Artificial Intelligence & Neural Network

Module 2

UNIT 2-Adversarial search

- **multiagent environments**, in which each agent needs to consider the actions of other agents and how they affect its own welfare.
-
- 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**.

UNIT-2

- 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.*

ADVERSIAL SEARCH

- Mathematical **game theory**, a branch of economics, views any **multiagent environmen**t as a game, provided that the impact of each agent on the others is "significant," regardless of whether the agents are cooperative or competitive.

- The environment with more than one agent is termed as **multi-agent environment** where each agent is an **opponent** of other agent, playing **against each other** considering the **action of other agent** and **effect of that action** on their performance.

- So, **Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called adversarial searches, often known as Games**.
- .

TYPES OF GAMES IN AI:

	Deterministic	Chance Moves
Perfect information	Chess	monopoly
Imperfect information	tic-tac-toe	poker

- **Perfect information:** A game with the perfect information is that in which agents can **look into the complete board** such that they can see each other moves.

- **Imperfect information:** If in a game agents **do not have all information** about the game and not aware with what's going on.

- **Deterministic games:** Deterministic games are those games which **follow a strict pattern and set of rules** for the games, and there is no randomness associated with them. Examples are chess, Checkers, Go, tic-tac-toe, etc.

- **Non-deterministic games(stochastic games):** Non-deterministic are those games which have **various unpredictable events** and has a factor of chance by using dice or cards.

ZERO-SUM GAME

- Zero-sum games are adversarial search which involves pure competition.
- In Zero-sum game each agent's **gain or loss of utility** is exactly balanced by the losses or gains of utility of another agent.
- One player of the game try to maximize one single value, while other player tries to minimize it.
- **Each move by one player in the game is called as ply.**
- Chess and tic-tac-toe are examples of a Zero-sum game.

Zero-sum game: Embedded thinking

- The Zero-sum game involved embedded thinking in which one agent or player is trying to figure out:
- What to do.
- How to decide the move
- Needs to think about his opponent as well
- The opponent also thinks what to do

FORMALIZATION OF THE PROBLEM

A game can be defined as a type of search in AI which can be formalized of the following elements:

- **Initial state:** It specifies how the game is set up at the start.
- **Player(s):** It specifies which player has moved in the state space.
- **Action(s):** It returns the set of legal moves in state space.
- **Result(s, a):** It is the transition model, which specifies the result of moves in the state space.
- **Terminal-Test(s):** Terminal test is true if the game is over, else it is false at any case. The state where the **game ends** is called terminal states.
- **Utility(s, p):** A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called **payoff function**.

For Chess, the outcomes are a win, loss, or draw and its payoff values are +1, 0, ½. And for tic-tac-toe, utility values are +1, -1, and 0.

- 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**.

Games

- In a multi-agent 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.
- Here 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 multi-agent 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.

- 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.
- 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.
- 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.
- Finally, we look at how state-of-the-art game-playing programs fare against human opposition and at directions for future developments.

GAME TREE:

- A game tree is a tree where **nodes** of the tree are the **game states** and **Edges** of the tree are the **moves by players**.
- Game tree involves **initial state, actions function, and result Function**.
- **Example: Tic-Tac-Toe game tree:**
- The following figure is showing part of the game-tree for tic-tac-toe game.

Following are some key points of the game:

- There are two players MAX and MIN.
- Players have an alternate turn and start with MAX.
- MAX maximizes the result of the game tree
- MIN minimizes the result.

MIN-MAX Game

- 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:
  - 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 1/2.
  - 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+0or 1/2 + 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 1/2.

MIN-MAX Game

- 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.

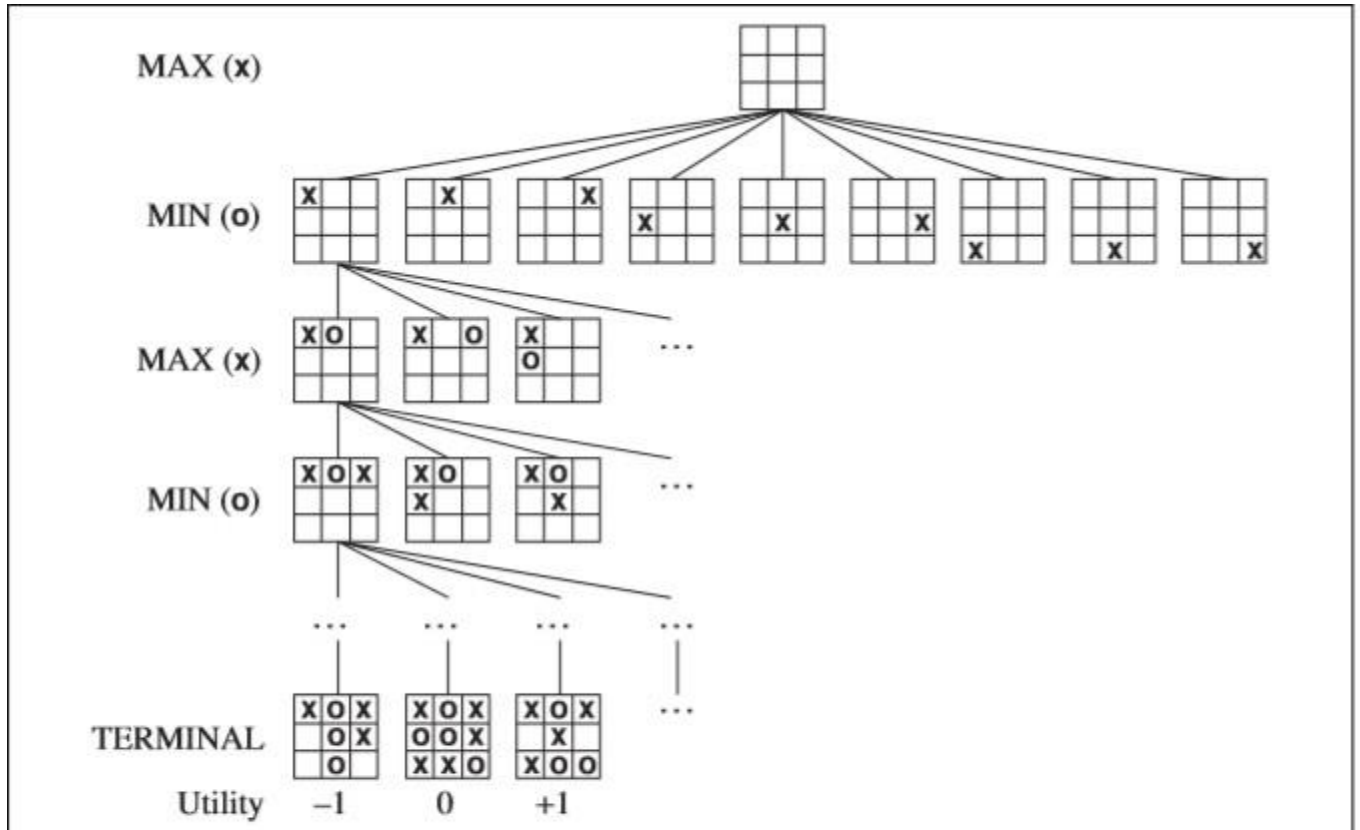


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities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

TIC TAC TOE Explanation:

- From the initial state, **MAX has 9 possible moves** as he starts first. MAX place x and MIN place o, and both player plays alternatively until we reach a leaf node where one player has three in a row or all squares are filled.
- Both players will compute each node, minimax, the minimax value which is the best achievable utility against an optimal adversary.
- Suppose both the players are well aware of the tic-tac-toe and playing the best play. Each player is doing his best to prevent another one from winning. MIN is acting against Max in the game.
- So in the game tree, we have a layer of Max, a layer of MIN, and each layer is called as **Ply**. Max place x, then MIN puts o to prevent Max from winning, and this game continues until the terminal node.
- In this either MIN wins, MAX wins, or it's a draw. This game-tree is the whole search space of possibilities that MIN and MAX are playing tic-tac-toe and taking turns alternately.

- Hence adversarial Search for the minimax procedure works as follows:
- It aims to find the optimal strategy for MAX to win the game.
- In the game tree, optimal leaf node could appear at any depth of the tree. It follows the approach of Depth-first search.
- Propagate the minimax values up to the tree until the terminal node discovered.
- In a given game tree, the optimal strategy can be determined from the minimax value of each node, which can be written as MINIMAX(n). MAX prefer to move to a state of maximum value and MIN prefer to move to a state of minimum value then:

- 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 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.

Optimal Decisions in Games

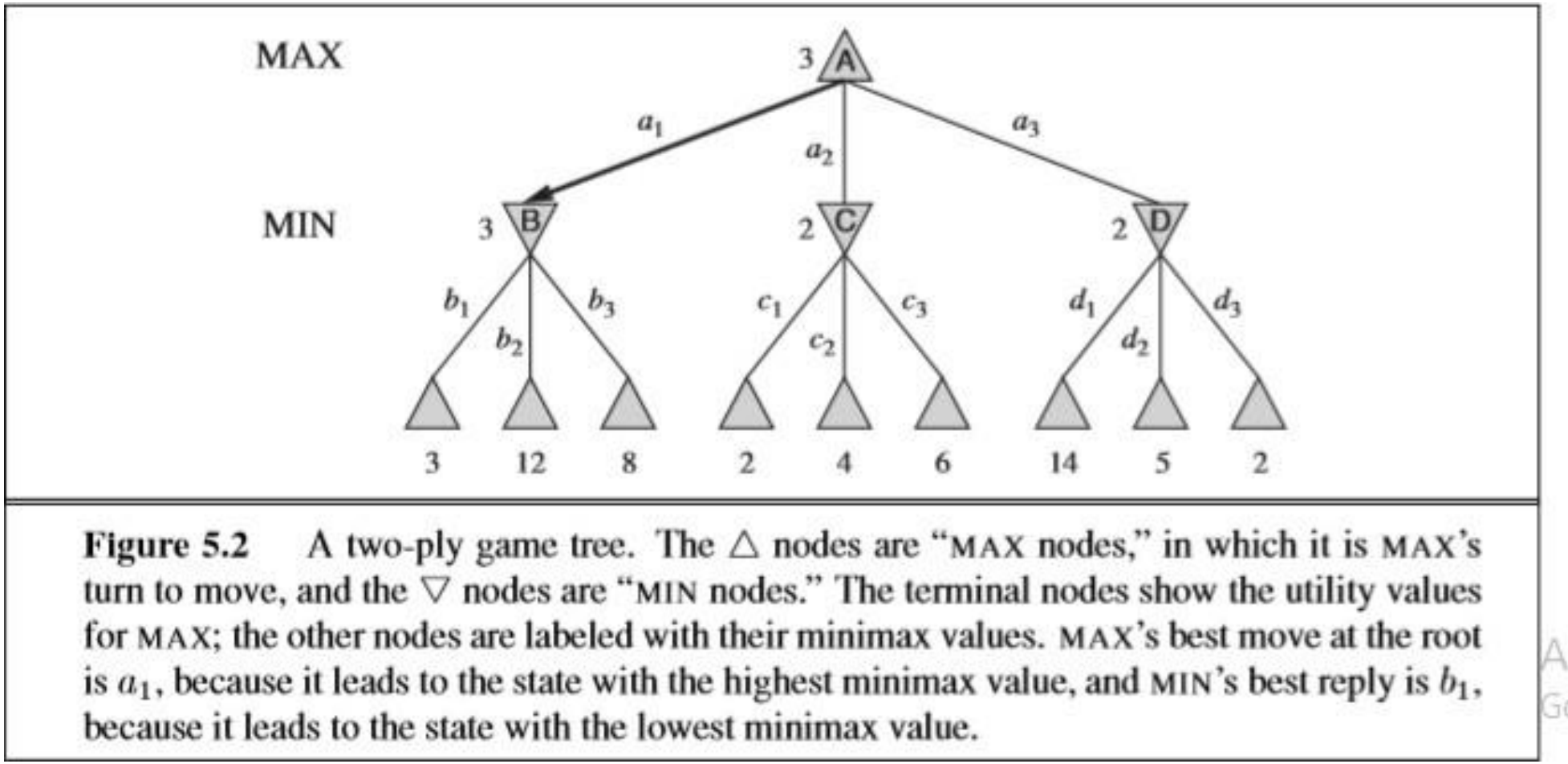


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state 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, 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:

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$

-

The minimax algorithm

```
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 $f(a)$.

- 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

MINI-MAX ALGORITHM IN ARTIFICIAL INTELLIGENCE

- Mini-max algorithm is a recursive or backtracking algorithm
- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various two-players game. This Algorithm computes the minimax decision for the current state.
- In this algorithm two players play the game, one is called MAX and other is called MIN.
- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

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.
- 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

Optimal decisions in multiplayer games

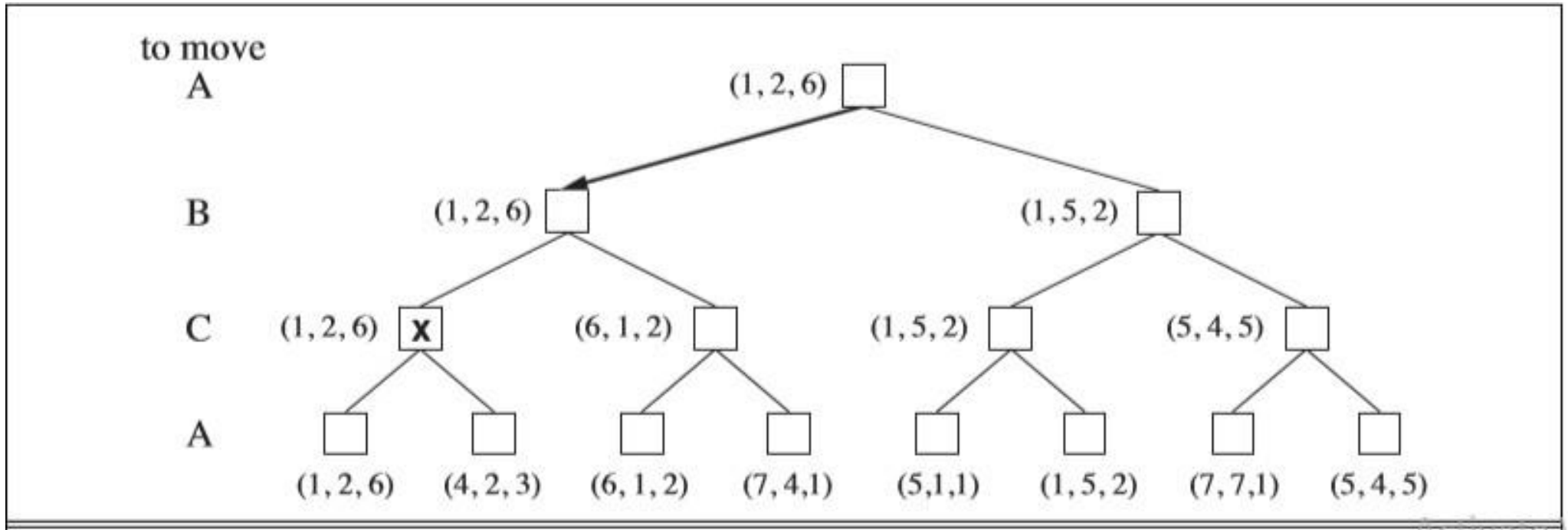


Figure 5.4 The first three plies of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Consider the node marked X in the game tree shown in Figure . 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 $<v_A = 4, v_B = 2, v_C = 3>$ Since6isbiggerthan3, 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.

Optimal decisions in multiplayer games

- 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.
- 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.

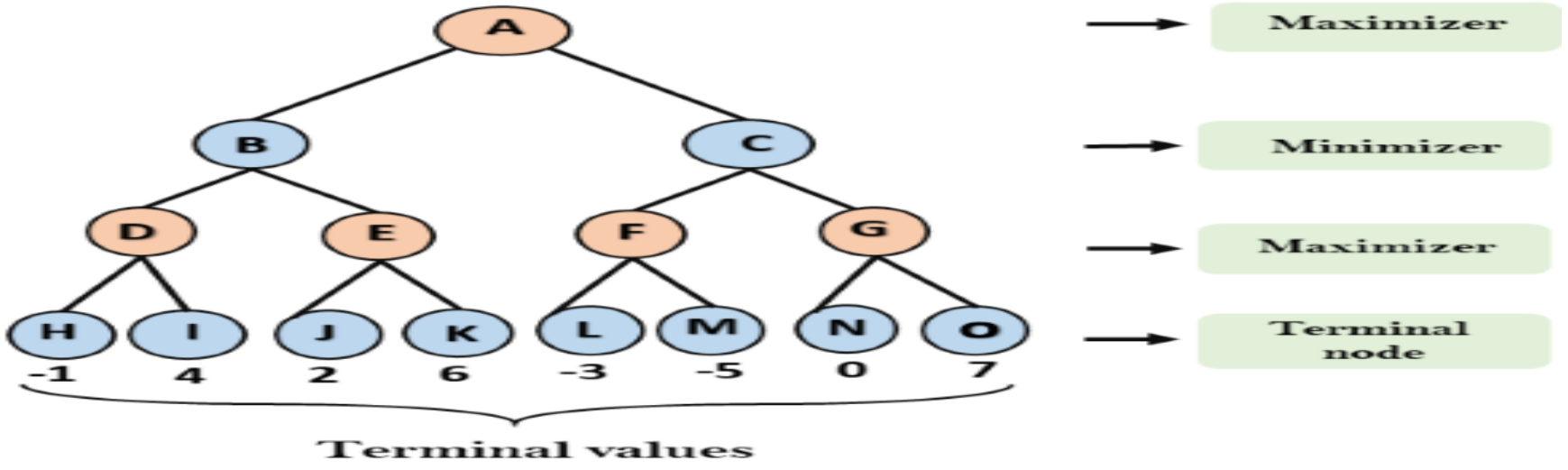
An Optimal Procedure for MINIMAX

- Designed to find the optimal strategy for Max and find best move:
  1. Generate the whole game tree to leaves
  2. Apply evaluation function to leaves
  3. Back-up values from leaves toward the root
     a. Max node computes the max of its child values
     b. Min node computes the min of its child values
  4. When value reaches the root:

     Choose Max value and the corresponding Move.

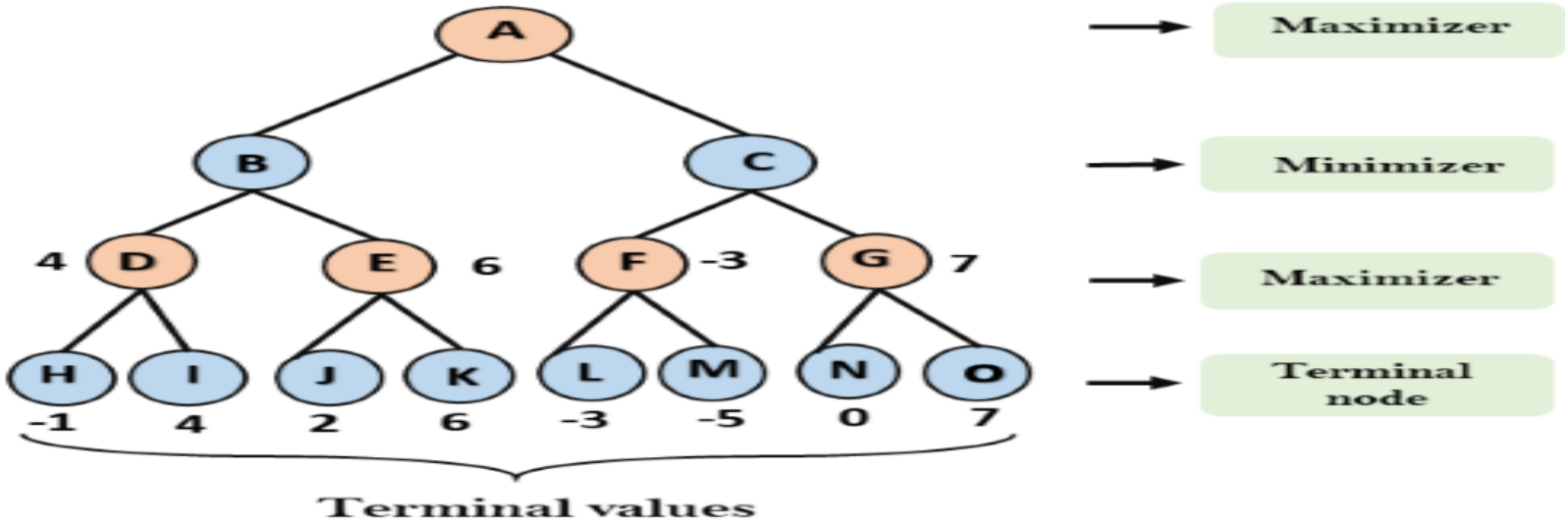
  However: It is impossible to develop the whole search tree, instead develop part of the tree and evaluate static evaluation function.

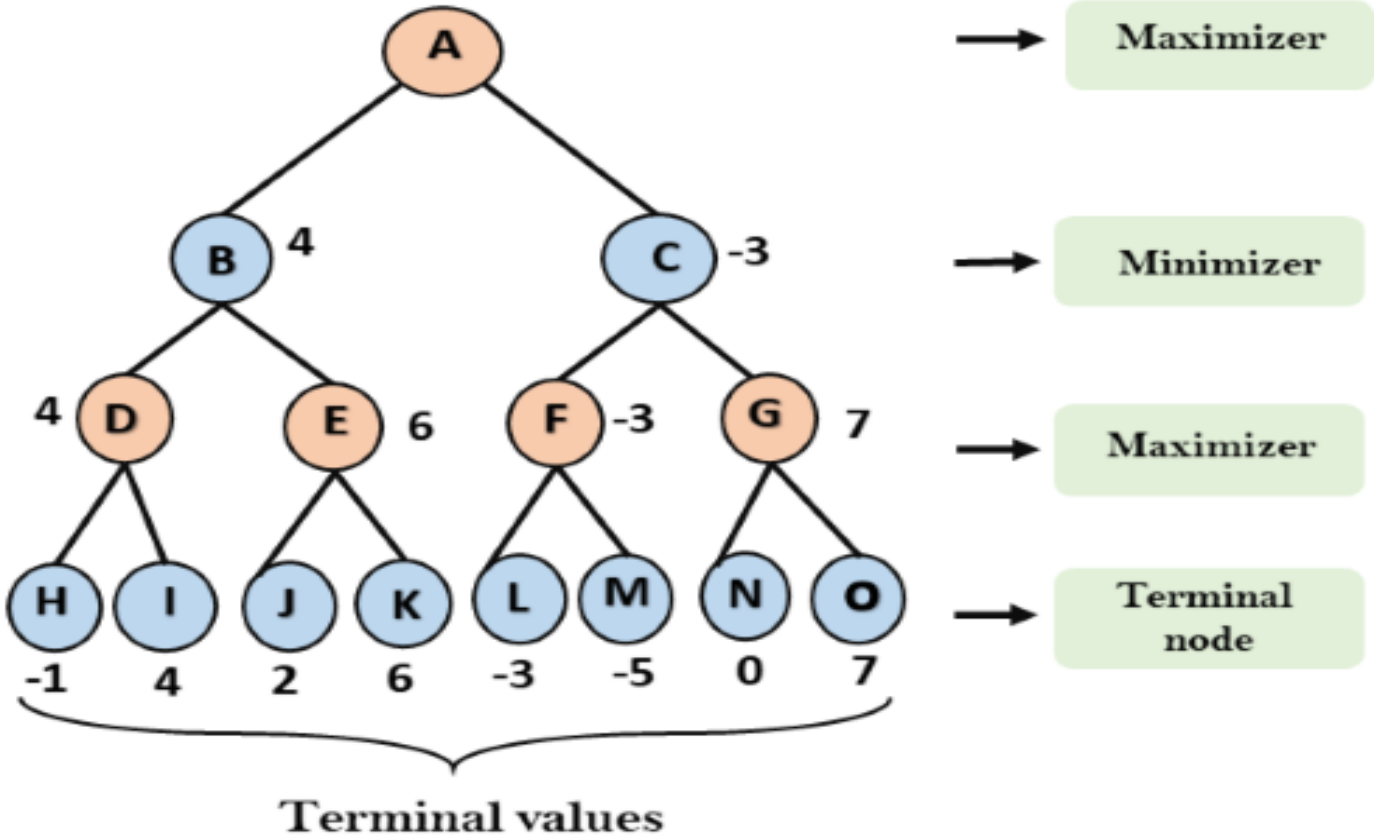
- **Step-1:** In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.



Step 2: Now, first we find the utilities value for the Maximizer, its initial value is **-∞**, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

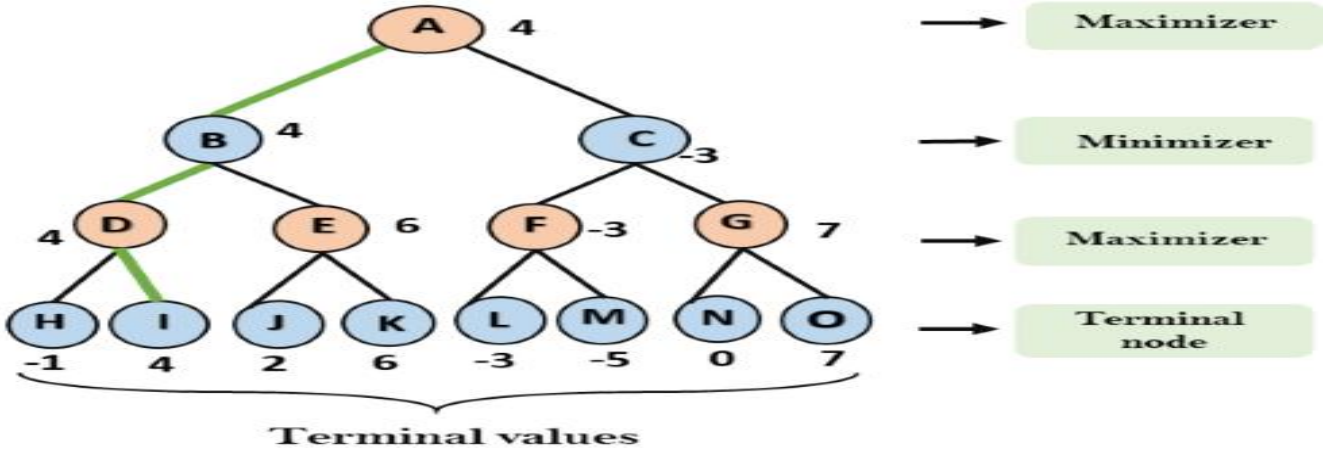
- For node D max(-1,- -∞) => max(-1,4)= 4
- For Node E max(2, -∞) => max(2, 6)= 6
- For Node F max(-3, -∞) => max(-3,-5) = -3
- For node G max(0, -∞) = max(0, 7) = 7





Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with **+∞**, and will find the 3rd layer node values.

- For node B= min(4,6) = 4
- For node C= min (-3, 7) = -3



Step 4: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

- For node A max(4, -3)= 4

Properties of Mini-Max algorithm:

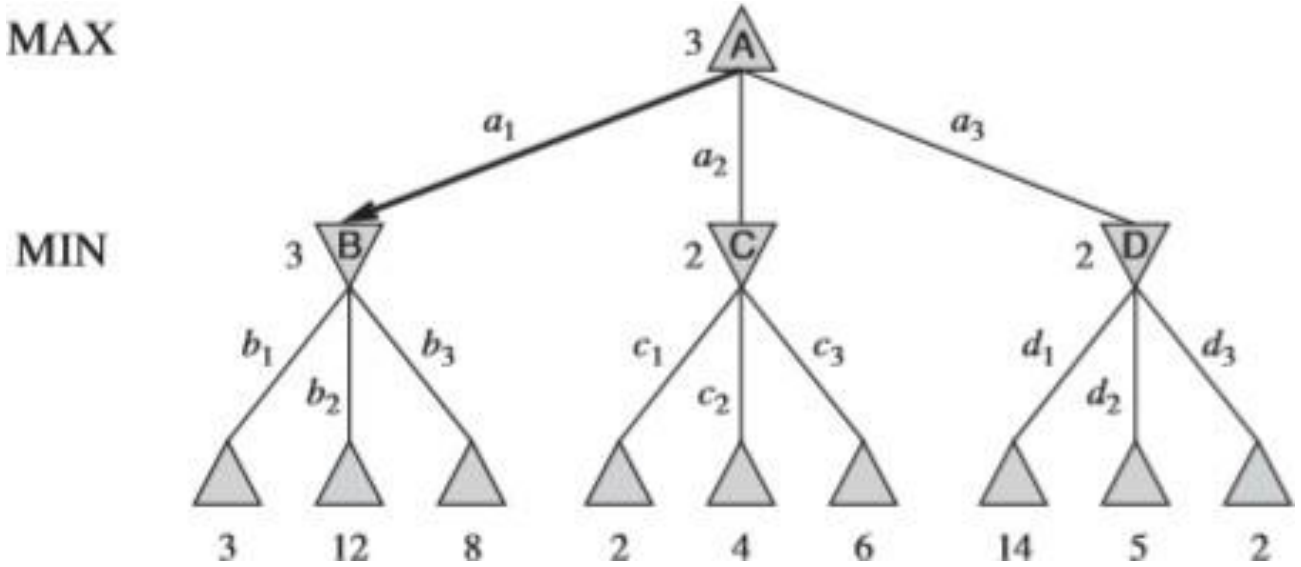
- **Complete-** Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- **Optimal-** Min-Max algorithm is optimal if both opponents are playing optimally.
- **Time complexity-** As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is **$O(b^m)$**, where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- **Space Complexity-** Space complexity of Mini-max algorithm is also similar to DFS which is **O(bm)**.

Limitation of the minimax Algorithm:

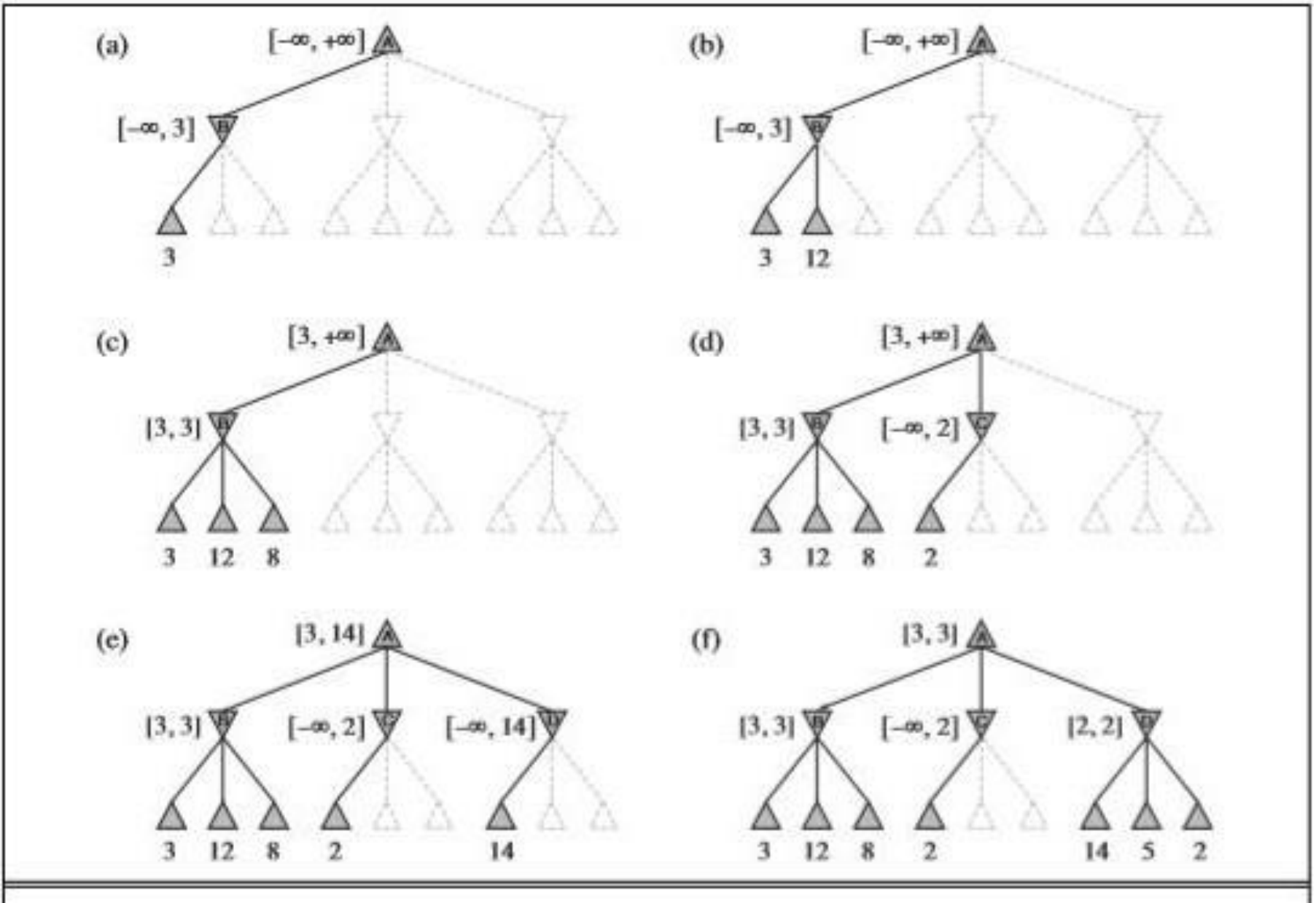
The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide. This limitation of the minimax algorithm can be improved from **alpha-beta pruning** which we have discussed in the next topic.

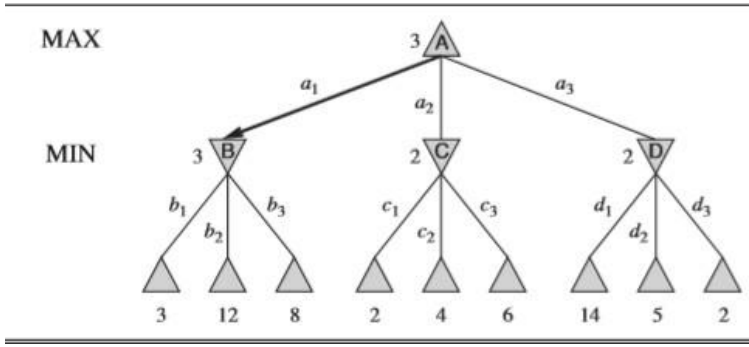
Alpha Beta Pruning

- The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree.
- The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree. Alpha Beta Pruning is used for this purpose.
- 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





Alpha Beta Pruning

- 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

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

- 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.

Alpha Beta Pruning

- The general principle is this: consider a node n 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.

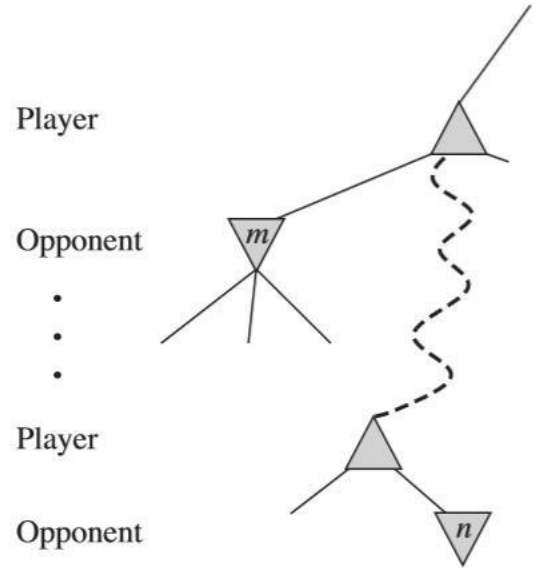


Figure 5.6 The general case for alpha–beta pruning. If m is better than n for Player, we will never get to n in play.

Alpha Beta Pruning

- 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. 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.

Alpha Beta Search Algorithm

```
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).

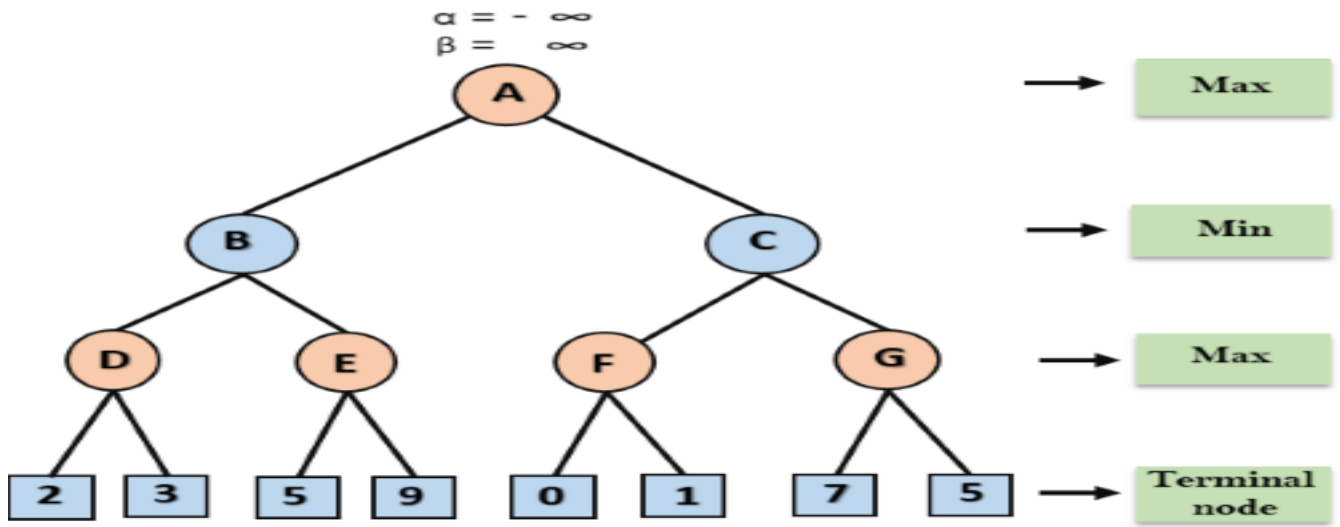
- Condition for Alpha-beta pruning: **α>=β**
- Key points about alpha-beta pruning:

- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

Condition for Alpha-beta pruning

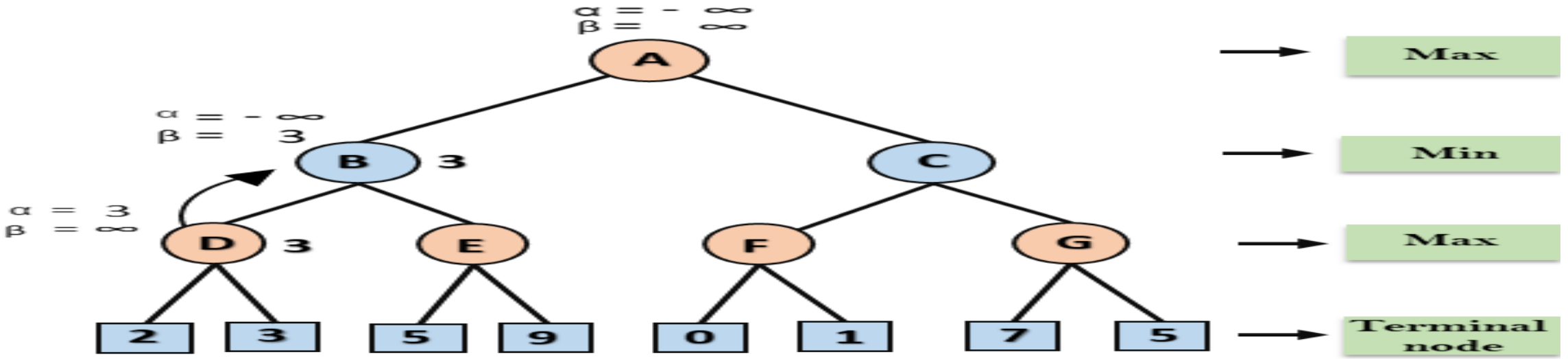
- Alpha: At any point along the Maximizer path, Alpha is the best option or the highest value we've discovered. The initial value for alpha is – ∞.
- Beta: At any point along the Minimizer path, Beta is the best option or the lowest value we've discovered.. The initial value for alpha is + ∞.
- The condition for Alpha-beta Pruning is that α >= β.
- The alpha and beta values of each node must be kept track of. Alpha can only be updated when it's MAX's time, and beta can only be updated when it's MIN's turn.
- MAX will update only alpha values and the MIN player will update only beta values.
- The node values will be passed to upper nodes instead of alpha and beta values during going into the tree's reverse.
- Alpha and Beta values only are passed to child nodes.

Step 1: At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.



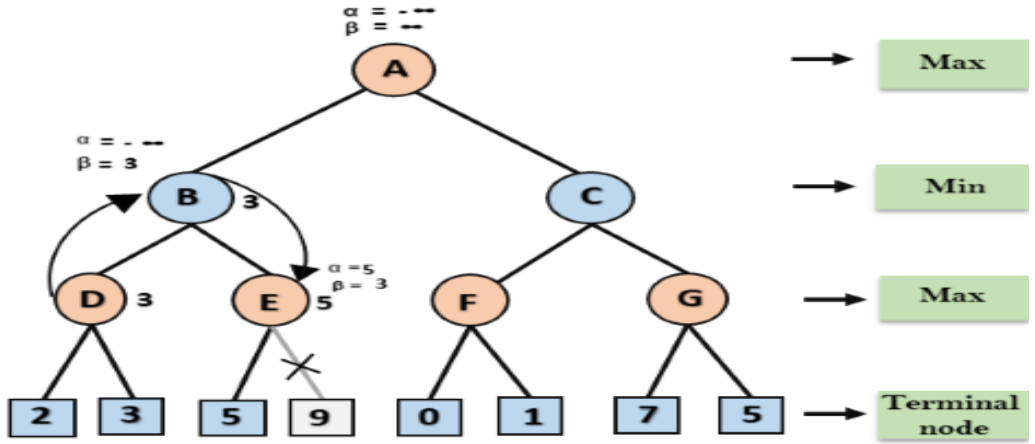
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.

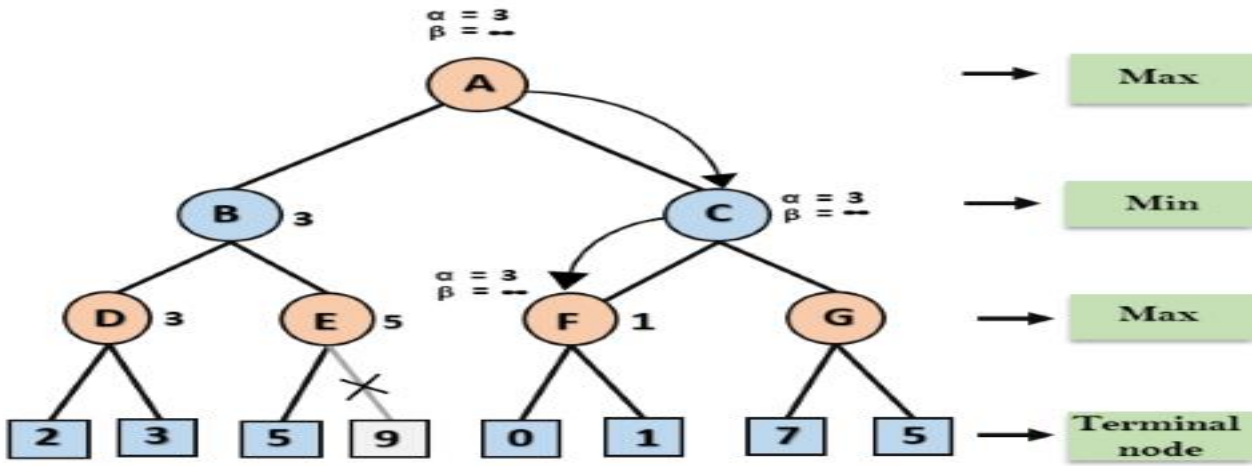


In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

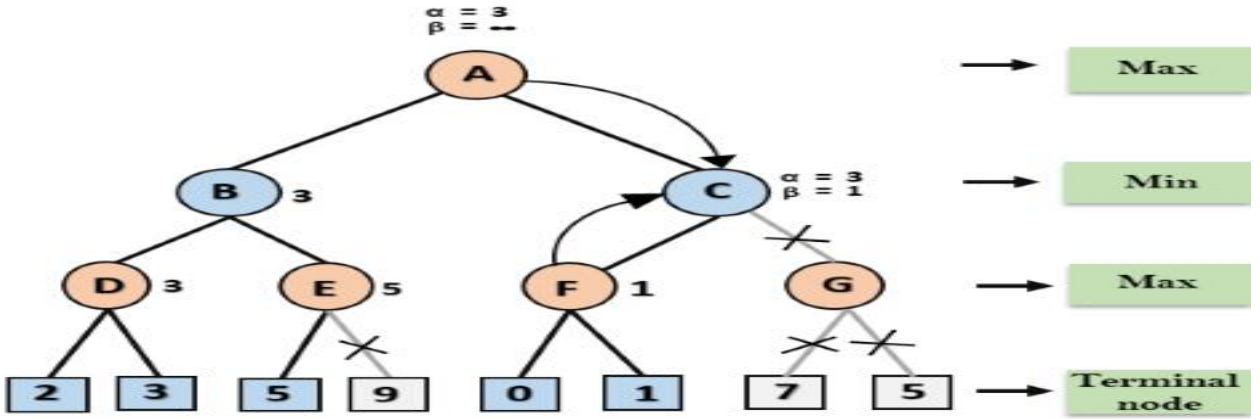
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.



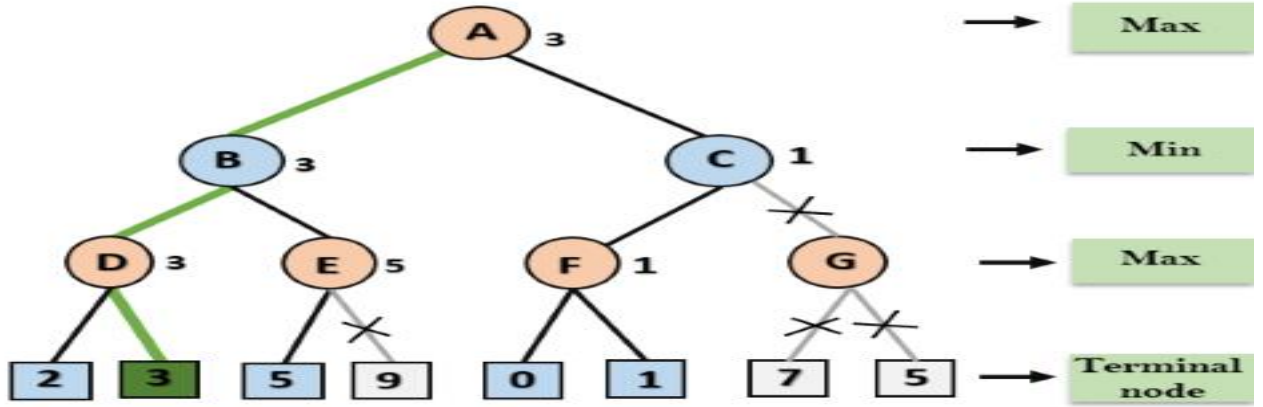
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.
At node C, α=3 and β= +∞, and the same values will be passed on to node F.
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



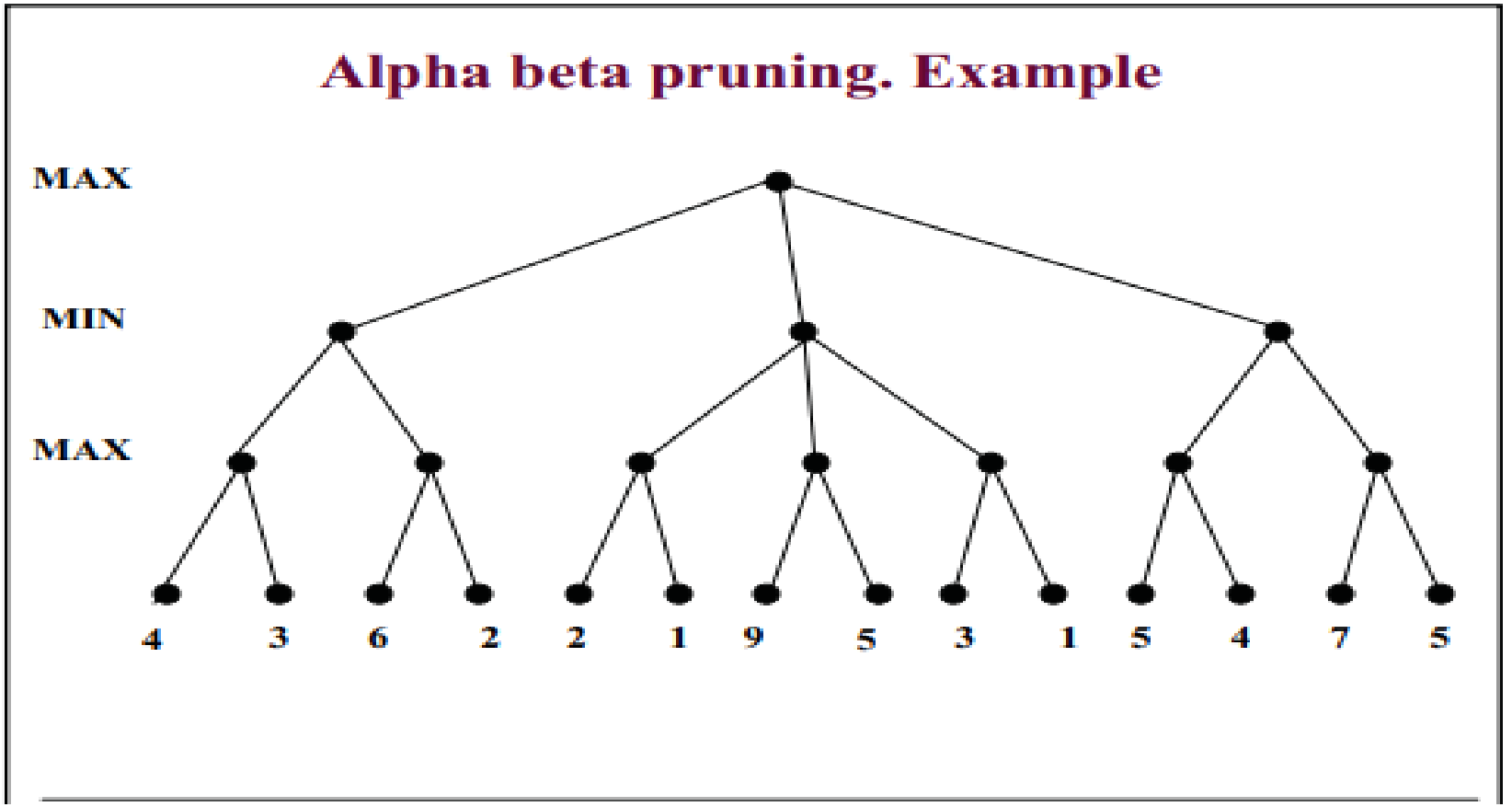
- **Step 7:** Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.



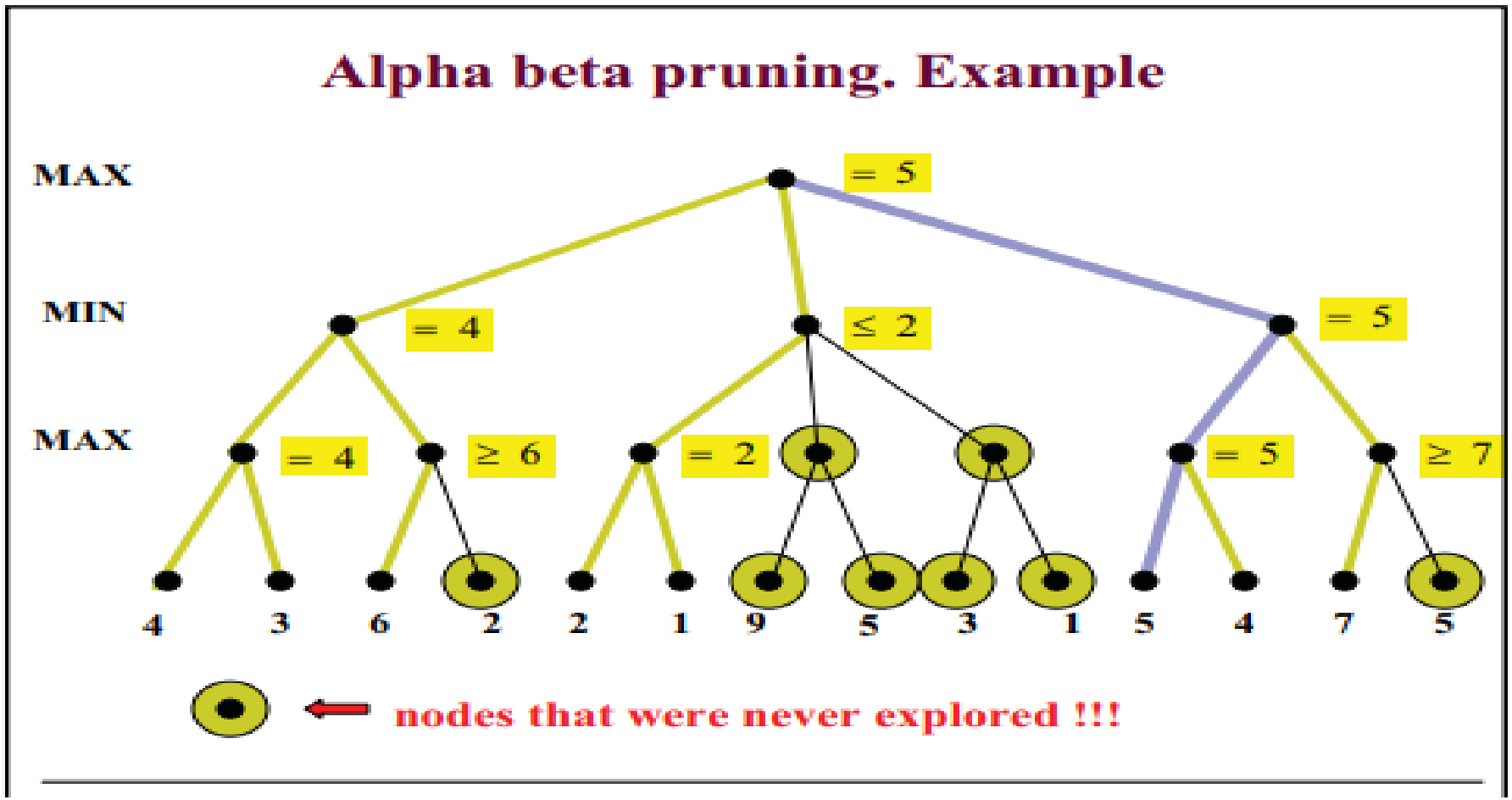
- **Step 8:** C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



Alpha beta pruning. Example



Alpha beta pruning. Example



SUMMARY

- 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.
- The **Best case time complexity** of the alpha-beta pruning algorithm is **O(bd/2)** (b is the branching factor and d is the depth of the game tree). In contrast, in the **worst case**, it will take time similar to the minimax algorithm **O(bd)** because we'll have to explore all the nodes.
- **Space Complexity-** Space complexity is **O(bm)**.

COMPLEXITY OF ALPHA BETA PRUNING

The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning.

It can be of two types:

Worst ordering: In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is **$O(b^m)$.**

Ideal ordering: The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering **is $O(b^{m/2})$.**

Constraints Satisfaction Problems

- A 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 state.

- From the point of view of the search algorithm, however, each state is atomic, or indivisible—a black box with no internal structure.

- We describe 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.

- 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.

DEFINING CONSTRAINT SATISFACTION PROBLEMS

- A constraint satisfaction problem consists of three components, V,D, and C:
  - **V is a set of variables, {V1,...,Vn}.**
  - **D is a set of domains, {D1,...,Dn}, one for each variable.**
  - **C is a set of constraints that specify allowable combinations of values.**
- Each domain Di consists of a set of allowable values, {v1,...,vk} for variable Vi.
- Each constraint Ci consists of a pair **<scope,rel>** 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 X1 and X2 both have the domain {A,B}, then the constraint saying the two variables must have different values can be written as <(X1,X2),[(A,B),(B,A)] or as <(X1,X2),X1 ≠ X2>

DEFINING CONSTRAINT SATISFACTION PROBLEMS

- 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, {Xi = vi, Xj = vj,...}.
- 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

Example problem: Map coloring

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 Di = {red, green,blue}.

- The constraints require neighboring regions to have distinct colors. Since there are nine places where regions border, there are nine constraints:

C = {SA = WA,SA = NT,SA = Q,SA = NSW,SA = V, WA = NT,NT = Q,Q = NSW,NSW = V } .

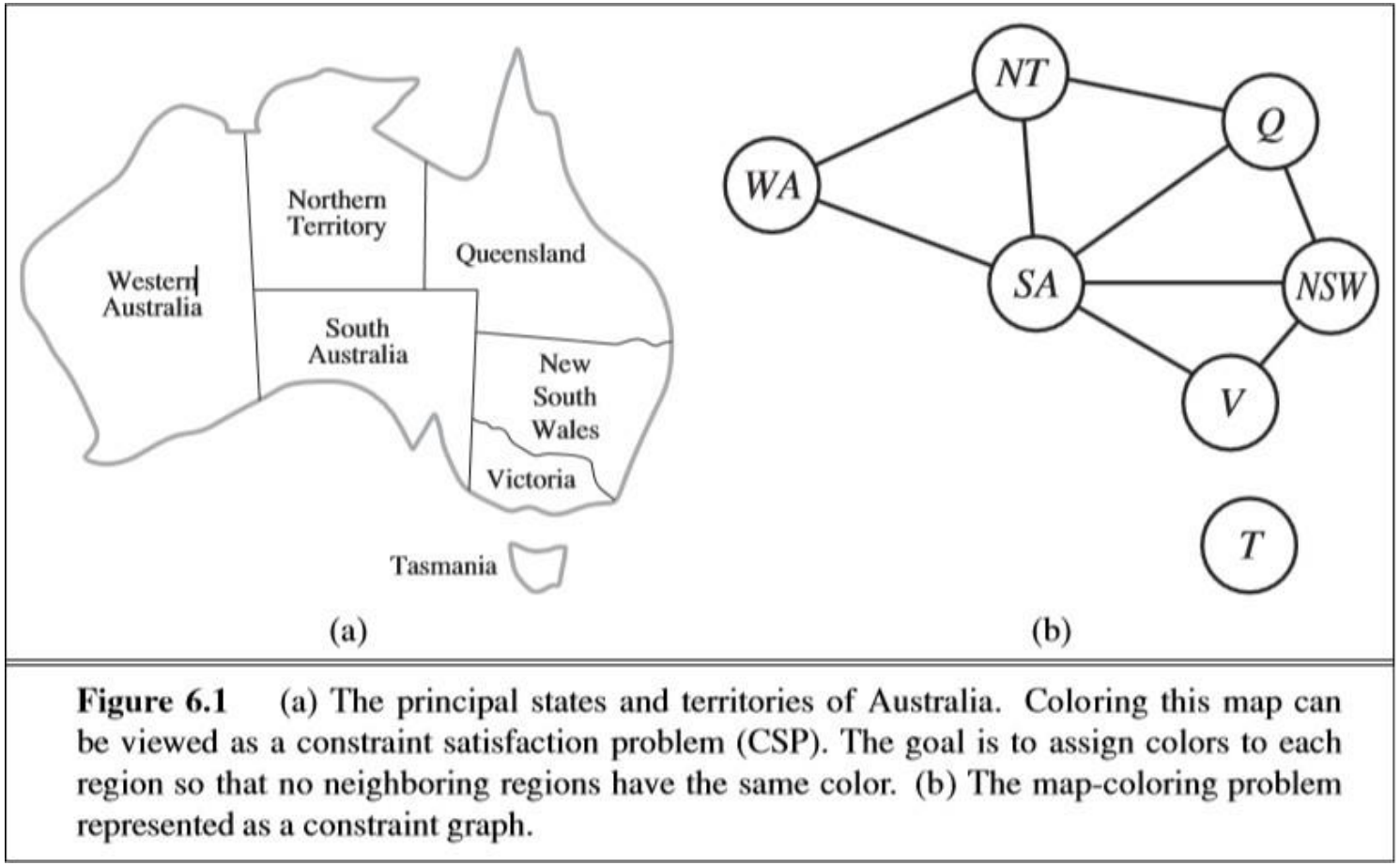


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

SA = WA is a shortcut for <(SA,WA),SA ≠WA>where SA = WA can be fully enumerated in turn as {(red ,green),(red ,blue),(green,red),(green,blue),(blue,red),(blue,green)} .

Example problem: Map coloring

- There are many possible solutions to this problem, such as

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

- 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.

Example: Job Shop Scheduling

- Factories have the problem of scheduling a day's worth of jobs, subject to various constraints.
- 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

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

Example: Job Shop Scheduling

- The value of each variable is the time that the task starts. Next we represent precedence constraints between individual tasks. Whenever a task T1 must occur before task T2, and task T1 takes duration d1 to complete, we add an arithmetic constraint of the form

$$T1 + d1 \leq T2$$

- In this 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

$$AxleF + 10 \leq WheelRF ;$$
$$AxleF + 10 \leq WheelLF;$$
$$AxleB + 10 \leq WheelRB;$$
$$AxleB + 10 \leq WheelLB .$$

- For each wheel, we must affix the wheel(which takes1minute), then tighten the nuts (2 minutes), and finally attach the hubcap (1 minute, but not represented yet):

$$WheelRF + 1 \leq NutsRF ;$$
$$NutsRF + 2 \leq CapRF ;$$
$$WheelLF + 1 \leq NutsLF;$$
$$NutsLF + 2 \leq CapLF;$$
$$WheelRB + 1 \leq NutsRB;$$
$$NutsRB + 2 \leq CapRB;$$
$$WheelLB + 1 \leq NutsLB;$$
$$NutsLB + 2 \leq CapLB .$$

Example: Job Shop Scheduling

- 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 AxleF and AxleB must not overlap in time; either one comes first or the other does:

$$(\text{AxleF} + 10 \leq \text{AxleB}) \text{ or } (\text{AxleB} + 10 \leq \text{AxleF}) .$$

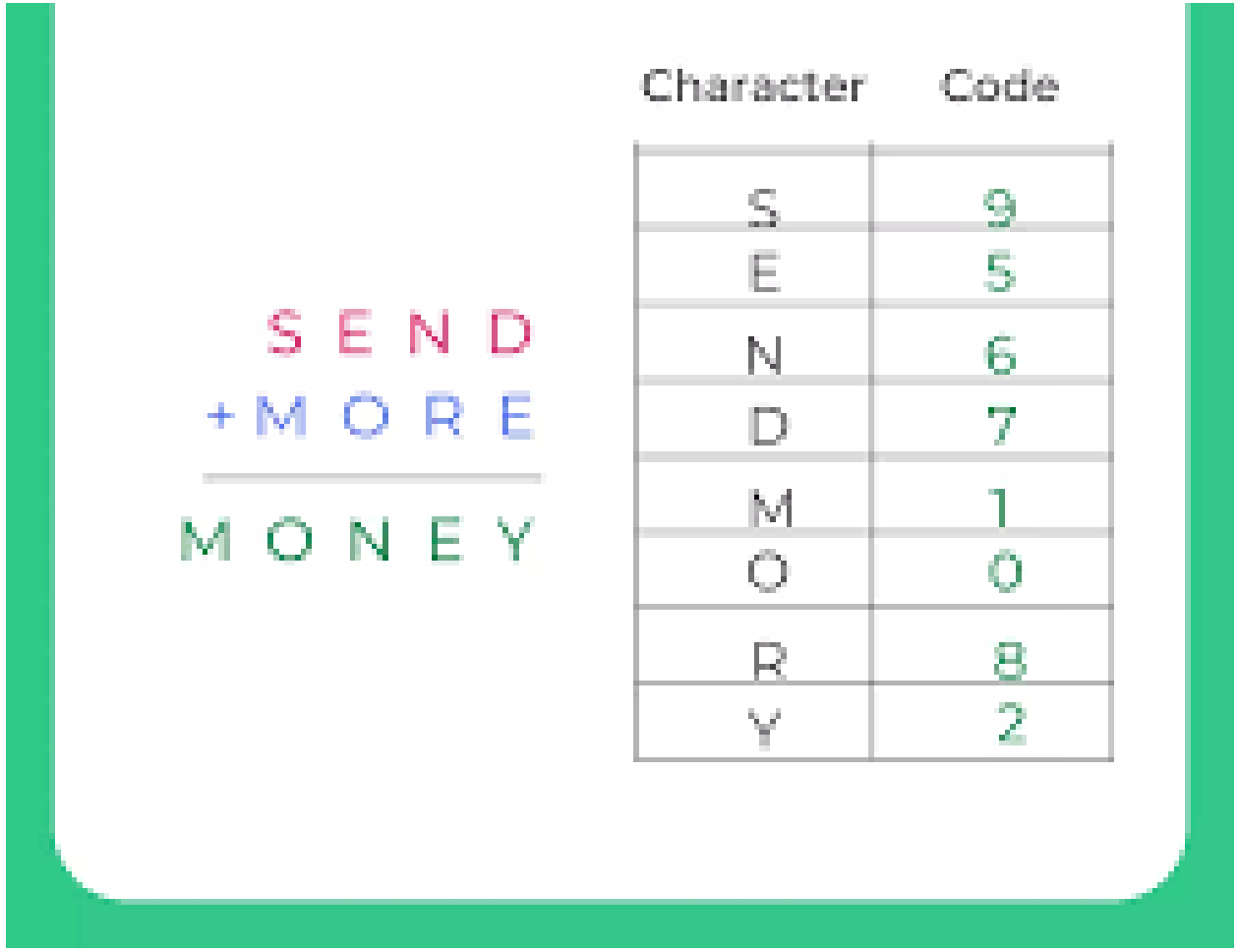
- We also need to assert that the inspection comes last and takes 3 minutes. For every variable except Inspect we add a constraint of the form

$$X + dX \leq \text{Inspect}.$$

- 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: Di = {1,2,3,...,27}
- This particular problem is trivial to solve, but CSPs have been applied to job-shop scheduling problems like this with thousands of variables.

Crypt arithmetic Problem

https://youtu.be/HC6Y49iTg1k?si=6lw5YwPf2jLCuaxD



Crypt arithmetic Problem

1.

```
  BASE
 +BALL
------
 GAMES
------
```


B	7
A	4
S	8
E	3
L	5
G	1
M	9

2.

```
  YOUR
 +YOU
------
 HEART
------
```


Y	9
O	4
U	2
R	6
H	1
E	0
A	3
T	8

Crypt arithmetic

More Cryptarithmetic Problem

```
BLACK      7 9 2 0 8
GREEN      5 3 4 4 6
-----------  ---------------
ORANGE   1 3 2 6 5 4
```

```
CRASH        3 6 8 4 5
HACKER    5 8 3 9 2 6
-----------  ------------------
REBOOT    6 2 0 7 7 1
```

```
CROSS      9 6 2 3 3
ROADS      6 2 5 1 3
-----------  ---------------
DANGER   1 5 8 7 4 6
```

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 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. The key idea is local consistency

Local Consistency

- If we treat each variable as a node in a graph 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
  - Node consistency
  - Arc consistency

- **Node Consistency:** A single variable 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}.
- 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

Local 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, Xi is arc-consistent with respect to another variable Xj if for every value in the current domain Di there is some value in the domain Dj that satisfies the binary constraint on the arc (Xi,Xj).
- 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

$$<(X,Y),\{(0,0),(1,1),(2,4),(3,9)\}>$$

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, thenY 's domain becomes {0,1,4,9} and the whole CSP is arc-consistent.

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 size0).
- 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.

Path Consistency

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

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 k^{th} 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.
- 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 X1. We are then guaranteed to be able to choose a value for X2 because the graph is 2-consistent, for X3 because it is 3-consistent, and so on.
  - Foreach variable Xi,we need only search through the d values in the domain to find a value consistent with X1,...,X i−1. We are guaranteed to find a solution in time $O(n^2d)$.

Local Search for CSPs

- Local search algorithms 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, 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.
- 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 heuristic.**

Local Search for CSPs

```
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.

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)

Logical Agents

Logical Agents

- Agents that reason logically
  - Knowledge based agents
  - Logic and Representations
  - Propositional (Boolean) Logic
- Knowledge based agents can accept new tasks in the form of explicitly described goals; they can achieve competence quickly by being told or learning new knowledge about the environment; and they can adapt to changes in the environment by updating the relevant knowledge.

Knowledge Based Agents

- The central component of a knowledge-based agent is its knowledge base, or KB.
- A knowledge base is a set of sentences.
- Each sentence is expressed in a language called a knowledge representation language and represents some assertion about the world.
- There must be a way to add new sentences to the knowledge base and a way to query what is known. The standard names for these operations are TELL and ASK, respectively.
- Both operations may involve inference—that is, deriving new sentences from old. Inference must obey the requirement that when one ASKs a question of the knowledge base, the answer should follow from what has been told (or TELLed) to the knowledge base previously
- Inference must obey the requirement that when one ASKs a question of the knowledge base, the answer should follow from what has been told (or TELLed) to the knowledge base previously

Knowledge Based Agent

- It takes a percept as input and returns an action.
- The agent maintains a knowledge base, KB, which may initially contain some background knowledge.
- Each time the agent program is called, it does **three things**.
- First, it TELLs the knowledge base what it perceives.
- Second, it ASKs the knowledge base what action it should perform. In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on.
- Third, the agent program TELLs the knowledge base which action was chosen, and the agent executes the action.

```
function KB-AGENT(percept) returns an action
  persistent: KB, a knowledge base
              t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

Figure 7.1 A generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

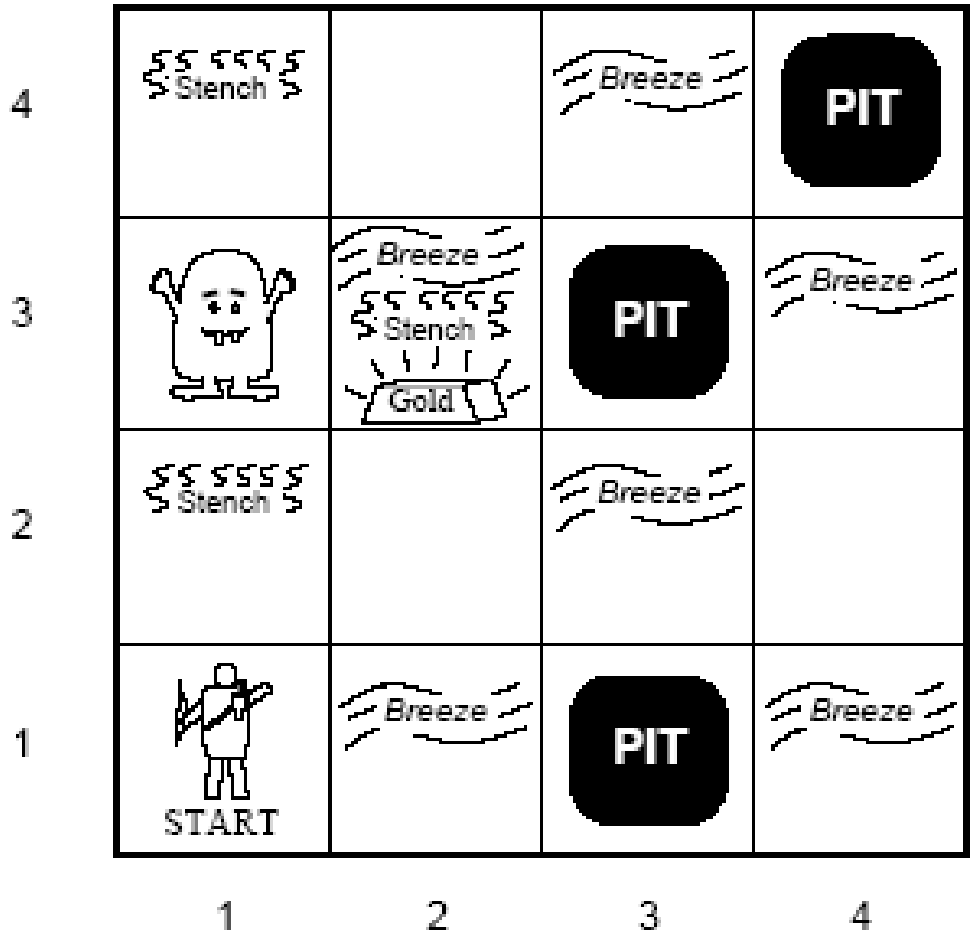
Knowledge Based Agent

- The details of the representation language are hidden inside **three functions** that implement the interface between the sensors and actuators on one side and the core representation and reasoning system on the other.
- MAKE-PERCEPT-SENTENCE constructs a sentence asserting that the agent perceived the given percept at the given time.
- MAKE-ACTION-QUERY constructs a sentence that asks what action should be done at the current time.
- MAKE-ACTION-SENTENCE constructs a sentence asserting that the chosen action was executed.
- The details of the inference mechanisms are hidden inside TELL and ASK.
- Because of the definitions of TELL and ASK, however, the knowledge-based agent is not an arbitrary program for calculating actions.

- It is amenable to a description at the knowledge level, where we need specify only what the agent knows and what its goals are, in order to fix its behavior.
- For example, an automated taxi might have the goal of taking a passenger from San Francisco to Marin County and might know that the Golden Gate Bridge is the only link between the two locations. Then we can expect it to cross the Golden Gate Bridge because it knows that that will achieve its goal.
- Notice that this analysis is independent of how the taxi works at the **implementation level**. It doesn't matter whether its geographical knowledge is implemented as linked lists or pixel maps, or whether it reasons by manipulating strings of symbols stored in registers or by propagating noisy signals in a network of neurons.
- A knowledge-based agent can be built simply by TELLing it what it needs to know. Starting with an empty knowledge base, the agent designer can TELL sentences one by one until the agent knows how to operate in its environment. This is called the **declarative approach** to system building.
- In contrast, the **procedural approach** encodes desired behaviors directly as program code.

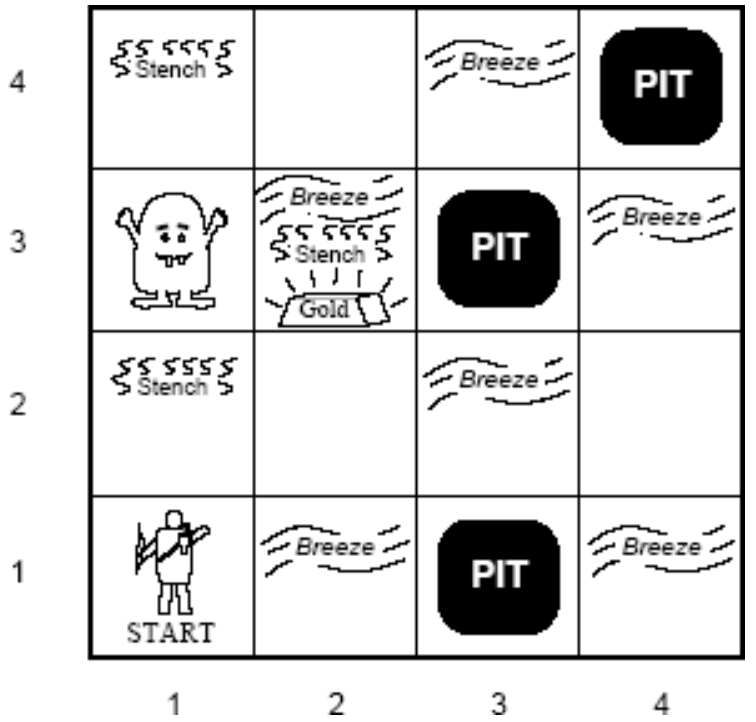
The WUMPUS WORLD Example

- Wumpus World is Computer Game.
- The wumpus world is a cave consisting of rooms connected by passage ways.
- Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room.
- The wumpus can be shot by an agent, but the agent has only one arrow.
- Some rooms contain bottomless pits that will trap anyone who wanders into these rooms.
- The only mitigating feature of this bleak environment is the possibility of finding a heap of gold.



The WUMPUS WORLD Example - PEAS Description

- **Performance measure:**
  - +1000 for climbing out of the cave with the gold,
  - –1000 for falling into a pit or being eaten by the wumpus,
  - –1 for each action taken and –10 for using up the arrow.
  - The game ends either when the agent dies or when the agent climbs out of the cave.
- **Environment:**
  - A 4×4 grid of rooms.
  - The agent always starts in the square labeled [1,1], facing to the right.
  - The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square.
  - In addition, each square other than the start can be a pit, with probability 0.2.



The WUMPUS WORLD Example - PEAS Description

- **Actuators:**
  - **go forward**
  - **turn right** 90 degrees
  - **turn left** 90 degrees
  - **grab**: Pick up an object that is in the same square as the agent
  - **shoot**: Fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits and kills the wumpus or hits the outer wall. The agent has only one arrow, so only the first Shoot action has any effect
  - **climb** is used to leave the cave. This action is only effective in the start square
  - **die**: This action automatically and irretrievably happens if the agent enters a square with a pit or a live wumpus
- **Sensors:**

The agent has five sensors, each of which gives a single bit of information: –

- In the square containing the wumpus and in the directly (not diagonally) adjacent squares, the agent will perceive a **Stench**.
- In the squares directly adjacent to a pit, the agent will perceive a **Breeze.**
- In the square where the gold is, the agent will perceive a **Glitter**.
- When an agent walks into a wall, it will perceive a **Bump.**
- When the wumpus is killed, it emits a woeful **Scream** that can be perceived anywhere in the cave.

The WUMPUS WORLD Example - PEAS Description

- The percepts are given as a **five-symbol list**. If there is a stench and a breeze, but no glitter, no bump, and no scream, the percept is

[Stench, Breeze, None, None, None]

WUMPUS Environment

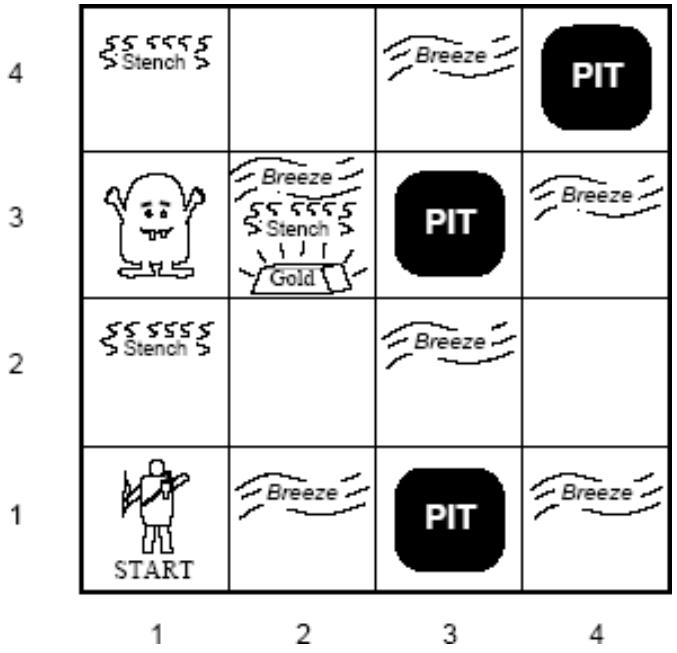
- Characterization of Wumpus World
  - Observable?
  - Deterministic ?
  - Episodic?
  - Static ?
  - Discrete ?
  - Single Agent ?

WUMPUS Environment

- Characterization of Wumpus World
  - Observable?
    - partial, only local perception
  - Deterministic ?
    - Yes, outcomes are specified
  - Episodic?
    - No, sequential at the level of actions
  - Static ?
    - Yes, Wumpus and pits do not move
  - Discrete ?
    - Yes
  - Single Agent ?
    - Yes

Knowledge Based Wumpus Agent

The agent's goal is to find the gold and bring it back to the start square as quickly as possible, without getting killed



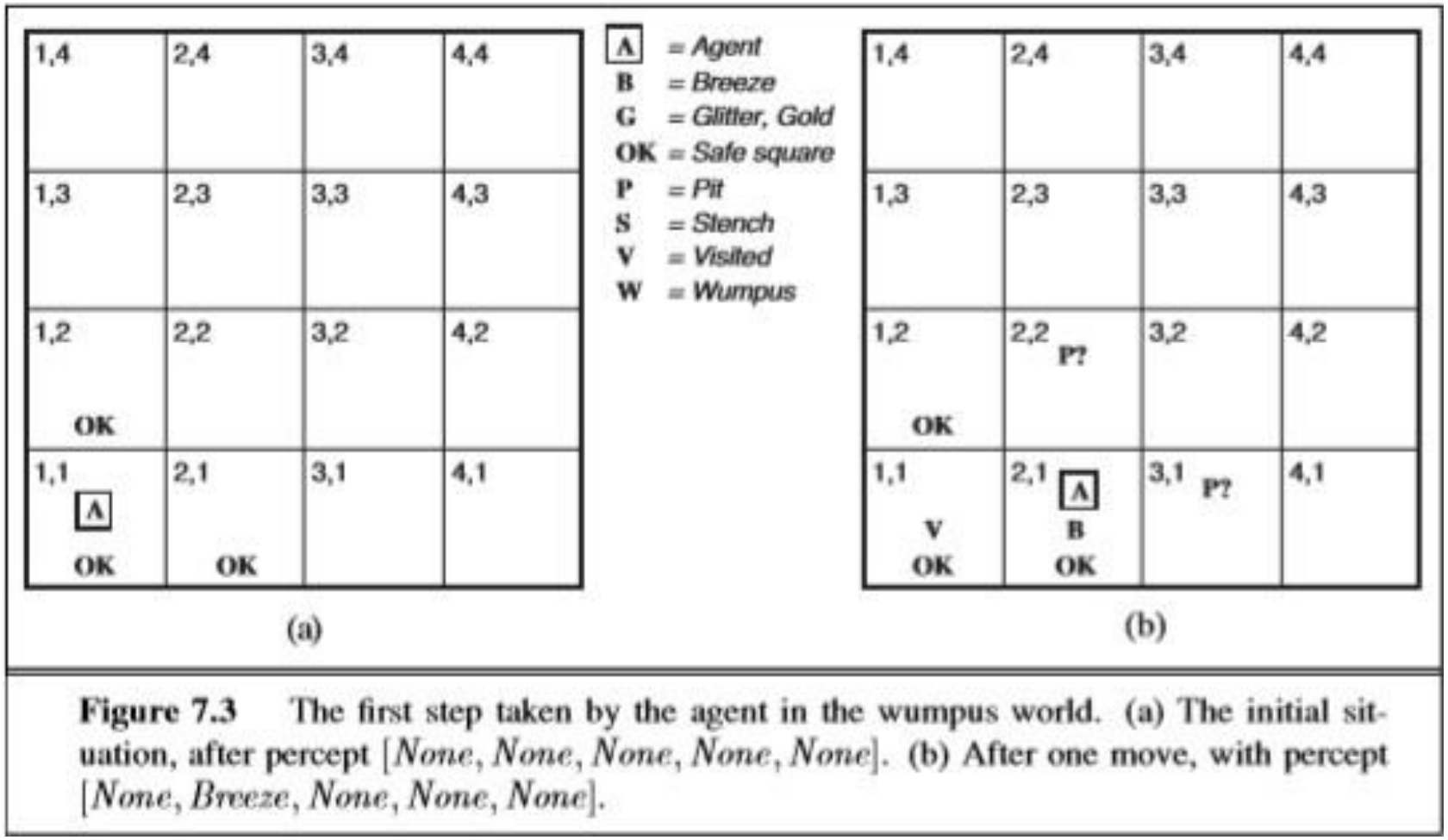
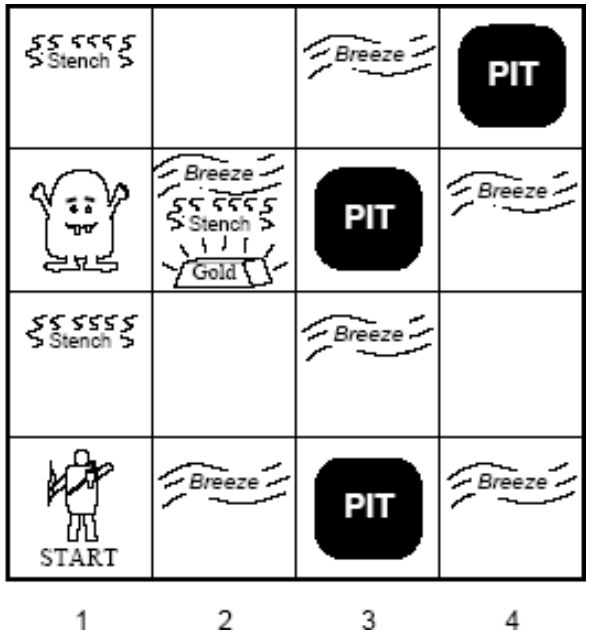


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None*, *None*, *None*, *None*, *None*]. (b) After one move, with percept [*None*, *Breeze*, *None*, *None*, *None*].

Knowledge Based Wumpus Agent



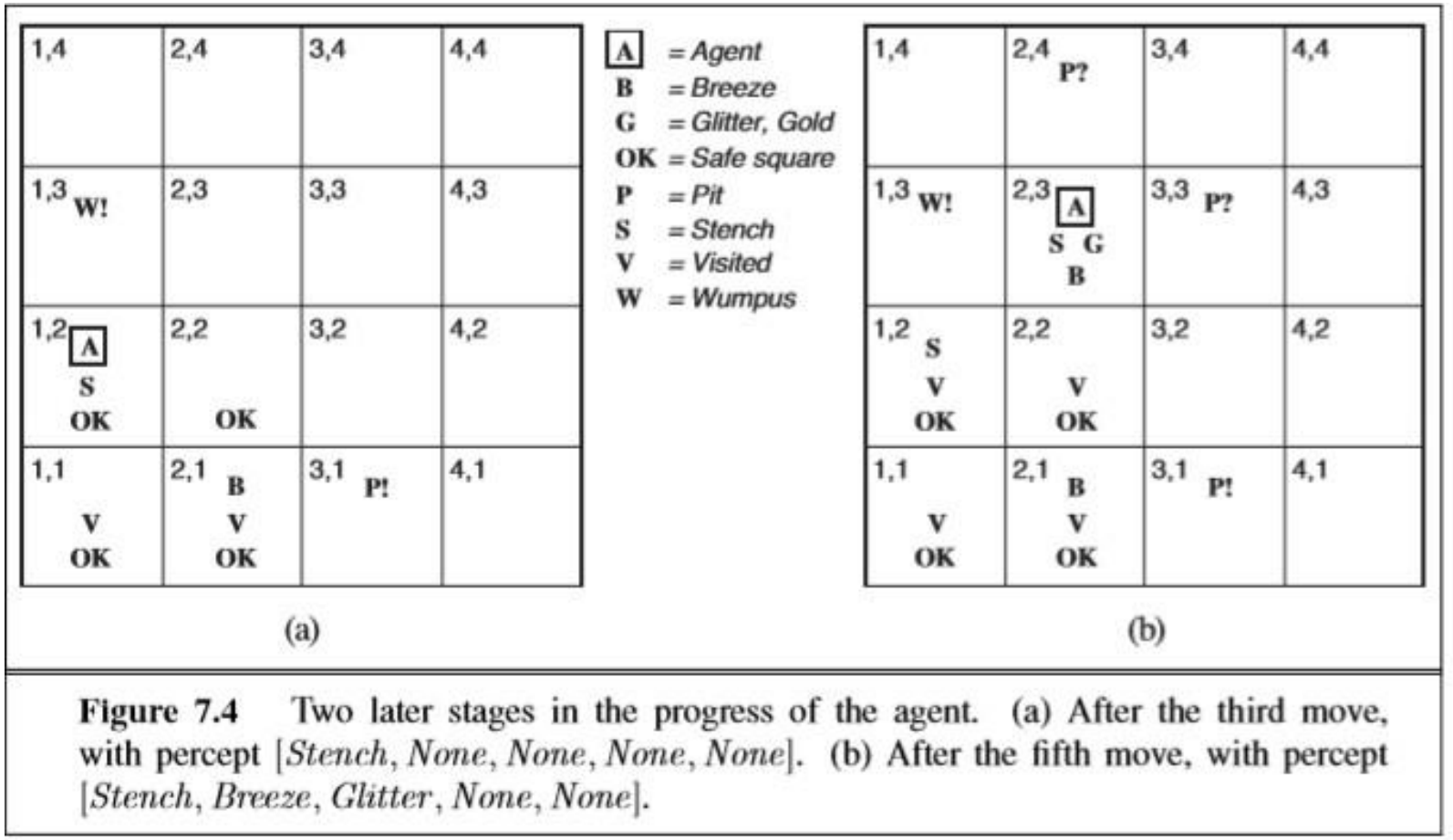


Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept [*Stench*, *None*, *None*, *None*, *None*]. (b) After the fifth move, with percept [*Stench*, *Breeze*, *Glitter*, *None*, *None*].

Note that in each case for which the agent draws a conclusion from the available information, that conclusion is guaranteed to be correct if the available information is correct. This is a fundamental property of logical reasoning.