**Adversarial search**

The previous chapters 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. 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**.

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. 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.

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.

Games, unlike most of the toy problems, 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 35100 or 10154nodes (although the search graph has “only” about 1040 distinct nodes). Games, like the realworld, therefore require the ability to make *some* decision even when calculating the *optimal* decision is infeasible. Games also penalize inefficiency severely. Whereas an implementationof A∗ search that is half as efficient will simply take twice as long to run to completion, a chessprogram that is half as efficient in using its available time probably will be beaten into theground, other things being equal. Game-playing research has therefore spawned a number ofinteresting 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 completesearch. We also discuss bridge, which includes elements of **imperfect information** becausenot all cards are visible to each player. Finally, we look at how state-of-the-art game-playingprograms fare against human opposition and at directions for future developments.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 gameis over. At the end of the game, points are awarded to the winning player and penalties aregiven to the loser. A game can be formally defined as a kind of search problem with the following elements:

• S0: 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 . 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 + . “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 . 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 1040 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.



**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

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 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 a1, a2, and a3. The possible replies to a1 for MIN are b1, b2, b3, 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 PLY the terminal states in this game range from 2 to 14. Given a game tree, the optimal strategy can be determined from the **minimax value** of each node, which we write as 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:



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 the **minimax decision** at the root: action a1 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 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.

**The 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(bm). 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. For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.



**Optimal decisions in multiplayer games**

Many popular games allow more than two players. Let us examine how to 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 <vA, vB, vC> 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 <vA =1, vB =2, vC =6> and <vA =4, vB =2, vC =3>. 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 <vA =1, vB =2, vC =6>. Hence, the backed-up value of X is this vector. The backed-up value of a node n is always the utility 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, 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.

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 <vA =1000, vB =1000> 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.

