CHAPTER 5

ADVERSARY SEARCH

This chapter concludes our discussion of search by examining the specific search issues that arise when analyzing game trees.

5.1 ASSUMPTIONS

AI research on game playing typically considers only games that have two specific properties: They are two-person games in which the players alternate moves, and they are games of perfect information, where the knowledge available to each player is the same. There are exceptions to each of these constraints, but most of the work does indeed make these assumptions.

Typical two-person games investigated by AI researchers are tic-tac-toe, checkers, chess, Go, Othello, and backgammon. Other than perhaps Go and Othello, these are probably familiar to you.

Go is played on a $19 \times 19$ board with stones of two colors; the object is to place your stones so that they surround those of your opponent. From an AI point of view, it is remarkable for two reasons:

1. The difference between two Go players can be measured quite precisely. A Go player who beats another by a certain margin in one game is very likely to beat him by a nearly identical margin in subsequent games.

2. Computer Go players are currently quite weak. The branching factor is large; perhaps more importantly, Go is a very “positional” game. The best move is (it seems) selected more on the basis of how the position “looks” than on intricate tactical analysis. In those cases where tactical analysis is required, the line that is analyzed is typically of very low branching factor (perhaps $b = 1$) and high depth (ten or more moves by each player).

Othello is played on a checkerboard. The pieces used are black on one
side and white on the other; a move for White is to place a piece so that a white piece is on each end of some line of Black's pieces. All such black pieces are then flipped and become White's; the object is to end the game with as many pieces of your color as possible. A typical Othello position is shown in Figure 5.1; White's legal moves are denoted by X's. The result if White moves to the square shown in boldface is depicted in the second part of the figure.

Computers are very good at Othello; it is a game that rewards the ability to conduct a brute-force search through one's alternatives and also a game for which it seems to be impossible to recognize a good position at a glance. In fact, computers are significantly better at Othello than people are.

The second assumption made in game search is that the game being considered is one of perfect information. This means that the information available to each of the two players is identical—there is nothing one player knows that the other player could not in principal also know.

Chess is a typical game of perfect information; the differences in beliefs between the two players reflect differing analyses of the position in which they find themselves—not a difference in their raw knowledge about where the pieces are located. Two chess players who disagree about the merits of some particular move will resolve the difference by analyzing the resulting position and not by revealing secrets to one another.

Poker is a classic example of a game of imperfect information. Here, one settles a dispute by revealing the contents of one's hand—information that is presumably not available to one's opponent.

Stratego is another game of imperfect information. Each player in this game has an army of pieces of various ranks that attack one another; in each individual battle, the piece of higher rank wins. Although both players know the locations of each other's pieces, only blue (for example) knows which blue pieces are of what rank.

In a game such as this, it is often sensible to attack a piece of unknown strength with a weak one; this is not because you expect to win the battle but because you can learn the strength of the opposing piece in this way. This is a typical feature of games of imperfect information—one can expend one's limited resources in order to improve the quality of the information available.
(This sort of activity occurs all the time, not just when playing games. When you stop to buy a paper to find out where some movie is playing, you are expending limited resources—time and money—to improve your knowledge about the behavior of the local theaters. And occasionally—just occasionally—stopping to get the paper will mean that you don’t have time to get to the movie; time really is a limited resource that you are expending in order to extend your knowledge in some way.)

Computers make quite good poker players, incidentally. Poker appears to be mostly a matter of working out the probabilities and bluffing with a straight face.

Bridge is another game of imperfect information that has attracted some interest in the AI community. Although there is substantial commercial motivation for developing a competent automated bridge player, the results so far have been disappointing. The reason, it seems, is that bridge is somewhat like Go in that correct bidding and play often hinge (at least initially) upon the sort of pattern recognition that computers find difficult.

The last game of imperfect information that I will mention is called Diplomacy. This is a game where the object is to take over the world by first negotiating treaties with one’s fellow players and then breaking them at a suitable moment. Interestingly enough, computers make excellent Diplomacy players once a suitable negotiation language has been developed.

## 5.2 MINIMAX

Let us return our attention to two-person games of perfect information. We have drawn such a game in Figure 5.2, where we have depicted the game as a tree where the root node is the starting position and the terminal nodes are the ending positions. These terminal nodes are labelled with either 1 or −1 depending on which of the players wins the game.

In a game such as this, one of the players will be trying to get to a node labelled −1, and the other will be trying to get to a node labelled 1. We will call the player trying to achieve an outcome of −1 the minimizer, and the other player the maximizer. We will assume that it is the maximizer’s turn to move in the starting position of Figure 5.2, which corresponds to the root node α.

What is the value of the node α in this position? It’s the maximizer’s turn, and whatever move he makes will end the game. He can move to either p or r and lose, or he can move to q and win. Assuming that he is playing sensibly, he will obviously choose q. This means that the node α is in fact a win for the maximizer, and can be labelled with 1.

What about node k? Whatever the minimizer does, the maximizer will win—either immediately if the minimizer moves to node n, or in one move if he moves to o. So we can label k with a 1 as well. The node i is different,
since here the minimizer has a winning option available (m). So this node should be labelled with a \(-1\).

The algorithm for backing the values up the trees is now apparent. A node with the maximizer to move should be labelled with the maximum value that labels any of its children; a node with the minimizer to move should be labelled with the minimum of its children's labels. We have done this in Figure 5.3, and we see from the fact that the starting position is
labelled with a 1 that perfect play will win this game for the maximizer. This technique is known as minimax.

Before proceeding, suppose that we had continued to back up from the node \( k \), determining that \( g \) and then \( c \) had the values 1. We can now stop our analysis and conclude immediately that the value to be assigned to \( a \) is also 1! The reason is that we know that the maximizer can win from node \( a \) by moving to node \( c \); there is no reason to look for another alternative to this winning line. “Real” games behave similarly; once we find a winning move in a chess game, we typically make that move without examining the alternatives.

Ignoring this possible enhancement, however, here is the basic algorithm for determining the values to be assigned to internal nodes in a game tree.

**PROCEDURE Minimax** To evaluate a node \( n \) in a game tree:

1. Expand the entire tree below \( n \).
2. Evaluate the terminal nodes as wins for the minimizer or maximizing.
3. Select an unlabelled node all of whose children have been assigned values. If there is no such node, return the value assigned to the node \( n \).
4. If the selected node is one at which the minimizer moves, assign it a value that is the minimum of the values of its children. If it is a maximizing node, assign it a value that is the maximum of the children’s values. Return to step 3.

If our game could end in a draw, this would correspond to terminal positions labelled 0 instead of 1 or \(-1\). Draws now fit neatly into the minimax formalism, since each player will pick a winning move if possible and search for a draw if no win exists.

As a matter of terminology, by a move in a game tree we will mean a pair of individual actions, one for each player; this is at odds with common usage. An action by only one player is typically called a half-move or a ply. Thus the depth of the tree in Figure 5.3 is three moves, or 6 ply.

How much memory and time are needed by the minimax algorithm in Procedure 5.2.1? Since the entire tree needs to be expanded, we can expect this procedure to need an exponential amount of space, as the whole fringe apparently needs to be stored before the values are backed up. A moment’s thought reveals, however, that the space needed can be reduced to an amount linear in the depth by searching in a depth-first instead of a breadth-first fashion. Thus in Figure 5.3, we would back the value \(-1\) up to node \( b \) before expanding the children of node \( c \), and so on. This produces the following result, where we update the values assigned to internal nodes as values are assigned to their children:
**PROCEDURE 5.2.2**

**Minimax** To evaluate a node $n$ in a game tree:

1. Set $L = \{n\}$, the unexpanded nodes in the tree.
2. Let $x$ be the first node on $L$. If $x = n$ and there is a value assigned to it, return this value.
3. If $x$ has been assigned a value $v_x$, let $p$ be the parent of $x$ and $v_p$ the value currently assigned to $p$. If $p$ is a minimizing node, set $v_p = \min(v_p, v_x)$. If $p$ is a maximizing node, set $v_p = \max(v_p, v_x)$. Remove $x$ from $L$ and return to step 2.
4. If $x$ has not been assigned a value and is a terminal node, assign it the value 1 or $-1$ depending on whether it is a win for the maximizer or minimizer respectively. Assign $x$ the value 0 if the position is a draw. Leave $x$ on $L$ (we still have to deal with its parent) and return to step 2.
5. If $x$ has not been assigned a value and is a nonterminal node, set $v_x$ to be $-\infty$ if $x$ is a maximizing node and $+\infty$ if $x$ is a minimizing node. (This is to make sure that the first minimization or maximization in step 4 is meaningful.) Add the children of $x$ to the front of $L$ and return to step 2.

Unfortunately, Procedure 5.2.2 continues to use an exponential amount of time to determine the value that is assigned to the node $n$. In general, it is simply impractical to expand the whole tree; as an example, we have already remarked in Chapter 1 that the chess tree contains some $10^{180}$ nodes.\(^{11}\)

How do people manage to play a game like chess? We don’t analyze it all the way to the end; we just look far enough ahead so that we can estimate who’s likely to win in some nonterminal position and then back up the values from the nonterminal positions we have found. Minimax still applies; we just apply it to internal nodes in the tree.

In order to do this, we need a way to assign a value to these internal nodes, so that given a node $n$ we can label it with some estimated value $e(n)$. If $e(n) = 1$, then we believe absolutely that the node is a win for the maximizer; if $e(n) = -1$, the node is believed to be a win for the minimizer. Intermediate values reflect lower levels of certainty; presumably $e(n) = 0$ is used to label a position in which neither side is perceived to have an advantage, just as $e(n) = 0$ is used to label a terminal node that is a draw (that is, a terminal node in which neither player has an advantage).

Crude evaluation functions are often easy to describe. In chess, for example, one can simply add up the values of the pieces each player has (where a pawn counts as 1, a knight 3, a rook 5, and so on according to

\(^{11}\) The graph of chess positions is somewhat smaller because many positions can be reached in a variety of ways. It’s still intractably large, though.
conventional chess wisdom) and then normalize the result so that a value between \(-1\) and \(1\) is returned. If the total value of white's pieces is \(w\) and the value of black's is \(b\), we might use

\[
\frac{w - b}{w + b}
\]  

(5.1)

as the value to be assigned to the overall position.

There are, however, other features that can be included. In chess, a plan of attack for one player often involves targeting the opponent's pawns. The most natural way to defend a pawn is by using another pawn as shown in Figure 5.4; the white pawn on square a4 defends the one on b5.

In some cases, however, it is impossible to use one pawn to defend another. The white pawn on square d3 in the figure cannot be defended by another pawn, since there are no white pawns on either of the two neighboring files (labelled with c and e in the figure). A pawn with no pawns on adjacent files is called isolated, and is a positional flaw whose value has been estimated at \(-1/3\) of a pawn.

Positional features abound in chess. How well protected is your king? How much room do you have in which to maneuver? Do you control the center of the board? And so on.

One interesting thing about these positional features is that evaluating them invariably involves more than simply locating a single one of your pieces on the board. Finding your bishops is a matter of simply that—finding them. But finding your isolated pawns involves finding your pawns and then analyzing their positions relative to your other pawns. As a result, including these positional features in the evaluation function makes \(e(n)\) more expensive to evaluate, and time spent computing \(e(n)\) for the various nodes in the tree is time that cannot be spent searching the tree itself. Here is the base-level/metadata trade-off again.

**FIGURE 5.4**

A pawn defending another

```
     8
    / \
   /   \
  7  
    \
    / \
   /   \
  6  
    \\   \\     \\    \\   \\ \\
  5  
    \\   \\     \\    \\   \\ \\
  4  
    \\   \\     \\    \\   \\ \\
  3  
    \\   \\     \\    \\   \\ \\
  2  
    \\   \\     \\    \\   \\ \\
  1  
    a b c d e f g h
```
Suppose, however, that we have selected some evaluation function \( e(n) \). The modified version of Procedure 5.2.2 is now:

**Procedure**

*Minimax*  
To evaluate a node \( n \) in a game tree:

1. Set \( L = \{n\} \).
2. Let \( x \) be the first node on \( L \). If \( x = n \) and there is a value assigned to it, return this value.
3. If \( x \) has been assigned a value \( v_x \), let \( p \) be the parent of \( x \) and \( v_p \) the value currently assigned to \( p \). If \( p \) is a minimizing node, set \( v_p = \min(v_p, v_x) \). If \( p \) is a maximizing node, set \( v_p = \max(v_p, v_x) \). Remove \( x \) from \( L \) and return to step 2.
4. If \( x \) has not been assigned a value and either \( x \) is a terminal node or we have decided not to expand the tree further, compute its value using the evaluation function. Return to step 2.
5. Otherwise, set \( v_x \) to be \(-\infty\) if \( x \) is a maximizing node and \(+\infty\) if \( x \) is a minimizing node. Add the children of \( x \) to the front of \( L \) and return to step 2.

The interesting new question is one that is implicit in step 4 of this procedure—how do we select the portion of the search tree to expand? Note that this is an issue only because we expect our evaluation function to be imperfect—we cannot typically look at a chess position and decide who will win without doing some amount of analysis. If our evaluation function were somehow perfect, we could simply evaluate the children of the root node and decide from that what to do!

The easiest approach to our computational difficulties is to expand the search to a constant depth, say \( p \) ply. The advantage of this is that it's simple—once again, any metalevel effort devoted to intricate decisions about which nodes to expand must be recovered in terms of more effective base-level search. The constant-depth approach is computationally very efficient, but there are several serious problems with it.

### 5.2.1 Quiescence and Singular Extensions

The first problem is that some portions of a game tree may well be "hotter" than others. A move that leads to tactical considerations involving lengthy exchanges of pieces should probably be investigated in more depth than one that leads to quieter positions. The obvious solution here is to search the tactical portions of the tree to greater depth than the quiet ones, but how are we to recognize these tactical portions automatically?

Tactical portions of the search space are characterized by rapidly changing values of the heuristic evaluation function \( e(n) \). In some sense, this is what it means for a position to be tactical—single moves by each player drastically affect the apparent value of the position.
This also makes it clear why these portions of the space should be searched to greater depth. Since \( e(n) \) is changing rapidly from one ply to the next, the value of \( e(n) \) at any particular point is likely to be fairly unreliable; if the tree can be searched to a depth at which \( e(n) \) is changing only slowly from ply to ply, this value is more likely to be accurate. The technical term quiescence is often used to describe the attempt to search to a depth at which the game becomes fairly quiet.

A related idea that has been implemented to address this concern is that of a singular extension. Here, the assumption is made that a particular node should be evaluated to greater depth if one of the opponent's moves leads to a result vastly preferable to him than all other options. The idea is that since this move is "forced" in some sense, the effective branching factor is small, and an extra ply or so of search will be computationally practical. Experimentation has shown that this idea can substantially improve the performance of chess-playing programs.

5.2.2 The Horizon Effect

The second general difficulty with searching a game tree to a fixed depth is known as the horizon effect, and is best illustrated by an example.

Suppose that we are searching a chess tree to a depth of 7 ply, and that our opponent has a threat that manifests itself at exactly this depth. We would expect that if we do not respond to the threat, our 7-ply search will reveal the problem and we will therefore be led to a move that addresses it.

But suppose that in the middle of the 7-ply combination, we have some useless move that serves no real purpose other than demanding a response from our opponent. Perhaps we throw in a "nuisance" check, forcing a king move before the rest of the combination can be executed.

The nuisance move hasn't really helped us in any way; in general, such maneuvers tend to make our position worse rather than better. But what the nuisance move has done is to make the length of our opponent's combination 9 ply instead of 7. As a result, our 7-ply search will fail to detect the threat when examining the line with the nuisance move in it, and will therefore conclude that the move defeats our opponent's threat! What we have done is to push the culmination of the threat over the search horizon, and mistakenly confused this with the presumably more effective (and necessary) option that deals with the threat instead.

This is a very subtle problem, since it isn't clear how we can distinguish between a move that really does neutralize the threat and one that only appears to do so by pushing the threat past the search horizon. The difference is that in the second case, the threat reappears two ply later; in the first case, it doesn't. But determining this involves a search to depth 9, not 7.
(People, of course, deal with this problem by realizing what’s going on and determining whether the threat has been addressed or not. But people know what they are doing when they play games; computers are just searching through partial position trees.)

There have been two solutions proposed to the horizon problem; neither is really satisfactory, but we will discuss them both.

**Secondary search** One proposed solution is to examine the search space beneath the apparently best move to see if something has in fact been pushed just beyond the horizon so that an alternate move should be chosen.

Unfortunately, although this technique can be used to detect the horizon effect, it doesn’t really tell us what to do about it. Consider the chess situation again. The move that actually addresses the threat may well appear to be a fairly weak one, since responding to our opponent’s threat means that we won’t be able to pursue our own intentions.

As a result, if we manage to use this sort of a secondary search to determine that the horizon effect invalidates the apparently best move at a depth of 7 ply, the second-best move is likely to fall prey to the same problem. The move that actually addresses the difficulty is not likely to be expanded until quite late in the search, and there will not be sufficient time to perform secondary searches beneath all of these other moves.

**The killer heuristic** The second idea that can help with the horizon problem is known as the killer heuristic. What this suggests is that if you find that a move that is good for your opponent, you look at this move early when considering your opponent’s options.

This can be combined with the ideas of the previous paragraph as follows: When the apparently best move is found, a secondary search is performed to check it for possible flaws. Let’s say that such a flaw is found; our opponent can defeat our 7-ply plan with a particular move at ply 8. We can now do partial secondary searches beneath our other alternatives to determine whether or not these alternatives suffer from the same difficulty. By constraining the size of all of the secondary searches except one, some instances of the horizon problem can be avoided.

**5.3 α-β SEARCH**

The real problem with adversary search, however, is the one that we referred to in our parenthetical remark in the previous section: Computers, playing games, are simply searching through large trees looking for nodes with certain mathematical properties. As a result, most of the nodes examined in the search for a move have no bearing on the course of the game—they are at best pointless and at worst suicidal. Chess-playing programs consider irrelevant pawn moves and meaningless piece sacrifices with the same care that they consider winning combinations or careful positional
improvements. If we want to improve the performance of these programs, we need a way to reduce the size of the search space.

The most powerful technique known for doing this is called $\alpha$-$\beta$ search. We've already seen an example of it in Figure 5.3, which we repeat here as Figure 5.5.

What we noticed about the search in Figure 5.5 is that once we realize that the maximizer can win by moving to $c$, we no longer need to analyze any of his other options from $a$. Specifically, the values assigned to all of the nodes under the node $b$ are guaranteed not to affect the overall value assigned to the position $a$.

Another example appears in Figure 5.6. Here, we are considering our option $c$ of attacking our opponent's queen and are assuming as usual that we are the maximizing player.

If we do attack our opponent's queen, we can be mated at the next move, thereby ending the game and achieving an outcome of $-1$ (that is, we lose). Clearly, there is no need to consider any of $c$'s other children—we've already seen enough to realize that we should not play $c$!
A slightly more complicated example appears in Figure 5.7. Here, suppose that we are analyzing the tree in a depth-first manner and have determined that the value to be assigned to node $b$ is $.03$, slightly favorable to us. We continue by examining our other option $c$, attacking our opponent's queen with our own.

The first response we consider is $d$, involving an exchange of queens. Further examination of $d$'s children $e$ and $f$ show that $f$ is the better of the two moves, and leads to a backed-up value of $-.05$ for the node $d$.

Before examining node $g$, suppose that we stop to take stock. Is $c$ ever going to be our final selection? If we move to $c$, the best result we can expect to obtain is $-.05$, since that's how well we'll do if our opponent replies with $d$. If $g$ is better for our opponent than $d$ is, the value to be assigned to $c$ will be even lower than $-.05$. Why should we accept such a value when we can obtain the outcome of $.03$ by moving to node $b$?

What we have shown, in effect, is that the node $c$ is never on the main line, which is the course the game would take if both players played optimally. Since $b$ is better for the maximizer than $c$ is and the maximizer can select between them, $c$ is guaranteed to be avoided. The values assigned to $g$ and the nodes under it can never impact the final value assigned to the node $a$, and the subtree under $g$ can be pruned.

We can reach the same conclusion algebraically. If we denote by $g$ the backed-up value assigned to the node $g$, then the value assigned to the node $c$ will be

$$c = \min(-.05, g)$$

since it is the minimizer who will choose between the alternatives $d$ and $g$.

Continuing, the value assigned to the root node $a$ is

$$a = \max[.03, \min(-.05, g)] = .03$$

(5.2)

where the second equality holds because $\min(-.05, g) \leq -.05 < .03$. Since
the value assigned to the node $a$ is independent of the value $g$, we see that the nodes below $g$ can be pruned.

A still more complicated example appears in Figure 5.8. Here, the values assigned to $b$ and to $f$ are .03 and $-.1$ respectively. What can we say about the children of the node $g$? Can it be shown that $g$ is not on the main line?

Well, if $g$ is going to be on the main line, then its ancestors will need to be as well, so that $e$ will be on the main line. But from $e$ the minimizer has the opportunity to move to node $f$ and obtain a payoff of $-.1$. Since this is worse than the payoff of .03 that the maximizer can obtain by moving to $b$, it follows that $g$ cannot be on the main line. Note that it is still possible that $h$ is on the main line, or that $i$ is—all we can say for sure is that $g$ is not. A situation such as this, where the pruned node is more than one ply below the reason for the pruning, is typically referred to as deep $\alpha\beta$ pruning.

What about the general case? Suppose that $n$ is some node in a game tree at which the maximizer gets to move (like the node $g$ in Figure 5.8), and that $s$ is a sibling of $n$ with backed-up value $v_s$ ($s$ would be the node $f$ in the above figure, so that $v_s = -.1$). Suppose also that $(p_0, p_1, \ldots, p_k)$ is the path from the root node down to $n$, with $p_0$ the root node and $p_k = n$. Now note that the nodes with odd indexes ($p_1, p_3, \text{and so on}$) are minimizing nodes; if any of these nodes has a sibling with backed-up value greater than $v_s$, then the node $n$ can be pruned. (In the figure, $p_1 = c$ has the sibling $b$ with backed-up value $.03 > v_s$.) Here is another way to put it.

**PROCEDURE 5.3.1** $\alpha\beta$ search To evaluate a node $n$ in a game tree:

1. Set $L = \{n\}$.
2. Let $x$ be the first node on $L$. If $x = n$ and there is a value assigned to it, return this value.
3. If $x$ has been assigned a value $v_x$, let $p$ be the parent of $x$; if $x$ has not been assigned a value, go to step 5. We first determine whether or not $p$ and its children can be pruned from the tree: If $p$ is a minimizing node, let $\alpha$ be the maximum of all the current values assigned to siblings of $p$ and of the minimizing nodes that are ancestors of $p$. If there are no such values, set $\alpha = -\infty$. If $v_x \leq \alpha$, remove $p$ and all of its descendants from $L$. If $p$ is a maximizing node, treat it similarly.

4. If $p$ cannot be pruned, let $v_p$ be the value currently assigned to $p$. If $p$ is a minimizing node, set $v_p = \min(v_p, v_x)$. If $p$ is a maximizing node, set $v_p = \max(v_p, v_x)$. Remove $x$ from $L$ and return to step 2.

5. If $x$ has not been assigned a value and is either a terminal node or we have decided not to expand the tree further, compute its value using the evaluation function. Return to step 2.

6. Otherwise, set $v_x$ to $-\infty$ if $x$ is a maximizing node and $+\infty$ if $x$ is a minimizing node. Add all of the children of $x$ to the front of $L$ and return to step 2.

The value corresponding to $\alpha$ but computed by considering $p$’s maximizing ancestors is called $\beta$; the two values lead to $\alpha$ and $\beta$ cutoffs respectively. Because of this, this method of reducing the size of the search tree is called $\alpha$-$\beta$ pruning. It’s a pretty dumb name.

All of this is well and good, but what does it buy us? Although we have seen that $\alpha$-$\beta$ pruning can reduce the size of the search space associated with a game tree, we haven’t discussed by how much the search space is reduced.

It is clear, for example, that in the worst case it is possible that $\alpha$-$\beta$ pruning fails to reduce the size of the search space at all. If we perversely order the children of every node so that the worst options are evaluated first, then the nodes examined later will always be the “main line” and will therefore never be pruned.

What about the best case? What if we somehow manage to order the nodes so that the best moves are examined first? We will still need to search the space to confirm that these actually are the best moves, but now $\alpha$-$\beta$ pruning can save us a great deal of work.

How much is saved? Suppose that we consider a response for the minimizer that is off the main line. In order to prune it, we need to examine just enough of the search space to demonstrate that it is a mistake for the minimizer—in other words, we have to examine the “refutation” of this move that shows how the maximizer can exploit it. In order to do this, we have to examine only one response on the maximizer’s part—the best one. Then we have to examine all of the minimizer’s options, the maximizer’s best response to each, and so on. So although the branching factor for the
minimizer is unchanged, the branching factor for the maximizer is reduced to just 1.

If the maximizer deviates, the analysis is similar, but with the roles of the two players reversed. It follows that the total number of nodes that we need to examine to depth \( d \) is approximately

\[
b^{d/2} + b^{d/2} \tag{5.3}
\]

where \( b \) is the branching factor in the game. The two terms correspond to analysis of the situations where the maximizer and minimizer deviate respectively. In each case, the search to depth \( d \) involves \( d/2 \) nodes with \( b \) children and \( d/2 \) nodes with only one child. Thus the size of the associated search space is \( b^{d/2} \) and the expression (5.3) follows.

Instead of searching a space of size \( b^d \), we need to search one of size \( 2b^{d/2} \); this is potentially a tremendous savings. It is frequently described in terms of an "effective" branching factor, the branching factor \( b' \) such that \( b'^d \) is the size of the space searched. In this case, we have

\[
b'^d = 2b^{d/2}
\]

leading to

\[
b' = 2^{1/d} \sqrt{b} \approx \sqrt{b}
\]

so that the effective branching factor is approximately the square root of the actual branching factor. Yet another way to look at this is to realize that in this best case, \( \alpha-\beta \) pruning allows us effectively to double the depth to which we can search in a fixed amount of time.

These observations mean that it is very important when using \( \alpha-\beta \) search to do a good job of ordering the children of any particular node. A variety of methods exist for this—the evaluation function can be applied to the internal nodes to order them heuristically, the killer heuristic discussed in Section 5.2.2 can be used to move nodes that work well elsewhere to the front of the list of children, and so on.

In practice, this turns out to be an effective way to approximately order the internal nodes: when searching a game tree using \( \alpha-\beta \) pruning. Sophisticated chess programs, for example, typically investigate only \( 1^{1/2} \) times the theoretically minimal number of nodes using this method. We can come quite close to the theoretical limits described above.

### 5.4 Further Reading

The world’s best Othello player is a computer program called BILL [Lee and Mahajan, 1990]. The Diplomacy player mentioned in the text is described in Kraus and Lehmann [1988].
Games that violate the typical AI assumptions by involving more than two players, simultaneous action, or including imperfect information are typically the focus of game theorists or economists and not computer scientists; a good introduction to game theory is Luce and Raiffa [1957]. Recently, some authors interested in distributed AI (the study of how our putatively intelligent artifacts will interact with one another) have observed that many game-theoretic results can be applied if these independent agents are viewed as players in a formal multiagent game. The work of Genesereth et al. [1986] and Rosenschein and Genesereth [1985], reprinted in Bond and Gasser [1988], is typical of this approach.

Singular extensions are introduced by Anantharaman et al. [1990] and are believed by many researchers to be the main reason that the chess program DEEP THOUGHT outperforms its predecessor, HITECH.

We remarked in the text that the techniques used to order the search in existing game-playing programs result in their behavior approximating that of the best-case analysis we have presented. As we have explained, this best case doubles the effective search depth; the worst case multiplies it by 1 (that is, leaves it unchanged). It is shown in Pearl [1982] that if the children of a node are randomly ordered, the effective search depth is multiplied by a factor of approximately 4/3.

5.5 EXERCISES

1. The work on game search assumes that deeper searches are more accurate than shallow ones, so that a search algorithm can improve its performance by searching to greater depth. Either prove this to be true or find a game and evaluation function for which it fails, in the sense that the result returned becomes uniformly less accurate as the search deepens. (Search to terminal nodes doesn’t count, of course, since it always evaluates correctly.)

2. Consider the chess evaluation function given by (5.1).
   (a) Under what circumstances will this evaluation function take the values +1, −1, or 0?
   (b) What material value should be assigned to a king if this evaluation function is used?
   (c) The conventional wisdom in chess is that if you are a pawn ahead, you should try to exchange other pieces so that your extra pawn has a more substantial effect. Does (5.1) support this idea?

3. What might be a sensible heuristic evaluation function for checkers? Consider only the number of men and kings each side has.

4. What modifications should be made to Procedure 5.2.3 to take advantage of the fact that the space being searched may be a graph instead