5 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.

5.1 GAMES

Chapter 2 introduced **multiagent environments**, in which each agent needs to consider the actions of other agents and how they affect its own welfare. The unpredictability of these other agents can introduce **contingencies** into the agent's problem-solving process, as discussed in Chapter 4. In this chapter we cover **competitive** environments, in which the agents' goals are in conflict, giving rise to **adversarial search** problems—often known as **games**.

GAME

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, the most common games are of a rather specialized kind—what game theorists call deterministic, turn-taking, two-player, **zero-sum games** of **perfect information** (such as chess). In our terminology, this means deterministic, fully observable environments in which two agents act alternately 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, the other player necessarily loses. It is this opposition between the agents' utility functions that makes the situation adversarial.

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 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.

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

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 take twice as long 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 5.5 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.

PRUNING

IMPERFECT INFORMATION

We first consider games with two players, whom we 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 elements:

- S_0: The **initial state**, which specifies how the game is set up at the start.
- PLAYER(s): Defines which player has the move in a state.
- ACTIONS(s): Returns the set of legal moves in a state.
- RESULT(s, a): The **transition model**, which defines the result of a move.
- TERMINAL-TEST(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- UTILITY(s, p): A **utility function** (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p. In chess, the outcome is a win, loss, or draw, with values $+1$, 0, or $\frac{1}{2}$. Some games have a wider variety of possible outcomes; the payoffs in backgammon range from 0 to $+192$. A **zero-sum game** is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game. Chess is zero-sum because every game has payoff of either $0 + 1$, $1 + 0$ or $\frac{1}{2} + \frac{1}{2}$. "Constant-sum" would have been a better term, but zero-sum is traditional and makes sense if you imagine each player is charged an entry fee of $\frac{1}{2}$.

TERMINAL TEST

TERMINAL STATES

GAME TREE

The initial state, ACTIONS function, and RESULT function define the **game tree** for the game—a tree where the nodes are game states and the edges are moves. Figure 5.1 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).

For tic-tac-toe the game tree is relatively small—fewer than $9! = 362,880$ terminal nodes. But for chess there are over 10^{40} nodes, so the game tree is best thought of as a theoretical construct that we cannot realize in the physical world. But regardless of the size of the game tree, it is MAX's job to search for a good move. We use the term **search tree** for a tree that is superimposed on the full game tree, and examines enough nodes to allow a player to determine what move to make.

SEARCH TREE

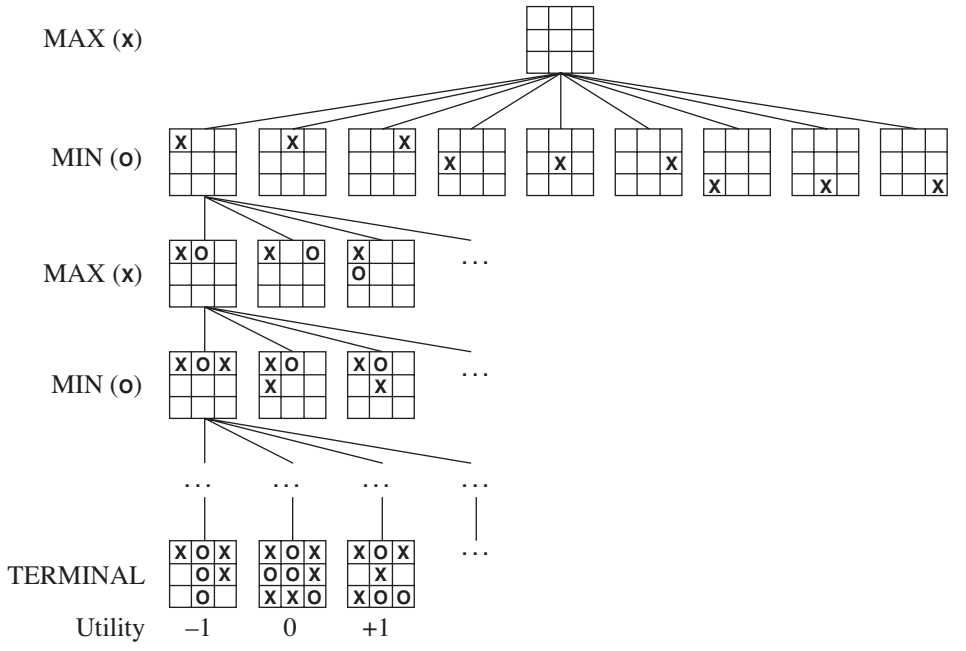


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

5.2 Optimal Decisions in Games

In a normal search problem, the optimal solution would be a sequence of actions leading to a goal state—a terminal state that is a win. In adversarial search, MIN has something 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

STRATEGY

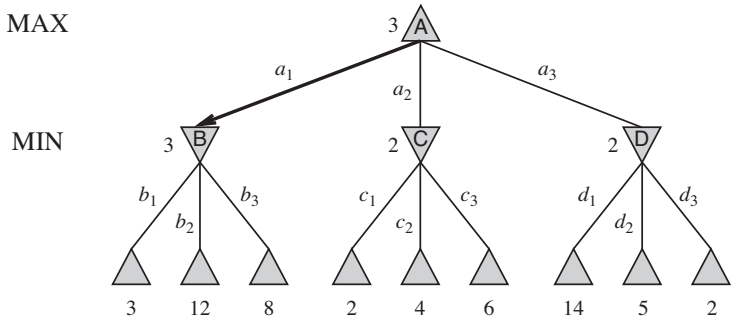


Figure 5.2 A two-ply game tree. The $\triangle$ nodes are "MAX nodes," in which it is MAX's turn to move, and the $\triangledown$ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

MIN, then MAX's moves in the states resulting from every possible response by MIN to *those* moves, and so on. This is exactly analogous to the AND–OR search algorithm (Figure 4.11) with MAX playing the role of OR and MIN equivalent to AND. Roughly speaking, an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent. We begin by showing how to find this optimal strategy.

Even a simple game like tic-tac-toe is too complex for us to draw the entire game tree on one page, so we will switch to the trivial game in Figure 5.2. 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 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.

PLY

MINIMAX VALUE

Given a game tree, the optimal strategy can be determined from the **minimax value** of each node, which we write as $\text{MINIMAX}(n)$. 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 prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value. So we have the following:

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Let us apply these definitions to the game tree in Figure 5.2. The terminal nodes on the bottom level get their utility values from the game's UTILITY function. The first MIN node, labeled B, has three successor states 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 successor states have minimax values 3, 2, and 2; so it has a minimax value of 3. We can also identify

MINIMAX DECISION

the **minimax decision** at the root: action a_1 is the optimal choice for MAX because it leads to the state 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 5.7) that MAX will do even better. Other strategies against suboptimal opponents may do better than the minimax strategy, but these strategies necessarily do worse against optimal opponents.

5.2.1 The minimax algorithm

MINIMAX ALGORITHM

The **minimax algorithm** (Figure 5.3) 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 of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds. For example, in Figure 5.2, 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 actions at once, or $O(m)$ for an algorithm that generates actions one at a time (see page 87). 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.

5.2.2 Optimal decisions in multiplayer games

Many popular games allow more than two players. Let us examine how to 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 5.4. 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, the backed-up value of X is this vector. The backed-up value of a node n is always the utility

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

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

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

Figure 5.3 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. The notation $\text{argmax}_{a \in S} f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

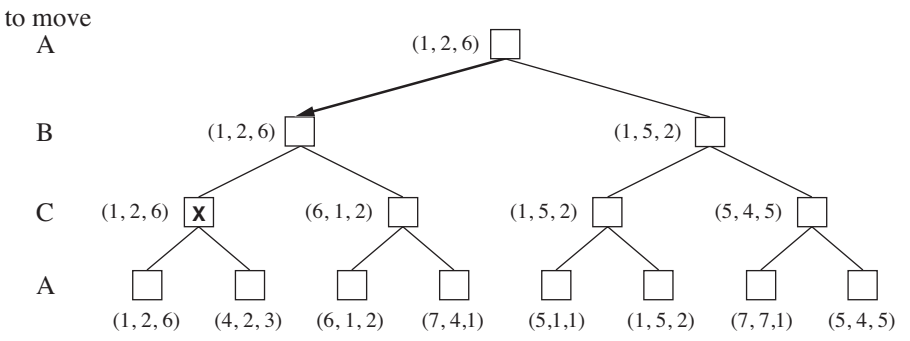


Figure 5.4 The first three plies of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

vector of the successor state with the highest value for the player choosing at n. Anyone who plays multiplayer games, such as Diplomacy, quickly becomes aware that much more is going on than in two-player games. Multiplayer games usually involve **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,

ALLIANCE

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, a social stigma attaches 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 17.5 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.

5.3 ALPHA–BETA PRUNING

The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree. Unfortunately, we can't eliminate the exponent, but it turns out 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 3 to eliminate large parts of the tree from consideration. The particular technique we 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.

ALPHA–BETA PRUNING

Consider again the two-ply game tree from Figure 5.2. 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 5.5. 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. Let the two unevaluated successors of node C in Figure 5.5 have values x and y. Then the value of the root node is given by

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

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.

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

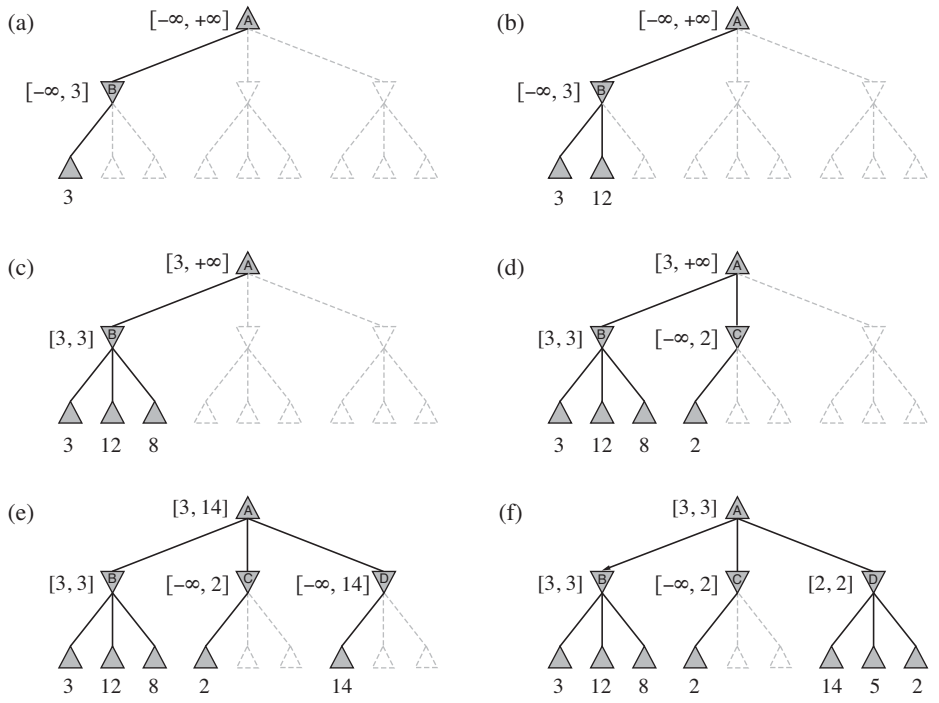


Figure 5.5 Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value 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 successor states, 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 successor states 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 successor states. 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.

somewhere in the tree (see Figure 5.6), 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 following two parameters that describe bounds on the backed-up values that appear anywhere along the path:

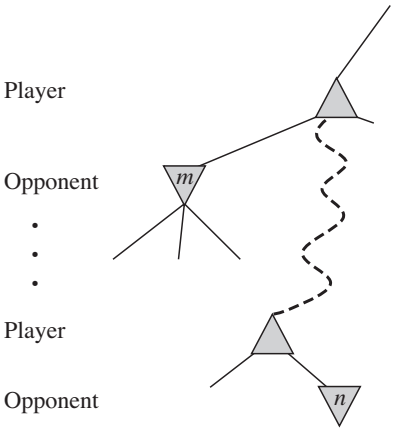


Figure 5.6 The general case for alpha–beta pruning. If m is better than n for Player, we will never get to n in play.

$\alpha =$ 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 5.7. We encourage you to trace its behavior when applied to the tree in Figure 5.5.

5.3.1 Move ordering

The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined. For example, in Figure 5.5(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 of D 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 this can be done,[2] then it turns out that alpha–beta needs to examine only $O(b^{m/2})$ nodes to pick the best move, instead of $O(b^m)$ for minimax. This means that the effective branching factor becomes $\sqrt{b}$ instead of b—for chess, about 6 instead of 35. Put another way, alpha–beta can solve a tree roughly twice as deep 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 $O(b^{3m/4})$ 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 $O(b^{m/2})$ result.

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

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, 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).

Adding dynamic move-ordering schemes, such as trying first the moves that were found to be best in the past, brings us quite close to the theoretical limit. The past could be the previous move—often the same threats remain—or it could come from previous exploration of the current move. One way to gain information from the current move is with iterative deepening search. First, search 1 ply deep and record the best path of moves. Then search 1 ply deeper, but use the recorded path to inform move ordering. As we saw in Chapter 3, iterative deepening on an exponential game tree adds only a constant fraction to the total search time, which can be more than made up from better move ordering. The best moves are often called **killer moves** and to try them first is called the killer move heuristic.

KILLER MOVES

In Chapter 3, we noted that repeated states in the search tree can cause an exponential increase in search cost. In many 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 $[a_2, b_2, a_1, b_1]$ both end up in the same position. It is worthwhile to store the evaluation of the resulting position in a hash table the first time it is encountered so that we don't have to recompute it on subsequent occurrences. The hash table of previously seen positions is traditionally called a **transposition table**; it is essentially identical to the *explored*

TRANSPOSITION

TRANSPOSITION TABLE

list in GRAPH-SEARCH (Section 3.3). 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, at some point it is not practical to keep *all* of them in the transposition table. Various strategies have been used to choose which nodes to keep and which to discard.

5.4 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. Claude Shannon's paper *Programming a Computer for Playing Chess* (1950) 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: replace the utility function by a heuristic evaluation function EVAL, which estimates the position's utility, and replace the terminal test by a **cutoff test** that decides when to apply EVAL. That gives us the following for heuristic minimax for state s and maximum depth d:

EVALUATION FUNCTION

CUTOFF TEST

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MIN}. \end{cases}$$

5.4.1 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 3 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 depends strongly 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: states that are wins must evaluate better than draws, which in turn must be better than losses. Otherwise, an agent using the evaluation function might err even if it can see ahead all the way to the end of the game. Second, the computation must not take too long! (The whole point is to search faster.) 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 no dice are 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.

Let us make this idea more concrete. Most evaluation functions work by calculating various **features** of the state—for example, in chess, we would have features for the number of white pawns, black pawns, white queens, black queens, and so on. 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. For example, one category contains all two-pawn vs. one-pawn endgames. 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 two-pawns vs. one-pawn category lead to a win (utility +1); 20% to a loss (0), and 8% to a draw (1/2). Then a reasonable evaluation for states in the category is the **expected value**: $(0.72 \times +1) + (0.20 \times 0) + (0.08 \times 1/2) = 0.76$. 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.

MATERIAL VALUE

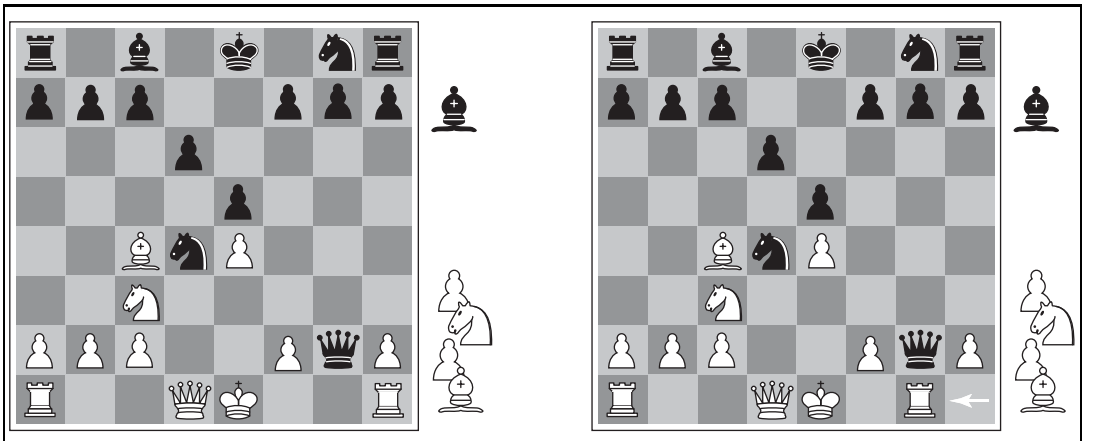
A secure advantage equivalent to a pawn gives a substantial likelihood of winning, and a secure advantage equivalent to three pawns should give almost certain victory, as illustrated in Figure 5.8(a). Mathematically, this kind of evaluation function is called a **weighted linear function** because it can be expressed as

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 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.



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

Figure 5.8 Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

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 (that is, when the *move number* feature is high or the *number of remaining pieces* feature is low).

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. 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.

5.4.2 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. We replace the two lines in Figure 5.7 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 a move is selected within the allocated time. A more robust approach is to apply iterative deepening. (See Chapter 3.) When time runs out, the program returns the move selected by the deepest completed search. As a bonus, iterative deepening also helps with move ordering.

These simple 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 5.8(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 is a probable 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 an opponent's move that causes serious damage and is ultimately unavoidable, but can be temporarily avoided by delaying tactics. Consider the chess game in Figure 5.9. It is clear that there is no way for the black bishop to escape. For example, the white rook can capture it by moving to h1, then a1, then a2; a capture at depth 6 ply. But Black does have a sequence of moves that pushes the capture of the bishop "over the horizon." Suppose Black searches to depth 8 ply. Most moves by Black will lead to the eventual capture of the bishop, and thus will be marked as "bad" moves. But Black will consider checking the white king with the pawn at e4. This will lead to the king capturing the pawn. Now Black will consider checking again, with the pawn at f5, leading to another pawn capture. That takes up 4 ply, and from there the remaining 4 ply is not enough to capture the bishop. Black thinks that the line of play has saved the bishop at the price of two pawns, when actually all it has done is push the inevitable capture of the bishop beyond the horizon that Black can see.

SINGULAR EXTENSION

One strategy to mitigate the horizon effect is the **singular extension**, a move that is "clearly better" than all other moves in a given position. Once discovered anywhere in the tree in the course of a search, this singular move is remembered. When the search reaches the normal depth limit, the algorithm checks to see if the singular extension is a legal move; if it is, the algorithm allows the move to be considered. This makes the tree deeper, but because there will be few singular extensions, it does not add many total nodes to the tree.

5.4.3 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 (at least with respect to the heuristic evaluation values). 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 consider only a few moves from each position (at least consciously). One approach to forward pruning is **beam search**: on each ply, consider only a "beam" of the n best moves (according to the evaluation function) rather than considering all possible moves.

FORWARD PRUNING

BEAM SEARCH

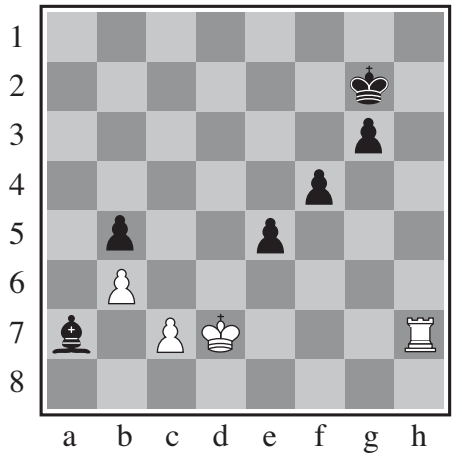


Figure 5.9 The horizon effect. With Black to move, the black bishop is surely doomed. But Black can forestall that event by checking the white king with its pawns, forcing the king to capture the pawns. This pushes the inevitable loss of the bishop over the horizon, and thus the pawn sacrifices are seen by the search algorithm as good moves rather than bad ones.

Unfortunately, this approach is rather dangerous because there is no guarantee that the best move will not be pruned away.

The PROBCUT, or probabilistic cut, algorithm (Buro, 1995) is a forward-pruning version of alpha–beta search that uses statistics gained from prior experience to lessen the chance that the best move will be pruned. Alpha–beta search prunes any node that is *provably* outside the current (α, β) window. PROBCUT also prunes nodes that are *probably* outside the window. It computes this probability by doing a shallow search to compute the backed-up value v of a node and then using past experience to estimate how likely it is that a score of v at depth d in the tree would be outside (α, β). Buro applied this technique to his Othello program, LOGISTELLO, and found that a version of his program with PROBCUT beat the regular version 64% of the time, even when the regular version was given twice as much time.

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 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 plies, which results in an expert level of play. Section 5.8 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.

5.4.4 Search versus lookup

Somehow it seems like overkill for a chess program to start a game by considering a tree of a billion game states, only to conclude that it will move its pawn to e4. Books describing good play in the opening and endgame in chess have been available for about a century (Tattersall, 1911). It is not surprising, therefore, that many game-playing programs use *table lookup* rather than search for the opening and ending of games.

For the openings, the computer is mostly relying on the expertise of humans. The best advice of human experts on how to play each opening is copied from books and entered into tables for the computer's use. However, computers can also gather statistics from a database of previously played games to see which opening sequences most often lead to a win. In the early moves there are few choices, and thus much expert commentary and past games on which to draw. Usually after ten moves we end up in a rarely seen position, and the program must switch from table lookup to search.

Near the end of the game there are again fewer possible positions, and thus more chance to do lookup. But here it is the computer that has the expertise: computer analysis of endgames goes far beyond anything achieved by humans. A human can tell you the general strategy for playing a king-and-rook-versus-king (KRK) endgame: reduce the opposing king's mobility by squeezing it toward one edge of the board, using your king to prevent the opponent from escaping the squeeze. Other endings, such as king, bishop, and knight versus king (KBNK), are difficult to master and have no succinct strategy description. A computer, on the other hand, can completely *solve* the endgame by producing a **policy**, which is a mapping from every possible state to the best move in that state. Then we can just look up the best move rather than recompute it anew. How big will the KBNK lookup table be? It turns out there are 462 ways that two kings can be placed on the board without being adjacent. After the kings are placed, there are 62 empty squares for the bishop, 61 for the knight, and two possible players to move next, so there are just $462 \times 62 \times 61 \times 2 = 3,494,568$ possible positions. Some of these are checkmates; mark them as such in a table. Then do a **retrograde** minimax search: reverse the rules of chess to do unmoves rather than moves. Any move by White that, no matter what move Black responds with, ends up in a position marked as a win, must also be a win. Continue this search until all 3,494,568 positions are resolved as win, loss, or draw, and you have an infallible lookup table for all KBNK endgames.

POLICY

RETROGRADE

Using this technique and a *tour de force* of optimization tricks, Ken Thompson (1986, 1996) and Lewis Stiller (1992, 1996) solved all chess endgames with up to five pieces and some with six pieces, 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 a capture or pawn move to occur within 50 moves. Later work by Marc Bourzutschky and Yakov Konoval (Bourzutschky, 2006) solved all pawnless six-piece and some seven-piece endgames; there is a KQNKRBN endgame that with best play requires 517 moves until a capture, which then leads to a mate.

If we could extend the chess endgame tables from 6 pieces to 32, then White would know on the opening move whether it would be a win, loss, or draw. This has not happened so far for chess, but it has happened for checkers, as explained in the historical notes section.

5.5 STOCHASTIC GAMES

STOCHASTIC GAMES

In real life, many unpredictable external events can put us into unforeseen situations. Many games mirror this unpredictability by including a random element, such as the throwing of dice. We call these **stochastic games**. 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 5.10, for example, White has rolled a 6–5 and has four possible moves.

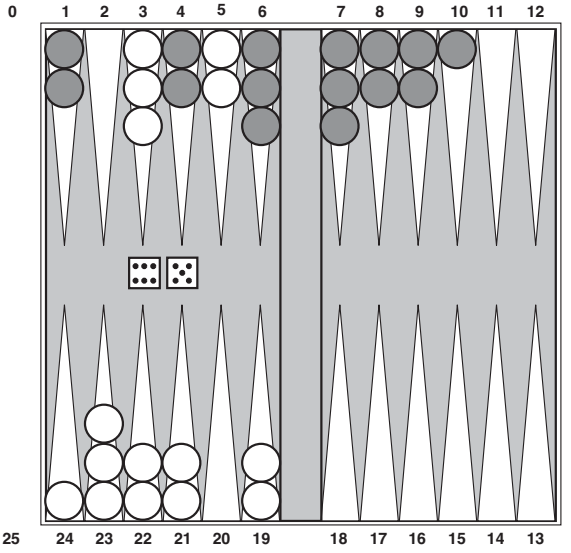


Figure 5.10 A typical backgammon position. The goal of the game is to move all one's pieces off the board. White moves clockwise toward 25, and Black moves counterclockwise toward 0. A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and 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), where the notation (5–11,11–16) means move one piece from position 5 to 11, and then move a piece from 11 to 16.

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 game tree in backgammon must include **chance nodes** in addition to MAX and MIN nodes. Chance nodes are shown as circles in Figure 5.11. The branches leading from each chance node denote the possible dice rolls; each branch is labeled with the roll and its probability. 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) each have a probability of 1/36, so we say $P(1\text{–}1) = 1/36$. The other 15 distinct rolls each have a 1/18 probability.

CHANCE NODES

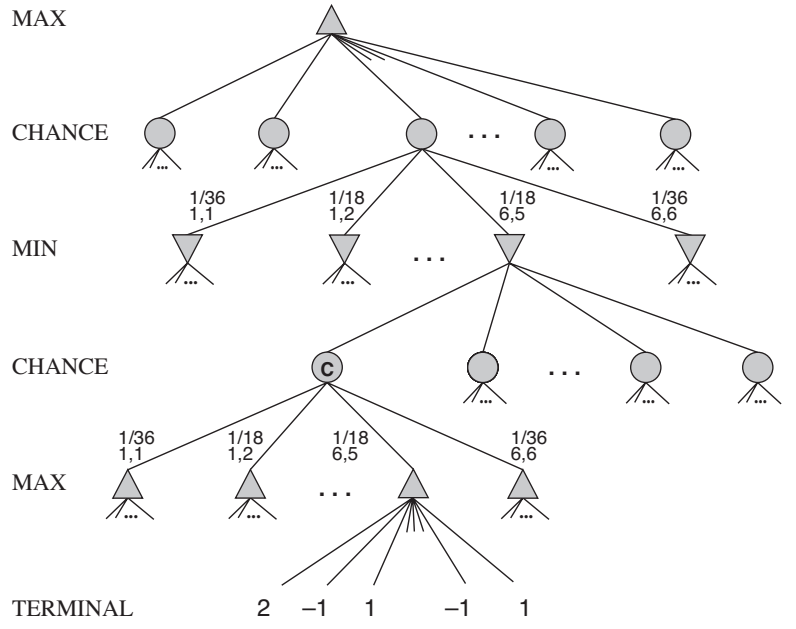


Figure 5.11 Schematic game tree for a backgammon position.

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, positions do not have definite minimax values. Instead, we can only calculate the **expected value** of a position: the average over all possible outcomes of the chance nodes.

EXPECTED VALUE

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. For chance nodes we compute the expected value, which is the sum of the value over all outcomes, weighted by the probability of each chance action:

EXPECTIMINIMAX VALUE

$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \\ \sum_r P(r)\text{EXPECTIMINIMAX}(\text{RESULT}(s,r)) & \text{if PLAYER}(s) = \text{CHANCE} \end{cases}$$

where r represents a possible dice roll (or other chance event) and $\text{RESULT}(s, r)$ is the same state as s, with the additional fact that the result of the dice roll is r.

5.5.1 Evaluation functions for games of chance

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 5.12 shows what happens: with an evaluation function that assigns the 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.

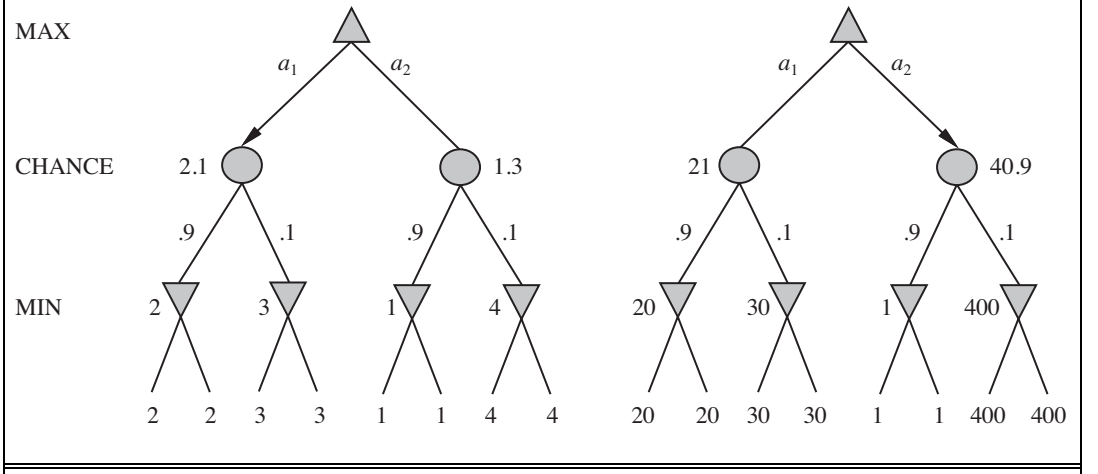


Figure 5.12 An order-preserving transformation on leaf values changes the best move.

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 does in $O(b^m)$ time, where b is the branching factor and m is the maximum depth of the game tree. 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.

It may have occurred to you that 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 5.11 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 in order 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, and in order to compute the average of a set of numbers, we must look at all the numbers. But if we put bounds on the possible values of the utility function, then we can arrive at bounds for the average without looking at every number. For example, say that all utility values are between -2 and $+2$; 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.

MONTE CARLO SIMULATION

An alternative is to do **Monte Carlo simulation** to evaluate a position. Start with an alpha–beta (or other) search algorithm. From a start position, have the algorithm play thousands of games against itself, using random dice rolls. In the case of backgammon, the resulting win percentage has been shown to be a good approximation of the value of the position, even if the algorithm has an imperfect heuristic and is searching only a few plies (Tesauro, 1995). For games with dice, this type of simulation is called a **rollout**.

ROLLOUT

5.6 PARTIALLY OBSERVABLE GAMES

Chess has often been described as war in miniature, but it lacks at least one major characteristic of real wars, namely, **partial observability**. In the "fog of war," the existence and disposition of enemy units is often unknown until revealed by direct contact. As a result, warfare includes the use of scouts and spies to gather information and the use of concealment and bluff to confuse the enemy. Partially observable games share these characteristics and are thus qualitatively different from the games described in the preceding sections.

5.6.1 Kriegspiel: Partially observable chess

In *deterministic* partially observable games, uncertainty about the state of the board arises entirely from lack of access to the choices made by the opponent. This class includes children's games such as Battleships (where each player's ships are placed in locations hidden from the opponent but do not move) and Stratego (where piece locations are known but piece types are hidden). We will examine the game of **Kriegspiel**, a partially observable variant of chess in which pieces can move but are completely invisible to the opponent.

KRIEGSPIEL

The rules of Kriegspiel are as follows: White and Black each see a board containing only their own pieces. A referee, who can see all the pieces, adjudicates the game and periodically makes announcements that are heard by both players. On his turn, White proposes to the referee any move that would be legal if there were no black pieces. If the move is in fact not legal (because of the black pieces), the referee announces "illegal." In this case, White may keep proposing moves until a legal one is found—and learns more about the location of Black's pieces in the process. Once a legal move is proposed, the referee announces one or

more of the following: "Capture on square *X*" if there is a capture, and "Check by *D*" if the black king is in check, where *D* is the direction of the check, and can be one of "Knight," "Rank," "File," "Long diagonal," or "Short diagonal." (In case of discovered check, the referee may make two "Check" announcements.) If Black is checkmated or stalemated, the referee says so; otherwise, it is Black's turn to move.

Kriegspiel may seem terrifyingly impossible, but humans manage it quite well and computer programs are beginning to catch up. It helps to recall the notion of a **belief state** as defined in Section 4.4 and illustrated in Figure 4.14—the set of all *logically possible* board states given the complete history of percepts to date. Initially, White's belief state is a singleton because Black's pieces haven't moved yet. After White makes a move and Black responds, White's belief state contains 20 positions because Black has 20 replies to any White move. Keeping track of the belief state as the game progresses is exactly the problem of **state estimation**, for which the update step is given in Equation (4.6). We can map Kriegspiel state estimation directly onto the partially observable, nondeterministic framework of Section 4.4 if we consider the opponent as the source of nondeterminism; that is, the RESULTS of White's move are composed from the (predictable) outcome of White's own move and the unpredictable outcome given by Black's reply.[3]

Given a current belief state, White may ask, "Can I win the game?" For a partially observable game, the notion of a **strategy** is altered; instead of specifying a move to make for each possible *move* the opponent might make, we need a move for every possible *percept sequence* that might be received. For Kriegspiel, a winning strategy, or **guaranteed checkmate**, is one that, for each possible percept sequence, leads to an actual checkmate for every possible board state in the current belief state, regardless of how the opponent moves. With this definition, the opponent's belief state is irrelevant—the strategy has to work even if the opponent can see all the pieces. This greatly simplifies the computation. Figure 5.13 shows part of a guaranteed checkmate for the KRK (king and rook against king) endgame. In this case, Black has just one piece (the king), so a belief state for White can be shown in a single board by marking each possible position of the Black king.

GUARANTEED CHECKMATE

The general AND-OR search algorithm can be applied to the belief-state space to find guaranteed checkmates, just as in Section 4.4. The incremental belief-state algorithm mentioned in that section often finds midgame checkmates up to depth 9—probably well beyond the abilities of human players.

In addition to guaranteed checkmates, Kriegspiel admits an entirely new concept that makes no sense in fully observable games: **probabilistic checkmate**. Such checkmates are still required to work in every board state in the belief state; they are probabilistic with respect to randomization of the winning player's moves. To get the basic idea, consider the problem of finding a lone black king using just the white king. Simply by moving randomly, the white king will *eventually* bump into the black king even if the latter tries to avoid this fate, since Black cannot keep guessing the right evasive moves indefinitely. In the terminology of probability theory, detection occurs *with probability* 1. The KBNK endgame—king, bishop

PROBABILISTIC CHECKMATE

[3] Sometimes, the belief state will become too large to represent just as a list of board states, but we will ignore this issue for now; Chapters 7 and 8 suggest methods for compactly representing very large belief states.

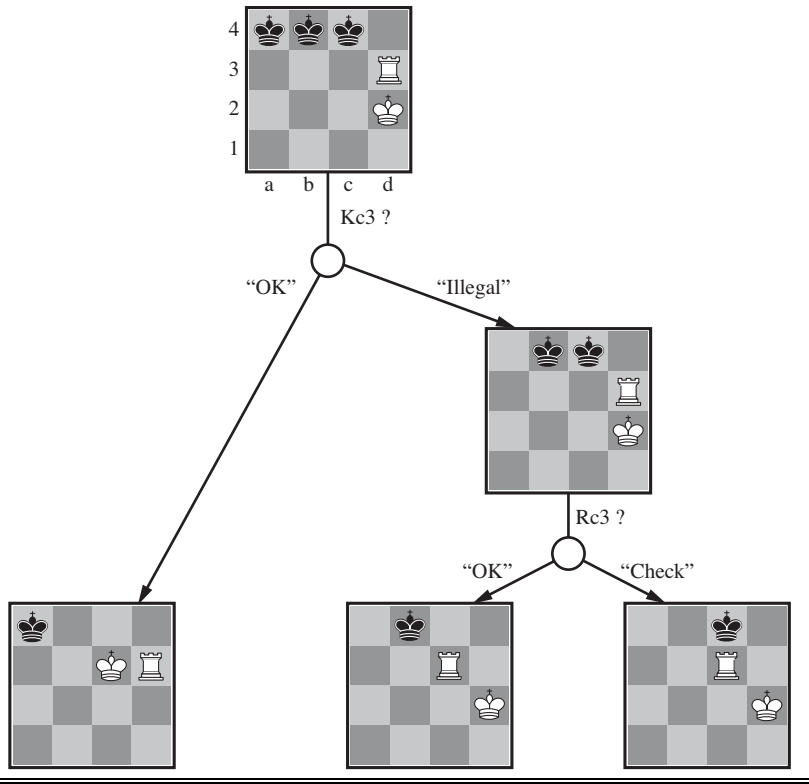


Figure 5.13 Part of a guaranteed checkmate in the KRK endgame, shown on a reduced board. In the initial belief state, Black's king is in one of three possible locations. By a combination of probing moves, the strategy narrows this down to one. Completion of the checkmate is left as an exercise.

and knight against king—is won in this sense; White presents Black with an infinite random sequence of choices, for one of which Black will guess incorrectly and reveal his position, leading to checkmate. The KBBK endgame, on the other hand, is won with probability $1 - \epsilon$. White can force a win only by leaving one of his bishops unprotected for one move. If Black happens to be in the right place and captures the bishop (a move that would lose if the bishops are protected), the game is drawn. White can choose to make the risky move at some randomly chosen point in the middle of a very long sequence, thus reducing ϵ to an arbitrarily small constant, but cannot reduce ϵ to zero.

It is quite rare that a guaranteed or probabilistic checkmate can be found within any reasonable depth, except in the endgame. Sometimes a checkmate strategy works for *some* of the board states in the current belief state but not others. Trying such a strategy may succeed, leading to an **accidental checkmate**—accidental in the sense that White could not *know* that it would be checkmate—if Black's pieces happen to be in the right places. (Most checkmates in games between humans are of this accidental nature.) This idea leads naturally to the question of *how likely* it is that a given strategy will win, which leads in turn to the question of *how likely* it is that each board state in the current belief state is the true board state.

ACCIDENTAL CHECKMATE

One's first inclination might be to propose that all board states in the current belief state are equally likely—but this can't be right. Consider, for example, White's belief state after Black's first move of the game. By definition (assuming that Black plays optimally), Black must have played an optimal move, so all board states resulting from suboptimal moves ought to be assigned zero probability. This argument is not quite right either, because *each player's goal is not just to move pieces to the right squares but also to minimize the information that the opponent has about their location.* Playing any *predictable* "optimal" strategy provides the opponent with information. Hence, optimal play in partially observable games requires a willingness to play somewhat *randomly*. (This is why restaurant hygiene inspectors do *random* inspection visits.) This means occasionally selecting moves that may seem "intrinsically" weak—but they gain strength from their very unpredictability, because the opponent is unlikely to have prepared any defense against them.

From these considerations, it seems that the probabilities associated with the board states in the current belief state can only be calculated given an optimal randomized strategy; in turn, computing that strategy seems to require knowing the probabilities of the various states the board might be in. This conundrum can be resolved by adopting the game-theoretic notion of an **equilibrium** solution, which we pursue further in Chapter 17. An equilibrium specifies an optimal randomized strategy for each player. Computing equilibria is prohibitively expensive, however, even for small games, and is out of the question for Kriegspiel. At present, the design of effective algorithms for general Kriegspiel play is an open research topic. Most systems perform bounded-depth lookahead in their own belief-state space, ignoring the opponent's belief state. Evaluation functions resemble those for the observable game but include a component for the size of the belief state—smaller is better!

5.6.2 Card games

Card games provide many examples of *stochastic* partial observability, where the missing information is generated randomly. For example, in many games, cards are dealt randomly at the beginning of the game, with each player receiving a hand 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 these 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! Even though this analogy turns out to be incorrect, it suggests an effective algorithm: consider all possible deals of the invisible cards; solve each one as if it were a fully observable game; and then choose the move that has the best outcome averaged over all the deals. Suppose that each deal s occurs with probability $P(s)$; then the move we want is

$$\operatorname*{argmax}_a \sum_s P(s)\, \textsc{Minimax}(\textsc{Result}(s,a)) \; . \tag{5.1}$$

Here, we run exact MINIMAX if computationally feasible; otherwise, we run H-MINIMAX.

Now, in most card games, the number of possible deals is rather large. For example, in bridge play, each player sees just two of the four hands; there are two unseen hands of 13 cards each, so the number of deals is $\binom{26}{13} = 10,400,600$. Solving even one deal is quite difficult, so solving ten million is out of the question. Instead, we resort to a Monte Carlo

approximation: instead of adding up *all* the deals, we take a *random sample* of N deals, where the probability of deal s appearing in the sample is proportional to $P(s)$:

$$\operatorname*{argmax}_a \frac{1}{N} \sum_{i=1}^{N} \textsc{Minimax}(\textsc{Result}(s_i, a)) \ . \tag{5.2}$$

(Notice that $P(s)$ does not appear explicitly in the summation, because the samples are already drawn according to $P(s)$.) As N grows large, the sum over the random sample tends to the exact value, but even for fairly small N—say, 100 to 1,000—the method gives a good approximation. It can also be applied to deterministic games such as Kriegspiel, given some reasonable estimate of $P(s)$.

For games like whist and hearts, where there is no bidding or betting phase before play commences, each deal will be equally likely and so the values of $P(s)$ are all equal. For bridge, play is preceded by a bidding phase in which each team indicates how many tricks it expects to win. Since players bid based on the cards they hold, the other players learn more about the probability of each deal. Taking this into account in deciding how to play the hand is tricky, for the reasons mentioned in our description of Kriegspiel: players may bid in such a way as to minimize the information conveyed to their opponents. Even so, the approach is quite effective for bridge, as we show in Section 5.7.

The strategy described in Equations 5.1 and 5.2 is sometimes called *averaging over clairvoyance* because it assumes that the game will become observable to both players immediately after the first move. Despite its intuitive appeal, the strategy can lead one astray. Consider the following story:

> Day 1: Road *A* leads to a heap of gold; Road *B* leads to a fork. Take the left fork and you'll find a bigger heap of gold, but take the right fork and you'll be run over by a bus.
> Day 2: Road *A* leads to a heap of gold; Road *B* leads to a fork. Take the right fork and you'll find a bigger heap of gold, but take the left fork and you'll be run over by a bus.
> Day 3: Road *A* leads to a heap of gold; Road *B* leads to a fork. One branch of the fork leads to a bigger heap of gold, but take the wrong fork and you'll be hit by a bus. Unfortunately you don't know which fork is which.

Averaging over clairvoyance leads to the following reasoning: on Day 1, *B* is the right choice; on Day 2, *B* is the right choice; on Day 3, the situation is the same as either Day 1 or Day 2, so *B* must still be the right choice.

Now we can see how averaging over clairvoyance fails: it does not consider the *belief state* that the agent will be in after acting. A belief state of total ignorance is not desirable, especially when one possibility is certain death. Because it assumes that every future state will automatically be one of perfect knowledge, the approach never selects actions that *gather information* (like the first move in Figure 5.13); nor will it choose actions that hide information from the opponent or provide information to a partner because it assumes that they already know the information; and it will never **bluff** in poker,[4] because it assumes the opponent can see its cards. In Chapter 17, we show how to construct algorithms that do all these things by virtue of solving the true partially observable decision problem.

BLUFF

[4] Bluffing—betting as if one's hand is good, even when it's not—is a core part of poker strategy.

5.7 STATE-OF-THE-ART GAME PROGRAMS

In 1965, the Russian mathematician Alexander Kronrod called chess "the *Drosophila* of artificial intelligence." John McCarthy disagrees: whereas geneticists use fruit flies to make discoveries that apply to biology more broadly, AI has used chess to do the equivalent of breeding very fast fruit flies. Perhaps a better analogy is that chess is to AI as Grand Prix motor racing is to the car industry: state-of-the-art game programs are blindingly fast, highly optimized machines that incorporate the latest engineering advances, but they aren't much use for doing the shopping or driving off-road. Nonetheless, racing and game-playing generate excitement and a steady stream of innovations that have been adopted by the wider community. In this section we look at what it takes to come out on top in various games.

CHESS

Chess: IBM's DEEP BLUE chess program, now retired, is well known for defeating world champion Garry Kasparov in a widely publicized exhibition match. Deep Blue ran on a parallel computer with 30 IBM RS/6000 processors doing alpha–beta search. The unique part was a configuration of 480 custom VLSI chess processors that performed move generation and move ordering for the last few levels of the tree, and evaluated the leaf nodes. Deep Blue searched up to 30 billion positions per move, reaching depth 14 routinely. The key to its success seems to have been its ability to generate singular 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 had the effect of substantially extending the effective search depth, allowing Deep Blue to play perfectly in some cases even when it was 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. But algorithmic improvements have allowed programs running on standard PCs to win World Computer Chess Championships. 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

HYDRA can be seen as the successor to DEEP BLUE. HYDRA runs on a 64-processor cluster with 1 gigabyte per processor and with custom hardware in the form of FPGA (Field Programmable Gate Array) chips. HYDRA reaches 200 million evaluations per second, about the same as Deep Blue, but HYDRA reaches 18 plies deep rather than just 14 because of aggressive use of the null move heuristic and forward pruning.

RYBKA, winner of the 2008 and 2009 World Computer Chess Championships, is considered the strongest current computer player. It uses an off-the-shelf 8-core 3.2 GHz Intel Xeon processor, but little is known about the design of the program. RYBKA's main advantage appears to be its evaluation function, which has been tuned by its main developer, International Master Vasik Rajlich, and at least three other grandmasters.

The most recent matches suggest that the top computer chess programs have pulled ahead of all human contenders. (See the historical notes for details.)

CHECKERS

Checkers: Jonathan Schaeffer and colleagues developed CHINOOK, which runs on regular PCs and uses alpha–beta search. Chinook defeated the long-running human champion in an abbreviated match in 1990, and since 2007 CHINOOK has been able to play perfectly by using alpha–beta search combined with a database of 39 trillion endgame positions.

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 5.5 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 reinforcement learning with neural networks 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 competitive with top human players. 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. Because the board is 19×19 and moves are allowed into (almost) every empty square, the branching factor starts at 361, which is too daunting for regular alpha–beta search methods. In addition, it is difficult to write an evaluation function because control of territory is often very unpredictable until the endgame. Therefore the top programs, such as MOGO, avoid alpha–beta search and instead use Monte Carlo rollouts. The trick is to decide what moves to make in the course of the rollout. There is no aggressive pruning; all moves are possible. The UCT (upper confidence bounds on trees) method works by making random moves in the first few iterations, and over time guiding the sampling process to prefer moves that have led to wins in previous samples. Some tricks are added, including *knowledge-based rules* that suggest particular moves whenever a given pattern is detected and *limited local search* to decide tactical questions. Some programs also include special techniques from **combinatorial game theory** to analyze endgames. These techniques decompose a position into sub-positions that can be analyzed separately and then combined (Berlekamp and Wolfe, 1994; Müller, 2003). The optimal solutions obtained in this way have surprised many professional Go players, who thought they had been playing optimally all along. Current Go programs play at the master level on a reduced 9×9 board, but are still at advanced amateur level on a full board.

COMBINATORIAL GAME THEORY

BRIDGE

Bridge is a card 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 in Section 5.6, optimal play in partially observable games like bridge can include elements of information gathering, communication, and careful weighing of probabilities. Many of these techniques are used in the Bridge Baron 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 11) involving high-level ideas, such as **finessing** and **squeezing**, that are familiar to bridge players.

EXPLANATION-BASED GENERALIZATION

The GIB program (Ginsberg, 1999) won the 2000 computer bridge championship quite decisively using the Monte Carlo method. Since then, other winning programs have followed GIB's lead. GIB's major innovation is using **explanation-based generalization** to compute and cache general rules for optimal play in various standard classes of situations rather than evaluating each situation individually. For example, in a situation where one player has the cards A-K-Q-J-4-3-2 of one suit and another player has 10-9-8-7-6-5, there are $7 \times 6 = 42$ ways that the first player can lead from that suit and the second player can follow. But GIB treats these situations as just two: the first player can lead either a high card or a low card; the exact cards played don't matter. With this optimization (and a few others), GIB can solve a 52-card, fully observable deal *exactly* in about a second. 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, not bidding) at the 1998 human world championship, far exceeding the expectations of many human experts.

There are several reasons why GIB plays at expert level with Monte Carlo simulation, whereas Kriegspiel programs do not. First, GIB's evaluation of the fully observable version of the game is exact, searching the full game tree, while Kriegspiel programs rely on inexact heuristics. But far more important is the fact that in bridge, most of the uncertainty in the partially observable information comes from the randomness of the deal, not from the adversarial play of the opponent. Monte Carlo simulation handles randomness well, but does not always handle strategy well, especially when the strategy involves the value of information.

SCRABBLE

Scrabble: Most people think the hard part about Scrabble is coming up with good words, but given the official dictionary, it turns out to be rather easy to program a move generator to find the highest-scoring move (Gordon, 1994). That doesn't mean the game is solved, however: merely taking the top-scoring move each turn results in a good but not expert player. The problem is that Scrabble is both partially observable and stochastic: you don't know what letters the other player has or what letters you will draw next. So playing Scrabble well combines the difficulties of backgammon and bridge. Nevertheless, in 2006, the QUACKLE program defeated the former world champion, David Boys, 3–2.

5.8 ALTERNATIVE APPROACHES

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 has

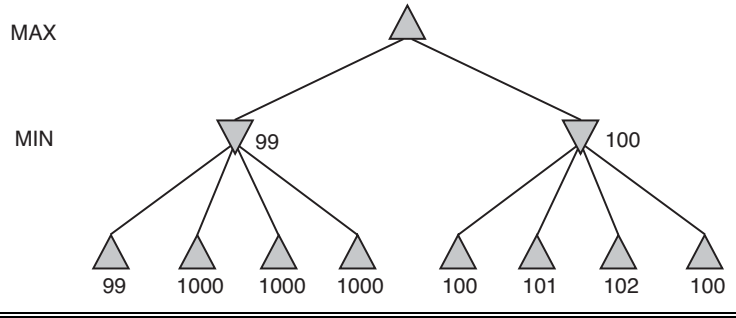


Figure 5.14 A two-ply game tree for which heuristic minimax may make an error.

been worked on for so long, the standard approach dominates other methods in tournament play. Some believe that this has caused game playing to become divorced from the mainstream of AI research: 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 heuristic minimax. It selects an optimal move in a given search tree *provided that the leaf node evaluations are exactly correct*. In reality, evaluations are usually crude estimates of the value of a position and can be considered to have large errors associated with them. Figure 5.14 shows a two-ply game tree for which minimax suggests taking the right-hand branch because $100 > 99$. That is the correct move if the evaluations are all correct. But of course the evaluation function is only approximate. Suppose that the evaluation of each node has an error that is independent of other nodes and is randomly distributed with mean zero and standard deviation of σ. Then when $\sigma = 5$, the left-hand branch is actually better 71% of the time, and 58% of the time when $\sigma = 2$. The intuition behind this is that the right-hand branch has four nodes that are close to 99; if an error in the evaluation of any one of the four makes the right-hand branch slip below 99, then the left-hand branch is better.

In reality, circumstances are actually worse than this because the error in the evaluation function is *not* independent. If we get one node wrong, the chances are high that nearby nodes in the tree will also be wrong. The fact that the node labeled 99 has siblings labeled 1000 suggests that in fact it might have a higher true value. We can use an evaluation function that returns a probability distribution over possible values, but it is difficult to combine these distributions properly, because we won't have a good model of the very strong dependencies that exist between the values of sibling nodes

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 alpha–beta algorithm 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 search tree, telling us that the only move is the best move and assigning it a value. But since we have to make the move anyway, knowing the move's value is useless. Similarly, if there is one obviously good move and several moves that are legal but lead to a quick loss, we

would not want alpha–beta to waste time determining a precise value for the lone good move. Better to just make the move quickly and save the time for later. 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 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 generate 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. 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 algorithms 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.

5.9 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, the **result** of each action, 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 by 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 a heuristic **evaluation function** that estimates the utility of a state.

- Many game programs precompute tables of best moves in the opening and endgame so that they can look up a move rather than search.
- 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, weighted by the probability of each child.
- Optimal play in games of **imperfect information**, such as Kriegspiel and 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 have bested even champion human players at games such as chess, checkers, and Othello. Humans retain the edge in several games of imperfect information, such as poker, bridge, and Kriegspiel, and in games with very large branching factors and little good heuristic knowledge, such as 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 did not understand the exponential complexity of search trees, claiming "the combinations involved in the Analytical Engine enormously surpassed any required, even by the game of chess." Babbage 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 traced to a 1912 paper by Ernst Zermelo, the developer of modern set theory. The paper unfortunately contained several errors and did not describe minimax correctly. On the other hand, it did lay out the ideas of retrograde analysis and proposed (but did not prove) what became known as Zermelo's theorem: that chess is determined—White can force a win or Black can or it is a draw; we just don't know which. Zermelo says that should we eventually know, "Chess would of course lose the character of a game at all." 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.

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–beta; it was the first chess program to do so. Alpha–beta pruning was described by Hart and Edwards (1961) and Hart *et al.* (1972). Alpha–beta was used by the "Kotok–McCarthy" chess program written by a student of John McCarthy (Kotok, 1962). Knuth and Moore (1975) proved the correctness of alpha–beta and analysed its time complexity. Pearl (1982b) shows alpha–beta to be asymptotically optimal among all fixed-depth game-tree search algorithms.

Several attempts have been made to overcome the problems with the "standard approach" that were outlined in Section 5.8. The first nonexhaustive heuristic 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 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. The expectiminimax algorithm was proposed by Donald Michie (1966). Bruce Ballard (1983) extended alpha–beta pruning to cover trees with chance nodes and Hauk (2004) reexamines this work and provides empirical results.

Koller and Pfeffer (1997) describe a system for completely solving partially observable games. The system is quite general, handling games whose optimal strategy requires randomized moves and games that are more complex than those handled by any previous system. Still, it can't handle games as complex as poker, bridge, and Kriegspiel. Frank *et al.* (1998) describe several variants of Monte Carlo search, including one where MIN has

complete information but MAX does not. Among deterministic, partially observable games, Kriegspiel has received the most attention. Ferguson demonstrated hand-derived randomized strategies for winning Kriegspiel with a bishop and knight (1992) or two bishops (1995) against a king. The first Kriegspiel programs concentrated on finding endgame checkmates and performed AND–OR search in belief-state space (Sakuta and Iida, 2002; Bolognesi and Ciancarini, 2003). Incremental belief-state algorithms enabled much more complex midgame checkmates to be found (Russell and Wolfe, 2005; Wolfe and Russell, 2007), but efficient state estimation remains the primary obstacle to effective general play (Parker *et al.*, 2005).

Chess was one of the first tasks undertaken in AI, with early efforts by many of the pioneers of computing, including Konrad Zuse in 1945, Norbert Wiener in his book *Cybernetics* (1948), and Alan Turing in 1950 (see Turing *et al.*, 1953). But it was Claude Shannon's article *Programming a Computer for Playing Chess* (1950) that had the most complete set of ideas, describing a representation for board positions, an evaluation function, quiescence search, and some ideas for selective (nonexhaustive) game-tree search. Slater (1950) and the commentators on his article also explored the possibilities for computer chess play.

D. G. Prinz (1952) completed a program that solved chess endgame problems but did not play a full game. Stan Ulam and a group at the Los Alamos National Lab produced a program that played chess on a 6×6 board with no bishops (Kister *et al.*, 1957). It could search 4 plies deep in about 12 minutes. Alex Bernstein wrote the first documented program to play a full game of standard chess (Bernstein and Roberts, 1958).[5]

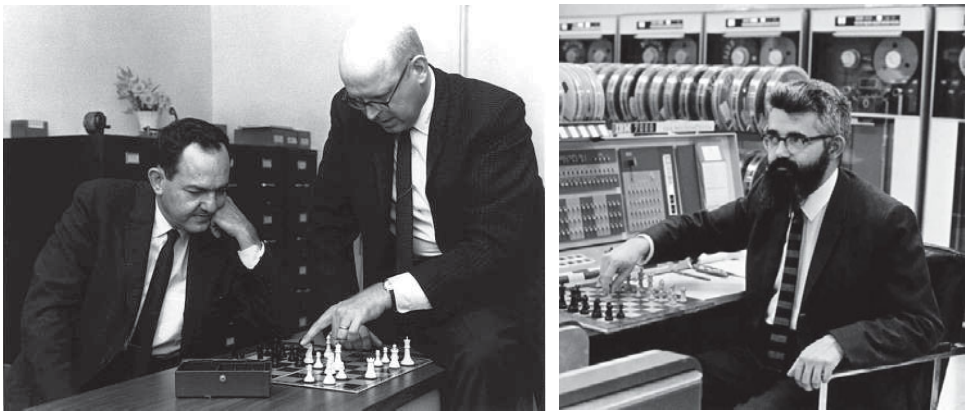
The first computer chess match featured the Kotok–McCarthy program from MIT (Kotok, 1962) and the ITEP program written in the mid-1960s at Moscow's Institute of Theoretical and Experimental Physics (Adelson-Velsky *et al.*, 1970). This intercontinental match was played by telegraph. It ended with a 3–1 victory for the ITEP program in 1967. The first chess program to compete successfully with humans was MIT's MACHACK-6 (Greenblatt *et al.*, 1967). Its Elo rating of approximately 1400 was well above the novice level of 1000.

The Fredkin Prize, established in 1980, offered awards for progressive milestones in chess play. The $5,000 prize for the first program to achieve a master rating went to BELLE (Condon and Thompson, 1982), which achieved a rating of 2250. The $10,000 prize for the first program to achieve a USCF (United States Chess Federation) rating of 2500 (near the grandmaster level) was awarded to DEEP THOUGHT (Hsu *et al.*, 1990) in 1989. The grand prize, $100,000, went to DEEP BLUE (Campbell *et al.*, 2002; Hsu, 2004) for its landmark victory over world champion Garry Kasparov in a 1997 exhibition match. 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)

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.

[5] A Russian program, BESM may have predated Bernstein's program.



(a) (b)

Figure 5.15 Pioneers in computer chess: (a) Herbert Simon and Allen Newell, developers of the NSS program (1958); (b) John McCarthy and the Kotok–McCarthy program on an IBM 7090 (1967).

In recent years, chess programs are pulling ahead of even the world's best humans. In 2004–2005 HYDRA defeated grand master Evgeny Vladimirov 3.5–0.5, world champion Ruslan Ponomariov 2–0, and seventh-ranked Michael Adams 5.5–0.5. In 2006, DEEP FRITZ beat world champion Vladimir Kramnik 4–2, and in 2007 RYBKA defeated several grand masters in games in which it gave odds (such as a pawn) to the human players. As of 2009, the highest Elo rating ever recorded was Kasparov's 2851. HYDRA (Donninger and Lorenz, 2004) is rated somewhere between 2850 and 3000, based mostly on its trouncing of Michael Adams. The RYBKA program is rated between 2900 and 3100, but this is based on a small number of games and is not considered reliable. Ross (2004) shows how human players have learned to exploit some of the weaknesses of the computer programs.

Checkers was the first of the classic games fully played by a computer. Christopher Strachey (1952) wrote the first working program for 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 (Samuel, 1959, 1967). 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. In 1962 it defeated Robert Nealy, a champion at "blind checkers," through an error on his part. 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.

The challenge started by Samuel was taken up by Jonathan Schaeffer of the University of Alberta. His CHINOOK program 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. A rematch at the 1994 world championship ended prematurely when Tinsley had to withdraw for health reasons. CHINOOK became the official world champion. Schaeffer kept on building on his database of endgames, and in 2007 "solved" checkers (Schaeffer *et al.*, 2007; Schaeffer, 2008). This had been predicted by Richard Bellman (1965). In the paper that introduced the dynamic programming approach to retrograde analysis, he wrote, "In checkers, the number of possible moves in any given situation is so small that we can confidently expect a complete digital computer solution to the problem of optimal play in this game." Bellman did not, however, fully appreciate the size of the checkers game tree. There are about 500 quadrillion positions. After 18 years of computation on a cluster of 50 or more machines, Jonathan Schaeffer's team completed an endgame table for all checkers positions with 10 or fewer pieces: over 39 trillion entries. From there, they were able to do forward alpha–beta search to derive a policy that proves that checkers is in fact a draw with best play by both sides. Note that this is an application of bidirectional search (Section 3.4.6). Building an endgame table for all of checkers would be impractical: it would require a billion gigabytes of storage. Searching without any table would also be impractical: the search tree has about 8^{47} positions, and would take thousands of years to search with today's technology. Only a combination of clever search, endgame data, and a drop in the price of processors and memory could solve checkers. Thus, checkers joins Qubic (Patashnik, 1980), Connect Four (Allis, 1988), and Nine-Men's Morris (Gasser, 1998) as games that have been solved by computer analysis.

Backgammon, a game of chance, was analyzed mathematically by Gerolamo Cardano (1663), but only taken up for computer play in the late 1970s, first with the BKG program (Berliner, 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 BKG was very lucky with the dice. Gerry Tesauro's (1995) TD-GAMMON played consistently at world champion level. The BGBLITZ program was the winner of the 2008 Computer Olympiad.

Go is a deterministic game, but the large branching factor makes it challeging. The key issues and early literature in computer Go are summarized by Bouzy and Cazenave (2001) and Müller (2002). Up to 1997 there were no competent Go programs. Now the best programs play *most* of their moves at the master level; the only problem is that over the course of a game they usually make at least one serious blunder that allows a strong opponent to win. Whereas alpha–beta search reigns in most games, many recent Go programs have adopted Monte Carlo methods based on the UCT (upper confidence bounds on trees) scheme (Kocsis and Szepesvari, 2006). The strongest Go program as of 2009 is Gelly and Silver's MOGO (Wang and Gelly, 2007; Gelly and Silver, 2008). In August 2008, MOGO scored a surprising win against top professional Myungwan Kim, albeit with MOGO receiving a handicap of nine stones (about the equivalent of a queen handicap in chess). Kim estimated MOGO's strength at 2–3 dan, the low end of advanced amateur. For this match, MOGO was run on an 800-processor 15 teraflop supercomputer (1000 times Deep Blue). A few weeks later, MOGO, with only a five-stone handicap, won against a 6-dan professional. In the 9×9 form of Go, MOGO is at approximately the 1-dan professional level. Rapid advances are likely as experimentation continues with new forms of Monte Carlo search. The *Computer Go*

Newsletter, published by the Computer Go Association, describes current developments.

Bridge: Smith *et al.* (1998) report on how their planning-based program won the 1998 computer bridge championship, and (Ginsberg, 2001) describes how his GIB program, based on Monte Carlo simulation, won the following computer championship and did surprisingly well against human players and standard book problem sets. From 2001–2007, the computer bridge championship was won five times by JACK and twice by WBRIDGE5. Neither has had academic articles explaining their structure, but both are rumored to use the Monte Carlo technique, which was first proposed for bridge by Levy (1989).

Scrabble: A good description of a top program, MAVEN, is given by its creator, Brian Sheppard (2002). Generating the highest-scoring move is described by Gordon (1994), and modeling opponents is covered by Richards and Amir (2007).

Soccer (Kitano *et al.*, 1997b; Visser *et al.*, 2008) and **billiards** (Lam and Greenspan, 2008; Archibald *et al.*, 2009) and other stochastic games with a continuous space of actions are beginning to attract attention in AI, both in simulation and with physical robot players.

Computer game competitions occur annually, and papers 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. The General Game Competition (Love *et al.*, 2006) tests programs that must learn to play an unknown game given only a logical description of the rules of the game. 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 *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, and other games. Since 1998, a biennial *Computers and Games* conference has been held.

EXERCISES

5.1 Suppose you have an oracle, $OM(s)$, that correctly predicts the opponent's move in any state. Using this, formulate the definition of a game as a (single-agent) search problem. Describe an algorithm for finding the optimal move.

5.2 Consider the problem of solving two 8-puzzles.

a. Give a complete problem formulation in the style of Chapter 3.

b. How large is the reachable state space? Give an exact numerical expression.

c. Suppose we make the problem adversarial as follows: the two players take turns moving; a coin is flipped to determine the puzzle on which to make a move in that turn; and the winner is the first to solve one puzzle. Which algorithm can be used to choose a move in this setting?

d. Give an informal proof that someone will eventually win if both play perfectly.

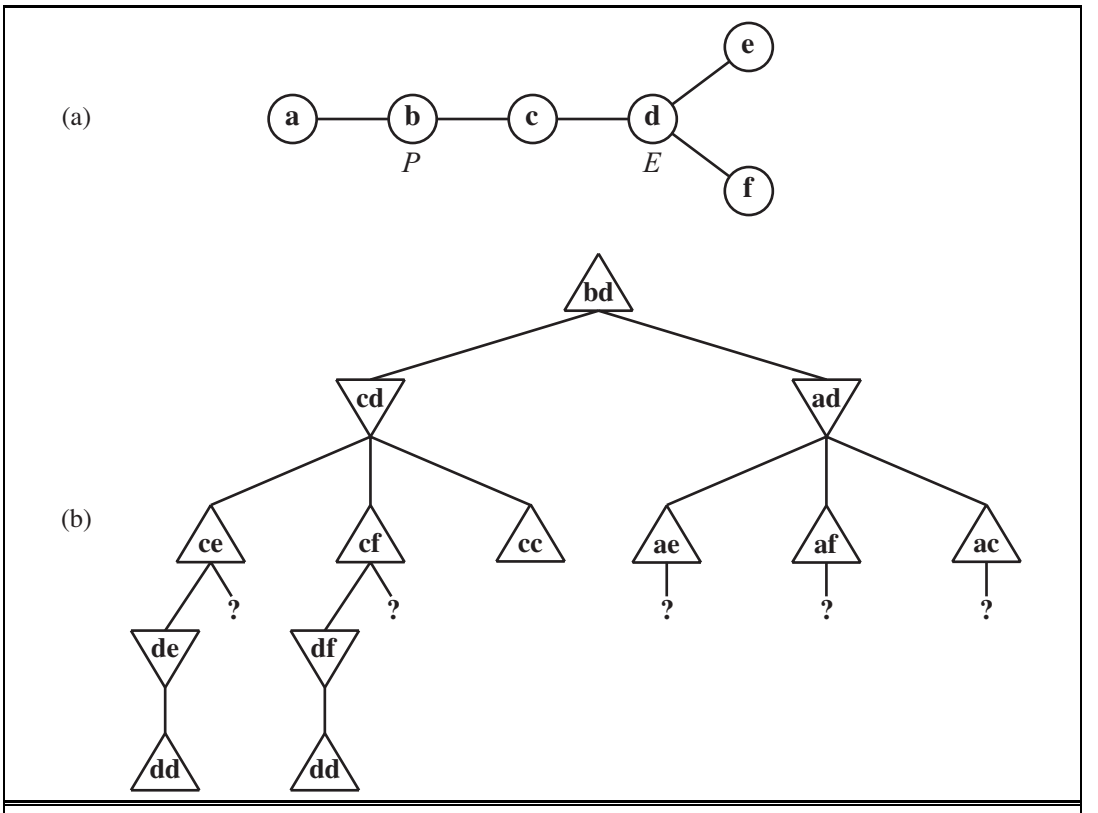


Figure 5.16 (a) A map where the cost of every edge is 1. Initially the pursuer P is at node **b** and the evader E is at node **d**. (b) A partial game tree for this map. Each node is labeled with the P, E positions. P moves first. Branches marked "?" have yet to be explored.

PURSUIT–EVASION

5.3 Imagine that, in Exercise 3.3, one of the friends wants to avoid the other. The problem then becomes a two-player **pursuit–evasion** game. We assume now that the players take turns moving. The game ends only when the players are on the same node; the terminal payoff to the pursuer is minus the total time taken. (The evader "wins" by never losing.) An example is shown in Figure 5.16.

a. Copy the game tree and mark the values of the terminal nodes.

b. Next to each internal node, write the strongest fact you can infer about its value (a number, one or more inequalities such as "≥ 14", or a "?").

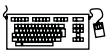
c. Beneath each question mark, write the name of the node reached by that branch.

d. Explain how a bound on the value of the nodes in (c) can be derived from consideration of shortest-path lengths on the map, and derive such bounds for these nodes. Remember the cost to get to each leaf as well as the cost to solve it.

e. Now suppose that the tree as given, with the leaf bounds from (d), is evaluated from left to right. Circle those "?" nodes that would *not* need to be expanded further, given the bounds from part (d), and cross out those that need not be considered at all.

f. Can you prove anything in general about who wins the game on a map that is a tree?

5.4 Describe and implement state descriptions, move generators, terminal tests, utility functions, and evaluation functions for one or more of the following stochastic games: Monopoly, Scrabble, bridge play with a given contract, or Texas hold'em poker.



5.5 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.

5.6 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.

5.7 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?

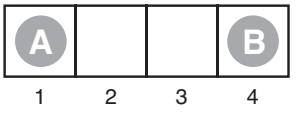


Figure 5.17 The starting position of a simple game. Player A moves first. The two players take turns moving, and each player must move his token to an open adjacent space in either direction. If the opponent occupies an adjacent space, then a player may jump over the opponent to the next open space if 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.

5.8 Consider the two-player game described in Figure 5.17.

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 box 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 their value is unclear, 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.

5.9 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.

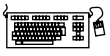
5.10 Consider the family of generalized tic-tac-toe games, defined as follows. Each particular game is specified by a set $\mathcal{S}$ of *squares* and a collection $\mathcal{W}$ of *winning positions*. Each winning position is a subset of $\mathcal{S}$. For example, in standard tic-tac-toe, $\mathcal{S}$ is a set of 9 squares and $\mathcal{W}$ is a collection of 8 subsets of $\mathcal{W}$: the three rows, the three columns, and the two diagonals. In other respects, the game is identical to standard tic-tac-toe. Starting from an empty board, players alternate placing their marks on an empty square. A player who marks every square in a winning position wins the game. It is a tie if all squares are marked and neither player has won.

a. Let $N = |\mathcal{S}|$, the number of squares. Give an upper bound on the number of nodes in the complete game tree for generalized tic-tac-toe as a function of N.

b. Give a lower bound on the size of the game tree for the worst case, where $\mathcal{W} = \{\}$.

c. Propose a plausible evaluation function that can be used for any instance of generalized tic-tac-toe. The function may depend on $\mathcal{S}$ and $\mathcal{W}$.

d. Assume that it is possible to generate a new board and check whether it is a winning position in $100N$ machine instructions and assume a 2 gigahertz processor. Ignore memory limitations. Using your estimate in (a), roughly how large a game tree can be completely solved by alpha–beta in a second of CPU time? a minute? an hour?



5.11 Develop a general game-playing program, capable of playing a variety of games.

a. Implement move generators and evaluation functions for one or more of the following games: Kalah, Othello, checkers, and chess.

b. Construct a general alpha–beta game-playing agent.

c. 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?

d. Implement a selective search algorithm, such as B* (Berliner, 1979), conspiracy number search (McAllester, 1988), or MGSS* (Russell and Wefald, 1989) and compare its performance to A*.

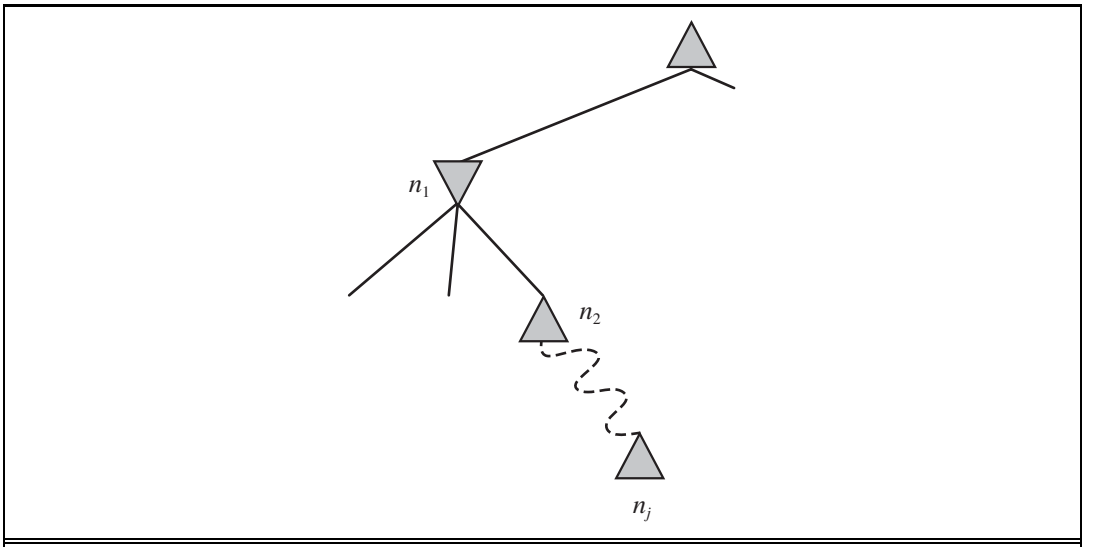


Figure 5.18 Situation when considering whether to prune node n_j.

5.12 Describe how the minimax and alpha–beta algorithms change for two-player, non-zero-sum games in which each player has a distinct utility function and both utility functions are known to both players. If there are no constraints on the two terminal utilities, is it possible for any node to be pruned by alpha–beta? What if the player's utility functions on any state differ by at most a constant k, making the game almost cooperative?

5.13 Develop a formal proof of correctness for alpha–beta pruning. To do this, consider the situation shown in Figure 5.18. 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. Mode n_1 takes on the minimum value among its children: $n_1 = \min(n_2, n_{21}, \ldots, n_{2b_2})$. 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 to 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.

5.14 Prove that alpha–beta pruning takes time $O(2^{m/2})$ with optimal move ordering, where m is the maximum depth of the game tree.

5.15 Suppose you have a chess program that can evaluate 10 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 2-gigabyte in-memory table? Will that be enough for the

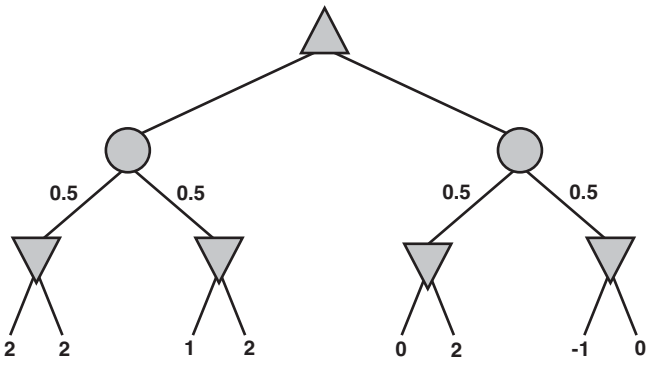


Figure 5.19 The complete game tree for a trivial game with chance nodes.

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 stored on disk. About how many evaluations could you do in the time it takes to do one disk seek with standard disk hardware?

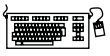
5.16 This question considers pruning in games with chance nodes. Figure 5.19 shows the complete game tree for a trivial game. Assume that the leaf nodes are to be evaluated in left-to-right order, and that before a leaf node is evaluated, we know nothing about its value—the range of possible values is $-\infty$ to ∞.

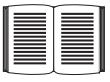
a. Copy the figure, mark the value of all the internal nodes, and indicate the best move at the root with an arrow.

b. Given the values of the first six leaves, do we need to evaluate the seventh and eighth leaves? Given the values of the first seven leaves, do we need to evaluate the eighth leaf? Explain your answers.

c. Suppose the leaf node values are known to lie between –2 and 2 inclusive. After the first two leaves are evaluated, what is the value range for the left-hand chance node?

d. Circle all the leaves that need not be evaluated under the assumption in (c).





5.17 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.

5.18 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.

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

- Generate some dice-roll sequences (say, 50) down to a suitable depth (say, 8).
- With known dice rolls, the game tree becomes deterministic. For each dice-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 (or why not)?

5.20 In the following, a "max" tree consists only of max nodes, whereas an "expectimax" tree consists of a max node at the root with alternating layers of chance and max nodes. At chance nodes, all outcome probabilities are nonzero. The goal is to *find the value of the root* with a bounded-depth search. For each of (a)–(f), either give an example or explain why this is impossible.

a. Assuming that leaf values are finite but unbounded, is pruning (as in alpha–beta) ever possible in a max tree?

b. Is pruning ever possible in an expectimax tree under the same conditions?

c. If leaf values are all nonnegative, is pruning ever possible in a max tree? Give an example, or explain why not.

d. If leaf values are all nonnegative, is pruning ever possible in an expectimax tree? Give an example, or explain why not.

e. If leaf values are all in the range $[0, 1]$, is pruning ever possible in a max tree? Give an example, or explain why not.

f. If leaf values are all in the range $[0, 1]$, is pruning ever possible in an expectimax tree?

g. Consider the outcomes of a chance node in an expectimax tree. Which of the following evaluation orders is most likely to yield pruning opportunities?

(i) Lowest probability first
(ii) Highest probability first
(iii) Doesn't make any difference

5.21 Which of the following are true and which are false? Give brief explanations.

a. In a fully observable, turn-taking, zero-sum game between two perfectly rational players, it does not help the first player to know what strategy the second player is using—that is, what move the second player will make, given the first player's move.

b. In a partially observable, turn-taking, zero-sum game between two perfectly rational players, it does not help the first player to know what move the second player will make, given the first player's move.

c. A perfectly rational backgammon agent never loses.

5.22 Consider carefully the interplay of chance events and partial information in each of the games in Exercise 5.4.

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 5.19 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 CONSTRAINT SATISFACTION PROBLEMS

In which we see how treating states as more than just little black boxes leads to the invention of a range of powerful new search methods and a deeper understanding of problem structure and complexity.

Chapters 3 and 4 explored the idea that problems can be solved by searching in a space of **states**. These states can be evaluated by domain-specific heuristics and tested to see whether they are goal states. From the point of view of the search algorithm, however, each state is atomic, or indivisible—a black box with no internal structure.

This chapter describes a way to solve a wide variety of problems more efficiently. We use a **factored representation** for each state: a set of variables, each of which has a value. A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a **constraint satisfaction problem**, or CSP.

CONSTRAINT SATISFACTION PROBLEM

CSP search algorithms take advantage of the structure of states and use *general-purpose* rather than *problem-specific* heuristics to enable the solution of complex problems. The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.

6.1 DEFINING CONSTRAINT SATISFACTION PROBLEMS

A constraint satisfaction problem consists of three components, X, D, and C:

X is a set of variables, $\{X_1, \ldots, X_n\}$.

D is a set of domains, $\{D_1, \ldots, D_n\}$, one for each variable.

C is a set of constraints that specify allowable combinations of values.

Each domain D_i consists of a set of allowable values, $\{v_1, \ldots, v_k\}$ for variable X_i. Each constraint C_i consists of a pair $\langle scope, rel \rangle$, where $scope$ is a tuple of variables that participate in the constraint and rel is a relation that defines the values that those variables can take on. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation. For example, if X_1 and X_2 both have

the domain {A,B}, then the constraint saying the two variables must have different values can be written as $\langle (X_1, X_2), [(A, B), (B, A)] \rangle$ or as $\langle (X_1, X_2), X_1 \neq X_2 \rangle$.

ASSIGNMENT
CONSISTENT
COMPLETE ASSIGNMENT
SOLUTION
PARTIAL ASSIGNMENT

To solve a CSP, we need to define a state space and the notion of a solution. Each state in a CSP is defined by an **assignment** of values to some or all of the variables, $\{X_i = v_i, X_j = v_j, \ldots\}$. An assignment that does not violate any constraints is called a **consistent** or legal assignment. A **complete assignment** is one in which every variable is assigned, and a **solution** to a CSP is a consistent, complete assignment. A **partial assignment** is one that assigns values to only some of the variables.

6.1.1 Example problem: Map coloring

Suppose that, having tired of Romania, we are looking at a map of Australia showing each of its states and territories (Figure 6.1(a)). We are given the task of coloring each region either red, green, or blue in such a way that no neighboring regions have the same color. To formulate this as a CSP, we define the variables to be the regions

$$X = \{ WA, NT, Q, NSW, V, SA, T \} .$$

The domain of each variable is the set $D_i = \{red, green, blue\}$. The constraints require neighboring regions to have distinct colors. Since there are nine places where regions border, there are nine constraints:

$$\begin{aligned} C = \{ & SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, \\ & WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V \} . \end{aligned}$$

Here we are using abbreviations; $SA \neq WA$ is a shortcut for $\langle (SA, WA), SA \neq WA \rangle$, where $SA \neq WA$ can be fully enumerated in turn as

$$\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\} .$$

There are many possible solutions to this problem, such as

$$\{ WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red \}.$$

CONSTRAINT GRAPH

It can be helpful to visualize a CSP as a **constraint graph**, as shown in Figure 6.1(b). The nodes of the graph correspond to variables of the problem, and a link connects any two variables that participate in a constraint.

Why formulate a problem as a CSP? One reason is that the CSPs yield a natural representation for a wide variety of problems; if you already have a CSP-solving system, it is often easier to solve a problem using it than to design a custom solution using another search technique. In addition, CSP solvers can be faster than state-space searchers because the CSP solver can quickly eliminate large swatches of the search space. For example, once we have chosen $\{SA = blue\}$ in the Australia problem, we can conclude that none of the five neighboring variables can take on the value $blue$. Without taking advantage of constraint propagation, a search procedure would have to consider $3^5 = 243$ assignments for the five neighboring variables; with constraint propagation we never have to consider $blue$ as a value, so we have only $2^5 = 32$ assignments to look at, a reduction of 87%.

In regular state-space search we can only ask: is this specific state a goal? No? What about this one? With CSPs, once we find out that a partial assignment is not a solution, we can

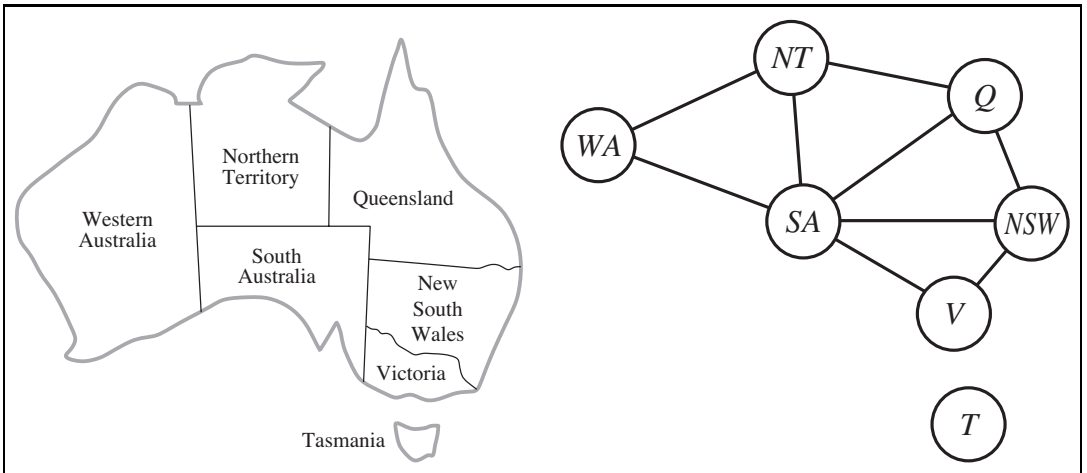


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

immediately discard further refinements of the partial assignment. Furthermore, we can see *why* the assignment is not a solution—we see which variables violate a constraint—so we can focus attention on the variables that matter. As a result, many problems that are intractable for regular state-space search can be solved quickly when formulated as a CSP.

6.1.2 Example problem: Job-shop scheduling

Factories have the problem of scheduling a day's worth of jobs, subject to various constraints. In practice, many of these problems are solved with CSP techniques. Consider the problem of scheduling the assembly of a car. The whole job is composed of tasks, and we can model each task as a variable, where the value of each variable is the time that the task starts, expressed as an integer number of minutes. Constraints can assert that one task must occur before another—for example, a wheel must be installed before the hubcap is put on—and that only so many tasks can go on at once. Constraints can also specify that a task takes a certain amount of time to complete.

We consider a small part of the car assembly, consisting of 15 tasks: install axles (front and back), affix all four wheels (right and left, front and back), tighten nuts for each wheel, affix hubcaps, and inspect the final assembly. We can represent the tasks with 15 variables:

$$X = \{Axle_F, Axle_B, Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LB}, Nuts_{RF}, Nuts_{LF}, Nuts_{RB}, Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect\} .$$

PRECEDENCE CONSTRAINTS

The value of each variable is the time that the task starts. Next we represent **precedence constraints** between individual tasks. Whenever a task T_1 must occur before task T_2, and task T_1 takes duration d_1 to complete, we add an arithmetic constraint of the form

$$T_1 + d_1 \leq T_2 .$$

In our example, the axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle, so we write

$$Axle_F + 10 \leq Wheel_{RF}; \quad Axle_F + 10 \leq Wheel_{LF};$$
$$Axle_B + 10 \leq Wheel_{RB}; \quad Axle_B + 10 \leq Wheel_{LB} .$$

Next we say that, for each wheel, we must affix the wheel (which takes 1 minute), then tighten the nuts (2 minutes), and finally attach the hubcap (1 minute, but not represented yet):

$$Wheel_{RF} + 1 \leq Nuts_{RF}; \quad Nuts_{RF} + 2 \leq Cap_{RF};$$
$$Wheel_{LF} + 1 \leq Nuts_{LF}; \quad Nuts_{LF} + 2 \leq Cap_{LF};$$
$$Wheel_{RB} + 1 \leq Nuts_{RB}; \quad Nuts_{RB} + 2 \leq Cap_{RB};$$
$$Wheel_{LB} + 1 \leq Nuts_{LB}; \quad Nuts_{LB} + 2 \leq Cap_{LB} .$$

DISJUNCTIVE CONSTRAINT

Suppose we have four workers to install wheels, but they have to share one tool that helps put the axle in place. We need a **disjunctive constraint** to say that $Axle_F$ and $Axle_B$ must not overlap in time; either one comes first or the other does:

$$(Axle_F + 10 \leq Axle_B) \quad \textbf{or} \quad (Axle_B + 10 \leq Axle_F) .$$

This looks like a more complicated constraint, combining arithmetic and logic. But it still reduces to a set of pairs of values that $Axle_F$ and $Axle_F$ can take on.

We also need to assert that the inspection comes last and takes 3 minutes. For every variable except $Inspect$ we add a constraint of the form $X + d_X \leq Inspect$. Finally, suppose there is a requirement to get the whole assembly done in 30 minutes. We can achieve that by limiting the domain of all variables:

$$D_i = \{1, 2, 3, \ldots, 27\} .$$

This particular problem is trivial to solve, but CSPs have been applied to job-shop scheduling problems like this with thousands of variables. In some cases, there are complicated constraints that are difficult to specify in the CSP formalism, and more advanced planning techniques are used, as discussed in Chapter 11.

6.1.3 Variations on the CSP formalism

DISCRETE DOMAIN
FINITE DOMAIN

The simplest kind of CSP involves variables that have **discrete**, **finite domains**. Map-coloring problems and scheduling with time limits are both of this kind. The 8-queens problem described in Chapter 3 can also be viewed as a finite-domain CSP, where the variables $Q_1, \ldots, Q_8$ are the positions of each queen in columns $1, \ldots, 8$ and each variable has the domain $D_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

INFINITE

A discrete domain can be **infinite**, such as the set of integers or strings. (If we didn't put a deadline on the job-scheduling problem, there would be an infinite number of start times for each variable.) With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combinations of values. Instead, a **constraint language** must be used that understands constraints such as $T_1 + d_1 \leq T_2$ directly, without enumerating the set of pairs of allowable values for (T_1, T_2). Special solution algorithms (which we do not discuss here) exist for **linear constraints** on integer variables—that is, constraints, such as the one just given, in which each variable appears only in linear form. It can be shown that no algorithm exists for solving general **nonlinear constraints** on integer variables.

CONSTRAINT LANGUAGE
LINEAR CONSTRAINTS
NONLINEAR CONSTRAINTS

CONTINUOUS DOMAINS

Constraint satisfaction problems with **continuous domains** are common in the real world and are widely studied in the field of operations research. For example, the scheduling of experiments on the Hubble Space Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical, precedence, and power constraints. The best-known category of continuous-domain CSPs is that of **linear programming** problems, where constraints must be linear equalities or inequalities. Linear programming problems can be solved in time polynomial in the number of variables. Problems with different types of constraints and objective functions have also been studied—quadratic programming, second-order conic programming, and so on.

UNARY CONSTRAINT

In addition to examining the types of variables that can appear in CSPs, it is useful to look at the types of constraints. The simplest type is the **unary constraint**, which restricts the value of a single variable. For example, in the map-coloring problem it could be the case that South Australians won't tolerate the color green; we can express that with the unary constraint $\langle(SA), SA \neq green\rangle$

BINARY CONSTRAINT

A **binary constraint** relates two variables. For example, $SA \neq NSW$ is a binary constraint. A binary CSP is one with only binary constraints; it can be represented as a constraint graph, as in Figure 6.1(b).

We can also describe higher-order constraints, such as asserting that the value of Y is between X and Z, with the ternary constraint $Between(X, Y, Z)$.

GLOBAL CONSTRAINT

A constraint involving an arbitrary number of variables is called a **global constraint**. (The name is traditional but confusing because it need not involve *all* the variables in a problem). One of the most common global constraints is $Alldiff$, which says that all of the variables involved in the constraint must have different values. In Sudoku problems (see Section 6.2.6), all variables in a row or column must satisfy an $Alldiff$ constraint. Another example is provided by **cryptarithmetic** puzzles. (See Figure 6.2(a).) Each letter in a cryptarithmetic puzzle represents a different digit. For the case in Figure 6.2(a), this would be represented as the global constraint $Alldiff(F, T, U, W, R, O)$. The addition constraints on the four columns of the puzzle can be written as the following n-ary constraints:

CRYPTARITHMETIC

$$
\begin{aligned}
&O + O = R + 10 \cdot C_{10} \\
&C_{10} + W + W = U + 10 \cdot C_{100} \\
&C_{100} + T + T = O + 10 \cdot C_{1000} \\
&C_{1000} = F \ ,
\end{aligned}
$$

where C_{10}, C_{100}, and C_{1000} are auxiliary variables representing the digit carried over into the tens, hundreds, or thousands column. These constraints can be represented in a **constraint hypergraph**, such as the one shown in Figure 6.2(b). A hypergraph consists of ordinary nodes (the circles in the figure) and hypernodes (the squares), which represent n-ary constraints.

CONSTRAINT HYPERGRAPH

Alternatively, as Exercise 6.6 asks you to prove, every finite-domain constraint can be reduced to a set of binary constraints if enough auxiliary variables are introduced, so we could transform any CSP into one with only binary constraints; this makes the algorithms simpler.

DUAL GRAPH

Another way to convert an n-ary CSP to a binary one is the **dual graph** transformation: create a new graph in which there will be one variable for each constraint in the original graph, and

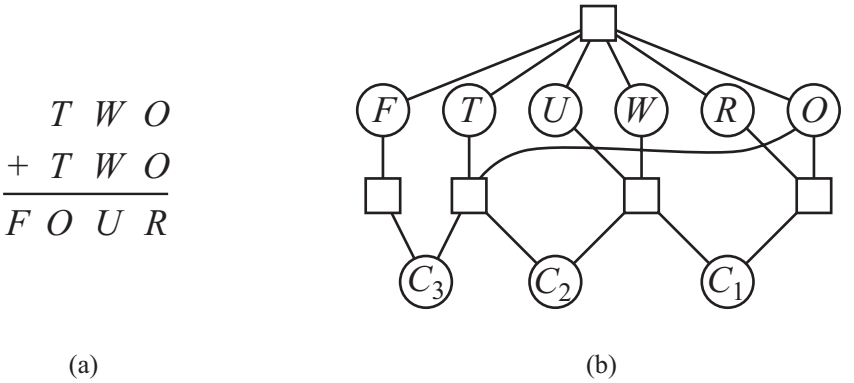


Figure 6.2 (a) A cryptarithmetic problem. Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed. (b) The constraint hypergraph for the cryptarithmetic problem, showing the *Alldiff* constraint (square box at the top) as well as the column addition constraints (four square boxes in the middle). The variables C_1, C_2, and C_3 represent the carry digits for the three columns.

one binary constraint for each pair of constraints in the original graph that share variables. For example, if the original graph has variables $\{X, Y, Z\}$ and constraints $\langle (X, Y, Z), C_1 \rangle$ and $\langle (X, Y), C_2 \rangle$ then the dual graph would have variables $\{C_1, C_2\}$ with the binary constraint $\langle (X, Y), R_1 \rangle$, where (X, Y) are the shared variables and R_1 is a new relation that defines the constraint between the shared variables, as specified by the original C_1 and C_2.

There are however two reasons why we might prefer a global constraint such as *Alldiff* rather than a set of binary constraints. First, it is easier and less error-prone to write the problem description using *Alldiff*. Second, it is possible to design special-purpose inference algorithms for global constraints that are not available for a set of more primitive constraints. We describe these inference algorithms in Section 6.2.5.

PREFERENCE CONSTRAINTS

The constraints we have described so far have all been absolute constraints, violation of which rules out a potential solution. Many real-world CSPs include **preference constraints** indicating which solutions are preferred. For example, in a university class-scheduling problem there are absolute constraints that no professor can teach two classes at the same time. But we also may allow preference constraints: Prof. R might prefer teaching in the morning, whereas Prof. N prefers teaching in the afternoon. A schedule that has Prof. R teaching at 2 p.m. would still be an allowable solution (unless Prof. R happens to be the department chair) but would not be an optimal one. Preference constraints can often be encoded as costs on individual variable assignments—for example, assigning an afternoon slot for Prof. R costs 2 points against the overall objective function, whereas a morning slot costs 1. With this formulation, CSPs with preferences can be solved with optimization search methods, either path-based or local. We call such a problem a **constraint optimization problem**, or COP. Linear programming problems do this kind of optimization.

CONSTRAINT OPTIMIZATION PROBLEM

6.2 CONSTRAINT PROPAGATION: INFERENCE IN CSPS

In regular state-space search, an algorithm can do only one thing: search. In CSPs there is a choice: an algorithm can search (choose a new variable assignment from several possibilities) or do a specific type of **inference** called **constraint propagation**: using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on. Constraint propagation may be intertwined with search, or it may be done as a preprocessing step, before search starts. Sometimes this preprocessing can solve the whole problem, so no search is required at all.

INFERENCE
CONSTRAINT PROPAGATION

LOCAL CONSISTENCY

The key idea is **local consistency**. If we treat each variable as a node in a graph (see Figure 6.1(b)) and each binary constraint as an arc, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph. There are different types of local consistency, which we now cover in turn.

6.2.1 Node consistency

NODE CONSISTENCY

A single variable (corresponding to a node in the CSP network) is **node-consistent** if all the values in the variable's domain satisfy the variable's unary constraints. For example, in the variant of the Australia map-coloring problem (Figure 6.1) where South Australians dislike green, the variable SA starts with domain $\{red, green, blue\}$, and we can make it node consistent by eliminating $green$, leaving SA with the reduced domain $\{red, blue\}$. We say that a network is node-consistent if every variable in the network is node-consistent.

It is always possible to eliminate all the unary constraints in a CSP by running node consistency. It is also possible to transform all n-ary constraints into binary ones (see Exercise 6.6). Because of this, it is common to define CSP solvers that work with only binary constraints; we make that assumption for the rest of this chapter, except where noted.

6.2.2 Arc consistency

ARC CONSISTENCY

A variable in a CSP is **arc-consistent** if every value in its domain satisfies the variable's binary constraints. More formally, X_i is arc-consistent with respect to another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j). A network is arc-consistent if every variable is arc consistent with every other variable. For example, consider the constraint $Y = X^2$ where the domain of both X and Y is the set of digits. We can write this constraint explicitly as

$$\langle (X, Y), \{(0, 0), (1, 1), (2, 4), (3, 9))\} \rangle \ .$$

To make X arc-consistent with respect to Y, we reduce X's domain to $\{0, 1, 2, 3\}$. If we also make Y arc-consistent with respect to X, then Y's domain becomes $\{0, 1, 4, 9\}$ and the whole CSP is arc-consistent.

On the other hand, arc consistency can do nothing for the Australia map-coloring problem. Consider the following inequality constraint on (SA, WA):

$$\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\} \ .$$

```
function AC-3( csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true
```

```
function REVISE( csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

Figure 6.3 The arc-consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved. The name "AC-3" was used by the algorithm's inventor (Mackworth, 1977) because it's the third version developed in the paper.

No matter what value you choose for SA (or for WA), there is a valid value for the other variable. So applying arc consistency has no effect on the domains of either variable.

The most popular algorithm for arc consistency is called AC-3 (see Figure 6.3). To make every variable arc-consistent, the AC-3 algorithm maintains a queue of arcs to consider. (Actually, the order of consideration is not important, so the data structure is really a set, but tradition calls it a queue.) Initially, the queue contains all the arcs in the CSP. AC-3 then pops off an arbitrary arc (X_i, X_j) from the queue and makes X_i arc-consistent with respect to X_j. If this leaves D_i unchanged, the algorithm just moves on to the next arc. But if this revises D_i (makes the domain smaller), then we add to the queue all arcs (X_k, X_i) where X_k is a neighbor of X_i. We need to do that because the change in D_i might enable further reductions in the domains of D_k, even if we have previously considered X_k. If D_i is revised down to nothing, then we know the whole CSP has no consistent solution, and AC-3 can immediately return failure. Otherwise, we keep checking, trying to remove values from the domains of variables until no more arcs are in the queue. At that point, we are left with a CSP that is equivalent to the original CSP—they both have the same solutions—but the arc-consistent CSP will in most cases be faster to search because its variables have smaller domains.

The complexity of AC-3 can be analyzed as follows. Assume a CSP with n variables, each with domain size at most d, and with c binary constraints (arcs). Each arc (X_k, X_i) can be inserted in the queue only d times because X_i has at most d values to delete. Checking

consistency of an arc can be done in $O(d^2)$ time, so we get $O(cd^3)$ total worst-case time.[1]

It is possible to extend the notion of arc consistency to handle n-ary rather than just binary constraints; this is called generalized arc consistency or sometimes hyperarc consistency, depending on the author. A variable X_i is **generalized arc consistent** with respect to an n-ary constraint if for every value v in the domain of X_i there exists a tuple of values that is a member of the constraint, has all its values taken from the domains of the corresponding variables, and has its X_i component equal to v. For example, if all variables have the domain $\{0, 1, 2, 3\}$, then to make the variable X consistent with the constraint $X < Y < Z$, we would have to eliminate 2 and 3 from the domain of X because the constraint cannot be satisfied when X is 2 or 3.

GENERALIZED ARC CONSISTENT

6.2.3 Path consistency

Arc consistency can go a long way toward reducing the domains of variables, sometimes finding a solution (by reducing every domain to size 1) and sometimes finding that the CSP cannot be solved (by reducing some domain to size 0). But for other networks, arc consistency fails to make enough inferences. Consider the map-coloring problem on Australia, but with only two colors allowed, red and blue. Arc consistency can do nothing because every variable is already arc consistent: each can be red with blue at the other end of the arc (or vice versa). But clearly there is no solution to the problem: because Western Australia, Northern Territory and South Australia all touch each other, we need at least three colors for them alone.

Arc consistency tightens down the domains (unary constraints) using the arcs (binary constraints). To make progress on problems like map coloring, we need a stronger notion of consistency. **Path consistency** tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables.

PATH CONSISTENCY

A two-variable set $\{X_i, X_j\}$ is path-consistent with respect to a third variable X_m if, for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraints on $\{X_i, X_j\}$, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$. This is called path consistency because one can think of it as looking at a path from X_i to X_j with X_m in the middle.

Let's see how path consistency fares in coloring the Australia map with two colors. We will make the set $\{WA, SA\}$ path consistent with respect to NT. We start by enumerating the consistent assignments to the set. In this case, there are only two: $\{WA = red, SA = blue\}$ and $\{WA = blue, SA = red\}$. We can see that with both of these assignments NT can be neither red nor $blue$ (because it would conflict with either WA or SA). Because there is no valid choice for NT, we eliminate both assignments, and we end up with no valid assignments for $\{WA, SA\}$. Therefore, we know that there can be no solution to this problem. The PC-2 algorithm (Mackworth, 1977) achieves path consistency in much the same way that AC-3 achieves arc consistency. Because it is so similar, we do not show it here.

[1] The AC-4 algorithm (Mohr and Henderson, 1986) runs in $O(cd^2)$ worst-case time but can be slower than AC-3 on average cases. See Exercise 6.13.

6.2.4 *K*-consistency

K-CONSISTENCY

Stronger forms of propagation can be defined with the notion of k**-consistency**. A CSP is k-consistent if, for any set of $k-1$ variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable. 1-consistency says that, given the empty set, we can make any set of one variable consistent: this is what we called node consistency. 2-consistency is the same as arc consistency. For binary constraint networks, 3-consistency is the same as path consistency.

STRONGLY K-CONSISTENT

A CSP is **strongly** k**-consistent** if it is k-consistent and is also $(k-1)$-consistent, $(k-2)$-consistent, ... all the way down to 1-consistent. Now suppose we have a CSP with n nodes and make it strongly n-consistent (i.e., strongly k-consistent for $k=n$). We can then solve the problem as follows: First, we choose a consistent value for X_1. We are then guaranteed to be able to choose a value for X_2 because the graph is 2-consistent, for X_3 because it is 3-consistent, and so on. For each variable X_i, we need only search through the d values in the domain to find a value consistent with $X_1, \ldots, X_{i-1}$. We are guaranteed to find a solution in time $O(n^2 d)$. Of course, there is no free lunch: any algorithm for establishing n-consistency must take time exponential in n in the worst case. Worse, n-consistency also requires space that is exponential in n. The memory issue is even more severe than the time. In practice, determining the appropriate level of consistency checking is mostly an empirical science. It can be said practitioners commonly compute 2-consistency and less commonly 3-consistency.

6.2.5 Global constraints

Remember that a **global constraint** is one involving an arbitrary number of variables (but not necessarily all variables). Global constraints occur frequently in real problems and can be handled by special-purpose algorithms that are more efficient than the general-purpose methods described so far. For example, the *Alldiff* constraint says that all the variables involved must have distinct values (as in the cryptarithmetic problem above and Sudoku puzzles below). One simple form of inconsistency detection for *Alldiff* constraints works as follows: if m variables are involved in the constraint, and if they have n possible distinct values altogether, and $m > n$, then the constraint cannot be satisfied.

This leads to the following simple algorithm: First, remove any variable in the constraint that has a singleton domain, and delete that variable's value from the domains of the remaining variables. Repeat as long as there are singleton variables. If at any point an empty domain is produced or there are more variables than domain values left, then an inconsistency has been detected.

This method can detect the inconsistency in the assignment $\{WA = red, NSW = red\}$ for Figure 6.1. Notice that the variables SA, NT, and Q are effectively connected by an *Alldiff* constraint because each pair must have two different colors. After applying AC-3 with the partial assignment, the domain of each variable is reduced to $\{green, blue\}$. That is, we have three variables and only two colors, so the *Alldiff* constraint is violated. Thus, a simple consistency procedure for a higher-order constraint is sometimes more effective than applying arc consistency to an equivalent set of binary constraints. There are more

complex inference algorithms for *Alldiff* (see van Hoeve and Katriel, 2006) that propagate more constraints but are more computationally expensive to run.

RESOURCE CONSTRAINT

Another important higher-order constraint is the **resource constraint**, sometimes called the *atmost* constraint. For example, in a scheduling problem, let $P_1, \ldots, P_4$ denote the numbers of personnel assigned to each of four tasks. The constraint that no more than 10 personnel are assigned in total is written as $Atmost(10, P_1, P_2, P_3, P_4)$. We can detect an inconsistency simply by checking the sum of the minimum values of the current domains; for example, if each variable has the domain $\{3, 4, 5, 6\}$, the *Atmost* constraint cannot be satisfied. We can also enforce consistency by deleting the maximum value of any domain if it is not consistent with the minimum values of the other domains. Thus, if each variable in our example has the domain $\{2, 3, 4, 5, 6\}$, the values 5 and 6 can be deleted from each domain.

For large resource-limited problems with integer values—such as logistical problems involving moving thousands of people in hundreds of vehicles—it is usually not possible to represent the domain of each variable as a large set of integers and gradually reduce that set by consistency-checking methods. Instead, domains are represented by upper and lower bounds and are managed by **bounds propagation**. For example, in an airline-scheduling problem, let's suppose there are two flights, F_1 and F_2, for which the planes have capacities 165 and 385, respectively. The initial domains for the numbers of passengers on each flight are then

BOUNDS PROPAGATION

$$D_1 = [0, 165] \quad \text{and} \quad D_2 = [0, 385] \ .$$

Now suppose we have the additional constraint that the two flights together must carry 420 people: $F_1 + F_2 = 420$. Propagating bounds constraints, we reduce the domains to

$$D_1 = [35, 165] \quad \text{and} \quad D_2 = [255, 385] \ .$$

BOUNDS CONSISTENT

We say that a CSP is **bounds consistent** if for every variable X, and for both the lower-bound and upper-bound values of X, there exists some value of Y that satisfies the constraint between X and Y for every variable Y. This kind of bounds propagation is widely used in practical constraint problems.

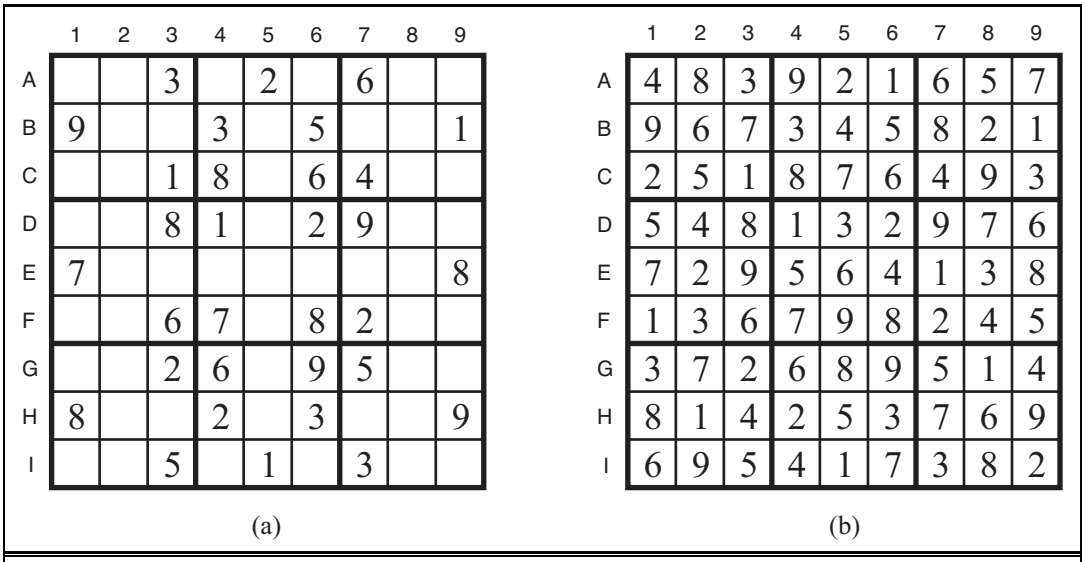
6.2.6 Sudoku example

SUDOKU

The popular **Sudoku** puzzle has introduced millions of people to constraint satisfaction problems, although they may not recognize it. A Sudoku board consists of 81 squares, some of which are initially filled with digits from 1 to 9. The puzzle is to fill in all the remaining squares such that no digit appears twice in any row, column, or 3×3 box (see Figure 6.4). A row, column, or box is called a **unit**.

The Sudoku puzzles that are printed in newspapers and puzzle books have the property that there is exactly one solution. Although some can be tricky to solve by hand, taking tens of minutes, even the hardest Sudoku problems yield to a CSP solver in less than 0.1 second.

A Sudoku puzzle can be considered a CSP with 81 variables, one for each square. We use the variable names *A1* through *A9* for the top row (left to right), down to *I1* through *I9* for the bottom row. The empty squares have the domain $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and the pre-filled squares have a domain consisting of a single value. In addition, there are 27 different



	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Figure 6.4 (a) A Sudoku puzzle and (b) its solution.

$Alldiff$ constraints: one for each row, column, and box of 9 squares.

$$
\begin{aligned}
&\mathit{Alldiff}(A1, A2, A3, A4, A5, A6, A7, A8, A9)\\
&\mathit{Alldiff}(B1, B2, B3, B4, B5, B6, B7, B8, B9)\\
&\ldots\\
&\mathit{Alldiff}(A1, B1, C1, D1, E1, F1, G1, H1, I1)\\
&\mathit{Alldiff}(A2, B2, C2, D2, E2, F2, G2, H2, I2)\\
&\ldots\\
&\mathit{Alldiff}(A1, A2, A3, B1, B2, B3, C1, C2, C3)\\
&\mathit{Alldiff}(A4, A5, A6, B4, B5, B6, C4, C5, C6)\\
&\ldots
\end{aligned}
$$

Let us see how far arc consistency can take us. Assume that the $Alldiff$ constraints have been expanded into binary constraints (such as $A1 \neq A2$) so that we can apply the AC-3 algorithm directly. Consider variable $E6$ from Figure 6.4(a)—the empty square between the 2 and the 8 in the middle box. From the constraints in the box, we can remove not only 2 and 8 but also 1 and 7 from $E6$'s domain. From the constraints in its column, we can eliminate 5, 6, 2, 8, 9, and 3. That leaves $E6$ with a domain of $\{4\}$; in other words, we know the answer for $E6$. Now consider variable $I6$—the square in the bottom middle box surrounded by 1, 3, and 3. Applying arc consistency in its column, we eliminate 5, 6, 2, 4 (since we now know $E6$ must be 4), 8, 9, and 3. We eliminate 1 by arc consistency with $I5$, and we are left with only the value 7 in the domain of $I6$. Now there are 8 known values in column 6, so arc consistency can infer that $A6$ must be 1. Inference continues along these lines, and eventually, AC-3 can solve the entire puzzle—all the variables have their domains reduced to a single value, as shown in Figure 6.4(b).

Of course, Sudoku would soon lose its appeal if every puzzle could be solved by a

mechanical application of AC-3, and indeed AC-3 works only for the easiest Sudoku puzzles. Slightly harder ones can be solved by PC-2, but at a greater computational cost: there are 255,960 different path constraints to consider in a Sudoku puzzle. To solve the hardest puzzles and to make efficient progress, we will have to be more clever.

Indeed, the appeal of Sudoku puzzles for the human solver is the need to be resourceful in applying more complex inference strategies. Aficionados give them colorful names, such as "naked triples." That strategy works as follows: in any unit (row, column or box), find three squares that each have a domain that contains the same three numbers or a subset of those numbers. For example, the three domains might be $\{1, 8\}$, $\{3, 8\}$, and $\{1, 3, 8\}$. From that we don't know which square contains 1, 3, or 8, but we do know that the three numbers must be distributed among the three squares. Therefore we can remove 1, 3, and 8 from the domains of every *other* square in the unit.

It is interesting to note how far we can go without saying much that is specific to Sudoku. We do of course have to say that there are 81 variables, that their domains are the digits 1 to 9, and that there are 27 $Alldiff$ constraints. But beyond that, all the strategies—arc consistency, path consistency, etc.—apply generally to all CSPs, not just to Sudoku problems. Even naked triples is really a strategy for enforcing consistency of $Alldiff$ constraints and has nothing to do with Sudoku *per se*. This is the power of the CSP formalism: for each new problem area, we only need to define the problem in terms of constraints; then the general constraint-solving mechanisms can take over.

6.3 Backtracking Search for CSPs

Sudoku problems are designed to be solved by inference over constraints. But many other CSPs cannot be solved by inference alone; there comes a time when we must search for a solution. In this section we look at backtracking search algorithms that work on partial assignments; in the next section we look at local search algorithms over complete assignments.

We could apply a standard depth-limited search (from Chapter 3). A state would be a partial assignment, and an action would be adding $var = value$ to the assignment. But for a CSP with n variables of domain size d, we quickly notice something terrible: the branching factor at the top level is nd because any of d values can be assigned to any of n variables. At the next level, the branching factor is $(n-1)d$, and so on for n levels. We generate a tree with $n! \cdot d^n$ leaves, even though there are only d^n possible complete assignments!

COMMUTATIVITY

Our seemingly reasonable but naive formulation ignores crucial property common to all CSPs: **commutativity**. A problem is commutative if the order of application of any given set of actions has no effect on the outcome. CSPs are commutative because when assigning values to variables, we reach the same partial assignment regardless of order. Therefore, we need only consider a *single* variable at each node in the search tree. For example, at the root node of a search tree for coloring the map of Australia, we might make a choice between $SA = red$, $SA = green$, and $SA = blue$, but we would never choose between $SA = red$ and $WA = blue$. With this restriction, the number of leaves is d^n, as we would hope.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or k-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

BACKTRACKING SEARCH

The term **backtracking search** is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in Figure 6.5. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value. Part of the search tree for the Australia problem is shown in Figure 6.6, where we have assigned variables in the order $WA, NT, Q, \ldots$. Because the representation of CSPs is standardized, there is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, action function, transition model, or goal test.

Notice that BACKTRACKING-SEARCH keeps only a single representation of a state and alters that representation rather than creating new ones, as described on page 87.

In Chapter 3 we improved the poor performance of uninformed search algorithms by supplying them with domain-specific heuristic functions derived from our knowledge of the problem. It turns out that we can solve CSPs efficiently *without* such domain-specific knowledge. Instead, we can add some sophistication to the unspecified functions in Figure 6.5, using them to address the following questions:

1. Which variable should be assigned next (SELECT-UNASSIGNED-VARIABLE), and in what order should its values be tried (ORDER-DOMAIN-VALUES)?

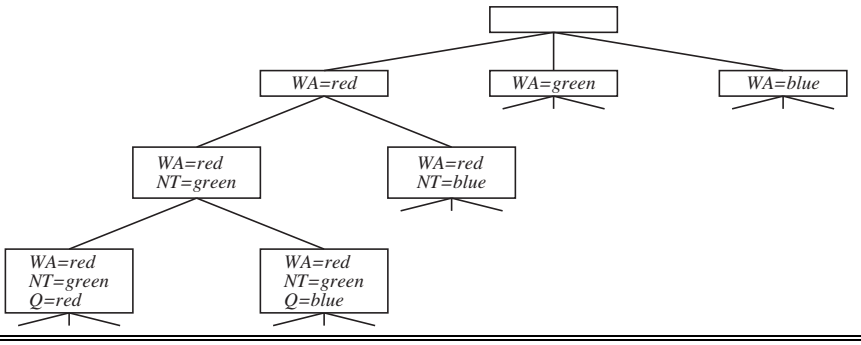


Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

2. What inferences should be performed at each step in the search (INFERENCE)?
3. When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

The subsections that follow answer each of these questions in turn.

6.3.1 Variable and value ordering

The backtracking algorithm contains the line

$$var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(csp) \ .$$

The simplest strategy for SELECT-UNASSIGNED-VARIABLE is to choose the next unassigned variable in order, $\{X_1, X_2, \ldots\}$. This static variable ordering seldom results in the most efficient search. For example, after the assignments for $WA = red$ and $NT = green$ in Figure 6.6, there is only one possible value for SA, so it makes sense to assign $SA = blue$ next rather than assigning Q. In fact, after SA is assigned, the choices for Q, NSW, and V are all forced. This intuitive idea—choosing the variable with the fewest "legal" values—is called the **minimum-remaining-values** (MRV) heuristic. It also has been called the "most constrained variable" or "fail-first" heuristic, the latter because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree. If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables. The MRV heuristic usually performs better than a random or static ordering, sometimes by a factor of 1,000 or more, although the results vary widely depending on the problem.

MINIMUM-REMAINING-VALUES

The MRV heuristic doesn't help at all in choosing the first region to color in Australia, because initially every region has three legal colors. In this case, the **degree heuristic** comes in handy. It attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables. In Figure 6.1, SA is the variable with highest degree, 5; the other variables have degree 2 or 3, except for T, which has degree 0. In fact, once SA is chosen, applying the degree heuristic solves the problem without any false steps—you can choose *any* consistent color at each choice point and still arrive at a solution with no backtracking. The minimum-remaining-

DEGREE HEURISTIC

values heuristic is usually a more powerful guide, but the degree heuristic can be useful as a tie-breaker.

LEAST-CONSTRAINING-VALUE

Once a variable has been selected, the algorithm must decide on the order in which to examine its values. For this, the **least-constraining-value** heuristic can be effective in some cases. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph. For example, suppose that in Figure 6.1 we have generated the partial assignment with $WA = red$ and $NT = green$ and that our next choice is for Q. Blue would be a bad choice because it eliminates the last legal value left for Q's neighbor, SA. The least-constraining-value heuristic therefore prefers red to blue. In general, the heuristic is trying to leave the maximum flexibility for subsequent variable assignments. Of course, if we are trying to find all the solutions to a problem, not just the first one, then the ordering does not matter because we have to consider every value anyway. The same holds if there are no solutions to the problem.

Why should variable selection be fail-first, but value selection be fail-last? It turns out that, for a wide variety of problems, a variable ordering that chooses a variable with the minimum number of remaining values helps minimize the number of nodes in the search tree by pruning larger parts of the tree earlier. For value ordering, the trick is that we only need one solution; therefore it makes sense to look for the most likely values first. If we wanted to enumerate all solutions rather than just find one, then value ordering would be irrelevant.

6.3.2 Interleaving search and inference

So far we have seen how AC-3 and other algorithms can infer reductions in the domain of variables *before* we begin the search. But inference can be even more powerful in the course of a search: every time we make a choice of a value for a variable, we have a brand-new opportunity to infer new domain reductions on the neighboring variables.

FORWARD CHECKING

One of the simplest forms of inference is called **forward checking**. Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X. Because forward checking only does arc consistency inferences, there is no reason to do forward checking if we have already done arc consistency as a preprocessing step.

Figure 6.7 shows the progress of backtracking search on the Australia CSP with forward checking. There are two important points to notice about this example. First, notice that after $WA = red$ and $Q = green$ are assigned, the domains of NT and SA are reduced to a single value; we have eliminated branching on these variables altogether by propagating information from WA and Q. A second point to notice is that after $V = blue$, the domain of SA is empty. Hence, forward checking has detected that the partial assignment $\{WA = red, Q = green, V = blue\}$ is inconsistent with the constraints of the problem, and the algorithm will therefore backtrack immediately.

For many problems the search will be more effective if we combine the MRV heuristic with forward checking. Consider Figure 6.7 after assigning $\{WA = red\}$. Intuitively, it seems that that assignment constrains its neighbors, NT and SA, so we should handle those

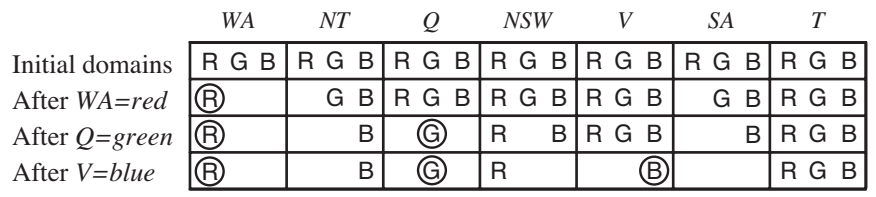


Figure 6.7 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes red from the domains of the neighboring variables NT and SA. After $Q = green$ is assigned, $green$ is deleted from the domains of NT, SA, and NSW. After $V = blue$ is assigned, $blue$ is deleted from the domains of NSW and SA, leaving SA with no legal values.

variables next, and then all the other variables will fall into place. That's exactly what happens with MRV: NT and SA have two values, so one of them is chosen first, then the other, then Q, NSW, and V in order. Finally T still has three values, and any one of them works. We can view forward checking as an efficient way to incrementally compute the information that the MRV heuristic needs to do its job.

Although forward checking detects many inconsistencies, it does not detect all of them. The problem is that it makes the current variable arc-consistent, but doesn't look ahead and make all the other variables arc-consistent. For example, consider the third row of Figure 6.7. It shows that when WA is red and Q is $green$, both NT and SA are forced to be blue. Forward checking does not look far enough ahead to notice that this is an inconsistency: NT and SA are adjacent and so cannot have the same value.

MAINTAINING ARC CONSISTENCY (MAC)

The algorithm called MAC (for **Maintaining Arc Consistency (MAC)**) detects this inconsistency. After a variable X_i is assigned a value, the INFERENCE procedure calls AC-3, but instead of a queue of all arcs in the CSP, we start with only the arcs (X_j, X_i) for all X_j that are unassigned variables that are neighbors of X_i. From there, AC-3 does constraint propagation in the usual way, and if any variable has its domain reduced to the empty set, the call to AC-3 fails and we know to backtrack immediately. We can see that MAC is strictly more powerful than forward checking because forward checking does the same thing as MAC on the initial arcs in MAC's queue; but unlike MAC, forward checking does not recursively propagate constraints when changes are made to the domains of variables.

6.3.3 Intelligent backtracking: Looking backward

The BACKTRACKING-SEARCH algorithm in Figure 6.5 has a very simple policy for what to do when a branch of the search fails: back up to the preceding variable and try a different value for it. This is called **chronological backtracking** because the *most recent* decision point is revisited. In this subsection, we consider better possibilities.

CHRONOLOGICAL BACKTRACKING

Consider what happens when we apply simple backtracking in Figure 6.1 with a fixed variable ordering Q, NSW, V, T, SA, WA, NT. Suppose we have generated the partial assignment $\{Q = red, NSW = green, V = blue, T = red\}$. When we try the next variable, SA, we see that every value violates a constraint. We back up to T and try a new color for

Tasmania! Obviously this is silly—recoloring Tasmania cannot possibly resolve the problem with South Australia.

A more intelligent approach to backtracking is to backtrack to a variable that might fix the problem—a variable that was responsible for making one of the possible values of SA impossible. To do this, we will keep track of a set of assignments that are in conflict with some value for SA. The set (in this case $\{Q = red, NSW = green, V = blue, \}$), is called the **conflict set** for SA. The **backjumping** method backtracks to the *most recent* assignment in the conflict set; in this case, backjumping would jump over Tasmania and try a new value for V. This method is easily implemented by a modification to BACKTRACK such that it accumulates the conflict set while checking for a legal value to assign. If no legal value is found, the algorithm should return the most recent element of the conflict set along with the failure indicator.

CONFLICT SET

BACKJUMPING

The sharp-eyed reader will have noticed that forward checking can supply the conflict set with no extra work: whenever forward checking based on an assignment $X = x$ deletes a value from Y's domain, it should add $X = x$ to Y's conflict set. If the last value is deleted from Y's domain, then the assignments in the conflict set of Y are added to the conflict set of X. Then, when we get to Y, we know immediately where to backtrack if needed.

The eagle-eyed reader will have noticed something odd: backjumping occurs when every value in a domain is in conflict with the current assignment; but forward checking detects this event and prevents the search from ever reaching such a node! In fact, it can be shown that *every* branch pruned by backjumping is also pruned by forward checking. Hence, simple backjumping is redundant in a forward-checking search or, indeed, in a search that uses stronger consistency checking, such as MAC.

Despite the observations of the preceding paragraph, the idea behind backjumping remains a good one: to backtrack based on the reasons for failure. Backjumping notices failure when a variable's domain becomes empty, but in many cases a branch is doomed long before this occurs. Consider again the partial assignment $\{WA = red, NSW = red\}$ (which, from our earlier discussion, is inconsistent). Suppose we try $T = red$ next and then assign NT, Q, V, SA. We know that no assignment can work for these last four variables, so eventually we run out of values to try at NT. Now, the question is, where to backtrack? Backjumping cannot work, because NT *does* have values consistent with the preceding assigned variables—NT doesn't have a complete conflict set of preceding variables that caused it to fail. We know, however, that the four variables NT, Q, V, and SA, *taken together*, failed because of a set of preceding variables, which must be those variables that directly conflict with the four. This leads to a deeper notion of the conflict set for a variable such as NT: it is that set of preceding variables that caused NT, *together with any subsequent variables*, to have no consistent solution. In this case, the set is WA and NSW, so the algorithm should backtrack to NSW and skip over Tasmania. A backjumping algorithm that uses conflict sets defined in this way is called **conflict-directed backjumping**.

CONFLICT-DIRECTED BACKJUMPING

We must now explain how these new conflict sets are computed. The method is in fact quite simple. The "terminal" failure of a branch of the search always occurs because a variable's domain becomes empty; that variable has a standard conflict set. In our example, SA fails, and its conflict set is (say) $\{WA, NT, Q\}$. We backjump to Q, and Q *absorbs*

the conflict set from SA (minus Q itself, of course) into its own direct conflict set, which is $\{NT, NSW\}$; the new conflict set is $\{WA, NT, NSW\}$. That is, there is no solution from Q onward, given the preceding assignment to $\{WA, NT, NSW\}$. Therefore, we backtrack to NT, the most recent of these. NT absorbs $\{WA, NT, NSW\} - \{NT\}$ into its own direct conflict set $\{WA\}$, giving $\{WA, NSW\}$ (as stated in the previous paragraph). Now the algorithm backjumps to NSW, as we would hope. To summarize: let X_j be the current variable, and let $conf(X_j)$ be its conflict set. If every possible value for X_j fails, backjump to the most recent variable X_i in $conf(X_j)$, and set

$$conf(X_i) \leftarrow conf(X_i) \cup conf(X_j) - \{X_i\} .$$

When we reach a contradiction, backjumping can tell us how far to back up, so we don't waste time changing variables that won't fix the problem. But we would also like to avoid running into the same problem again. When the search arrives at a contradiction, we know that some subset of the conflict set is responsible for the problem. **Constraint learning** is the idea of finding a minimum set of variables from the conflict set that causes the problem. This set of variables, along with their corresponding values, is called a **no-good**. We then record the no-good, either by adding a new constraint to the CSP or by keeping a separate cache of no-goods.

CONSTRAINT LEARNING

NO-GOOD

For example, consider the state $\{WA = red, NT = green, Q = blue\}$ in the bottom row of Figure 6.6. Forward checking can tell us this state is a no-good because there is no valid assignment to SA. In this particular case, recording the no-good would not help, because once we prune this branch from the search tree, we will never encounter this combination again. But suppose that the search tree in Figure 6.6 were actually part of a larger search tree that started by first assigning values for V and T. Then it would be worthwhile to record $\{WA = red, NT = green, Q = blue\}$ as a no-good because we are going to run into the same problem again for each possible set of assignments to V and T.

No-goods can be effectively used by forward checking or by backjumping. Constraint learning is one of the most important techniques used by modern CSP solvers to achieve efficiency on complex problems.

6.4 Local Search for CSPs

Local search algorithms (see Section 4.1) turn out to be effective in solving many CSPs. They use a complete-state formulation: the initial state assigns a value to every variable, and the search changes the value of one variable at a time. For example, in the 8-queens problem (see Figure 4.3), the initial state might be a random configuration of 8 queens in 8 columns, and each step moves a single queen to a new position in its column. Typically, the initial guess violates several constraints. The point of local search is to eliminate the violated constraints.[2]

In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variables—the **min-conflicts**

MIN-CONFLICTS

[2] Local search can easily be extended to constraint optimization problems (COPs). In that case, all the techniques for hill climbing and simulated annealing can be applied to optimize the objective function.

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
      if current is a solution for csp then return current
      var ← a randomly chosen conflicted variable from csp.VARIABLES
      value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
      set var = value in current
  return failure
```

Figure 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

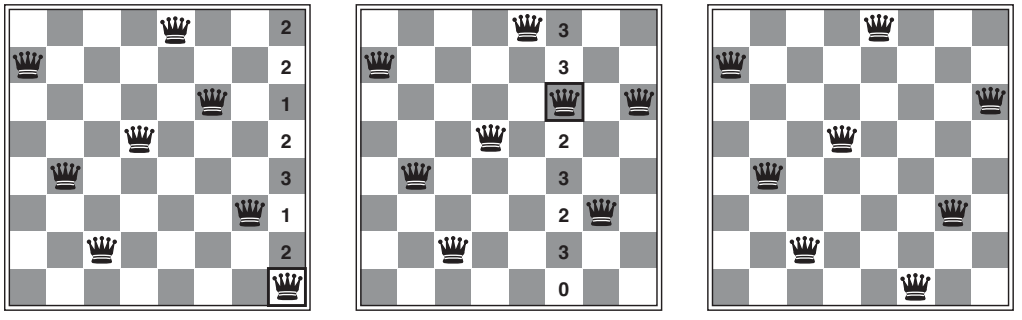


Figure 6.9 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

heuristic. The algorithm is shown in Figure 6.8 and its application to an 8-queens problem is diagrammed in Figure 6.9.

Min-conflicts is surprisingly effective for many CSPs. Amazingly, on the n-queens problem, if you don't count the initial placement of queens, the run time of min-conflicts is roughly *independent of problem size*. It solves even the *million*-queens problem in an average of 50 steps (after the initial assignment). This remarkable observation was the stimulus leading to a great deal of research in the 1990s on local search and the distinction between easy and hard problems, which we take up in Chapter 7. Roughly speaking, n-queens is easy for local search because solutions are densely distributed throughout the state space. Min-conflicts also works well for hard problems. For example, it has been used to schedule observations for the Hubble Space Telescope, reducing the time taken to schedule a week of observations from three weeks (!) to around 10 minutes.

All the local search techniques from Section 4.1 are candidates for application to CSPs, and some of those have proved especially effective. The landscape of a CSP under the min-conflicts heuristic usually has a series of plateaux. There may be millions of variable assignments that are only one conflict away from a solution. Plateau search—allowing sideways moves to another state with the same score—can help local search find its way off this plateau. This wandering on the plateau can be directed with **tabu search**: keeping a small list of recently visited states and forbidding the algorithm to return to those states. Simulated annealing can also be used to escape from plateaux.

CONSTRAINT WEIGHTING

Another technique, called **constraint weighting**, can help concentrate the search on the important constraints. Each constraint is given a numeric weight, W_i, initially all 1. At each step of the search, the algorithm chooses a variable/value pair to change that will result in the lowest total weight of all violated constraints. The weights are then adjusted by incrementing the weight of each constraint that is violated by the current assignment. This has two benefits: it adds topography to plateaux, making sure that it is possible to improve from the current state, and it also, over time, adds weight to the constraints that are proving difficult to solve.

Another advantage of local search is that it can be used in an online setting when the problem changes. This is particularly important in scheduling problems. A week's airline schedule may involve thousands of flights and tens of thousands of personnel assignments, but bad weather at one airport can render the schedule infeasible. We would like to repair the schedule with a minimum number of changes. This can be easily done with a local search algorithm starting from the current schedule. A backtracking search with the new set of constraints usually requires much more time and might find a solution with many changes from the current schedule.

6.5 THE STRUCTURE OF PROBLEMS

In this section, we examine ways in which the *structure* of the problem, as represented by the constraint graph, can be used to find solutions quickly. Most of the approaches here also apply to other problems besides CSPs, such as probabilistic reasoning. After all, the only way we can possibly hope to deal with the real world is to decompose it into many subproblems. Looking again at the constraint graph for Australia (Figure 6.1(b), repeated as Figure 6.12(a)), one fact stands out: Tasmania is not connected to the mainland.[3] Intuitively, it is obvious that coloring Tasmania and coloring the mainland are **independent subproblems**—any solution for the mainland combined with any solution for Tasmania yields a solution for the whole map. Independence can be ascertained simply by finding **connected components** of the constraint graph. Each component corresponds to a subproblem CSP_i. If assignment S_i is a solution of CSP_i, then $\bigcup_i S_i$ is a solution of $\bigcup_i CSP_i$. Why is this important? Consider the following: suppose each CSP_i has c variables from the total of n variables, where c is a constant. Then there are n/c subproblems, each of which takes at most d^c work to solve,

INDEPENDENT SUBPROBLEMS

CONNECTED COMPONENT

[3] A careful cartographer or patriotic Tasmanian might object that Tasmania should not be colored the same as its nearest mainland neighbor, to avoid the impression that it *might* be part of that state.

where d is the size of the domain. Hence, the total work is $O(d^c n/c)$, which is *linear* in n; without the decomposition, the total work is $O(d^n)$, which is exponential in n. Let's make this more concrete: dividing a Boolean CSP with 80 variables into four subproblems reduces the worst-case solution time from the lifetime of the universe down to less than a second.

Completely independent subproblems are delicious, then, but rare. Fortunately, some other graph structures are also easy to solve. For example, a constraint graph is a **tree** when any two variables are connected by only one path. We show that *any tree-structured CSP can be solved in time linear in the number of variables.*[4] The key is a new notion of consistency, called **directed arc consistency** or DAC. A CSP is defined to be directed arc-consistent under an ordering of variables $X_1, X_2, \ldots, X_n$ if and only if every X_i is arc-consistent with each X_j for $j > i$.

DIRECTED ARC CONSISTENCY

To solve a tree-structured CSP, first pick any variable to be the root of the tree, and choose an ordering of the variables such that each variable appears after its parent in the tree. Such an ordering is called a **topological sort**. Figure 6.10(a) shows a sample tree and (b) shows one possible ordering. Any tree with n nodes has $n-1$ arcs, so we can make this graph directed arc-consistent in $O(n)$ steps, each of which must compare up to d possible domain values for two variables, for a total time of $O(nd^2)$. Once we have a directed arc-consistent graph, we can just march down the list of variables and choose any remaining value. Since each link from a parent to its child is arc consistent, we know that for any value we choose for the parent, there will be a valid value left to choose for the child. That means we won't have to backtrack; we can move linearly through the variables. The complete algorithm is shown in Figure 6.11.

TOPOLOGICAL SORT

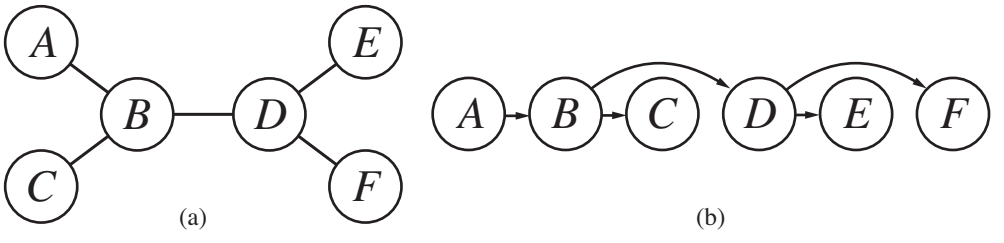


Figure 6.10 (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with A as the root. This is known as a **topological sort** of the variables.

Now that we have an efficient algorithm for trees, we can consider whether more general constraint graphs can be *reduced* to trees somehow. There are two primary ways to do this, one based on removing nodes and one based on collapsing nodes together.

The first approach involves assigning values to some variables so that the remaining variables form a tree. Consider the constraint graph for Australia, shown again in Figure 6.12(a). If we could delete South Australia, the graph would become a tree, as in (b). Fortunately, we can do this (in the graph, not the continent) by fixing a value for SA and

[4] Sadly, very few regions of the world have tree-structured maps, although Sulawesi comes close.

```
function TREE-CSP-SOLVER(csp) returns a solution, or failure
  inputs: csp, a CSP with components X, D, C

  n ← number of variables in X
  assignment ← an empty assignment
  root ← any variable in X
  X ← TOPOLOGICALSORT(X, root)
  for j = n down to 2 do
    MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
    if it cannot be made consistent then return failure
  for i = 1 to n do
    assignment[X_i] ← any consistent value from D_i
    if there is no consistent value then return failure
  return assignment
```

Figure 6.11 The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

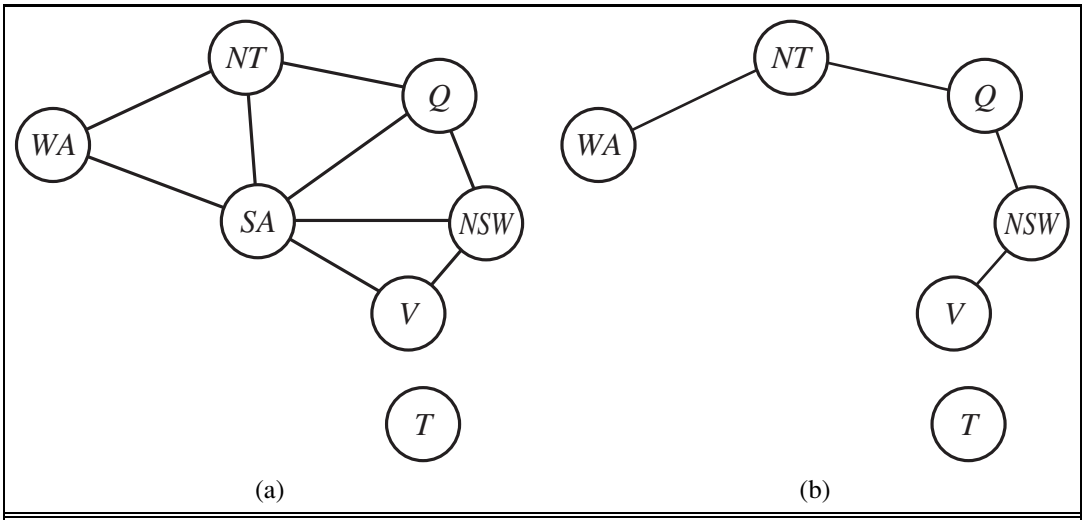


Figure 6.12 (a) The original constraint graph from Figure 6.1. (b) The constraint graph after the removal of SA.

deleting from the domains of the other variables any values that are inconsistent with the value chosen for SA.

Now, any solution for the CSP after SA and its constraints are removed will be consistent with the value chosen for SA. (This works for binary CSPs; the situation is more complicated with higher-order constraints.) Therefore, we can solve the remaining tree with the algorithm given above and thus solve the whole problem. Of course, in the general case (as opposed to map coloring), the value chosen for SA could be the wrong one, so we would need to try each possible value. The general algorithm is as follows:

1. Choose a subset S of the CSP's variables such that the constraint graph becomes a tree after removal of S. S is called a **cycle cutset**.
2. For each possible assignment to the variables in S that satisfies all constraints on S,
   (a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S, and
   (b) If the remaining CSP has a solution, return it together with the assignment for S.

CYCLE CUTSET

If the cycle cutset has size c, then the total run time is $O(d^c \cdot (n-c)d^2)$: we have to try each of the d^c combinations of values for the variables in S, and for each combination we must solve a tree problem of size $n-c$. If the graph is "nearly a tree," then c will be small and the savings over straight backtracking will be huge. In the worst case, however, c can be as large as $(n-2)$. Finding the *smallest* cycle cutset is NP-hard, but several efficient approximation algorithms are known. The overall algorithmic approach is called **cutset conditioning**; it comes up again in Chapter 14, where it is used for reasoning about probabilities.

CUTSET CONDITIONING

TREE DECOMPOSITION

The second approach is based on constructing a **tree decomposition** of the constraint graph into a set of connected subproblems. Each subproblem is solved independently, and the resulting solutions are then combined. Like most divide-and-conquer algorithms, this works well if no subproblem is too large. Figure 6.13 shows a tree decomposition of the map-coloring problem into five subproblems. A tree decomposition must satisfy the following three requirements:

- Every variable in the original problem appears in at least one of the subproblems.
- If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.
- If a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems.

The first two conditions ensure that all the variables and constraints are represented in the decomposition. The third condition seems rather technical, but simply reflects the constraint that any given variable must have the same value in every subproblem in which it appears; the links joining subproblems in the tree enforce this constraint. For example, SA appears in all four of the connected subproblems in Figure 6.13. You can verify from Figure 6.12 that this decomposition makes sense.

We solve each subproblem independently; if any one has no solution, we know the entire problem has no solution. If we can solve all the subproblems, then we attempt to construct a global solution as follows. First, we view each subproblem as a "mega-variable" whose domain is the set of all solutions for the subproblem. For example, the leftmost subproblem in Figure 6.13 is a map-coloring problem with three variables and hence has six solutions—one is $\{WA = red, SA = blue, NT = green\}$. Then, we solve the constraints connecting the subproblems, using the efficient algorithm for trees given earlier. The constraints between subproblems simply insist that the subproblem solutions agree on their shared variables. For example, given the solution $\{WA = red, SA = blue, NT = green\}$ for the first subproblem, the only consistent solution for the next subproblem is $\{SA = blue, NT = green, Q = red\}$.

A given constraint graph admits many tree decompositions; in choosing a decomposition, the aim is to make the subproblems as small as possible. The **tree width** of a tree

TREE WIDTH

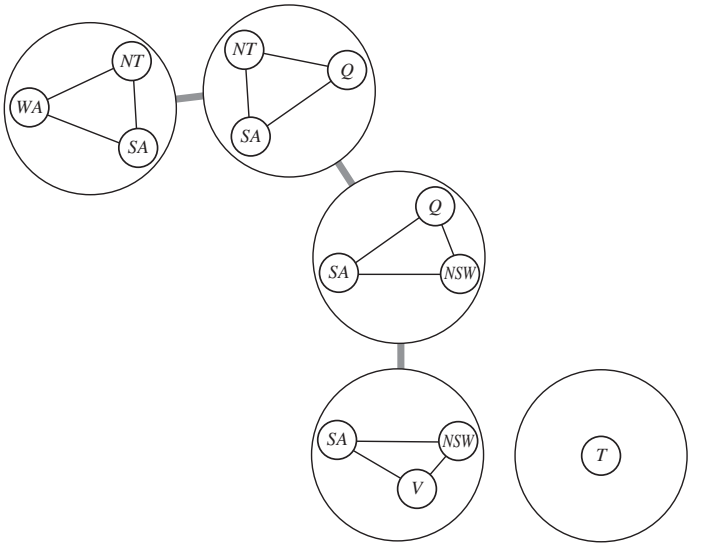


Figure 6.13 A tree decomposition of the constraint graph in Figure 6.12(a).

decomposition of a graph is one less than the size of the largest subproblem; the tree width of the graph itself is defined to be the minimum tree width among all its tree decompositions. If a graph has tree width w and we are given the corresponding tree decomposition, then the problem can be solved in $O(nd^{w+1})$ time. Hence, *CSPs with constraint graphs of bounded tree width are solvable in polynomial time.* Unfortunately, finding the decomposition with minimal tree width is NP-hard, but there are heuristic methods that work well in practice.

So far, we have looked at the structure of the constraint graph. There can be important structure in the *values* of variables as well. Consider the map-coloring problem with n colors. For every consistent solution, there is actually a set of $n!$ solutions formed by permuting the color names. For example, on the Australia map we know that WA, NT, and SA must all have different colors, but there are $3! = 6$ ways to assign the three colors to these three regions. This is called **value symmetry**. We would like to reduce the search space by a factor of $n!$ by breaking the symmetry. We do this by introducing a **symmetry-breaking constraint**. For our example, we might impose an arbitrary ordering constraint, $NT < SA < WA$, that requires the three values to be in alphabetical order. This constraint ensures that only one of the $n!$ solutions is possible: $\{NT = blue, SA = green, WA = red\}$.

VALUE SYMMETRY
SYMMETRY-BREAKING CONSTRAINT

For map coloring, it was easy to find a constraint that eliminates the symmetry, and in general it is possible to find constraints that eliminate all but one symmetric solution in polynomial time, but it is NP-hard to eliminate all symmetry among intermediate sets of values during search. In practice, breaking value symmetry has proved to be important and effective on a wide range of problems.