**UNIT II Optimal Decision in Games**

* A game can be formally defined as a kind of search problem with the following components:

1. The **initial state**, which includes the board position and identifies the player to move.
2. A **successor function**, which returns a list of *(move, state)* pairs, each indicating a legal move and the resulting state.
3. A **terminal test,** which determines when the galme is over. States where the game has ended are called terminal states.
4. A **utility function** (also called an **objective function** or **payoff function**), which gives a numeric value for the terminal states. In chess, the outcome is a win, loss, or draw, with values +1, -1, or 0.
5. The initial state and the legal moves for each side define the **game tree** for the game.
6. **Figure 1** shows part of the game tree for tic-tac-toe (noughts and crosses).
7. From the initial state, MAX has nine possible moves.
8. 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.
9. 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).
10. It is MAX'S job to use the search tree (particularly the utility of terminal states) to determine the best move.

****

**Figure 1:** A (partial) search tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the search tree, giving alternating moves by MIN (0) and MAX, until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

**Optimal strategies**

1. In a normal search problem, the optimal solution would be a sequence of moves leading to a goal state-a terminal state that is a win.
2. The possible moves for MAX at the root node are labelled **a1, a2,** and ***a3****.*
3. The possible replies to **a1**for MIN are **b1, b2, b3***,* and so on.
4. This particular game ends after one move each by MAX and MIN.
5. (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.
6. Given a game tree, the optimal strategy can be determined by examining the **minimax value** of each node, which we write as MINIMAX- VALUE (n).
7. 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.
8. Obviously, the minimax value of a terminal state is just its utility.
9. Furthermore, given a choice, MAX will prefer to move to a state of maximum value, whereas MIN prefers a state of minimum value. So we have the following:



Let us apply these definitions to the game tree in **Figure 2**.

1. The terminal nodes on the bottom level are already labelled with their utility values.
2. The first MIN node, labelled B, has three successors with values 3, 12, and 8, so its minimax value is 3.
3. Similarly, the other two MIN nodes have minimax value 2.
4. The root node is a MAX node; its successors have minimax values 3, 2, and 2; so it has a minimax value of 3.
5. We can also identify the **minimax decision** at the root: action ***a1*** is the optimal choice for MAX because it leads to the successor with the highest minimax value.
6. This definition of optimal play for MAX assumes that MIN also plays optimally-it maximizes the worst-case outcome for MAX.



**Figure 2:** A two-ply game tree. The Δ nodes are "MAX nodes," in which it is MAX'S turn to move, and the Δ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labelled with their minimax values. MAX'S best move at the root is **a1**, because it leads to the successor with the highest minimax value, and MIN'S best reply is **b1**, because it leads to the successor with the lowest minimax value.

**The minimax algorithm**

1. The **minimax algorithm** (**Figure 3**) computes the minimax decision from the current state.
2. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations.
3. 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.
4. For example, in **Figure 2**, the algorithm first recurses down to the three bottomleft nodes, and uses the **UTILITY function** on them to discover that their values are 3, 12, and 8 respectively.
5. Then it takes the minimum of these values, 3, and returns it as the backed-up value of node B.
6. A similar process gives the backed up values of 2 for *C* and 2 for *D.*
7. Finally, we take the maximum of 3,2, and 2 to get the backed-up value of 3 for the root node.



**Figure 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.

1. The minimax algorithm performs a complete depth-first exploration of the game tree.
2. 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).
3. The space complexity is O(bm) for an algorithm that generates all successors at once, or O(m) for an algorithm that generates successors one at a time.

**Optimal decisions in multiplayer games**

1. Many popular games allow more than two players. Let us examine how to extend the minimax idea to multiplayer games.
2. This is straightforward from the technical viewpoint, but raises some interesting new conceptual issues.
3. First, we need to replace the single value for each nocle with a vector of values.
4. For example, in a three-player game with players A, B, and C,a vector (vA,v ~vc,) is associated with each node.
5. For terminal states, this vector gives the utility of the state from each player's viewpoint.
6. (In two-player, zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite.)
7. The simplest way to implement this is to have the UTILITY function return a vector of utilities.
8. Now we have to consider nonterminal states. Consider the node marked X in the game tree shown in **Figure 4.**



**Figure 4:** The first three ply of a game tree with three players (A, *B,* C). Each node is labelled with values from the viewpoint of each player. The best move is marked at the root.

1. 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.
2. This means that if state X is reached, subsequent play will lead to a terminal state with utilities (vA = 1, *vB* = *2, vC* = 6).
3. Hence the backed-up value of X is this vector. In general, the backed-up value of a node n is the utility vector of whichever successor has the highest value for the player choosing at n.
4. Multiplayer games usually involve **alliances,** whether formal or informal, among the players. Alliances are made and broken as the game proceeds.
5. 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.
6. 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.
7. 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.

**Alpha Beta Pruning**

1. The problem with minimax search is that the numbeir of game states it has to examine is exponential in the number of moves.
2. 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 in order to eliminate large parts of the tree from consideration.
3. The particular technique we will examine is called **alpha-beta pruning.** When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.
4. Consider again the two-ply game tree from **Figure 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.** The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes.
5. As a simplification of the formula for MINIMAX-VALUE. Let the two unevaluated successors of node C in **Figure 5** have values x and y and let z be the minimum of x and y. The value of the root node is given by
6. In other words, the value of the root and hence the minimax decision are independent of the values of the pruned leaves x and y.



**Figure 5:** Stages in the calculation of the optimal decision for the game tree in **Figure 2.**

At each point, we show the range of possible values for each node.

(a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of at most 3.

(b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3.

(c) The third leaf below B has a value of 8; we have seen all B's successors, so the value of B is exactly 3. Now, we can infer that the value of the root is at least 3, because MAX has a choice worth 3 at the root.

(d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of at most 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in loohng at the other successors of C. This is an example of alpha-beta pruning.

(e) The first leaf below D has the value 14, so D is worth at most 14. This is still higher than MAX'S best alternative (i.e., 3), so we need to keep exploring D's successors. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14.

(f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX'S decision at the root is to move to B, giving a value of 3.

1. Alpha-beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves.
2. The general principle is this: consider a node n somewhere in the tree (see **Figure 6**), such that Player has a choice of moving to that node.
3. 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.
4. 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:

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

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

1. 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.

****

**Figure 6:** Alpha-beta pruning: the general case. If m is better than n for Player, we will never get to n in play.

1. The complete algorithm is given in **Figure 7.**
2. The effectiveness of alpha-beta pruning is highly dependent on the order in which the successors are examined.
3. For example, in **Figure 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.
4. If the third successor had been generated first, we would have been able to prune the other two. This suggests that it might be worthwhile to try to examine first the successors that are likely to be best.
5. If we assume that this can be done then it turns out that alpha-beta needs to examine only 0(bd/2) nodes to pick the best move, instead of O(bd) for minimax.
6. In games, repeated states occur frequently because of transpositions-different permutations of the move sequence that end up in the same position.

****

**Figure 7:** The alpha-beta search algorithm.

Notice that these routines are the same as the MINIMAX routines in **Figure 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).