6 ADVERSARIAL SEARCH

In which we examine the problems that arise when we try to plan ahead in a world where other agents are planning against us.

6.1 GAMES

Chapter 2 introduced **multiagent environments**, in which any given agent will need to consider the actions of other agents and how they affect its own welfare. The unpredictability of these other agents can introduce many possible **contingencies** into the agent's problem-solving process, as discussed in Chapter 3. The distinction between **cooperative** and **competitive** multiagent environments was also introduced in Chapter 2. Competitive environments, in which the agents' goals are in conflict, give rise to **adversarial search** problems—often known as **games**.

GAMES

Mathematical **game theory**, a branch of economics, views any multiagent environment as a game provided that the impact of each agent on the others is "significant," regardless of whether the agents are cooperative or competitive.[1] In AI, "games" are usually of a rather specialized kind—what game theorists call deterministic, turn-taking, two-player, **zero-sum games** of **perfect information**. In our terminology, this means deterministic, fully observable environments in which there are two agents whose actions must alternate and in which the utility values at the end of the game are always equal and opposite. For example, if one player wins a game of chess (+1), the other player necessarily loses (–1). It is this opposition between the agents' utility functions that makes the situation adversarial. We will consider multiplayer games, non-zero-sum games, and stochastic games briefly in this chapter, but will delay discussion of game theory proper until Chapter 17.

ZERO-SUM GAMES

PERFECT INFORMATION

Games have engaged the intellectual faculties of humans—sometimes to an alarming degree—for as long as civilization has existed. For AI researchers, the abstract nature of games makes them an appealing subject for study. The state of a game is easy to represent, and agents are usually restricted to a small number of actions whose outcomes are defined by

[1] Environments with very many agents are best viewed as **economies** rather than games.

precise rules. Physical games, such as croquet and ice hockey, have much more complicated descriptions, a much larger range of possible actions, and rather imprecise rules defining the legality of actions. With the exception of robot soccer, these physical games have not attracted much interest in the AI community.

Game playing was one of the first tasks undertaken in AI. By 1950, almost as soon as computers became programmable, chess had been tackled by Konrad Zuse (the inventor of the first programmable computer and the first programming language), by Claude Shannon (the inventor of information theory), by Norbert Wiener (the creator of modern control theory), and by Alan Turing. Since then, there has been steady progress in the standard of play, to the point that machines have surpassed humans in checkers and Othello, have defeated human champions (although not every time) in chess and backgammon, and are competitive in many other games. The main exception is Go, in which computers perform at the amateur level.

Games, unlike most of the toy problems studied in Chapter 3, are interesting *because* they are too hard to solve. For example, chess has an average branching factor of about 35, and games often go to 50 moves by each player, so the search tree has about 35^{100} or 10^{154} nodes (although the search graph has "only" about 10^{40} distinct nodes). Games, like the real world, therefore require the ability to make *some* decision even when calculating the *optimal* decision is infeasible. Games also penalize inefficiency severely. Whereas an implementation of A* search that is half as efficient will simply cost twice as much to run to completion, a chess program that is half as efficient in using its available time probably will be beaten into the ground, other things being equal. Game-playing research has therefore spawned a number of interesting ideas on how to make the best possible use of time.

We begin with a definition of the optimal move and an algorithm for finding it. We then look at techniques for choosing a good move when time is limited. **Pruning** allows us to ignore portions of the search tree that make no difference to the final choice, and heuristic **evaluation functions** allow us to approximate the true utility of a state without doing a complete search. Section 6.5 discusses games such as backgammon that include an element of chance; we also discuss bridge, which includes elements of **imperfect information** because not all cards are visible to each player. Finally, we look at how state-of-the-art game-playing programs fare against human opposition and at directions for future developments.

IMPERFECT INFORMATION

6.2 Optimal Decisions in Games

We will consider games with two players, whom we will call MAX and MIN for reasons that will soon become obvious. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A game can be formally defined as a kind of search problem with the following components:

- The **initial state**, which includes the board position and identifies the player to move.
- A **successor function**, which returns a list of (*move, state*) pairs, each indicating a legal move and the resulting state.

TERMINAL TEST

- A **terminal test**, which determines when the game is over. States where the game has ended are called **terminal states**.
- A **utility function** (also called an objective function or payoff function), which gives a numeric value for the terminal states. In chess, the outcome is a win, loss, or draw, with values $+1$, -1, or 0. Some games have a wider variety of possible outcomes; the payoffs in backgammon range from $+192$ to -192. This chapter deals mainly with zero-sum games, although we will briefly mention non-zero-sum games.

GAME TREE

The initial state and the legal moves for each side define the **game tree** for the game. Figure 6.1 shows part of the game tree for tic-tac-toe (noughts and crosses). From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled. The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names). It is MAX's job to use the search tree (particularly the utility of terminal states) to determine the best move.

Optimal strategies

In a normal search problem, the optimal solution would be a sequence of moves leading to a goal state—a terminal state that is a win. In a game, on the other hand, MIN has something

STRATEGY

to say about it. MAX therefore must find a contingent **strategy**, which specifies MAX's move in the initial state, then MAX's moves in the states resulting from every possible response by MIN, then MAX's moves in the states resulting from every possible response by MIN to *those* moves, and so on. Roughly speaking, an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent. We will begin by showing how to find this optimal strategy, even though it should be infeasible for MAX to compute it for games more complex than tic-tac-toe.

Even a simple game like tic-tac-toe is too complex for us to draw the entire game tree, so we will switch to the trivial game in Figure 6.2. The possible moves for MAX at the root node are labeled a_1, a_2, and a_3. The possible replies to a_1 for MIN are b_1, b_2, b_3, and so on. This particular game ends after one move each by MAX and MIN. (In game parlance, we say

PLY

that this tree is one move deep, consisting of two half-moves, each of which is called a **ply**.) The utilities of the terminal states in this game range from 2 to 14.

Given a game tree, the optimal strategy can be determined by examining the **minimax**

MINIMAX VALUE

value of each node, which we write as MINIMAX-VALUE(n). The minimax value of a node is the utility (for MAX) of being in the corresponding state, *assuming that both players play optimally* from there to the end of the game. Obviously, the minimax value of a terminal state is just its utility. Furthermore, given a choice, MAX will prefer to move to a state of maximum value, whereas MIN prefers a state of minimum value. So we have the following:

$$\text{MINIMAX-VALUE}(n) = \begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\ \max_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MIN node.} \end{cases}$$

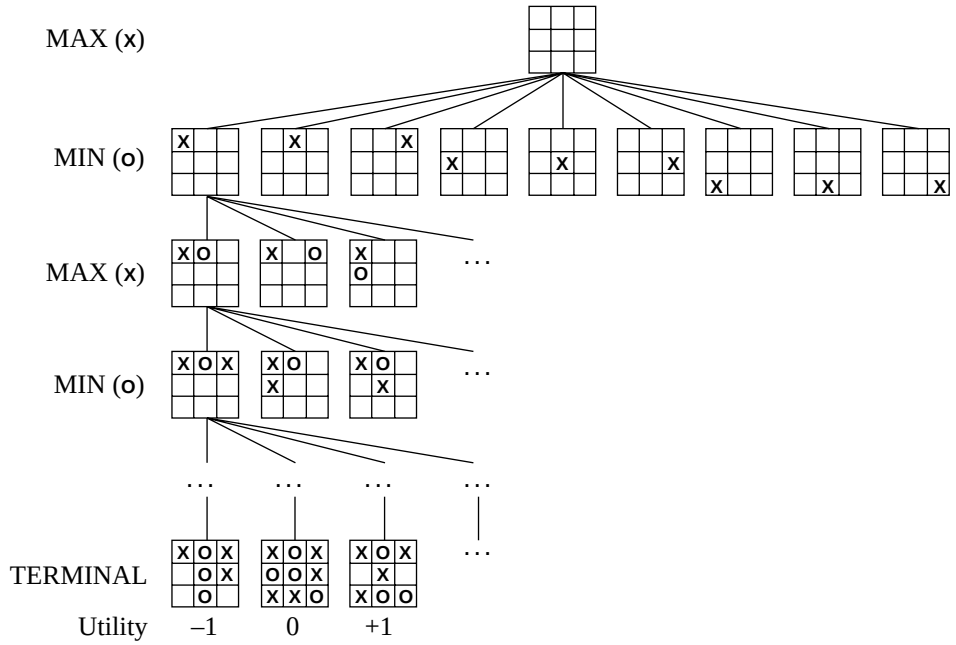


Figure 6.1 A (partial) search tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the search tree, giving alternating moves by MIN (O) and MAX, until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

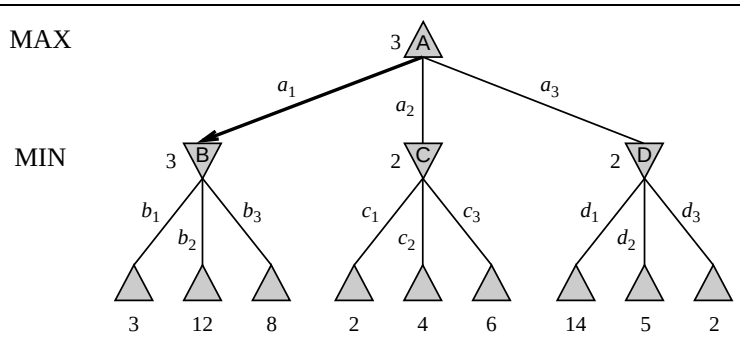


Figure 6.2 A two-ply game tree. The $\triangle$ nodes are "MAX nodes," in which it is MAX's turn to move, and the $\triangledown$ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the successor with the highest minimax value, and MIN's best reply is b_1, because it leads to the successor with the lowest minimax value.

Let us apply these definitions to the game tree in Figure 6.2. The terminal nodes on the bottom level are already labeled with their utility values. The first MIN node, labeled B, has three successors with values 3, 12, and 8, so its minimax value is 3. Similarly, the other two MIN nodes have minimax value 2. The root node is a MAX node; its successors have minimax

MINIMAX DECISION

values 3, 2, and 2; so it has a minimax value of 3. We can also identify the **minimax decision** at the root: action a_1 is the optimal choice for MAX because it leads to the successor with the highest minimax value.

This definition of optimal play for MAX assumes that MIN also plays optimally—it maximizes the *worst-case* outcome for MAX. What if MIN does not play optimally? Then it is easy to show (Exercise 6.2) that MAX will do even better. There may be other strategies against suboptimal opponents that do better than the minimax strategy; but these strategies necessarily do worse against optimal opponents.

The minimax algorithm

MINIMAX ALGORITHM

The **minimax algorithm** (Figure 6.3) computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves

BACKED UP

of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds. For example, in Figure 6.2, the algorithm first recurses down to the three bottom-left nodes, and uses the UTILITY function on them to discover that their values are 3, 12, and 8 respectively. Then it takes the minimum of these values, 3, and returns it as the backed-up value of node B. A similar process gives the backed up values of 2 for C and 2 for D. Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 for the root node.

The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m, and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$. The space complexity is $O(bm)$ for an algorithm that generates all successors at once, or $O(m)$ for an algorithm that generates successors one at a time (see page 76). For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

Optimal decisions in multiplayer games

Many popular games allow more than two players. Let us examine how to obtain extend the minimax idea to multiplayer games. This is straightforward from the technical viewpoint, but raises some interesting new conceptual issues.

First, we need to replace the single value for each node with a *vector* of values. For example, in a three-player game with players A, B, and C, a vector $\langle v_A, v_B, v_C \rangle$ is associated with each node. For terminal states, this vector gives the utility of the state from each player's viewpoint. (In two-player, zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite.) The simplest way to implement this is to have the UTILITY function return a vector of utilities.

Now we have to consider nonterminal states. Consider the node marked X in the game tree shown in Figure 6.4. In that state, player C chooses what to do. The two choices lead to terminal states with utility vectors $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$ and $\langle v_A = 4, v_B = 2, v_C = 3 \rangle$. Since 6 is bigger than 3, C should choose the first move. This means that if state X is reached, subsequent play will lead to a terminal state with utilities $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$. Hence,

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game

  v ← MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v

---

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s))
  return v

---

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s))
  return v
```

Figure 6.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state.

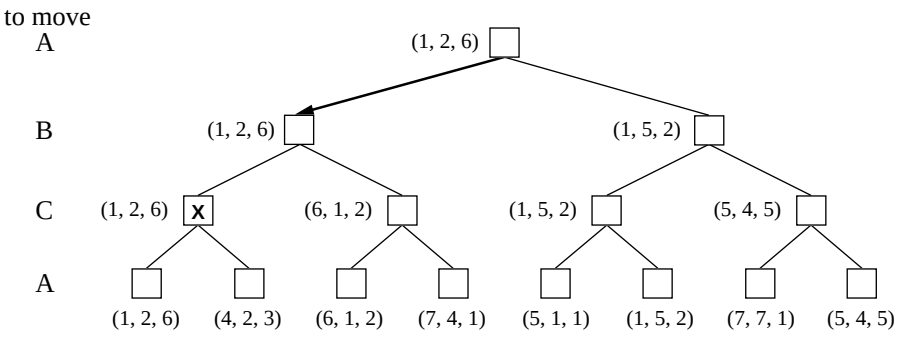


Figure 6.4 The first three ply of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

the backed-up value of X is this vector. In general, the backed-up value of a node n is the utility vector of whichever successor has the highest value for the player choosing at n.

Anyone who plays multiplayer games, such as Diplomacy$^{\mathrm{TM}}$, quickly becomes aware that there is a lot more going on than in two-player games. multiplayer games usually involve

ALLIANCES **alliances**, whether formal or informal, among the players. Alliances are made and broken

as the game proceeds. How are we to understand such behavior? Are alliances a natural consequence of optimal strategies for each player in a multiplayer game? It turns out that they can be. For example suppose A and B are in weak positions and C is in a stronger position. Then it is often optimal for both A and B to attack C rather than each other, lest C destroy each of them individually. In this way, collaboration emerges from purely selfish behavior. Of course, as soon as C weakens under the joint onslaught, the alliance loses its value, and either A or B could violate the agreement. In some cases, explicit alliances merely make concrete what would have happened anyway. In other cases there is a social stigma to breaking an alliance, so players must balance the immediate advantage of breaking an alliance against the long-term disadvantage of being perceived as untrustworthy. See Section 17.6 for more on these complications.

If the game is not zero-sum, then collaboration can also occur with just two players. Suppose, for example, that there is a terminal state with utilities $\langle v_A = 1000, v_B = 1000 \rangle$, and that 1000 is the highest possible utility for each player. Then the optimal strategy is for both players to do everything possible to reach this state—that is, the players will automatically cooperate to achieve a mutually desirable goal.

6.3 ALPHA–BETA PRUNING

The problem with minimax search is that the number of game states it has to examine is exponential in the number of moves. Unfortunately we can't eliminate the exponent, but we can effectively cut it in half. The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree. That is, we can borrow the idea of **pruning** from Chapter 4 in order to eliminate large parts of the tree from consideration.

ALPHA–BETA PRUNING

The particular technique we will examine is called **alpha–beta pruning**. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Consider again the two-ply game tree from Figure 6.2. Let's go through the calculation of the optimal decision once more, this time paying careful attention to what we know at each point in the process. The steps are explained in Figure 6.5. The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes.

Another way to look at this is as a simplification of the formula for MINIMAX-VALUE. Let the two unevaluated successors of node C in Figure 6.5 have values x and y and let z be the minimum of x and y. The value of the root node is given by

$$\begin{aligned}
\text{MINIMAX-VALUE}(\mathit{root}) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2,x,y), 2) \\
&= \max(3,z,2) \qquad \text{where } z \leq 2 \\
&= 3.
\end{aligned}$$

In other words, the value of the root and hence the minimax decision are *independent* of the values of the pruned leaves x and y.

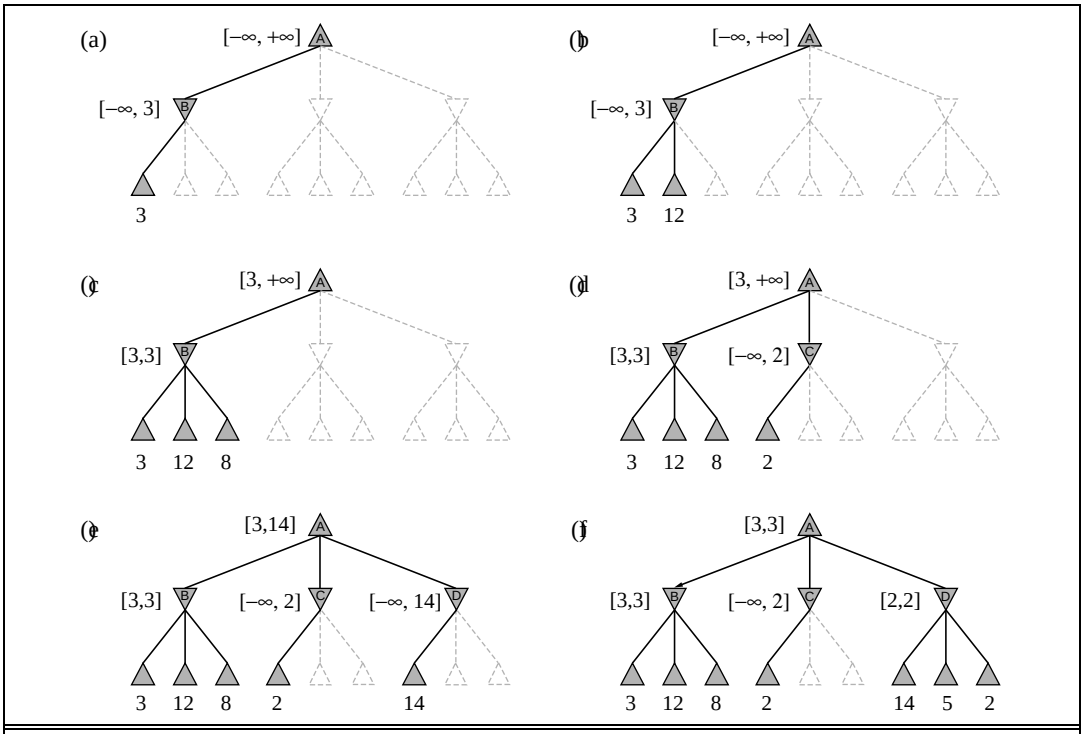


Figure 6.5 Stages in the calculation of the optimal decision for the game tree in Figure 6.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successors, so the value of B is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successors of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successors. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

Alpha–beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves. The general principle is this: consider a node n somewhere in the tree (see Figure 6.6), such that Player has a choice of moving to that node. If Player has a better choice m either at the parent node of n or at any choice point further up, then n *will never be reached in actual play.* So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

Remember that minimax search is depth-first, so at any one time we just have to consider the nodes along a single path in the tree. Alpha–beta pruning gets its name from the

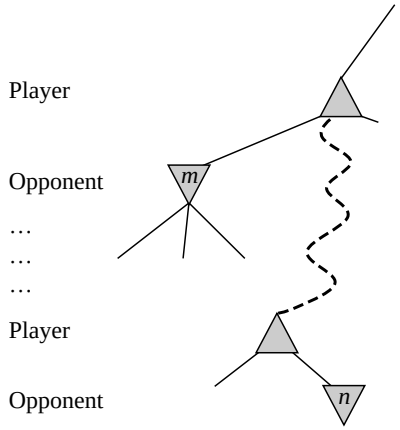


Figure 6.6 Alpha–beta pruning: the general case. If m is better than n for Player, we will never get to n in play.

following two parameters that describe bounds on the backed-up values that appear anywhere along the path:

$\alpha =$ the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.

$\beta =$ the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

Alpha–beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively. The complete algorithm is given in Figure 6.7. We encourage the reader to trace its behavior when applied to the tree in Figure 6.5.

The effectiveness of alpha–beta pruning is highly dependent on the order in which the successors are examined. For example, in Figure 6.5(e) and (f), we could not prune any successors of D at all because the worst successors (from the point of view of MIN) were generated first. If the third successor had been generated first, we would have been able to prune the other two. This suggests that it might be worthwhile to try to examine first the successors that are likely to be best.

If we assume that this can be done,[2] then it turns out that alpha–beta needs to examine only $O(b^{d/2})$ nodes to pick the best move, instead of $O(b^d)$ for minimax. This means that the effective branching factor becomes $\sqrt{b}$ instead of b—for chess, 6 instead of 35. Put another way, alpha–beta can look ahead roughly twice as far as minimax in the same amount of time. If successors are examined in random order rather than best-first, the total number of nodes examined will be roughly $O(b^{3d/4})$ for moderate b. For chess, a fairly simple ordering function (such as trying captures first, then threats, then forward moves, and then backward moves) gets you to within about a factor of 2 of the best-case $O(b^{d/2})$ result. Adding dynamic

[2] Obviously, it cannot be done perfectly; otherwise the ordering function could be used to play a perfect game!

```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game

  v ← MAX-VALUE(state, −∞, +∞)
  return the action in SUCCESSORS(state) with value v

---

function MAX-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

---

function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 6.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX routines in Figure 6.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

move-ordering schemes, such as trying first the moves that were found to be best last time, brings us quite close to the theoretical limit.

In Chapter 3, we noted that repeated states in the search tree can cause an exponential increase in search cost. In games, repeated states occur frequently because of **transpositions**—different permutations of the move sequence that end up in the same position. For example, if White has one move a_1 that can be answered by Black with b_1 and an unrelated move a_2 on the other side of the board that can be answered by b_2, then the sequences $[a_1, b_1, a_2, b_2]$ and $[a_1, b_2, a_2, b_1]$ both end up in the same position (as do the permutations beginning with a_2). It is worthwhile to store the evaluation of this position in a hash table the first time it is encountered, so that we don't have to recompute it on subsequent occurrences.

TRANSPOSITIONS

TRANSPOSITION TABLE

The hash table of previously seen positions is traditionally called a **transposition table**; it is essentially identical to the *closed* list in GRAPH-SEARCH (page 83). Using a transposition table can have a dramatic effect, sometimes as much as doubling the reachable search depth in chess. On the other hand, if we are evaluating a million nodes per second, it is not practical to keep *all* of them in the transposition table. Various strategies have been used to choose the most valuable ones.

6.4 IMPERFECT, REAL-TIME DECISIONS

The minimax algorithm generates the entire game search space, whereas the alpha–beta algorithm allows us to prune large parts of it. However, alpha–beta still has to search all the way to terminal states for at least a portion of the search space. This depth is usually not practical, because moves must be made in a reasonable amount of time—typically a few minutes at most. Shannon's 1950 paper, *Programming a computer for playing chess*, proposed instead that programs should cut off the search earlier and apply a heuristic **evaluation function** to states in the search, effectively turning nonterminal nodes into terminal leaves. In other words, the suggestion is to alter minimax or alpha–beta in two ways: the utility function is replaced by a heuristic evaluation function EVAL, which gives an estimate of the position's utility, and the terminal test is replaced by a **cutoff test** that decides when to apply EVAL.

CUTOFF TEST

Evaluation functions

An evaluation function returns an *estimate* of the expected utility of the game from a given position, just as the heuristic functions of Chapter 4 return an estimate of the distance to the goal. The idea of an estimator was not new when Shannon proposed it. For centuries, chess players (and aficionados of other games) have developed ways of judging the value of a position, because humans are even more limited in the amount of search they can do than are computer programs. It should be clear that the performance of a game-playing program is dependent on the quality of its evaluation function. An inaccurate evaluation function will guide an agent toward positions that turn out to be lost. How exactly do we design good evaluation functions?

First, the evaluation function should order the *terminal* states in the same way as the true utility function; otherwise, an agent using it might select suboptimal moves even if it can see ahead all the way to the end of the game. Second, the computation must not take too long! (The evaluation function could call MINIMAX-DECISION as a subroutine and calculate the exact value of the position, but that would defeat the whole purpose: to save time.) Third, for nonterminal states, the evaluation function should be strongly correlated with the actual chances of winning.

One might well wonder about the phrase "chances of winning." After all, chess is not a game of chance: we know the current state with certainty, and there are no dice involved. But if the search must be cut off at nonterminal states, then the algorithm will necessarily be *uncertain* about the final outcomes of those states. This type of uncertainty is induced by

computational, rather than informational, limitations. Given the limited amount of computation that the evaluation function is allowed to do for a given state, the best it can do is make a guess about the final outcome.

FEATURES

Let us make this idea more concrete. Most evaluation functions work by calculating various **features** of the state—for example, the number of pawns possessed by each side in a game of chess. The features, taken together, define various *categories* or *equivalence classes* of states: the states in each category have the same values for all the features. Any given category, generally speaking, will contain some states that lead to wins, some that lead to draws, and some that lead to losses. The evaluation function cannot know which states are which, but it can return a single value that reflects the *proportion* of states with each outcome. For example, suppose our experience suggests that 72% of the states encountered in the category lead to a win (utility +1); 20% to a loss (–1), and 8% to a draw (0). Then a reasonable evaluation for states in the category is the weighted average or **expected value**: $(0.72 \times +1) + (0.20 \times -1) + (0.08 \times 0) = 0.52$. In principle, the expected value can be determined for each category, resulting in an evaluation function that works for any state. As with terminal states, the evaluation function need not return actual expected values, as long as the *ordering* of the states is the same.

EXPECTED VALUE

In practice, this kind of analysis requires too many categories and hence too much experience to estimate all the probabilities of winning. Instead, most evaluation functions compute separate numerical contributions from each feature and then *combine* them to find the total value. For example, introductory chess books give an approximate **material value** for each piece: each pawn is worth 1, a knight or bishop is worth 3, a rook 5, and the queen 9. Other features such as "good pawn structure" and "king safety" might be worth half a pawn, say. These feature values are then simply added up to obtain the evaluation of the position. A secure advantage equivalent to a pawn gives a substantial likelihood of winning, and a secure advantage equivalent to three pawns should give almost certain victory, as illustrated in Figure 6.8(a). Mathematically, this kind of evaluation function is called a **weighted linear function**, because it can be expressed as

MATERIAL VALUE

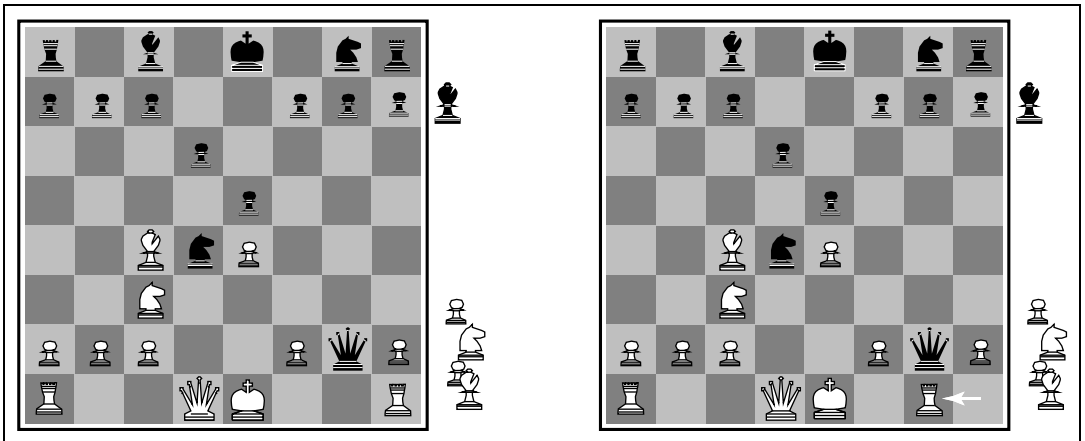
WEIGHTED LINEAR FUNCTION

$$\textsc{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s) \ ,$$

where each w_i is a weight and each f_i is a feature of the position. For chess, the f_i could be the numbers of each kind of piece on the board, and the w_i could be the values of the pieces (1 for pawn, 3 for bishop, etc.).

Adding up the values of features seems like a reasonable thing to do, but in fact it involves a very strong assumption: that the contribution of each feature is *independent* of the values of the other features. For example, assigning the value 3 to a bishop ignores the fact that bishops are more powerful in the endgame, when they have a lot of space to maneuver. For this reason, current programs for chess and other games also use *nonlinear* combinations of features. For example, a pair of bishops might be worth slightly more than twice the value of a single bishop, and a bishop is worth more in the endgame than at the beginning.

The astute reader will have noticed that the features and weights are *not* part of the rules of chess! They come from centuries of human chess-playing experience. Given the



(a) White to move (b) White to move

Figure 6.8 Two slightly different chess positions. In (a), black has an advantage of a knight and two pawns and will win the game. In (b), black will lose after white captures the queen.

linear form of the evaluation, the features and weights result in the best approximation to the true ordering of states by value. In particular, experience suggests that a secure material advantage of more than one point will probably win the game, all other things being equal; a three-point advantage is sufficient for near-certain victory. In games where this kind of experience is not available, the weights of the evaluation function can be estimated by the machine learning techniques of Chapter 18. Reassuringly, applying these techniques to chess has confirmed that a bishop is indeed worth about three pawns.

Cutting off search

The next step is to modify ALPHA-BETA-SEARCH so that it will call the heuristic EVAL function when it is appropriate to cut off the search. In terms of implementation, we replace the two lines in Figure 6.7 that mention TERMINAL-TEST with the following line:

if CUTOFF-TEST($state$, $depth$) **then return** EVAL($state$)

We also must arrange for some bookkeeping so that the current $depth$ is incremented on each recursive call. The most straightforward approach to controlling the amount of search is to set a fixed depth limit, so that CUTOFF-TEST($state$, $depth$) returns $true$ for all $depth$ greater than some fixed depth d. (It must also return $true$ for all terminal states, just as TERMINAL-TEST did.) The depth d is chosen so that the amount of time used will not exceed what the rules of the game allow.

A more robust approach is to apply iterative deepening, as defined in Chapter 3. When time runs out, the program returns the move selected by the deepest completed search. However, these approaches can lead to errors due to the approximate nature of the evaluation function. Consider again the simple evaluation function for chess based on material advantage. Suppose the program searches to the depth limit, reaching the position in Figure 6.8(b),

where Black is ahead by a knight and two pawns. It would report this as the heuristic value of the state, thereby declaring that the state will likely lead to a win by Black. But White's next move captures Black's queen with no compensation. Hence, the position is really won for White, but this can be seen only by looking ahead one more ply.

QUIESCENCE

Obviously, a more sophisticated cutoff test is needed. The evaluation function should be applied only to positions that are **quiescent**—that is, unlikely to exhibit wild swings in value in the near future. In chess, for example, positions in which favorable captures can be made are not quiescent for an evaluation function that just counts material. Nonquiescent positions can be expanded further until quiescent positions are reached. This extra search is called a **quiescence search**; sometimes it is restricted to consider only certain types of moves, such as capture moves, that will quickly resolve the uncertainties in the position.

QUIESCENCE SEARCH

HORIZON EFFECT

The **horizon effect** is more difficult to eliminate. It arises when the program is facing a move by the opponent that causes serious damage and is ultimately unavoidable. Consider the chess game in Figure 6.9. Black is ahead in material, but if White can advance its pawn from the seventh row to the eighth, the pawn will become a queen and create an easy win for White. Black can forestall this outcome for 14 ply by checking White with the rook, but inevitably the pawn will become a queen. The problem with fixed-depth search is that it believes that these stalling moves have avoided the queening move—we say that the stalling moves push the inevitable queening move "over the search horizon" to a place where it cannot be detected.

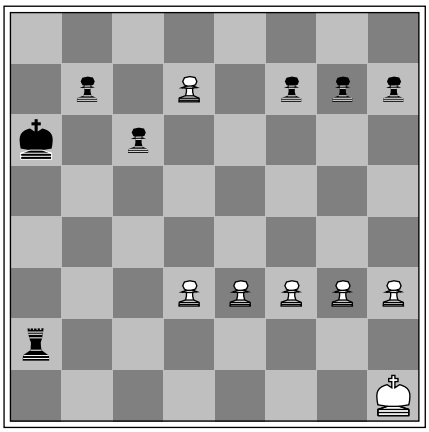
SINGULAR EXTENSIONS

As hardware improvements lead to deeper searches, one expects that the horizon effect will occur less frequently—very long delaying sequences are quite rare. The use of **singular extensions** has also been quite effective in avoiding the horizon effect without adding too much search cost. A singular extension is a move that is "clearly better" than all other moves in a given position. A singular-extension search can go beyond the normal depth limit without incurring much cost because its branching factor is 1. (Quiescence search can be thought of as a variant of singular extensions.) In Figure 6.9, a singular extension search will find the eventual queening move, provided that black's checking moves and white's king moves can be identified as "clearly better" than the alternatives.

FORWARD PRUNING

So far we have talked about cutting off search at a certain level and about doing alpha–beta pruning that provably has no effect on the result. It is also possible to do **forward pruning**, meaning that some moves at a given node are pruned immediately without further consideration. Clearly, most humans playing chess only consider a few moves from each position (at least consciously). Unfortunately, the approach is rather dangerous because there is no guarantee that the best move will not be pruned away. This can be disastrous if applied near the root, because every so often the program will miss some "obvious" moves. Forward pruning can be used safely in special situations—for example, when two moves are symmetric or otherwise equivalent, only one of them need be considered—or for nodes that are deep in the search tree.

Combining all the techniques described here results in a program that can play creditable chess (or other games). Let us assume we have implemented an evaluation function for chess, a reasonable cutoff test with a quiescence search, and a large transposition table. Let us also assume that, after months of tedious bit-bashing, we can generate and evaluate



Black to move

Figure 6.9 The horizon effect. A series of checks by the black rook forces the inevitable queening move by white "over the horizon" and makes this position look like a win for black, when it is really a win for white.

around a million nodes per second on the latest PC, allowing us to search roughly 200 million nodes per move under standard time controls (three minutes per move). The branching factor for chess is about 35, on average, and 35^5 is about 50 million, so if we used minimax search we could look ahead only about five plies. Though not incompetent, such a program can be fooled easily by an average human chess player, who can occasionally plan six or eight plies ahead. With alpha–beta search we get to about 10 ply, which results in an expert level of play. Section 6.7 describes additional pruning techniques that can extend the effective search depth to roughly 14 plies. To reach grandmaster status we would need an extensively tuned evaluation function and a large database of optimal opening and endgame moves. It wouldn't hurt to have a supercomputer to run the program on.

6.5 Games That Include an Element of Chance

In real life, there are many unpredictable external events that put us into unforeseen situations. Many games mirror this unpredictability by including a random element, such as the throwing of dice. In this way, they take us a step nearer reality, and it is worthwhile to see how this affects the decision-making process.

Backgammon is a typical game that combines luck and skill. Dice are rolled at the beginning of a player's turn to determine the legal moves. In the backgammon position of Figure 6.10, for example, white has rolled a 6–5, and has four possible moves.

Although White knows what his or her own legal moves are, White does not know what Black is going to roll and thus does not know what Black's legal moves will be. That means White cannot construct a standard game tree of the sort we saw in chess and tic-tac-toe. A

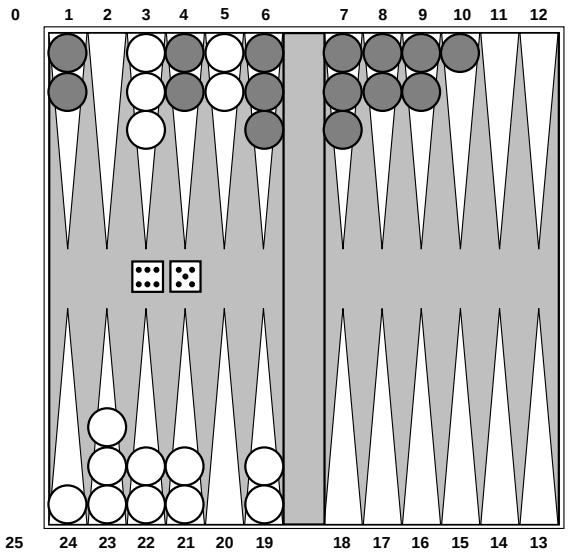


Figure 6.10 A typical backgammon position. The goal of the game is to move all one's pieces off the board. White moves clockwise toward 25, and black moves counterclockwise toward 0. A piece can move to any position unless there are multiple opponent pieces there; if there is one opponent, it is captured and must start over. In the position shown, White has rolled 6–5 and must choose among four legal moves: (5–10,5–11), (5–11,19–24), (5–10,10–16), and (5–11,11–16).

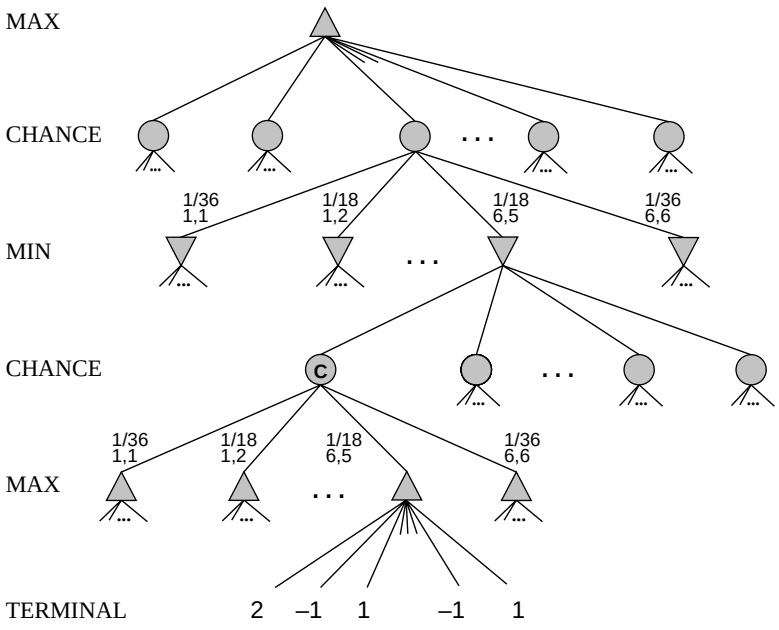


Figure 6.11 Schematic game tree for a backgammon position.

CHANCE NODES

game tree in backgammon must include **chance nodes** in addition to MAX and MIN nodes. Chance nodes are shown as circles in Figure 6.11. The branches leading from each chance node denote the possible dice rolls, and each is labeled with the roll and the chance that it will occur. There are 36 ways to roll two dice, each equally likely; but because a 6–5 is the same as a 5–6, there are only 21 distinct rolls. The six doubles (1–1 through 6–6) have a 1/36 chance of coming up, the other 15 distinct rolls a 1/18 chance each.

The next step is to understand how to make correct decisions. Obviously, we still want to pick the move that leads to the best position. However, the resulting positions do not have definite minimax values. Instead, we can only calculate the **expected value**, where the expectation is taken over all the possible dice rolls that could occur. This leads us to generalize the **minimax value** for deterministic games to an **expectiminimax value** for games with chance nodes. Terminal nodes and MAX and MIN nodes (for which the dice roll is known) work exactly the same way as before; chance nodes are evaluated by taking the weighted average of the values resulting from all possible dice rolls, that is,

EXPECTIMINIMAX VALUE

$$\text{EXPECTIMINIMAX}(n) = \begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\ \max_{s \in Successors(n)} \text{EXPECTIMINIMAX}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in Successors(n)} \text{EXPECTIMINIMAX}(s) & \text{if } n \text{ is a MIN node} \\ \sum_{s \in Successors(n)} P(s) \cdot \text{EXPECTIMINIMAX}(s) & \text{if } n \text{ is a chance node} \end{cases}$$

where the successor function for a chance node n simply augments the state of n with each possible dice roll to produce each successor s and $P(s)$ is the probability that that dice roll occurs. These equations can be backed up recursively all the way to the root of the tree, just as in minimax. We leave the details of the algorithm as an exercise.

Position evaluation in games with chance nodes

As with minimax, the obvious approximation to make with expectiminimax is to cut the search off at some point and apply an evaluation function to each leaf. One might think that evaluation functions for games such as backgammon should be just like evaluation functions for chess—they just need to give higher scores to better positions. But in fact, the presence of chance nodes means that one has to be more careful about what the evaluation values mean. Figure 6.12 shows what happens: with an evaluation function that assigns values [1, 2, 3, 4] to the leaves, move A_1 is best; with values [1, 20, 30, 400], move A_2 is best. Hence, the program behaves totally differently if we make a change in the scale of some evaluation values! It turns out that, to avoid this sensitivity, the evaluation function must be a *positive linear* transformation of the probability of winning from a position (or, more generally, of the expected utility of the position). This is an important and general property of situations in which uncertainty is involved, and we discuss it further in Chapter 16.

Complexity of expectiminimax

If the program knew in advance all the dice rolls that would occur for the rest of the game, solving a game with dice would be just like solving a game without dice, which minimax

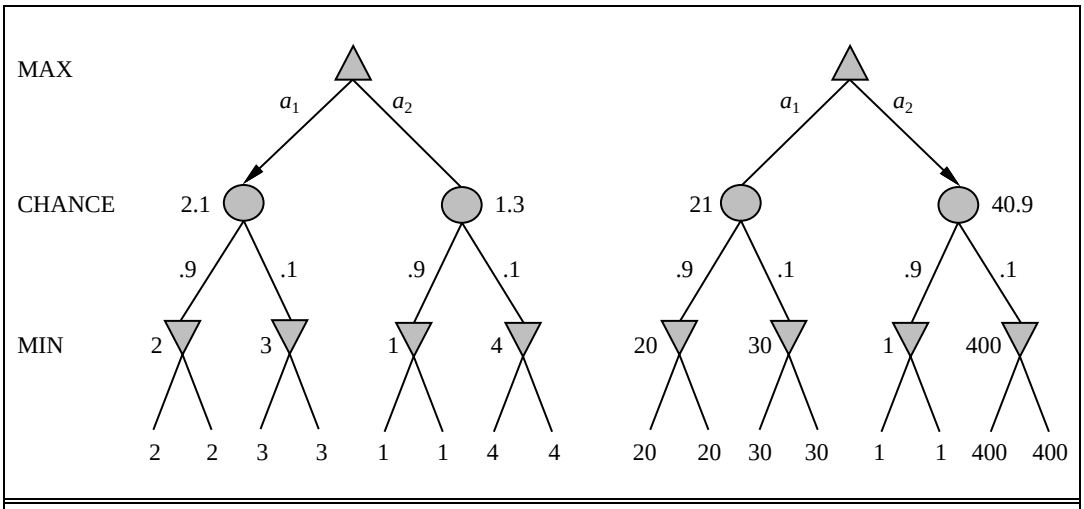


Figure 6.12 An order-preserving transformation on leaf values changes the best move.

does in $O(b^m)$ time. Because expectiminimax is also considering all the possible dice-roll sequences, it will take $O(b^m n^m)$, where n is the number of distinct rolls.

Even if the search depth is limited to some small depth d, the extra cost compared with that of minimax makes it unrealistic to consider looking ahead very far in most games of chance. In backgammon n is 21 and b is usually around 20, but in some situations can be as high as 4000 for dice rolls that are doubles. Three plies is probably all we could manage.

Another way to think about the problem is this: the advantage of alpha–beta is that it ignores future developments that just are not going to happen, given best play. Thus, it concentrates on likely occurrences. In games with dice, there are *no* likely sequences of moves, because for those moves to take place, the dice would first have to come out the right way to make them legal. This is a general problem whenever uncertainty enters the picture: the possibilities are multiplied enormously, and forming detailed plans of action becomes pointless, because the world probably will not play along.

No doubt it will have occurred to the reader that perhaps something like alpha–beta pruning could be applied to game trees with chance nodes. It turns out that it can. The analysis for MIN and MAX nodes is unchanged, but we can also prune chance nodes, using a bit of ingenuity. Consider the chance node C in Figure 6.11 and what happens to its value as we examine and evaluate its children. Is it possible to find an upper bound on the value of C before we have looked at all its children? (Recall that this is what alpha–beta needs to prune a node and its subtree.) At first sight, it might seem impossible, because the value of C is the *average* of its children's values. Until we have looked at all the dice rolls, this average could be anything, because the unexamined children might have any value at all. But if we put bounds on the possible values of the utility function, then we can arrive at bounds for the average. For example, if we say that all utility values are between $+3$ and -3, then the value of leaf nodes is bounded, and in turn we *can* place an upper bound on the value of a chance node without looking at all its children.

Card games

Card games are interesting for many reasons besides their connection with gambling. Among the huge variety of games, we will focus on those in which cards are dealt randomly at the beginning of the game, with each player receiving a hand of cards that is not visible to the other players. Such games include bridge, whist, hearts, and some forms of poker.

At first sight, it might seem that card games are just like dice games: the cards are dealt randomly and determine the moves available to each player, but all the dice are rolled at the beginning! We will pursue this observation further. It will turn out to be quite useful in practice. It is also quite wrong, for interesting reasons.

Imagine two players, MAX and MIN, playing some practice hands of four-card two handed bridge with all the cards showing. The hands are as follows, with MAX to play first:

MAX : $\heartsuit 6\ \diamondsuit 6\ \clubsuit 9\ 8$ MIN : $\heartsuit 4\ \spadesuit 2\ \clubsuit 10\ 5$.

Suppose that MAX leads the $\clubsuit 9$. MIN must now follow suit, playing either the $\clubsuit 10$ or the $\clubsuit 5$. MIN plays the $\clubsuit 10$ and wins the trick. MIN goes next and leads the $\spadesuit 2$. MAX has no spades (and so cannot win the trick) and therefore must throw away some card. The obvious choice is the $\diamondsuit 6$ because the other two remaining cards are winners. Now, whichever card MIN leads for the next trick, MAX will win both remaining tricks and the game will be tied at two tricks each. It is easy to show, using a suitable variant of minimax (Exercise 6.12), that MAX's lead of the $\clubsuit 9$ is in fact an optimal choice.

Now let's modify MIN's hand, replacing the $\heartsuit 4$ with the $\diamondsuit 4$:

MAX : $\heartsuit 6\ \diamondsuit 6\ \clubsuit 9\ 8$ MIN : $\diamondsuit 4\ \spadesuit 2\ \clubsuit 10\ 5$.

The two cases are entirely symmetric: play will be identical, except that on the second trick MAX will throw away the $\heartsuit 6$. Again, the game will be tied at two tricks each and the lead of the $\clubsuit 9$ is an optimal choice.

So far, so good. Now let's hide one of MIN's cards: MAX knows that MIN has either the first hand (with the $\heartsuit 4$) or the second hand (with the $\diamondsuit 4$), but has no idea which. MAX reasons as follows:

> The $\clubsuit 9$ is an optimal choice against MIN's first hand and against MIN's second hand, so it must be optimal now because I know that MIN has one of the two hands.

More generally, MAX is using what we might call "averaging over clairvoyancy." The idea is to evaluate a given course of action when there are unseen cards by first computing the minimax value of that action for each possible deal of the cards, and then computing the expected value over all deals using the probability of each deal.

If you think this is reasonable (or if you have no idea because you don't understand bridge), consider the following story:

> Day 1: Road *A* leads to a heap of gold pieces; Road *B* leads to a fork. Take the left fork and you'll find a mound of jewels, but take the right fork and you'll be run over by a bus.
> Day 2: Road *A* leads to a heap of gold pieces; Road *B* leads to a fork. Take the right fork and you'll find a mound of jewels, but take the left fork and you'll be run over by a bus.
> Day 3: Road *A* leads to a heap of gold pieces; Road *B* leads to a fork. Guess correctly and you'll find a mound of jewels, but guess incorrectly and you'll be run over by a bus.

Obviously, it's not unreasonable to take Road *B* on the first two days. No sane person, though, would take Road *B* on Day 3. Yet this is exactly what averaging over clairvoyancy suggests: Road *B* is optimal in the situations of Day 1 and Day 2; therefore it is optimal on Day 3, because one of the two previous situations must hold. Let us return to the card game: after MAX leads the ♣9, MIN wins with the ♣10. As before, MIN leads the ♠2, and now MAX is at the fork in the road without any instructions. If MAX throws away the ♡6 and MIN still has the ♡4, the ♡4 becomes a winner and MAX loses the game. Similarly, If MAX throws away the ♢6 and MIN still has the ♢4, MAX also loses. Therefore, playing the ♣9 first leads to a situation where MAX has a 50% chance of losing. (It would be much better to play the ♡6 and the ♢6 first, guaranteeing a tied game.)

The lesson to be drawn from all this is that when information is missing, one must consider *what information one will have* at each point in the game. The problem with MAX's algorithm is that it assumes that in each possible deal, play will proceed *as if all the cards are visible*. As our example shows, this leads MAX to act as if all *future* uncertainty will be resolved when the time comes. MAX's algorithm will also never decide to *gather* information (or *provide* information to a partner), because within each deal there's no need to do so; yet in games such as bridge, it is often a good idea to play a card that will help one discover things about one's opponent's cards or that will tell one's partner about one's own cards. These kinds of behaviors are generated automatically by an optimal algorithm for games of imperfect information. Such an algorithm searches not in the space of world states (hands of cards), but in the space of **belief states** (beliefs about who has which cards, with what probabilities). We will be able to explain the algorithm properly in Chapter 17, once we have developed the necessary probabilistic machinery. In that chapter, we will also expand on one final and very important point: in games of imperfect information, it's best to give away as little information to the opponent as possible, and often the best way to do this is to act *unpredictably*. This is why restaurant hygiene inspectors do random inspection visits.

6.6 STATE-OF-THE-ART GAME PROGRAMS

One might say that game playing is to AI as Grand Prix motor racing is to the car industry: state-of-the-art game programs are blindingly fast, incredibly well-tuned machines that incorporate very advanced engineering techniques, but they aren't much use for doing the shopping. Although some researchers believe that game playing is somewhat irrelevant to mainstream AI, it continues to generate both excitement and a steady stream of innovations that have been adopted by the wider community.

CHESS

Chess: In 1957, Herbert Simon predicted that within 10 years computers would beat the human world champion. Forty years later, the Deep Blue program defeated Garry Kasparov in a six-game exhibition match. Simon was wrong, but only by a factor of 4. Kasparov wrote:

> The decisive game of the match was Game 2, which left a scar in my memory . . . we saw something that went well beyond our wildest expectations of how well a computer would be able to foresee the long-term positional consequences of its decisions. The machine

> refused to move to a position that had a decisive short-term advantage—showing a very human sense of danger. (Kasparov, 1997)

Deep Blue was developed by Murray Campbell, Feng-Hsiung Hsu, and Joseph Hoane at IBM (see Campbell *et al.*, 2002), building on the Deep Thought design developed earlier by Campbell and Hsu at Carnegie Mellon. The winning machine was a parallel computer with 30 IBM RS/6000 processors running the "software search" and 480 custom VLSI chess processors that performed move generation (including move ordering), the "hardware search" for the last few levels of the tree, and the evaluation of leaf nodes. Deep Blue searched 126 million nodes per second on average, with a peak speed of 330 million nodes per second. It generated up to 30 billion positions per move, reaching depth 14 routinely. The heart of the machine is a standard iterative-deepening alpha–beta search with a transposition table, but the key to its success seems to have been its ability to generate extensions beyond the depth limit for sufficiently interesting lines of forcing/forced moves. In some cases the search reached a depth of 40 plies. The evaluation function had over 8000 features, many of them describing highly specific patterns of pieces. An "opening book" of about 4000 positions was used, as well as a database of 700,000 grandmaster games from which consensus recommendations could be extracted. The system also used a large endgame database of solved positions, containing all positions with five pieces and many with six pieces. This database has the effect of substantially extending the effective search depth, allowing Deep Blue to play perfectly in some cases even when it is many moves away from checkmate.

The success of Deep Blue reinforced the widely held belief that progress in computer game-playing has come primarily from ever-more-powerful hardware—a view encouraged by IBM. Deep Blue's creators, on the other hand, state that the search extensions and evaluation function were also critical (Campbell *et al.*, 2002). Moreover, we know that several recent algorithmic improvements have allowed programs running on standard PCs to win every World Computer-Chess Championship since 1992, often defeating massively parallel opponents that could search 1000 times more nodes. A variety of pruning heuristics are used to reduce the effective branching factor to less than 3 (compared with the actual branching factor of about 35). The most important of these is the **null move** heuristic, which generates a good lower bound on the value of a position, using a shallow search in which the opponent gets to move twice at the beginning. This lower bound often allows alpha–beta pruning without the expense of a full-depth search. Also important is **futility pruning**, which helps decide in advance which moves will cause a beta cutoff in the successor nodes.

NULL MOVE

FUTILITY PRUNING

The Deep Blue team declined a chance for a rematch with Kasparov. Instead, the most recent major competition in 2002 featured the program FRITZ against world champion Vladimir Kramnik. The eight game match ended in a draw. The conditions of the match were much more favorable to the human, and the hardware was an ordinary PC, not a supercomputer. Still, Kramnik commented that "It is now clear that the top program and the world champion are approximately equal."

CHECKERS

Checkers: Beginning in 1952, Arthur Samuel of IBM, working in his spare time, developed a checkers program that learned its own evaluation function by playing itself thousands of times. We describe this idea in more detail in Chapter 21. Samuel's program began as a

novice, but after only a few days' self-play had improved itself beyond Samuel's own level (although he was not a strong player). In 1962 it defeated Robert Nealy, a champion at "blind checkers," through an error on his part. Many people that felt this meant computers were superior to people at checkers, but this was not the case. Still, when one considers that Samuel's computing equipment (an IBM 704) had 10,000 words of main memory, magnetic tape for long-term storage, and a .000001–GHz processor, the win remains a great accomplishment.

Few other people attempted to do better until Jonathan Schaeffer and colleagues developed Chinook, which runs on regular PCs and uses alpha–beta search. Chinook uses a precomputed database of all 444 billion positions with eight or fewer pieces on the board to make its endgame play flawless. Chinook came in second in the 1990 U.S. Open and earned the right to challenge for the world championship. It then ran up against a problem, in the form of Marion Tinsley. Dr. Tinsley had been world champion for over 40 years, losing only three games in all that time. In the first match against Chinook, Tinsley suffered his fourth and fifth losses, but won the match 20.5–18.5. The world championship match in August 1994 between Tinsley and Chinook ended prematurely when Tinsley had to withdraw for health reasons. Chinook became the official world champion.

Schaeffer believes that, with enough computing power, the database of endgames could be enlarged to the point where a forward search from the initial position would always reach solved positions, i.e., checkers would be completely solved. (Chinook has announced a win as early as move 5.) This kind of exhaustive analysis can be done by hand for 3×3 tic-tac-toe and has been done by computer for Qubic ($4 \times 4 \times 4$ tic-tac-toe), Go-Moku (five in a row), and Nine-Men's Morris (Gasser, 1998). Remarkable work by Ken Thompson and Lewis Stiller (1992) solved all five-piece and some six-piece chess endgames, making them available on the Internet. Stiller discovered one case where a forced mate existed but required 262 moves; this caused some consternation because the rules of chess require some "progress" to occur within 50 moves.

OTHELLO

Othello, also called Reversi, is probably more popular as a computer game than as a board game. It has a smaller search space than chess, usually 5 to 15 legal moves, but evaluation expertise had to be developed from scratch. In 1997, the Logistello program (Buro, 2002) defeated the human world champion, Takeshi Murakami , by six games to none. It is generally acknowledged that humans are no match for computers at Othello.

BACKGAMMON

Backgammon: Section 6.5 explained why the inclusion of uncertainty from dice rolls makes deep search an expensive luxury. Most work on backgammon has gone into improving the evaluation function. Gerry Tesauro (1992) combined Samuel's reinforcement learning method with neural network techniques (Chapter 20) to develop a remarkably accurate evaluator that is used with a search to depth 2 or 3. After playing more than a million training games against itself, Tesauro's program, TD-GAMMON, is reliably ranked among the top three players in the world. The program's opinions on the opening moves of the game have in some cases radically altered the received wisdom.

GO

Go is the most popular board game in Asia, requiring at least as much discipline from its professionals as chess. Because the board is 19×19, the branching factor starts at 361, which is too daunting for regular search methods. Up to 1997 there were no competent

programs at all, but now programs often play respectable moves. Most of the best programs combine pattern recognition techniques (when the following pattern of pieces appears, this move should be considered) with limited search (decide whether these pieces can be captured, staying within the local area). The strongest programs at the time of writing are probably Chen Zhixing's Goemate and Michael Reiss' Go4++, each rated somewhere around 10 kyu (weak amateur). Go is an area that is likely to benefit from intensive investigation using more sophisticated reasoning methods. Success may come from finding ways to integrate several lines of local reasoning about each of the many, loosely connected "subgames" into which Go can be decomposed. Such techniques would be of enormous value for intelligent systems in general.

BRIDGE

Bridge is a game of imperfect information: a player's cards are hidden from the other players. Bridge is also a *multiplayer* game with four players instead of two, although the players are paired into two teams. As we saw in Section 6.5, optimal play in bridge can include elements of information-gathering, communication, bluffing, and careful weighing of probabilities. Many of these techniques are used in the Bridge Baron^{TM} program (Smith *et al.*, 1998), which won the 1997 computer bridge championship. While it does not play optimally, Bridge Baron is one of the few successful game-playing systems to use complex, hierarchical plans (see Chapter 12) involving high-level ideas such as **finessing** and **squeezing** that are familiar to bridge players.

The GIB program (Ginsberg, 1999) won the 2000 championship quite decisively. GIB uses the "averaging over clairvoyancy" method, with two crucial modifications. First, rather than examining how well each choice works for every possible arrangement of the hidden cards—of which there can be up to 10 million—it examines a random sample of 100 arrangements. Second, GIB uses **explanation-based generalization** to compute and cache general rules for optimal play in various standard classes of situations. This enables it to solve each deal *exactly*. GIB's tactical accuracy makes up for its inability to reason about information. It finished 12th in a field of 35 in the par contest (involving just play of the hand) at the 1998 human world championship, far exceeding the expectations of many human experts.

6.7 DISCUSSION

Because calculating optimal decisions in games is intractable in most cases, all algorithms must make some assumptions and approximations. The standard approach, based on minimax, evaluation functions, and alpha–beta, is just one way to do this. Probably because it was proposed so early on, the standard approach been developed intensively and dominates other methods in tournament play. Some in the field believe that this has caused game playing to become divorced from the mainstream of AI research, because the standard approach no longer provides much room for new insight into general questions of decision making. In this section, we look at the alternatives.

First, let us consider minimax. Minimax selects an optimal move in a given search tree *provided that the leaf node evaluations are exactly correct*. In reality, evaluations are

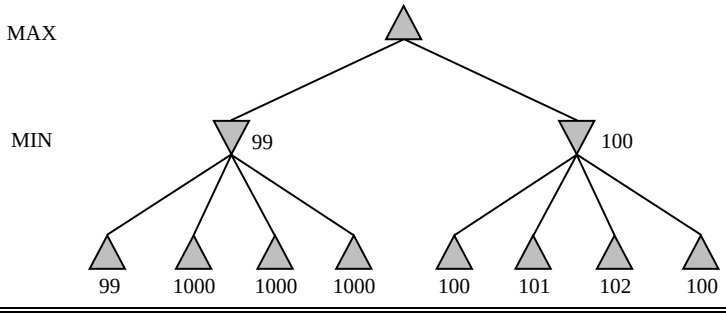


Figure 6.13 A two-ply game tree for which minimax may be inappropriate.

usually crude estimates of the value of a position and can be considered to have large errors associated with them. Figure 6.13 shows a two-ply game tree for which minimax seems inappropriate. Minimax suggests taking the right-hand branch, whereas it is quite likely that the true value of the left-hand branch is higher. The minimax choice relies on the assumption that *all* of the nodes labeled with values 100, 101, 102, and 100 are *actually* better than the node labeled with value 99. However, the fact that the node labeled 99 has siblings labeled 1000 suggests that in fact it might have a higher true value. One way to deal with this problem is to have an evaluation that returns a *probability distribution* over possible values. Then one can calculate the probability distribution for the parent's value using standard statistical techniques. Unfortunately, the values of sibling nodes are usually highly correlated, so this can be an expensive calculation, requiring hard to obtain information.

Next, we consider the search algorithm that generates the tree. The aim of an algorithm designer is to specify a computation that runs quickly and yields a good move. The most obvious problem with the alpha–beta algorithm is that it is designed not just to select a good move, but also to calculate bounds on the values of all the legal moves. To see why this extra information is unnecessary, consider a position in which there is only one legal move. Alpha–beta search still will generate and evaluate a large, and totally useless, search tree. Of course, we can insert a test into the algorithm, but this merely hides the underlying problem: many of the calculations done by alpha–beta are largely irrelevant. Having only one legal move is not much different from having several legal moves, one of which is fine and the rest of which are obviously disastrous. In a "clear favorite" situation like this, it would be better to reach a quick decision after a small amount of search than to waste time that could be more productively used later on a more problematic position. This leads to the idea of the *utility of a node expansion*. A good search algorithm should select node expansions of high utility—that is, ones that are likely to lead to the discovery of a significantly better move. If there are no node expansions whose utility is higher than their cost (in terms of time), then the algorithm should stop searching and make a move. Notice that this works not only for clear-favorite situations, but also for the case of *symmetrical* moves, for which no amount of search will show that one move is better than another.

METAREASONING

This kind of reasoning about what computations to do is called **metareasoning** (reasoning about reasoning). It applies not just to game playing, but to any kind of reasoning

at all. All computations are done in the service of trying to reach better decisions, all have costs, and all have some likelihood of resulting in a certain improvement in decision quality. Alpha–beta incorporates the simplest kind of metareasoning, namely, a theorem to the effect that certain branches of the tree can be ignored without loss. It is possible to do much better. In Chapter 16, we will see how these ideas can be made precise and implementable.

Finally, let us reexamine the nature of search itself. Algorithms for heuristic search and for game playing work by generating sequences of concrete states, starting from the initial state and then applying an evaluation function. Clearly, this is not how humans play games. In chess, one often has a particular goal in mind—for example, trapping the opponent's queen—and can use this goal to *selectively* generate plausible plans for achieving it. This kind of **goal-directed reasoning** or **planning** sometimes eliminates combinatorial search altogether. (See Part IV.) David Wilkins' (1980) PARADISE is the only program to have used goal-directed reasoning successfully in chess: it was capable of solving some chess problems requiring an 18-move combination. As yet there is no good understanding of how to *combine* the two kinds of algorithm into a robust and efficient system, although Bridge Baron might be a step in the right direction. A fully integrated system would be a significant achievement not just for game-playing research, but also for AI research in general, because it would be a good basis for a general intelligent agent.

6.8 SUMMARY

We have looked at a variety of games to understand what optimal play means and to understand how to play well in practice. The most important ideas are as follows:

- A game can be defined by the **initial state** (how the board is set up), the legal **actions** in each state, a **terminal test** (which says when the game is over), and a **utility function** that applies to terminal states.
- In two-player zero-sum games with **perfect information**, the **minimax** algorithm can select optimal moves using a depth-first enumeration of the game tree.
- The **alpha–beta** search algorithm computes the same optimal move as minimax, but achieves much greater efficiency by eliminating subtrees that are provably irrelevant.
- Usually, it is not feasible to consider the whole game tree (even with alpha–beta), so we need to cut the search off at some point and apply an **evaluation function** that gives an estimate of the utility of a state.
- Games of chance can be handled by an extension to the minimax algorithm that evaluates a **chance node** by taking the average utility of all its children nodes, weighted by the probability of each child.
- Optimal play in games of **imperfect information**, such as bridge, requires reasoning about the current and future **belief states** of each player. A simple approximation can be obtained by averaging the value of an action over each possible configuration of missing information.

- Programs can match or beat the best human players in checkers, Othello, and backgammon and are close behind in bridge. A program has beaten the world chess champion in one exhibition match. Programs remain at the amateur level in Go.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The early history of mechanical game playing was marred by numerous frauds. The most notorious of these was Baron Wolfgang von Kempelen's (1734-1804) "The Turk," a supposed chess-playing automaton that defeated Napoleon before being exposed as a magician's trick cabinet housing a human chess expert (see Levitt, 2000). It played from 1769 to 1854. In 1846, Charles Babbage (who had been fascinated by the Turk) appears to have contributed the first serious discussion of the feasibility of computer chess and checkers (Morrison and Morrison, 1961). He also designed, but did not build, a special-purpose machine for playing tic-tac-toe. The first true game-playing machine was built around 1890 by the Spanish engineer Leonardo Torres y Quevedo. It specialized in the "KRK" (king and rook vs. king) chess endgame, guaranteeing a win with king and rook from any position.

The minimax algorithm is often traced to a paper published in 1912 by Ernst Zermelo, the developer of modern set theory. The paper unfortunately contained several errors and did not describe minimax correctly. A solid foundation for game theory was developed in the seminal work *Theory of Games and Economic Behavior* (von Neumann and Morgenstern, 1944), which included an analysis showing that some games *require* strategies that are randomized (or otherwise unpredictable). See Chapter 17 for more information.

Many influential figures of the early computer era were intrigued by the possibility of computer chess. Konrad Zuse (1945), the first person to design a programmable computer, developed fairly detailed ideas about how it might be done. Norbert Wiener's (1948) influential book *Cybernetics* discussed one possible design for a chess program, including the ideas of minimax search, depth cutoffs, and evaluation functions. Claude Shannon (1950) laid out the basic principles of modern game-playing programs in much more detail than Wiener. He introduced the idea of quiescence search and described some ideas for selective (nonexhaustive) game-tree search. Slater (1950) and the commentators on his article also explored the possibilities for computer chess play. In particular, I. J. Good (1950) developed the notion of quiescence independently of Shannon.

In 1951, Alan Turing wrote the first computer program capable of playing a full game of chess (see Turing *et al.*, 1953). But Turing's program never actually ran on a computer; it was tested by hand simulation against a very weak human player, who defeated it. Meanwhile D. G. Prinz (1952) had written, and actually run, a program that solved chess problems, although it did not play a full game. Alex Bernstein wrote the first program to play a full game of standard chess (Bernstein and Roberts, 1958; Bernstein *et al.*, 1958).[3]

John McCarthy conceived the idea of alpha–beta search in 1956, although he did not publish it. The NSS chess program (Newell *et al.*, 1958) used a simplified version of alpha–

[3] Newell *et al.* (1958) mention a Russian program, BESM, that may have predated Bernstein's program.

beta; it was the first chess program to do so. According to Nilsson (1971), Arthur Samuel's checkers program (Samuel, 1959, 1967) also used alpha–beta, although Samuel did not mention it in the published reports on the system. Papers describing alpha–beta were published in the early 1960s (Hart and Edwards, 1961; Brudno, 1963; Slagle, 1963b). An implementation of full alpha–beta is described by Slagle and Dixon (1969) in a program for playing the game of Kalah. Alpha–beta was also used by the "Kotok–McCarthy" chess program written by a student of John McCarthy (Kotok, 1962). Knuth and Moore (1975) provide a history of alpha–beta, along with a proof of its correctness and a time complexity analysis. Their analysis of alpha–beta with random successor ordering showed an asymptotic complexity of $O((b/\log b)^d)$, which seemed rather dismal because the effective branching factor $b/\log b$ is not much less than b itself. They then realized that the asymptotic formula is accurate only for $b > 1000$ or so, whereas the often-quoted $O(b^{3d/4})$ applies to the range of branching factors encountered in actual games. Pearl (1982b) shows alpha–beta to be asymptotically optimal among all fixed-depth game-tree search algorithms.

The first computer chess match featured the Kotok-McCarthy program and the "ITEP" program written in the mid-1960s at Moscow's Institute of Theoretical and Experimental Physics (Adelson-Velsky *et al.*, 1970). This intercontinental match was played by telegraph. It ended with a 3–1 victory for the ITEP program in 1967. The first chess program to compete successfully with humans was MacHack 6 (Greenblatt *et al.*, 1967). Its rating of approximately 1400 was well above the novice level of 1000, but it fell far short of the rating of 2800 or more that would have been needed to fulfill Herb Simon's 1957 prediction that a computer program would be world chess champion within 10 years (Simon and Newell, 1958).

Beginning with the first ACM North American Computer-Chess Championship in 1970, competition among chess programs became serious. Programs in the early 1970s became extremely complicated, with various kinds of tricks for eliminating some branches of search, for generating plausible moves, and so on. In 1974, the first World Computer-Chess Championship was held in Stockholm and won by Kaissa (Adelson-Velsky *et al.*, 1975), another program from ITEP. Kaissa used the much more straightforward approach of exhaustive alpha–beta search combined with quiescence search. The dominance of this approach was confirmed by the convincing victory of CHESS 4.6 in the 1977 World Computer-Chess Championship. CHESS 4.6 examined up to 400,000 positions per move and had a rating of 1900.

A later version of Greenblatt's MacHack 6 was the first chess program to run on custom hardware designed specifically for chess (Moussouris *et al.*, 1979), but the first program to achieve notable success through the use of custom hardware was Belle (Condon and Thompson, 1982). Belle's move generation and position evaluation hardware enabled it to explore several million positions per move. Belle achieved a rating of 2250, becoming the first master-level program. The HITECH system, also a special-purpose computer, was designed by former World Correspondence Chess Champion Hans Berliner and his student Carl Ebeling at CMU to allow rapid calculation of evaluation functions (Ebeling, 1987; Berliner and Ebeling, 1989). Generating about 10 million positions per move, HITECH became North American computer champion in 1985 and was the first program to defeat a human grandmaster, in 1987. Deep Thought, which was also developed at CMU, went further in the direction of pure search speed (Hsu *et al.*, 1990). It achieved a rating of 2551 and was the

forerunner of Deep Blue. The Fredkin Prize, established in 1980, offered $5000 to the first program to achieve a master rating, $10,000 to the first program to achieve a USCF (United States Chess Federation) rating of 2500 (near the grandmaster level), and $100,000 for the first program to defeat the human world champion. The $5000 prize was claimed by Belle in 1983, the $10,000 prize by Deep Thought in 1989, and the $100,000 prize by Deep Blue for its victory over Garry Kasparov in 1997. It is important to remember that Deep Blue's success was due to algorithmic improvements as well as hardware (Hsu, 1999; Campbell *et al.*, 2002). Techniques such as the null-move heuristic (Beal, 1990) have led to programs that are quite selective in their searches. The last three World Computer-Chess Championships in 1992, 1995, and 1999 were won by programs running on standard PCs. Probably the most complete description of a modern chess program is provided by Ernst Heinz (2000), whose DARKTHOUGHT program was the highest-ranked noncommercial PC program at the 1999 world championships.

Several attempts have been made to overcome the problems with the "standard approach" that were outlined in Section 6.7. The first selective search algorithm with some theoretical grounding was probably B* (Berliner, 1979), which attempts to maintain interval bounds on the possible value of a node in the game tree, rather than giving it a single point-valued estimate. Leaf nodes are selected for expansion in an attempt to refine the top-level bounds until one move is "clearly best." Palay (1985) extends the B* idea using probability distributions on values in place of intervals. David McAllester's (1988) conspiracy number search expands leaf nodes that, by changing their values, could cause the program to prefer a new move at the root. MGSS* (Russell and Wefald, 1989) uses the decision-theoretic techniques of Chapter 16 to estimate the value of expanding each leaf in terms of the expected improvement in decision quality at the root. It outplayed an alpha–beta algorithm at Othello despite searching an order of magnitude fewer nodes. The MGSS* approach is, in principle, applicable to the control of any form of deliberation.

Alpha–beta search is in many ways the two-player analog of depth-first branch-and-bound, which is dominated by A* in the single-agent case. The SSS* algorithm (Stockman, 1979) can be viewed as a two-player A* and never expands more nodes than alpha–beta to reach the same decision. The memory requirements and computational overhead of the queue make SSS* in its original form impractical, but a linear-space version has been developed from the RBFS algorithm (Korf and Chickering, 1996). Plaat *et al.* (1996) developed a new view of SSS* as a combination of alpha–beta and transposition tables, showing how to overcome the drawbacks of the original algorithm and developing a new variant called MTD(*f*) that has been adopted by a number of top programs.

D. F. Beal (1980) and Dana Nau (1980, 1983) studied the weaknesses of minimax applied to approximate evaluations. They showed that under certain independence assumptions about the distribution of leaf values in the tree, minimaxing can yield values at the root that are actually *less* reliable than the direct use of the evaluation function itself. Pearl's book *Heuristics* (1984) partially explains this apparent paradox and analyzes many game-playing algorithms. Baum and Smith (1997) propose a probability-based replacement for minimax, showing that it results in better choices in certain games. There is still little theory of the effects of cutting off search at different levels and applying evaluation functions.

The expectiminimax algorithm was proposed by Donald Michie (1966), although of course it follows directly from the principles of game-tree evaluation due to von Neumann and Morgenstern. Bruce Ballard (1983) extended alpha–beta pruning to cover trees with chance nodes. The first successful backgammon program was BKG (Berliner, 1977, 1980b); it used a complex, manually constructed evaluation function and searched only to depth 1. It was the first program to defeat a human world champion at a major classic game (Berliner, 1980a). Berliner readily acknowledged that this was a very short exhibition match (not a world championship match) and that BKG was very lucky with the dice. Work by Gerry Tesauro, first on NEUROGAMMON (Tesauro, 1989) and later on TD-GAMMON (Tesauro, 1995), showed that much better results could be obtained via reinforcement learning, which we will cover in Chapter 21.

Checkers, rather than chess, was the first of the classic games fully played by a computer. Christopher Strachey (1952) wrote the first working program for checkers. Schaeffer (1997) gives a highly readable, "warts and all" account of the development of his Chinook world champion checkers program.

The first Go-playing programs were developed somewhat later than those for checkers and chess (Lefkovitz, 1960; Remus, 1962) and have progressed more slowly. Ryder (1971) used a pure search-based approach with a variety of selective pruning methods to overcome the enormous branching factor. Zobrist (1970) used condition–action rules to suggest plausible moves when known patterns appeared. Reitman and Wilcox (1979) combined rules and search to good effect, and most modern programs have followed this hybrid approach. Müller (2002) summarizes the state of the art of computerized Go and provides a wealth of references. Anshelevich (2000) uses related techniques for the game of Hex. The *Computer Go Newsletter*, published by the Computer Go Association, describes current developments.

Papers on computer game playing appear in a variety of venues. The rather misleadingly named conference proceedings *Heuristic Programming in Artificial Intelligence* report on the Computer Olympiads, which include a wide variety of games. There are also several edited collections of important papers on game-playing research (Levy, 1988a, 1988b; Marsland and Schaeffer, 1990). The International Computer Chess Association (ICCA), founded in 1977, publishes the quarterly *ICGA Journal* (formerly the *ICCA Journal*). Important papers have been published in the serial anthology *Advances in Computer Chess*, starting with Clarke (1977). Volume 134 of the journal *Artificial Intelligence* (2002) contains descriptions of state-of-the-art programs for chess, Othello, Hex, shogi, Go, backgammon, poker, Scrabble.$^{\text{TM}}$and other games.

EXERCISES

6.1 This problem exercises the basic concepts of game playing, using tic-tac-toe (noughts and crosses) as an example. We define X_n as the number of rows, columns, or diagonals with exactly n X's and no O's. Similarly, O_n is the number of rows, columns, or diagonals with just n O's. The utility function assigns $+1$ to any position with $X_3 = 1$ and -1 to any

position with $O_3 = 1$. All other terminal positions have utility 0. For nonterminal positions, we use a linear evaluation function defined as $Eval(s) = 3X_2(s)+X_1(s)-(3O_2(s)+O_1(s))$.

a. Approximately how many possible games of tic-tac-toe are there?

b. Show the whole game tree starting from an empty board down to depth 2 (i.e., one X and one O on the board), taking symmetry into account.

c. Mark on your tree the evaluations of all the positions at depth 2.

d. Using the minimax algorithm, mark on your tree the backed-up values for the positions at depths 1 and 0, and use those values to choose the best starting move.

e. Circle the nodes at depth 2 that would *not* be evaluated if alpha–beta pruning were applied, assuming the nodes are generated *in the optimal order for alpha–beta pruning.*

6.2 Prove the following assertion: for every game tree, the utility obtained by MAX using minimax decisions against a suboptimal MIN will be never be lower than the utility obtained playing against an optimal MIN. Can you come up with a game tree in which MAX can do still better using a *suboptimal* strategy against a suboptimal MIN?

6.3 Consider the two-player game described in Figure 6.14.

a. Draw the complete game tree, using the following conventions:

- Write each state as (s_A, s_B) where s_A and s_B denote the token locations.
- Put each terminal state in a square boxes and write its game value in a circle.
- Put *loop states* (states that already appear on the path to the root) in double square boxes. Since it is not clear how to assign values to loop states, annotate each with a "?" in a circle.

b. Now mark each node with its backed-up minimax value (also in a circle). Explain how you handled the "?" values and why.

c. Explain why the standard minimax algorithm would fail on this game tree and briefly sketch how you might fix it, drawing on your answer to (b). Does your modified algorithm give optimal decisions for all games with loops?

d. This 4-square game can be generalized to n squares for any $n > 2$. Prove that A wins if n is even and loses if n is odd.

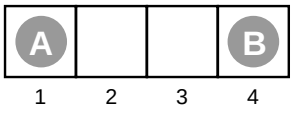
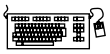


Figure 6.14 The starting position of a simple game. Player A moves first. The two players take turns moving, and each player must move his token to an open adjacent space *in either direction.* If the opponent occupies an adjacent space, then a player may jump over the opponent to the next open space if any. (For example, if A is on 3 and B is on 2, then A may move back to 1.) The game ends when one player reaches the opposite end of the board. If player A reaches space 4 first, then the value of the game to A is $+1$; if player B reaches space 1 first, then the value of the game to A is -1.



6.4 Implement move generators and evaluation functions for one or more of the following games: Kalah, Othello, checkers, and chess. Construct a general alpha–beta game-playing agent that uses your implementation. Compare the effect of increasing search depth, improving move ordering, and improving the evaluation function. How close does your effective branching factor come to the ideal case of perfect move ordering?

6.5 Develop a formal proof of correctness for alpha-beta pruning. To do this, consider the situation shown in Figure 6.15. The question is whether to prune node n_j, which is a max-node and a descendant of node n_1. The basic idea is to prune it if and only if the minimax value of n_1 can be shown to be independent of the value of n_j.

a. The value of n_1 is given by

$$n_1 = \min(n_2, n_{21}, \ldots, n_{2b_2}) \ .$$

Find a similar expression for n_2 and hence an expression for n_1 in terms of n_j.

b. Let l_i be the minimum (or maximum) value of the nodes to the *left* of node n_i at depth i, whose minimax value is already known. Similarly, let r_i be the minimum (or maximum) value of the unexplored nodes the right of n_i at depth i. Rewrite your expression for n_1 in terms of the l_i and r_i values.

c. Now reformulate the expression to show that in order to affect n_1, n_j must not exceed a certain bound derived from the l_i values.

d. Repeat the process for the case where n_j is a min-node.

6.6 Implement the expectiminimax algorithm and the *-alpha–beta algorithm, which is described by Ballard (1983), for pruning game trees with chance nodes. Try them on a game such as backgammon and measure the pruning effectiveness of *-alpha–beta.

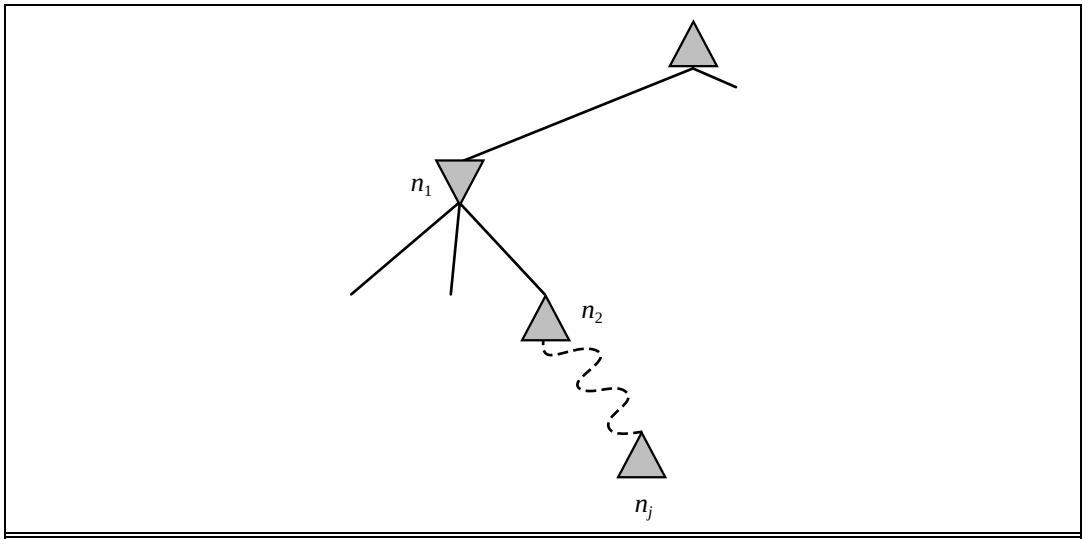


Figure 6.15 Situation when considering whether to prune node n_j.

6.7 Prove that with a positive linear transformation of leaf values (i.e., transforming a value x to $ax + b$ where $a > 0$), the choice of move remains unchanged in a game tree, even when there are chance nodes.

6.8 Consider the following procedure for choosing moves in games with chance nodes:

- Generate some die-roll sequences (say, 50) down to a suitable depth (say, 8).
- With known die rolls, the game tree becomes deterministic. For each die-roll sequence, solve the resulting deterministic game tree using alpha–beta.
- Use the results to estimate the value of each move and to choose the best.

Will this procedure work well? Why (not)?

6.9 Describe and implement a *real-time, multiplayer* game-playing environment, where time is part of the environment state and players are given fixed time allocations.

6.10 Describe or implement state descriptions, move generators, terminal tests, utility functions, and evaluation functions for one or more of the following games: Monopoly, Scrabble, bridge (assuming a given contract), and poker (choose your favorite variety).

6.11 Consider carefully the interplay of chance events and partial information in each of the games in Exercise 6.10.

a. For which is the standard expectiminimax model appropriate? Implement the algorithm and run it in your game-playing agent, with appropriate modifications to the game-playing environment.

b. For which would the scheme described in Exercise 6.8 be appropriate?

c. Discuss how you might deal with the fact that in some of the games, the players do not have the same knowledge of the current state.

6.12 The minimax algorithm assumes that players take turns moving, but in card games such as whist and bridge, the winner of the previous trick plays first on the next trick.

a. Modify the algorithm to work properly for these games. You may assume that a function WINNER($trick$) is available that reports which card wins a given trick.

b. Draw the game tree for the first pair of hands shown on page 179.

6.13 The Chinook checkers program makes extensive use of endgame databases, which provide exact values for every position with eight or fewer pieces. How might such databases be generated efficiently?

6.14 Discuss how well the standard approach to game playing would apply to games such as tennis, pool, and croquet, which take place in a continuous physical state space.

6.15 Describe how the minimax and alpha–beta algorithms change for two-player, **non-zero-sum games** in which each player has his or her own utility function. You may assume that each player knows the other's utility function. If there are no constraints on the two terminal utilities, is it possible for any node to be pruned by alpha–beta?

6.16 Suppose you have a chess program that can evaluate 1 million nodes per second. Decide on a compact representation of a game state for storage in a transposition table. About how many entries can you fit in a 500MB in-memory table? Will that be enough for the three minutes of search allocated for one move? How many table lookups can you do in the time it would take to do one evaluation? Now suppose the transposition table is larger than can fit in memory. About how many evaluations could you do in the time it takes to do one disk seek with standard disk hardware?