Control Strategies for Two-Player Games

BRUCE ABRAMSON

Computer Science Department, University of Southern California, Los Angeles, California 90089

Computer games have been around for almost as long as computers. Most of these games, however, have been designed in a rather ad hoc manner because many of their basic components have never been adequately defined. In this paper some deficiencies in the standard model of computer games, the minimax model, are pointed out and the issues that a general theory must address are outlined. Most of the discussion is done in the context of control strategies, or sets of criteria for move selection. A survey of control strategies brings together results from two fields: implementations of real games and theoretical predictions derived on simplified game-trees. The interplay between these results suggests a series of open problems that have arisen during the course of both analytic experimentation and practical experience as the basis for a formal theory.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*graphs*; *trees*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*pattern matching*; *sorting and searching*; I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems—*games*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search—*graph and tree search strategies*; *heuristic methods*; *plan execution, formation, generation*; J.m [**Computer Applications**]: Miscellaneous

General Terms: Algorithms, Experimentation, Theory

Additional Key Words and Phrases: Computer chess, control strategy, decision quality, game playing, game-trees, heuristic analysis, minimax algorithm, two-player games

INTRODUCTION: COMPUTER GAMES, WHY AND HOW?

In 1950, when computer science was still in its infancy and the term "artificial intelligence" had yet to be coined, Claude Shannon published a paper called "Programming a Computer for Playing Chess" [Shannon 1950]. He justified the study of chess programming by claiming that, aside from being an interesting problem in its own right, chess bears a close resemblance to a wide variety of more significant problems, including translation, logical deduction, symbolic computation, military decision making, and musical composition. Skillful performance in any of these fields is generally considered to require thought, and satisfactory solutions, although usually attainable, are rarely trivial. Chess has certain attractive features that these more complex tasks do not: The available options (moves) and goal (checkmate) are sharply defined, and the discrete model of chess fits well into a modern digital computer. Shannon then went on to outline the basics of this model and describe a method by which chess could be implemented on a computer.

The discrete model to which Shannon referred is called a game-tree, and it is the general mathematical model on which the theory of two-player zero-sum games of perfect information is based [von Neumann

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 0360-0300/89/0600-0137 $01.50

CONTENTS

INTRODUCTION:
COMPUTER GAMES, WHY AND HOW?
1. BACKGROUND: BASIC GAME-TREE SEARCHING PROCEDURES
2. GAME PROGRAMMING
2.1 Type-A Strategies: Full-Width Minimax Searches
2.2 Type-B Strategies: Selective Searches
3. THE ANALYSIS OF HEURISTIC SEARCH
3.1 The Benefits of Lookahead
3.2 Alternatives to Minimax
4. DISCUSSION: RELATING THEORY TO PRACTICE
5. AREAS FOR FUTURE INVESTIGATION
ACKNOWLEDGMENTS
REFERENCES

and Morgenstern 1944]. Chess belongs to this class of games; it is perfect information because all legal moves are known to both players at all times, and it is zero sum because one player's loss equals the other's gain. At the top of the chess tree is a single *root* node, which represents the initial setup of the board. For each legal opening move, there is an arc leading to another node, corresponding to the board after that move has been made. There is one arc leaving the root for each legal opening, and the nodes that they lead to define the setups possible after one move has been made. More generally, a game-tree is a recursively defined structure that consists of a root node representing the current state and a finite set of arcs representing legal moves. The arcs point to the potential next states, each of which, in turn, is a smaller game-tree. The number of arcs leaving a node is referred to as its *branching factor*, and the distance of a node from the root is its *depth*. If b and d are the average branching factor and depth of a tree, respectively, the tree contains approximately b^d nodes. A node with no outgoing arcs is a *leaf*, or terminal node, and represents a position from which no legal moves can be made. When the current state of the game is a leaf, the game terminates. Each leaf has a value associated with it, corresponding to the payoff of that particular outcome. Technically, a game can have any payoff (say a dollar value associated with each outcome), but for most standard parlor games, the values are restricted to WIN and LOSS (and sometimes DRAW).

In two-player games, the players take turns moving, or alternate choosing next moves from among the children of the current state. In addition, if the game is zero sum, one player attempts to choose the move of maximum value, and the other that of minimum value. A procedure that tells a player which move to choose is a strategy for controlling the flow of the game, or a *control strategy*. In principle, the decision of which choice to make is a simple one. Any state one move away from a leaf can be assigned the value of its best child, where best is either the maximum or the minimum, depending on whose turn it is. States two moves away from the leaves then take on the value of their best children, and so on, until each child of the current state is assigned a value. The best move is then chosen. This method of assigning values and choosing moves is called the *minimax algorithm*, and it defines the optimal move to be made from each state in the game. An example of the minimax algorithm is shown in Figure 1 on the tree of Nim, a simple game that plays an important role in the mathematical theory of games [Berlekamp et al. 1982]. Unlike Nim, however, most interesting games generate trees that are too large to be searched in their entirety, and thus an alternative control strategy must be adopted. The checkers tree, for example, contains roughly 10^{40} moves, and the chess tree in the neighborhood of 10^{120} [Nilsson 1980]. In these games, the tree is searched to some limit, and domain-specific heuristic information is applied to *tip* nodes (nonleaf nodes at the search frontier that cannot be expanded due to computational constraints) to return an estimated value as calculated by a *static evaluation function*. The control strategy must base its decisions on these estimates, rather than on the actual values.

When the tip values are exact, the tree is *complete*. Otherwise, the tips are internal

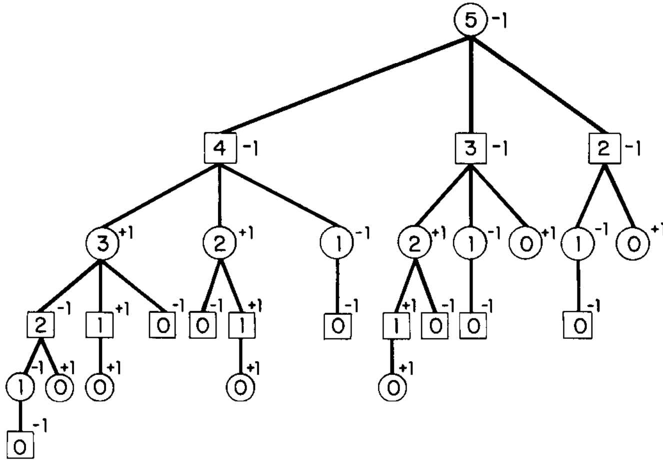

Figure 1. The game of Nim is played with five stones. The players take turns removing one, two, or three stones from play. The player removing the last stone loses. Thus, a circle containing a 0 is a win for circle (Max), and a square with a 0 is a win for square (Min). Max wins are denoted by +1, Min wins by −1, and the minimax value of each node is drawn to the right of the node. Since the root has a minimax value of −1, Nim is a forced win for the player moving second (Min).

nodes, and the tree is *partial.*[1] Complete trees are well understood but rarely applicable. Applications must rely on partial trees, which are invariably implemented in an ad hoc manner. Partial game-trees have appeared primarily in two subfields of artificial intelligence, game programming and the analysis of heuristic search methods. The emphases of these fields are quite different. Game programming is concerned with the development of computer programs that play specific games well, it is hoped at or beyond the level of a human expert. Thus, the models that have been studied are "real" games (typically chess), and the metric for success is performance versus machine or human opponents. Heuristic analysis, on the other hand, is concerned with investigating the accuracy of heuristic techniques, as compared to some ideal. Since the trees generated by interesting games tend to be both too large for the ideal to be calculated, and too complex to be dealt with analytically, simplified models and "artificial" games have been defined. Both fields have contributed interesting results about control strategies. Nevertheless, the interplay between them has been minimal; games that have actually been implemented have not been analyzed, and theoretical predictions have not been considered when the implementations were designed. Because of this, very little about the general theory of partial game-trees is known.

The various control strategies that have been used for two-player games are surveyed in this paper. Some basic game-tree search procedures are described in Section 1. Strategies that have been implemented or proposed in the context of real games are discussed in Section 2, while some theoretical results derived on simple models and strategies motivated by these results are outlined in Section 3. Areas in which interaction between the applied and theoretical aspects of the field could be

[1] Technically, it is possible to have internal nodes with exact values, as well. However, once the outcome of a game is known, the game is effectively over. Thus, any node with a known exact value can be treated like a leaf, and the distinction between complete and partial trees remains valid.

beneficial to both are discussed in Section 4, and some directions for future research are suggested in Section 5.

1. BACKGROUND: BASIC GAME-TREE SEARCHING PROCEDURES

Shannon's analysis included the description of two families of control strategies for the chess tree, type A and type B. A type-A strategy behaves as if the tip nodes are leaves, and applies minimax to the estimates calculated by the static evaluator. This involves a full-width, fixed-depth search (consideration of all possibilities up to a set distance away from the root), and uses heuristics only in assigning the values to the tips. Because the values it minimaxes are estimates, this technique does not always make the optimal move, and thus should be distinguished from minimax on complete trees, which does. For the sake of clarity, throughout the rest of this paper, minimax on partial trees will be referred to as *partial minimax*. The underlying assumption behind the use of partial minimax is that the estimates are reasonably accurate; the success of the strategy depends on the validity of this assumption. Type-B strategies, on the other hand, only consider reasonable moves. Heuristics are used not only to calculate tip values, but also to decide which moves are worth considering. Throughout most of the early history of chess programming, the general feeling was that whereas type-A strategies are easier to implement, type-B reasoning is necessary for expert performance [Berliner 1973; Shannon 1950]. The current state of the art, however, involves extremely powerful special-purpose architectures devoted primarily to playing type-A chess [Berliner and Ebeling 1986; Condon and Thompson 1983]. The level of expertise that some of these machines have attained (bordering on grandmaster as of early 1988 [Berliner 1988]), seems to disprove this earlier belief.

In the years since 1950, two important observations have led to innovative techniques that are now standard: There is an easily recognizable class of moves that will not be selected by minimax, and a preset search depth may not fully exploit the computational resources available. These observations led, respectively, to the development of *α–β pruning* and *iterative deepening search*. The exact origins of α–β are disputed, but the earliest papers in which it was discussed in detail were probably Edwards and Hart [1963] and Brudno [1963]. The α–β algorithm prunes by recording boundaries within which the minimax value of a node may fall. The parameter α represents a lower bound on the value that will be assigned to a maximizing node, and β an upper bound on the value of a minimizing node. Descendants whose minimax values fall outside the range are pruned, and their subtrees can be ignored. To ensure that the correct (minimax) choice is not missed, α and β start at minus and plus infinity, respectively, and are updated as the tree is traversed. Figure 2 shows an example of α–β pruning. The sensitivity of α–β to the order in which nodes are examined was first pointed out by Slagle and Dixon [1969]. The algorithm's behavior under several different orders was analyzed by Fuller et al. [1973], Knuth and Moore [1973], Newborn [1977], and Baudet [1978], where it was shown that in the best case, α and β cutoffs can actually double the search depth. On the average, however, α–β cuts the effective branching factor from b to approximately $b^{3/4}$, and allows the search depth to be extended by 25 percent. The relative efficiency of several of the algorithm's variations was discussed by Marsland [1983], while the asymptotic optimality of α–β over the class of all game searching algorithms, in terms of the average branching factor (i.e., averaged over all possible node orderings) was proved by Pearl [1982].

In addition to α–β, there are two other pruning algorithms worth mentioning, SSS* [Stockman 1979] and SCOUT [Pearl 1980], both of which have been shown to occasionally expand fewer nodes than α–β [Campbell and Marsland 1983; Ibaraki 1986; Roizen and Pearl 1983]. SSS*, first introduced by Stockman [1979], is a pruning algorithm that operates under the basic principles of a best-first search. Best-first strategies consider one of the issues that minimax completely failed to address: node

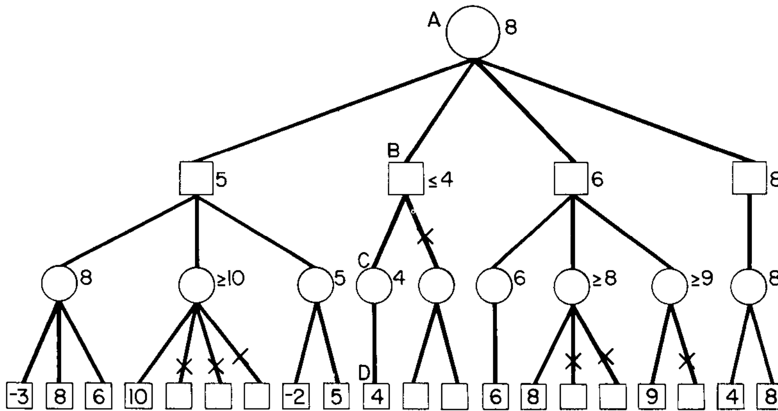

Figure 2. Minimax Search with α–β pruning: Static values are drawn inside the nodes, minimax values outside and to the right. Branches with X's through them have been pruned. To understand how a branch is pruned, consider nodes A, B, C, and D. D is a leaf, statically evaluated at 4. Since D is the only child of C, C gets a minimax value of 4. This means that B (a Min node) must have a value no greater than 4. Since A (a Max node) already has a child valued at 5, B will not be chosen, and its remaining children can be pruned.

expansion order. The order in which nodes are expanded is crucial to achieving optimal efficiency; node ordering is the sole determining factor as to whether α–β will attain either its best-case or its worst-case efficiency. In SSS*, tip node values are used for ordering nodes for further expansion—the one that appears to have the greatest merit is expanded first. As a result, SSS* will prune effectively even on trees in which the natural (left-to-right) ordering would render α–β useless. For a detailed description of SSS*, and an example of the algorithm at work see Pearl [1984]. Although SSS*, on the average, expands between one-third and one-half of the nodes that α–β does, the speedup is not sufficient to offset the additional bookkeeping and storage costs incurred. For these reasons, SSS* has yet to be used in a successful performance-oriented game program. Recent innovations in (relatively) space efficient versions of the algorithm [Bhattacharya and Bagchi 1986; Marsland et al. 1987], however, may eventually lead to a competitive program that uses SSS*.

The newest of these pruning algorithms, SCOUT [Pearl 1980], was motivated by the desire to reduce search effort by *testing* node values rather than by *evaluating* nodes. Although the difference between these approaches may not be immediately clear, full evaluation of a node involves generating all of its successors, while a node value may easily fail a test after only one (or a few) of its children have been generated. For a detailed explanation of SCOUT and an example of how it works see Pearl [1984]. Unlike SSS*, SCOUT has already proved itself useful to game programmers. Although pure SCOUT offers little in the way of speedup over α–β, some minor modifications to the algorithm can provide additional cutoffs [Reinefeld 1983]. Furthermore, SCOUT can be combined with another idea, *principal variation splitting* [Campbell 1981], to decompose game-trees in a manner that leads to efficient parallel minimax searches [Marsland 1986; Marsland and Campbell 1982]. The resultant algorithm, known as PVS (for principal variation search), maximizes the portion of the game-tree that can be pruned by attempting to rapidly determine the best value of α (and β). Since α and β define a window within which the (partial) minimax value must lie, the smaller the window, the greater the pruning power. PVS refines α

(or β) by searching several ply along the principal variation (i.e., the path that appears most promising), before searching the rest of the tree. PVS, which is basically a window management scheme that is equally applicable to sequential and parallel machines, is frequently implemented on parallel chess machines via its companion processor allocation algorithm, *PV-split*. The benefit of PVS is that the best possible α values are used during the search of all nonprincipal variations; its major drawback lies in the increased amount of idle processor time. (In a parallel implementation of PVS, processors dedicated to searching nonprincipal variations remain idle while waiting to be notified of a good value for α [Marsland and Campbell 1982].)

PVS makes use of ideas that were first investigated in the context of several different algorithms, including α–β, SCOUT, and aspiration search (discussed in Section 2.2.1), to develop a powerful parallel search strategy [Marsland 1986; Marsland and Campbell 1982]. Parallel algorithms for game-tree pruning, with a particular emphasis on their applicability to chess, have been surveyed by Marsland [1986], Marsland and Campbell [1982], and Marsland and Popowich [1985]. Issues related to interprocessor communication [Newborn 1985], speedups due to parallel pruning [Schaeffer 1986], limitations on the number of processors that can be effectively used to speed up pruning algorithms, and some interesting uses for processors beyond that limit [Schaeffer 1987] have all been discussed in the context of chess programs. Despite the many recent developments in pruning algorithms, parallel computing, and clever table lookup schemes [Marsland 1986; Marsland and Campbell 1982], however, the basic control strategy underlying these programs remains the same: partial minimax. The major benefit accrued by speedup has been the extension of the depth of minimax searches by several ply (and, of course, the corresponding increase in the strength of chess programs).

Unlike the pruning algorithms, which are unique to two-player games, iterative deepening is a completely general search paradigm. Iterative deepening allows the search to proceed until a preset time, rather than a preset depth, is reached. This is accomplished by first performing a full-width (or α–β) search to depth 1, then to depth 2, then to depth 3, etc. When the procedure times out, it makes the move that was singled out as best by the deepest search completed. The advantage of using iterative deepening in games was first demonstrated by Chess 4.5 [Slate and Atkin 1977], one of the most powerful chess programs of the 1970s. Although the technique quickly became a standard component of chess programs, it was in the context of one-player games that it was first analyzed. This analysis, which proved that iterative deepening is a time (number of nodes expanded) and space (amount of bookkeeping required) optimal tree search [Korf 1985], provided the first formal explanation of the algorithm's power.

The importance of these refinements to minimax is twofold. In their pure form, they have become part of the standard implementation of type-A strategies, crucial to the development of some very powerful programs discussed in Section 2.1. In this context, they address the issue of search efficiency; by increasing the efficiency of partial minimax, they allow larger portions of the tree to be searched, and more accurate decisions to be made. In addition, various modifications to α–β have formed the basis of many of the type-B strategies discussed in Section 2.2. These modifications generally are not guaranteed to return the same value as would a minimax search, and raise some interesting questions related to decision quality.

2. GAME PROGRAMMING

Even partial game-trees have their limits—programs that play backgammon [Berliner 1980], Go, Scrabble, and poker [Bramer 1983] have used significantly different models. Their practical applicability seems to be restricted to perfect information games with "manageable" branching factors, such as chess, checkers, kalah, and Othello. Although some interesting machine learning experiments have been run

using checkers as the example domain [Griffith 1974; Samuel 1959, 1967], the game that has generated the most interest, from Shannon's article on, is chess. In 1975, Newborn wrote that "all the chess programs that have ever been written and that are of any significance today are based on Shannon's ideas," and "improvements in programs are due primarily to advances in computer hardware, software, and programming efforts, rather than fundamental breakthroughs in how to program computers to play better" [Newborn 1975, p. 11]. To a great extent, this is still true in 1988. Nearly all control strategies contain an element of partial minimax; the major distinction between them is whether all paths are searched to the same depth (type A) or not (type B), and if not, what the criterion for expanding nodes is.

At the heart of every type-A strategy lies a fixed-depth, full-width (partial) minimax search. In addition to α–β and iterative deepening, many powerful modern chess programs (such as Hitech [Berliner and Ebeling 1986] and Chiptest [Anantharaman et al. 1988]) augment their basic searches with a secondary, or *staged search* [Campbell and Marsland 1983; Kaindl 1983b; Marsland 1986]. A secondary search is exactly what it sounds like: a second search started at a point other than the root, generally conducted to validate the quality of the move that appears best [Greenblatt et al. 1967; Levy 1976]. A variety of different heuristics have been proposed as bases of secondary searches. The one that seems to be enjoying the greatest degree of current popularity, however, is probably the *null move*, which allows a player to pass (i.e., not move) [Beal 1987]; a line of play in which a player given two consecutive moves still cannot do anything useful can generally be cut off without sacrificing much in the way of accuracy.[2] Experiments with the null-move heuristic have shown it to be useful at tactical chess [Beal 1987; Goetsch and Campbell 1988; Schaeffer 1987], and thus a powerful technique for augmenting brute-force searches; it can verify that the move believed to be (strategically) optimal at the search horizon remains (tactically) sound several ply deeper in the tree. Although secondary searches technically violate the fixed-depth characterization, strategies that use them can still be classified as type A. Fixed-depth, full-width minimax is comparatively easy to implement and conceptually simple to justify—as the estimates approach exact, the procedure approaches optimal performance. Errors occur only in static evaluation. The major drawback to type-A strategies is the amount of computation required for a full-width search; even with the help of a pruning algorithm, the large branching factor in most games limits the search to a relatively small portion of the tree.

Type-B strategies, on the other hand, constitute a rather large and diverse family. Their common feature is that they expand only the most promising lines of play. Node *expansion* is defined as the *generation* of a node's children. In a full-width search, first the root is expanded by generating all of its children. Then all nodes at depth 1 are expanded, generating all depth-2 nodes, etc. In chess, a variety of domain-specific techniques like *transposition tables* and *the killer heuristic* have been used to generate moves in an efficient order; moves that, in the past, have resulted in cutoffs are generated first [Marsland and Campbell 1982]. This approach maximizes the likelihood that a line of play which will eventually be cut off is cut off quickly, thereby saving a great deal of wasted search effort. When used in conjunction with a type-A strategy, transposition tables and the killer heuristic help order nodes in a manner that maximizes the pruning power of α–β. When they are combined with type-B strategies, however, they result in incremental move generation and searches along only promising lines of play: an approach that leads to deep, focused searches—a highly desirable combination in a secondary search.

One of the first computer chess matches, designed in part to compare the performances of the two strategy types, pitted the Kotok–McCarthy type-B program against the ITEP type-A program in four games.

[2] In chess, a position in which all moves are disadvantageous to the player moving is known as a *zugzwang* position.

When ITEP searched three ply, both games were draws. When it searched five ply, it won both [Newborn 1975]. The problem with the Kotok–McCarthy program was that it was not sufficiently selective in its choice of nodes to forward prune. In the words of former world chess champion Mikhail Botvinnik, "The rule for rejecting moves was so constituted that the machine threw the baby out with the bath water" [Botvinnik 1970, p. 3]. Thus, the domination of a type-A program over a type-B program does not necessarily indicate that selective strategies are inferior to those relying on brute force; it simply highlights the relative difficulty in implementing them.

2.1 Type-A Strategies: Full-Width Minimax Searches

Many of the earliest chess programs used type-A strategies and were able to achieve modest performance [Berliner 1978; Newborn 1975]. The major difficulty these programs faced was that practical computational limits were reached while search was still shallow. The standard measure of search depth is *ply*, or the number of consecutive moves considered. In an average chess game, each player has between 40 and 50 moves. A complete search, then, would have to exceed 80–100 ply. In most of these early programs, only three or four ply were searched. In addition, artificial termination of search at a uniform depth implies that anything not detectable at the search frontier is effectively nonexistent. Thus, these programs generally failed to realize when they were in the midst of a complex tactical maneuver, such as a material trade in chess. This problem is known as the horizon effect, and is a necessary consequence of the decision to terminate search uniformly [Berliner 1973]. The first method proposed to alleviate the effect was a secondary search, with the selected tip as root [Greenblatt et al. 1967]. This approach, however, does not remove the horizon; it merely extends it.

Positions that are not affected by the horizon are called *quiescent*, or quiet, because there is no imminent threat that will radically shift the game from what was anticipated at the horizon. The importance of applying the static evaluator only to quiescent positions was pointed out by Shannon; the issue of how to determine which positions are quiescent, however, is still largely open. Some attempts to resolve the problem are discussed by Kaindl [1983a]. Beal proposed consistency as a means for detecting quiescence. A node is consistent if its static value is the same as its backed-up value from a one-ply search [Beal 1980]. He later modified this *consistency search* to *locked-value search* [Beal 1982]. A value is locked if it has two children with the same best value. If this value is not correct, both of the children must have been evaluated incorrectly. Although there is no guarantee that this approach helps detect quiescence, it is generally safe to assume that single errors are more likely to occur than double errors. Thus, a locked value is less likely to be an anomaly brought about by the horizon effect than a nonlocked value. (An idea similar to locked values led to an interesting type-B strategy, conspiracy search [McAllester 1985], which is discussed in Section 2.2.1.) A more common approach to quiescence detection and correction is to perform some sort of secondary search beyond the frontier for positions that include captures, checks, or move promotions, and to consider all other positions quiescent. Many programs, including Chess 4.5, used this method [Slate and Atkin 1977].

Despite these difficulties, many successful programs have used type-A strategies. One of the general assumptions underlying them all is that the deeper the search, the better the performance. Although it was believed at one point that the limits of brute force search would be reached long before a computer could play master level chess [Berliner 1973; Botvinnik 1984], state-of-the-art technology has resulted in special-purpose architectures that have done just that. The first such machine, Belle, relied almost totally on speed to become the first computer to achieve master rating [Condon and Thompson 1982, 1983]. Speed allowed Belle to search to a previously unachievable eight ply, make more accurate decisions, and apparently avoid the horizon effect

[Berliner 1981]. IAGO, an Othello program that plays at about world championship level, also used a standard full-width, α–β search with iterative deepening. The development of IAGO stressed analysis of positions, and resulted in a very strong static evaluation function. This function, combined with the relatively small tree of Othello (relative to chess, that is), accounts for the program's success [Rosenbloom 1982]. Chess 4.5, which introduced iterative deepening, also included a hash table to avoid redundant searches [Slate and Atkin 1977]. When a node is encountered, its value is entered in the table. If it is reached a second time, the subtree beneath it need not be searched. The avoidance of redundancy allows deeper searches to be performed without requiring additional time.

The *M & N procedure* attempts to improve performance by making better use of the information gathered at the tips, rather than by speedup. This is done by finding the minimax value and adding a bonus function to it, where the bonus is an experimentally derived function of the M maximum or N minimum values. Thus, the backed-up value contains information about the best several choices, not only the single best. Using the game of kalah as their example domain, Slagle and Dixon showed that this procedure improves play to about the level that would be achieved by extending partial minimax an additional ply. Its major drawback is that pruning techniques become more complicated and less helpful [Slagle and Dixon 1970]. The notion of saving multiple nodes was also used by Harris [1974] to devise *bandwidth search.* The idea underlying this search is that making the optimal choice is not always necessary, as long as one that is not too far from optimal is guaranteed. If an evaluation function with constant bounded error can be found, any node whose value is within those bounds of the currently most promising one may, in fact, be best. The original domain of bandwidth search was one-player games, where the idea of a constant bounded error was considered a weakening of the admissibility requirement. Whereas an admissible function never overestimates the true value, a function with constant bounded error never overestimates by more than e or underestimates by more than d. This scheme has the advantage of not discarding all moves right away, thereby allowing for occasional error recovery. Its disadvantage is that it does not search for the minimax value, but chooses the first node found within $(e + d)$ of it. The algorithm fared well in Four Score (a three-dimensional, 4-by-4 tictactoe game) competition [Harris 1974], but holds little promise for more complex games because of the difficulty of finding heuristics that are guaranteed to satisfy the bandwidth conditions.

The current state-of-the-art of chess programming lies in special-purpose architectures, such as Hitech [Berliner and Ebeling 1986] and Chiptest [Anantharaman et al. 1988]. These machines use many of the techniques that have already been discussed, including parallel PVS searches, transposition tables, the killer heuristic, staged searches, etc. The architecture upon which Hitech is based is called *SUPREM*, an acronym for Search Using Pattern Recognition as the Evaluation Mechanism. SUPREM links two machines together, one smart but slow (the oracle), and one very fast (the searcher). The oracle is the source of game-specific analyses, and is responsible for downloading preprocessed pattern recognition data to the searcher, which then conducts a high-speed parallel search [Berliner and Ebeling 1986]. At the heart of these chess machines, then, lies a very deep, very fast, type-A strategy, which is augmented with an incrementally growing number of chess-specific heuristics for evaluation, move generation, node ordering, secondary searches, and handling of special cases in which the machine has shown a weakness [Anantharaman et al. 1988; Berliner 1988; Berliner and Ebeling 1986; Goetsch and Campbell 1988]. These heuristics constitute the machines' domain knowledge; as this knowledge increases, the number of incorrect moves made decreases, and the quality of play improves. The resultant combination of general search techniques and domain-specific information is quite powerful; Hitech is already close to grandmaster rating, and may reach that coveted level before this article appears in print [Berliner 1988].

In many respects, these dedicated chess machines constitute the culmination of several decades of research directed toward a specific goal: the development of grandmaster (and eventually world champion) chess-playing machines. Although these machines are significant as demonstrations of the first nonnumeric domain in which computers can outperform (nearly all) humans, much of the work currently being done to improve their performance is highly specific to chess; the domain-independent control strategy component appears to have been essentially solved. In my opinion, one of the unfortunate side effects of chess programmers' phenomenal success is that the field of computer *chess* in the 1980s has left the field of computer *games* behind. Although it is possible that some of the chess heuristics currently being considered as ways of improving machine performance will have implications to other games, wide applicability is clearly no longer the field's emphasis. Nevertheless, there are many things that the work on computer chess has taught, and continues to teach, researchers interested not only in other games, but in other areas of decision making, as well.

2.2 Type-B Strategies: Selective Searches

One of the features common to all type-B strategies is the use of domain-specific information to select promising nodes and lines of play. This approach is believed to be the method used by human experts [Berliner 1973, 1977b; Botvinnik 1970, 1984]. In addition, it muffles the combinatorial explosion by drastically reducing the effective branching factor, hopefully leading to improved performance. The domain-specific knowledge required to make type-B strategies work can be infused in various forms. Michie identified three types of knowledge that are useful in game programs: rote memory (dictionary entries of board positions), "theorems,"[3] and pattern knowledge [Michie 1977]. Hash tables of the type used by Chess 4.5 [Slate and Atkin 1977], the inclusion of standard book openings in Belle and other programs [Condon and Thompson 1982, 1983; Thompson 1982], and the endgame library of PIONEER [Botvinnik 1984] are all examples of successful uses of rote memory. In addition, large tables have proved very useful in machine learning experiments that develop better static evaluation functions for checkers [Griffith 1974; Samuel 1967]. In terms of the design of search strategies that attempt to mimic human approaches to problem solving, however, rote memory is not particularly helpful. Heuristics and pattern recognition, on the other hand, have each led to the development of some interesting strategies.

2.2.1 Forward-Pruning Strategies

Heuristics are general guidelines built into a program. Bratko and Michie [1980] wrote a program to play the chess endgame of KRKN (King and Rook vs. King and kNight), which included an advice table, or a list of general heuristics like "avoid mate." This is a rather nonstandard use of heuristics, however. Typically, they are included in the evaluation function and the forward pruning criteria [Berliner 1977a, 1977b], not in separate data structures. Forward pruning is a technique used by many type-B strategies. Unlike α–β, which only prunes nodes that will not be chosen, forward pruning techniques ignore all nodes that do not look very promising, thereby running the risk of missing the correct choice.

One heuristic that has been used to define a type-B strategy is to expand only nodes that look at least as good as the current best. This heuristic defines a technique called *razoring* [Birmingham and Kent 1977], a procedure that, at first glance, looks strikingly similar to α–β. In fact, the only difference between them lies in the criteria used for determining the promise of a node. α–β relies on backed up minimax values, razoring on the static evaluation. Thus, while razoring prunes nodes that do not look good, α–β only eliminates nodes that are not good. Unlike α–β, then,

[3] The term "theorems" is confusing, because the knowledge used is generally not a theorem in the mathematical sense. In my opinion, the term "heuristics" is more accurate, and will be used throughout the rest of the paper.

razoring cannot guarantee that it will find the minimax value.[4] Razoring should be used in addition to α–β, not instead of it. In the worst case, then, razoring will prune the same nodes as α–β, with only the added cost of some extra evaluations. In the average case, however, razoring will prune nodes earlier than α–β, narrow the branching factor more rapidly, and deepen search, all in exchange for occasionally missing the best choice. The preliminary experiments described by Birmingham and Kent [1977] showed that in the exchange, razoring gained, on the average, an order of magnitude over α–β in a four-ply tree, in terms of the number of nodes expanded. Since no further experimentation has been reported, the utility of razoring to state-of-the-art programs, which search to depths in excess of eight ply, is unknown. Razoring is illustrated in Figure 3.

Another idea that has been considered is to start the search with an idea about the true value of the root. This rule of thumb has already been discussed in the context of PVS, but its initial implementation was in a procedure called *aspiration search*, which was discussed and analyzed by Brudno [1963], Marsland [1983], and Marsland and Campbell [1982]. Its development was motivated by the observation that α–β works best if the node that will eventually be returned by minimax is among the first nodes examined [Slagle and Dixon 1969]. The reason for this is rather straightforward: If the best alternative is considered first, the α and β values are quickly set to define a narrow range around the minimax value of the root, thereby resulting in a great deal of pruning. The predetermined upper and lower bounds, then, can serve the roles of α and β. If the procedure used to determine these bounds is fairly accurate, the search tree can be narrowed quickly. However, because the bounds do not start at plus and minus infinity, it is possible that the initial estimate was wrong. Once again, the guarantee of returning the minimax value is lost. The use of this heuristic as a means of pruning absurd moves was discussed by Adelson-Velskiy et al. [1975]. Figure 4 shows how aspiration search can be used to augment α–β.

The B* algorithm [Berliner 1979] uses a simple heuristic of a very different nature, "terminate the search when an intelligent move can be made." This algorithm was motivated by the desire to avoid the horizon effect by defining natural criteria for terminating search. The search proceeds in a best-first manner, and attempts to prove that one of the potential next moves is, in fact, the best. By concentrating only on the part of the tree that appears to be most promising, B* (and best-first searches in general) avoids wasting time searching the rest of the tree. Berliner's adaptation of best-first searches to game-trees included the first modification to Shannon's original model. Instead of associating a single value with each node, B* uses two evaluation functions, one to determine an optimistic value, or upper bound, and one for a pessimistic value, or lower bound. The search is conducted with two proof procedures, PROVEBEST, which attempts to raise the lower bound of the most promising node above the upper bounds of its siblings, and DISPROVEREST, which tries to lower the upper bounds of the siblings beneath its lower bound. The search terminates when the most promising choice has been proven best. Figure 5 illustrates the use of these procedures. Although B* sounds particularly appealing from both the speedup and cognitive modeling viewpoints, it does have its dark side. Like all best-first searches, a good deal of storage space is needed to keep track of the promise of each node on the generated-not-expanded (*open*) list, so that the focus of the search can shift as necessary. More significantly, though, is that B* is not guaranteed to terminate before time runs out, and thus, like SSS*, is probably inapplicable to computer chess. If this occurs, it loses the edge of making intelligent decisions, and has to choose whatever looks best at the time. The success of B* lies, to

[4] Two points about razoring are probably worth noting. First, if razored nodes were restricted to those on the search horizon, speedups would not be as pronounced as those reported by Birmingham and Kent [1977], but fewer good nodes would be discarded. Second, even at the horizon, razoring fails at chess *zugzwang* positions [Birmingham and Kent 1977].

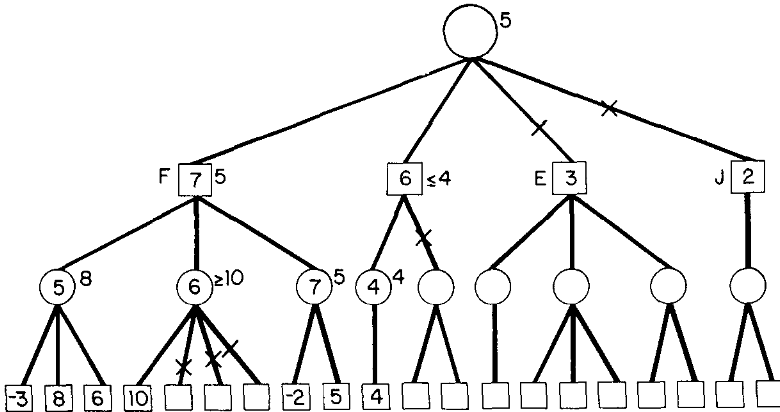

Figure 3. This is what happens when razoring is added to the pruning in Example 2. Node E's static evaluation was 3, which is worse than node F's backed up value of 5. Thus, it was pruned. The same is true for node J, whose static value is 2.

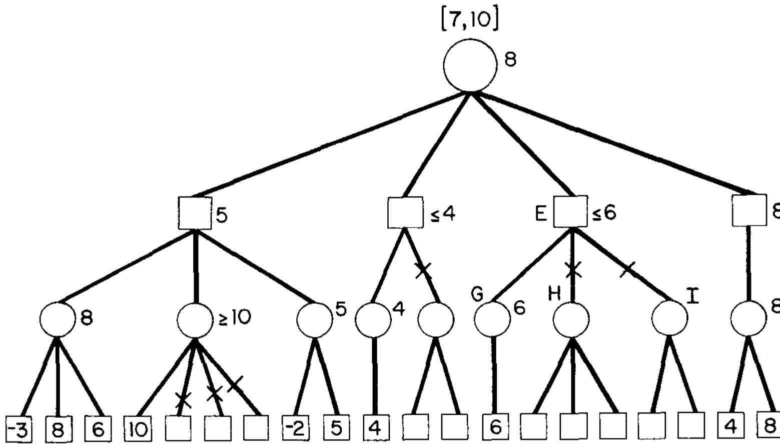

Figure 4. The $\alpha-\beta$ pruning of Figure 2 is now augmented with an aspiration search. The precomputed range of possible values for the root is [7, 10]. When node G is expanded, it becomes clear that the value of node E will not exceed 6. Since this falls outside the range of possible values, nodes H and I can be pruned.

a great extent, in its ability to correctly select the most promising node and most efficient proof procedure. Several variations that focused on proof procedure selection have been studied [Berliner 1979; Palay 1982], and a scheme that selected them probabilistically was shown to be somewhat stronger than one that made deterministic choices.

The advantage of ranges, of course, is that they contain more information than point probabilities. Palay extended this reasoning one step farther, and devised the idea of passing entire probability distributions. A distribution contains complete information about the likely location of a node's value, and thus retains considerably more information than just a range. He combined this idea with the control aspects of the B* algorithm to yield a powerful best-first search strategy, PSVB* [Palay 1985], and showed that by using distributions, an increase in efficiency of 91 percent over the use of ranges is possible.

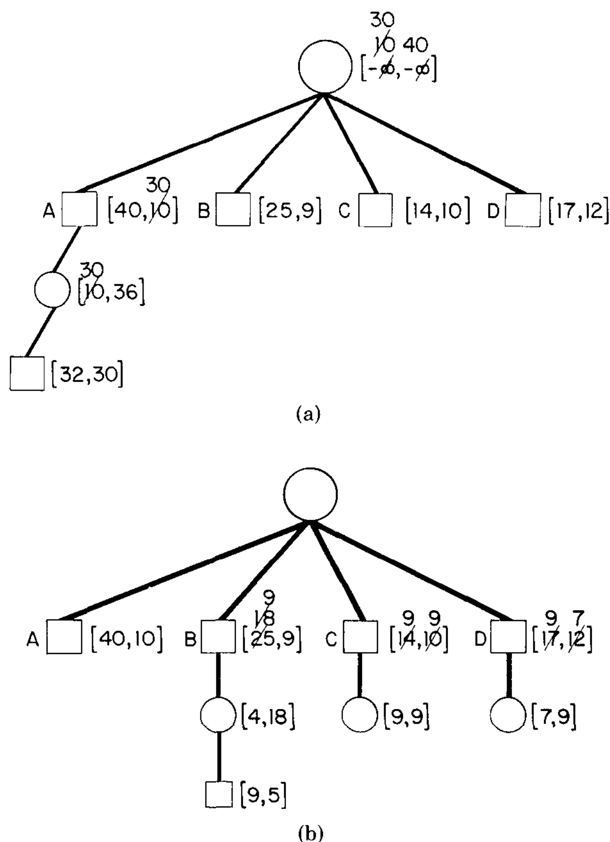

Figure 5. The proof procedures of B*: (a) PROVEBEST; (b) DISPROVEREST. Node A has the highest upper bound, and is thus the most promising node. In (a), A is expanded until its lower bound is greater than the upper bounds of nodes B, C, and D. In (b), the upper bounds of B, C, and D are pushed below the lower bound of A.

An interesting idea that has recently been suggested as a selection criterion is "attempt to stabilize the value of the root." This heuristic was used to develop a procedure called *conspiracy search* [McAllester 1985]. The value of a node is stable if deeper searches are unlikely to have any major effect on it. In a conspiracy search, the root's stability is measured in terms of conspiracy numbers, the number of leaves whose values must change to affect its (the root's) value. If the number of conspirators required to change the root value is above a certain threshold, the value is assumed to be accurate. At any given point during the search, the possible values of the root are restricted to the interval $[V_{min}, V_{max}]$, where V_{min} and V_{max} are the values of its minimum and maximum accessible descendants at the search frontier, respectively. To update the range, either prove that the minimizing player can avoid V_{max}, or that the maximizing player can avoid V_{min}. The decision of which to prove at each point can be made with the help of the conspiracy numbers. Unlike B*, there is no need to change the evaluation function (or to use multiple functions) to derive the interval; a single function will suffice. An example of conspiracy search is given in Figure 6. Alone among the procedures discussed in this section, conspiracy search

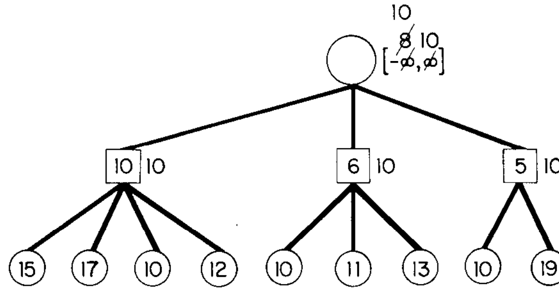

Figure 6. Conspiracy search. In this example, a node is considered stable if at least three leaves must change to affect it. When the tree is built out to depth 1, the range of possible values is [5, 10]. When built out another level, all three of the root's children get backed up values of 10. In order for the root value not to be 10, then three leaves must change (one grandchild from each child). Thus, the root is considered stable at 10.

has been neither analyzed nor implemented. This is not due to any fundamental flaws in the procedure, but rather to the fact that it was only recently proposed. However, the heuristic on which it is based sounds reasonable, and studies aimed at determining its value as a control strategy are expected to appear shortly.

Perhaps the newest of the type-B strategies is Rivest's use of *generalized p-means* to approximate minimax values [Rivest 1987]. Briefly stated, a p-mean is defined as follows:

Let $\mathbf{a} = (a_1, \ldots, a_n)$ be a vector of n positive real numbers.
Let p be a nonzero real number.
Then a generalized p-mean of $\mathbf{a}$, $M_p(\mathbf{a})$, is given by

$$M_p(\mathbf{a}) = \left(\frac{1}{n} \sum_{i=1}^{n} a_i^p\right)^{1/p}.$$

The two points that make $M_p(\mathbf{a})$ ideal for estimating minimax values are

$$\lim_{p\to\infty} M_p(\mathbf{a}) = \text{MAX}(a_1, \ldots, a_n)$$

and

$$\lim_{p\to-\infty} M_p(\mathbf{a}) = \text{MIN}(a_1, \ldots, a_n).$$

Unlike MAX or MIN, M_p has derivatives that are continuous everywhere. This makes M_p ideal for a formal sensitivity analysis, similar in spirit to the informal (or less formal) analyses of locked-value search [Beal 1982] and conspiracy search [McAllester 1985]. The partial derivative of a root node's p-mean with respect to a given tip node indicates the sensitivity of the root's value to a change in the value of that tip. Thus, a strategy that expands the most sensitive tip node should quickly refine a good estimate of the root's minimax value. In addition to guiding the growth of search trees along the most sensitive gradient, p-means also suggest a way of breaking ties among nodes with identical minimax values: Select the one with the greatest degree of stability. Rivest also reported some preliminary experiments on the game of Connect Four, in which a program using generalized p-means to approximate minimax values was pitted against a standard α–β program. In this tournament, α–β was shown to be superior if the programs were given equivalent amounts of time. If, however, the programs were allowed to examine an equal number of moves (in unequal time), the minimax approximation scheme emerged victorious [Rivest 1987]. No further experiments have been reported, although the idea of p-means and Rivest's Connect Four results do suggest several interesting questions.

2.2.2 Plan-Based Strategies

Although pattern knowledge has not been used as extensively as heuristic knowledge in the design of type-B strategies, several

interesting systems have used patterns to plan in game domains. For example, Bramer described an optimal program for the chess endgame KPK (King and Pawn vs. King) that used pattern knowledge [Bramer 1980]. The PARADISE chess program (PAttern Recognition Applied to DIrecting SEarch) [Wilkins 1977, 1980, 1982] relies almost completely on pattern recognition to direct search. Like the B* algorithm, PARADISE expresses node values as ranges, attempts to prove that one move is the best, and terminates search based on knowledge, not parameters (such as depth or time). Unlike B*, however, PARADISE uses a large collection of plans, or sequences of moves to be made from various positions, to avoid the errors caused by the horizon effect. To compensate for the expense of maintaining the knowledge base, a small tree is searched. This is possible because the use of plans drastically reduces the branching factor by relying on complete sequences of moves, rather than on individual nodes. Information is communicated from one part of the tree to another using a "hodgepodge" collection of cutoffs that control the search and indicate when searches along abandoned lines should be resumed. Because of the rather ad hoc nature of these criteria, PARADISE does not make a major contribution to the theory of control strategies; its major contributions are to the fields of planning and pattern recognition.

One major issue raised by PARADISE, though, is how to generate plans to store in a data base. PARADISE's plans generally lead to a goal other than winning the game. Pitrat [1977, 1980] devised a general scheme for generating plans, and showed a few examples of its application to a simplified chess domain. His program is given a description of the initial state, and told to find a combination of moves that will lead to a specific goal. In order to succeed, the program must be given a less ambitious goal than "win the game." If the goal cannot be met as given, the program fails. Thus, it seems that the successful implementation of pattern recognition knowledge is directly related to the definition of inexact, or approximate tasks, which it is hoped correspond to the ultimate goal of winning. Evidently, guiding plans toward a most promising node is considerably harder than guiding searches toward one. Whereas many search programs have performed reasonably well without defining specific goals, plans need them to succeed.

Botvinnik [1970, 1984] identified the development of inexact goals to guide inexact search as one of the most important problems in the design of intelligent systems. This points out a fundamental flaw in the original definition of partial game trees: There is no clear understanding of what a static evaluator is attempting to estimate. Ostensibly, it should approximate the actual value of the node (the value that would be returned by the minimax algorithm on the complete tree), although when a multivalued function is used to estimate a binary-valued one, it is unclear precisely what is being estimated. One precise, domain-independent model of static evaluation, which I have proposed in several articles, is the expected value of the leaves beneath the node being evaluated, or the *expected-outcome* model of two-player evaluators [Abramson 1987, 1988; Abramson and Korf 1987]. In general, two-player evaluators are described as domain-specific measures of a position's "worth" [Nilsson 1980], "merit," "strength" [Pearl 1984], "quality" [Winston 1977], or "promise" [Rich 1983]; evaluator design usually involves obtaining an expert assessment of significant domain features and their relative importance to the ultimate goal of winning the game. In Othello, for example, significant features often include equivalence classes of squares, mobility, and stability [Abramson 1987; Maggs 1979; Rosenbloom 1982], while chess evaluators frequently include material, pawn formation, mobility, and King safety [Hartmann 1987a, 1987b; Levy 1976; Shannon 1950].

The problem of developing static evaluators and that of determining inexact goals are strongly related; in the domain of chess, for example, Botvinnik identified material advantage—one of the most popular simple chess evaluators—as the inexact goal [Botvinnik 1984]. Botvinnik's analysis of chess is significant, because as a long-standing

world champion and a trained computer expert, he is among the few individuals who have combined domain knowledge with knowledge engineering skills. His discussion of inexact goals led to his development of the *chess master's method*, an ambitious control strategy that made heavy use of both heuristic and pattern knowledge [Botvinnik 1982]. The pattern knowledge component appears in the form of a game-specific action tree, which augments and directs the search tree. As the search tree grows, the action tree records purposeful moves, or goals that can be attained without exhausting resources. Heuristic knowledge is available through three general limitation principles: (1) if improvement is possible, it is contained in the tree; (2) new possibilities should be considered only if they promise an improvement; (3) only goals that do not exhaust the time limits should be considered. Botvinnik's scheme incorporates many of the ideas found in other control strategies. The limitation principles indicate a best-first approach to node expansion, and the action tree is similar to a dynamically constructed knowledge base of plans. Despite Botvinnik's prowess as a chess player, the resultant system, PIONEER, was never able to play competitive chess. It was, however, quite successful in the realm of planning maintenance repair schedules for power stations in the Soviet Union [Botvinnik 1984].

The communication between the (game-specific) action tree and (general) search tree parallels the relationship between the oracle and searcher in Hitech [Berliner and Ebeling 1986]. The difference between them, which may, in part, account for PIONEER's failure and Hitech's success, lies in the role of knowledge. PIONEER uses it for strategic purposes, Hitech for evaluation. In either case, the lesson is the same: A functioning intelligent system needs both a general methodology and domain-specific knowledge.

3. THE ANALYSIS OF HEURISTIC SEARCH

Throughout most of the 1950s, 1960s, and 1970s, all of the analytic work done on game-trees dealt with determining the efficiency of node ordering and α–β. During this period, a near-universal assumption was that partial minimax, although fallible, was the strongest conceivable control strategy, and its performance would improve directly with the length of lookahead and accuracy of the static evaluator. Alternative, or modified strategies, of the type discussed in Section 2.2 were motivated by the desire to either model cognitive activities [Newell et al. 1963] or improve performance faster than search depth could be extended [Berliner 1973]. The perception of partial minimax has undergone a radical change in the 1980s. The advent of parallel algorithms like PV-split and specialized chess architectures allowed lookahead to be lengthened, the effect of the horizon to be largely overcome, and computers to play master-level chess. By 1985, there were even commercially available chess machines playing at or near the master level [Kopec 1985]. At about the same time that these machines were convincing game programmers of the inherent power of full-width α–β minimax search, analyses of the procedure began questioning its theoretical accuracy [Nau 1983a].

One of the reasons that these analyses were so long in the coming is the complexity of the models. Games like chess were chosen as abstractions of the real world that were simple enough to allow computer simulation and experimentation. They are nowhere near simple enough to allow mathematical analysis. Thus, a new model had to be defined that was an abstraction of the game-tree. The most frequently studied model to date is the *(d, b, F)-tree* [Pearl 1980, 1984]. A (d, b, F)-tree has a uniform leaf depth d, a uniform branching factor b, and leaf values assigned by a set of identically distributed random variables drawn from a common distribution function, F. With the definition of the model and a family of "artificial" games that embody the simplifications, studies of (and challenges to) the accuracy of partial minimax became possible. In the 1980s, then, the primary motivation behind defining alternative strategies to minimax has become the discovery of a procedure that is correct

in some theoretical sense. This section is divided into two parts. In the first the results that caused minimax to fall into disfavor with theoreticians are outlined, and in the second some control strategies for partial game-trees that avoid these theoretical difficulties are described.

3.1 The Benefits of Lookahead

The aspect of partial minimax that has come under fire in the 1980s is lookahead. Lookahead is generally assumed to be helpful because it results in the proper move being chosen more often. Over the years, an impressive body of empirical evidence has been amassed to support the validity of this claim, all in the form of successful programs that rely on it. Nevertheless, recent analyses have uncovered some surprising results.

The potential futility of looking ahead was addressed by Pearl, who derived the *minimax convergence theorem* [Pearl 1980, 1984]. This theorem states that in a deep enough (d, b, F)-tree, the root value is essentially predetermined; the value is a function of b, and the variance a function of d. Specifically, as $d \rightarrow \infty$, if F is continuous (and assuming the same player always makes the last move), the minimax value of the root converges to the $(1 - \xi_b)$*-quantile* of F, where ξ_b is the solution of $x^b + x - 1 = 0$. If, on the other hand, F is discrete with values

$$v_1 < v_2 < \cdots < v_M, \text{ and } 1 - \xi_b \neq F(v_i)$$

for all i, the root's value converges to the smallest i satisfying

$$F(v_{i-1}) < 1 - \xi_b < F(v_i).$$

In a binary ($b = 2$) tree, for example, $1 - \xi_b = 1 - (\sqrt{5} - 1)/2 \approx 0.382$. Thus, if F is continuously distributed between 0 and 1, the root converges to 0.382. If F is restricted to the integers between 0 and 100, then $F(38) < 0.382 < F(39)$, and the root converges to 39. Perhaps the most interesting case occurs when F is a binary function, in which each leaf is a win ($v_W = 1$) with probability P, and a loss ($v_L = 0$) with probability $1 - P$. The convergence theorem for discrete distributions indicates that if $P > \xi_b$, the root will converge to 1, and if $P < \xi_b$, it will converge to 0. The only condition under which a fair game (one in which either player may win) is possible, then, is when the root fails to converge, or $P = \xi_b$. Nau pointed out that the minimax convergence theorem does not account for a widely observed phenomenon known as biasing (this can be observed in real games, not only (d, b, F)-trees)—the tendency for the player searching to perceive himself as winning if the search depth is odd, and losing if it is even [Nau 1982b]. He showed that this results from errors in the static evaluator, and derived the *last player theorem*, which states that the value returned by partial minimax on a (d, b, F)-tree approaches ξ_b if one player moved at the bottom level of the search, but $1 - \xi_b$ if the last move belonged to the other. This theorem makes an important statement about the way lookahead values should be interpreted: Values returned from alternating depths form two distinct sequences, and must be considered separately.

Minimax convergence indicates that there are instances in which lookahead is not helpful. The theorem can be viewed as an outcome of the weak law of large numbers: As the number of events in the sample space increases, the deviation of the observed outcome from the expected outcome decreases. In the case of (d, b, F)-trees, an event is the assignment of a leaf value, the observed outcome is the minimax value, and the expected outcome is $1 - \xi_b$. As $d \rightarrow \infty$, the number of leaves grows exponentially, indicating that the observed minimax value will always converge to the predicted value. Since the degree to which a node may deviate from $1 - \xi_b$ depends only on its height (distance from the leaves), lookahead is more or less worthless in the portion of the tree high above the leaves; all choices are roughly equivalent because the children of the root all converged to the same value. When that occurs, random play (on (d, b, F)-trees) is just as effective as lookahead. On the other hand, when play approaches the end of the game, there are too few leaves for the weak law of large numbers of apply, and the values may vary greatly from $1 - \xi_b$. At this

stage of the game, lookahead may help determine these values more accurately.

An even stronger statement about lookahead is made by another recent discovery, *minimax pathology*, a phenomenon whereby the decisions made by partial minimax may become less reliable as lookahead length increases. Nau (and independently, Beal [1980]) performed an error analysis on (d, b, F)-trees in which F was a uniform random distribution of binary values, and discovered that for an infinite class of game-trees, as search depth increases, so does the probability that an incorrect move will be made [Nau 1983a].

Pathology was first demonstrated on a family of board-splitting games developed by Pearl [1984] as simple games with all the properties of (d, b, F)-trees. In board splitting, a square b^d-by-b^d board is covered with 1's and 0's. The first player splits the board vertically into b sections, keeps one in play, and discards the rest. The second player splits the remaining portion horizontally, doing the same. After d rounds (a depth of $2d$ ply), only one square remains. If that square contains a 1, the horizontal splitter wins. Otherwise, the vertical splitter wins. A board-splitting game with a uniform random distribution of terminal values is called a P-game. A game with a clustering of similar values among neighboring leaves is called an N-game.[5] In all of these games, the board shrinks as play proceeds, making it possible to devise evaluation functions whose accuracy improves as the tree is descended. Nau used one such function, the percentage of 1's on a board, to show that P-games are pathological while N-games are not. This led him to conclude that the cause of pathology lay in the uniform random distribution of leaf values. When leaf values are distributed uniformly, the values of sibling nodes throughout the tree are mutually independent. Since naturally occurring sibling nodes tend to have highly related values, pathology has never been observed in real games [Nau 1982a]. Similar conclusions about trees with clustering among their terminal values were reached by Beal [1982] and Bratko and Gams [1982].

Pearl [1983] pointed out that minimax pathology is not simply a statistical aberration. Partial minimax involves propagating functions of estimates. In general, this is not the same as calculating the estimate of a function. The anomaly is not that P-games are pathological, but rather that chess is not! He performed an in-depth error analysis of minimax, and discovered that if, as is frequently claimed, the power of lookahead lies in increased visibility (more accurate static evaluations deeper in the tree), this increase must be at least 50 percent for each additional ply. Since this is almost never true in real games, he concluded that the 50 percent must be taken over all nodes found at the deeper level. In most games, certain positions qualify as *traps*, terminal positions that are located high in the tree (thus called because they trap one player into an early loss). The presence of terminal nodes in the vicinity of the search frontier drastically increases the accuracy of their ancestors, and results in the necessary improvement. If the (d, b, F)-tree is modified by making every internal node a trap with probability q exceeding a certain threshold,

$$q \geq 1 - \frac{(1 + B)^{1-(1/B)}}{B},$$

this improvement is reached, and pathology should be avoided [Pearl 1984]. In an earlier paper, I extended the idea of traps to any node whose W/L value could be determined exactly, or *f-wins* [Abramson 1986]. I demonstrated experimentally that if f-wins occur when increasing densities at deeper levels in the tree, pathology can be avoided with an overall density considerably below the predicted threshold.

Michon [1983] suggested a more realistic model than the (d, b, F)-tree, the recursive random game (RRG). From every position in an RRG, there are n legal moves ($n = 0, 1, 2, \ldots$), with probability f_n. The value f_0 indicates the probability that a node is a

[5] An N-game board is set up by randomly assigning 1's and −1's to each branch in the game-tree. If the sum of the branches leading from the root to a leaf is positive, a 1 is placed in the square corresponding to that leaf. Otherwise, the square gets a 0.

leaf, in which case it is randomly assigned either W or L. He used RRGs to analyze both pathology and quiescence. In terms of quiescence, nonquiescent positions in most games were shown to correspond to positions with relatively few options, or small branching factors. In terms of pathology, he showed that games with uniform branching factors are bound to be pathological, while games whose branching factors follow a geometric distribution are not. The various remedies to pathological behavior combine to give a solid explanation of the phenomenon: Pathology occurred because of oversimplifications in the original (d, b, F)-tree model. The removal of *any single* uniformity assumption resulted in a nonpathological tree: N-games targeted the uniform distribution, traps the uniform depth, and RRGs the uniform branching factor. When f-wins removed the uniform terminal density as well, pathology became even easier to avoid.

Pathology, minimax convergence, and the last player theorem all contribute to the understanding of partial minimax in general, and lookahead in particular. Because these results were all derived on simplified models, it is unclear exactly how applicable their direct mathematical implications are to real games. Nevertheless, they do reveal some important points that might be as true for chess and checkers as they are for board splitting:

- Lookahead is not always beneficial.
- Improved visibility, in and of itself, is not a sufficient explanation of why lookahead is helpful (when it is).
- Values returned by lookahead to different depths should only be compared if the same player moved last in both cases.
- The more uniform the tree, the more likely it is that partial minimax is not the proper control strategy to be using.

3.2 Alternatives to Minimax

Perhaps the most significant outcome of the phenomena discussed in the previous section is that for the first time, the sanctity of minimax was taken to task. These challenges to the accepted standard have motivated the design of several nonminimax control strategies. To understand these strategies, it is important to recall two of the basic assumptions underlying the optimality of minimax: perfect play by both players, and accurate information (e.g., accurately evaluated tips). Since, for the most part, neither of these conditions ever holds, alternative control strategies may lead to better performance. These inaccuracies are particularly relevant in light of game programming's original objective: the understanding of decisions. Although an assumption of presumed perfection may actually simplify a discussion of games, it is meaningless beyond fairly contrived settings. Thus, dropping these assumptions may have an impact not only on game programming per se but on the relationship between games and decisions, as well.

Pearl [1981] suggested the method of *product propagation.* This strategy assumes that the static evaluator returns the probability that a node is a forced win. If an internal node is a forced win for player 1, all of its children must be forced losses for player 2, and vice versa. In other words,

$$\Pr[\text{h is a win node}] = \Pi(1 - \Pr[\text{h}' \text{ is a win node} \mid \text{h}' \text{ is a child of h}]).$$

These alternating products propagate the probabilities back up the tree. Nau [1983b] showed that when this control strategy is used instead of minimax, pathology disappears. Tzeng proved that given a (d, b, F)-tree with independent sibling values and an evaluation function that does, in fact, return the probability of forcing a win, product propagation will outperform any equally informed algorithm. Among the assumptions inherent to product propagation, however, is the independence of sibling nodes. Since this is clearly not true in real games, there is no reason to assume that this strategy is even reasonable in most interesting domains. Nevertheless, Nau et al. [1983; Tzeng and Purdom 1983] ran some Monte Carlo experiments that demonstrated that even on N-games (which have interdependent sibling values), product propagation played well against partial

minimax. A strategy that averaged the values returned by product propagation and partial minimax (by simply adding them and dividing by 2) outperformed either strategy alone. In a similar set of experiments, Chi and Nau ran a tournament on some variants of kalah, in which product propagation (and the averaging scheme) beat partial minimax [Chi and Nau 1986, 1987]. Although these results should be enough to indicate that partial minimax is not always the strongest possible strategy, and to stress the need for further analyses, their outcomes were frequently not significant enough (in terms of statistical hypothesis testing) to reveal anything conclusive.

Ballard [1983] developed a control strategy for searching game-trees with chance nodes, **-minimax,* which assigns each chance node the average of its children's values. He and Reibman contended that the problem with partial minimax is that it, erroneously assumes perfect play on the part of the opponent. They modified *-minimax to *minimax in the presence of error.* In this system, each player assigns his opponent an expected strength. This strength is used to determine subjective probabilities indicating the likelihood that a given move will be chosen. Minimax corresponds to a strength of 1 (perfect play), and *-minimax chance nodes to a strength of 0 (random play). Imperfect play should lie somewhere between the two, and can be modeled by calculating a weighted sum of the subjective probabilities [Reibman and Ballard 1983]. Empirical studies performed on (d, b, F)-trees with correlated sibling values showed that this strategy outperformed the addition of a ply to minimax. Another alternative strategy is *minimum variance pruning* [Truscott 1979]. In this strategy, nodes are assigned probability density functions (pdf's) describing the likely location of the minimax value, and the subtree with the minimum variance is expanded first. The motivation underlying this procedure is similar to that behind conspiracy search and locked-value search: Stable values are likely to be accurate, and small variances indicate stable values. Although this strategy was proposed by Truscott in 1979, it has been neither fully developed nor tested, and is thus of unknown value.

4. DISCUSSION: RELATING THEORY TO PRACTICE

To date, none of the strategies described in the previous section have been successfully implemented in real games (i.e., competition-oriented programs). At the same time, none of the heuristics used to design type-B strategies have been successfully analyzed. There are good reasons for both of these; the theoretical strategies tend to require a full-width search (pruning techniques have yet to be devised), and have generally been developed using assumptions that are not valid in real games. Determining the accuracy of a type-B strategy, on the other hand, would probably require a model too complex to be analyzed. Because of their different orientations, there has been minimal interplay between the results of heuristic analysts and game designers. This isolationist tendency has, in turn, allowed game programming to flourish without ever developing a firm theoretical groundwork. There is overwhelming evidence that partial minimax is a reasonable control strategy, that its performance will improve with greater programming innovations, and that programs using it will play excellent chess on fast enough machines (the performance of Belle [Condon and Thompson 1982, 1983], Hitech [Berliner and Ebeling 1986], and Chiptest [Anantharaman et al. 1988] are cases in point). There is no evidence that it is in any sense correct. The discovery of pathology indicates that there are instances, albeit specialized ones, in which the traditional assumptions of minimax are false. The subsequent resolution of the phenomenon through the imposition of nonuniformity implies very definite strengths and weaknesses of the procedure; the fewer distinguishing characteristics among the nodes, the worse is the performance of minimax. Viewed in this way, pathology suggests a point that should be directly applicable to real games: Use partial minimax when there is a clear choice (i.e., when the best available minimax value is sub-

stantially better than the second best). Otherwise, if all values are clustered around some intermediate range, use another strategy, perhaps a weighted sum of modified M & N.

This split between theory and practice, although understandable, is somewhat disturbing. Experience dictates that the dismissal of unobserved theoretical predictions as irrelevant is unwise. In operations research, for example, the simplex method has long been used to efficiently solve linear programming applications. The proof of a worst-case exponential running time, and the artificial construction of examples that caused it to run poorly, motivated other models and approaches, to the point where one has been developed that not only appears to be theoretically correct, but also promises to have serious commercial potential as well [Karmakar 1984].

The incorporation of probability distributions into B*-like algorithms by Palay [1985] is encouraging. Although probabilities were used in almost all the analytic studies, prior to this work they played a more or less inconsequential role in the implementations. The original motivation for Palay's use of distributions was to more accurately assess the probable location of a node's true (minimax) value than could be done with either single numbers or ranges. However, a radically different interpretation of probability distributions is possible. Point values imply the existence of a true value that is being estimated. A range implies an unknown exact value, but one that can be bounded. In sharp contrast, the use of probability distributions may imply that there is no true value. Instead, the nodes are random variables that will be instantiated at different values with varying probabilities. The resultant model of partial game-trees is probabilistic, rather than deterministic, in nature, and corresponds to the static evaluators designed under the expected-outcome model [Abramson 1987]. In a probabilistic context, minimax is almost certainly nonoptimal. The assumptions that should go into devising a control strategy for probabilistic trees include imperfect play and an unwillingness to commit to anything beyond the next move. Implicit in the minimax procedure is the statement that *if* play reaches node X, node Y (the best of X's children) *will* be chosen. If node X is not the current node, this represents a premature commitment that may or may not make sense in a deterministic domain, but certainly does not in a probabilistic one.

5. AREAS FOR FUTURE INVESTIGATION

Every component of partial game-trees leaves many problems open. The trees represent a mathematical model that has been extensively used, but rarely studied. Throughout the course of this survey, virtually every definition in the model was shown to be vague, and every assumption was questioned at least once. The areas of difficulty that relate most directly to the design of control strategies (in no particular order) are:

- *The static evaluator*: Is an inexact goal necessary, or can search truly be guided effectively toward a number whose meaning is vague? Are there features of the tree that can serve as inexact goals, or must the information be game specific?
- *The role of knowledge*: How much game-specific information is really required to design a successful program? Should control strategies be defined for trees (the general model) or for games (the specific)? Can planning be used effectively to play games at expert levels, or only to augment more standard search techniques? Could the idea of an advice table be extended to include human interaction? Would a domain like chess lend itself to a rule-based expert system, in which strategy decisions were based on responses learned from experts?
- *The limitations of minimax*: Does the leap of type-A chess machines beyond the master level indicate that there are no limits to the strategy's power?
- *The mathematical model*: What is the correct model for two-player, zero-sum games of perfect information? Is it probabilistic or deterministic in nature? What is the optimal control strategy for making decisions in this model? How closely does this model approximate real games?

What is the relationship between partial and complete game-trees?
- *The role of lookahead*: Why is lookahead advantageous? Under what conditions will it not help? What is the correct criterion for the termination of search? How can quiescent nodes on the search frontier be recognized? Can the horizon effect be avoided completely, and if so, how?

It is hoped that, within the next few years, a new general theory of partial game trees will begin to answer some of these questions. This type of theoretical groundwork will have profound ramifications in both the analysis of heuristics and the design of games, and should be actively pursued by practitioners of both fields.

ACKNOWLEDGMENTS

I would like to thank Jonathan Gross, Richard Korf, Andrew Mayer, and Igor Roizen for supplying feedback on both the content and the style of this paper, and Jonathan Schaeffer for providing me with several interesting references from the ICCA Journal. Many of the comments supplied by the (anonymous) referees were helpful, as well. Most of this research was conducted while I was a student at Columbia University and in residence at UCLA. This research was supported in part by the National Science Foundation under grants IST-85-15302 and IST-8513989, and by a University of Southern California Faculty Research Initiation Fund.

REFERENCES

ABRAMSON, B. 1986. An explanation of and cure for minimax pathology. In *Uncertainty in Artificial Intelligence*, L. Kanal and J. Lemmer, Eds. North-Holland, Amsterdam, pp. 495–504.

ABRAMSON, B. 1987. The expected-outcome model of two-player games. Ph.D. thesis, Department of Computer Science, Columbia University, New York, 1987.

ABRAMSON, B. 1988. Learning expected-outcome evaluators in chess. In *Proceedings of the AAAI 1988 Spring Symposium Series: Computer Game Playing* (Palo Alto, Calf., Mar.). AAAI, Menlo Park, Calif., pp. 26–28.

ABRAMSON, B., AND KORF, R. 1987. A model of two-player evaluation functions. In *Proceedings of the 6th National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 90–94.

ADELSON-VELSKIY, G. M., ARLAZAROV, V. L., AND DONSKOY, M. V. 1975. Some methods of controlling the tree search in chess programs. *Artif. Intell. 6*, 361–371.

ANANTHARAMAN, T., CAMPBELL, M., AND HSIU, F. 1988. Singular extensions: Adding selectivity to brute force searching. In *Proceedings of the AAAI 1988 Spring Symposium Series: Computer Game Playing.* AAAI, Menlo Park, Calif., pp. 8–13.

BALLARD, B. W. 1983. The *-minimax procedure for trees containing chance nodes. *Artif. Intell. 21*, 327–350.

BAUDET, G. M. 1978. On the branching factor of the alpha-beta pruning algorithm. *Artif. Intell. 10* (2), 173–199.

BEAL, D. F. 1980. An analysis of minimax. In *Advances in Computer Chess 2*, M. R. B. Clarke, Ed. Edinburgh, Univ. Press, Edinburgh, Scotland, pp. 17–24.

BEAL, D. F. 1982. Benefits of minimax search. In *Advances in Computer Chess 3*, M. R. B. Clarke, Ed. Pergamon Press, Elmsford, N.Y., pp. 103–109.

BEAL, D. F. 1987. Experiments with the null move. In *Advances in Computer Chess 5*. North-Holland, Amsterdam.

BERLEKAMP, E., CONWAY, J. H., AND GUY, R. K. 1982. *Winning Ways*. Academic Press, Orlando, Fla. (In two volumes.)

BERLINER, H. J. 1973. Some necessary conditions for a master chess program. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (Stanford, Calif., Aug. 20–23). Morgan Kaufmann, Los Altos, Calif., pp. 77–85.

BERLINER, H. J. 1977a. A representation and some mechanisms for a problem-solving chess program. In *Advances in Computer Chess*, M. R. B. Clarke, Ed. Edinburgh Univ. Press, Edinburgh, Scotland, pp. 7–29.

BERLINER, H. J. 1977b. The use of domain-dependent descriptions in tree searching. In *Perspectives on Computer Science.* Academic Press, Orlando, Fla., pp. 39–62.

BERLINER, H. J. 1978. A chronology of computer chess and its literature. *Artif. Intell. 10*, 201–214.

BERLINER, H. J. 1979. The b* tree search algorithm: A best-first proof procedure. *Artif. Intell. 21*, 23–40.

BERLINER, H. J. 1980. Backgammon computer program beats world champion. *Artif. Intell. 14*, 205–220.

BERLINER, H. J. 1981. An examination of brute force intelligence. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence* (Vancouver, B.C., Canada, Aug. 24–28). Morgan Kaufmann, Los Altos, Calif., pp. 581–587.

BERLINER, H. J. 1988. New Hitech computer chess success. *AI Mag. 2*, 133.

BERLINER, H. J., AND EBELING, C. 1986. The supreme architecture: A new intelligent paradigm. *Artif. Intell. 28*, 3–8.

BHATTACHARYA, S., AND BAGCHI, A., 1986. Making best use of available memory when searching game-trees. In *Proceedings of the 5th National Conference on Artificial Intelligence* (Philadelphia, Pa., Aug. 11–15), pp. 163–167.

BIRMINGHAM, J. A., AND KENT, P. 1977. Tree searching and tree pruning techniques. In *Advances in Computer Chess*, M. R. B. Clarke, Ed. Edinburgh Univ. Press, Edinburgh, Scotland, pp. 89–96.

BOTVINNIK, M. M. 1970. *Computers, Chess, and Long Range Planning* (A. Brown, transl.). Springer-Verlag, New York.

BOTVINNIK, M. M. 1982. Decision making and computers. In *Advances in Computer Chess 3*, M. R. B. Clarke, Ed. Pergamon Press, Elmsford, N.Y., pp. 169–180.

BOTVINNIK, M. M. 1984. *Computers in Chess: Solving Inexact Search Problems* (A. Brown, transl.). Springer-Verlag, New York.

BRAMER, M. A. 1980. An optimal algorithm for king and pawn against king using pattern knowledge. In *Advances in Computer Chess 2*, M. R. B. Clarke, Ed. Edinburgh Univ. Press, Edinburgh, Scotland.

BRAMER, M. A. Ed. 1983. *Computer Game Playing: Theory and Practice.* Ellis Horwood Ltd. 1, Chichester, W. Sussex, England

BRATKO, I., AND GAMS, M. 1982. Error analysis of the minimax principle. In *Advances in Computer Chess 3*, M. R. B. Clarke, Pergamon Press, Elmsford, N.Y., pp. 1–16.

BRATKO, I., AND MICHIE, D. 1980. An advice program for a complex chess programming task. *Comput. J. 23*, 353–359.

BRUDNO, A. L. 1963. Bounds and valuations for abridging the search of estimates. In *Problems of Cybernetics*. Pergamon Press, Elmsford, N.Y., pp. 225–241.

CAMPBELL, M. 1981. Algorithms for the parallel search of game-trees. M.S. thesis, Computer Science Dept., University of Alberta, Alberta, Canada, Aug.

CAMPBELL, M. S., AND MARSLAND, T. A. 1983. A comparison of minimax tree search algorithms. *Artif. Intell. 20*, 347–367.

CHI, P.-C., AND NAU, D. S. 1986. Predicting the performance of minimax and product in game-tree searching. In *Proceedings of the 2nd Workshop of Uncertainty in Artificial Intelligence* (Philadelphia, Pa., Aug.), pp. 49–55.

CHI, P.-C., AND NAU, D. S. 1987. Comparing minimax and product in a variety of games. In *Proceedings of the 6th National Conference on Artificial Intelligence* (Seattle, Wash., July 13–17), pp. 100–104.

CONDON, J. H., AND THOMPSON, K. 1982. Belle chess hardware. In *Advances in Computer Chess 3*, M. R. B. Clarke, Ed. Pergamon Press, Elmsford, N.Y., pp. 45–54.

CONDON, J. H., AND THOMPSON, K. 1983. Belle. In *Chess Skill in Man and Machine*, P. W. Frey, Ed. Springer-Verlag, New York, pp. 201–210.

EDWARDS, D., AND HART, T. 1963. The alpha–beta heuristic. Tech. Rep. 30, MIT AI Memo, Computer Science Dept., Massachusetts Institute of Technology, Cambridge, Mass., Oct. Originally published as the Tree Prune Algorithm, Dec. 1961.

FULLER, S., GASCHNIG, J., AND GILLOGLY, J. 1973. Analysis of the alpha–beta pruning algorithm. Tech. Rep., Computer Science Dept., Carnegie-Mellon University, Pittsburgh, Pa.

GOETSCH, G., AND CAMPBELL, M. 1988. Experiments with the null move heuristic in chess. In *Proceedings of the AAAI 1988 Spring Symposium Series: Computer Game Playing.* AAAI, Menlo Park, Calif., pp. 14–18.

GREENBLATT, R. D., EASTLAKE, III, D. E., AND CROCKER, S. D. 1967. The Greenblatt chess program. In *Proceedings of the 1967 Fall Joint Computer Conference* (Washington, D.C.), pp. 801–810.

GRIFFITH, A. K. 1974. A comparison and evaluation of three machine learning procedures as applied to the game of checkers. *Artif. Intell. 5*, 137–148.

HARRIS, L. R. 1974. The heuristic search under conditions of error. *Artif. Intell. 5*, 217–234.

HARTMANN, D. 1987a. How to extract relevant knowledge from grandmaster games, Part 1. *Int. Comput. Chess Assoc. J. 10* (1), 14–36.

HARTMANN, D. 1987b. How to extract relevant knowledge from grandmaster games, Part 2. *Int. Comput. Chess Assoc. J. 10* (2), 78–90.

IBARAKI, T. 1986. Generalization of alpha–beta and SSS* search procedures. *Artif. Intell. 29* (1), 73–118.

KAINDL, H. 1983a. Quiescence search in computer chess. In *Computer Game Playing: Theory and Practice*, M. A. Bramer, Ed. Ellis Horwood Ltd., Chichester, W. Sussex, England, pp. 39–52.

KAINDL, H. 1983b. Searching to variable depth in computer chess. In *Proceedings of the 8th International Conference on Artificial Intelligence* (Karlsruhe, West Germany, Aug. 8–12). Morgan Kaufmann, Los Altos, Calif., pp. 760–762.

KARMAKAR, N. 1984. A new polynomial time algorithm for linear programming. In *Proceedings of the 16th Annual ACM Symposium on the Theory of Computing* (Washington, D.C., Apr. 30–May 2). ACM, New York, pp. 302–311.

KNUTH, D. E., AND MOORE, R. W. 1975. An analysis of alpha–beta pruning. *Artif. Intell. 6*, 293–326.

KOPEC, D. 1985. Chess computers. *Abacus, 2*, 10–28.

KORF, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell. 27*, 97–109.

LEVY, D. 1976. *Chess and Computers.* Computer Science Press, Potomac, Md.

MAGGS, P. B. 1979. Programming strategies in the game of Reversi. *BYTE, 4*, 66–79.

MARSLAND, T. A. 1983. Relative efficiency of alpha–beta implementations. In *Proceedings of the 8th International Conference on Artificial Intelligence* (Karlsruhe, West Germany, Aug. 8–12). Morgan Kaufmann, Los Altos, Calif. pp. 763–766.

MARSLAND, T. A. 1986. A review of game-tree pruning. *Int. Comput. Chess Assoc. J. 9* (1), 3–19.

MARSLAND, T. A., AND CAMPBELL, M. 1982. Parallel search of strongly ordered game trees. *ACM Comput. Surv. 14* (4, Dec.), 533–551.

MARSLAND, T. A., AND POPOWICH, F. 1985. Parallel game-tree search. *IEEE Trans. Pattern Anal. Mach. Intell. 7* (4, July), 442–452.

MARSLAND, T. A., REINEFELD, A., AND SCHAEFFER, J. 1987. Low overhead alternatives to SSS*. *Artif. Intell. 31* (2), 185–199.

MCALLESTER, D. A. 1985. A new procedure for growing min-max trees. Tech. rep., MIT Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass., July.

MICHIE, D. 1977. King and rook against king: Historical background and a problem on the infinite board. In *Advances in Computer Chess*, M. R. B. Clarke, Ed. Edinburgh Univ. Press, Edinburgh, Scotland.

MICHON, G. P. 1983. Recursive random games: A probabilistic model for perfect information games. Ph.D. thesis, Computer Science Dept., University of California at Los Angeles, 1983.

NAU, D. S. 1982a. An investigation of the causes of pathology in games. *Artif. Intell. 19*, 257–278.

NAU, D. S. 1982b. The last player theorem. *Artif. Intell. 18*, 53–65.

NAU, D. S. 1983a. Decision quality as a function of search depth on game trees. *J. ACM 30*, 4 (oct.) 687–708.

NAU, D. S. 1983b. Pathology on game trees revisited, and an alternative to minimax. *Artif. Intell. 21*, 221–244.

NAU, D. S., PURDOM, P., AND TZENG, C.-H. 1983. Experiments on alternatives to minimax. Tech. rep., Computer Science Dept., Univ. of Maryland, College Park, Md., Oct.

NEWBORN, M. M. 1975. *Computer Chess*. Academic Press, Orlando, Fla.

NEWBORN, M. M. 1977. The efficiency of the alpha-beta search on trees with branch-dependent terminal node scores. *Artif. Intell. 8* (2), 137–153.

NEWBORN, M. M. 1985. A parallel search chess program. In *ACM '85—The Range of Computing: Mid-80's Perspective* (Denver, Colo., Oct. 14–16). ACM, New York, pp. 272–277.

NEWELL, A., SHAW, J. C., AND SIMON, H. A. 1963. Chess playing programs and the problem of complexity. In *Computers and Thought*, E. Feigenbaum and J. Feldman, Eds. McGraw-Hill, New York, pp. 39–70.

NILSSON, N. J. 1980. *Principles of Artificial Intelligence*. Tioga Publ., Palo Alto, Calif.

PALAY, A. J. 1982. The b* tree search algorithm—new results. *Artif. Intell. 19*, 145–163.

PALAY, A. J. 1985. *Searching With Probabilities*. Pitman, London.

PEARL, J. 1980. Asymptotic properties of minimax trees and game-searching procedures. *Artif. Intell. 14*, 113–138.

PEARL, J. 1981. Heuristic search theory: A survey of recent results. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, Los Altos, Calif., pp. 24–28.

PEARL, J. 1982. The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. *Commun. ACM, 25*, 8 (Aug.) 559–564.

PEARL, J. 1983. On the nature of pathology in game searching. *Artif. Intell. 20*, 427–453.

PEARL, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison Wesley, Reading, Mass.

PITRAT, J. 1977. A chess combination program which uses plans. *Artif. Intell. 21*, 275–321.

PITRAT, J. 1980. The behaviour of a chess combination program using plans. In *Advances in Computer Chess 2*, M. R. B. Clarke, Ed. Edinburgh Univ. Press, Edinburgh, Scotland.

REIBMAN, A. L., AND BALLARD, B. W. 1983. Nonminimax search strategies for use against fallible opponents. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence* (Karlsruhe, West Germany, Aug. 8–12). Morgan Kaufmann, Los Altos, Calif., pp. 338–342.

REINEFELD, A. 1983. An improvement of the scout tree-search algorithm. *Int. Comput. Chess Assoc. J. 6* (4), 4–14.

RICH, E. 1983. *Artificial Intelligence*. McGraw-Hill, New York.

RIVEST, R. 1987. Game tree searching by min/max approximation. *Artif. Intell. 34* (1), 77–96.

ROIZEN, I., AND PEARL, J. 1983. A minimax algorithm better than alpha-beta? Yes and no. *Artif. Intell. 21*, 199–220.

ROSENBLOOM, P. S. 1982. A world-championship-level Othello program. *Artif. Intell. 19*, 279–320.

SAMUEL, A. L. 1959. Some studies in machine learning using the game of checkers. *IBM J. 3*, 211–229.

SAMUEL, A. L. 1967. Some studies in machine learning using the game of Checkers II—recent progress. *IBM J. 11*, 601–617.

SCHAEFFER, J. 1986. Improved parallel alpha-beta search. In *ACM/IEEE Fall Joint Computer Conference Proceedings* (Dallas, Tex., Nov. 2–6), pp. 519–527.

SCHAEFFER, J. 1987. Speculative computing. *Int. Comput. Chess Assoc. J. 10* (3), 118–124.

SHANNON, C. E. 1950. Programming a computer for playing chess. *Philos. Mag. 41*, 256–275.

SLAGLE, J. R., AND DIXON, J. K. 1969. Experiments with some programs that search trees. *J. ACM*, 2 (Apr.) *16*, 189–207.

SLAGLE, J. R., AND DIXON, J. K. 1970. Experiments with the M & N tree-searching procedure. *Commun. ACM 13*, 3 (Mar.) 147–154.

SLATE, D. J., AND ATKIN, L. R. 1977. Chess 4.5—the Northwestern University chess program. In *Chess Skill in Man and Machine*, P. W. Frey, Ed. Springer-Verlag, New York, pp. 82–118.

STOCKMAN, G. C. 1979. A minimax algorithm better than alpha-beta? *Artif. Intell. 12*, 179–196.

THOMPSON, K. 1982. Computer chess strength. In *Advances in Computer Chess 3*, M. R. B. Clarke, Ed. Pergamon Press, Elmsford, N.Y., pp. 55–56.

TRUSCOTT, T. 1979. Minimum variance tree searching. In *Proceedings of the 1st International Symposium on Policy Analysis and Information Systems* (Durham, N.C.) pp. 203–209.

TZENG, C.-H., AND PURDOM, P. W. 1983. A theory of game trees. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence* (Karlsrahe, West Germany, Aug. 8–12) Morgan Kaufmann, Los Altos, Calif., pp. 416–419.

VON NEUMANN, J., AND MORGENSTERN, O. 1944. *Theory of Games and Economic Behavior*. Princeton Univ. Press, Princeton, N.J.

WILKINS, D. 1977. Using chess knowledge to reduce search. In *Chess Skill in Man and Machine*, P. W. Frey, Ed. Springer-Verlag, New York.

WILKINS, D. 1980. Using patterns and plans in chess. *Artif. Intell. 14*, 165–203.

WILKINS, D. 1982. Using knowledge to control tree searching. *Artif. Intell. 18*, 1–51.

WINSTON, P. H. 1977. *Artificial Intelligence*. Addison-Wesley, Reading, Mass.

Received May 1986; final revision accepted December 1988.