, so $\hat{h}$ is not changed. The OPEN list is now: $(\langle s,S,35\rangle, \langle r,L,35\rangle, \langle g,S,12\rangle, \langle n,S,5\rangle)$. Node s is a max node with a brother. It matches case 2, which replaces s by its brother t. Node t is evaluated to value 36 by case 4, which again does not lower the sharpest upper bound of 35. The OPEN list is now: $(\langle t,S,35\rangle, \langle r,L,35\rangle, \langle g,S,12\rangle, \langle n,S,5\rangle)$. Node t matches case 3, which is followed by case 1, inserting p and purging the OPEN list of entry r. The list is now: $(\langle p,S,35\rangle, \langle g,S,12\rangle, \langle n,S,5\rangle)$. Since max node p has no brothers, case 3 applies, which is followed by case 1. Case 1 inserts the root a into the list, and purges the list of all the children of a. The list now becomes the single *solved* entry $\langle a,S,35\rangle$, which satisfies the termination condition of SSS*. The minimax value is 35.

A.3 MT-SSS* Example

Now we will use the tree from the previous examples to show how the reformulation of SSS* works.

SSS* finds f_{root} by determining a sequence of upper bounds on it. The idea behind MT-SSS* is that these upper bounds can also be found using a null-window Alpha-Beta call. The null-window call creates the solution tree. This solution tree is stored in memory, so that it can be refined in later passes. It turns out that in this way Alpha-Beta will expand the same solution trees as SSS*. We will show how this works in detail using an example. Comparing it to the SSS* example illustrates that both formulations expand the same trees. (In appendix B the equivalence of the two formulations is discussed in some detail.)

Alpha-Beta is used to construct solution trees. The postcondition of the Alpha-Beta procedure in section 2.1.3 suggests that using outcome 2, we can have it return an upper bound if we make it fail low. To create a fail low, Alpha-Beta must be called with a search window greater than any possible leaf node value. Alpha-Beta, when called with such a window, will find the same upper bound, and expand the same max solution tree, as SSS*. This can be seen intuitively because both Alpha-Beta and SSS* expand the children of a node in a left-to-right order.

The example tree in figure A.2 is searched to determine its minimax value. A number of stages, or passes, can be distinguished in the traversal of this tree. At the end of each pass a full max solution tree exists, which determines a better upper bound on the minimax value. Also, solved nodes represent min solution trees. These two solution trees are shown in the tables. Note that in the figures the nodes are still numbered a to t in the order in which SSS* visits them first, not MT-SSS*.

First pass: (see figures A.14 and A.15)
In the first pass the left-most max solution tree is expanded to create the first non-trivial upper bound on the minimax value of the root. SSS* builds the max solution tree shown in figure A.14, using cases 4, 5, and 6 of the Γ operator. Instead of using Γ cases 4, 5 and 6 and a sorted OPEN list, MT-SSS* uses MT to compute the bound, by traversing

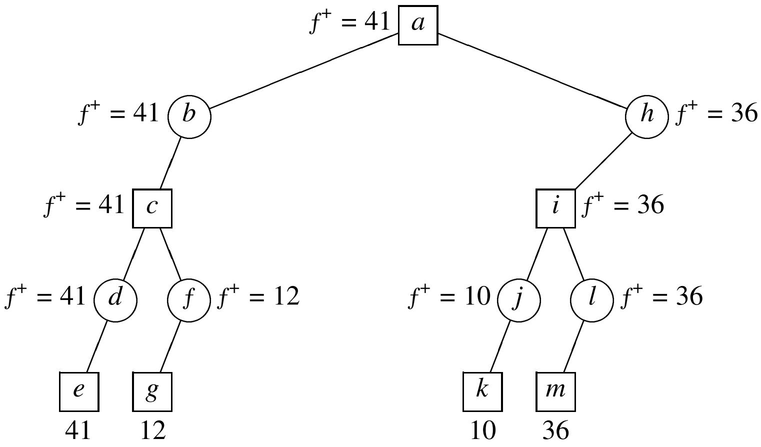


Figure A.14: MT-SSS* Pass 1

#	n	γ_n	g_n	cutoff?	f_n^-	T_n^-	f_n^+	T_n^+
1	a	$+\infty$	$-\infty$		$-\infty$		$+\infty$	
2	b	$+\infty$	$+\infty$		$-\infty$		$+\infty$	
3	c	$+\infty$	$-\infty$		$-\infty$		$+\infty$	
4	d	$+\infty$	$+\infty$		$-\infty$		$+\infty$	
5	e	$+\infty$	41		$-\infty$	e	41	e
6	d	$+\infty$	41	$41 \not\geq +\infty$	$-\infty$		41	d, e
7	c	$+\infty$	41	$41 < +\infty$	$-\infty$		$+\infty$	
8	f	$+\infty$	$+\infty$		$-\infty$		$+\infty$	
9	g	$+\infty$	12		$-\infty$	g	12	g
10	f	$+\infty$	12	$12 \not\geq +\infty$	$-\infty$		12	f, g
11	c	$+\infty$	41	$\perp$	$-\infty$		41	c, d, e, f, g
12	b	$+\infty$	41	$41 \not\geq +\infty$	$-\infty$		41	b, c, d, e, f, g
13	a	$+\infty$	41	$41 < +\infty$	$-\infty$		$+\infty$	
14	h	$+\infty$	$+\infty$		$-\infty$		$+\infty$	
15	i	$+\infty$	$-\infty$		$-\infty$		$+\infty$	
16	j	$+\infty$	$+\infty$		$-\infty$		$+\infty$	
17	k	$+\infty$	10		$-\infty$	k	10	k
18	j	$+\infty$	10	$10 \not\geq +\infty$	$-\infty$		10	j, k
19	i	$+\infty$	10	$10 < +\infty$	$-\infty$		$+\infty$	
20	l	$+\infty$	$+\infty$		$-\infty$		$+\infty$	
21	m	$+\infty$	36		$-\infty$	m	36	m
22	l	$+\infty$	36	$36 \not\geq +\infty$	$-\infty$		36	l, m
23	i	$+\infty$	36	$\perp$	$-\infty$		36	i, j, k, l, m
24	h	$+\infty$	36	$36 \not\geq +\infty$	$-\infty$		36	h, i, j, k, l, m
25	a	$+\infty$	41	$\perp$	$-\infty$		41	$a, b, c, d, e, f, g, h, i, j, k, l, m$

Figure A.15: MT-SSS* Table Pass 1

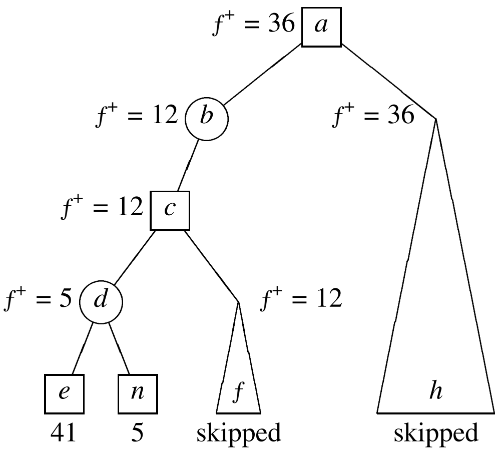


Figure A.16: MT-SSS* Pass 2

#	n	γ_n	g_n	cutoff?	f_n^-	T_n^-	f_n^+	T_n^+
26	a	41	$-\infty$		$-\infty$		41	$a,b,c,d,e,f,g,h,i,j,k,l,m$
27	b	41	$+\infty$		$-\infty$		41	b,c,d,e,f,g
28	c	41	$-\infty$		$-\infty$		41	c,d,e,f,g
29	d	41	$+\infty$		$-\infty$		41	d,e
30	e	41	41	$41 \not\geq +\infty$	41	e	$+\infty$	e
31	n	41	5		$-\infty$	n	5	n
32	d	41	5	$\perp$	$-\infty$	d,e,n	5	d,n
33	c	41	5	$5 < 41$	$-\infty$		$+\infty$	
34	f	41	12	$12 \not\geq 41$	$-\infty$		12	f,g
35	c	41	12	$\perp$	$-\infty$	c,d,e,n	12	c,d,n,f,g
36	b	41	12	$12 \not\geq 41$	$-\infty$		12	b,c,d,n,f,g
37	a	41	12	$12 < 41$	$-\infty$		12	$a,b,c,d,n,f,g,h,i,j,k,l,m$
38	h	41	36	$36 \not\geq 41$	$-\infty$		36	h,i,j,k,l,m
39	a	41	36	$\perp$	$-\infty$		36	$a,b,c,d,n,f,g,h,i,j,k,l,m$

Figure A.17: MT-SSS* Table Pass 2

solution trees. At unexpanded nodes, f^- is $-\infty$, and f^+ is $+\infty$. A call MT(G,∞) will cause an alpha cutoff at all min nodes, since all internal calls return values $g < \gamma = \infty$. No beta cutoffs at max nodes will occur, since all $g < \gamma$. We see that the call MT(a,∞) on the tree in figure A.2 will traverse the tree in figure A.14, conforming to Alpha-Beta's postcondition. Due to the "store" operation in figure 3.2, this tree is saved in memory so that its backed-up values can be used in a later pass. The max solution tree stored at the end of this pass consists of the nodes $a, b, c, d, e, f, g, h, i, j, k, l$ and m, yielding an upper bound of 41.

Second pass: (see figures A.16 and A.17)
This pass lowers the upper bound on f from 41 to 36. How can we use Alpha-Beta to do this? Since the max solution tree defining the upper bound of 41 has been stored by the previous MT call, Alpha-Beta can re-traverse the nodes on the principal variation

(a, b, c, d, e) to find the critical leaf e, and see whether expanding its brother will yield a search tree with a lower minimax value. Finding this critical leaf, and selecting its brother for expansion is the essence of the "best-first" behavior of SSS* (and MT-SSS*). The critical leaf e has a min parent, node d, so expanding the brother can lower its value, which will, in turn, lower the minimax value at the root of the max solution tree. Since this value is the maximum of its leaves, there is no point in expanding brothers of non-critical leaves, because then node e will keep the value of the root at 41. Thus, based on the information that the max solution tree provides, there is only one node (n) whose expansion makes sense. Other nodes are worse, since they cannot change the bound at the root.

To give Alpha-Beta the task of returning a value lower than $f^+ = 41$, we give it a search window which will cause it to fail low. The old window of $\langle \infty - 1, \infty \rangle$ will not do, since the code in figure 3.2 will cause it to return from both nodes b and h, with a value of 41, lower than $+\infty$, but not the lower upper bound. A better choice is the search window $\langle f^+ - 1, f^+ \rangle$, or $\langle 40, 41 \rangle$, which prompts MT to descend the principal variation and return as soon as a lower f^+ on node a is found. It descends to nodes b, c, d, e and continues to search node n. It will back up value 5 to node d and cause a cutoff. The value of d is no longer determined by e but by n. Node e is no longer part of the max solution tree that determines the sharpest upper bound. It has been proven that e can be erased from memory as long as we remember that n is the new best child (not shown in the MT code). The value 5 is backed up to c. No beta cutoff occurs at c, so f's bound is retrieved. Since $f^+ \not\geq \gamma$ at node f, MT does not enter it, but uses $c.f^+ = 12$ for g'. 12 is backed up to b, where it causes an alpha cutoff. Next, 12 is backed up to a. Since $g < \gamma$, node h is probed, but since $c.f^+ \not\geq \gamma$ $(36 \not\geq 41)$ it is not entered. The call MT$(a, 41)$ fails low with value 36, the sharper upper bound. The max solution tree defining this bound consists of nodes $a, b, c, d, n, f, g, h, i, j, k, l$ and m (that is, node e has been replaced with n).

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

Third Pass: (see figures A.18 and A.19)
In the previous pass, the upper bound was lowered from 41 to 36. A call MT$(a, 36)$ is performed. From the previous search, we know that b has an $f^+ \not\geq 36$ so it is not entered; $h.f^+ \geq 36$, so it is. The algorithm follows the principal variation to the node giving the 36 (h to i to l to m). The brother of m is expanded. The bound on the minimax value at the root has now been improved from 36 to 35. The max solution tree defining this bound consists of nodes $a, b, c, d, n, f, g, h, i, j, k, l$ and o.

Fourth Pass: (see figures A.20 and A.21)
This is the last pass of MT-SSS*. We will find that the upper bound cannot be lowered. A call with window $\langle f^+ - 1, f^+ \rangle$, or MT$(a, 35)$, is performed. In this pass we will not

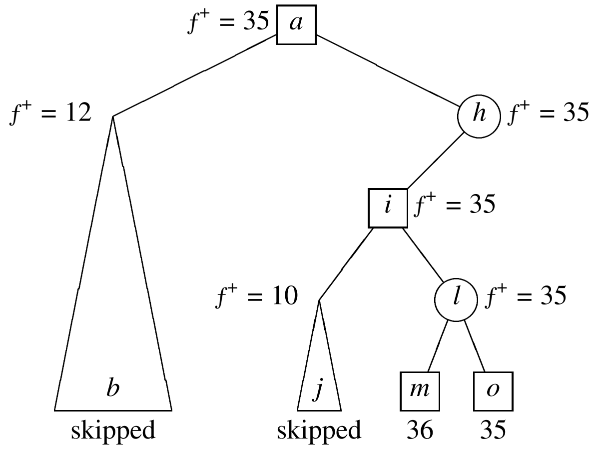


Figure A.18: MT-SSS* Pass 3

#	n	γ_n	g_n	cutoff?	f_n^-	T_n^-	f_n^+	T_n^+
40	a	36	$-\infty$		$-\infty$		36	$a, b, c, d, n, f, g, h, i, j, k, l, m$
41	b	36	12	$12 \not\geq 36$	$-\infty$		12	b, c, d, n, f, g
42	h	36	$+\infty$		$-\infty$		36	h, i, j, k, l, m
43	i	36	$-\infty$		$-\infty$		36	i, j, k, l, m
44	j	36	10	$10 \not\geq 36$	$-\infty$		10	j, k
45	l	36	$+\infty$		$-\infty$		36	l, m
46	m	36	36	$36 \neq +\infty$	36	m	$+\infty$	m
47	o	36	35		$-\infty$	o	35	o
48	l	36	35	$\perp$	$-\infty$	l, m, o	35	l, o
49	i	36	35	$\perp$	$-\infty$	i, l, m, o	35	i, j, k, l, o
50	h	36	35	$35 \not\geq 36$	$-\infty$		35	h, i, j, k, l, o
51	a	36	35	$\perp$	$-\infty$		35	$a, b, c, d, n, f, g, h, i, j, k, l, o$

Figure A.19: MT-SSS* Table Pass 3

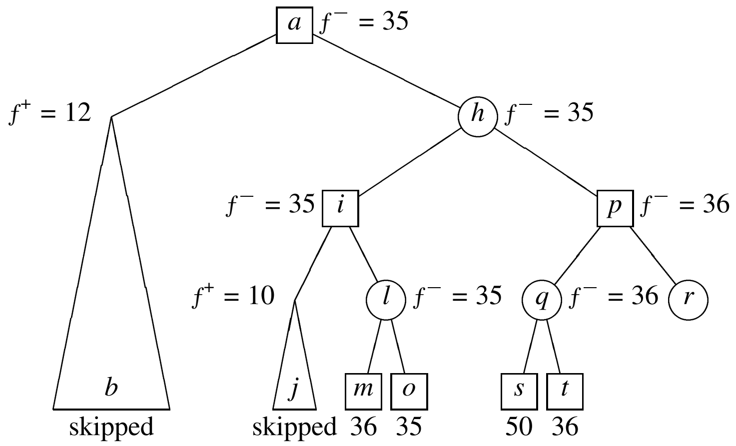


Figure A.20: MT-SSS* Pass 4

#	n	γ_n	g_n	cutoff?	f_n^-	T_n^-	f_n^+	T_n^+
52	a	35	$-\infty$		$-\infty$		35	$a,b,c,d,n,f,$ g,h,i,j,k,l,o
53	b	35	12	$12 \not\geq 35$	$-\infty$		12	b,c,d,e,f,g
54	h	35	$+\infty$		$-\infty$		35	h,i,j,k,l,o
55	i	35	$-\infty$		$-\infty$	i,l,m,o	35	i,j,k,l,o
56	j	35	34	$34 \not\geq 35$	$-\infty$		34	j,k
57	l	35	$+\infty$		$-\infty$		36	l,m
58	m	35	36	$36 \not< 35$	36	m	$+\infty$	m
59	o	35	35	$35 \not= +\infty$	35	o	$+\infty$	o
60	l	35	35	$\perp$	35	l,m,o	$+\infty$	l,o
61	i	35	35	$\perp$	35	i,l,m,o	$+\infty$	i,j,k,l,o
62	p	35	$-\infty$		$-\infty$		$+\infty$	
63	q	35	$+\infty$		$-\infty$		$+\infty$	
64	s	35	50		50	s	$+\infty$	s
65	t	35	36		36	t	$+\infty$	t
66	q	35	36	$\perp$	36	q,s,t	$+\infty$	q,t
67	p	35	36	$36 \not< 35$	36	p,q,s,t	$+\infty$	
68	h	35	35	$\perp$	35	h,i,l,m,o,p,q,s,t	$+\infty$	h,i,j,k,l,o
69	a	35	35	$\perp$	35	$a,h,i,l,m,$ o,p,q,s,t	$+\infty$	$a,b,c,d,n,f,$ g,h,i,j,k,l,o

Figure A.21: MT-SSS* Table Pass 4

find a fail low as usual, but a fail high with return value 35. The return value is now a lower bound, backed-up by a min solution tree (all children of a min node included, only one for each max node).

How does Alpha-Beta traverse this min solution tree? The search follows the critical path a, h, i, l and o. At node l, child m has $c.f^- \not< \gamma$, so it is not entered. Now o is entered but not evaluated since it is no longer open. Node o returns the stored bound $o.f^+ = 35$. The value of the children is retrieved from storage. Note that the previous pass stored an f^+ value for l and o, while this pass stores an f^-. Node i cannot lower h's value ($g \geq \gamma, 35 \geq 35$, no cutoff occurs), so the search explores p. Node p expands q which, in turn, searches s and t. Since p is a maximizing node, the value of q (36) causes a cutoff: $g \not< \gamma$, node r is not searched. Both of h's children are ≥ 35. Node h returns 35, and so does a. Node a was searched attempting to show whether its value was $<$ or ≥ 35. Node h provides the answer: greater than or equal. This call to MT fails high, meaning we have a lower bound of 35 on the search. The previous call to MT established an upper bound of 35. Thus the minimax value of the tree is proven to be 35.

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

In the previous four passes we called MT (Alpha-Beta) with a special search window to have it emulate SSS*. This sequence of calls, creating a sequence of fail lows until the final fail high, can be captured in a single loop, given by the pseudo code of figure 3.1.

Appendix B

Equivalence of SSS* and MT-SSS*

This appendix discusses the ideas behind the equivalence of Stockman's SSS* and the new reformulation MT-SSS*. We do not present a formal proof in this appendix. That can be found in [106]. An outline of the full proof can be found in appendix A of [113, 112]. The aim of the current appendix is to convey the idea behind the equivalence. It can be regarded as an "informal proof," or as an illustration supporting the formal proof, which requires a certain amount of familiarity with the old formulation of SSS*.

B.1 MT and List-ops

Figure B.1 shows an extended version of MT, to be called by MT-SSS*, shown in figure B.2. The list operations (list-ops) between **{*** and ***}** are inserted to show the equivalence of MT-SSS* and Stockman's SSS*. The value of $\hat{h}$ is the current value of γ. (In implementations of MT-SSS* the list-ops should not be included.) The claim is that, when called by MT-SSS*, the list operations in MT cause the same Γ operations to be applied as in Stockman's original formulation. These list-ops are the core of the equivalence proof [106]. Their place in the MT code shows in a clear and concise way where and how the SSS* operators fit in the way Alpha-Beta traverses a tree. The example in appendix A.3 can be used to check this. Without the list-ops, the call MT(n, γ) is an ordinary null-window Alpha-Beta$(n, \gamma - 1, \gamma)$ search, except that MT uses one bound, making the code a bit simpler.

MT-SSS* and SSS* are equivalent in the sense that they evaluate the same leaf nodes in the same order, when called on the same minimax tree. In the previous chapters and examples SSS* and MT-SSS* have been treated primarily as algorithms that manipulate max solution trees. The original SSS* formulation stresses the min-solution-tree view. The full story is, of course, that both max and min solution trees are manipulated by SSS*, just as Alpha-Beta's postcondition in section 2.1.3 shows that Alpha-Beta also constructs both types of trees.

The critical tree that proves the minimax value of a game tree is a union of the max solution tree with the lowest possible upper bound, and the min solution tree with

```
/* For equivalence with SSS* this function must be called by MT-SSS*   */
/* (see figure B.2)                                                     */
/* MT: storage enhanced null-window Alpha-Beta(n, γ − 1, γ).            */
/* n is the node to be searched, γ − 1 is α , γ is β in the call.       */
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
          {* List-op(4, n); *}
          g := eval(n);
      else if n.f⁺ = +∞ then g := n.f⁻; else g := n.f⁺;
  else if n = max then
      {* retrieve n.f⁻, n.f⁺; if n.f⁺ = +∞ and n.f⁻ = −∞ then List-op(6, n); *}
      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);
              {* if g′ ≥ γ then List-op(1, c); *}
          else g′ := c.f⁺;
          g := max(g, g′);
          c := nextbrother(c);
  else /* n is a min node */
      {* retrieve n.f⁻, n.f⁺; if n.f⁺ = +∞ and n.f⁻ = −∞ then List-op(5, n); *}
      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);
              {* if g′ ≥ γ then
                  if c < lastchild(n) then List-op(2, c); else List-op(3, c); *}
          else g′ := c.f⁻;
          g := min(g, g′);
          c := nextbrother(c);
  /* Store one bound per node. Delete possible 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 B.1: MT with SSS*'s List-Ops

function MT-SSS*(n) $\to f$;
  $g := +\infty$;
  repeat
    $\gamma := g$;
    $g :=$ MT(n, γ);
  until $g = \gamma$;
  return g;

Figure B.2: SSS* as a Sequence of MT Searches

the highest possible lower bound. SSS* can be regarded as an interleaving of two processes, one working downward on the max solution tree part of the final minimal tree, and one working upward on the min solution tree part. First a max solution tree is expanded downwards from the root. Next, SSS* constructs a min solution tree growing upwards off of the critical leaf. The aim is to create a min solution tree that reaches all the way up to the root, and has the same value as the max solution tree, because that signals that the critical tree that proves the minimax value has been constructed.

However, often this process is ended prematurely, when SSS* finds that the value of the min solution tree drops below that of another node/min solution tree, which causes the whole process to start over again, giving SSS* its interleaved nature.

B.2 MT and the Six Operators

Turning to figures B.2, B.1, and 2.17, the construction of solution trees by the six operators will be discussed in a little more detail, focusing on the circumstances in which SSS* and MT invoke the six Γ operators.

Many of the operations that SSS* performs have a minor, local, effect: a max solution tree is being expanded, or SSS* backtracks to the node where the next one will be grown. However, at some points operations with a more major, global, nature are performed. When operator 4 has evaluated the last leaf of a max solution tree, and all the entries in the OPEN list are *solved*, the highest $\hat{h}$ value drops. The selection of the next entry, through SSS*'s sorting of the OPEN list, is a major decision point, where SSS* selects a node which is in a certain sense globally "best" (figure 2.18).

In MT-SSS* the minor, local, operations are performed by the MT code, by having a null-window Alpha-Beta search construct a solution tree. The major decision to find the next node which is globally "best" occurs in MT-SSS* when MT is called at the root to traverse the critical path to the critical leaf. In MT-SSS* the global decisions are more visible than in SSS*, since they coincide with a new pass of the main loop.

A number of features of SSS* play an important role in this discussion. At any time, the highest $\hat{h}$ value in the OPEN list is an upper bound on the minimax value of the root of the game tree. The entries are the leaves of a partial max solution tree. When the OPEN list consists only of *solved* entries, a pass of the main loop of MT-SSS* has ended (assuming that only one critical path exists). The entries in the list are the leaves of a total max solution tree, that is rooted at the root of the game tree.

Concerning MT, the following pre- and postcondition hold (following Alpha-Beta's postcondition in section 2.1.3): at a fail high, MT has traversed a min solution tree, whose value $f^-(n)$ is stored with node n. At a fail low, MT has traversed a max solution tree, whose value $f^+(n)$ is stored with node n. Since MT is called by MT-SSS* with $\gamma = f^+(root)$, the node on which MT is called is either open, or part of a max solution tree.

The SSS* notions *live* and *solved* correspond to the Alpha-Beta notions *open* (children not yet expanded) and *closed* (all children expanded). The term *sub* max solution tree is used for a max solution tree whose root is an interior node, not necessarily the root of the game tree, in other words, a smaller, deeper, max solution tree. A sub max solution tree may even be as small as a single leaf. The same applies to the term sub min solution trees.

It will be shown that the list-ops in MT cause the same operations to be performed on the OPEN list as the original SSS* operators. One of the key points is that SSS* always selects the entry with the highest $\hat{h}$ value. (Other preconditions of the operators, such as being applied to an open node or to a max node, are easily verified from the MT code in figure B.1.) Therefore, the focus of attention in the following discussion will be on how MT performs all the list-ops on the (left-most) node with the highest $\hat{h}$ value.

The key circumstances in which the six operators are invoked are discussed from the viewpoint of SSS* and MT.

SSS:* At the start of the algorithm, the OPEN list is empty. The entry with the highest $\hat{h}$ value is the root. SSS* now expands the left-most max solution tree using the operators 4, 5, and 6.

MT: MT selects this entry, below which the left-most max solution tree is expanded (by Alpha-Beta's postcondition), applying list-op's 4, 5, and 6. Examination of the SSS* operators on the one hand, and the MT code on the other, shows easily that the list-op's in MT follow the same expansion sequence as the operators in SSS*.

SSS:* At the start of another pass of MT-SSS*, the OPEN list contains only *solved* entries, that form the leaves of a total max solution tree. The highest entry is an upper bound on the minimax value of the root, and contains the critical leaf of the max solution tree.

MT: The conventional Alpha-Beta cutoff decisions cause MT to traverse the left-most critical path of the max solution tree, along which $\gamma = f^+(n) = f^+(root)$ holds, until the left-most critical leaf is reached. (In closed max nodes the left-most child with $c.f^+ = \gamma$ is entered; the highest child among the children in the max solution tree. In closed min nodes the left-most child with $c.f^- = -\infty < \gamma$ is entered; the only child in the max solution tree.) Thus, both SSS* and MT select the critical leaf to be operated upon.

Next, SSS* selects the node where a sub max solution tree will be expanded, in order to try to lower the upper bound on the minimax value of the root.

SSS:* Depending on the parity of the depth of the tree, either operator 1, or 2/3 are performed on the critical leaf. (Turning to the MT code, the critical leaf's value is equal to γ, so the inner MT call fails high with $g' \geq \gamma$.) Assuming an even depth of the tree, operator 2 or 3 are performed. These operators expand the "best" node: the node (or rather, the sub max solution tree which happens to be only a single node) that can influence the value at the root (see also figure 2.18). If the critical leaf has an unexpanded brother, then operator 2 selects it.

MT: In MT the unexpanded brother is selected because the return value $g' \geq \gamma$ does not cause a cutoff. (The MT-SSS* code ensures that $\gamma = f^{+}(root)$ is equal to the highest $\hat{h}$ value).

SSS:* If all open brothers of the critical leaf have been expanded with decreasing their $\hat{h}$ value, operator 3 applies, which performs half of the back-up operation to start expanding the next smallest sub max solution tree. Operator 1 applies next, which does the other half of the back-up motion, putting a max node at the front of the list, to be replaced by operator 2 by its next brother, starting the expansion of the sub max solution tree by operator 4, 5 and 6 just as when the sub max solution tree were only a single node, only it is bigger. (For odd depth trees, the part of the back-up process starting with operator 1 also applies.)

MT: In MT the back-up motion of operator 3 occurs because there are no more children, causing MT to automatically back-up to its parent. The back-up motion of operator 1 occurs because the return value of the inner MT call is still a fail high ($g' \geq \gamma$), which causes a cutoff in the max parent because $g \not< \gamma$, causing MT to back-up. In the min node where MT ends up, the return value is still a fail high, which causes no cutoff, so the next child is expanded, just like operator 2 does. (Of course, if no brother exists for operator 2, operator 3 applies. And if operator 1 puts the root on the list, which has no brothers, MT/MT-SSS* and SSS* both terminate.) Thus MT and SSS* also select the same nodes when the place to expand the sub max solution tree is to be determined.

Expansion of the sub max solution tree succeeds if the value of $\hat{h}$ is lowered, or in MT terms, if the inner MT calls fail low. If it fails high, a min solution tree will be constructed. (In the MT case this follows from the Alpha-Beta postcondition.)

SSS:* The question of failing high or low occurs when operator 4 is performed. A "fail high" causes the $\hat{h}$ value of the first entry to stay the same, so the backing-up operators for solved entries 1, 2 and 3 are applied, in the same fashion as described previously. A "fail low" causes another entry to have the highest $\hat{h}$ value. This entry has been inserted by operator 6, the only operator that inserts multiple entries into the list. All of the entries inserted by operator 6 have the same $\hat{h}$ value. The only operator that can lower an $\hat{h}$ value is number 4. When this happens, the next brother (or rather, (great) uncle) appears at the front. (No entries from outside the sub max solution tree to be constructed can enter, because their $\hat{h}$ value is lower.) In this way, as long as the expansions "fail low", a max solution tree is expanded in the same way as the left-most max solution tree in the first pass.

MT: In MT, construction of a sub max solution tree after a fail low is easy to see from Alpha-Beta's postcondition, or from the code. (The same applies for a sub min solution tree after a fail high).

This concludes the description of a single pass of MT-SSS*. Next, the globally "best" node is selected—in SSS* since the OPEN list is sorted, in MT-SSS* because the stored $f^+(n)$ values together with the γ value cause MT to traverse the critical path.

The previous explanation indicated that SSS* and MT perform the same operations on the OPEN list for the following cases:

- At the first call to MT from MT-SSS*, where the left-most max solution tree is constructed.
- When finding the critical leaf in subsequent calls to MT from the main loop of MT-SSS*, where the OPEN list contains the leaves of a max solution tree.
- When further expanding parts of the tree in order to get a better upper bound on the root (fail low), or finding that the current upper bound is the minimax value (fail high).

Comparing SSS* to MT/MT-SSS*, the ingenuity with which the six Γ cases emulate Alpha-Beta's behavior in MT-SSS* is extraordinary. Why SSS* was created in its old form is easier to see if one realizes that its roots lie in AO*, an algorithm for the search of problem-reduction spaces [93].

Appendix C

Test Data

This appendix gives the test positions with which most of the reported test results were computed. (The extra positions that were used to verify that the base test set did not cause anomalies are not shown.) Furthermore we give the absolute node counts (the best in bold) for each position for the deepest search depth.

C.1 Test Results

Checkers

Cumulative Leaf Nodes Checkers Depth 17

position	Alpha-Beta	NegaScout	SSS*	DUAL*	MTD(f)
1	126183	111189	106907	**98465**	114964
2	177188	155517	192163	136168	**128615**
3	271082	285198	301536	302170	**232238**
4	945249	888190	779443	**771386**	794280
5	1978773	1884569	2213865	1869080	**1661477**
6	1368018	**1228123**	1504601	1350407	1283644
7	544271	**481183**	550118	538703	510826
8	905571	917252	1034249	794801	**746069**
9	1518870	1344296	1510397	1490632	**981401**
10	1763143	**1869436**	2260341	2176786	1998528
11	221754	193222	**190221**	194031	199732
12	207108	**189654**	206880	197521	195524
13	203402	182482	192235	217626	**178357**
14	253300	261241	248463	**212910**	239799
15	265731	217568	268320	228979	**205280**
16	248657	250896	248168	241268	**238638**
17	357382	**320275**	398641	382976	383503
18	718949	672121	767435	726969	**623981**
29	1319426	1040648	1244843	**815568**	946250
20	1159464	1066905	1106151	1632830	**886752**

Cumulative Total Nodes Checkers Depth 17

position	Alpha-Beta	NegaScout	SSS*	DUAL*	MTD(f)
1	413369	**367200**	570840	398806	459440
2	462424	417435	1193339	**405075**	412199
3	620027	667162	1052940	728422	**539190**
4	2352550	2253635	2926647	2426232	**2016395**
5	4800897	4716967	7103273	5598219	**4124889**
6	3227618	**2937729**	5584087	4074537	3124792
7	1321635	**1191009**	1982897	2074539	1305613
8	2173069	2217950	3041809	3097442	**1903700**
9	3658536	3346953	4208761	5059490	**2587216**
10	3916120	**4130046**	6597076	7265863	4554377
11	567607	**503305**	633484	531527	523338
12	535462	**507324**	736516	585877	535851
13	511587	**470443**	641281	602930	482941
14	681776	697866	833239	709308	**634044**
15	707039	611411	970302	864961	**540321**
16	656466	658201	977425	901367	**639200**
17	906151	**844337**	1279247	1227917	1018729
18	1820759	**1738552**	2528449	2711356	1755457
29	3290314	2680018	4024730	3266939	**2491371**
20	3124797	2870023	3197680	6197403	**2690201**

Othello

Cumulative Leaf Nodes Othello Depth 10

position	Alpha-Beta	NegaScout	SSS*	DUAL*	MTD(f)
1	1077545	1002150	**756286**	1044572	973214
2	342372	309981	400052	309283	**291823**
3	469632	422933	482805	435760	**410908**
4	810910	748961	**594689**	774421	754279
5	554011	499434	528925	507417	**493501**
6	533061	**481959**	487469	506341	496562
7	511596	**455207**	499995	472456	459245
8	550836	505633	577386	509321	**502990**
9	443277	**365431**	606806	419831	394294
10	541548	519187	545360	512457	**504142**
11	675550	660471	**463393**	681217	466876
12	734587	609609	719514	**570924**	635436
13	578096	**489623**	517986	536282	495934
14	517719	479973	462101	447383	**407302**
15	684021	595341	738158	583758	**558692**
16	2103805	1874684	2336257	2146225	**1785653**
17	444027	**381167**	497447	387313	381482
18	693612	665237	**610446**	618819	660550
19	643278	562910	596583	577897	**518339**
20	816195	**700857**	768269	707928	704796

Cumulative Total Nodes Othello Depth 10

position	Alpha-Beta	NegaScout	SSS*	DUAL*	MTD(f)
1	1720270	1617060	**1531266**	1768573	1588262
2	580163	541031	884386	564478	**524143**
3	779481	715389	996805	781187	**699880**
4	1380704	**1300157**	1377430	1361942	1310286
5	955789	**881531**	1283857	960307	882869
6	975970	**900137**	1099004	961814	927423
7	766394	**687075**	1037853	775328	688037
8	936839	**863573**	1265701	944988	871360
9	783713	**668933**	1244449	807264	723717
10	1006877	973282	1329432	971021	**940991**
11	958764	941033	764296	1004557	**679615**
12	1241744	1033213	1503985	**1022343**	1094600
13	963782	**845619**	1292204	943211	873842
14	856701	807066	896673	780930	**661439**
15	1105961	986293	1513070	1019654	**915995**
16	3207698	2902247	4565011	3552642	**2753854**
17	769704	**662209**	1277545	693966	666419
18	1104089	**1059736**	1359768	986829	1092532
19	1015573	896343	1281957	982266	**834850**
20	1375433	**1164485**	1670091	1236846	1181879

Chess

Cumulative Leaf Nodes Chess Depth 8

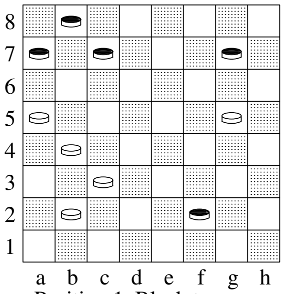
position	Alpha-Beta	NegaScout	SSS*	DUAL*	MTD(f)
1	2139803	1985777	**1785412**	1978323	1812787
2	2644826	2476699	2789442	**2084553**	2204018
3	2579806	2427222	2605278	2543878	**1772911**
4	2981122	2534212	3068148	2622208	**2186732**
5	2704505	2384329	2581678	2417116	**2306988**
6	3144654	2608290	2959795	2619766	**2593478**
7	1519760	1477958	**1467838**	1504933	1473132
8	5341059	5317246	5008482	**3414294**	3862095
9	4886358	4053813	5356186	3906343	**3722252**
10	7567304	6327725	5662936	5934283	**5413685**
11	2604254	**2424632**	2930218	2568160	2433724
12	3751738	3383563	2420545	3321493	**2205522**
13	2751157	2660323	2396603	2604666	**2064121**
14	2970794	2517235	3161045	**2362827**	2697990
15	1301380	1210901	1213741	1308549	**1174032**
16	1256342	1224924	1402361	1207203	**1178989**
17	1120616	1111323	1161968	**1065096**	1118400
18	1287345	1238906	1277886	1278149	**1173362**
19	2560676	2363614	2540931	2393803	**2132042**
20	2095458	1977936	2220139	**1951231**	1970067

Cumulative Total Nodes Chess Depth 8

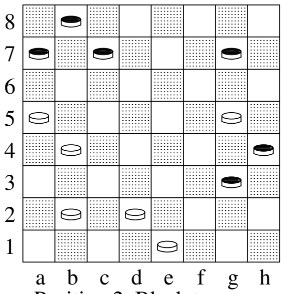
position	Alpha-Beta	NegaScout	SSS*	DUAL*	MTD(f)
1	3303219	3103360	5902747	3485589	**2920250**
2	3867251	3648085	6695953	3318056	**3267966**
3	3970801	3723289	7834576	3958079	**2882594**
4	4670576	4031181	8728375	4283256	**3546285**
5	4393422	3863494	6477649	4041426	**3742236**
6	4937220	**4178390**	10703081	4298577	4188882
7	2731907	2682243	3948657	2993329	**2665386**
8	7604601	7487371	12764983	**5498919**	5728846
9	7160185	6056626	14830764	6217694	**5588940**
10	10339575	8620155	21572998	8714449	**7470636**
11	4068559	**3819434**	11293758	4140659	3907065
12	5625271	5062261	8174419	5292912	**3464844**
13	4319258	4178469	9141174	4243100	**3309548**
14	4518683	3837605	9609051	**3782715**	4175020
15	2107983	1988196	2616457	2475369	**1951313**
16	2240408	2187733	4827452	2315079	**2127931**
17	1980620	**1973707**	4868220	2046282	2110399
18	2164189	2110233	7863155	2226773	**2020108**
19	4011716	3708911	12487591	3905568	**3389902**
20	3447220	**3272595**	12443353	3407857	3276117

C.2 Test Positions

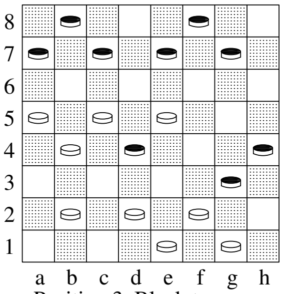
Checkers



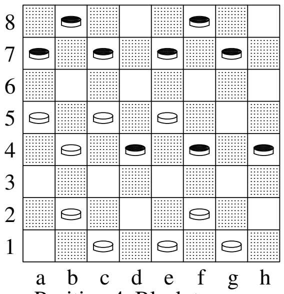
Position 1, Black to move



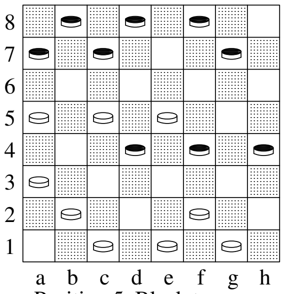
Position 2, Black to move



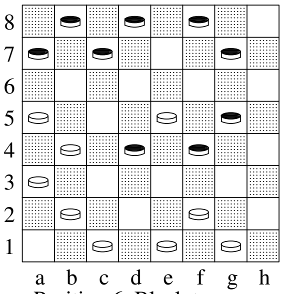
Position 3, Black to move



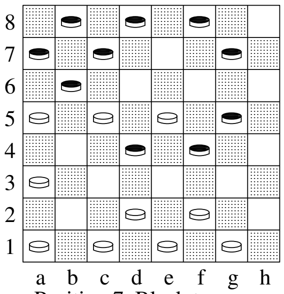
Position 4, Black to move



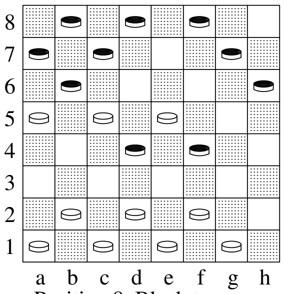
Position 5, Black to move



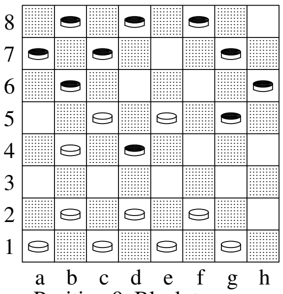
Position 6, Black to move



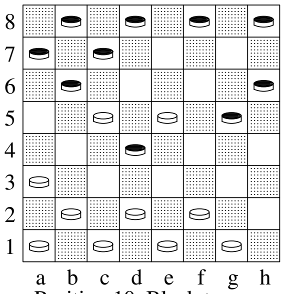
Position 7, Black to move



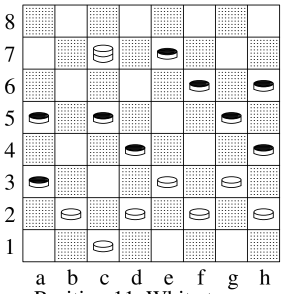
Position 8, Black to move



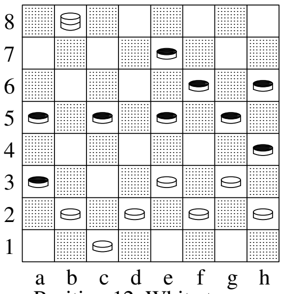
Position 9, Black to move



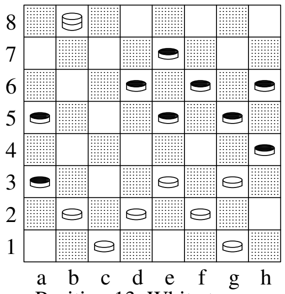
Position 10, Black to move



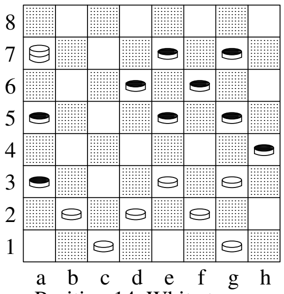
Position 11, White to move



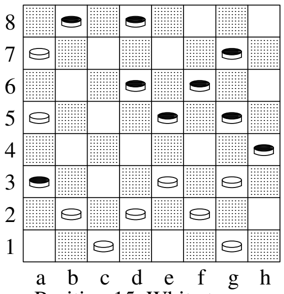
Position 12, White to move



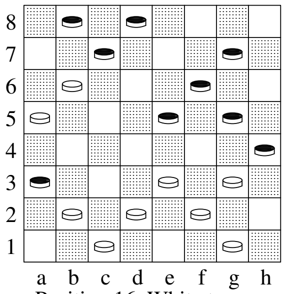
Position 13, White to move



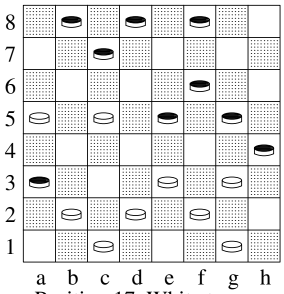
Position 14, White to move



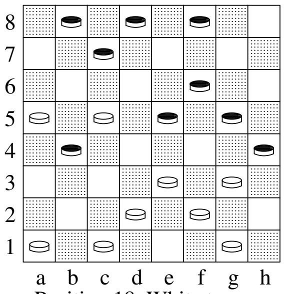
Position 15, White to move



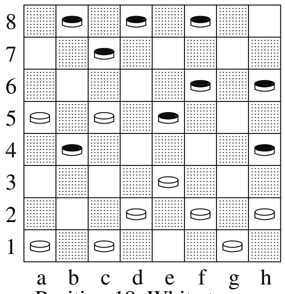
Position 16, White to move



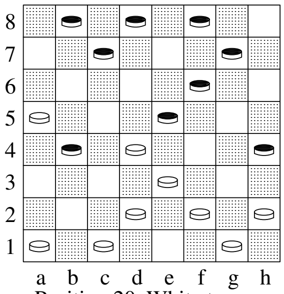
Position 17, White to move



Position 18, White to move

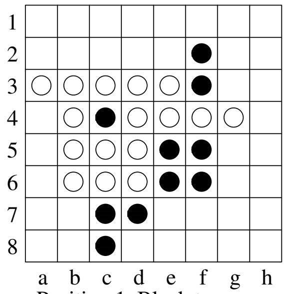


Position 19, White to move

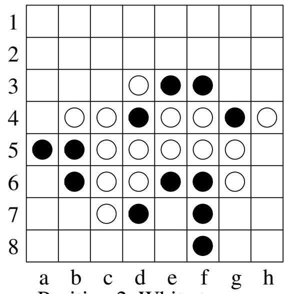


Position 20, White to move

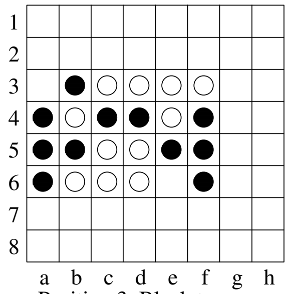
Othello



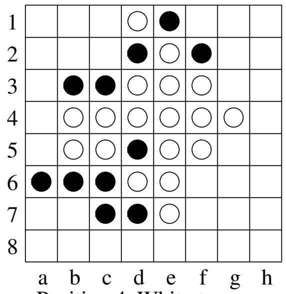
Position 1, Black to move



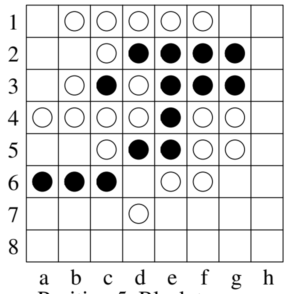
Position 2, White to move



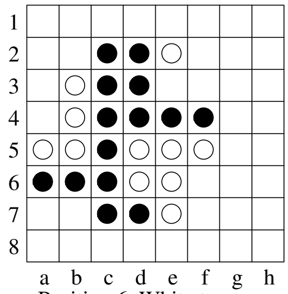
Position 3, Black to move



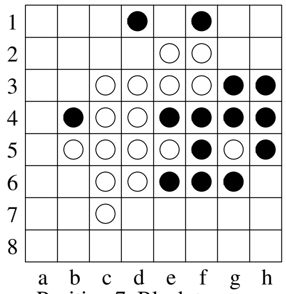
Position 4, White to move



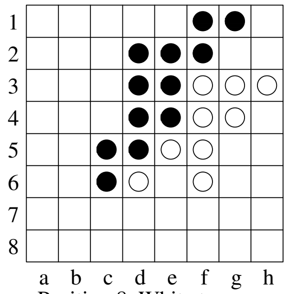
Position 5, Black to move



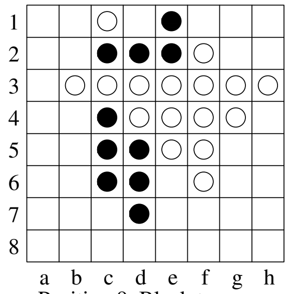
Position 6, White to move



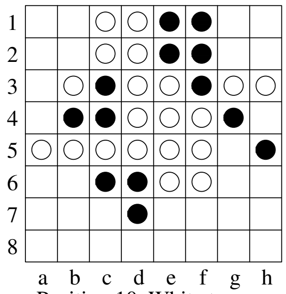
Position 7, Black to move



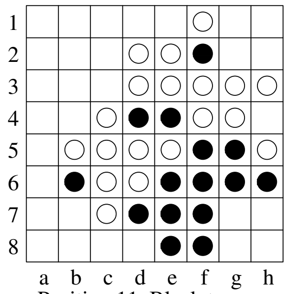
Position 8, White to move



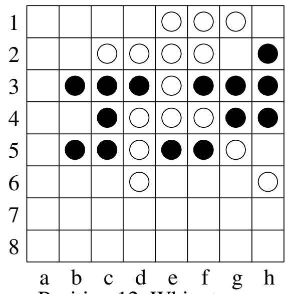
Position 9, Black to move



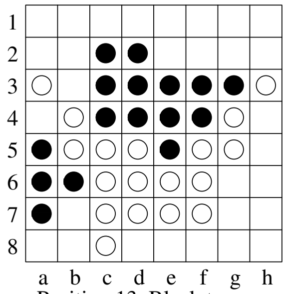
Position 10, White to move



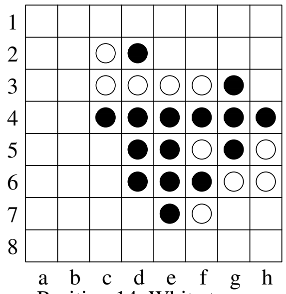
Position 11, Black to move



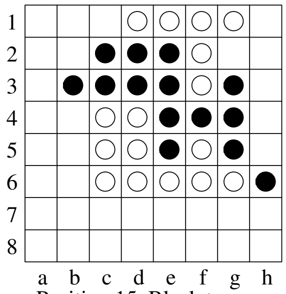
Position 12, White to move



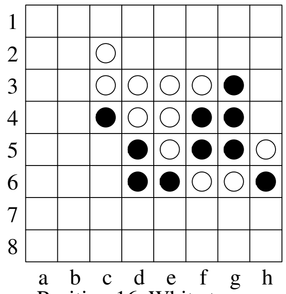
Position 13, Black to move



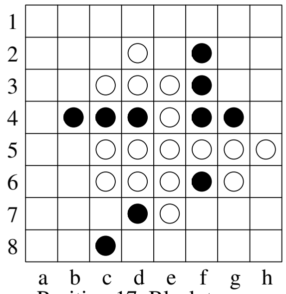
Position 14, White to move



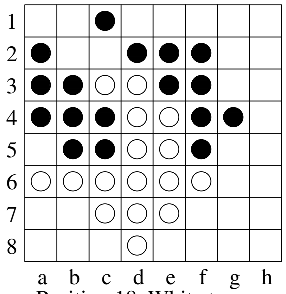
Position 15, Black to move



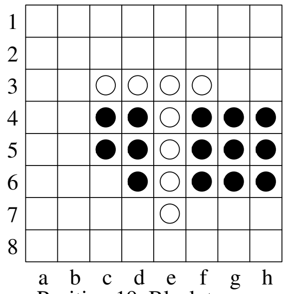
Position 16, White to move



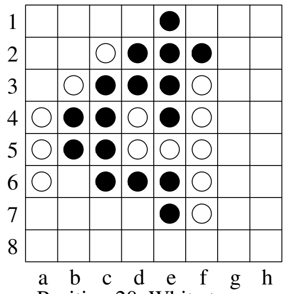
Position 17, Black to move



Position 18, White to move



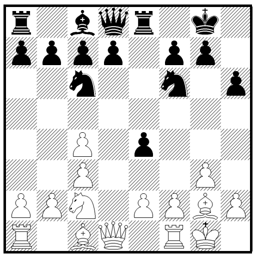
Position 19, Black to move

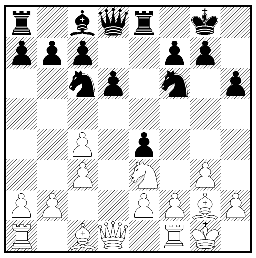


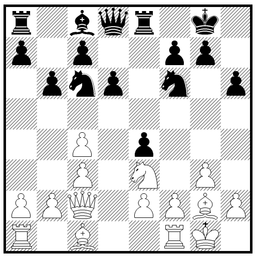
Position 20, White to move

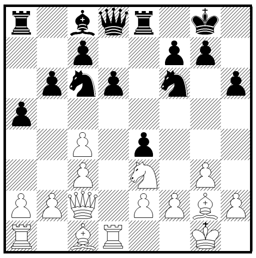
Chess

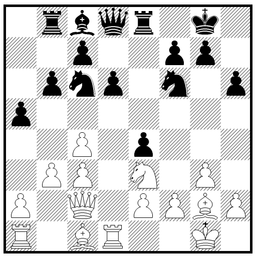
For all 20 positions: White to move.

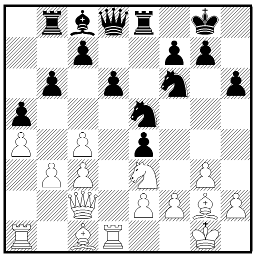

Position 1

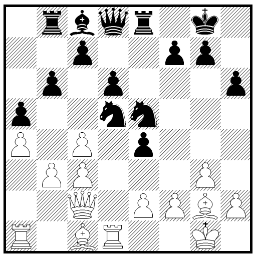

Position 2

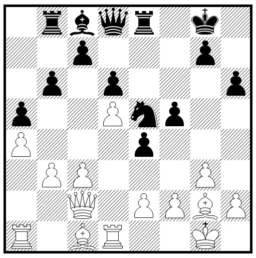

Position 3

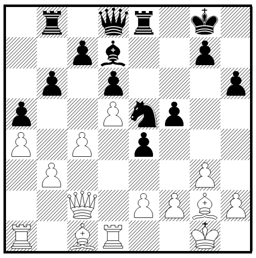

Position 4

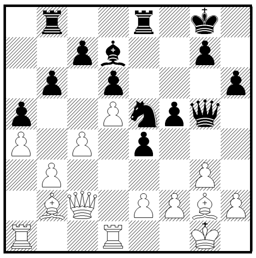

Position 5

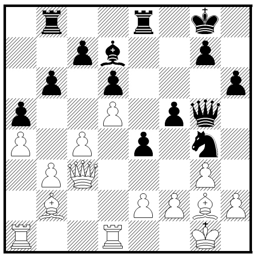

Position 6

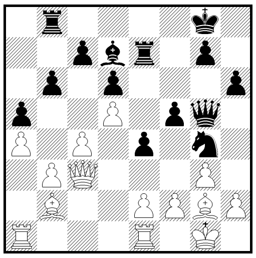

Position 7

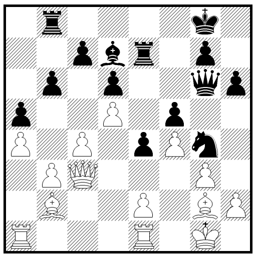

Position 8

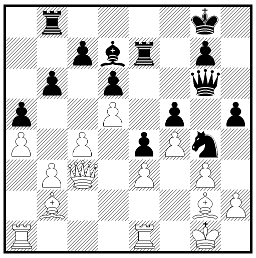

Position 9

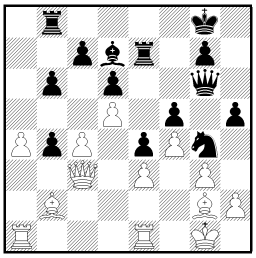

Position 10

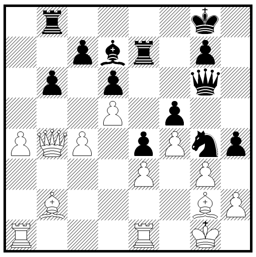

Position 11

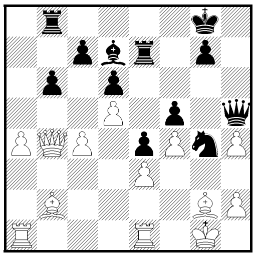

Position 12

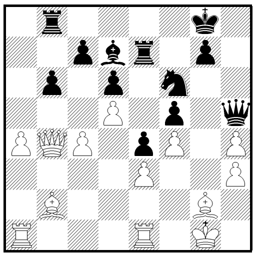

Position 13

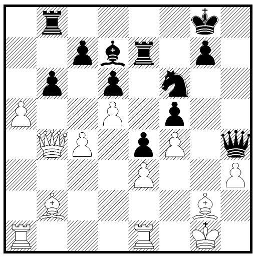

Position 14

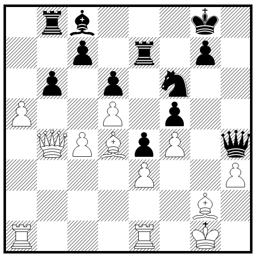

Position 15


Position 16


Position 17


Position 18


Position 19


Position 20

References

[1] Georgi M. Adelson-Velsky, Vladimir L. Arlazarov, and Mikhail V. Donskoy. Some methods of controlling the tree search in chess programs. *Artificial Intelligence*, 6(4):361–371, 1975.

[2] L. Victor Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, Maastricht, The Netherlands, September 1994.

[3] L. Victor Allis, Maarten van der Meulen, and H. Jaap van den Herik. Proof-number search. *Artificial Intelligence*, 66:91–124, March 1994.

[4] Claude Amiguet. *Contrôleurs Distribués pour la Programmation Heuristique*. PhD thesis, Swiss Federal Institute of Technology, Department of Computer Science, Lausanne, Switzerland, 1991.

[5] Thomas S. Anantharaman. *A Statistical Study of Selective Min-Max Search*. PhD thesis, Department of Computing Science, Carnegie Mellon University, Pittsburg, PA, 1990.

[6] Thomas S. Anantharaman, Murray S. Campbell, and Feng hsiung Hsu. Singular extensions: Adding selectivity to brute-force searching. *Artificial Intelligence*, 43(1):99–109, 1990.

[7] Bruce W. Ballard. The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21:327–350, 1983.

[8] Gérard M. Baudet. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1978.

[9] 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.

[10] Don F. Beal. Experiments with the null-move. In D.F. Beal, editor, *Advances in Computer Chess 5*, pages 65–79, Amsterdam, 1989. Elsevier Science Publishers.

[11] Don F. Beal. A generalized quiescence search algorithm. *Artificial Intelligence*, 43(1):85–98, 1990.

[12] Hans J. Berliner. *Chess as problem solving: The development of a tactics analyzer*. PhD thesis, Carnegie-Mellon University, Department of Computer Science, March 1974.

[13] Hans J. Berliner. The B* tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12:23–40, 1979.

[14] Hans J. Berliner. Backgammon computer program beats world champion. *Artificial Intelligence*, 14:205–220, 1980.

[15] Hans J. Berliner. Computer backgammon. *Scientific American*, 243(1):64–72, 1980.

[16] Hans J. Berliner and Christopher McConnell. B* probability based search. *Artificial Intelligence*, 1995. To appear. Also published as Technical Report CMU-CS-94-168, Carnegie Mellon University, Pittsburgh, PA.

[17] Alan Bernstein, M. de V. Roberts, T. Arbuckle, and M.A. Belsky. A chess-playing program for the IBM 704 computer. In *Proceedings Western Joint Computer Conference*, pages 157–159, 1958.

[18] Subir Bhattacharya. Experimenting with revisits in game tree search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, volume 1, pages 243–249, Montreal, Canada, August 1995.

[19] Subir Bhattacharya and Amitava Bagchi. Searching game trees in parallel using SSS*. In *Proceedings IJCAI-89*, pages 42–47, 1989.

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

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

[22] Subir Bhattacharya and Amitava Bagchi. A general framework for minimax search in game trees. *Information processing letters*, 52:295–301, 1994.

[23] Robert D. Blumofe and Charles E. Leiserson. Scheduling large-scale parallel computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[24] Dennis M. Breuker, Jos W.H.M. Uiterwijk, and H. Jaap van den Herik. Replacement schemes for transposition tables. *ICCA Journal*, 17(4):183–193, December 1994.

[25] Mark G. Brockington. Improvements to parallel Alpha-Beta algorithms. Technical Report (Internal), Department of Computing Science, University of Alberta, Edmonton, Canada, 1994.

[26] Mark G. Brockington. A taxonomy of parallel game-tree search algorithms. *ICCA Journal*, 1995. Submitted for publication.

[27] A.L. Brudno. Bounds and valuations for shortening the search of estimates. *Problems of Cybernetics*, 10:225–241, 1963. Appeared originally in Russian in Problemy Kibernetiki, vol. 10, pp. 141–150, 1963.

[28] 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, The Netherlands, May 1994.

[29] Arie de Bruin, Wim Pijls, and Aske Plaat. Solution trees as a basis for game-tree search. *ICCA Journal*, 17(4):207–219, December 1994.

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

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

[32] Murray S. Campbell. Algorithms for the parallel search of game trees. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Canada, August 1981. Available as Technical Report 81-8.

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

[34] Joe H. Condon and Ken Thompson. Belle chess hardware. In M.R.B. Clarke, editor, *Advances in Computer Chess 3*, pages 45–54. Pergamon Press, Oxford, 1982.

[35] 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*, pages 25–43. Pergamon Press, Oxford, 1982.

[36] Joseph C. Culberson and Jonathan Schaeffer. Efficiently searching the 15-puzzle. Technical Report TR-94-08, Department of Computing Science, University of Alberta, Edmonton, Canada, 1994.

[37] Thomas L. Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI-88)*, pages 49–54, Minneapolis/Saint Paul, Minnesota, August 1988.

[38] Claude G. Diderich. Evaluation des performances de l'algorithme SSS* avec phases de synchronisation sur une machine parallèle à mémoires distribuées. Technical report, Swiss Federal Institute of Technology, Department of Computer Science, Laboratory for Theoretical Computer Science, Lausanne, Switzerland, June 1992.

[39] Mikhail Donskoy and Jonathan Schaeffer. Perspectives on falling from grace. In *New Directions in Game-Tree Search*, pages 85–93, Edmonton, Alberta, Canada, 1989. Also appeared in ICCA Journal, 12(3), pp. 155–163, 1989, and in Chess, Computers and Cognition, T. Anthony Marsland and Jonathan Schaeffer (eds.), Springer-Verlag, pp. 259–268, 1990.

[40] Carl Ebeling. *All the Right Moves*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, MIT Press, Cambridge, Massachusetts, 1987.

[41] Rainer Feldmann. *Spielbaumsuche mit massiv parallelen Systemen*. PhD thesis, Universität-Gesamthochschule Paderborn, Germany, May 1993. English translation available: Game Tree Search on Massively Parallel Systems.

[42] Rainer Feldmann, Burkhard Monien, Peter Mysliwietz, and Oliver Vornberger. Distributed game tree search. In Vipin Kumar, P.S. Gopalakrishnan, and Laveen N. Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 66–101. Springer-Verlag, 1990.

[43] Rainer Feldmann, Peter Mysliwietz, and Burkhard Monien. A fully distributed chess program. In Don Beal, editor, *Advances in Computer Chess 6*, pages 1–27. Ellis Horwood, 1990.

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

[45] John P. Fishburn and Raphael A. Finkel. Parallel alpha-beta search on arachne. Technical Report 394, Computer Sciences Department, University of Wisconsin, Madison, WI, 1980.

[46] Aviezri S. Fraenkel. Selected bibliography on combinatorial games and some related material. In *Proceedings Combinatorial Games Workshop*, Berkeley, CA, 1995. MSRI. Also in Electronic Journal of Combinatorics, Nov 1994, http://ejc.math.gatech.edu:8080/Journal/Surveys/index.html.

[47] Peter W. Frey, editor. *Chess Skill in Man and Machine*. Springer-Verlag, New York, NY, 1977.

[48] Frederic Friedel. Pentium Genius beats Kasparov: A report on the Intel Speed Chess Grand Prix in London. *ICCA Journal*, 17(3):153–158, September 1994.

[49] Frederic Friedel and Frans Morsch. The Intel world chess express challenge. *ICCA Journal*, 17(2):98–104, June 1994.

[50] Ralph Gasser. *Efficiently Harnessing Computational Resources for Exhaustive Search*. PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 1995.

[51] Gordon Goetsch and Murray S. Campbell. Experiments with the null move heuristic in chess. In *AAAI Spring Symposium Proceedings*, pages 14–18. AAAI, 1988.

[52] Richard D. Greenblatt, D.E. Eastlake, and S.D. Crocker. The Greenblatt chess program. In *Fall Joint Computer Conference*, volume 31, pages 801–810, 1967.

[53] H. Jaap van den Herik. *Computerschaak, Schaakwereld en Kunstmatige Intelligentie*. PhD thesis, Delft University of Technology, The Netherlands, 1983.

[54] 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.

[55] Feng hsiung Hsu, Thomas S. Anantharaman, Murray S. Campbell, and Andreas Nowatzyk. Deep thought. In T. Anthony Marsland and Jonathan Schaeffer, editors, *Computers, Chess, and Cognition*, pages 55–78. Springer Verlag, 1990.

[56] Feng hsiung Hsu, Thomas S. Anantharaman, Murray S. Campbell, and Andreas Nowatzyk. A grandmaster chess machine. *Scientific American*, 263(4):44–50, October 1990.

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

[58] Toshihide Ibaraki. Game solving procedure H* is unsurpassed. In D. S. Johnson et al., editor, *Discrete Algorithms and Complexity*, pages 185–200. Academic Press, Inc., 1987.

[59] Toshihide Ibaraki and Yoshiroh Katoh. Searching minimax game trees under memory space constraint. *Annals of Mathematics and Artificial Intelligence*, 1:141–153, 1990.

[60] Christopher F. Joerg and Bradley C. Kuszmaul. Massively parallel chess. In *Third DIMACS Parallel Implementation Challenge*, Rutgers University, NJ, 1994. Available as ftp://theory.lcs.mit.edu/pub/bradley/dimacs94.ps.Z.

[61] Andreas Junghanns, Christian Posthoff, and Michael Schlosser. Search with fuzzy numbers. In *Proceedings FUZZ IEEE/IFES'95*, pages 979–986, Yokohama, Japan, March 1995.

[62] Hermann Kaindl, Gerhard Kainz, Angelika Leeb, and Harald Smetana. How to use limited memory in heuristic search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, volume 1, pages 236–242, August 1995.

[63] 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.

[64] Donald E. Knuth. *The Art of Computer Programming, Volume Three, searching and Sorting*. Addison-Wesley Publishing Co., Reading, MA, 1973.

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

[66] Danny Kopec and Ivan Bratko. The Bratko-Kopec experiment: A comparison of human and computer performance in chess. In M.R.B. Clarke, editor, *Advances in Computer Chess 3*, pages 57–72. Pergamon Press, 1982.

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

[68] Richard E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42:189–211, 1990.

[69] Richard E. Korf and David W. Chickering. Best-first minimax search: First results. In *Proceedings of the AAAI'93 Fall Symposium*, pages 39–47. AAAI Press, October 1993.

[70] 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.

[71] Hans-Joachim Kraas. *Zur Parallelisierung des SSS*-Algorithmus*. PhD thesis, TU Braunschweig, Germany, January 1990.

[72] Vipin Kumar and Laveen N. Kanal. A general branch and bound formulation for understanding and synthesizing AND/OR tree search procedures. *Artificial Intelligence*, 21:179–198, 1983.

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

[74] Vipin Kumar and Laveen N. Kanal. A general branch and bound formulation for AND/OR graph and game tree search. In *Search in Artificial Intelligence*. Springer Verlag, 1988.

[75] Bradley C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1994.

[76] James B.H. Kwa. BS*: An admissible bidirectional staged heuristic search algorithm. *Artificial Intelligence*, 38:95–109, February 1989.

[77] Kai-Fu Lee and Sanjoy Mahajan. The development of a world class othello program. *Artificial Intelligence*, 43:21–36, 1990.

[78] Daniel B. Leifker and Laveen N. Kanal. A hybrid SSS*/alpha-beta algorithm for parallel search of game trees. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 1044–1046, 1985.

[79] David Levy. *Computer Chess Compendium*. Springer-Verlag, 1988.

[80] Giovanni Manzini. BIDA*: An improved perimeter search algorithm. *Artificial Intelligence*, 75(2):347–360, June 1995.

[81] T. Anthony Marsland. A review of game-tree pruning. *ICCA Journal*, 9(1):3–19, March 1986.

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

[83] T. Anthony Marsland and Fred Popowich. Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(4):442–452, July 1985.

[84] T. Anthony Marsland and Alexander Reinefeld. Heuristic search in one and two player games. Technical Report 93-02, University of Alberta, Edmonton, Canada, Paderborn Center for Parallel Computing, Germany, February 1993.

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

[86] T. Anthony Marsland and N. Srimani. Phased state space search. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, pages 514–518, Dallas, TX, November 1986.

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

[88] Agata Muszycka and Rajjan Shinghal. An empirical comparison of pruning strategies in game trees. *IEEE Transactions on Systems, Man and Cybernetics*, 15(3):389–399, May/June 1985.

[89] Peter Mysliwietz. Personal communication. January 1996.

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

[91] Dana S. Nau. Pathology on game trees revisited, and an alternative to minimaxing. *Artificial Intelligence*, 21:221–244, 1983.

[92] Allen Newell, Cliff Shaw, and Herbert Simon. Chess playing programs and the problem of complexity. *IBM Journal of Research and Development*, 4(2):320–335, 1958. Also in Computers and Thought, Feigenbaum and Feldman (eds.), pp. 39–70, McGraw-Hill, (1963).

[93] Nils J. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill Book Company, New York, NY, 1971.

[94] Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, Palo Alto, CA, 1980.

[95] Andrew J. Palay. *Searching with Probabilities*. PhD thesis, Computer Science Department, Carnegie-Mellon University, Pittsburg, PA, 1985.

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

[97] Judea Pearl. The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. *Communications of the ACM*, 25(8):559–564, August 1982.

[98] Judea Pearl. On the nature of pathology in game searching. *Artificial Intelligence*, 20:427–453, 1983.

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

[100] Judea Pearl and Richard E. Korf. Search techniques. *Annual Review of Computer Science*, 2:451–467, 1987.

[101] Wim Pijls. *Shortest Paths and Game Trees*. PhD thesis, Erasmus University Rotterdam, The Netherlands, November 1991.

[102] Wim Pijls and Arie de Bruin. Another view on the SSS* algorithm. In T. Asano, T. Ibaraki, H. Imai, and T. Nishizeki, editors, *Algorithms, SIGAL '90, Tokyo*, volume 450 of *LNCS*, pages 211–220. Springer-Verlag, August 1990.

[103] Wim Pijls and Arie de Bruin. Searching informed game trees. Technical Report EUR-CS-92-02, Erasmus University Rotterdam, The Netherlands, October 1992. Extended abstract in Proceedings CSN 92, pp. 246–256, and Algorithms

and Computation, ISAAC 92 (T. Ibaraki, ed), pp. 332–341, LNCS 650 Springer-Verlag.

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

[105] Wim Pijls and Arie de Bruin. Generalizing alpha-beta. In H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk, editors, *Advances in Computer Chess 7*, pages 219–236. University of Limburg, Maastricht, The Netherlands, 1994.

[106] 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.

[107] 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.

[108] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. A new paradigm for minimax search. Technical Report TR-CS-94-18, Department of Computing Science, University of Alberta, Edmonton, AB, Canada, December 1994.

[109] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. SSS* = α-β + TT. Technical Report TR-CS-94-17, Department of Computing Science, University of Alberta, Edmonton, AB, Canada, December 1994.

[110] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. Best-first and depth-first minimax search in practice. In *Proceedings of Computing Science in the Netherlands*, November 1995. A slightly longer version has been presented as: “An Algorithm Faster than NegaScout and SSS* in Practice,” at the Computer Strategy and Game Programming Workshop, World Computer Chess Championship, Hong Kong, May 1995.

[111] 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, Montreal, Canada, August 1995.

[112] 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, June 1995.

[113] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, To appear in the fall of 1996. Accepted for publication, November 1995.

[114] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. Exploiting graph properties of game trees. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96)*, Portland, OR, August 1996. American Association for Artificial Intelligence, AAAI Press. Accepted for Publication.

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

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

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

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

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

[120] Ronald L. Rivest. Game tree searching by min/max approximation. *Artificial Intelligence*, 34(1):77–96, 1987.

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

[122] Stuart Russell and Eric Wefald. *Do The Right Thing*. The MIT Press, Cambridge, MA, 1991.

[123] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:211–229, 1959. Reprinted in Computers and Thought, Feigenbaum and Feldman, eds., McGraw-Hill, 1963, pp. 71–105, and in Computer Games I, David Levy, ed., Springer, 1987, pp. 335–365.

[124] Arthur L. Samuel. Some studies in machine learning using the game of checkers. II — recent progress. *IBM Journal of Research and Development*, 11:601–617, 1967. Reprinted in Computer Games I, David Levy, ed, Springer, 1987, pp. 366–400.

[125] 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.

[126] Jonathan Schaeffer. Speculative computing. *ICCA Journal*, 10(3):118–124, 1987.

[127] Jonathan Schaeffer. Distributed game-tree searching. *Journal of Parallel and Distributed Computing*, 6:90–114, 1989.

[128] 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.

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

[130] Jonathan Schaeffer. Personal communication. 1994.

[131] 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.

[132] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. Chinook: The world man-machine checkers champion. *AI Magazine*, 1995. Accepted for publication.

[133] Jonathan Schaeffer, Paul Lu, Duane Szafron, and Robert Lake. A re-examination of brute-force search. In *Games: Planning and Learning: AAAI 1993 Fall Symposium*, pages 51–58, Report FS9302, AAAI, 1993.

[134] Jonathan Schaeffer and Aske Plaat. New advances in alpha-beta searching. In *Proceedings of the 24th ACM Computer Science Conference 1996*, pages 124–130, February 16–18 1996.

[135] Claude E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(7):256–275, 1950. See also: Levy, D.N.L. (ed.) Computer Games I, New York, NY, Springer-Verlag, pp. 81–88, 1988.

[136] James H. Slagle and John K. Dixon. Experiments with some programs that search game trees. *Journal of the ACM*, 16(2):189–207, April 1969.

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

[138] Bradley S. Stewart, Ching-Fang Liaw, and Chelsea C. White III. A bibliography of heuristic search research through 1992. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(2):268–293, February 1994.

[139] George C. Stockman. *A problem-reduction approach to the linguistic analysis of waveforms*. PhD thesis, University of Maryland, Department of Computer Science, May 1977. Technical Report TR-538.

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

[141] George C. Stockman. Personal communication. July 1995.

[142] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, March 1995.

[143] Ken Thompson. Computer chess strength. In M.R.B. Clarke, editor, *Advances in Computer Chess 3*, pages 55–56. Pergamon Press, Oxford, 1982.

[144] Jos W.H.M. Uiterwijk. The Kasparov – Deep Blue match. *ICCA Journal*, 19(1):38–41, March 1996.

[145] Oliver Vornberger and Burkhard Monien. Parallel alpha-beta versus parallel SSS*. In *IFIP Conference on Distributed Processing*, pages 613–625, Amsterdam, The Netherlands, October 1987. North-Holland.

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

[147] Jean-Christophe Weill. The ABDADA distributed minimax search algorithm. In *Proceedings of the 24th ACM Computer Science Conference*, pages 131–138, Philadelphia, PA, February 1996.

[148] Shlomo Zilberstein. Optimizing decision quality with contract algorithms. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, volume 2, pages 1576–1582, Montreal, Canada, August 1995.

Index

Samenvatting

Zoeken & Her-zoeken Onderzocht

Dit proefschrift gaat over zoekalgoritmen. Zoekalgoritmen worden vaak aan de hand van hun expansie-strategie gekarakteriseerd. Een mogelijke strategie is de depth-first strategie, een eenvoudig backtrack mechanisme waarbij de volgorde waarin knopen gegenereerd worden bepaalt hoe de zoekruimte doorlopen wordt. Een alternatief is de best-first strategie, ontworpen om het gebruik van domein-afhankelijke heuristische informatie mogelijk te maken. Door veelbelovende wegen eerst te bewandelen, zijn best-first algoritmen in het algemeen efficiënter dan depth-first algoritmen.

Voor minimax spelprogramma's (zoals voor schaken en dammen) is de efficiëntie van het zoek algoritme van cruciaal belang. Echter, alle goede programma's zijn gebaseerd op een depth-first algoritme, niettegenstaande het succes van best-first strategieën bij andere toepassingen.

In dit onderzoek worden een depth-first algoritme, Alpha-Beta, en een best-first algoritme, SSS*, nader beschouwd. De heersende opinie zegt dat SSS* in potentie efficiënter zoekt, maar dat de ingewikkelde formulering en het exponentiële geheugengebruik het tot een onpraktisch algoritme maken. Dit onderzoek laat zien dat er een verrassend eenvoudige relatie tussen de twee algoritmen bestaat: SSS* is te beschouwen als een speciaal geval van Alpha-Beta. Empirisch vervolgonderzoek toont aan dat de heersende opinie over SSS* fout is: het is geen ingewikkeld algoritme, het gebruikt niet te veel geheugen, maar het is ook niet efficiënter dan depth-first algoritmen.

In de loop der jaren heeft onderzoek naar Alpha-Beta vele verbeteringen opgeleverd, zoals transpositietabellen en minimale zoekwindows met her-zoeken. Deze verbeteringen maken het mogelijk dat een depth-first procedure gebruikt kan worden om de zoekruimte op een best-first manier te doorlopen. Op basis van deze inzichten wordt een nieuw algoritme gepresenteerd, MTD(f), dat beter presteert dan zowel SSS* als NegaScout, de in de praktijk meest gebruikte Alpha-Beta variant.

Naast best-first strategieën behandelt dit proefschrift ook andere mogelijkheden om minimax algoritmen te verbeteren. In het algemeen wordt aangenomen dat geen enkel algoritme dat de minimax waarde wil vaststellen efficiënter kan zijn dan de best-case van Alpha-Beta – dit is de zogeheten minimale boom. In de praktijk is deze opvatting niet juist. De echte minimale graaf die de minimax waarde bepaalt blijkt aanmerkelijk kleiner te zijn dan Alpha-Beta's best-case. Dientengevolge is er meer ruimte voor verbetering van minimax zoekalgoritmen dan in het algemeen wordt aangenomen.

The Tinbergen Institute is the Netherlands Research Institute and Graduate School for Economics and Business, which was founded in 1987 by the Faculties of Economics and Econometrics of the Erasmus University in Rotterdam, the University of Amsterdam and the Free University in Amsterdam. The Institute is named after the late Professor Jan Tinbergen, Dutch Nobel Prize laureate in economics in 1969. The Tinbergen Institute is located in Amsterdam and Rotterdam. Copies of the books which are published in the Tinbergen Institute Research Series can be ordered through Thesis Publishers, P.O. Box 14791, 1001 LG Amsterdam, The Netherlands, phone: +3120 6255429; fax: +3120 6203395.

The following books already appeared in this series:

1. O.H. SWANK, *Policy makers, voters and optimal control, estimation of the preferences behind monetary and fiscal policy in the United States.*
2. J. VAN DER BORG, *Tourism and urban development. The impact of tourism on urban development: towards a theory of urban tourism, and its application to the case of Venice, Italy.*
3. A. JOLINK, *Liberté, Egalité, Rareté. The evolutionary economics of Léon Walras.*
4. R.B. BUITENDIJK, *Towards an effective use of relational database management systems.*
5. R.M. VERBURG, *The two faces of interest. The problem of order and the origins of political economy and sociology as distinctive fields of inquiry in the Scottish Enlightenment.*
6. H.P. VAN DALEN, *Economic policy in a demographically divided world.*
7. P.J. VERBEEK, *Two case studies on manpower planning in an Airline.*
8. M.W. HOFKES, *Modelling and computation of general equilibrium.*
9. T.C.R. VAN SOMEREN, *Innovatie, emulatie en tijd. De rol van de organisatorische vernieuwingen in het economische proces.*
10. M. VAN VLIET, *Optimization of manufacturing system design.*
11. R.M.C. VAN WAES, *Architectures for Information Management. A pragmatic approach on architectural concepts and their application in dynamic environments.*
12. K. NIMAKO, *Economic change and political conflict in Ghana 1600-1990.*
13. J.M.C. VOLLERING, *Care services for the elderly in the Netherlands. The PACKAGE model.*
14. S. ZHANG, *Stochastic queue location problems.*
15. C. GORTER, *The dynamics of unemployment and vacancies on regional labour markets.*
16. P. KOFMAN, *Managing primary commodity trade (on the use of futures markets).*
17. P.Th. VAN DE LAAR, *Financieringsgedrag in de Rotterdamse maritieme sector, 1945-1960.*
18. P.H.B.F. FRANSES, *Model selection and seasonality in time series.*
19. P.W. VAN WIJCK, *Inkomensverdelingsbeleid in Nederland. Over individuele voorkeuren en distributieve effecten.*
20. A.E. VAN HEERWAARDEN, *Ordering of risks. Theory and actuarial applications.*
21. J.C.J.M. VAN DEN BERGH, *Dynamic models for sustainable development.*
22. H. XIN, *Statistics of bivariate extreme values.*
23. C.P. VAN BEERS, *Exports of developing countries. Differences between South-South and South-North trade and their implications for economic development.*

24. L. BROERSMA, *The relation between unemployment and interest rate. Empirical evidence and theoretical justification.*
25. E. SMEITINK, *Stochastic models for repairable systems.*
26. M. DE LANGE, *Essays on the theory of financial intermediation.*
27. S.J. KOOPMAN, *Diagnostic checking and intra-daily effects in time series models.*
28. R.J. BOUCHERIE, *Product-form in queueing networks.*
29. F.A.G. WINDMEIJER, *Goodness of fit in linear and qualitative-choice models.*
30. M. LINDEBOOM, *Empirical duration models for the labour market.*
31. S.T.H. STORM, *Macro-economic considerations in the choice of an agricultural policy.*
32. H.E. ROMEIJN, *Global optimization by random walk sampling methods.*
33. R.W. VAN ZIJP, *Austrian and new classical business cycle theories.*
34. J.A. VIJLBRIEF, *Unemployment insurance and the Dutch labour market.*
35. G.E. HEBBINK, *Human capital, labour demand and wages. Aspects of labour market heterogeneity.*
36. J.J.M. POTTERS, *Lobbying and pressure: theory and experiments.*
37. P. BOSWIJK, *Cointegration, identification and exogeneity. Inference in structural error correction models.*
38. M. BOUMANS, *A case of limited physics transfer. Jan Tinbergen's resources for re-shaping economics.*
39. J.B.J.M. DE KORT, *Edge-disjointness in combinatorial optimization: problems and algorithms.*
40. J.F.J. DE MUNNIK, *The valuation of interest rates derivative securities.*
41. J.C.A. POTJES, *Empirical studies in Japanese retailing.*
42. J.-K. MARTIJN, *Exchange-rate variability and trade: Essays on the impact of exchange-rate variability on international trade flows.*
43. J.B.L.M. VERBEEK, *Studies on economic growth theory. The role of imperfections and externalities.*
44. R.H. VAN HET KAAR, *Medezeggenschap bij fusie en ontvlechting.*
45. F. KALSHOVEN, *Over Marxistische economie in Nederland, 1883-1939.*
46. W. SWAAN, *Behaviour and institutions under economic reform. Price regulation and market behaviour in Hungary.*
47. J.J. CAPEL, *Exchange rates and strategic decisions of firms.*
48. M.F.M. CANOY, *Bertrand meets the fox and the owl. Essays in the theory of price competition.*
49. H.M. KAT, *The efficiency of dynamic trading strategies in imperfect markets.*
50. E.A.M. BULDER, *The social economics of old age: strategies to maintain income in later life in the Netherlands 1880-1940.*
51. J. BARENDREGT, *The Dutch Money Purge. The monetary consequences of German occupation and their redress after liberation, 1940-1952.*
52. Nanda PIERSMA, *Combinatorial optimization and empirical processes.*
53. M.J.C. SIJBRANDS, *Strategische en logistieke besluitvorming. Een empirisch onderzoek naar informatiegebruik en instrumentele ondersteuning.*

54. H.J. VAN DER SLUIS, *Heuristics for complex inventory systems. Deterministic and stochastic problems.*
55. E.F.M. WUBBEN, *Markets, uncertainty and decision-making. A history of the introduction of uncertainty into economics.*
56. V.J. BATELAAN, *Organizational culture and strategy. A study of cultural influences on the formulation of strategies, goals, and objectives in two companies.*
57. R.M. DE JONG, *Asymptotic theory of expanding parameter space methods and data dependence in econometrics.*
58. D.P.M. DE WIT, *Portfolio management of common stock and real estate assets. An empirical investigation into the stochastic properties of common stock and equity real-estate.*
59. A. LAGENDIJK, *The internationalisation of the Spanish automobile industry and its regional impact. The emergence of a growth-periphery.*
60. B.M. KLING, *Life insurance, a non-life approach.*
61. J.H. GROOTENDORST, *De markthuur op kantorenmarkten in Nederland.*
62. M. DINGENA, *The creation of meaning in advertising. Interaction of figurative advertising and individual differences in processing styles.*
63. R.T. LIE, *Economische dynamiek en toplocaties. Locatiekarakteristieken en prijsontwikkeling van kantoren in een aantal grote Europese steden.*
64. R.L.M. PEETERS, *System identification based on Riemannian geometry: theory and algorithms.*
65. O. MEMEDOVIC, *On the theory and measurement of comparative advantage. An empirical analysis of Yugoslav trade in manufactures with the OECD countries, 1970-1986.*
66. S. FISCHER, *The paradox of information technology management.*
67. P.A. STORK, *Modelling international financial markets: an empirical study.*
68. R.A. BELDERBOS, *Strategic trade policy and multinational enterprises: essays on trade and investment by Japanese electronics firms.*
69. I.T. VAN DEN DOEL, *Dynamics in cross-section and panel data models.*
70. M.W. DELL, *Maximum price regulations and resulting parallel and black markets.*
71. Bo CHEN, *Worst case performance of scheduling heuristics.*
72. P.W. CHRISTIAANSE, *Strategic advantage and the exploitability of information technology. An empirical study of the effects of IT on supplier-distributor relationships in the US airline industry.*
73. L. LEI, *User participation and the success of information system development. An integrated model of user-specialist relationships.*
74. J.H. BAGGEN, *Duurzame mobiliteit. Duurzame ontwikkeling en de voorzieningenstructuur van het personenvervoer in de Randstad.*
75. R.A. BOSCHMA, *Looking through a window of locational opportunity. A long-term spatial analysis of techno-industrial upheavals in Great-Britain and Belgium.*
76. C.A.G. SNEEP, *Innovation management in the Agro-food industry.*
77. F.R. KLEIBERGEN, *Identifiability and nonstationarity in classical and Bayesian econometrics.*
78. R.F. VAN DE WIJNGAERT, *Trade Unions and collective bargaining in the Netherlands.*

79. M. BOOGAARD, *Defusing the software crisis: Information systems flexibility through data independence.*
80. E.M. VERMEULEN, *Corporate risk management. A multi-factor approach.*
81. R.A. ZUIDWIJK, *Complementary triangular forms for pairs of matrices and operators.*
82. M.H. GOEDHART, *Financial planning in divisionalised firms: models and methods.*
83. E. EGGINK, *A symmetric approach to the labor market: an application of the sem method.*
84. A.F. CORRELJÉ, *The Spanish Oil Industry: Structural change and modernization.*
85. A.H.M. LELIVELD, *Social security in developing countries; operation and dynamics of social security mechanisms in rural Swaziland.*
86. Y.M. PRINCE, *Price cost margins in Dutch manufacturing: with an emphasis on cyclical and firm size effects.*
87. W.E. KUIPER, *Farmers, prices and rational expectations.*
88. B. ROORDA, *Global total least squares. A method for the construction of open approximate models from vector time series.*
89. A.I. MARTINS BOTTO DE BARROS, *Discrete and fractional programming techniques for location models.*
90. J.A. DOS SANTOS GROMICHO, *Quasiconvex optimization and location models.*
91. R.B. KOOL, *Aspects of enlarging the market effects in the Dutch health care.*
92. B. LEEFTINK, *The desirability of currency unification theory and some evidence.*
93. André VAN VLIET, *Heuristic and optimal methods for the three-dimensional bin packing problem.*
94. C.C.J.M. HEYNEN, *Models for option evaluation in alternative price-movements.*
95. Yvonne VAN EVERDINGEN, *Adoption and diffusion of the European currency unit. An empirical study among European companies.*
96. Robin DE VILDER, *Endogenous business cycles.*
97. Joachim J. STIBORA and Albert DE VAAL, *Services and services trade: A theoretical inquiry.*
98. Ronald VAN DER BIE, *"Een doorlopende groote roes". De economische ontwikkeling van Nederland, 1913-1921.*
99. W.J. JANSEN, *International capital mobility and asset demand. Six empirical studies.*
100. Natasha E. STROEKER, *Second-hand markets for consumer durables.*
101. Annette VAN DEN BERG, *Trade union growth and decline in The Netherlands.*
102. Patrick VAN DER LAAG, *An analysis of refinement operators in inductive logic programming.*
103. Euro BEINAT, *Multiattribute value functions for environmental management.*
104. H.A. VAN KLINK, *Towards the borderless mainport Rotterdam. An analysis of functional, special and administrative dynamics in port systems.*
105. W.H.J. HASSINK, *Worker flows and the employment adjustment of firms. An empirical investigation.*
106. S. LIU, *Contributions to matrix calculus and applications in econometrics.*
107. Mirjam VAN PRAAG, *Determinants of successful entrepreneurship.*
108. Erik VERHOEF, *Economic efficiency and social feasibility in the regulation of road transport externalities.*

109. Ilaria BRAMEZZA, *The competitiveness of a European city and the role of urban management in improving the city's performance. The cases of the Central Veneto and Rotterdam regions.*
110. Guido BIESSEN, *East European foreign trade and system changes.*
111. André LUCAS, *Outlier robust unit root analysis.*
112. Tineke FOKKEMA, *Residential moving behaviour of the elderly: an explanatory analysis for the Netherlands.*
113. Turan EROL, *Exchange rate policy in Turkey. External competitiveness and internal stability studied through a macromodel.*
114. Thijs DE RUYTER VAN STEVENINCK, *The impact of capital imports on the Argentine economy, 1970–1989*
115. Dennis DANNENBURG, *Actuarial credibility models: evaluations and extensions.*
116. Carsten FOLKERTSMA, *On equivalence scales.*
117. Aske PLAAT, *Research Re: search & Re-search.*