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), outperforms 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.

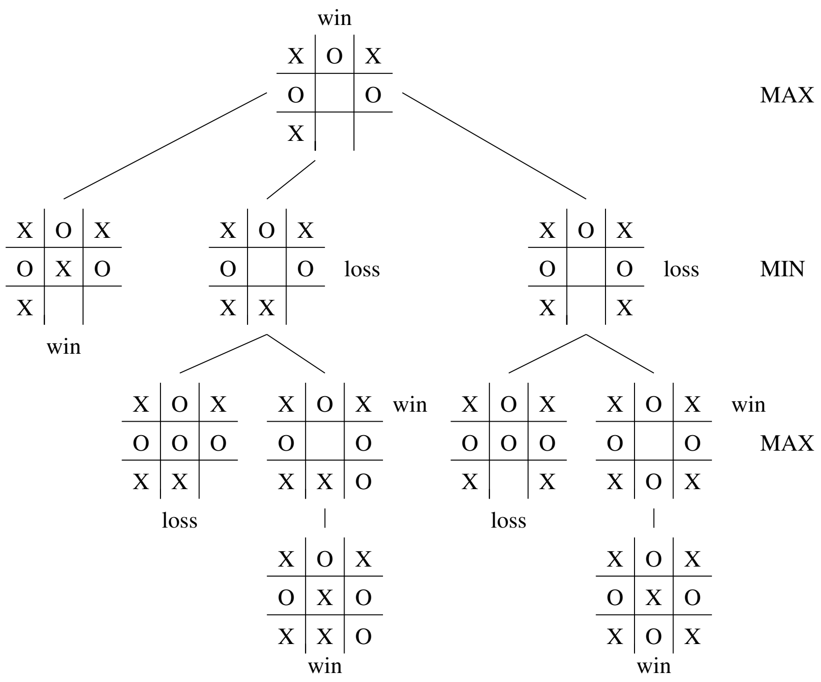

Figure 2.1: Possible Lines of Play for Tic-tac-toe Position

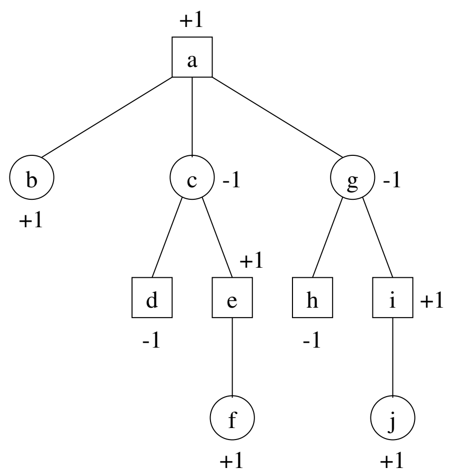

Figure 2.2: Minimax Tree of the Possible Lines of Play

```
function minimax(n) → f;
    if n = leaf then return eval(n);
    else if n = max then
        g := −∞;
        c := firstchild(n);
        while c ≠ ⊥ do
            g := max(g, minimax(c));
            c := nextbrother(c);
    else /* n is a min node */
        g := +∞;
        c := firstchild(n);
        while c ≠ ⊥ do
            g := min(g, minimax(c));
            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

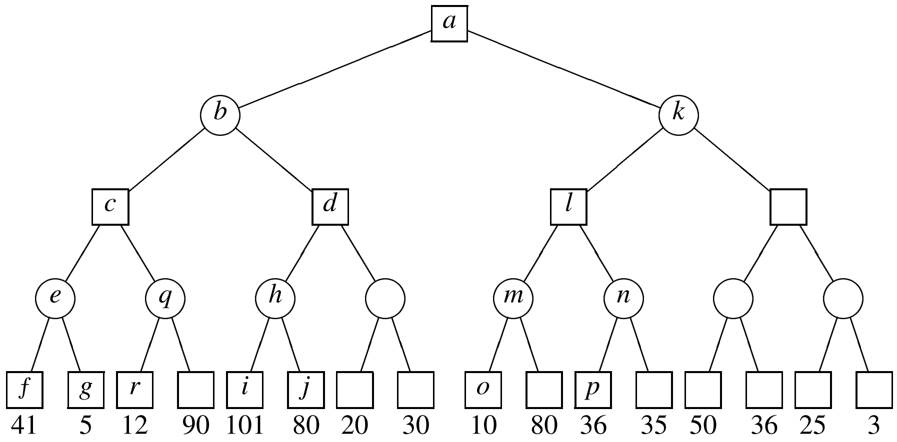

Figure 2.4: Example Tree for Bounds

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.

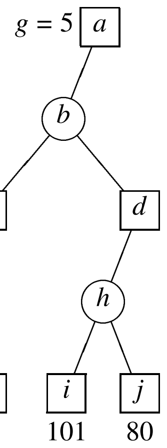

Figure 2.5: A Tree for a Lower Bound, a Min Solution Tree

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

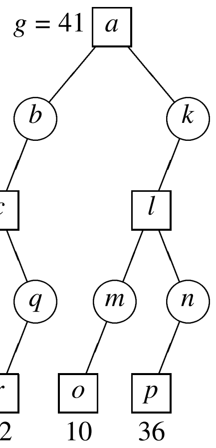

Figure 2.6: A Tree for an Upper Bound, a Max Solution Tree

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, 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 2.7: The Alpha-Beta Function

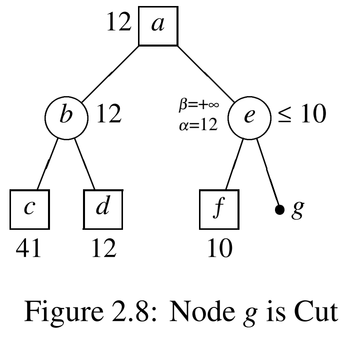

Figure 2.8: Node g is Cut Off

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

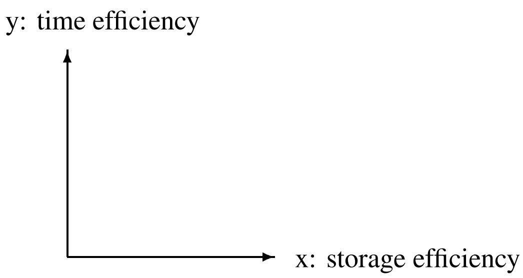

Figure 2.9: Two Performance Dimensions

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

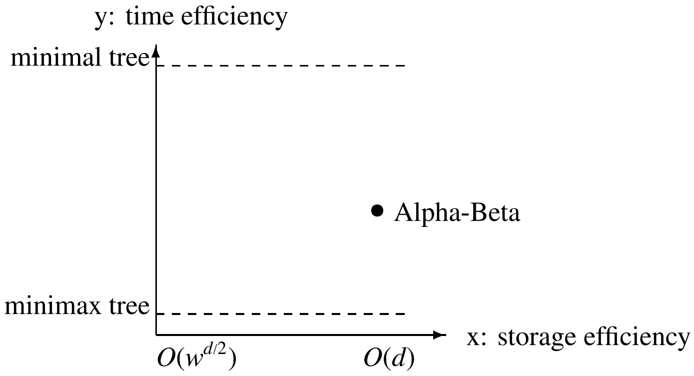

Figure 2.10: Alpha-Beta's Performance Picture

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) $\rightarrow f$;
    α := estimate $-$ delta;
    β := estimate + delta;
    g := alphabeta(n, α, β);
    if $g \leq \alpha$ **then**
        g := alphabeta($n, -\infty, g$);
    else if $g \geq \beta$ **then**
        g := alphabeta($n, g, +\infty$);
    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

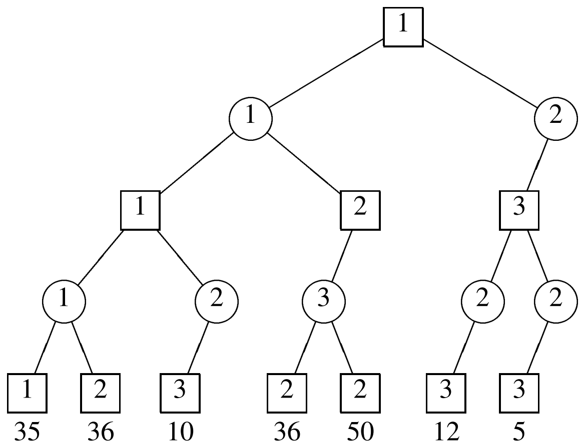

Figure 2.12: Minimal Tree (with node types)

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

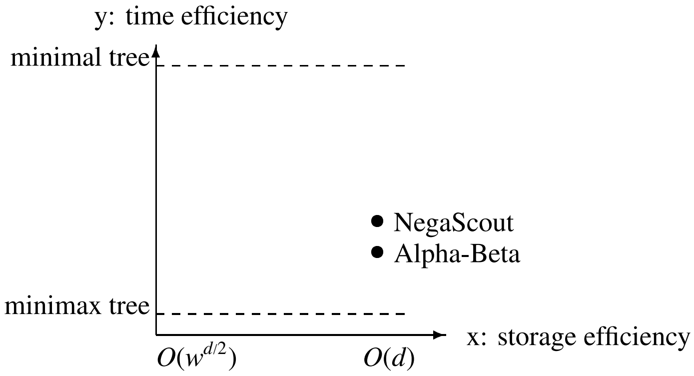

Figure 2.14: NegaScout's Performance Picture

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) $\rightarrow g$;
  if retrieve(n) = ok **then**
    if $n.f^- \geq \beta$ **then return** $n.f^-$;
    if $n.f^+ \leq \alpha$ **then return** $n.f^+$;
    $\alpha := \max(\alpha, n.f^-)$;
    $\beta := \min(\beta, n.f^+)$;
  if n = leaf **then** $g :=$ 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);
  if $g < \beta$ **then** $n.f^+ := g$;
  if $g > \alpha$ **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

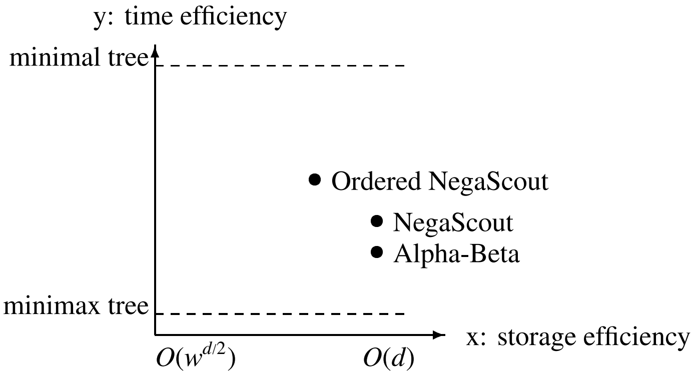

Figure 2.16: Performance Picture with Better Ordering

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

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

(5) Go to (2)

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

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 $\langle \text{next}(n), \text{LIVE}, \hat{h} \rangle$ 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 $\langle n, \text{SOLVED}, \min(\hat{h}, f(n)) \rangle$ on OPEN list (interior) in front of all states of lesser merit. Ties are resolved left-first.
5	s = 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]

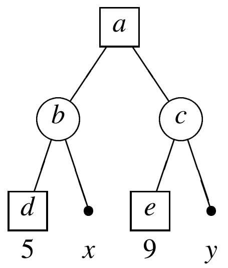

Figure 2.18: Which is "Best:" x or y?

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.

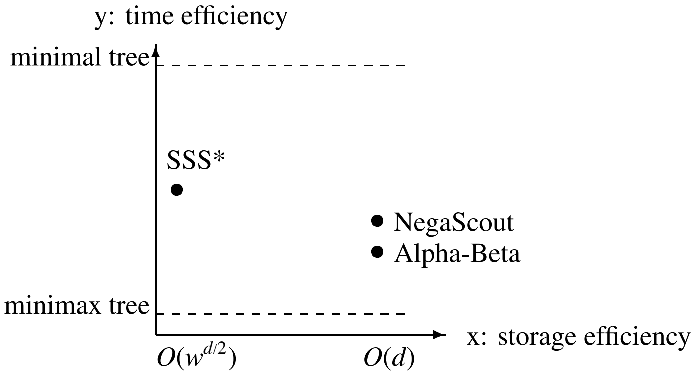

Figure 2.19: Performance Picture of Best-First Search Without Enhancements

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.