6 Trees and Adversarial Search

In this chapter, you learn about how programs can play board games, such as checkers and chess, in which one choice leads to another producing a tree of choices. In contrast with the choices in Chapter 4 and Chapter 5, the choices here are the interdigitated choices of two adversaries.

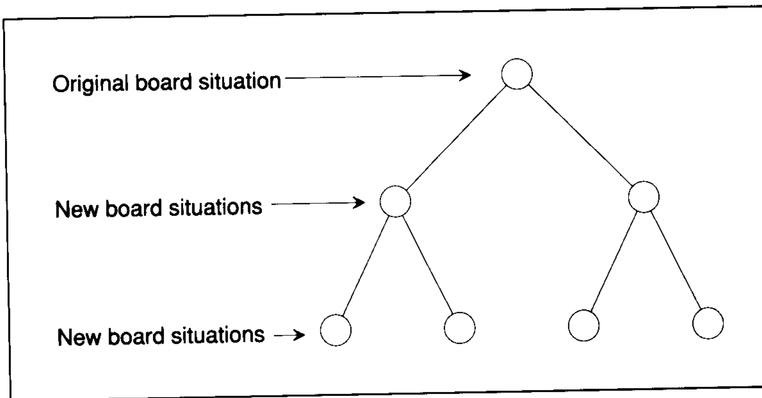
In particular, you learn about *minimax search*, the basic method for deciding what to do, and about *alpha–beta pruning*, an idea that greatly reduces search by *stopping work on guaranteed losers*. You also learn about *progressive deepening* and *heuristic continuation*, both of which help programs to allocate search effort more effectively.

Once you have finished this chapter, you will know that you should not waste time exploring alternatives that are sure to be bad, and that you should spend a lot of time buttressing conclusions that are based on only flimsy evidence.

ALGORITHMIC METHODS

In this section, you learn how game situations can be represented in trees, and you learn how those trees can be searched so as to make the most promising move.

Figure 6.1 Games raise a new issue: competition. The nodes in a game tree represent board configurations, and the branches indicate how moves can connect them.



Nodes Represent Board Positions

As shown in figure 6.1, the natural way to represent what can happen in a game is to use a **game tree**, which is a special kind of semantic tree in which the nodes denote board configurations, and the branches indicate how one board configuration can be transformed into another by a single move. Of course, there is special twist in that the decisions are made by two adversaries who take turns making decisions.

The **ply** of a game tree, p, is the number of levels of the tree, including the root level. If the depth of a tree is d, then $p = d + 1$. In the chess literature, a **move** consists of one player's choice and another player's reaction. Here, however, we will be informal, referring to each choice as a *move*.

Exhaustive Search Is Impossible

Using something like the British Museum procedure, introduced in Chapter 5, to search game trees is definitely out. For chess, for example, if we take the effective branching factor to be 16 and the effective depth to be 100, then the number of branches in an exhaustive survey of chess possibilities would be on the order of 10^{120}—a ridiculously large number. In fact, if all the atoms in the universe had been computing chess moves at picosecond speeds since the big bang (if any), the analysis would be just getting started.

At the other end of the spectrum, if only there were some infallible way to rank the members of a set of board situations, it would be a simple matter to play by selecting the move that leads to the best situation that can be reached by one move. No search would be necessary. Unfortunately, no such situation-ranking procedure exists. When one board situation is enormously superior to another, a simple measure, such as a piece count, is likely to reflect that superiority, but always relying on such a measure to rank the available moves from a given situation produces poor results. Some other strategy is needed.

One other strategy is to use a situation analyzer only after several rounds of move and countermove. This approach cannot be pursued too far because the number of alternatives soon becomes unthinkable, but if search terminates at some reasonable depth, perhaps the leaf-node situations can be compared, yielding a basis for move selection. Of course, the underlying presumption of this approach is that the merit of a move clarifies as the move is pursued and that the lookahead procedure can extend far enough that even rough board-evaluation procedures may be satisfactory.

The Minimax Procedure Is a Lookahead Procedure

Suppose we have a situation analyzer that converts all judgments about board situations into a single, overall quality number. Further suppose that positive numbers, by convention, indicate favor to one player, and negative numbers indicate favor to the other. The degree of favor increases with the absolute value of the number.

The process of computing a number that reflects board quality is called **static evaluation**. The procedure that does the computation is called a **static evaluator**, and the number it computes is called the **static evaluation score**.

The player hoping for positive numbers is called the **maximizing player** or the **maximizer**. The other player is called the **minimizing player** or **minimizer**.

A **game tree** is a representation

That is a semantic tree

In which

▷ Nodes denote board configurations

▷ Branches denote moves

With writers that

▷ Establish that a node is for the maximizer or for the minimizer

▷ Connect a board configuration with a board-configuration description

With readers that

▷ Determine whether the node is for the maximizer or minimizer

▷ Produce a board configuration's description

The maximizer looks for a move that leads to a large positive number, and assumes that the minimizer will try to force the play toward situations with strongly negative static evaluations.

Thus, in the stylized, miniature game tree shown in figure 6.2, the maximizer might hope to get to the situation yielding a static score of 8. But the maximizer knows that the minimizer can choose a move deflecting the play toward the situation with a score of 1. In general, the decisions of the maximizer must take cognizance of the choices available to the minimizer at the next level down. Similarly, the decisions of the minimizer must take cognizance of the choices available to the maximizer at the next level down.

Eventually, however, the limit of exploration is reached and the static evaluator provides a direct basis for selecting among alternatives. In the example, the static evaluations at the bottom determine that the minimizer can choose between effective scores of 2 and 1 at the level just up from the static evaluations. Knowing these effective scores, the maximizer can make the best choice at the next level up. Clearly, the maximizer chooses to move toward the node from which the minimizer can do no better than to hold the effective score to 2. Again, the scores at one level determine the action and the effective score at the next level up.

The procedure by which the scoring information passes up the game tree is called the MINIMAX **procedure**, because the score at each node is either the minimum or the maximum of the scores at the nodes immediately below:

To perform a minimax search using MINIMAX,

- ▷ If the limit of search has been reached, compute the static value of the current position relative to the appropriate player. Report the result.
- ▷ Otherwise, if the level is a minimizing level, use MINIMAX on the children of the current position. Report the minimum of the results.
- ▷ Otherwise, the level is a maximizing level. Use MINIMAX on the children of the current position. Report the maximum of the results.

Note that the whole idea of minimaxing rests on the translation of board quality into a single, summarizing number, the static value. Unfortunately, a number is a poor summary.

Also note that minimaxing can be expensive, because either the generation of paths or static evaluation can require a lot of computation. Which costs more depends on how the move generator and static evaluator have been implemented.

The Alpha-Beta Procedure Prunes Game Trees

At first, it might seem that the static evaluator must be used on each leaf node at the bottom of the search tree. But, fortunately, this is not so.

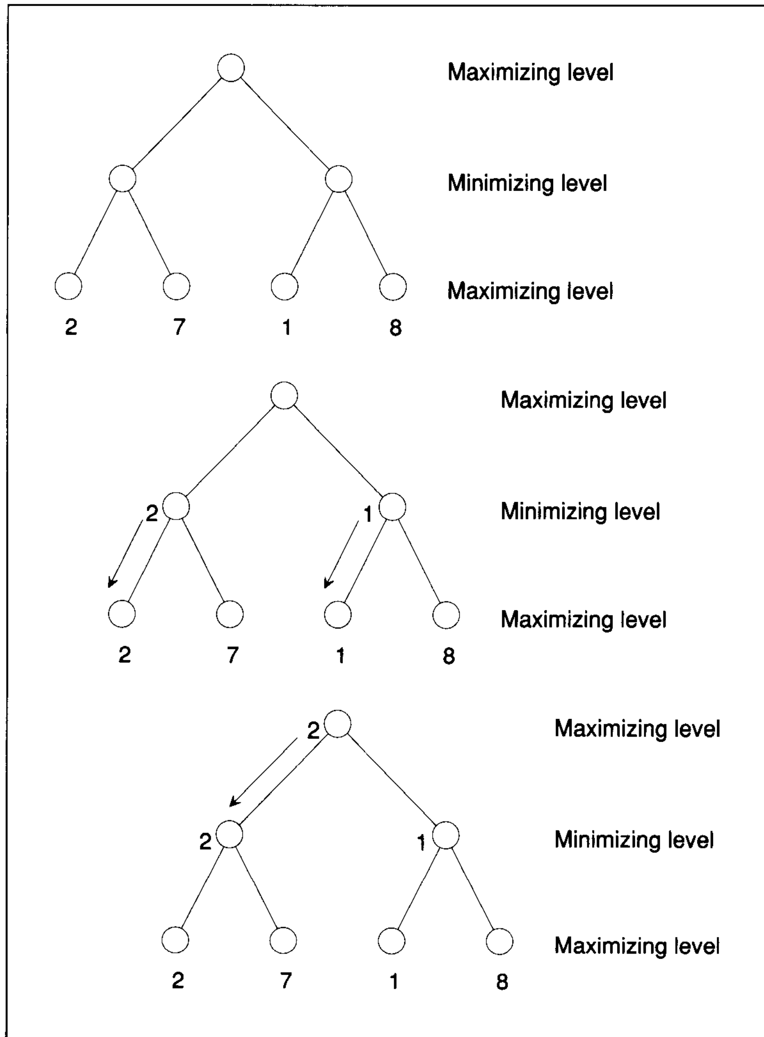


Figure 6.2 Minimaxing is a method for determining moves. Minimaxing employs a static evaluator to calculate advantage-specifying numbers for the game situations at the bottom of a partially developed game tree. One player works toward the higher numbers, seeking the advantage, while the opponent goes for the lower numbers.

There is a procedure that reduces both the number of tree branches that must be generated and the number of static evaluations that must be done, thus cutting down on the work to be done overall. It is somewhat like the branch-and-bound idea in that some paths are demonstrated to be bad even though not followed to the lookahead limit.

Consider the situation shown at the top of figure 6.3, in which the static evaluator has already been used on the first two leaf-node situations. Performing the MINIMAX procedure on the scores of 2 and 7 determines that the minimizing player is guaranteed a score of 2 if the maximizer takes the left branch at the top node. This move in turn ensures that the maximizer is guaranteed a score at least as good as 2 at the top. This guarantee

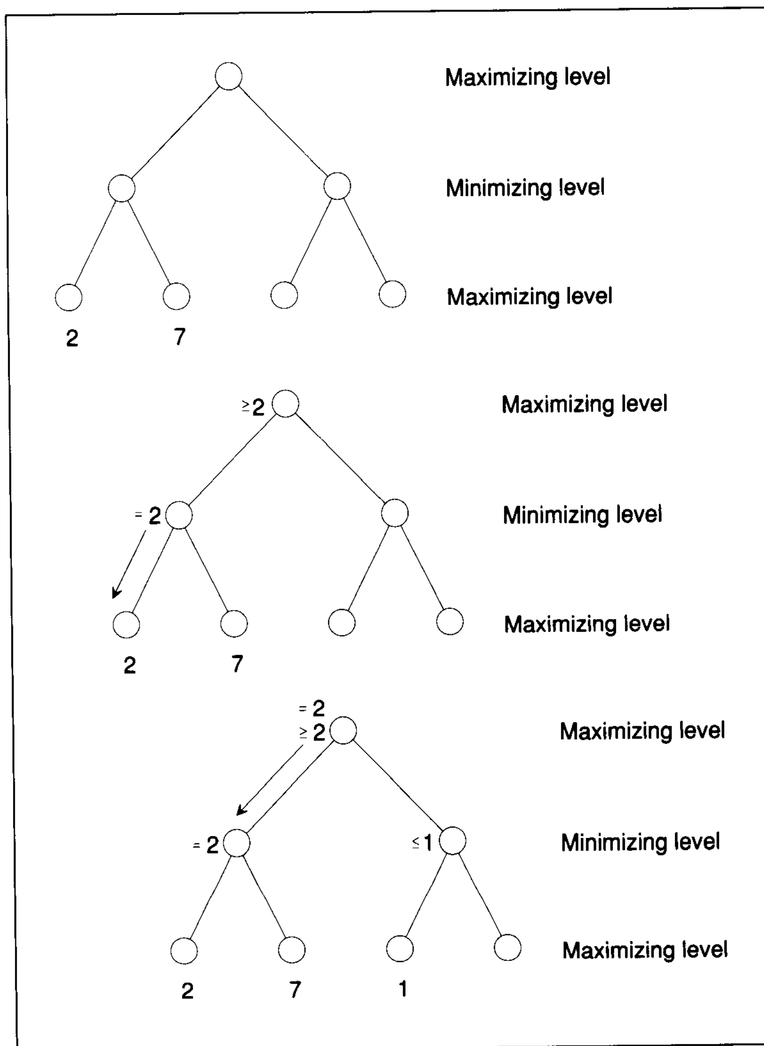


Figure 6.3 The ALPHA-BETA procedure at work. There is no need to explore the right side of the tree fully, because there is no way the result could alter the move decision. Once movement to the right is shown to be worse than movement to the left, there is no need to see how much worse.

is clear, even before any other static evaluations are made, because the maximizer can certainly elect the left branch whenever the right branch turns out to result in a lower score. This situation is indicated at the top node in the middle of figure 6.3.

Now suppose the static value of the next leaf node is 1. Evidently, the minimizer is guaranteed a score at least as low as 1 by the same reasoning that showed that the maximizer is guaranteed a score at least as high as 2 at the top. At the maximizer levels, a *good* score is a larger value; at the minimizer levels, a *good* score is a smaller value.

Look closely at the tree. Does it make sense to go on to the board situation at the final node? Can the value produced there by the static

evaluator possibly matter? Strangely, the answer is No. For surely if the maximizer knows that it is guaranteed a score of 2 along the left branch, it needs to know no more about the right branch other than that it can get a score of no higher than 1 there. The last node evaluated could be +100, or −100, or any number whatever, without affecting the result. The maximizer's score is 2, as shown in the bottom of figure 6.3.

On reflection, it is clear that, if an opponent has one response establishing that a potential move is bad, there is no need to check any other responses to the potential move. More generally, you have an instance of the following powerful idea:

The **alpha–beta principle**:

▷ If you have an idea that is surely bad, do not take time to see how truly awful it is.

This idea is called the alpha–beta principle because, as you see later, it is embodied in the ALPHA–BETA *procedure*, which uses two parameters, traditionally called alpha and beta, to keep track of expectations.

In the special context of games, the alpha–beta principle dictates that, whenever you discover a fact about a given node, you should check what you know about ancestor nodes. It may be that no further work is sensible below the parent node. Also, it may be that the best that you can hope for at the parent node can be revised or determined exactly.

With the alpha–beta principle translated into instructions for dealing with score changes, you can work through a larger example. Unfortunately, it is a bit difficult to see how static evaluations intermix with conclusions about node values on paper. We must make do with boxed event numbers placed beside each conclusion showing the order in which the conclusions are determined. These numbers are shown in the example of figure 6.4, in which we look at another stylized tree with a depth of 3 and a uniform branching factor of 3:

1–2. Moving down the left branch at every decision point, the search penetrates to the bottom where a static value of 8 is unearthed. This 8 clearly means that the maximizer is guaranteed a score at least as high as 8 with the three choices available. A note to this effect is placed by step 2.

3–5. To be sure no score higher than 8 can be found, the maximizer examines the two other moves available to it. Because 7 and 3 both indicate inferior moves, the maximizer concludes that the highest score achievable is exactly 8 and that the correct move is the first one examined.

6. Nailing down the maximizer's score at the lowest node enables you to draw a conclusion about what the minimizer can hope for at

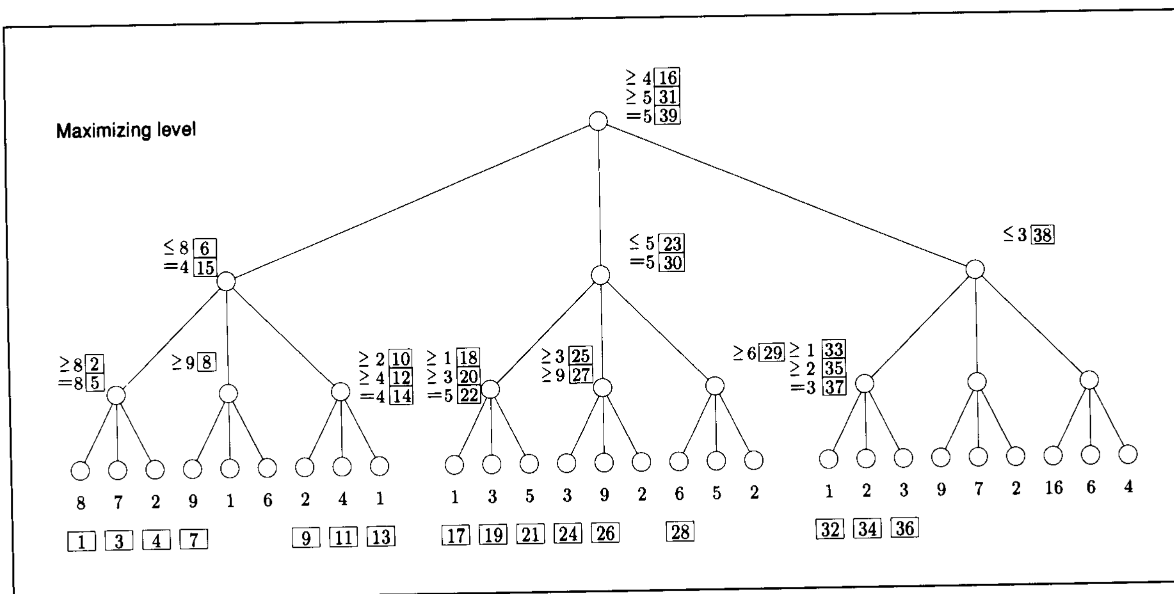


Figure 6.4 A game tree of depth 3 and branching factor 3. The boxed numbers show the order in which conclusions are drawn. Note that only 16 static evaluations are made, rather than the 27 required without alpha–beta pruning. Evidently the best play for the maximizer is down the middle branch.

the next level up. Because one move is now known to lead to a situation that gives the maximizer a score of 8, you know that the minimizer at the next level up can achieve a score of 8 or lower.

7–8. To see whether the minimizer can do better at the second level, you must examine his two remaining moves. The first leads to a situation from which the maximizer can score at least a 9. Here cutoff occurs. By taking the left branch, the minimizer forces a score of 8; but by taking the middle branch, the minimizer allows a score that is certainly no lower than 9 and will be higher if the other maximizer choices are higher. Hence, the middle branch is bad for the minimizer, there is no need to go on to find out how bad it is, and there is consequently no need for two static evaluations. There is no change in the minimizer's worst-case expectation; it is still 8.

9–14. The minimizer must still investigate its last option, the one to the right. You need to see what the maximizer can do there. The next series of steps bounces between static evaluations and conclusions about the maximizer's situation immediately above them. The conclusion is that the maximizer's score is 4.

15. Discovering that the right branch leads to a forced score of 4, the minimizer would take the right branch, because 4 is less than 8, the previous low score.

16. Now a bound can be placed at the top level. The maximizer, surveying the situation there, sees that its left branch leads to a score of 4, so it now knows it will score at least that high, and

perhaps better. To see if it can do better, it must look at its middle and right branches.

17–22. Deciding how the minimizer will react at the end of the middle branch requires knowing what happens along the left branch descending from there. Here, the maximizer is in action, discovering that the best play is to a position with a score of 5.

23. Until something definite was known about what the maximizer could do, no bounds could be placed on the minimizer's potential. Knowing that the maximizer scores 5 along the left branch, however, is knowing something definite. The conclusion is that the minimizer can obtain a score at least as low as 5.

24–27. In working out what the maximizer can do below the minimizer's middle branch, you discover partway through the analysis that the maximizer can reach a score of 9. But 9 is a poor choice relative to the known option of the minimizer that ensures a 5. Cutoff occurs again. There is no point in investigating the other maximizer option, so you avoid one static evaluation.

28–29. Looking at the minimizer's right branch quickly shows that it, too, gives the maximizer a chance to force the play to a higher score than the minimizer can achieve along the left branch. Cutoff saves two static evaluations here.

30. Because there are no more branches to investigate, the minimizer's score of 5 is no longer merely a bound; 5 is the actual value achievable.

31. The maximizer at the top, seeing a choice leading to a higher score through the middle branch, chooses that branch tentatively and knows now that it can score at least as high as 5.

32–37. Now the maximizer's right-branch choice at the top must be explored. Diving into the tree, bouncing about a bit, leads to the conclusion that the minimizer sees a left-branch choice ensuring a score of 3.

38. The minimizer can conclude that the left-branch score is a bound on how low a score it can obtain.

39. Knowing the minimizer can force play to a situation with a score of 3, the maximizer at the top level concludes that there is no point in exploring the right branch farther. After all, a score of 5 follows a middle-branch move. Note that this saves six static evaluations, as well as two move generations.

It is not unusual to get lost in this demonstration. Even seasoned game specialists still find magic in the ALPHA–BETA procedure. Each individual conclusion seems right, but somehow the global result is strange and hard to believe.

Note that, in the example, you never had to look more than one level up to decide whether or not to stop exploration. In deeper trees, with four or more levels, so-called **deep cutoffs** can occur, forcing a longer look.

One way to keep track of all the bookkeeping is to use a procedure with parameters, alpha and beta, that record all the necessary observations. The ALPHA–BETA procedure is started on the root node with an alpha value of $-\infty$ and a beta value of $+\infty$; ALPHA–BETA then calls itself recursively with a narrowing range between the alpha and beta values:

To perform minimax search with the ALPHA–BETA procedure,

- ▷ If the level is the top level, let alpha be $-\infty$ and let beta be ∞.
- ▷ If the limit of search has been reached, compute the static value of the current position relative to the appropriate player. Report the result.
- ▷ If the level is a minimizing level,
  - ▷ Until all children are examined with ALPHA–BETA or until alpha is equal to or greater than beta,
    - ▷ Use the ALPHA–BETA procedure, with the current alpha and beta values, on a child; note the value reported.
    - ▷ Compare the value reported with the beta value; if the reported value is smaller, reset beta to the new value.
  - ▷ Report beta.
- ▷ Otherwise, the level is a maximizing level:
  - ▷ Until all children are examined with ALPHA–BETA or alpha is equal to or greater than beta,
    - ▷ Use the ALPHA–BETA procedure, with the current alpha and beta value, on a child; note the value reported.
    - ▷ Compare the value reported with the alpha value; if the reported value is larger, reset alpha to the new value.
  - ▷ Report alpha.

Alpha-Beta May Not Prune Many Branches from the Tree

One way deepen your understanding of the ALPHA–BETA procedure is to ask about its the best-case and worst-case performance.

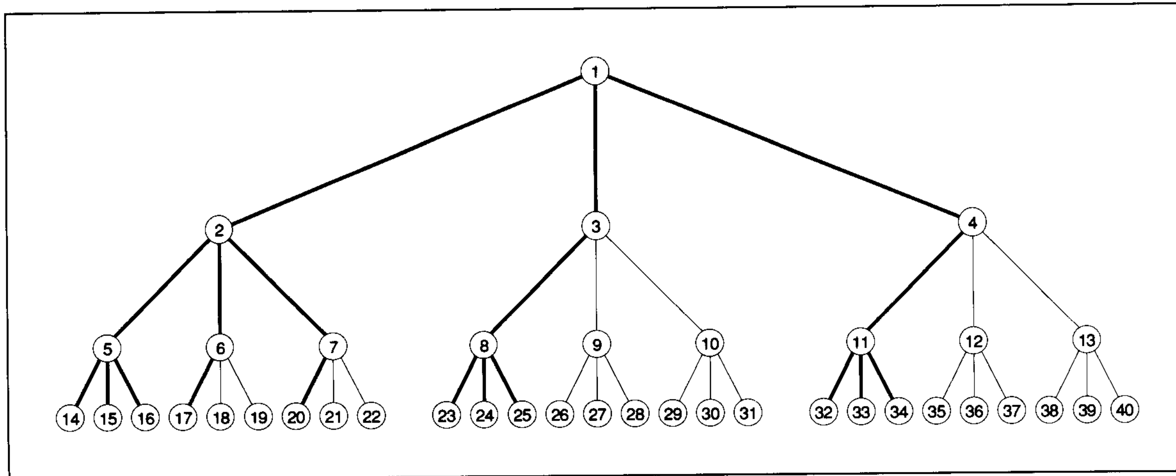


Figure 6.5 An ideal situation from the perspective of the ALPHA–BETA procedure. The ALPHA–BETA procedure cuts the exponent of exponential explosion in half, because not all the adversary's options need to be considered in verifying the left-branch choices. In a tree with depth 3 and branching factor 3, the ALPHA–BETA procedure can reduce the number of required static evaluations from 27 to 11.

In the worst case, for some trees, the branches can be ordered such that the ALPHA–BETA procedure does nothing. For other trees, however, there is no way to order the branches to avoid all alpha–beta cutoff.

For trees ordered by a cooperative oracle, the ALPHA–BETA procedure does a great deal. To see why, suppose a tree is ordered with each player's best move being the leftmost alternative at every node. Then, clearly, the best move of the player at the top is to the left. But how many static evaluations are needed for the topmost player to be sure that this move is optimal? To approach the question, consider the tree of depth 3 and branching factor 3 shown in figure 6.5.

Presuming that the best moves for both players are always to the left, then the value of the leftmost move for the maximizing player at the top is the static value of the board situation at the bottom left. This static value provides the maximizer with a concrete measurement against which the quality of the alternatives can be compared. The maximizer does not need to consider all the minimizer's replies to those alternatives, however.

To verify the correct move at a given node in an ordered tree, a player needs to consider relatively few of the leaf nodes descendant from the immediate alternatives to the move to be verified. All leaf nodes found below nonoptimal moves by the player's opponent can be ignored.

Why is it necessary to deal with all the options of the moving player while ignoring all but one of the moving-player's opponent's moves? This point is a sticky one. To understand the explanation, you need to pay close attention to the alpha–beta principle: if an opponent has some response that makes a move bad no matter what the moving player does subsequently, then the move is bad.

The key to understanding lies in the words *some*, and *no matter what.* The *some* suggests that the moving player should analyze its opponent's moves wherever the opponent has a choice, hoping that the selected move

certifies the conclusion. But to be sure that the conclusion holds no matter what the moving player might do, the moving player must check out all its choices.

Thus, the hope in the example in figure 6.5 is that only the leftmost branch from node 3 to node 8 will need exploration. All the maximizer's counterresponses to that move must be checked, so static evaluations need to be made at nodes 23, 24, and 25.

These evaluations establish that the maximizer's score at node 8, which in turn sets a bound on what the minimizer can do at node 3, which, by comparison with the minimizer's score at node 2, should show that no further work below node 3 makes any sense. Similar reasoning applies to node 4, which leads to static evaluations at node 32, node 33, and node 34.

Now, however, the question is, How can the maximizer be sure that the score transferred up the left edge is valid? Surely, it must verify that an intelligent minimizer at node 2 would select the leftmost branch. It can do this verification by assuming the number coming up the left edge from node 5 is correct and then rejecting the alternatives as efficiently as possible. But, by the same arguments used at node 1, it is clear that not all the minimizer's opponent's options need to be examined. Again, branching occurs only at every second level, working out from the choice to be verified along the left edge. Static evaluations must be done at nodes 17 and 20.

Finally, there is the question of the minimizer's assumption about the number coming up from node 5. Answering this question requires exploring all of the maximizer's alternatives, resulting in static evaluations at node 15 and node 16 to ensure that the static evaluation done at node 14 yields the correct number to transfer up to node 5.

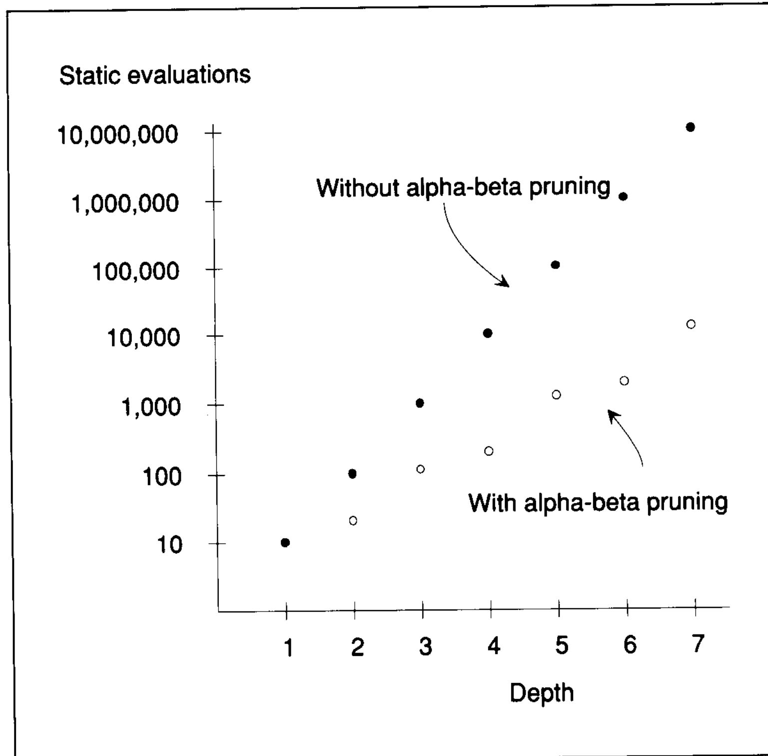
You need to make only 11 of the 27 possible static evaluations to discover the best move when, by luck, the alternatives in the tree have been nicely ordered. In deeper trees with more branching, the saving is more dramatic. In fact, it can be demonstrated that the number of static evaluations, s, needed to discover the best move in an optimally arranged tree is given by the following formula, where b is the branching factor and d is the depth of the tree:

$$s = \begin{cases} 2b^{d/2} - 1 & \text{for } d \text{ even;} \\ b^{(d+1)/2} + b^{(d-1)/2} - 1 & \text{for } d \text{ odd.} \end{cases}$$

A straightforward proof by induction verifies the formula. You need only to generalize the line of argument used in the previous example, focusing on the idea that verification of a choice requires a full investigation of only every second level. Note that the formula is certainly correct for $d = 1$, because it then simplifies to b. For $d = 3$ and $b = 3$, the formula yields 11, which nicely resonates with the conclusion reached for the example.

But be warned: The formula is valid for only the special case in which a tree is perfectly arranged. As such, the formula is an approximation to what can actually be expected; if there were a way of arranging the tree

Figure 6.6 Explosive growth. The ALPHA–BETA procedure reduces the rate of explosive growth, but does not prevent it. The branching factor is assumed to be 10.



with the best moves on the left, clearly there would be no point in using alpha–beta pruning. Noting this fact is not the same as saying that the exercise has been fruitless, however. It establishes the lower bound on the number of static evaluations that would be needed in a real game. It is a lower bound that may or may not be close to the real result, depending on how well the moves are, in fact, arranged. The real result must lie somewhere between the worst case, for which static values must be computed for all b^d leaf nodes, and the best case, for which static values must be computed for approximately $2b^{d/2}$ leaf nodes. In practice, the number of static evaluations seems nearer to the best case than the worst case, nature proving unusually beneficent.

Still, the amount of work required becomes impossibly large with increasing depth. The ALPHA–BETA procedure merely wins a temporary reprieve from the effects of the explosive, exponential growth. The procedure does not prevent the explosive growth, as figure 6.6 shows.

HEURISTIC METHODS

In this section, you learn how to search game trees under time pressure so you can make a reasonable move within allowed time limits. You also learn

how to allocate computational effort so you are as confident as possible that the best choice found is at least a good choice.

Progressive Deepening Keeps Computing Within Time Bounds

In tournaments, players are required to make a certain number of moves within time limits enforced by a relentless clock. This rule creates a problem, because the time required to search to any fixed depth depends on the situation. To search to a fixed depth, independent of the evolving game, you have to use a conservative choice for the fixed depth. Otherwise, you will lose games by running out of time. On the other hand, a conservative choice means that you have to be content to do less search than you could most of the time.

The way to wriggle out of this dilemma is to analyze each situation to depth 1, then to depth 2, then to depth 3, and so on until the amount of time set aside for the move is used up. This way, there is always a move choice ready to go. The choice is determined by the analysis at one level less deep than the analysis in progress when time runs out. This method is called **progressive deepening**.

At first, it might seem that a great deal of time would be wasted in extra analysis at shallow levels. Curiously, however, little time is wasted. To see why, let us suppose, for simplicity, that the dominant cost in analysis is that for static evaluation. The number of nodes requiring static evaluation at the bottom of a tree with depth d and effective branching factor b is b^d. With a bit of algebra, it is easy to show that the number of nodes in the rest of the tree is

$$b^0 + b^1 + \cdots + b^{d-1} = \frac{b^d - 1}{b - 1}.$$

Thus, the ratio of the number of nodes in the bottom level to the number of nodes up to the bottom level is

$$\frac{b^d(b-1)}{b^d - 1} \approx b - 1.$$

For $b = 16$, the number of static evaluations needed to do minimaxing at every level up to the bottom level is but one-fifteenth of the static evaluation needed to do minimaxing at the bottom level. Insurance against running out of time is a good buy.

Heuristic Continuation Fights the Horizon Effect

When a move appears to be distinctly better than the other available moves, that move is said to be **forced**. Interestingly, you can usually determine whether a move is forced by analyzing the local static-value situation. For example, when one of your moves captures a piece, that capture move generally leads to a node whose static value is much better than the rest because most static evaluators weight piece count heavily.

The **singular-extension heuristic** dictates that search should continue as long as one move's static value is much better than the rest, indicating a forced move, for if that static value proves to be wrong, the move is wrong. As shown by figure 6.7, for example, the maximizer's capture at the top of the tree produces a static value that is very different from that of other moves. Similarly, the minimizer's responding capture produces a static value that is much different from the other moves. Eventually, however, all available moves have static values within a narrow range, suggesting that there is forced move at that point; hence, no further search is dictated by the singular-extension heuristic.

Another occasion for search continuation occurs when your analysis stops just before your opponent captures one of your pieces or you capture one of your opponent's pieces. The **search-until-quiescent heuristic** dictates that search should continue until no such captures are imminent.

If you do not use the search-until-quiescent heuristic, the singular-extension heuristic, or a similar heuristic continuation heuristic, you risk harm from the **horizon effect**. Early chess programs searched all paths to the same depth, thus establishing an horizon beyond which disasters often lurked. As a result, those early programs often seized low-valued pieces along paths that led to the loss of high-valued pieces farther on.

In figure 6.7, for example, if the maximizer fails to look beyond the horizon shown, then the maximizer is seduced into a move that leads to a node with a static value of +6. Unfortunately, that move proves to be less advantageous than other moves with lower static values.

Heuristic Pruning Also Limits Search

Pruning procedures are used occasionally, albeit not often, in combination with alpha–beta search to limit tree growth by reducing the effective branching factor.

One way to prune a game tree is to arrange for the branching factor to vary with depth of penetration, possibly using **tapered search** to direct more effort into the more promising moves. You can perform tapered search by ranking each node's children, perhaps using a fast static evaluator, and then deploying the following formula:

$$b(\text{child}) = b(\text{parent}) - r(\text{child}),$$

where $b(\text{child})$ is the number of branches to be retained at some child node, $b(\text{parent})$ is the number of branches retained by the child node's parent, and $r(\text{child})$ is the rank of the child node among its siblings. If a node is one of five children and ranks second most plausible among those five, then it should itself have $5 - 2 = 3$ children. An example of a tree formed using this approach is shown in figure 6.8.

Another way of cutting off disasters is to stop search from going down through apparently bad moves no matter what. If only one line of play makes any sense at all, that line would be the only one pursued.

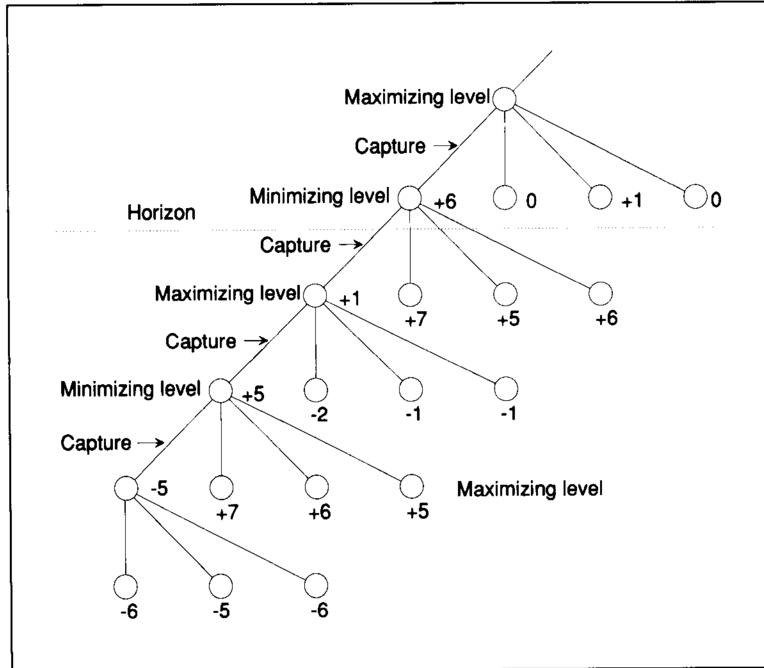


Figure 6.7 An example illustrating how the singular-extension heuristic can fight the horizon effect. All numbers shown are static values. Looking ahead by only one level, the leftmost move looks best, for it has a static value of 6. Actually, the leftmost move is the worst move because its minimax score, determined by looking ahead until there are no forced moves, is −1. Thus the singular-extension heuristic prevents a blunder.

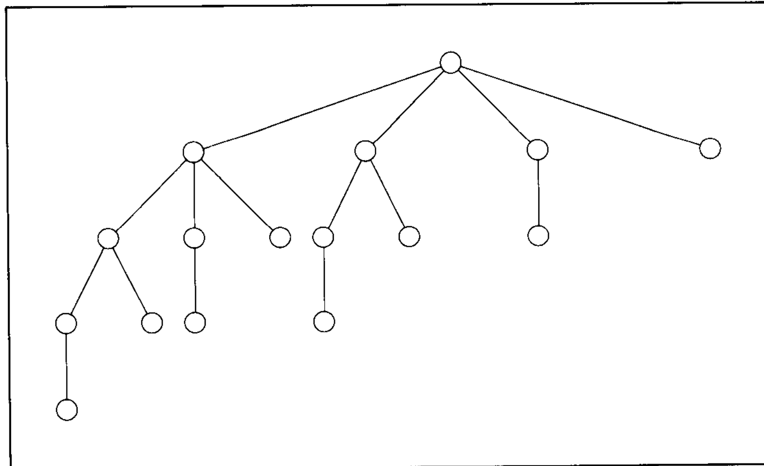


Figure 6.8 Tapering search gives more attention to the more plausible moves. Here, the tapering procedure reduces the search with increases in depth and decreases in plausibility.

Needless to say, any heuristic that limits branching acts in opposition to lines of play that temporarily forfeit pieces for eventual position advantage. Because they trim off the moves that appear bad on the surface, procedures that limit branching are unlikely to discover spectacular moves that seem disastrous for a time, but then win back everything lost and more. There will be no queen sacrifices.

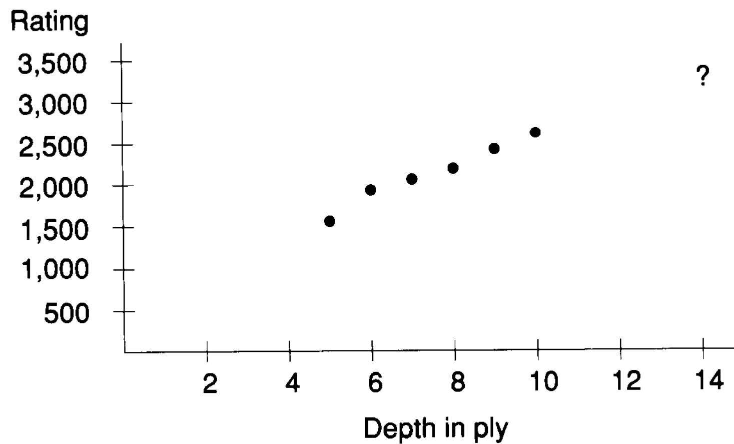
DEEP THOUGHT Plays Grandmaster Chess

To the surprise of many chess experts, an extremely good game of chess can be played by a search-based chess program, as long as the program can search deeply enough. The best such program at this writing, the DEEP THOUGHT program, uses a sophisticated special-purpose computer to perform the search. Using a special-purpose computer, and exploiting the alpha–beta procedure, DEEP THOUGHT usually is able to search down to about 10 ply.

Using the singular extension heuristic, DEEP THOUGHT often goes much further still, shocking some human opponents with its ability to penetrate complicated situations.

DEEP THOUGHT's static evaluator considers piece count, piece placement, pawn structure, passed pawns, and the arrangement of pawns and rooks on columns, which are called files in the chess vernacular.

Curiously, the playing strength of several generations of search-oriented chess programs seems proportional to the number of ply that the programs can search. In the following illustration, program strength, as measured by the U.S. Chess Federation's rating scale, is plotted against search depth in ply:



A successor to DEEP THOUGHT, projected to be 1000 times faster, is under development. Given a real branching factor of 35 to 40 for chess, the effective branching factor is about 6. Thus, 1000 times more speed should enable DEEP THOUGHT's successor to search to about 14 ply. If the relation between ply and rating continues to hold, this next-generation machine should have a rating in the vicinity of 3400, which is well above the 2900 rating of Gary Kasparov, the current world champion.

SUMMARY

- In the context of board games, nodes represent board positions, and branches represent moves.
- The depth, d, of a game tree is the number of moves played. The ply, p, is the depth plus one. The branching factor, b, is the average number of moves available at each node. You cannot play most games by working out all possible paths to game conclusion, because the number of paths, b^d, is too large.
- The MINIMAX procedure is based on the analysis of numeric assessments computed by a static evaluator at the bottom of a game tree.
- The ALPHA–BETA procedure prunes game trees, often cutting the effective branching factor in half, enabling search to go twice as far. The ALPHA–BETA procedure embodies the idea that you should not waste time analyzing options that you know are bad.
- Progressive deepening finds a move with one-move lookahead, then two-move, then three-move, and so on, until time runs out. Progressive deepening both guarantees that there is always a move ready to play and ensures that no time is squandered.
- Chess-playing programs now perform so well that many people believe they will soon overpower all human opponents. All the best chess-playing programs use a form of heuristic continuation to play out capture sequences.

BACKGROUND

The basic minimax approach to games was laid out by Claude E. Shannon [1950], who anticipated most of the subsequent work to date. The classic papers on checkers are by Arthur L. Samuel [1959, 1967]. They deal extensively with tree-pruning heuristics, schemes for combining evidence, and methods for adaptive parameter improvement.

The term *horizon effect* was coined by Hans J. Berliner [1973, 1978]. Berliner's work on chess is complemented by seminal progress on backgammon [Berliner, 1980].

DEEP THOUGHT was developed by Feng-hsiung Hsu, Thomas Anantharaman, Murray Campbell, and Andreas Nowatzyk [1990]. Hsu conceived the singular-extension idea for searching game trees beyond normal depth.

David A. McAllester has proposed an extremely clever approach to growing search trees sensibly without using alpha–beta pruning [1988]. McAllester's approach introduces the notion of a *conspiracy number*, which is the minimum number of static values that you must change to change what the moving player will do. In general, small conspiracy numbers indicate a need for further search.