ARTIFICIAL INTELLIGENCE – CS323

MODULE 3

Module-3: Adversarial Search	RBT Levels: 1, 2, 3	Hours 08

Adversarial Search: Games, Optimal Decisions in Games, Alpha-Beta Pruning, Imperfect Real-Time Decisions, Stochastic Games, Partially Observable Games, Alternative Approaches.

Activity:
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 poker.

T1: Chapter 5 - 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.8

3.1 Adversarial Games

Adversarial search, or **game-tree search,** is a technique for analyzing an adversarial game in order to try to determine who can win the game and what moves the players should make in order to win.

Adversarial search is one of the oldest topics in Artificial Intelligence. The original ideas for adversarial search were developed by Shannon in 1950 and independently by Turing in 1951, in the context of the game of chess—and their ideas still form the basis for the techniques used today.

3.1.1 Zero-sum game

A **zero-sum game** is a situation in game theory where any gain by one player must be matched by an equivalent loss by another player, resulting in the total gains and losses of all participants adding up to zero. This means there is no possibility for a "win-win" or "lose-lose" outcome; a player can only improve their position by worsening someone else's.

Examples

- **Poker**: The total amount of money on the table remains the same. When one player wins money, another player must have lost that same amount.
- **Matching Pennies:** Two players simultaneously show a penny as either heads or tails. One player wins if the pennies match (both heads or both tails), while the other wins if they don't. The winner takes both pennies from the loser.

Why It's Called Zero-Sum
The name comes from the fact that if you add up all the gains and subtract all the losses, the net sum equals zero. For example, if Player A wins Rs.10 (a gain of +10) and Player B loses Rs.10 (a loss of -10), the total sum is +10 - 10 = 0.

Zero-sum games are of **perfect information** (such as chess). 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.

A **zero-sum game** is 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 ½ + ½ . "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 ½ .

3.1.2 Games

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.

Games 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 therefore **require the ability to make *some* decision** (even when calculating the *optimal* decision is infeasible).

Games also **penalize inefficiency severely**. Game-playing research has therefore spawned a number of interesting ideas on **how to make the best possible use of time**.

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.

Pruning allows us to ignore portions of the search tree that make no difference to the final choice.

Heuristic **evaluation functions** allow us to approximate the true utility of a state without doing a complete search.

3.1.3 Two player games

- We first consider games with two players, whom we call MAX and MIN.
- 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 ½ .

Some games have a wider variety of possible outcomes; the payoffs in backgammon range from 0 to +192.

3.1.4 Game Tree

- A game tree is one where the nodes are game states and the edges are moves.
- The initial state, ACTIONS function, and RESULT function define the **game tree** for the game.

A **game tree** is a visual representation of every possible sequence of moves in the game, starting from the initial state (the root node) and branching out with each player's potential move until a win, loss, or draw is reached (the leaf nodes). Each node represents a specific state of the game, and the tree shows all possible games, which can be explored by algorithms to find the optimal strategy.

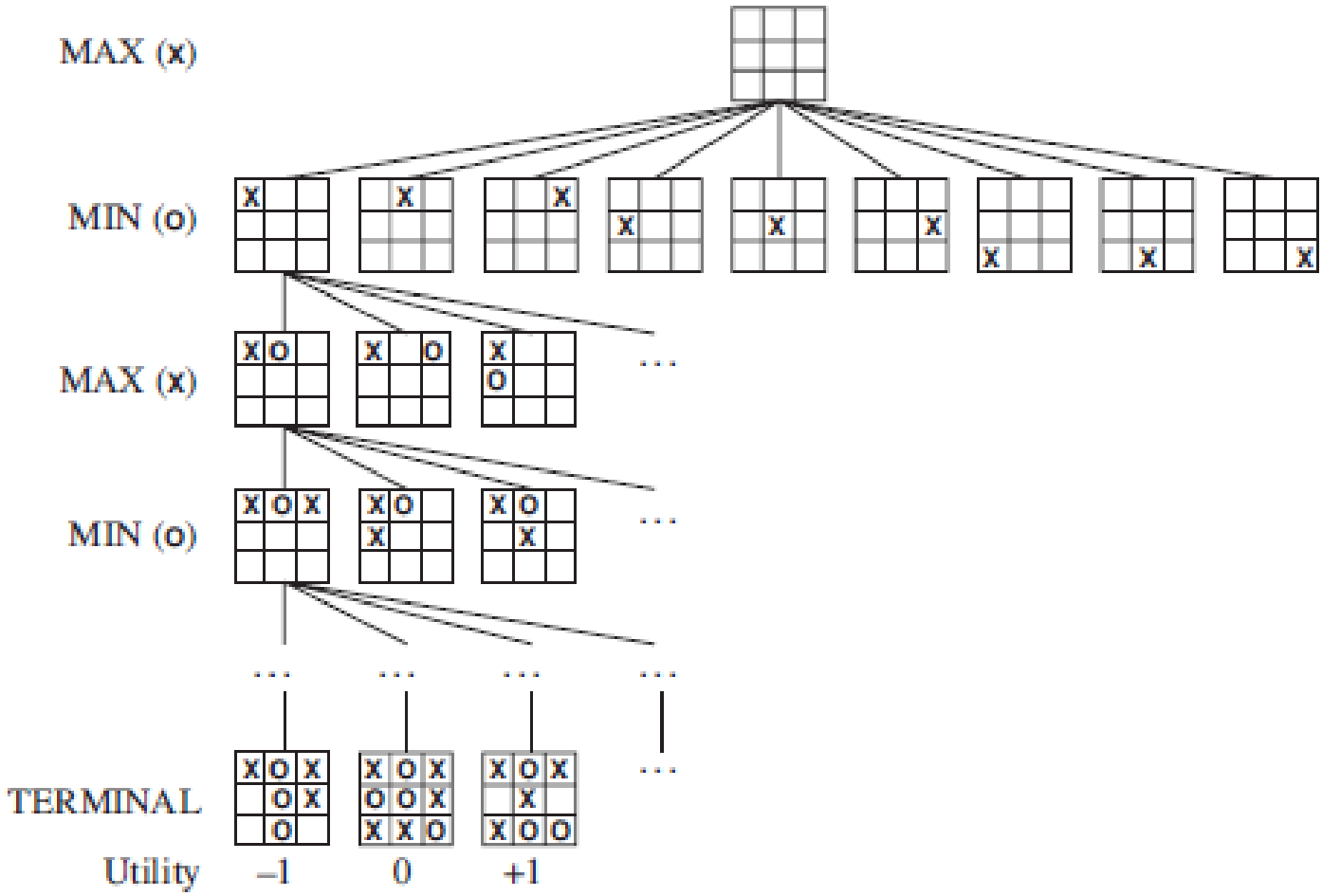


Figure 3.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN(O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Figure 3.1 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.

Regardless of the size of the game tree, it is MAX's job to search for a good move.

Search tree is a tree that is superimposed on the full game tree, and examines enough nodes to allow a player to determine what move to make.

3.2OPTIMAL DECISIONS IN GAMES

In a 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.

Roughly speaking, an **optimal strategy** leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.

Figure 3.2 shows a trivial game. 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.

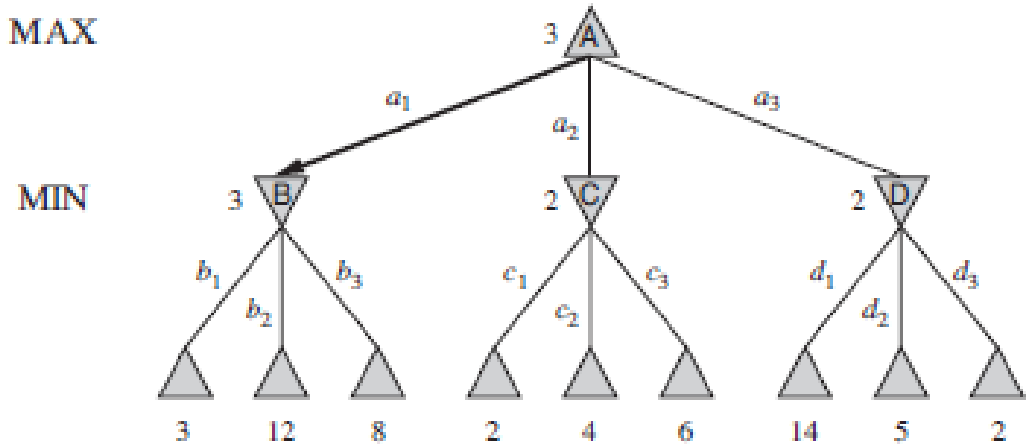


Figure 3.2 A two-ply game tree. The Δ nodes are "MAX nodes," in which it is MAX's turn to move, and the ∇ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads

to the state with the lowest minimax value.

Given a game tree, the optimal strategy can be determined from the **minimax value** of each node, 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.
- 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:

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

Let us apply these definitions to the game tree in Figure 3.2.

- 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 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 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.

3.2.1 The minimax algorithm

function MINIMAX-DECISION(state) **returns** an action
  return $\text{argmax}_{a \in \text{ACTIONS}(s)}$ 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
```

The minimax algorithm is an algorithm for calculating minimax decisions. It 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.

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).

This is called the **minimax decision** because it maximises the utility for Max on the assumption that Min will play perfectly to minimise it.

Minimax Algorithm Tracing:

For example, in Figure 3.2, 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 backedup 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

Minimax(A) = argmax{ MIN-VALUE(RESULT(A,a_1)), MIN-VALUE(RESULT(A,a_2)), MIN-VALUE(RESULT(A,a_3))}

= argmax{ MIN-VALUE(B), MIN-VALUE(C), MIN-VALUE(D)}

MIN-VALUE(B) calculation: v = ∞
for b_1: v=MIN(∞,3)=3
for b_2: v=MIN(3,12)=3
for b_3: v=MIN(3,8)=3
return 3

MIN-VALUE(C) calculation:
v = ∞
for c_1: v=MIN(∞,2)=2
for c_2: v=MIN(2,4)=2
for c_3: v=MIN(2,6)=2
return 2

MIN-VALUE(D) calculation: v = ∞
for b_1: v=MIN(∞,14)=14
for b_2: v=MIN(14,5)=5
for b_3: v=MIN(5,2)=2
return 2

Minimax(A) = argmax{ MIN-VALUE(B), MIN-VALUE(C), MIN-VALUE(D)}

= argmax{3, 2, 2}

= 3

Minimax Algorithm – Time and Space Complexity:

Suppose m is the maximum depth of the tree is m and there are b legal moves at each point (branching factor).

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.

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.

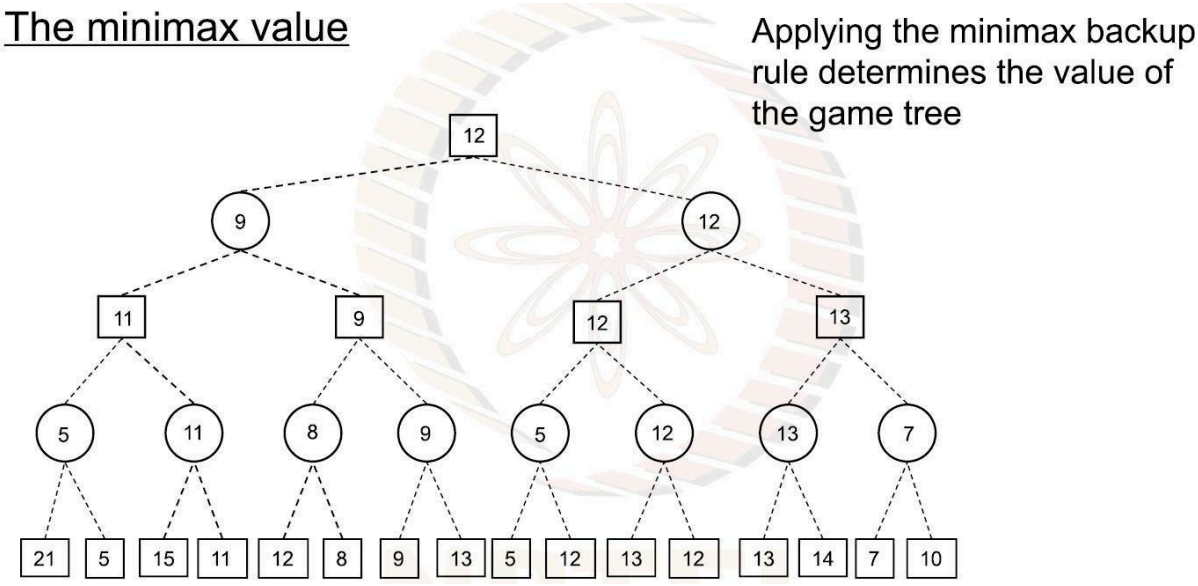


Figure 3.3 Application of Minimax Algorithm

3.2.2 Optimal decisions in multiplayer games

Extending the minimax idea to multiplayer games 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 $<v_A, v_B, v_C>$ is associated with each node.

For terminal states, this vector gives the utility of the state from each player's viewpoint.

The simplest way to implement this is to have the UTILITY function return a vector of utilities.

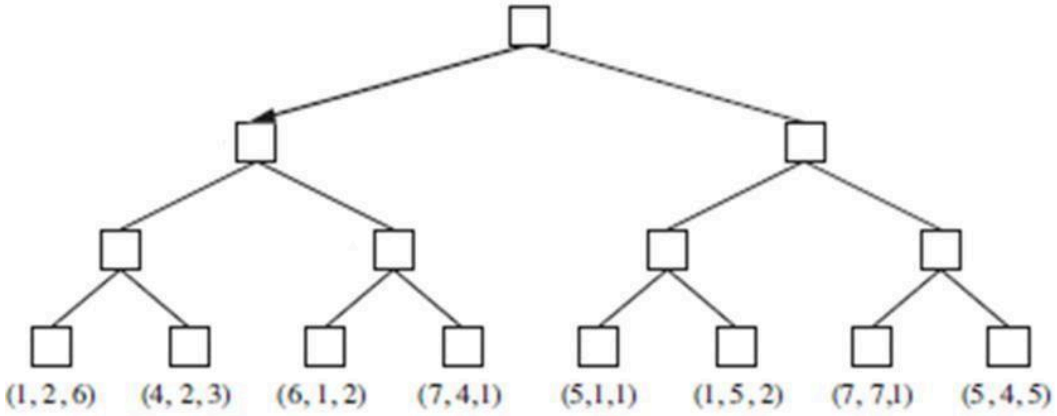
(In two-player, zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite.)

Consider nonterminal states.

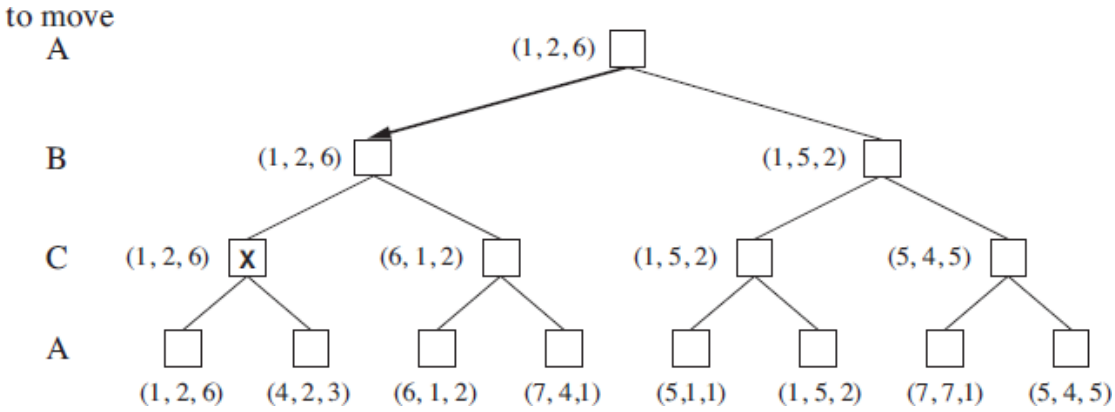
Consider the node marked X in the game tree shown in Figure 3.4. In that state, player C chooses what to do. The two choices lead to terminal states with utility vectors $<v_A=1, v_B=2, v_C=6>$ and $<v_A=4, v_B=2, v_C=3>$. Since 6 is bigger than 3, C should choose the first move.

This means that if state X is reached, subsequent play will lead to a terminal state with utilities $<v_A=1, v_B=2, v_C=6>$. 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.



(a)



(b)

Figure 3.4

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

Multiplayer games usually involve **alliances**, whether formal or informal, among the players. Alliances are made and broken as the game proceeds. 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 $<v_A = 1000, v_B = 1000>$ 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.