Game Trees

Many two-player games can be efficiently represented using trees, called **game trees**. A game tree is an instance of a tree in which the root node rep- resents the state before any moves have been made, the nodes in the tree represent possible states of the game (or **positions**), and arcs in the tree represent moves.

It is usual to represent the two players' moves on alternate levels of the game tree, so that all edges leading from the root node to the first level rep- resent possible moves for the first player, and edges from the first level to the second represent moves for the second player, and so on.

Leaf nodes in the tree represent final states, where the game has been won, lost, or drawn. In simple games, a goal node might represent a state in which the computer has won, but for more complex games such as chess and Go, the concept of a goal state is rarely of use.

One approach to playing a game might be for the computer to use a tree search algorithm such as depth-first or breadth-first search, looking for a goal state (i.e., a final state of the game where the computer has won). Unfortunately, this approach does not work because there is another intelligence involved in the game. We will consider this to be a rational, informed opponent who plays to win. Whether this opponent is human, or another computer does not matter—or should not matter—but for the purposes of this section of the book, we will refer to the opponent as being human, to differentiate him or her from the computer.

Consider the game tree shown in Figure 6.1. This partial tree represents the game of tic-tac-toe, in which the computer is playing noughts, and the human opponent is playing crosses. The branching factor of the root node is 9 because there are nine squares in which the computer can place its first nought. The branching factor of the next level of the tree is 8, then 7 for the next level, and so on. The tree shown in Figure 6.1 is clearly just a part of that tree and has been pruned to enable it to fit comfortably on the page.

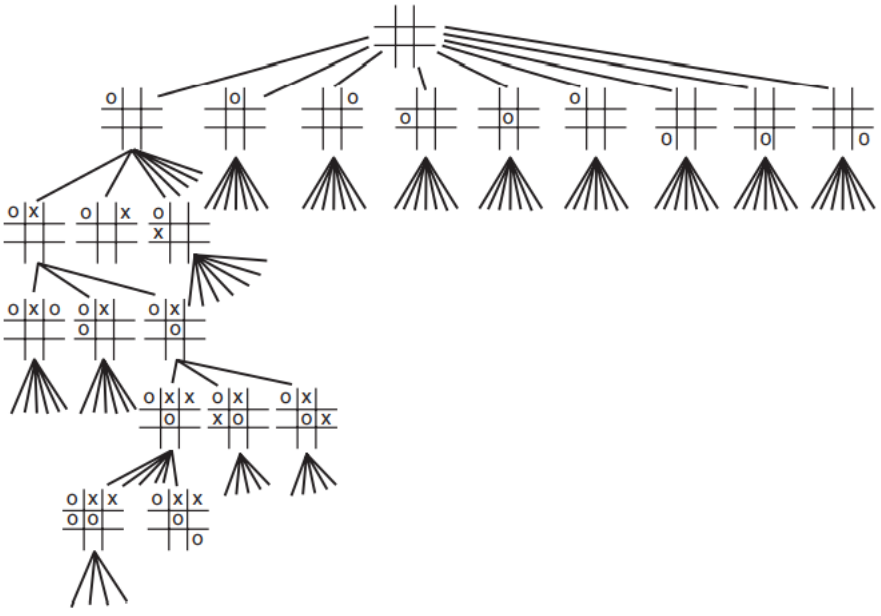

Figure 6.1
A partial game tree for the game tic-tac-toe

For a computer to use this tree to make decisions about moves in a game of tic-tac-toe, it needs to use an evaluation function, which enables it to decide whether a given position in the game is good or bad. If we use exhaustive search, then we only need a function that can recognize a win, a loss, and a draw. Then, the computer can treat "win" states as goal nodes and carry out search in the normal way.

Rationality, Zero Sum, and Other Assumptions

All the methods discussed in this chapter are designed for games with two players. In most of the games, there is no element of chance (in other words, no dice are thrown, or cards drawn), and the players have complete knowledge of the state of the game, which means that the players do not conceal information (apart from their strategies and plans) from their opponents. This sets games such as chess and Go aside from games such as poker, in which there is an element of chance, and it is also important that players conceal information from each other.

Most of the games we will consider in this chapter are **zero-sum games**, which means that if the overall score at the end of a game for each player can be 1 (a win), 0 (a draw), or —1 (a loss), then the total score for both players for any game must always be 0. In other words, if one player wins, the other must lose. The only other alternative is that both players draw. For this reason, we consider the search techniques that are discussed here to be **adversarial** methods because each player is not only trying to win but to cause the opponent to lose. In the algorithms such as Minimax and alpha–beta that are discussed later, it is important that the computer can assume that the opponent is rational and adversarial. In other words, the computer needs to assume that the opponent will play to win.

In discussing game trees, we use the concept of ply, which refers to the depth of the tree. In particular, we refer to the ply of lookahead. When a computer evaluates a game tree to ply 5, it is examining the tree to a depth of 5. The 4th ply in a game tree is the level at depth 4 below the root node.

Because the games we are talking about involve two players, sequential plies in the tree will alternately represent the two players. Hence, a game tree with a ply of 8 will represent a total of eight choices in the game, which corresponds to four moves for each player. It is usual to use the word ply to represent a single level of choice in the game tree, but for the word move to represent two such choices—one for each player.

Evaluation Functions

Evaluation functions (also known as **static evaluators** because they are used to evaluate a game from just one static position) are vital to most game-playing computer programs. This is because it is almost never possible to search the game tree fully due to its size. Hence, a search will rarely reach a leaf node in the tree at which the game is either won, lost, or drawn, which means that the software needs to be able to **cut off** search and evaluate the position of the board at that node. Hence, an evaluation function is used to examine a particular position of the board and estimate how well the computer is doing, or how likely it is to win from this position. Due to the enormous number of positions that must be evaluated in game playing, the evaluation function usually needs to be extremely efficient, to avoid slowing down game play.

One question is how the evaluation function will compare two positions. In other words, given positions A and B, what relative values will it give those positions? If A is a clearly better position than B, perhaps A should receive a much higher score than B. In general, as we will see elsewhere, to be successful, the evaluation function does not need to give values that linearly represent the quality of positions: To be effective, it just needs to give a higher score to a better position.

An evaluation function for a chess game might look at the number of pieces, taking into account the relative values of particular pieces, and might also look at pawn development, control over the center of the board, attack strength, and so on. Such evaluation functions can be extremely complex and, as we will see, are essential to building successful chess-playing software.

Evaluation functions are usually **weighted linear functions**, meaning that a number of different scores are determined for a given position and simply added together in a weighted fashion. So, a very simplistic evaluation function for chess might count the number of queens, the number of pawns, the number of bishops, and so on, and add them up using weights to indicate the relative values of those pieces:

q = number of queens
r = number of rooks
n = number of knights
b = number of bishops
p = number of pawns

$score = 9q + 5r + 3b + 3n + p$

If two computer programs were to compete with each other at a game such as checkers, and the two programs had equivalent processing capabilities and speeds and used the same algorithms for examining the search tree, then the game would be decided by the quality of the programs' evaluation functions.

In general, the evaluation functions for game-playing programs do not need to be perfect but need to give a good way of comparing two positions to determine which is the better. Of course, in games as complex as chess and Go, this is not an easy question: two grandmasters will sometimes differ on the evaluation of a position.

As we will see, one way to develop an accurate evaluation function is to actually play games from each position and see who wins. If the play is perfect on both sides, then this will give a good indication of what the evaluation of the starting position should be.

This method has been used successfully for games such as checkers, but for games such as chess and Go, the number of possible positions is so huge that evaluating even a small proportion of them is not feasible. Hence, it is necessary to develop an evaluation function that is dynamic and is able to accurately evaluate positions it has never seen before.

Searching Game Trees

Even for the game of tic-tac-toe, a part of whose game tree is illustrated in Figure 6.1, it can be inefficient for the computer to exhaustively search the tree because it has a maximum depth of 9 and a maximum branching factor of 9, meaning there are approximately $9 \times 8 \times 7 \times \ldots \times 2 \times 1$ nodes in the tree, which means more than 350,000 nodes to examine. Actually, this is a very small game tree compared with the trees used in games like chess or Go, where there are many more possible moves at each step and the tree can potentially have infinite depth.

In fact, using exhaustive search on game trees is almost never a good idea for games with any degree of complexity. Typically, the tree will have very high branching factors (e.g., a game tree representing chess has an average branching factor of 38) and often will be very deep. Exhaustively searching such trees is just not possible using current computer technology, and so in this chapter, we will explore methods that are used to **prune** the game tree and heuristics that are used to evaluate positions.

There is another problem with using exhaustive search to find goal nodes in the game tree. When the computer has identified a goal state, it has sim- ply identified that it can win the game, but this might not be the case because the opponent will be doing everything, he or she can stop the computer from winning. In other words, the computer can choose one arc in the game tree, but the opponent will choose the next one. It may be that depth-first search reveals a path to a leaf node where the computer wins, but the computer must also assume that the opponent will be attempting to choose a different path, where the computer loses.

So, as we see later in this chapter, the computer can use methods like depth- first or breadth-first search to identify the game tree, but more sophisticated methods need to be used to choose the correct moves.

Minimax

When evaluating game trees, it is usual to assume that the computer is attempting to maximize some score that the opponent is trying to minimize. Normally we would consider this score to be the result of the evaluation function for a given position, so we would usually have a high positive score mean a good position for the computer, a score of 0 mean a neutral position, and a high negative score mean a good position for the opponent.

The Minimax algorithm is used to choose good moves. It is assumed that a suitable static evaluation function is available, which is able to give an over- all score to a given position. In applying Minimax, the static evaluator will only be used on leaf nodes, and the values of the leaf nodes will be filtered up through the tree, to pick out the best path that the computer can achieve.

This is done by assuming that the opponent will play **rationally** and will always play the move that is best for him or her, and thus worst for the computer. The principle behind Minimax is that a path through the tree is chosen by assuming that at its turn (a **max node**), the computer will choose the move that will give the highest eventual static evaluation, and that at the human opponent's turn (a **min node**), he or she will choose the move that will give the lowest static evaluation. So, the computer's aim is to maximize the lowest possible score that can be achieved.

Figure 6.2 shows how Minimax works on a very simple game tree. Note that the best result that max can achieve is a score of 6. If max chooses the left branch as its first choice, then min will inevitably choose the right branch, which leaves max a choice of 1 or 3. In this case, max will choose a score of 3. If max starts by choosing the right branch, min will have a choice between a path that leads to a score of 7 or a path that leads to a score of 6. It will therefore choose the left branch, leaving max a choice between 2 and 6.

Figure 6.2 shows how Minimax can use depth-first search to traverse the game tree. The arrows start from the root node at the top and go down to the bottom of the left branch.

This leads to a max node, which will get a score of 5. The value 5 is there- fore passed up to the parent of this max node. Following the right path from this min node leads to another max node, this time getting a score of 3. This comes back up to the min node, which now chooses the minimum of 3 and 5, and selects 3. Eventually, having traversed the whole tree, the best result for max comes back up to the root node: 6.

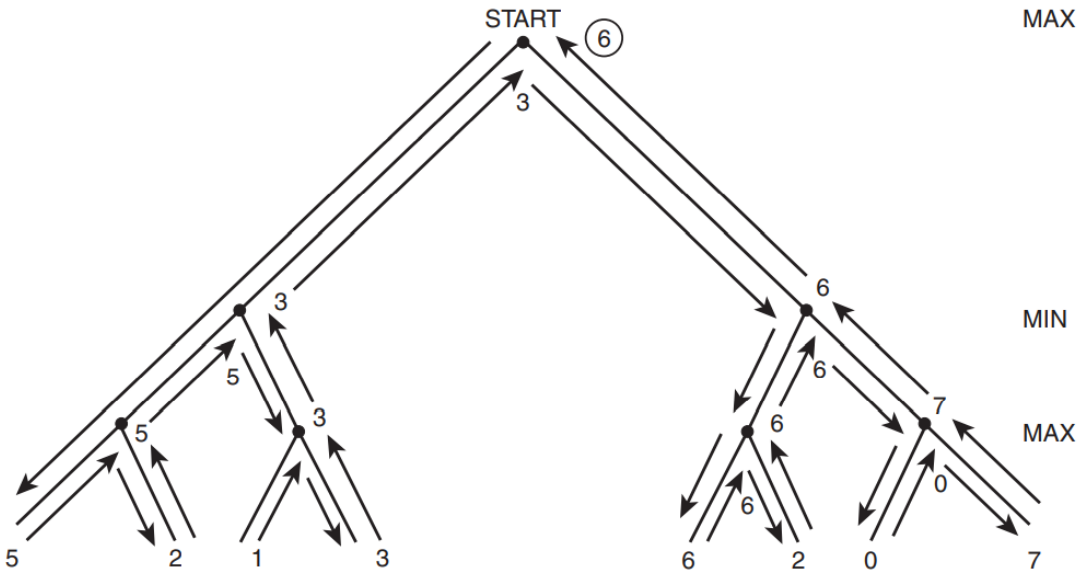

Figure 6.2

Illustrating how minimax works on a very simple game tree. The arrows show the order in which the nodes are examined by the algorithm, and the values that are passed through the tree.

The Minimax function provides a best available score for a given node as follows:

```
Function minimax (current_node)
{
     if is_leaf (current_node)
          then return static_evaluation (current_node);if
     is_min_node (current_node)
          then return min (minimax (children_of
          (current_node)));
     if is_max_node (current_node)
          then return max (minimax (children_of
          (current_node)));
     // this point will never be reached since
     // every node must be a leaf node, a min node or a
     // max node.
}
```

This is a recursive function because to evaluate the scores for the children of the current node, the Minimax algorithm must be applied recursively to those children until a leaf node is reached.

Minimax can also be performed nonrecursively, starting at the leaf nodes and working systematically up the tree, in a reverse breadth-first search.

Bounded Lookahead

Minimax, as we have defined it, is a very simple algorithm and is unsuitable for use in many games, such as chess or Go, where the game tree is extremely large. The problem is that in order to run Minimax, the entire game tree must be examined, and for games such as chess, this is not possible due to the potential depth of the tree and the large branching factor.

In such cases, **bounded lookahead** is very commonly used and can be combined with Minimax. The idea of bounded lookahead is that the search tree is only examined to a particular depth. All nodes at this depth are considered to be leaf nodes and are evaluated using a static evaluation function. This corresponds well to the way in which a human plays chess. Even the greatest grand- masters are not able to look forward to see every possible move that will occur in a game. Chess players look forward a few moves, and good chess players may look forward a dozen or more moves. They are looking for a move that leads to as favorable a position as they can find and are using their own static evaluator to determine which positions are the most favorable.

Hence, the Minimax algorithm with bounded lookahead is defined as follows:

```
Function bounded_minimax (current_node, max_depth)
{
     if is_leaf (current_node)
          then return static_evaluation (current_node);if
     depth_of (current_node) == max_depth
          then return static_evaluation (current_node);if
     is_min_node (current_node)
          then return min (minimax (children_of
          (current_node)));
     if is_max_node (current_node)
          then return max (minimax (children_of
          (current_node)));
     // this point will never be reached since
     // every node must be a leaf node, a min node or a
     // max node.
}
```

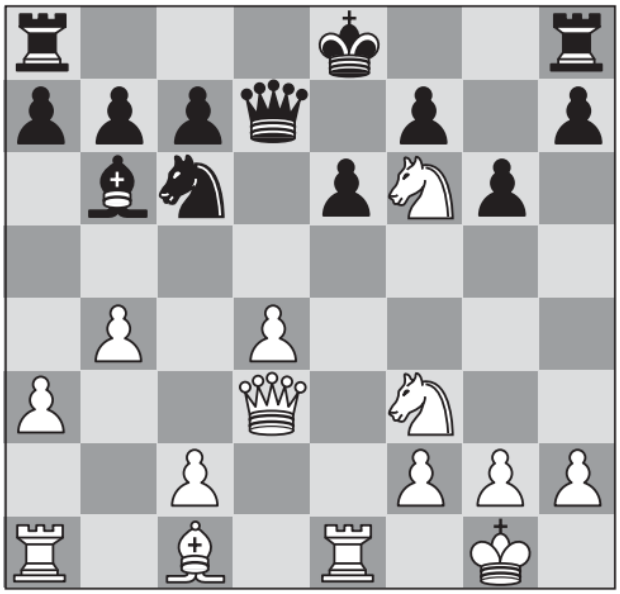

Figure 6.3
Chess position with black to move

In fact, it is not necessarily sensible to apply a fixed cut-off point for search. The reason for this can be seen from the chess position shown in Figure 6.3. If bounded Minimax search cut off search at this node, it might consider the position to be reasonably even because the two players have the same pieces, which are roughly equally well developed. In fact, although it is black's turn to move, white will almost certainly take black's queen after this move, meaning that the position is extremely strong for white.

This problem must be avoided if a computer program is to play chess or any other game successfully. One way to avoid the problem is to only cut off search at positions that are deemed to be quiescent. A quiescent position is one where the next move is unlikely to cause a large change in the relative positions of the two players. So, a position where a piece can be captured without a corresponding recapture is not quiescent.

Another problem with bounded Minimax search is the **horizon problem**. This problem involves an extremely long sequence of moves that clearly lead to a strong advantage for one player, but where the sequence of moves, although potentially obvious to a human player, takes more moves than is allowed by the bounded search. Hence, the significant end of the sequence has been pushed over the horizon. This was a particular problem for **Chinook**, the checkers-playing program that we learn more about in Section 6.5.

There is no universal solution to the horizon problem, but one method to minimize its effects is to always search a few ply deeper when a position is found that appears to be particularly good. The singular-extension heuristic is defined as follows: if a static evaluation of a move is much better than that of other moves being evaluated, continue searching.

Alpha-Beta Pruning

Bounded lookahead can help to make smaller the part of the game tree that needs to be examined. In some cases, it is extremely useful to be able to prune sections of the game tree. Using alpha–beta pruning, it is possible to remove sections of the game tree that are not worth examining, to make searching for a good move more efficient.

The principle behind alpha–beta pruning is that if a move is determined to be worse than another move that has already been examined, then further examining the possible consequences of that worse move is pointless.

Consider the partial game tree in Figure 6.4.

This very simple game tree has five leaf nodes. The top arc represents a choice by the computer, and so is a **maximizing level** (in other words, the top node is a max node). After calculating the static evaluation function for the first four leaf nodes, it becomes unnecessary to evaluate the score for the fifth. The reason for this can be understood as follows:

In choosing the left-hand path from the root node, it is possible to achieve a score of 3 or 5. Because this level is a minimizing level, the opponent can be expected to choose the move that leads to a score of 3. So, by choosing the left-hand arc from the root node, the computer can achieve a score of 3.

By choosing the right-hand arc, the computer can achieve a score of 7 or 1, or a mystery value. Because the opponent is aiming to minimize the score, he or she could choose the position with a score of 1, which is worse than the value the computer could achieve by choosing the left-hand path. So, the value of the rightmost leaf node doesn't matter—the computer must not choose the right-hand arc because it definitely leads to a score of at best 1 (assuming the opponent does not irrationally choose the 7 option).

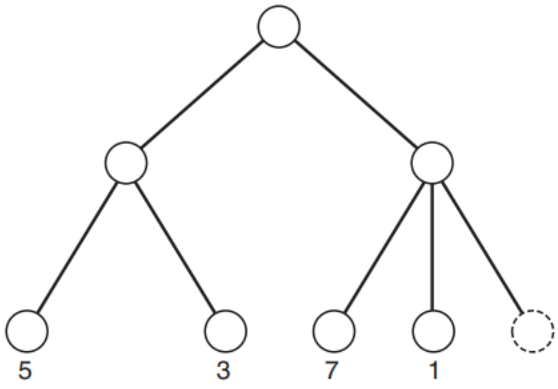

Figure 6.4
A partial game tree

The Effectiveness of Alpha–Beta Pruning

In this contrived example, alpha–beta pruning removes only one leaf node from the tree, but in larger game trees, it can result in fairly valuable reductions in tree size. However, as Winston (1993) showed, it will not necessarily remove large portions of a game tree. In fact, in the worst case, alpha–beta pruning will not prune any searches from the game tree, but even in this case it will compute the same result as Minimax and will not perform any less efficiently.

The alpha–beta pruning method provides its best performance when the game tree is arranged such that the best choice at each level is the first one (i.e., the left-most choice) to be examined by the algorithm. With such a game tree, a Minimax algorithm using alpha–beta cut-off will examine a game tree to double the depth that a Minimax algorithm without alpha–beta pruning would examine in the same number of steps.

This can be shown as follows:

If a game tree is arranged optimally, then the number of nodes that must be examined to find the best move using alpha–beta pruning can be derived as follows:

$$S = \begin{cases} 2b^{d/2} - 1 & \text{if } d \text{ is even} \\ b^{(d+1)/2} + b^{(d-1)/2} - 1 & \text{if } d \text{ is odd} \end{cases}$$

where

b = branching factor of game tree

d = depth of game tree

s = number of nodes that must be examined

This means that approximately

$$s = 2b^{d/2}$$

Without alpha–beta pruning, where all nodes must be examined:

$$s = b^{d}$$

Hence, we can consider that using alpha–beta pruning reduces the effective branching factor from b to $\sqrt{b}$, meaning that in a fixed period of time, Minimax with alpha–beta pruning can look twice as far in the game tree as Minimax without pruning.

This represents a significant improvement—for example, in chess it reduces the effective branching factor from around 38 to around 6—but it must be remembered that this assumes that the game tree is arranged optimally (such that the best choice is always the left-most choice). In reality, it might provide far less improvement.

It was found that in implementing the Deep Blue chess computer (see Section 6.6), use of the alpha–beta method did in fact reduce the average branching factor of the chess game tree from 38 to around 6.

Implementation

The alpha-beta pruning algorithm is implemented as follows:

- The game tree is traversed in depth-first order. At each non-leaf node, a value is stored. For max nodes, this value is called alpha, and for min nodes, the value is beta.
- An alpha value is the maximum (best) value found so far in the max node's descendants.
- A beta value is minimum (best) value found so far in the min node's descendants.

In the following pseudo-code implementation, we use the function call `beta_value_of (min_ancestor_of (current_node))`, which returns the beta value of some min node ancestor of the current node to see how it com- pares with the alpha value of the current node. Similarly, `alpha_value_of (max_ancestor_of (current_node))` returns the alpha value of some max node ancestor of the current node in order that it be compared with the beta value of the current node.

```
Function alpha_beta (current_node)
{
       if is_leaf (current_node)
              then return static_evaluation (current_node);
       if is_max_node (current_node) and
              alpha_value_of (current_node) >=
              beta_value_of (min_ancestor_of (current_node))
       then cut_off_search_below (current_node);
       if is_min_node (current_node) and
              beta_value_of (current_node) <=
              alpha_value_of (max_ancestor_of (current_node))
       then cut_off_search_below (current_node);
}
```

To avoid searching back up the tree for ancestor values, values are propagated down the tree as follows:

- For each max node, the minimum beta value for all its min node ancestors is stored as beta.
- For each min node, the maximum alpha value for all its max node ancestors is stored as alpha.
- Hence, each non-leaf node will have a beta value and an alpha value stored.
- Initially, the root node is assigned an alpha value of negative infinity and a beta value of infinity.

So, the `alpha_beta` function can be modified as follows. In the following pseudo-code implementation, the variable `children` is used to represent all of the children of the current node, so the following line:

```
alpha = max (alpha, alpha_beta (children, alpha, beta));
```

means that alpha is set to the greatest of the current value of alpha, and the values of the current node's children, calculated by recursively calling `alpha_beta`.

```
       Function alpha_beta (current_node, alpha, beta)
       {
              if is_root_node (current_node)
              then
              {
                     alpha = -infinity
                     beta = infinity
              }
              if is_leaf (current_node)
              then return static_evaluation (current_node);
              if is_max_node (current_node)
              then
              {
```

```
                alpha = max (alpha, alpha_beta (children, alpha, beta));
                if alpha >= beta
                then cut_off_search_below (current_node);
        }
        if is_min_node (current_node)
        then
        {
                beta = min (beta, alpha_beta (children, alpha, beta));
                if beta <= alpha
                then cut_off_search_below (current_node);
        }
}
```

To see how alpha–beta pruning works in practice, let us examine the game tree shown in Figure 6.5.

The non-leaf nodes in the tree are labeled from a to g, and the leaf nodes have scores assigned to them by static evaluation: a is a max node; b and c are min nodes; and d, e, f, and g are max nodes.

Following the tree by depth-first search, the first step is to follow the path a, b, d and then to the three children of d. This gives an alpha value for d of 3. This is passed up to b, which now has a beta value of 3, and an alpha value that has been passed down from a of negative infinity.

Now, the first child of e is examined and has a score of 4. In this case, clearly there is no need to examine the other children of e because the minimizing choice at node b will definitely do worse by choosing e rather than d. So cut- off is applied here, and the nodes with scores of 5 and 7 are never examined.

The full analysis of the tree is shown in Table 6.1, which shows how the scores move through the tree from step to step of the process.

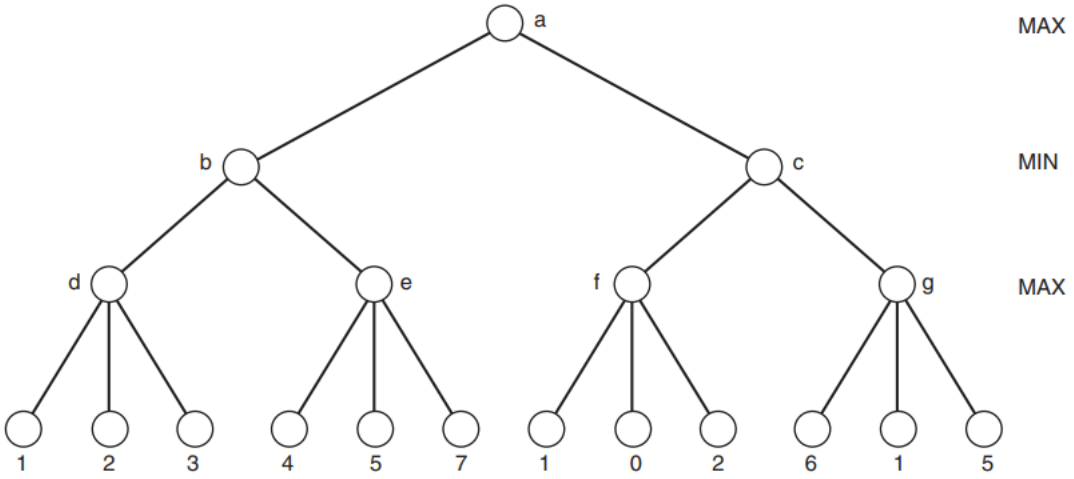

Figure 6.5
A simple game tree

TABLE 6.1 Analysis of alpha–beta pruning for the game tree in Figure 6.5

Step	Node	Alpha	Beta	Notes
1	a	$-\infty$	∞	Alpha starts as $-\infty$ and beta starts as ∞.
2	b	$-\infty$	∞	
3	d	$-\infty$	∞	
4	d	1	∞	
5	d	2	∞	
6	d	3	∞	At this stage, we have examined the three children of d and have obtained an alpha value of 3, which is passed back up to node b.
7	b	$-\infty$	3	At this min node, we can clearly achieve a score of 3 or better (lower). Now we need to examine the children of e to see if we can get a lower score.
8	e	$-\infty$	3	
9	e	4	3	CUT-OFF. A score of 4 can be obtained from the first child of e. Min clearly will do better to choose d rather than e because if he chooses e, max can get at least 4, which is worse for min than 3. Hence, we can now ignore the other children of e.
10	a	3	∞	The value of 3 has been passed back up to the root node, a. Hence, max now knows that he can score at least 3. He now needs to see if he can do better.
11	c	3	∞	
12	f	3	∞	We now examine the three children of f and find that none of them is better than 3. So, we pass back a value of 3 to c.
13	c	3	3	CUT-OFF. Max has already found that by taking the left-hand branch, he can achieve a score of 3. Now it seems that if he chooses the right-hand branch, min can choose f, which will mean he can only achieve a score of 2. So cut-off can now occur because there is no need to examine g or its children.

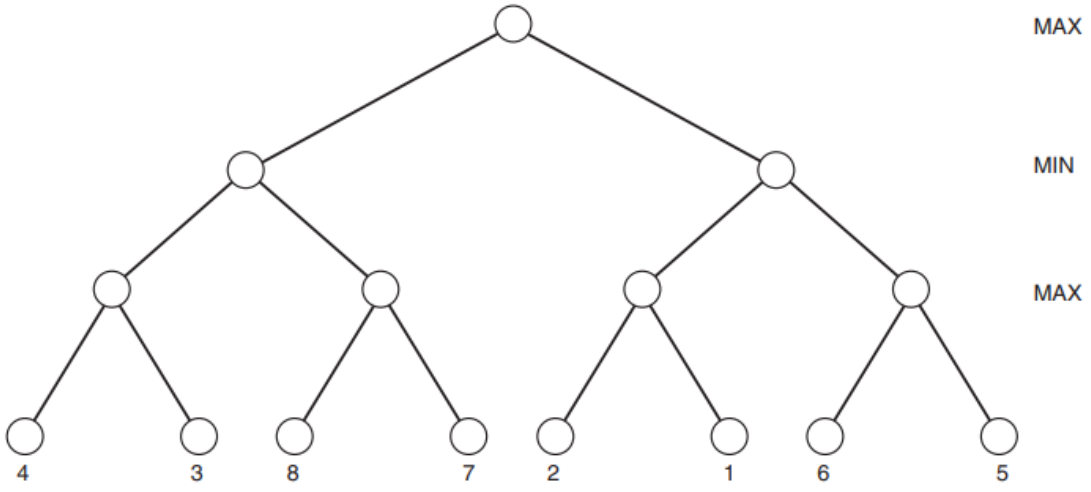

Figure 6.6

A game tree optimized for alpha–beta search

Hence, out of the 12 leaf nodes in the tree, the algorithm has needed to examine only 7 to conclude that the best move for max to make is b, in which case min will choose d and max will choose the right-hand node, ending with a static evaluation of 3.

At this stage, we can easily see that this is the right answer because if max chooses c, then min will clearly choose f, resulting in max being able to achieve a score of only 2.

An ideal tree for alpha–beta pruning is shown in Figure 6.6.

In this case, the Minimax algorithm with alpha–beta cut-off will need to examine only five of the eight leaf nodes.

What is Logic?

Logic is concerned with reasoning and the validity of arguments. In general, in logic, we are not concerned with the truth of statements, but rather with their **validity**. That is to say, although the following argument is clearly logical, it is not something that we would consider to be true:

> All lemons are blue
> Mary is a lemon
> Therefore, Mary is blue

This set of statements is considered to be **valid** because the conclusion (Mary is blue) follows logically from the other two statements, which we often call the **premises**.

The reason that validity and truth can be separated in this way is simple: a piece of a reasoning is considered to be valid if its conclusion is true in cases where its premises are also true. Hence, a valid set of statements such as the ones above can give a false conclusion, provided one or more of the premises are also false.

We can say: *a piece of reasoning is valid if it leads to a true conclusion in every situation where the premises are true.*

Logic is concerned with **truth values**. The possible truth values are true and false. These can be considered to be the fundamental units of logic, and almost all logic is ultimately concerned with these truth values.

Logic is widely used in computer science, and particularly in Artificial Intelligence. Logic is widely used as a representational method for Artificial Intelligence. Unlike some other representations (such as frames, which are described in detail in Chapter 3), logic allows us to easily reason about negatives (such as, “this book is not red”) and disjunctions (“or”—such as, “He’s either a soldier or a sailor”).

Why Logic is used in Artificial Intelligence

Logic is also often used as a representational method for communicating concepts and theories within the Artificial Intelligence community. In addition, logic is used to represent language in systems that are able to understand and analyze human language.

As we will see, one of the main weaknesses of traditional logic is its inability to deal with **uncertainty**. Logical statements must be expressed in terms of truth or falsehood—it is not possible to reason, in classical logic, about possibilities. We will see different versions of logic such as **modal logics** that provide some ability to reason about possibilities, and also probabilistic methods and fuzzy logic that provide much more rigorous ways to reason in uncertain situations.

Logical Operators

In reasoning about truth values, we need to use a number of **operators**, which can be applied to truth values. We are familiar with several of these operators from everyday language:

> I like apples **and** oranges.
> You can have an ice cream **or** a cake.
> **If** you come from France, **then** you speak French.
> I am **not** stupid!

Here we see the four most basic logical operators being used in everyday language. The operators are:

- And
- Or
- Not
- if . . . then . . . (usually called **implies**)

These operators work more or less as we expect them to. One important point to note is that or is slightly different from the way we usually use it. In the sentence, "You can have an icecream or a cake," the mother is usually suggesting to her child that he can only have one of the items, but not both. This is referred to as an **exclusive-or** in logic because the case where both are allowed is excluded. The version of or that is used in logic is called **inclusive-or** and allows the case with both options.

The operators are usually written using the following symbols, although other symbols are sometimes used, according to the context:

and	∧
or	∨
not	¬
implies	→
iff	↔

Iff is an abbreviation that is commonly used to mean "if and only if." We see later that this is a stronger form of implies that holds true if one thing implies another, and also the second thing implies the first.

For example, "you can have an ice-cream if and only if you eat your dinner." It may not be immediately apparent why this is different from "you can have an icecream if you eat your dinner." This is because most mothers really mean *iff* when they use *if* in this way.

Translating between English and Logical Notation

To use logic, it is first necessary to convert facts and rules about the real world into logical expressions using the logical operators described in Section 7.4. Without a reasonable amount of experience at this translation, it can seem quite a daunting task in some cases.

Let us examine some examples.

First, we will consider the simple operators, ∧, ∨, and ¬.

Sentences that use the word and in English to express more than one concept, all of which is true at once, can be easily translated into logic using the AND operator, ∧.

For example:

> "It is raining and it is Tuesday."

might be expressed as:

> $R \wedge T$

Where *R* means "it is raining" and *T* means "it is Tuesday." Note that we have been fairly arbitrary in our choice of these terms. This is all right, as long as the terms are chosen in such a way that they represent the problem adequately. For example, if it is not necessary to discuss where it is raining, *R* is probably enough. If we need to write expressions such as "it is raining in New York" or "it is raining heavily" or even "it rained for 30 minutes on Thursday," then *R* will probably not suffice.

To express more complex concepts like these, we usually use predicates. Hence, for example, we might translate "it is raining in New York" as:

> N(R)

We might equally well choose to write it as:

> R(N)

This depends on whether we consider the rain to be a property of New York, or vice versa. In other words, when we write *N(R)*, we are saying that a property of the rain is that it is in New York, whereas with *R(N)* we are saying that a property of New York is that it is raining.

Which we use depends on the problem we are solving. It is likely that if we are solving a problem about New York, we would use *R(N)*, whereas if we are solving a problem about the location of various types of weather, we might use *N(R)*.

Let us return now to the logical operators. The expression "it is raining in New York, and I'm either getting sick or just very tired" can be expressed as follows:

> $R(N) \wedge (S(I) \vee T(I))$

Here we have used both the $\wedge$ operator, and the $\vee$ operator to express a col- lection of statements. The statement can be broken down into two sections, which is indicated by the use of parentheses. The section in the parentheses is $S(I) \vee T(I)$, which means "I'm either getting sick OR I'm very tired". This expression is "AND'ed" with the part outside the parentheses, which is *R(N)*.

Finally, the $\neg$ operator is applied exactly as you would expect—to express negation. For example,

> It is not raining in New York,

might be expressed as

> $\neg R(N)$

It is important to get the $\neg$ in the right place. For example: "I'm either not well or just very tired" would be translated as

> $\neg W(I) \vee T(I)$

The position of the $\neg$ here indicates that it is bound to *W(I)* and does not play any role in affecting *T(I)*. This idea of **precedence** is explained further in Section 7.7.

Now let us see how the $\rightarrow$ operator is used. Often when dealing with logic we are discussing rules, which express concepts such as "if it is raining then I will get wet."

This sentence might be translated into logic as

$$R \rightarrow W(I)$$

This is read "R implies W(I)" or "IF R THEN W(I)". By replacing the symbols *R* and *W(I)* with their respective English language equivalents, we can see that this sentence can be read as

"raining implies I'll get wet"

or "IF it's raining THEN I'll get wet."

Implication can be used to express much more complex concepts than this. For example, "Whenever he eats sandwiches that have pickles in them, he ends up either asleep at his desk or singing loud songs" might be translated as

$$S(y) \wedge E(x, y) \wedge P(y) \rightarrow A(x) \vee (S(x, z) \wedge L(z))$$

Here we have used the following symbol translations:

S(y) means that *y* is a sandwich.
E(x, y) means that *x* (the man) eats *y* (the sandwich).
P(y) means that *y* (the sandwich) has pickles in it.
A(x) means that *x* ends up asleep at his desk.
S(x, z) means that *x* (the man) sings *z* (songs).
L(z) means that *z* (the songs) are loud.

In fact, there are better ways to express this kind of sentence, as we will see when we examine the quantifiers ∃ and ∀ in Section 7.13.

The important thing to realize is that the choice of variables and predicates is important, but that you can choose any variables and predicates that map well to your problem and that help you to solve the problem. For example, in the example we have just looked at, we could perfectly well have used instead

$$S \rightarrow A \vee L$$

where *S* means "he eats a sandwich which has pickles in it," A means "he ends up asleep at his desk," and *L* means "he sings loud songs."

The choice of granularity is important, but there is no right or wrong way to make this choice. In this simpler logical expression, we have chosen to express a simple relationship between three variables, which makes sense if those variables are all that we care about—in other words, we don't need to know anything else about the sandwich, or the songs, or the man, and the facts we examine are simply whether or not he eats a sandwich with pickles, sleeps at his desk, and sings loud songs. The first translation we gave is more appropriate if we need to examine these concepts in more detail and reason more deeply about the entities involved.

Note that we have thus far tended to use single letters to represent logical variables. It is also perfectly acceptable to use longer variable names, and thus to write expressions such as the following:

$$\text{Fish}(x) \wedge \text{living}(x) \rightarrow \text{has\_scales}(x)$$

This kind of notation is obviously more useful when writing logical expressions that are intended to be read by humans but when manipulated by a computer do not add any value.

The Deduction Theorem

A useful rule known as the deduction theorem provides us with a way to make propositional logic proofs easier. The rule is as follows:

$$\text{if } A \bigcup \{B\} \vdash C \text{ then } A \vdash (B \rightarrow C)$$

Here A is a set of wff's, which makes up our assumptions. Note that this rule is *true* even if A is the empty set. $A \cup \{B\}$ means the union of the set *A* with the set consisting of one element, *B*.

The rule also holds in reverse:

$$\text{if } A \vdash (B \rightarrow C) \text{ then } A \bigcup \{B\} \vdash C$$

Let us see an example of a proof using the deduction theorem.

Our aim is to prove the following:

$$\{A \rightarrow B\} \vdash A \rightarrow (C \rightarrow B)$$

Recall the axiom that was presented earlier:

$$A \rightarrow (B \rightarrow A)$$

Because propositional logic is monotonic (see Section 7.18), we can add in an additional assumption, that A is *true*:

$$A$$

Now, by applying modus ponens to this assumption and our hypothesis, $A \rightarrow B$, we arrive at

$$B$$

We can now apply our axiom

$$B \rightarrow (C \rightarrow B)$$

And by modus ponens on the above two lines, we get

$$C \rightarrow B$$

Hence, we have shown that

$$\{A \rightarrow B\} \bigcup A \vdash (C \rightarrow B)$$

And, therefore, by the deduction theorem

$$\{A \rightarrow B\} \vdash A \rightarrow (C \rightarrow B)$$

Soundness

We have seen that a logical system such as propositional logic consists of a syntax, a semantics, and a set of rules of deduction. A logical system also has a set of fundamental truths, which are known as axioms. The axioms are the basic rules that are known to be true and from which all other **theorems** within the system can be proved.

An axiom of propositional logic, for example, is

$$A \rightarrow (B \rightarrow A)$$

A theorem of a logical system is a statement that can be proved by applying the rules of deduction to the axioms in the system.

If *A* is a theorem, then we write

$$\vdash A$$

A logical system is described as being **sound** if every theorem is logically valid, or a tautology.

It can be proved by induction that both propositional logic and FOPL are sound.

Completeness

A logical system is **complete** if every tautology is a theorem—in other words, if every valid statement in the logic can be proved by applying the rules of deduction to the axioms. Both propositional logic and FOPL are complete. The proofs that these systems are complete are rather complex.

Decidability

A logical system is decidable if it is possible to produce an algorithm that will determine whether any wff is a theorem. In other words, if a logical system is decidable, then a computer can be used to determine whether logical expressions in that system are valid or not.

We can prove that propositional logic is decidable by using the fact that it is complete. Thanks to the completeness of propositional logic, we can prove that a wff *A* is a theorem by showing that it is a tautology. To show if a wff is a tautology, we simply need to draw up a truth table for that wff and show that all the lines have *true* as the result. This can clearly be done algorithmically because we know that a truth table for *n* values has 2^n lines and is therefore finite, for a finite number of variables.

FOPL, on the other hand, is not decidable. This is due to the fact that it is not possible to develop an algorithm that will determine whether an arbitrary wff in FOPL is logically valid.

Monotonicity

A logical system is described as being monotonic if a valid proof in the system cannot be made invalid by adding additional premises or assumptions. In other words, if we find that we can prove a conclusion *C* by applying rules of deduction to a premise *B* with assumptions *A*, then adding additional assumptions *A′* and *B′* will not stop us from being able to deduce *C*.

Both propositional logic and FOPL are monotonic. Elsewhere in this book, we learn about probability theory, which is not a monotonic system.

Monotonicity of a logical system can be expressed as follows:

> If we can prove $\{A, B\} \vdash C$,
> then we can also prove: $\{A, B, A', B'\} \vdash C$.

Note that *A′* and *B′* can be anything, including $\neg A$ and $\neg B$. In other words, even adding contradictory assumptions does not stop us from making the proof in a monotonic system. In fact, it turns out that adding contradictory assumptions allows us to prove *anything*, including invalid conclusions.

This makes sense if we recall the line in the truth table for $\rightarrow$, which shows that *false* $\rightarrow$ *true*. By adding a contradictory assumption, we make our assumptions *false* and can thus prove any conclusion.

Abduction and Inductive Reasoning

The kind of reasoning that we have seen so far in this chapter has been **deductive reasoning**, which in general is based on the use of modus ponens and the other deductive rules of reasoning. This kind of reasoning assumes that we are dealing with certainties and does not allow us to reason about things of which we are not certain. As we see elsewhere in this book, there is another kind of reasoning, **inductive reasoning**, which does not have the same logical basis but can be extremely powerful for dealing with situations in which we lack certainty.

Strangely, another form of reasoning, **abduction**, is based on a common fallacy, which can be expressed as

$$\frac{B \qquad A \rightarrow B}{A}$$

Note that abduction is very similar to modus ponens but is not logically sound. A typical example of using this rule might be “When Jack is sick, he doesn’t come to work. Jack is not at work today. Therefore, Jack is sick.”

In fact, Jack may be having a holiday, or attending a funeral, or it may be Sunday or Christmas Day.

Given that this type of reasoning is invalid, why are we discussing it here? It turns out that although abduction does not provide a logically sound model for reasoning, it does provide a model that works reasonably well in the real world because it allows us to observe a phenomenon and propose a possible explanation or cause for that phenomenon without complete knowledge.

Inductive reasoning enables us to make predictions about what will hap- pen, based on what has happened in the past. Humans use inductive reasoning all the time without realizing it. In fact, our entire lives are based around inductive reasoning, for example, “the sun came up yesterday and the day before, and every day I know about before that, so it will come up again tomorrow.” It’s possible it won’t, but it seems fairly unlikely. This kind of reasoning becomes more powerful when we apply probabilities to it, as in “I’ve noticed that nearly every bird I see is a swallow. Therefore, it’s quite likely that that bird is a swallow.”

As we will see, these kinds of reasoning are extremely useful for dealing with uncertainty and are the basis of most of the learning techniques used in Artificial Intelligence.

Modal Logics and Possible Worlds

The forms of logic that we have dealt with so far deal with facts and properties of objects that are either true or false. In these **classical logics**, we do not consider the possibility that things change or that things might not always be as they are now.

Modal logics are an extension of classical logic that allow us to reason about possibilities and certainties. In other words, using a modal logic, we can express ideas such as "although the sky is usually blue, it isn't always" (for example, at night). In this way, we can reason about possible worlds. A possible world is a universe or scenario that could logically come about.

The following statements may not be true in our world, but they are possible, in the sense that they are not illogical, and could be true in a possible world:

Trees are all blue.
Dogs can fly.
People have no legs.

It is possible that some of these statements will become true in the future, or even that they were true in the past. It is also possible to imagine an alternative universe in which these statements are true now. The following statements, on the other hand, cannot be true in any possible world:

$A \wedge \neg A$

$(x > y) \wedge (y > z) \wedge (z > x)$

The first of these illustrates the **law of the excluded middle**, which simply states that a fact must be either true or false: it cannot be both true and false. It also cannot be the case that a fact is neither true nor false. This is a law of classical logic, and as we see in Chapter 18, it is possible to have a logical system without the law of the excluded middle, and in which a fact can be both true *and* false.

The second statement cannot be true by the laws of mathematics. We are not interested in possible worlds in which the laws of logic and mathematics do not hold.

A statement that may be true or false, depending on the situation, is called **contingent**. A statement that must always have the same truth value, regardless of which possible world we consider, is **noncontingent**. Hence, the following statements are contingent:

$A \wedge B$
$A \vee B$

I like ice cream.
The sky is blue.

The following statements are noncontingent:

$A \vee \neg A$
$A \wedge \neg A$

If you like all ice cream, then you like this ice cream.

Clearly, a noncontingent statement can be either true or false, but the fact that it is noncontingent means it will always have that same truth value.

If a statement A is contingent, then we say that *A* is **possibly** true, which is written $\Diamond A$

If A is noncontingent, then it is **necessarily** true, which is written $\Box A$

Reasoning in Modal Logic

It is not possible to draw up a truth table for the operators $\Diamond$ and $\Box$. (Consider the four possible truth tables for unary operators—it should be clear that none of these matches these operators.) It is possible, however, to reason about them.

The following rules are examples of the axioms that can be used to reason in this kind of modal logic:

$\Box A \rightarrow \Diamond A$

$\Box \neg A \rightarrow \neg \Diamond A$

$\Diamond A \rightarrow \neg \Box A$

Although truth tables cannot be drawn up to prove these rules, you should be able to reason about them using your understanding of the meaning of the $\Box$ and $\Diamond$ operators.

Modal logic, and other nonclassical logics, are discussed in Chapter 17.

Dealing with Change

As we have seen, classical logics do not deal well with change. They assume that if an object has a property, then it will always have that property and always has had it. Of course, this is not true of very many things in the real world, and a logical system that allows things to change is needed. The situation and event calculi are covered in more detail in Chapters 17 and 19.

Chapter Summary

- ✓ Logic is primarily concerned with the logical validity of statements, rather than with truth.
- ✓ Logic is widely used in Artificial Intelligence as a representational method.
- ✓ Abduction and inductive reasoning are good at dealing with uncertainty, unlike classical logic.
- ✓ The main operators of propositional logic are $\wedge$, $\vee$, $\neg$, $\rightarrow$, and $\leftrightarrow$ (and, or, not, implies, and iff).
- ✓ The behavior of these logical operators can be expressed in truth tables. Truth tables can also be used to solve complex problems.
- ✓ Propositional logic deals with simple propositions such as "I like cheese." First-order predicate logic allows us to reason about more complex statements such as "All people who eat cheese like cats," using the quantifiers $\forall$ and $\exists$ ("for all", and "there exists").
- ✓ A statement that is always true in any situation is called a tautology.
  $A \vee \neg A$ is an example of a tautology.
- ✓ Two statements are logically equivalent if they have the same truth tables.
- ✓ First-order predicate logic is sound and complete, but not decidable. Propositional logic is sound, complete, and decidable.
- ✓ Modal logics allow us to reason about certainty.