VTU EXAMINATION 2025-2026 (ODD SEMESTER)

Intelligent System and Machine Learning Algorithm – BEC515A

Module 1

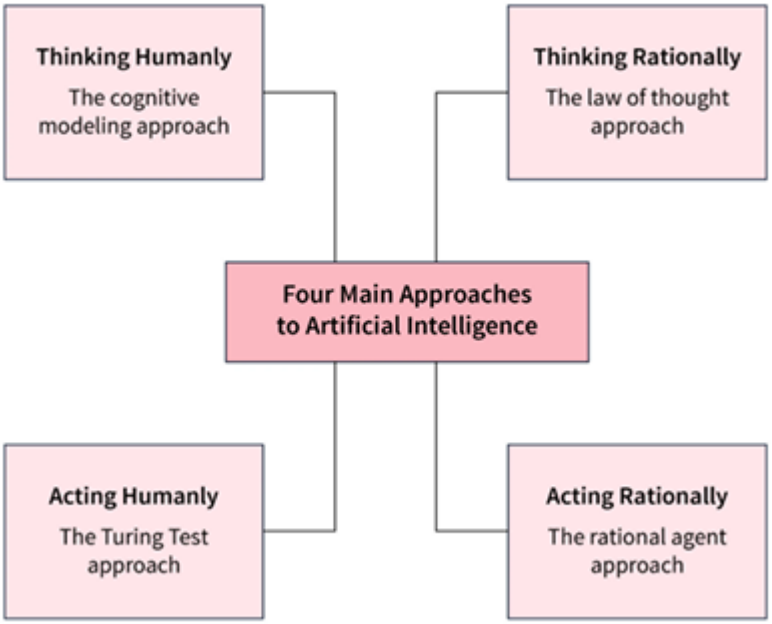
Qn.1.a Explain the following four approaches to AI

i) Acting Humanly
ii) Thinking Humanly
iii) Acting Rationaly
iv) Thinking Rationaly

1.b Explain the foundation of AI in detail.

1.a The four Approaches to AI:

AI is a branch of computer science that focuses on building intelligent machines that function and respond just like people. Machines, particularly computer systems, simulate human intelligence processes in this process. These include learning (acquiring knowledge and rules for using it), reasoning (using rules to arrive at approximate conclusions), and self-correction.



Thinking Humanly (The Cognitive Approach) Thinking humanly, or a cognitive approach is an approach to artificial intelligence (AI) and machine learning that is inspired by the way humans think and learn. The cognitive approach aims to develop AI systems that can mimic human thought processes and behaviours, such as perception, reasoning, and problem-solving. This approach emphasizes the importance of understanding human cognition and how it can be replicated in machines, rather than focusing solely on statistical or mathematical models. One example of the cognitive approach is the development of expert systems, which are computer programs that can solve complex problems in a particular domain, such as medical diagnosis or financial planning.

Acting Humanly (The Turing Test Approach) Acting humanly, also known as the Turing Test approach, is an approach to artificial intelligence (AI) and machine learning that focuses on creating machines that can simulate human-like behaviour and thought processes to the point where they are indistinguishable from humans. The Turing Test approach is based on the idea that a machine can be considered intelligent if it can convincingly pass a test that was proposed by British mathematician and computer scientist Alan Turing. The Turing Test involves a human evaluator engaging in a natural language conversation with a machine and a human, without knowing which is which. If the machine can successfully convince the evaluator that it is the human, then it is considered to have passed the Turing Test. The Turing Test approach has led to the development of a wide range of AI technologies, including chatbots, virtual assistants, and recommendation engines.

Thinking Rationally (The Laws of Thought Approach) ,Thinking rationally, or the laws of thought approach is an approach to artificial intelligence (AI) and machine learning that is based on the principles of formal logic and reasoning. The laws of thought approach aims to develop AI systems that can reason logically and make decisions based on a set of predefined rules. In the laws of thought approach, AI systems are designed to reason deductively, by starting with a set of premises and using logical rules to conclude. This approach is often used in expert systems, where a knowledge base of facts and rules is used to solve complex problems in a particular domain.

Acting Rationally (The Rational Agent Approach) Acting rationally, also known as the rational agent approach, is an approach to artificial intelligence (AI) and machine learning that focuses on creating intelligent agents that can act in the world to achieve their goals. The rational agent approach is based on the idea of rationality, which involves making decisions that maximize the chances of achieving one's goals, given the available information and resources. The rational agent approach emphasizes the importance of designing agents that can reason under uncertainty and adapt to changing environments, rather than simply following a set of predefined rules. One example of the rational agent approach is reinforcement learning, which involves training an agent to make decisions in an environment based on rewards and punishments. The agent learns to maximize its rewards by trying different actions and observing the outcomes.

1.b The foundation of AI:

Artificial Intelligence (AI) is fast becoming an indispensable part of our modern world, revolutionizing numerous industries and transforming the way we live and work. The path to this remarkable technological advancement was paved with significant milestones and breakthroughs that shaped the development of AI systems as we know them today. In 1931, Goedel layed the foundation of Theoretical Computer Science 1920-30s: He published the first universal formal language and showed that math itself is either flawed or allows for unprovable but true statements. In 1936, Turing reformulated Goedel’s result and church’s extension thereof. The gestation of artificial intelligence (1943–1955). The first work that is now generally recognized as AI was done by Warren McCulloch and Walter Pitts (1943). They drew on three sources: knowledge of the basic physiology and function of neurons in

the brain; a formal analysis of propositional logic due to Russell and Whitehead; and Turing's theory of computation. They proposed a model of artificial neurons in which each neuron is characterized as being "on" or "off," with a switch to "on" occurring in response to stimulation by a sufficient number of neighboring neurons. Donald Hebb (1949) demonstrated a simple updating rule for modifying the connection strengths between neurons. His rule, now called Hebbian learning, remains an influential model to this day. Two undergraduate students at Harvard, Marvin Minsky and Dean Edmonds, built the first neural network computer in 1950. 1950 Alan Turing in his article "Computing Machinery and Intelligence." Therein, he introduced the Turing Test, machine learning, genetic algorithms, and reinforcement learning. The birth of artificial intelligence (1956). In 1956, John McCarthy coined the term "Artificial Intelligence" as the topic of the Dartmouth Conference, the first conference devoted to the subject. Early enthusiasm, great expectations (1952–1969). In 1957, The General Problem Solver (GPS) demonstrated by Newell, Shaw & Simon. In 1958, John McCarthy (MIT) invented the Lisp language. In 1959, Arthur Samuel (IBM) wrote the first game-playing program, for checkers, to achieve sufficient skill to challenge a world champion. In 1963, Ivan Sutherland's MIT dissertation on Sketchpad introduced the idea of interactive graphics into computing. In 1966, Ross Quillian (PhD dissertation, Carnegie Inst. of Technology; now CMU) demonstrated semantic nets. In 1967, Dendral program (Edward Feigenbaum, Joshua Lederberg, Bruce Buchanan, Georgia Sutherland at Stanford) demonstrated to interpret mass spectra on organic chemical compounds. First successful knowledge-based program for scientific reasoning. In 1967, Doug Engelbart invented the mouse at SRI. In 1968, Marvin Minsky & Seymour Papert publish Perceptrons, demonstrating limits of simple neural nets. A dose of reality (1966–1973) The first AI winter (1974-1980). The initial AI winter, occurring from 1974 to 1980, is known as a tough period for artificial intelligence (AI). During this time, there was a substantial decrease in research funding, and AI faced a sense of letdown. The duration between years 1974 to 1980 was the first AI winter duration. AI winter refers to the time period where computer scientist dealt with a severe shortage of funding from government for AI researches. During AI winters, an interest of publicity on artificial intelligence was decreased. A boom of AI (1980-1987) Between 1980 and 1987, AI underwent a renaissance and newfound vitality after the challenging era of the First AI Winter. Here are notable occurrences from this timeframe: In 1980, the first national conference of the American Association of Artificial Intelligence was held at Stanford University. Year 1980: After AI's winter duration, AI came back with an "Expert System". Expert systems were programmed to emulate the decision-making ability of a human expert. Additionally, Symbolics Lisp machines were brought into commercial use, marking the onset of an AI resurgence. However, in subsequent years, the Lisp machine market experienced a significant downturn. Year 1981: Danny Hillis created parallel computers tailored for AI and various computational functions, featuring an architecture akin to contemporary GPUs. Year 1984: Marvin Minsky and Roger Schank introduced the phrase "AI winter" during a gathering of the Association for the Advancement of Artificial Intelligence. They cautioned the business world that exaggerated expectations about AI would result in disillusionment and the eventual downfall of the industry, which indeed occurred three years later.

Year 1985: Judea Pearl introduced Bayesian network causal analysis, presenting statistical methods for encoding uncertainty in computer systems. The second AI winter (1987-1993). The duration between the years 1987 to 1993 was the second AI Winter duration. Again Investors and government stopped in funding for AI research as due to high cost but not efficient result. The expert system such as XCON was very cost effective. The emergence of intelligent agents (1993-2011) Between 1993 and 2011, there were significant leaps forward in artificial intelligence (AI), particularly in the development of intelligent computer programs. During this era, AI professionals shifted their emphasis from attempting to match human intelligence to crafting pragmatic, ingenious software tailored to specific tasks. Here are some noteworthy occurrences from this timeframe:

Year 1997: In 1997, IBM's Deep Blue achieved a historic milestone by defeating world chess champion Gary Kasparov, marking the first time a computer triumphed over a reigning world chess champion. Moreover, Sepp Hochreiter and Jürgen Schmidhuber introduced the Long Short-Term Memory recurrent neural network, revolutionizing the capability to process entire sequences of data such as speech or video.

Year 2002: for the first time, AI entered the home in the form of Roomba, a vacuum cleaner. o Year 2006: AI came into the Business world till the year 2006. Companies like Facebook, Twitter, and Netflix also started using AI.

Year 2009: Rajat Raina, Anand Madhavan, and Andrew Ng released the paper titled "Utilizing Graphics Processors for Extensive Deep Unsupervised Learning," introducing the concept of employing GPUs for the training of expansive neural networks.

Year 2011: Jürgen Schmidhuber, Dan Claudiu Cire?an, Ueli Meier, and Jonathan Masci created the initial CNN that attained "superhuman" performance by emerging as the victor in the German Traffic Sign Recognition competition. Furthermore, Apple launched Siri, a voice-activated personal assistant capable of generating responses and executing actions in response to voice commands. Deep learning, big data and artificial general intelligence (2011-present).

Qn. 2a. Briefly, explain the properties of task environment?

2b. Explain the four basic kinds of agent programs?

2a. The properties of task environment:

In Artificial Intelligence (AI), various types of agents operate to achieve specific goals. The PEAS system is a critical framework used to categorize these agents based on their performance, environment, actuators, and sensors. Understanding the PEAS system is essential for grasping how different AI agents function effectively in diverse environments. Among these agents, Rational Agents are considered the most efficient, consistently choosing the optimal path for maximum efficiency. PEAS stands for Performance measure, Environment, Actuator, Sensor.

• Performance Measure: Performance measure is a quantitative measure that evaluates the outcomes of an agent's actions against a predefined goal. The performance measure is crucial because it guides the agent's decision-making process, ensuring that it acts in a way that maximizes its success. For example, in a self-driving car, the performance measure could include criteria such as safety (avoiding accidents), efficiency (minimizing travel time), and comfort (ensuring a smooth ride). The car's AI will aim to optimize these factors through its actions.

• Environment: The environment includes all external factors and conditions that the agent must consider when making decisions. The environment can vary significantly depending on the type of agent and its task. For instance, in the case of a robotic vacuum cleaner, the environment includes the layout of the room, the presence of obstacles, and the type of floor surface. The robot must adapt to these conditions to clean effectively. Understanding the environment is critical for designing AI systems because it affects how the agent perceives its surroundings and interacts with them.

• Actuators: They are responsible for executing the actions decided by the agent based on its perceptions and decisions. In essence, actuators are the "hands and feet" of the agent, enabling it to carry out tasks. In a robotic arm, actuators would be the motors and joints that move the arm to pick up objects. In a software-based AI, actuators might be commands sent to other software components or systems to perform specific tasks. The design and choice of actuators are crucial because they directly affect the agent's ability to perform its functions in the environment.

• Sensors: Sensors collect data from the environment, which is then processed by the agent to make informed decisions. Sensors are the "eyes and ears" of the agent, providing it with the necessary information to act intelligently. In the context of an autonomous drone, sensors might include cameras, GPS, and altimeters, which provide the drone with information about its location, altitude, and surroundings. This data is essential for the drone to navigate and perform tasks like aerial photography or package delivery. The quality and variety of sensors used in an AI system greatly influence its ability to perceive and understand its environment. This framework not only helps in structuring intelligent agents but also ensures that they are well-equipped to achieve their goals in various environments. By carefully considering the performance measure, environment, actuators, and sensors, developers can create AI systems that are more capable, adaptable, and successful in their tasks.

2b. The four basic kinds of agent programs:

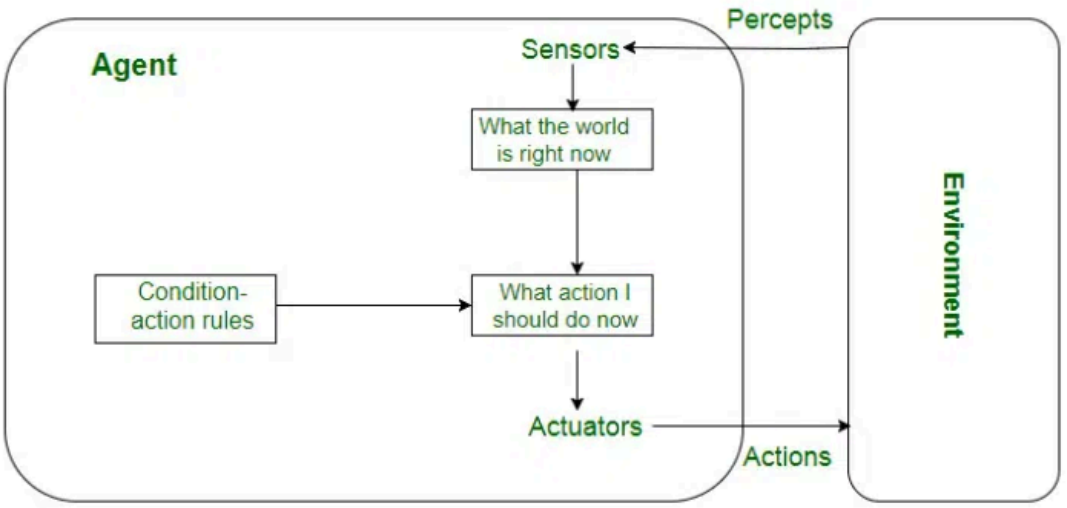
The job of AI is to design an agent program that implements the agent function—the mapping from percepts to actions. Agent program will run on some sort of computing device with physical sensors and actuators—we call this the architecture, agent = architecture + program

- Agents are the entities that perceive their environment and take actions to achieve specific goals. These agents exhibit diverse behaviors and capabilities, ranging from simple reactive responses to sophisticated decision-making.

Four basic kinds of agent programs

Simple reflex agents
Model-based reflex agents
Goal-based agents
Goal-based agents consider future actions and the desirability of their outcomes.
Utility-based agents.
Simple reflex agents

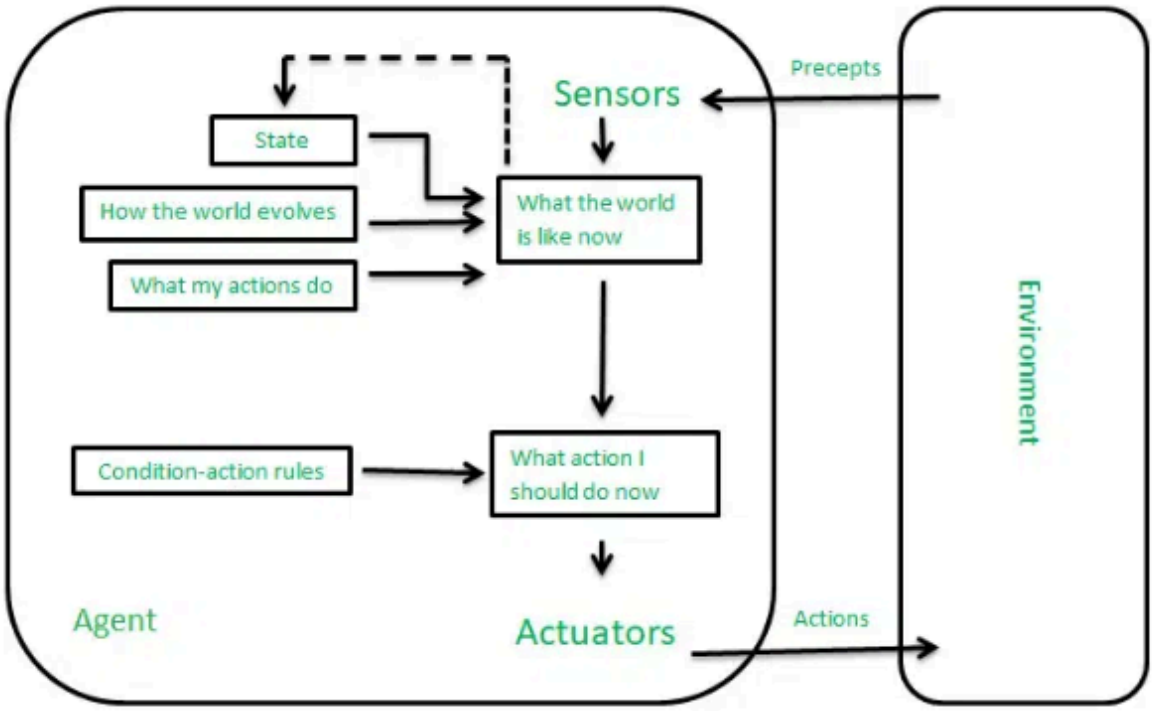
- Simple reflex agents make decisions based solely on the current input, without considering the past or potential future outcomes. They react directly to the current situation without internal state or memory.
- It operate in a continuous loop.
- Example: A thermostat that turns on the heater when the temperature drops below a certain threshold but doesn't consider previous temperature readings or long-term weather forecasts.



Model-based reflex agents

- Model-based reflex agents enhance simple reflex agents by incorporating internal representations of the environment.
- These models allow agents to predict the outcomes of their actions and make more informed decisions.
- By maintaining internal states reflecting unobserved aspects of the environment and utilizing past perceptions, these agents develop a comprehensive understanding of the world.
- This approach equips them to effectively navigate complex environments, adapt to changing conditions, and handle partial observability.

- Example: A self-driving system not only responds to present road conditions but also takes into account its knowledge of traffic rules, road maps, and past experiences to navigate safely.



There are basically 5 steps:

1. Perception

The agent perceives the current state of the environment through sensors, which provide it with information about the current state, such as presence of obstacles, objects and other agents.

2. Modeling the Environment:

The agent maintains the internal model of the environment, which includes information about the state of the world, the possible actions it can take and the expected outcomes of those actions.

Strength:

1. They can handle partially observable environment.
2. It is more flexible.
3. They can cause the internal model to make predictions about how the environment might react to their actions.

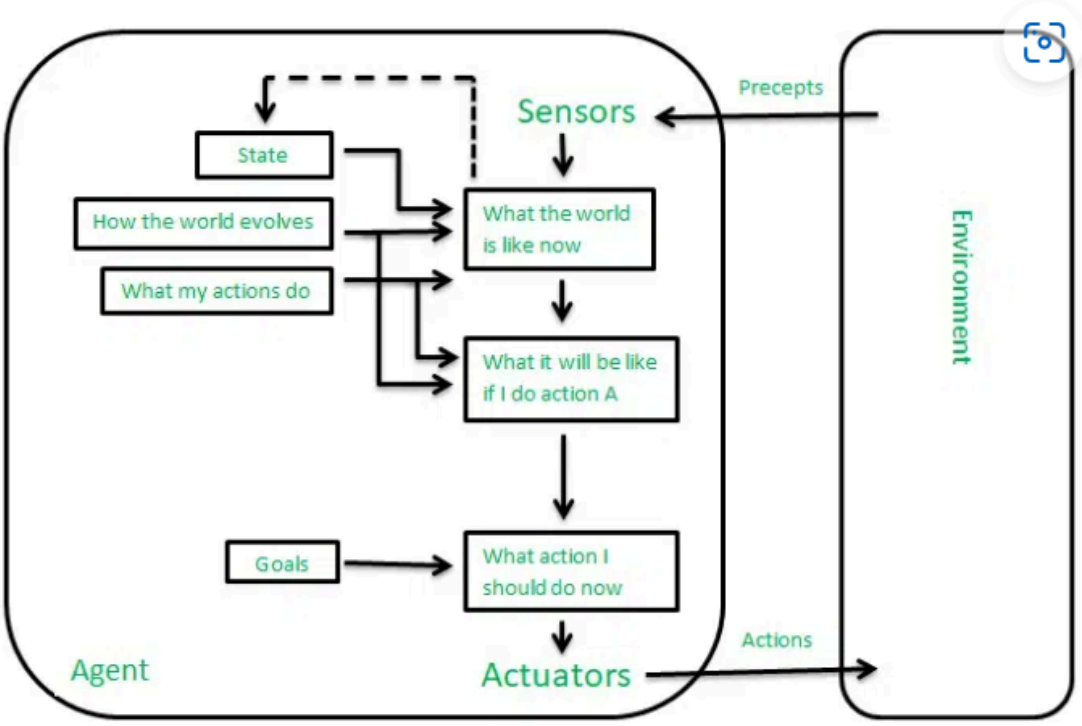
Weakness:

1. Increased level of complexity

2. The agents performance rely on accuracy of its internal model.
3. Limited learning.

Goal-based agents

Goal-based agents have predefined objectives or goals that they aim to achieve. By combining descriptions of goals and models of the environment, these agents plan to achieve different objectives, like reaching particular destinations. They use search and planning methods to create sequences of actions that enhance decision-making in order to achieve goals. Goal-based agents differ from reflex agents by including forward-thinking and future-oriented decision-making processes. Example: A delivery robot tasked with delivering packages to specific locations. It analyzes its current position, destination, available routes, and obstacles to plan an optimal path towards delivering the package.

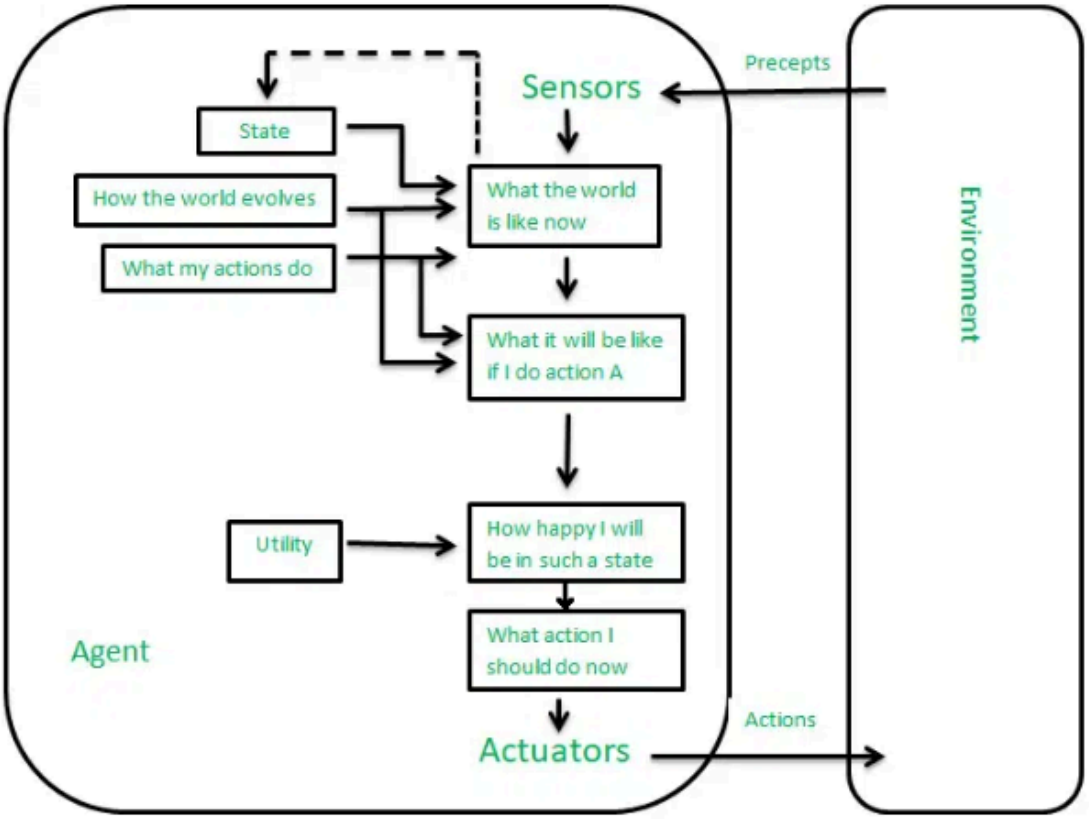


Goal based agent extends the concept of model based agents by incorporating a planning to achieve the goal. It have a specific objective (goal) in mind. They actively plan their actions to achieve it. To reach their goal, these AI agents employ planning algorithms. The planning process involves examining a tree of possibilities, with each branch representing a different actions the agent can take.

Utility-based agents

Utility-based agents go beyond basic goal-oriented methods by taking into account not only the accomplishment of goals, but also the quality of outcomes. They use utility functions to value various states, enabling detailed comparisons and trade-offs among different goals.

These agents optimize overall satisfaction by maximizing expected utility, considering uncertainties and partial observability in complex environments. Even though the concept of utility-based agents may seem simple, implementing them effectively involves complex modeling of the environment, perception, reasoning, and learning, along with clever algorithms to decide on the best course of action in the face of computational challenges. Example: An investment advisor algorithm suggests investment options by considering factors such as potential returns, risk tolerance, and liquidity requirements, with the goal of maximizing the investor's long-term financial satisfaction.



Module 2

Qn. 3a Explain goal formulation and problem formulation with examples

3b Explain the five components of a problem.

3c Discuss the toy problem and real world problems with an example of each.

3a Goal formulation and problem formulation with examples:

Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving. In this case, it makes sense for the agent to adopt the goal of getting to Bengaluru.

Problem formulation is the process of deciding what actions and states to consider, given a goal. Problem formulation is the process of deciding what actions and states to consider,

given a goal. Let us assume that the agent will consider actions at the level of driving from one major town to another. Our agent has now adopted the goal of driving to Bucharest and is considering where to go from Arad. Three roads lead out of Arad, one toward Sibiu, one to Timisoara, and one to Zerind. None of these achieves the goal, so unless the agent is familiar with the geography of Romania. But suppose the agent has a map of Romania. The point of a map is to provide the agent with information about the states. The agent can use this information to consider subsequent stages of a hypothetical journey via each of the three towns, trying to find a journey that eventually gets to Bucharest. Once it has found a path on the map from Arad to Bucharest, it can achieve its goal by carrying out the driving actions that correspond to the legs of the journey. In general, an agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value. For now, we assume that the environment is observable, discrete and deterministic. Under these assumptions, the solution to any problem is a fixed sequence of actions.

- The process of looking for a sequence of actions that reaches the goal is called search.
- A search algorithm takes a problem as input and returns a solution in the form of an

action sequence.

- Once a solution is found, the actions it recommends can be carried out.
- This is called the execution phase.
- After formulating a goal and a problem to solve.

The agent calls a search procedure to solve it. It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do typically, the first action of the sequence and then removing that step from the sequence. Once the solution has been executed, the agent will formulate a new goal.

Assumptions:

The agent has a map of India. We assume that the environment is observable, so the agent always knows the current state. We also assume the environment is discrete, so at any given state there are only finitely many actions to choose from. We will assume the environment is known, so the agent knows which states are reached by each action. We assume that the environment is deterministic, so each action has exactly one outcome. The process of looking for a sequence of actions that reaches the goal is called search. The process of looking for a sequence of actions that reaches the goal is called **search**. A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the execution phase. We have a simple “formulate, search, execute” design for the agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

3b. The five components of a problem:

A problem can be defined formally by five components:

1.The initial state that the agent starts in. For example, the initial state for our agent in Romania might be described as In(Arad).

2.A description of the possible actions available to the agent. Given a particular states, ACTIONS(s) returns the set of actions that can be executed in s. We say that each of these

actions is applicable in s. For example, from the state In(Arad), the applicable actions are {Go(Sibiu), Go(Timisoara), Go(Zerind)}.

3.A description of what each action does; the formal name for this is the transition model, specified by a function RESULT(s, a) that returns the state that results from doing action a in

state s. For example, we have. RESULT(In(Arad),Go(Zerind)) = In(Zerind) .

4.The goal test, which determines whether a given state is a goal state. Sometimes there is an

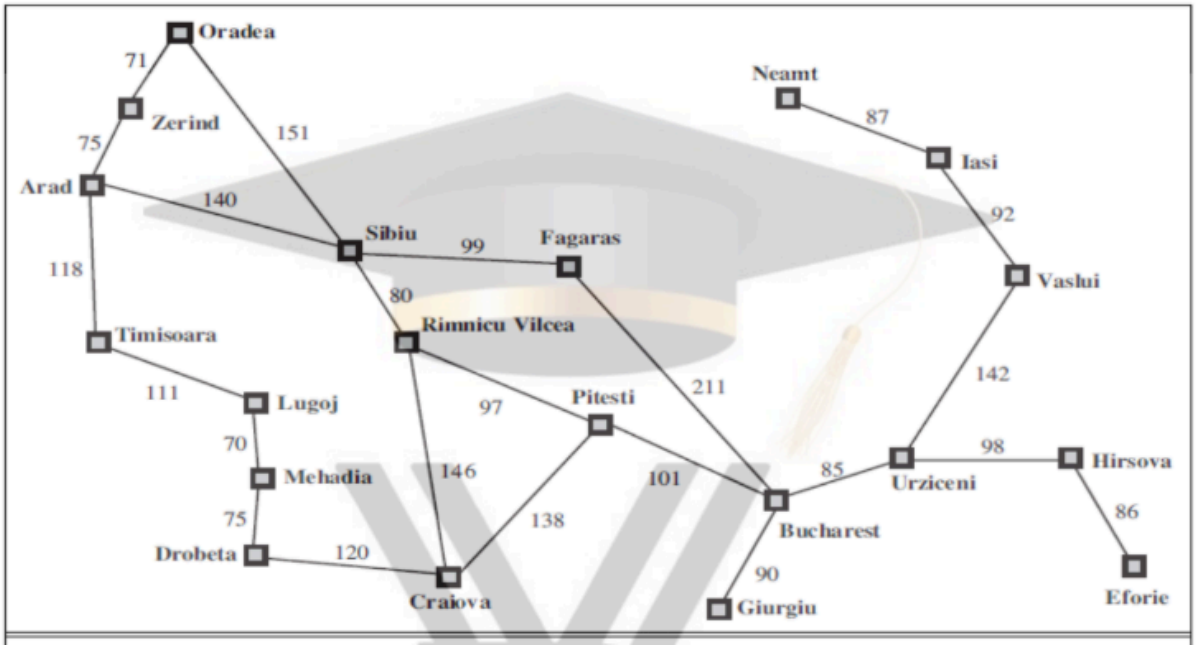
explicit set of possible goal states, and the test simply checks whether the given state is one of

them. The agent's goal in Romania is the singleton set {In(Bucharest)}.

5.A path cost function that assigns a numeric cost to each path. The problem solving agent

chooses a cost function that reflects its own performance measure. For the agent trying to get to Bucharest, it is length in kilometers. The cost of a path can be described as the sum of the costs of the individual actions along the path.

The step cost of taking action a in state s to reach state s` is denoted by c(s, a, s`). The step costs for Romania are shown in Figure as route distances. We assume that step costs are nonnegative.



3c The toy problem and real world problems with an example of each:

TOY PROBLEMS

vacuum world

This can be formulated as a problem as follows:

States: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are 2 × 2

2 = 8 possible world states. A larger environment with n locations has n2n states. Initial state: Any state can be designated as the initial state.

Actions: In this simple environment, each state has just three actions: Left, Right, and Suck.

Larger environments might also include Up and Down.

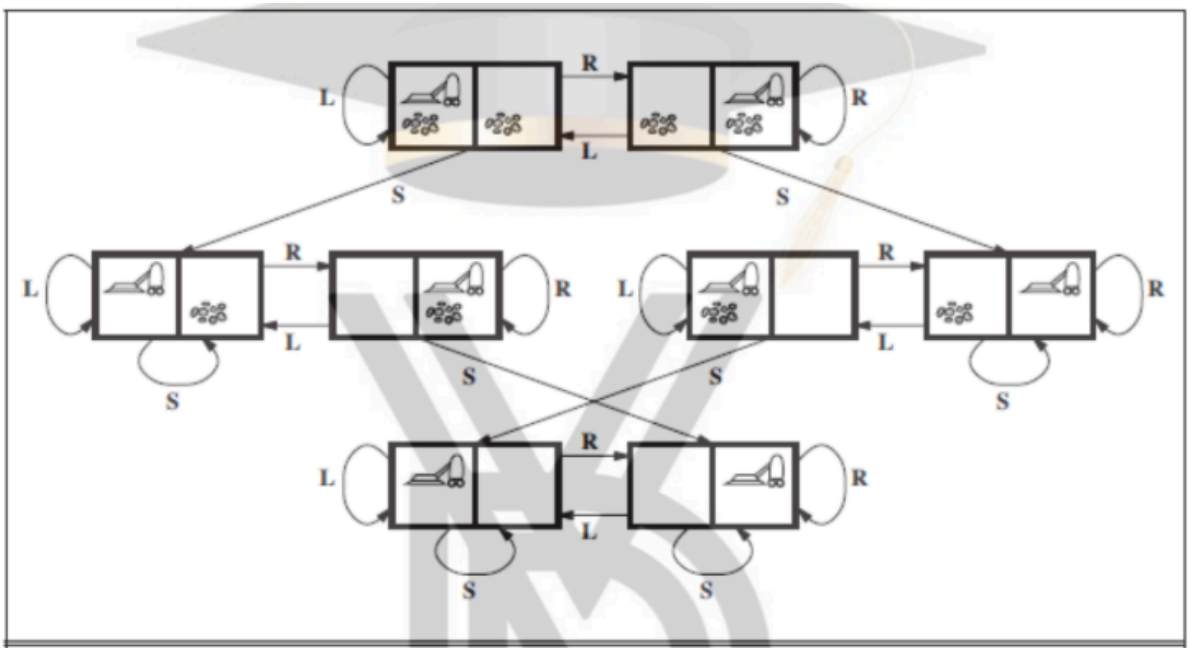
Transition model: The actions have their expected effects, except that moving Left in the

leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no

effect. The complete state space is shown in Figure.

Goal test: This checks whether all the squares are clean.

Path cost: Each step costs 1, so the path cost is the number of steps in the path.



REAL-WORLD PROBLEMS

We have already seen how the route-finding problem is defined in terms of specified locations and transitions along links between them. Route-finding algorithms are used in a variety of applications.

The airline travel problems

The airline travel problems that must be solved by a travel-planning Web site:

• States: Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these "historical" aspects.

• Initial state: This is specified by the user's query.

• Actions: Take any flight from the current location, in any seat class, leaving after the current

time, leaving enough time for within-airport transfer if needed.

• Transition model: The state resulting from taking a flight will have the flight's destination as

the current location and the flight's arrival time as the current time.

• Goal test: Are we at the final destination specified by the user?

• Path cost: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

Touring problems

Touring problems are closely related to route-finding problems, but with an important difference. Consider, for example, the problem "Visit every city in Figure at least once, starting and ending in Bucharest." As with route finding, the actions correspond to trips between adjacent cities. The state space, however, is quite different. Each state must include not just the current location but also the set of cities the agent has visited. So the initial state would be In(Bucharest), Visited({Bucharest}), a typical intermediate state would be In(Vaslui),Visited({Bucharest , Urziceni , Vaslui}), and the goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

Qn.4a Explain Breadth-first search and depth first search strategies with examples.

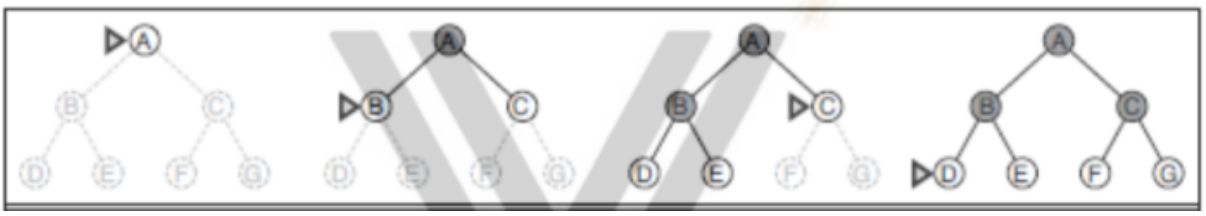
4b Explain Iterative deepening depth-first search and Bidirectional search strategies.

4a. Breadth-first search and depth first search strategies with examples:

Breadth-first search is an instance of the general graph-search algorithm in which the shallowest unexpanded node is chosen for expansion. This is achieved very simply by using a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. There is one slight tweak on the general graph-search algorithm, which is that the goal test is applied to each node when it is generated rather than when it is selected for expansion. Now, the shallowest goal node is not necessarily the optimal one; technically, breadth-first search is optimal if the path cost is a non decreasing function of the depth of the node. Pseudocode is given in Figures shows the progress of the search on a simple binary tree.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.



Depth-first search

Depth-first search always expands the deepest node in the current frontier of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search "backs up" to the next deepest node that still has unexplored successors. The depth-first search algorithm uses a LIFO queue or stack. A LIFO queue means that the most recently generated node is chosen for expansion.

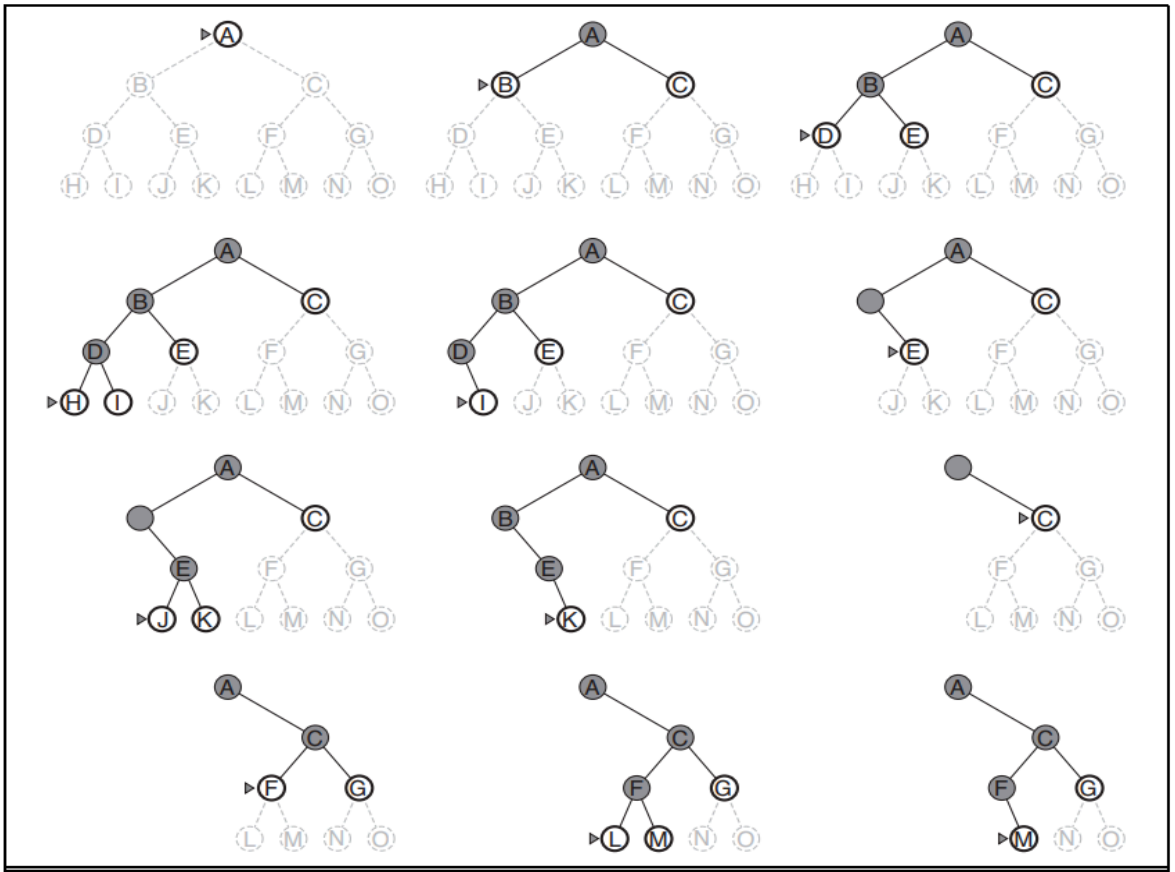


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

Performance Measures:

The graph-search version, which avoids repeated states and redundant paths, is complete in finite state spaces because it will event. The tree-search version, on the other hand, is not

complete. Depth-first tree search can be modified at no extra memory cost so that it checks new states against those on the path from the root to the current node; this avoids infinite loops in finite state spaces but does not avoid the proliferation of redundant paths. In infinite state spaces, both versions fail if an infinite non-goal path is encountered. Both versions are nonoptimal. The time complexity of depth-first graph search is bounded by the size of the state space (which may be infinite, of course). A depth-first tree search, on the other hand, may generate all of the O(bm) nodes in the search tree, where m is the maximum depth of any node; this can be much greater than the size of the state space. Note that m itself can be much larger than d (the depth of the shallowest solution) and is infinite if the tree is unbounded. So far, depth-first search seems to have no clear advantage over breadth-first search, so why do we include it, the reason is the space complexity. For a graph search, there is no advantage, but a depth-first tree search needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 3.17 A recursive implementation of depth-limited tree search.

4b. Iterative deepening depth-first search and Bidirectional search strategies:

Iterative deepening depth-first search

- Iterative deepening search is a general strategy, often used in combination with depth-first tree search that finds the best depth limit.
- It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found.
- This will occur when the depth limit reaches d, the depth of the shallowest goal node.
- Iterative deepening combines the benefits of depth-first and breadth-first search.
- Like depth-first search, its memory requirements are modest: O(bd
- Like breadth-first search, it is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

Figure 3.18 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.

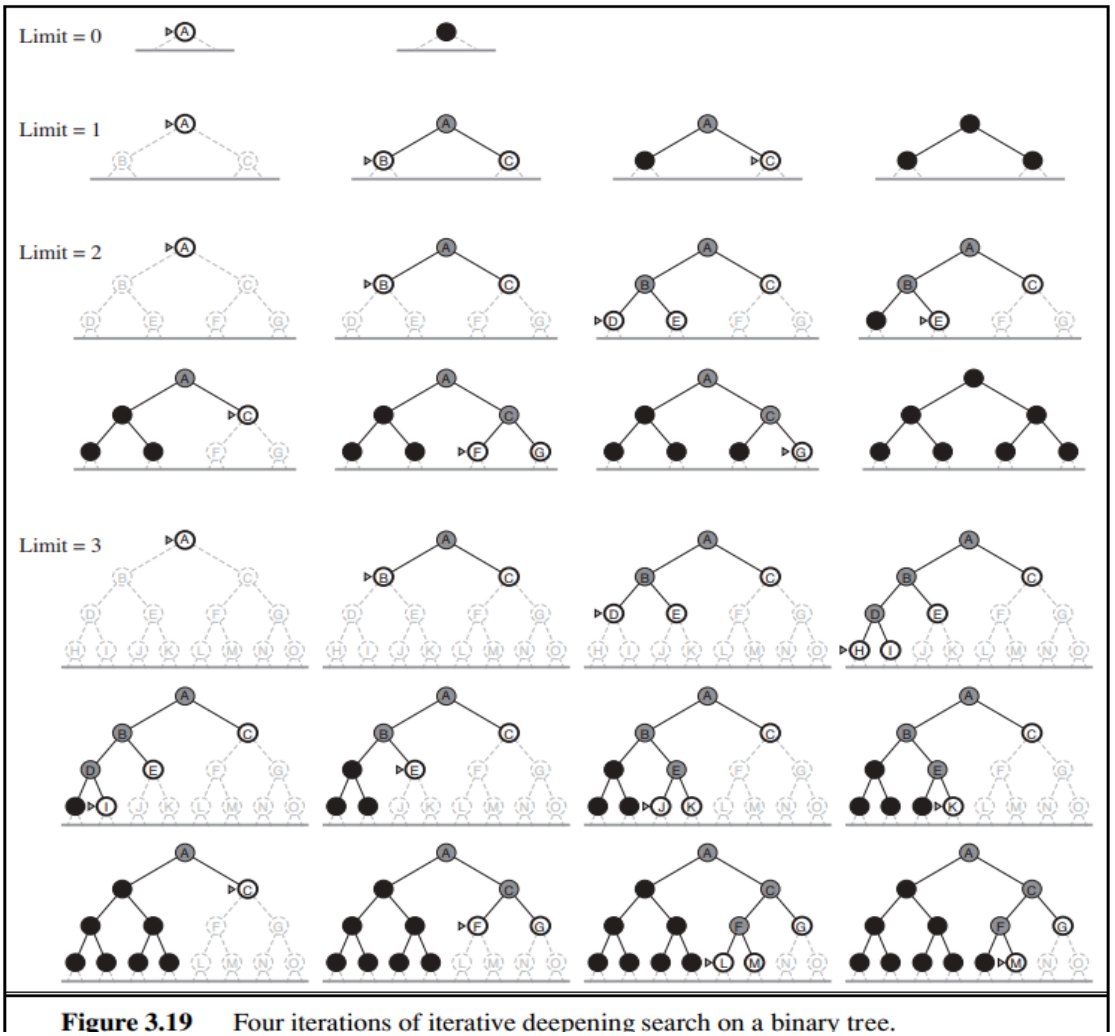


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

Bidirectional search

The idea behind bidirectional search is to run two simultaneous searches—one forward from the initial state and the other backward from the goal, hoping that the two searches meet in the middle (Figure). The motivation is that bd/2 + bd/2 is much less than bd, or in the figure, the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal. Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found. It is important to realize that the first such solution found may not be optimal, even if the two searches are both breadth-first;

some additional search is required to make sure there isn't another short-cut across the gap. The check can be done when each node is generated or selected for expansion and, with a hash table, will take constant time. For example, if a problem has solution depth d=6, and each direction runs breadth-first search one node at a time, then in the worst case the two searches meet when they have generated all of the nodes at depth 3. For b=10, this means a total of 2,220 node generations, compared with 1,111,110 for a standard breadth-first search. Thus, the time complexity of bidirectional search using breadth-first searches in both directions is O(bd/2). The space complexity is also O(bd/2). We can reduce this by roughly half if one of the two searches is done by iterative deepening, but at least one of the frontiers must be kept in memory so that the intersection check can be done. This space requirement is the most significant weakness of bidirectional search.

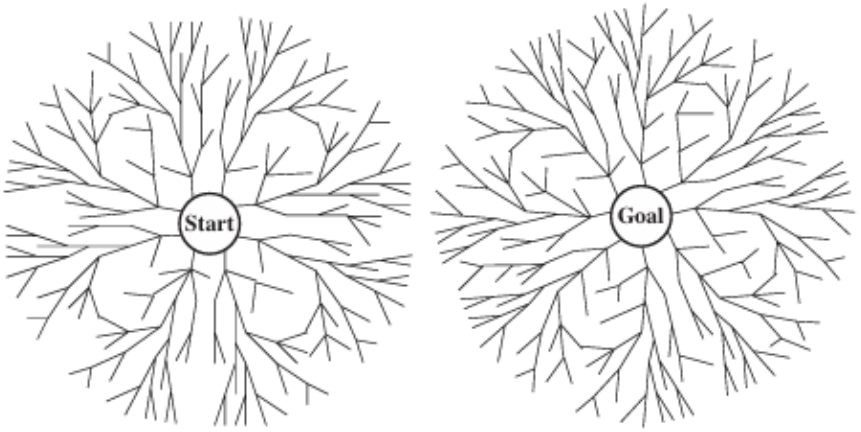


Figure - A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

The reduction in time complexity makes bidirectional search attractive, but how do we search backward? This is not as easy as it sounds. Let the predecessors of a state x be all those states that have x as a successor. Bidirectional search requires a method for computing predecessors. When all the actions in the state space are reversible, the predecessors of x are just its successors. Other cases may require substantial ingenuity. Consider the question of what we mean by "the goal" in searching "backward from the goal." For the 8-puzzle and for finding a route in Romania, there is just one goal state, so the backward search is very much like the forward search. If there are several explicitly listed goal states—for example, the two dirt-free goal states in Figure, then we can construct a new dummy goal state whose immediate predecessors are all the actual goal states. But if the goal is an abstract description, such as the goal that "no queen attacks another queen" in the n-queens problem, then bidirectional search is difficult to use.

MODULE 3

Qn. 5a Explain A* search and memory bounded heuristic search with examples

5b. Discuss heuristic function in detail.

5a. A* search and memory bounded heuristic search with examples:

The core of the A* algorithm is based on cost functions and heuristics. It uses two main parameters: g(n): The actual cost from the starting node to any node n. h(n): The heuristic estimated cost from node n to the goal. This is where A* integrates knowledge beyond the graph to guide the search. The sum, f(n)=g(n)+h(n) f(n)=g(n)+h(n), represents the total estimated cost of the cheapest solution. The

A* algorithm functions by maintaining a priority queue (or open set) of all possible paths along the graph, prioritizing them based on their f(n) values.

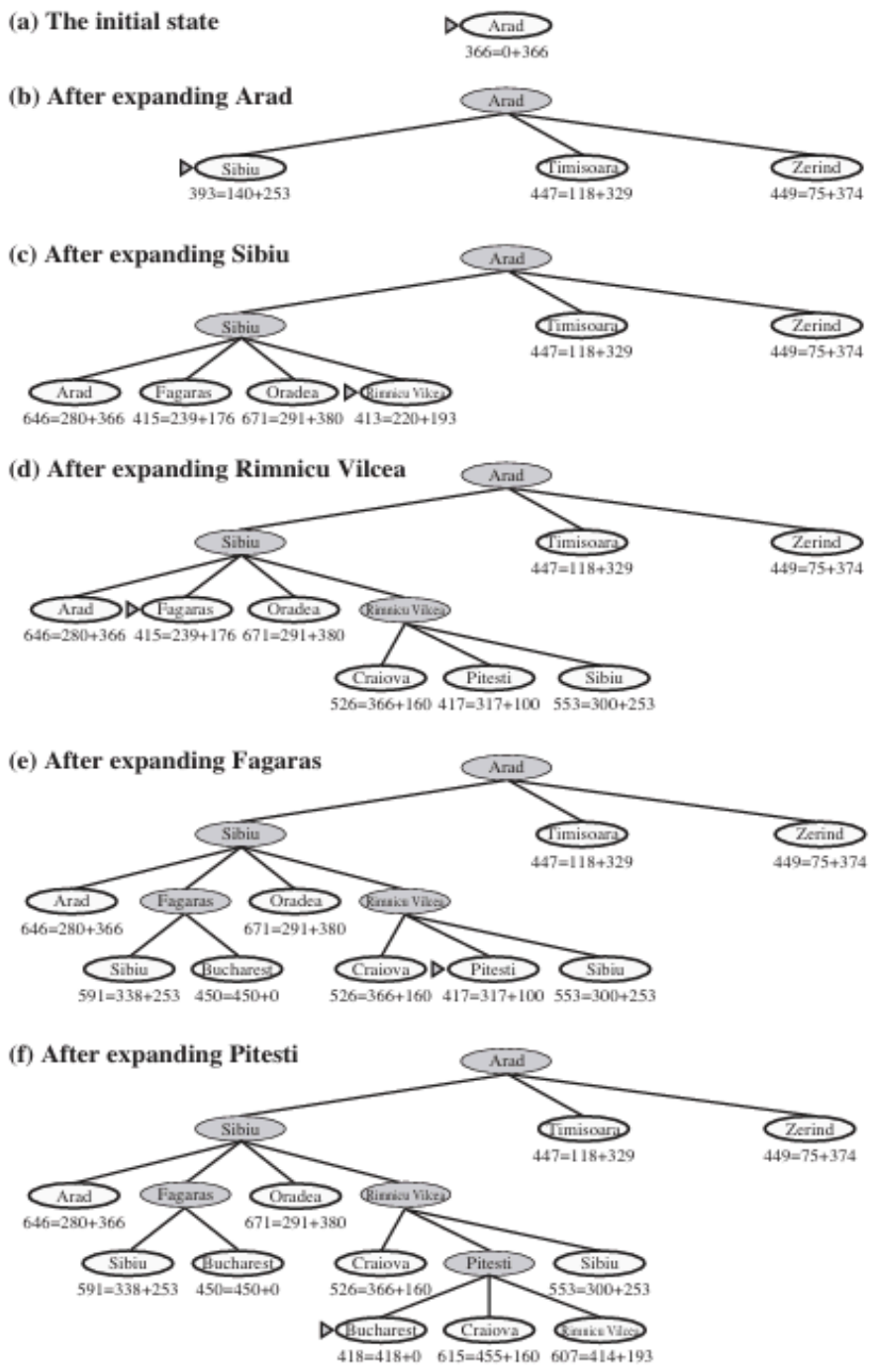
The steps of the algorithm are as follows:

1. Initialization: Start by adding the initial node to the open set with its f(n).

2. Loop: While the open set is not empty, the node with the lowest f(n) value is removed from the queue. 3. Goal Check: If this node is the goal, the algorithm terminates and returns the discovered path.

4. Node Expansion: Otherwise, expand the node (find all its neighbors), calculating g, h, and f values for each neighbor. Add each neighbor to the open set if it's not already present, or if a better path to this neighbor is found.

5. Repeat: The loop repeats until the goal is reached or if there are no more nodes in the open set, indicating no available path.



Memory-bounded heuristic search

The simplest way to reduce memory requirements for A∗ is to adapt the idea of iterative deepening to the heuristic search context, resulting in the iterative-deepening A∗ (IDA∗)al gorithm. The main difference between IDA∗ and standard iterative deepening is that the cutoff used is the f-cost (g+h) rather than the depth; at each iteration, the cutoff value is the small est f-cost of any node that exceeded the cutoff on the previous iteration. IDA∗ is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes. Unfortunately, it suffers from the same difficulties with real valued costs as does the iterative version of uniform-cost search. It seems sensible, therefore, to use all available memory. Two algorithms that do this are MA∗ (memory-bounded A∗) and SMA∗ (simplified MA∗). SMA∗ proceeds just like A∗, expanding the best

leaf until memory is full. At this point, it cannot add a new node to the search tree without dropping an old one. SMA∗ always drops the worst leaf node—the one with the highest f-value. Like RBFS, SMA∗ then backs up the value of the forgotten node to its parent. In this way, the ancestor of a forgotten subtree knows the quality of the best path in that subtree. With this information, SMA∗ regenerates the subtree only when all other paths have been shown to look worse than the path it has forgotten. Another way of saying this is that, if all the descendants of a node n are forgotten, then we will not know which way to go from n, but we will still have an idea of how worthwhile it is to go anywhere from n. The complete algorithm is too complicated to reproduce here,10 but there is one subtlety worth mentioning. We said that SMA∗ expands the best leaf and deletes the worst leaf.

What if all the leaf nodes have the same f-value?

To avoid selecting the same node for deletion and expansion, SMA∗ expands the newest best leaf and deletes the oldest worst leaf. These coincide when there is only one leaf, but in that case, the current search tree must be a single path from root to leaf that fills all of memory. If the leaf is not a goal node, then even if it is on an optimal solution path, that solution is not reachable with the available memory. Therefore, the node can be discarded exactly as if it had no successors. SMA∗ is complete if there is any reachable solution, that is if d, the depth of the shallowest goal node, is less than the memory size (expressed in nodes). It is optimal if any optimal solution is reachable; otherwise, it returns the best reachable solution. In practical terms, SMA∗ is a fairly robust choice for finding optimal solutions, particularly when the state space is a graph, step costs are not uniform, and node generation is expensive compared to the overhead of maintaining the frontier and the explored set.

5b. Heuristic function:

The 8-puzzle was one of the earliest heuristic search problems. The object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration. The average solution cost for a randomly generated 8-puzzle instance is about 22 steps. The branching factor is about 3. When the empty tile is in the middle, four moves are possible; when it is in a corner, two; and when it is along an edge, three. This means that an exhaustive tree search to depth 22 would look at about $322 \approx 3.1 \times 1010$ states. A graph search would cut this down by a factor of about 170,000 because only 9!/2= 181,440 distinct states are reachable. This is a manageable number, but to find a good heuristic function. If we want to find the shortest solutions by using A∗, we need a heuristic function that never overestimates the number of steps to the goal. There is a long history of such heuristics for the 15-puzzle; here are two commonly used candidates: • h1 = the number of misplaced tiles. All of the eight tiles are out of position, so the start state would have h1 =8. h1 is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once. h2 = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the city block distance or Manhattan distance. h2 is also admissible because all any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of h2 =3+1+2+2+2+3+3+2=18. As expected, neither of these overestimates the true solution cost, which is 26. The effect of heuristic accuracy on performance One way to characterize the quality of a heuristic is the effective branching factor b∗. If the total number of nodes generated by A∗ for a particular problem is N and the solution depth is d, then b∗ is the branching factor that a uniform tree of depth d would have to have in order to contain N +1nodes. Thus, $N+1=1+b*+(b*)2+\cdots+(b*)d$. For example, if A∗ finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92. The effective branching factor can vary across problem instances, but usually it is fairly constant for sufficiently hard problems. Therefore, experimental measurements

of b∗ on a small set of problems can provide a good guide to the heuristic's overall usefulness. A well-designed heuristic would have a value of b∗ close to 1, allowing fairly large problems to be solved at reasonable computational cost.

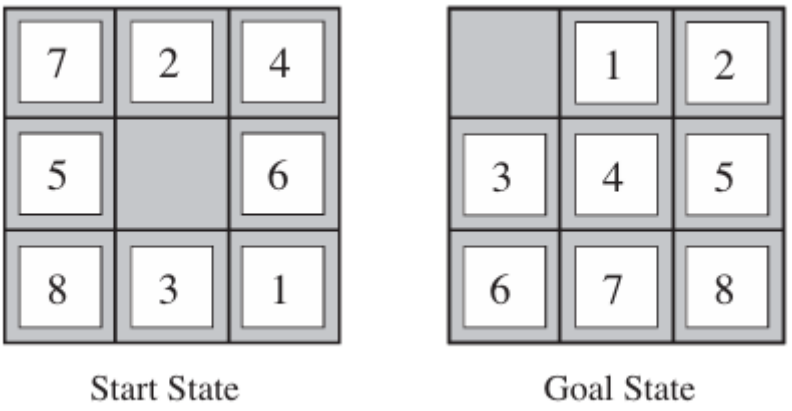


Figure – A typical instance of the 8-puzzle. The solution is 26 steps long.

Q6 a. Explain the Wumpus World Knowledge based agent with example.

Solution: The Wumpus World is a classic Artificial Intelligence problem used to illustrate how a **knowledge-based agent** can use logical reasoning to make decisions in an uncertain environment.

Environment Description

Wumpus World is a **4 × 4 grid** world containing:

- One **Wumpus** (a monster)
- One **Gold**
- One or more **Pits**
- The **Agent** (starts at square [1,1])

The agent's goal is to **find the gold and exit safely** without falling into pits or being eaten by the Wumpus.

Performance Measure

- +1000 for exiting with gold
- –1000 for falling into pit or eaten by Wumpus
- –1 for each move
- –10 for shooting arrow

Percepts (Sensors Information)

At each cell, the agent perceives:

- **Stench** → Wumpus in adjacent cell
- **Breeze** → Pit in adjacent cell
- **Glitter** → Gold in same cell
- **Bump** → Hit a wall
- **Scream** → Wumpus killed

Knowledge-Based Agent in Wumpus World

A **Knowledge-Based Agent (KBA)**:

- Stores facts in a **Knowledge Base (KB)**
- Uses **propositional logic** for reasoning
- Updates KB with percepts
- Applies inference rules to derive safe moves

Structure of KBA:

1. **Tell** → Add percepts to KB
2. **Ask** → Query KB to decide action

Example logical rules:

- If no breeze at (1,1) → No pit in adjacent cells
- If breeze at (x,y) → Pit exists in one of its neighbors
- If no stench at (x,y) → No Wumpus nearby

Example Explanation

Assume agent starts at **[1,1]**.

Step 1:
Percept: No breeze, No stench

Inference:

- No pit in [1,2] and [2,1]
- No Wumpus in [1,2] and [2,1]
- These cells are **safe**

Step 2:
Move to [1,2]
Percept: Breeze

Inference:

- Pit in [1,3] or [2,2]

Step 3:
Move to [2,1]
Percept: No breeze

Inference:

- No pit in [2,2]

From previous step:

- Therefore pit must be in [1,3]

Thus, the agent uses **logical inference** to determine safe squares.

Wumpus World demonstrates how a **knowledge-based agent uses logic, percepts, and inference rules** to operate in an uncertain environment.

It is a fundamental example for understanding **logical reasoning and intelligent agents** in Artificial Intelligence.

Q6b. Explain the syntax and semantics of propositional logic.

Propositional Logic is the simplest form of logic used in Artificial Intelligence to represent knowledge using statements that are either True (T) or False (F). The syntax of propositional logic defines the rules for forming valid sentences, also called well-formed formulas (WFFs). It consists of propositional symbols such as P, Q, R, logical connectives like negation (¬), conjunction (∧), disjunction (∨), implication (→), and biconditional (↔), along with parentheses for proper grouping. According to formation rules, every propositional symbol is a sentence, and if P and Q are sentences, then expressions such as ¬P, (P ∧ Q), (P ∨ Q), (P → Q), and (P ↔ Q) are also valid sentences. Thus, syntax deals only with the structure or form of logical expressions.

The semantics of propositional logic defines the meaning of these sentences by assigning truth values. Each propositional symbol is assigned either True or False, and the truth value of compound statements is determined using truth tables. For example, $P \wedge Q$ is true only when both P and Q are true; $P \vee Q$ is true if at least one is true; and $P \rightarrow Q$ is false only when P is true and Q is false. An interpretation that makes a sentence true is called a model. A sentence true in all interpretations is a tautology, false in all interpretations is a contradiction, and otherwise is a contingency. Hence, syntax specifies the structure of expressions, while semantics provides their meaning and truth evaluation.

Module 4

Q7a. What is Machine Learning ? Explain different perspective and issues in machine learning.

Machine Learning (ML) is a branch of Artificial Intelligence that enables computers to learn from data and improve their performance without being explicitly programmed. A formal definition given by Tom M. Mitchell states: *"A computer program is said to learn from experience E with respect to some task T and performance measure P, if its performance at task T, as measured by P, improves with experience E."* Thus, ML systems identify patterns from historical data and use them to make predictions or decisions.

Machine Learning can be viewed from different perspectives. From a statistical perspective, ML is seen as a method of building probabilistic models to make inferences from data. It focuses on estimating unknown parameters and minimizing prediction error. From a computer science perspective, ML is considered as the design of efficient algorithms that can automatically improve with experience. It emphasizes computational efficiency and scalability. From a data mining perspective, ML is used to discover hidden patterns, associations, or knowledge from large datasets. From a function approximation perspective, learning is viewed as approximating an unknown target function that maps inputs to outputs. From a biological perspective, ML is inspired by the human brain and learning process, such as artificial neural networks that mimic neurons.

Despite its advantages, Machine Learning faces several issues. One major issue is overfitting, where the model performs well on training data but poorly on unseen data. Another issue is underfitting, where the model is too simple to capture the underlying pattern. The quality and quantity of data greatly affect performance, as noisy, incomplete, or biased data leads to inaccurate models. Curse of dimensionality is another problem when dealing with high-dimensional data. Selecting appropriate features and models is also challenging. Additionally, ML systems may face issues related to computational complexity, interpretability, and ethical concerns such as bias and fairness.

In conclusion, Machine Learning is a powerful approach that enables systems to learn from data from various perspectives including statistical, computational, and biological viewpoints. However, careful handling of data, model selection, and evaluation is necessary to overcome practical challenges and build reliable learning systems.

Q8a. Explain the final design of the checkers learning program.

a computer program is said to learn from **Experience (E)** with respect to **Task (T)** and **Performance measure (P)** if its performance improves over time. In the checkers problem:

- **Task (T):** Play checkers
- **Performance (P):** Percentage of games won
- **Experience (E):** Playing games against itself or opponents

1. Choosing the Training Experience

The program gains experience by playing games against itself. Each game generates a sequence of board states and the final outcome (win/loss/draw). These board states serve as training examples for learning.

2. Choosing the Target Function

The target function to be learned is:

$$V(b) = \text{Value of board position } b$$

Where:

- $V(b) = 1$ if the board is a winning position
- $V(b) = 0$ if losing
- $V(b) = 0.5$ if draw

The program uses this evaluation function to choose the best move by selecting the move that leads to the board state with the highest estimated value.

3. Choosing a Representation for the Target Function

Instead of storing all board positions explicitly, the program represents $V(b)$ as a **linear combination of board features**:

$$V(b) = w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

Where:

- x_1 = Number of black pieces
- x_2 = Number of red pieces
- x_3 = Number of black kings
- x_4 = Number of red kings
- etc.
- w_i = Weights to be learned

Thus, learning means adjusting the weights to better estimate board value.

4. Choosing a Learning Algorithm

The program uses the **Least Mean Squares (LMS)** or **Gradient Descent** algorithm to update weights.

Weight update rule:

$$w_i \leftarrow w_i + \eta(V_{train}(b) - V(b))x_i$$

Where:

- η = Learning rate
- $V_{train}(b)$ = Target value
- $V(b)$ = Current estimated value

The difference term represents the **error**, which is used to adjust weights.

5. Final System Operation

1. Initialize weights randomly.
2. Play a game and record board states.
3. Use the game outcome to assign training values.
4. Update weights using LMS rule.
5. Repeat for many games.

Over time, the evaluation function improves, leading to better move selection and higher winning percentage.

Q8b. Explain find s algorithm with example.

The **Find-S algorithm** is a simple concept learning algorithm It is used to learn the **most specific hypothesis** that is consistent with all positive training examples.

1. Concept Learning Problem

In concept learning, the task is to learn a target function that maps instances to either **Positive (+)** or **Negative (–)** examples. The Find-S algorithm searches the hypothesis space from the most specific hypothesis to a more general one.

2. Hypothesis Representation

Each hypothesis is represented as a vector of attribute constraints.

For example, consider attributes:

- Sky ∈ {Sunny, Rainy}
- AirTemp ∈ {Warm, Cold}
- Humidity ∈ {Normal, High}
- Wind ∈ {Strong, Weak}

A hypothesis is written as:

$$h = \langle x_1, x_2, x_3, x_4 \rangle$$

Where:

- Specific value → attribute must match
- "?" → any value allowed
- "Ø" → most specific (no value)

3. Find-S Algorithm Steps

1. Initialize h to the **most specific hypothesis** (all attributes Ø).
2. For each **positive training example**:
    - Compare it with h.
    - For every attribute:

- If h has Ø, replace it with the example's value.
- If h has a different value, replace it with "?".

3. Ignore all negative examples.
4. Output the final hypothesis h.

Consider the training data:

Sky	AirTemp	Humidity	Wind	EnjoySport
Sunny	Warm	Normal	Strong	+
Sunny	Warm	High	Strong	+
Rainy	Cold	High	Strong	–
Sunny	Warm	High	Weak	+

Step 1:

Initialize

$$h_0 = \langle Ø, Ø, Ø, Ø \rangle$$

Step 2: First positive example

$$h_1 = \langle Sunny, Warm, Normal, Strong \rangle$$

Step 3: Second positive example

Humidity differs → replace with "?"

$$h_2 = \langle Sunny, \downarrow rm, ?, Strong \rangle$$

Step 4: Ignore negative example.

Step 5: Third positive example

Wind differs → replace with "?"

$$h_3 = \langle Sunny, Warm, ?, ? \rangle$$

Final Hypothesis:

$$h = \langle Sunny, Warm, ?, ? \rangle$$

This means:

EnjoySport = Yes when Sky = Sunny and AirTemp = Warm, regardless of Humidity and Wind.

Q8c. Explain the list then eliminate algorithm.

The **List-Then-Eliminate algorithm** is a concept learning method It is a simple algorithm used to determine the set of all hypotheses that are consistent with the given training examples. Unlike Find-S, which finds only the most specific consistent hypothesis, List-Then-Eliminate maintains all hypotheses that remain consistent with the observed data.

In concept learning, the goal is to identify a target concept from a hypothesis space H, given a set of labeled training examples. The List-Then-Eliminate algorithm begins by initializing the hypothesis space H to contain **all possible hypotheses**. Each hypothesis represents a possible description of the target concept. The algorithm then processes each training example one by one. For each example, it eliminates any hypothesis from H that is inconsistent with that example. If the example is positive, all hypotheses that classify it as negative are removed. If the example is negative, all hypotheses that classify it as positive are removed. This process continues until all training examples have been considered.

For example, consider a simple EnjoySport problem with attributes Sky {Sunny, Rainy} and AirTemp {Warm, Cold}. Initially, the hypothesis space contains all possible combinations such as ⟨Sunny, Warm⟩, ⟨Sunny, Cold⟩, ⟨Rainy, Warm⟩, ⟨Rainy, Cold⟩, ⟨?, Warm⟩, ⟨Sunny, ?⟩, ⟨?, ?⟩, etc. Suppose the first training example is (Sunny, Warm) labeled positive. All hypotheses that do not classify this example as positive are eliminated. If the next example is (Rainy, Cold) labeled negative, then all hypotheses that classify it as positive are removed. After processing all examples, the remaining hypotheses form the **version space**, which is the set of all hypotheses consistent with the training data.

The main advantage of List-Then-Eliminate is that it guarantees finding all consistent hypotheses, thereby clearly identifying the version space. However, its major limitation is that it becomes computationally expensive when the hypothesis space is very large, since it requires explicitly listing and checking all hypotheses. In conclusion, the List-Then-Eliminate algorithm systematically narrows down the hypothesis space by removing inconsistent hypotheses and forms the basis for more advanced algorithms like Candidate-Elimination in concept learning.

Q9a. Explain the steps involved to prepare the data for machine learning algorithms in end-end machine learning project.

Ans:

Data preparation is one of the most important steps in the machine learning workflow. Real-world data is often incomplete, noisy, and inconsistent. Before training a model, it must be cleaned, transformed, and formatted to improve accuracy and generalization. According to the machine learning pipeline, data preprocessing ensures that the learning algorithm receives meaningful and standardized data.

Key Steps in Data Preprocessing

1. Data Collection and Inspection

The first step is gathering raw data from various sources (CSV files, databases, APIs, sensors, etc.). After collection, tools like **Pandas** are used to inspect data using:

- info()
- describe()
- Visualizations (histograms, scatter plots)

2. Handling Missing Values

Missing values can negatively impact training and predictions. Common techniques include:

- **Removing missing data**
  - Remove rows or columns using:

```
data.dropna()
```

- **Imputation**

Replace missing values using:

  - Mean or median (numeric data)
  - Mode (categorical data)

Scikit-Learn provides:

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
```

3. Handling categorical attributes - Handling categorical attributes is an essential part of data preprocessing because most machine learning algorithms require numerical inputs and cannot work directly with text labels. The first step is to identify the types of categorical data, such as nominal (no order, like colors or city names) and ordinal (ordered categories, like size: small, medium, large). For ordinal data, label encoding can be used to assign numerical values based on the natural order. For nominal data, converting categories to numbers without implying hierarchy is important, and techniques like one-hot encoding are commonly used. One-hot encoding creates binary columns for each category, ensuring the model does not assume any order between values. In Scikit-Learn, tools like OrdinalEncoder and OneHotEncoder help automate this process. Properly encoding categorical data ensures that the machine learning algorithm interprets the information correctly and improves model performance.

4. Feature Scaling: A step that ensures all numerical features contribute equally to a machine learning model. In real-world datasets, different features often exist on different scales—for example, age may range from 1 to 100, while income may range in thousands or lakhs. Algorithms like k-Nearest Neighbors, Support Vector Machines, Neural Networks, and Linear Regression are sensitive to such variations and may give more importance to features with larger numeric values. To avoid this bias, feature scaling transforms the data into a common range. Two widely used methods are normalization and standardization. Normalization (Min-Max scaling) converts values into a fixed range, typically 0 to 1. Standardization adjusts data so that it has a mean of 0

and a standard deviation of 1, making the distribution centered and uniform. Scikit-Learn provides tools like StandardScaler and MinMaxScaler for this purpose. Proper feature scaling improves model accuracy, speeds up convergence during training, and ensures better generalization of machine learning algorithms.

5. Feature engineering: is the process of creating new meaningful features or modifying existing ones to improve model performance. It involves techniques such as combining features, extracting useful patterns, or transforming raw data into more informative representations. Examples include creating ratios, polynomial features, or extracting date components like day, month, or year. Good feature engineering helps the model learn relationships more effectively and reduces the need for complex algorithms. Overall, it plays a significant role in improving accuracy and generalization in machine learning tasks.

6. Train-test split: this is a technique used to divide a dataset into two parts: one for training the machine learning model and the other for evaluating its performance. The training set is used to teach model patterns in the data, while the test set checks how well the model generalizes to unseen data. A common split ratio is 80% for training and 20% for testing, although this may vary depending on dataset size. This helps prevent overfitting, where a model performs well on training data but poorly on new data. In Scikit-Learn, the train_test_split() function is commonly used to perform this step.

Q9b Explain how to select and train a machine learning model

Selecting and training a Machine Learning model is a systematic process that involves choosing an appropriate model for a given task and optimizing it using data. A machine learning program improves its performance on a task through experience. Therefore, proper model selection and training are essential to ensure high performance and generalization.

The first step in model selection is **defining the problem and task type**, such as classification, regression, or clustering. Based on the nature of the output variable and data characteristics, a suitable model is chosen. For example, Linear Regression is used for continuous output prediction, Decision Trees and k-Nearest Neighbors are used for classification problems, and Neural Networks are used for complex pattern recognition tasks. The choice of model depends on factors such as size of dataset, interpretability requirements, computational cost, and complexity of the problem.

The next step is **data preparation**, which includes data cleaning, handling missing values, encoding categorical variables, normalization or standardization of features, and splitting the dataset into training and testing sets. Usually, the dataset is divided into training data (for learning) and test data (for evaluation). Sometimes, a validation set is also used for tuning parameters.

Training the model involves feeding the training data into the algorithm so that it can learn patterns. During training, the model adjusts its parameters to minimize an error or loss function. For example, in regression problems, Mean Squared Error (MSE) is minimized, while in classification problems, cross-entropy loss may be used. Optimization techniques such as Gradient Descent are commonly applied to update parameters iteratively.

After training, the model is **evaluated** using test data to measure its performance. Common performance metrics include accuracy, precision, recall, F1-score for classification, and Mean Squared Error for regression. If the model shows poor performance due to overfitting or underfitting, techniques such as regularization, cross-validation, feature selection, or hyperparameter tuning are applied.

Q10a. Explain the following performance measures.

(i) Measuring accuracy using cross validation.

(ii) Confusion matrix

Performance evaluation is an important step in Machine Learning to measure how well a model generalizes to unseen data. Proper performance measures help in comparing different models and selecting the best one.

(i) Measuring Accuracy using Cross-Validation:
Cross-validation is a statistical technique used to estimate the accuracy of a model on unseen data. Instead of dividing the dataset into only one training and testing set, cross-validation divides the dataset into *k* equal-sized subsets or folds. This method is commonly called **k-fold cross-validation**. In this approach, the model is trained on (k–1) folds and tested on the remaining one fold. This process is repeated k times, each time using a different fold as the test set. The final accuracy is calculated as the average of the accuracies obtained in all k trials. For example, in 5-fold cross-validation, the dataset is divided into 5 parts, and the model is trained and tested 5 times. Cross-validation reduces bias due to random sampling and provides a more reliable estimate of model performance. It also helps in detecting overfitting and selecting appropriate model parameters.

(ii) Confusion Matrix:
A confusion matrix is a tabular representation used to evaluate the performance of a classification model. It compares the actual class labels with the predicted class labels. For a binary classification problem, the confusion matrix consists of four components: True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN). True Positive represents correctly predicted positive instances, True Negative represents correctly predicted negative instances, False Positive represents negative instances wrongly predicted as positive, and False Negative represents positive instances wrongly predicted as negative. From the confusion matrix, several performance metrics can be calculated such as Accuracy, Precision, Recall, and F1-score. Accuracy is defined as (TP + TN) / Total instances. Precision measures correctness of positive predictions, and Recall measures the ability to identify all positive instances. Thus, the confusion matrix provides detailed insight into classification performance beyond simple accuracy.

Q10b. Explain the following classifiers.

(i)Multiclass classification

(ii) Multi label classification

Classification is a supervised learning task in which a model assigns input data to predefined categories or classes. Depending on the number and nature of class labels, classification problems are categorized into multiclass and multilabel classification.

(i) Multiclass Classification:

Multiclass classification is a type of classification problem in which each instance belongs to exactly one class out of more than two possible classes. Unlike binary classification, which involves only two classes, multiclass classification deals with three or more mutually exclusive classes. For example, classifying an email as *spam, promotional, or personal*, or recognizing handwritten digits from 0 to 9 are multiclass problems. In this type, each input is assigned only one label. Algorithms such as Decision Trees, k-Nearest Neighbors, Naïve Bayes, and Neural Networks naturally support multiclass classification. For algorithms that are designed for binary classification, strategies such as One-vs-Rest (OvR) and One-vs-One (OvO) are used to extend them to multiclass problems. Performance is typically evaluated using accuracy, precision, recall, F1-score, and confusion matrix. Thus, multiclass classification handles problems where categories are mutually exclusive.

(ii) Multi-label Classification:

Multi-label classification is a type of classification problem in which each instance can belong to more than one class simultaneously. Unlike multiclass classification, the class labels are not mutually exclusive. For example, in movie genre classification, a movie may belong to both *action* and *comedy* genres. Similarly, in text categorization, a document may be labeled as both *science* and *technology*. In multi-label classification, the output is a set of labels rather than a single label. Techniques used include problem transformation methods such as Binary Relevance (converting into multiple binary classification problems), Classifier Chains, and algorithm adaptation methods like multi-label kNN or neural networks with sigmoid activation. Evaluation measures include Hamming Loss, Precision, Recall, and F1-score adapted for multiple labels.