Artificial Intelligence & Machine Learning

[MCPC2003]

LECTURE NOTES

MCA–III SEM (2025-2026)

Lecture Notes
by
Dr. Mamata Nayak
HOD (IT)

Dr. Ambedkar Memorial Institute of IT & Management Science
Jagda, Rourkela-42
(NAAC Accredited)

MCPC2003 ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING (3-0-0)

Course Objectives:

1. To provide a strong foundation on fundamental concepts in Artificial Intelligence
2. To Provide a basic exposition to the goals and methods of Computational Intelligence
3. To study of the design of intelligent computational techniques
4. To enable problem-solving through various machine learning techniques.

Module I

Introduction to AI - Intelligent Agents, Problem-Solving Agents, **Searching for Solutions** - Breadth-first search, Depth-first search, Hill-climbing search, simulated annealing search, Local Search in Continuous Spaces. Heuristic functions, Hill Climbing, Best First Search, A*, Adversarial Search: Game Playing, Min-Max Search, Alpha - Beta Pruning.

Module II

Knowledge and Reasoning: A Knowledge Based Agent, WUMPUS WORLD Environment, Propositional Logic, First Order Predicate Logic, Forward and Backward Chaining. Expert Systems: Introduction, Design of Expert systems.

Module III

Introduction MLP. Type of Human Learning, Type of Machine Learning: Supervised, unsupervised, reinforcement, General Model of Learning Agents

Module IV

Supervised: holdout method, K-fold cross-validation method, boot strapping, simple-regression method, unsupervised: clustering, association, reinforcement learning model.

Course Outcomes: Upon successful completion of this course, students should be able to:

CO1: Apply the Computational Intelligence techniques in applications which involve searching, reasoning and learning.
CO2: Understand artificial intelligence techniques for information retrieval
CO3: Apply Computational Intelligence techniques primarily for machine learning.
CO4: Apply the machine learning techniques for problem solving and validating

Text Books:

1. Elaine Rich, Kevin Knight, Shivshankar B Nair, Artificial Intelligence, McGraw Hill, 3rd Edition.
2. Tom Mitchell, Machine Learning, McGraw Hill , 1997,

Reference Books

1. Richard O. Duda, Peter E. Hart, David G. Stork, Pattern classification, Wiley , (2nd edition).Wiley, New York, 2001
2. Dan W. Patterson, “Introduction to Artificial Intelligence and Expert Systems”, 1st Edition,1996, PHI Learning Pvt. Ltd., New Delhi.
3. Nills J. Nilsson, “Artificial Intelligence: A New Synthesis”, 2nd Edition, 2000, Elsevier IndiaPublications, New Delhi.
4. Christopher M. Bishop, Pattern Recognition and Machine Learning, Springer , 2011 edition
5. Ian Goodfellow, YoshuaBengio, Aaron Courville, Deep Learning, MIT Press , 2016

Module-I

What is Artificial Intelligence?

It is a branch of Computer Science that pursues creating the computers or machines as intelligent as human beings. It is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable.

Definition: Artificial Intelligence is the study of how to make computers do things, which, at the moment, people do better. According to the father of Artificial Intelligence, John McCarthy, it is "The science and engineering of making intelligent machines, especially intelligent computer programs".
Intelligence relates to tasks involving higher mental processes, e.g. creativity, solving problems, pattern recognition, classification, learning, induction, deduction, building analogies, optimization, language processing, knowledge and many more. Intelligence is the computational part of the ability to achieve goals. Intelligent behavior is depicted by perceiving one's environment, acting in complex environments, learning and understanding from experience, reasoning to solve problems and discover hidden knowledge, applying knowledge successfully in new situations, thinking abstractly, using analogies, communicating with others and more.

Four Approaches to AI:

1. **Thinking Humanly:** The cognitive modeling approach (e.g., *How does the human mind work?*)
2. **Thinking Rationally:** The "laws of thought" approach (e.g., *Using logic to deduce answers.*)
3. **Acting Humanly:** The Turing Test approach (e.g., *Can the machine fool a human?*)
4. **Acting Rationally:** The **Rational Agent** approach (e.g., *Doing the right thing to maximize performance.*)

History of AI:

Important research that laid the groundwork for AI:
In 1931, Goedel layed the foundation of Theoretical Computer Science1920-30s:
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.
In 1956, John McCarthy coined the term "Artificial Intelligence" as the topic of the Dartmouth Conference, the first conference devoted to the subject.
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.

In 1972, Prolog developed by Alain Colmerauer.

In Mid 80's, Neural Networks become widely used with the Backpropagation algorithm (first described by Werbos in 1974).

In 1990, Major advances in all areas of AI, with significant demonstrations in machine learning, intelligent tutoring, case-based reasoning, multi-agent planning, scheduling, uncertain reasoning, data mining, natural language understanding and translation, vision, virtual reality, games, and other topics.

In 1997, Deep Blue beats the World Chess Champion Kasparov.

In 2002,iRobot, founded by researchers at the MIT Artificial Intelligence Lab, introduced Roomba, a vacuum cleaning robot. By 2006, two million had been sold. described by Werbos in 1974).

Sub Areas of AI:

1) Game Playing Deep Blue Chess program beat world champion Gary Kasparov

2) Speech Recognition PEGASUS spoken language interface to American Airlines' EAASY SABRE reseration system, which allows users to obtain flight information and make reservations over the telephone. The 1990s has seen significant advances in speech recognition so that limited systems are now successful.

3) Computer Vision Face recognition programs in use by banks, government, etc. The ALVINN system from CMU autonomously drove a van from Washington, D.C. to San Diego (all but 52 of 2,849 miles), averaging 63 mph day and night, and in all weather conditions. Handwriting recognition, electronics and manufacturing inspection, photo interpretation, baggage inspection, reverse engineering to automatically construct a 3D geometric model.

4) Expert Systems Application-specific systems that rely on obtaining the knowledge of human experts in an area and programming that knowledge into a system.

   a. Diagnostic Systems : MYCIN system for diagnosing bacterial infections of the blood and suggesting treatments. Intellipath pathology diagnosis system (AMA approved). Pathfinder medical diagnosis system, which suggests tests and makes diagnoses. Whirlpool customer assistance center.
   b. System Configuration DEC's XCON system for custom hardware configuration. Radiotherapy treatment planning.
   c. Financial Decision Making Credit card companies, mortgage companies, banks, and the U.S. government employ AI systems to detect fraud and expedite financial transactions. For example, AMEX credit check.
   d. Classification Systems Put information into one of a fixed set of categories using several sources of information. E.g., financial decision making systems. NASA developed a system for classifying very faint areas in astronomical images into either stars or galaxies with very high accuracy by learning from human experts' classifications.

5) Mathematical Theorem Proving Use inference methods to prove new theorems.

6) Natural Language Understanding AltaVista's translation of web pages. Translation of Catepillar Truck manuals into 20 languages.

7) Scheduling and Planning Automatic scheduling for manufacturing. DARPA's DART system used in Desert Storm and Desert Shield operations to plan logistics of people and supplies. American Airlines rerouting contingency planner. European space agency planning and scheduling of spacecraft assembly, integration and verification.
8) Artificial Neural Networks
9) Machine Learning

Application of AI

AI algorithms have attracted close attention of researchers and have also been applied successfully to solve problems in engineering. Nevertheless, for large and complex problems, AI algorithms consume considerable computation time due to stochastic feature of the search approaches

1) Business; financial strategies
2) Engineering: check design, offer suggestions to create new product, expert systems for all engineering problems
3) Manufacturing: assembly, inspection and maintenance
4) Medicine: monitoring, diagnosing
5) Education: in teaching
6) Fraud detection
7) Object identification
8) Information retrieval
9) Space shuttle scheduling

Building AI Systems:

1) Perception Intelligent biological systems are physically embodied in the world and experience the world through their sensors (senses). For an automatic vehicle, input may be images from a camera and range information from a rangefinder. For a medical diagnosis system, perception is the set of symptoms and test results that have been obtained and input to the system manually.
2) Reasoning Inference, decision-making, classification from what is sensed and what the internal "model" is of the world. Might be a neural network, logical deduction system, Hidden Markov Model induction, heuristic searching a problem space, Bayes Network inference, genetic algorithms, etc. Includes areas of knowledge representation, problem solving, decision theory, planning, game theory, machine learning, uncertainty reasoning, etc.
3) Action Biological systems interact within their environment by actuation, speech, etc. All behaviour is centred around actions in the world. Examples include controlling the steering of a Mars rover or autonomous vehicle, or suggesting tests and making diagnoses for a medical diagnosis system. Includes areas of robot actuation, natural language generation, and speech synthesis.

Intelligent Agent

Agent: Anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators. An Agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.

- ✓ A human agent has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for actuators.
- ✓ A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators.
- ✓ A software agent receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

Percept: The agent's raw perceptual input at a given time (a snapshot). We use the term percept to refer to the agent's perceptual inputs at any given instant.

- **Percept Sequence:** The complete history of everything the agent has ever perceived.
- **Agent Function (f):** An abstract, mathematical description of an agent's behavior, mapping every possible percept sequence to an action:
- Agent Function: f:Percept Sequence$*\rightarrow$Action
- **Agent Program:** The concrete implementation (code) that executes the agent function within its physical or software **Architecture**.
- Architecture + Agent Program=Agent

 Rational Agent: An agent that acts to achieve the best outcome or, when there is uncertainty, the best expected outcome. given its current knowledge.

 Performance Measure (P): The criteria that defines the success of an agent's behavior. It is what the agent strives to maximize (e.g., maximizing profit, minimizing error, completing the most tasks).

 PEAS Description: A tool for specifying the context of an agent:

- **P**erformance Measure (e.g., safe, fast, legal, comfortable trip).
- **E**nvironment (e.g., Roads, traffic, pedestrians, weather).
- **A**ctuators (e.g., Steering, accelerator, brake, horn, display).
- **S**ensors (e.g., Cameras, GPS, speedometer, odometer, sonar).
- *Example: Self-Driving Car*.

Environment Properties

Feature	**Definition**	Example
Fully Observable vs. Partially Observable	Agent's sensors give it access to the **entire** state of the environment.	Chess (Fully Observable)
Deterministic vs. Stochastic	The next state is completely determined by the current state and the agent's action (no randomness).	Card Games (Stochastic)
Episodic vs. Sequential	Action choice in one episode does not affect future episodes. Sequential tasks require planning ahead.	Part-Picking Robot (Episodic)
Static vs. Dynamic	The environment does not change while the agent is deliberating.	Crossword Puzzle (Static)
Discrete vs. Continuous	Limited, clearly defined percepts/actions (e.g., on/off, city-to-city) vs. numerical ranges (e.g., speed, temperature).	Digital Camera (Discrete)

Agents and environments:

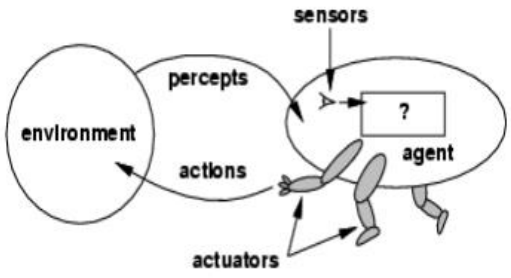


Fig- Agent and Environment

Types of Intelligent Agents (The Structure of the Program)

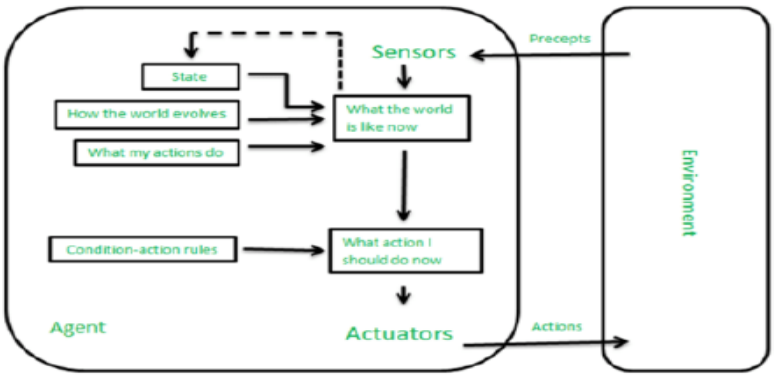
Agents are typically categorized in order of increasing complexity and capability.

Simple Reflex Agent

- **Logic: Condition-Action Rule:** *If [current condition], then [action].*
- **Decision Basis:** Acts only on the **current percept**, ignoring the percept history. It is memoryless.
- **Limitation:** Only works if the environment is **fully observable** and the appropriate action is obvious from the immediate percept (e.g., a simple thermostat).

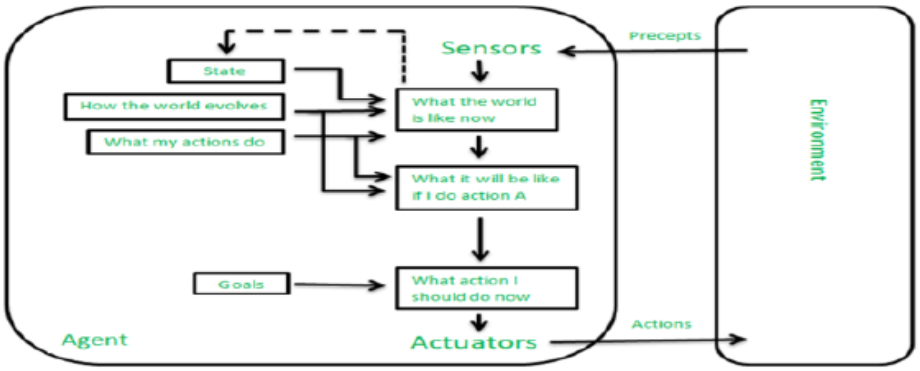
Model-Based Reflex Agent

- **Logic:** Maintains an **Internal State** (memory) to track the unobserved aspects of the environment.
- **Components:**
    1. **How the world evolves:** What the agent's state is likely to be next.
    2. **How the environment is determined by the agent's actions:** The effect of its own actions.
- **Decision Basis:** Current percept + Internal State (Model of the World).
- **Advantage:** Can operate in **partially observable** environments by inferring the true state (e.g., a robot vacuum cleaner that maps out where it has cleaned).



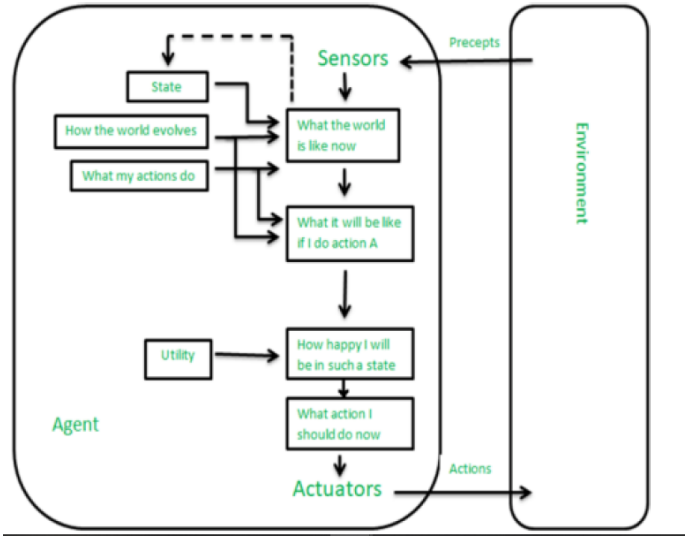
Goal-Based Agent

- **Logic:** Maintains an internal state **AND** has **explicit goals** to achieve.
- **Decision Basis:** Chooses actions by considering the future: Will the outcome of this action lead me toward my goal?
- **Mechanism:** Typically involves **search** and **planning** algorithms (like A* or DFS) to find a sequence of actions that achieves the goal.
- **Advantage:** Provides flexibility, as the agent can change its plan if the environment changes (e.g., a navigation system planning a route).



Utility-Based Agent

- **Logic:** Has goals, but also maximizes a **Utility Function**.
- **Utility Function:** A measure of how much an agent "likes" a given state, often based on a trade-off between competing goals (e.g., speed, safety, cost).
- **Decision Basis:** Chooses the action that leads to the state with the highest **Expected Utility**.
- **Advantage:** Necessary for complex, **stochastic** environments where there are multiple goals, or where achieving the goal has varying degrees of success (e.g., an autonomous stock trading agent balancing risk and return).



Learning Agent (The ML Component)

- **Concept:** Any of the above agent types can be turned into a learning agent by adding a **Learning Element**.
- **Structure:**
    1. **Performance Element:** The current *agent program* that selects actions.
    2. **Learning Element:** Responsible for making improvements by using feedback. This is the **Machine Learning** component.
    3. **Critic:** Measures the agent's performance (how well it's doing) based on the **Performance Standard** (utility/reward function) and provides feedback to the learning element.
    4. **Problem Generator:** Suggests new, exploratory actions to ensure the agent gathers new, informative experiences (exploration vs. exploitation).
- **Role of ML:** Machine Learning algorithms (e.g., Reinforcement Learning, Deep Learning) are implemented in the **Learning Element** to update the internal state, the rules, or the utility function, allowing the agent to become **autonomous** and adapt to unforeseen environments.

Problem-Solving Agents

- **Definition:** A problem-solving agent is a type of **Goal-Based Agent** that decides what action to take by finding a sequence of actions that leads from the current state to a desired goal state.
- **Core Principle:** It uses **search** and **planning** techniques to explore possible sequences of actions before executing them.
- **Contrasts with Reflex Agents:** Unlike simple reflex agents, which react based only on the current percept, a problem-solving agent actively analyzes a situation and considers the *long-term consequences* of its actions.

The Problem-Solving Process

A problem-solving agent typically follows four main steps:

1. **Goal Formulation:**
    - The first and simplest step.
    - Based on the current situation and the performance measure, the agent defines the set of **Goal States** it wants to achieve.
    - *Example:* For a trip, the goal is "Be in City X."
2. **Problem Formulation (State-Space Representation):**
    - The process of deciding what actions and states to consider to achieve the goal.
    - This defines the **State Space**—the set of all possible states reachable from the initial state.
    - It requires five key components:
        - **Initial State:** The starting point of the agent.
        - **Actions:** A description of all possible actions available to the agent (e.g., in a route-finding problem, the actions are "Drive from City A to City B").

- **Transition Model (Successor Function):** A description of the resulting state after performing a given action in a given state: Result(State,Action)→New State.
- **Goal Test:** A condition that determines if a given state is the goal state.
- **Path Cost:** A function that assigns a numerical cost to a sequence of actions (a path). The agent typically seeks a path with the lowest cost (**Optimal Solution**).

3. **Search:**
   - The core computational step.
   - The agent explores the State Space (a search tree/graph) to find a sequence of actions (a path) that leads from the initial state to a goal state.
   - **Input:** The formal problem definition.
   - **Output:** A **Solution** (a sequence of actions).
4. **Execution:**
   - The agent executes the first action of the calculated solution path.
   - After execution, the agent receives a new percept, and the cycle (Perceive, Formulate, Search, Execute) repeats.

Problem Characteristics and Problem Types

Characteristic	Implication for the Agent	Example
Ignorable vs. Recoverable	Mistakes can be undone (simpler search, like the 8-puzzle) vs. mistakes are permanent (requires careful, planned actions, like a self-driving car).	Tuning ML Hyperparameters (Ignorable)
Deterministic vs. Stochastic	Predictable action outcomes vs. uncertain outcomes (requires more complex, probability-based planning).	Chess (Deterministic)
Static vs. Dynamic	Environment does not change while the agent is planning (simpler search) vs. environment changes (requires online search or re-planning).	Crossword Puzzle (Static)

Export to Sheets

Search Algorithms (The ML/AI Techniques)

Search algorithms are the heart of the problem-solving agent and are classified into two main types:

A. Uninformed Search Strategies (Blind Search)

These algorithms have no information about the state space beyond what is provided in the problem formulation. They explore systematically.

- **Breadth-First Search (BFS):** Explores all nodes at a given depth before moving to the next level. **Complete** (will find a solution if one exists) and **Optimal** (finds the shortest path in terms of number of steps).
- **Uniform-Cost Search (UCS):** Expands the node with the lowest **Path Cost** first. **Optimal** for any positive step cost.

- **Depth-First Search (DFS):** Explores as far down a path as possible before backtracking. Not guaranteed to be optimal or complete (can get stuck in an infinite path).
- **Iterative Deepening Search (IDS):** Combines the completeness/optimality of BFS with the space-efficiency of DFS.

B. Informed Search Strategies (Heuristic Search)

These algorithms use a **Heuristic Function (h(n))** to estimate the cost from the current state (n) to the goal state. This guides the search, making it much more efficient.

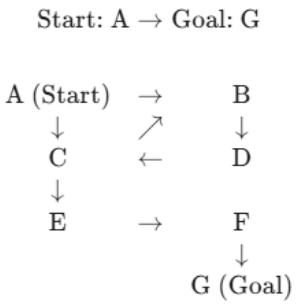
- **Greedy Best-First Search:** Expands the node that appears closest to the goal, as estimated by h(n).
- *A Search:** Combines the actual cost to reach a state, g(n), with the estimated cost to reach the goal, h(n). It expands the node with the lowest total estimated cost: f(n)=g(n)+h(n). **Optimal** and **Complete** if the heuristic is **admissible** (never overestimates the cost to the goal).

Breadth-First Search (BFS):

- Breadth-First Search (BFS) is a fundamental **uninformed search algorithm** used for traversing or searching tree or graph data structures. It is characterized by its systematic, exhaustive approach of exploring the search space **level by level** (or depth by depth).
- **The Traversal Rule (Breadth):** BFS always expands the nodes that are closest to the starting node first. It completely explores all nodes at depth d before moving on to any node at depth d+1.
- It operates like ripples spreading on a pond. From the starting node, it expands all immediate neighbors (Depth 1). Once all Depth 1 nodes are visited, it moves on to all nodes at Depth 2, then Depth 3, and so on.
- **Data Structure:** This level-by-level processing is rigorously enforced by a **Queue (FIFO)**. Nodes are expanded in the order they were discovered, ensuring that a node will only be expanded after all its "shallower" siblings and ancestors are complete.
- It is the **shortest-path guarantee** search for unweighted graphs. Because it always explores the minimal-depth nodes first, the very first time it finds the goal, it is guaranteed to have found a path with the fewest steps.
- **Space Complexity:** High, O(bd), where b is the branching factor and d is the depth of the shallowest goal, as it stores all nodes at the current level.

Example: Finding a Path in a City Map

Imagine searching for the shortest route from **City A** (Start) to **City G** (Goal) in the following simplified graph:



BFS Traversal Steps

We use a **Queue** to track the nodes to be visited and a **Visited** set to prevent cycles.

Step	Queue (FIFO)	Node Expanded	Neighbors Added	Path/Goal Found?	Depth
0	{A}	-	-	-	0
1	{C,B}	**A**	C,B	No	1
2	{B,E}	**C**	E (A is Visited)	No	2
3	{E,D}	**B**	D (A is Visited)	No	2
4	{D,F}	**E**	F (C is Visited)	No	3
5	{F}	**D**	(B is Visited, C is Visited)	No	3
6	{G}	**F**	G (E is Visited)	No	4
7	{$ \emptyset $}	**G**	-	**GOAL FOUND!**	5

Path: BFS found the path A→C→E→F→G.

Depth First Search(DFS)

Depth-First Search (DFS) is a fundamental **uninformed search algorithm** used to traverse or search through trees and graphs. DFS is a **graph or tree traversal algorithm** that explores the depth of a structure before exploring its breadth. In simple terms, DFS tries to **go as deep as possible in one direction** before backtracking and trying other branches. This characteristic makes DFS different from other search strategies like Breadth-First Search (BFS), which explores all neighboring nodes at the same level before moving deeper.

It is Uninformed / blind search

Search Strategy: LIFO (Last In First Out) → uses **stack** (explicit or recursion).

Completeness: Not complete in infinite-depth or cyclic graphs.

Optimality: Not optimal (may not find the shortest path).

Time Complexity: O(b^m)

- b = branching factor
- m = maximum depth of the search tree

Space Complexity: O(b × m) (less than Breadth-First Search for large branching factors).
The main idea is to explore **as far as possible along a branch** before backtracking.

DFS is widely used in:

- Solving puzzles and games
- Graph traversal
- Pathfinding
- Topological sorting
- Maze solving

DFS Algorithm – Step by Step

1. Start at the initial (root) node.
2. Push the starting node onto the stack (or call recursively).
3. Pop the top node from the stack and visit it.
4. If the node is the **goal**, stop.
5. Else, push all **unvisited successors** of this node onto the stack.
6. Repeat steps 3–5 until the stack is empty or the goal is found.

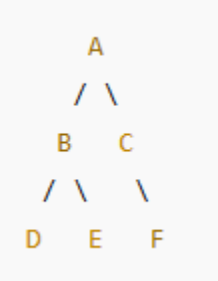
Pseudocode-

```
DFS(start, goal):
   create a stack S
   push start onto S
   mark start as visited

   while S is not empty:
      node = S.pop()
      if node == goal:
         return "Goal Found"
      for each neighbor of node:
         if neighbor not visited:
            push neighbor onto S
            mark neighbor as visited
   return "Goal Not Found"
```

Python Code-

```
def dfs(node, goal, visited):
   if node == goal:
      print("Goal Found")
      return True
   visited.add(node)
   for neighbor in graph[node]:
      if neighbor not in visited:
         if dfs(neighbor, goal, visited):
```

```
            return True
    return False
```



DFS Traversal Order (starting from A):
A → B → D → E → C → F

DFS vs BFS

Feature	DFS	BFS
Data Structure	Stack / Recursion	Queue
Space Requirement	Low	High
Finds Shortest Path	✗ No	✓ Yes
Completeness	No (in infinite trees)	Yes
Preferred For	Deep graphs, memory efficiency	Shallow graphs, optimal solutions

Advantages of DFS

- Uses less memory.
- Can be easily implemented using recursion.
- Useful in topological sorting, cycle detection, pathfinding in deep structures.

Disadvantages of DFS

- May get stuck in infinite loops without proper checks.
- Not guaranteed to find the shortest path.
- May explore unnecessary paths.

Applications of DFS

- Solving mazes and puzzles (e.g., **Sudoku**).
- Detecting cycles in a graph.
- Topological sorting in Directed Acyclic Graphs (DAG).
- Pathfinding in **game AI**.
- Component finding in **graph connectivity** problems.

Hill-climbing Search

Hill climbing search is a **local search algorithm** used in Artificial Intelligence for **optimization and search problems**. It starts with an **arbitrary solution** and **iteratively moves** to a neighboring solution with a **better (higher or lower) evaluation**. The algorithm is inspired by the idea of "**climbing up a hill**" to reach the **top (optimum)**.

Hill climbing works by **continually improving the current state**. It uses an **evaluation function (heuristic)** to decide the best next move. It **does not maintain a search tree** like other search algorithms. It is **greedy in nature**—always chooses the best neighbor.

Algorithm Steps

1. **Start** with an initial state.
2. **Evaluate** the current state.
3. **Generate neighbors** (possible next states).
4. **Choose the neighbor** with the best evaluation.
5. If the neighbor is **better than the current state**,
   → Move to that neighbor.
   Else,
   → **Stop** (local maximum reached).
6. **Repeat** until termination condition is met.

Types of Hill Climbing

1. **Simple Hill Climbing:**
   - Examines neighbors one by one.
   - Moves to the first better neighbor found.
2. **Steepest-Ascent Hill Climbing:**
   - Evaluates all neighbors.
   - Moves to the best of all neighbors.
3. **Stochastic Hill Climbing:**
   - Chooses **random neighbors** and decides probabilistically whether to move.

Hill climbing limitations:

- **Local maxima:** A peak lower than the global maximum.
- **Plateaus:** Flat areas where neighbors have equal value.
- **Ridges:** Paths that require indirect moves to improve.

Applications:

- Function optimization
- Robot path planning
- Scheduling problems
- Feature selection in Machine Learning
- Game playing (e.g., puzzle solving)

Simulated annealing search:

Simulated annealing is a **probabilistic local search algorithm** inspired by the **annealing process in metallurgy**, where a metal is **heated and then slowly cooled** to remove defects and achieve a more stable structure. Unlike hill climbing, simulated annealing **allows occasional "downhill" moves** (worse solutions) to **escape local optima** and **increase the chance of reaching the global optimum**.

Algorithm Steps

1. **Start** with:
   - Initial state `current`
   - Initial temperature `T`
2. **Repeat** until stopping condition:
   - Generate a **random neighbor** `next`.
   - Compute the **change in value**:
     $\Delta E =$ `Value(next)` − `Value(current)`
   - If $\Delta E > 0$ (better move), **accept** `next`.
   - Else (worse move), accept `next` with probability
     P = e^(ΔE / T) (Boltzmann probability)
   - **Decrease temperature** `T` gradually.
3. **Return** the best state found.

The **cooling schedule** controls how temperature decreases:

- **Exponential cooling:** `T = T × α` $(0 < \alpha < 1)$
- **Linear cooling:** `T = T - β`
- **Logarithmic cooling:** decreases slowly to give more exploration.

Advantages Over Hill Climbing

Hill Climbing	Simulated Annealing
Deterministic	Probabilistic
Easily stuck in local maxima	Can escape local maxima
Fast convergence	Slower but more likely to find global optima
Greedy search	Balanced exploration and exploitation

Applications

- Combinatorial optimization problems (e.g., Travelling salesman problem)
- Scheduling and timetabling
- VLSI chip design
- Function optimization in continuous domains
- Machine learning parameter tuning
- Path planning and routing problems

Pseudocode

```
function SimulatedAnnealing(problem, schedule):
   current ← initial_state(problem)
   for t = 1 to ∞:
      T ← schedule(t)
      if T = 0:
         return current
      next ← random_successor(current)
      ΔE ← Value(next) − Value(current)
      if ΔE > 0:
         current ← next
      else if random(0,1) < exp(ΔE / T):
         current ← next
```

Local Search in Continuous Spaces

Local search is a family of algorithms that focus on finding optimal or near-optimal solutions by **exploring neighboring states** of the current solution. Most introductory local search algorithms like Hill climbing search or Simulated annealing are often explained in **discrete search spaces** (e.g., puzzles, graphs). However, many real-world problems—such as **function optimization**, **parameter tuning**, and **robotics control**—are defined in **continuous domains**, where:

- States are **points in $\mathbb{R}^n$ (n-dimensional real space)**.
- The **neighboring states** can be **infinitely many**.
- Search methods must use **gradient information** or **random sampling**.

Continuous Local Search

- **Infinite state space** (unlike discrete problems).
- Often **objective functions** are continuous and differentiable.
- Movement is not from node to node, but **along a curve or surface**.
- Requires methods to **efficiently explore neighborhoods**.
- Common in engineering, machine learning, and control problems.

A. **Gradient Descent / Ascent**

Gradient descent is one of the most common techniques:

- Moves in the direction of **steepest descent** of the function.
- Useful when **gradient can be computed** analytically or numerically.

Update Rule:

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

Where:

- α = learning rate (step size)
- $\nabla f(x_t)$ = gradient at point x_t

Gradient ascent is similar but moves toward increasing values.

B. **Random-Restart Hill Climbing**

- Start at **multiple random points** in the continuous space.
- Run hill climbing or gradient search from each start.
- Keep the **best solution found**.
- Helps avoid local maxima/minima.

C. **Stochastic / Heuristic Methods**

- Instead of exact gradients, these use **probabilistic steps** to explore:
  - Simulated annealing
  - Genetic algorithm
  - Particle swarm optimization
- Useful when the **objective function is not differentiable** or **highly complex**.

Handling Local Optima

Continuous landscapes may have:

- **Local minima/maxima**
- **Plateaus** (flat regions)
- **Ridges and valleys**

Techniques to overcome these:

- Random restarts
- Variable step sizes
- Momentum in gradient descent
- Annealing or noise addition

Advantages of Local Search in Continuous Domains

- **Memory efficient** — stores only the current state and value.
- Works well for **high-dimensional real-valued problems**.
- Fast convergence in smooth landscapes.
- Easily combined with **optimization heuristics**.

Limitations

- May converge to **local optima** rather than global.
- Performance depends on:
  - Step size selection
  - Initialization point
  - Shape of the search landscape
- Gradients may be **difficult or expensive** to compute.

Applications

- **Machine learning model training** (e.g., neural networks)
- Parameter optimization in control systems
- Robotics path planning
- Engineering design optimization
- Economic and financial modeling

```
function LocalSearchContinuous(f, x0, step_size, max_iter):
   x ← x0
   for i in 1..max_iter:
      neighbor ← x + random_direction() * step_size
      if f(neighbor) < f(x):      # for minimization
         x ← neighbor
   return x
```

Heuristic functions

Heuristic function plays a central role in **informed search algorithms**. A heuristic function is used to **guide the search** toward the goal more efficiently by **estimating the cost** from the current state to the goal state.

- It provides **domain-specific knowledge** to speed up the search.
- It **does not guarantee optimality**, but helps to find good solutions faster.
- Commonly denoted by **h(n)**, where **n** is a node (state).

Heuristic = "rule of thumb" or an **educated guess** about the best path to the goal.

A **heuristic function** h(n)h(n)h(n) is a function that **estimates the cost of the cheapest path** from node n to a goal state.

- h(n)=0h(n) = 0h(n)=0 if nnn is a goal node.
- h(n)h(n)h(n) is always **non-negative**.
- The better the heuristic, the faster the search algorithm performs.Role of Heuristic Functions in Search

Heuristic functions are used in:

- Best-first search
- A* search algorithm
- Greedy search algorithm
- Hill climbing search

Greedy search: uses h(n)h(n)h(n) only.
*A search:** uses f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n), where:

- g(n)g(n)g(n) = cost from start node to current node.
- h(n)h(n)h(n) = estimated cost from current node to goal.

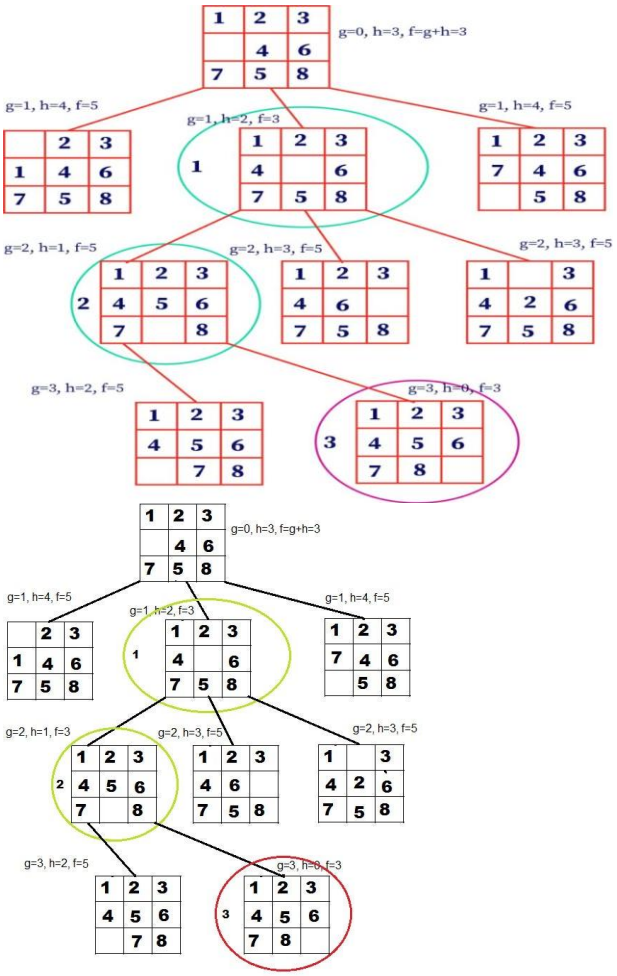
- f(n)f(n)f(n) = estimated total cost of path through nnn.

A good heuristic should:

- Be **easy and fast to compute**.
- **Reduce the number of states** explored during the search.
- **Guide the search efficiently** toward the goal.
- Be **reasonably accurate** in estimating the remaining cost.

Examples of Heuristic Functions

A. 8-Puzzle Problem



- Goal: move tiles to reach a target configuration.
- Common heuristics:
  - h1(n)h_1(n)h1(n): Number of misplaced tiles.
  - h2(n)h_2(n)h2(n): Total Manhattan distance (sum of distances each tile is from its goal position).

B. Route-Finding Problem

- Heuristic: **Straight-line distance** between the current city and the goal city.

C. Maze Problem

- Heuristic: **Euclidean or Manhattan distance** between current cell and goal cell.

---

7. Designing Heuristic Functions

Steps to design effective heuristics:

1. Use **domain knowledge** to estimate the cost.
2. Ensure **admissibility** if optimality is required.
3. Simplify **complex calculations** for efficiency.
4. Sometimes combine multiple heuristics (e.g., h(n)=max⁡(h1(n),h2(n))h(n) = \max(h_1(n), h_2(n))h(n)=max(h1(n),h2(n))).

---

8. Evaluation of Heuristics

- **Accuracy**: How close h(n)h(n)h(n) is to the actual cost.
- **Efficiency**: How quickly h(n)h(n)h(n) can be computed.
- **Informedness**: Better heuristics lead to fewer nodes expanded.
- **Dominance**: If h2(n)≥h1(n)h_2(n) \geq h_1(n)h2(n)≥h1(n) for all nnn, and both are admissible, then h2h_2h2 is said to **dominate** h1h_1h1.

---

9. Advantages

- Speeds up search algorithms significantly.
- Helps reach near-optimal solutions faster.
- Reduces search complexity in large state spaces.
- Can be adapted to many different domains.

---

10. Limitations

- May not guarantee optimal solution (if not admissible).
- Poor heuristics can lead to inefficient search.
- Designing good heuristics often requires **expert domain knowledge**.
- Computation of heuristic may itself be expensive.

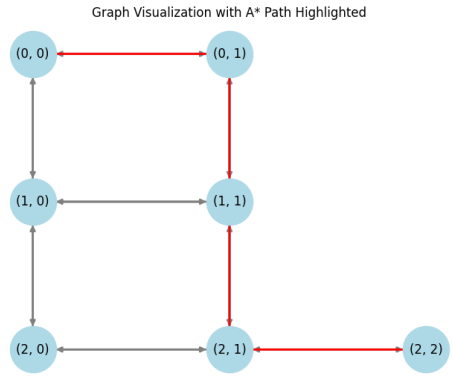
---

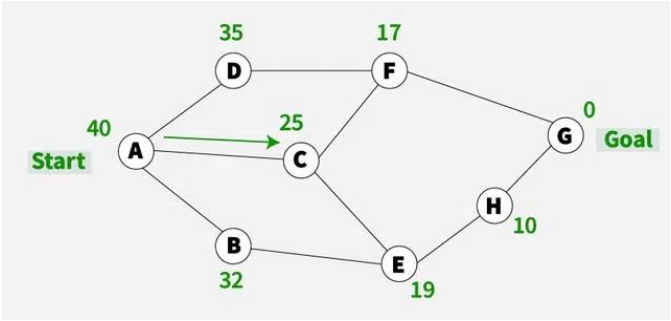
11. Pseudocode: Greedy Best-First Search Using h(n)

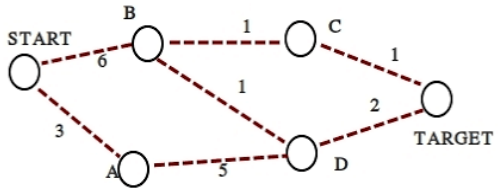
```
function GreedyBestFirstSearch(problem):
    frontier ← priority queue ordered by h(n)
```

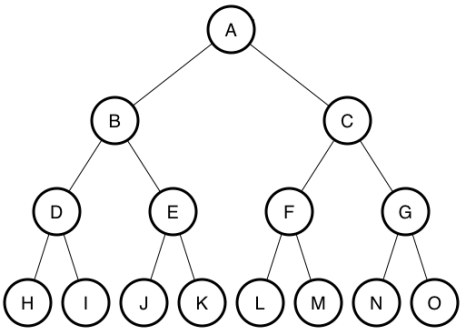
```
frontier.insert(problem.initial_state)
explored ← {}
while frontier not empty:
    node ← frontier.pop()
    if problem.goal_test(node):
        return solution(node)
    explored.add(node)
    for child in expand(node):
        if child not in explored:
            frontier.insert(child)
```

Visualization









- Heuristic functions **guide** the search in the direction of the **goal**.
- Better heuristics = **fewer expanded nodes** = **faster convergence**

Best First Search

Best-first search is an **informed search algorithm** that uses a **heuristic function** to **select the most promising node** to explore next.

- It combines features of both **depth-first** and **breadth-first search**.
- It **prioritizes nodes** based on an **evaluation function**:

f(n)=h(n)f(n) = h(n)f(n)=h(n)

or sometimes
f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n)

- It **aims to reach the goal faster** by guiding the search toward the most promising direction.

"Best" refers to the node with the **lowest evaluation function value**.

- **Heuristic Function (h(n))**: Estimated cost from current node to goal.
- **Evaluation Function (f(n))**: Used to rank nodes for exploration.
- **Priority Queue (OPEN list)**: Stores frontier nodes ordered by f(n).
- **Closed List**: Keeps track of visited nodes

Algorithm Steps

1. **Initialize** the OPEN list with the start node.
2. **Repeat** until OPEN is empty:
    - Choose the node `n` from OPEN with the **lowest f(n)**.
    - If `n` is the goal → return the solution.
    - Else, expand `n` and generate its successors.
    - Evaluate successors and add them to OPEN.
3. Keep track of visited nodes to **avoid cycles**.

Variants of Best-First Search

A. Greedy Best-First Search

- Uses only heuristic function:
  f(n)=h(n)f(n) = h(n)f(n)=h(n)
- Selects the node that appears closest to the goal.
- Fast but **may not be optimal**.

B. **A Search***

- Combines actual cost and heuristic:
  f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n)
- Guarantees **optimality** if heuristic is admissible.

```
function BestFirstSearch(problem):
   frontier ← priority queue ordered by f(n)
   frontier.insert(initial_state)
   explored ← {}
   while frontier is not empty:
      node ← frontier.pop()
      if goal_test(node):
         return solution(node)
      explored.add(node)
```

```
for child in expand(node):
   if child not in explored and child not in frontier:
      frontier.insert(child)
```

Characteristics

Feature	Best-First Search
Type of search	Informed
Data structure used	Priority Queue
Completeness	Yes (if branching factor is finite)
Optimality	Not guaranteed (Greedy), yes with A*
Time complexity	Depends on heuristic and branching factor
Space complexity	Can be high due to storing frontier

Advantages

- Explores **most promising paths first**.
- More efficient than uninformed search (e.g., BFS, DFS).
- Can **significantly reduce search space**.
- Works well when a **good heuristic** is available.

Limitations

- Performance depends on **quality of heuristic function**.
- May get stuck in loops without proper cycle checking.
- **Greedy Best-First** may not find the optimal path.
- May consume a lot of memory (due to storing all frontier nodes).

Applications

- Route/path planning problems
- Game playing and puzzle solving (e.g., 8-puzzle problem)
- Web crawling and search engines
- Network routing
- Robot navigation

A* Search

A* search algorithm is one of the **most popular and efficient informed search algorithms** in Artificial Intelligence.

It combines the advantages of:

- **Uniform Cost Search** (optimality), and
- **Greedy Best-First Search** (fast goal-directed search).

A* search uses both:

- **g(n)** → cost from start node to current node, and
- **h(n)** → estimated cost from current node to goal.

f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n)

- f(n)f(n)f(n) = total estimated cost of the path through node nnn.
- The node with the **lowest f(n)** is expanded first.
- **g(n)**: Actual cost from start to node n.
- **h(n)**: Heuristic estimate from node n to goal.
- **f(n)**: Estimated total path cost (g + h).
- **OPEN list**: Frontier nodes, ordered by f(n).
- **CLOSED list**: Explored nodes.

Algorithm Steps

1. Initialize:
    - Add start node to OPEN list.
    - f(start)=g(start)+h(start)f(start) = g(start) + h(start)f(start)=g(start)+h(start).
2. While OPEN is not empty:
    - Select node `n` with **lowest f(n)** from OPEN.
    - If `n` is the goal → return path.
    - Move `n` to CLOSED.
    - Expand all neighbors of `n`.
    - For each neighbor:
        - Compute g, h, f values.
        - If neighbor not in OPEN or better path found → update it.
3. If OPEN is empty and no goal found → failure.

```
function A*(start, goal):
  open ← priority queue containing start
  g[start] ← 0
  f[start] ← g[start] + h(start)

  while open is not empty:
    n ← node in open with lowest f(n)
    if n = goal:
      return reconstruct_path(n)

    remove n from open
    add n to closed

    for each neighbor of n:
      tentative_g ← g[n] + cost(n, neighbor)
      if neighbor in closed and tentative_g ≥ g[neighbor]:
        continue
      if neighbor not in open or tentative_g < g[neighbor]:
        parent[neighbor] ← n
        g[neighbor] ← tentative_g
```

```
    f[neighbor] ← g[neighbor] + h(neighbor)
    if neighbor not in open:
       add neighbor to open
```

The performance of A* depends heavily on the **quality of the heuristic** h(n)h(n)h(n).

- If h(n)=0h(n) = 0h(n)=0 → A* behaves like Uniform Cost Search.
- If h(n)h(n)h(n) is **admissible** and **consistent**, A* guarantees **optimal solution**.

Examples of heuristics:

- **Straight-line distance** for pathfinding.
- **Manhattan distance** in grid/maze problems.
- **Number of misplaced tiles** in 8-puzzle problem.

Properties of A* Search

Property	**Description**
Type	Informed search
Evaluation	f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n)
Completeness	Yes (if branching factor finite and step costs positive)
Optimality	Yes (with admissible heuristic)
Time complexity	Exponential in worst case
Space complexity	High (stores all generated nodes)

Advantages

Adversarial Search:

In artificial intelligence, search problems are not always limited to a single agent. Many real-world problems involve **two or more agents** interacting with each other. These agents are often in **competition**, where one tries to **maximize its gain** while the other tries to **minimize it**. Such problems are called **adversarial search problems** because each agent works against the other.

A **game** is a structured form of interaction between two or more agents, usually governed by a set of rules. Unlike ordinary search, in a game the outcome of an action depends not only on the agent's decision but also on the **opponent's move**. For example, in chess, the effectiveness of your move depends on how your opponent responds to it.

- In **single-agent search problems**, the environment is **deterministic and fully observable**.

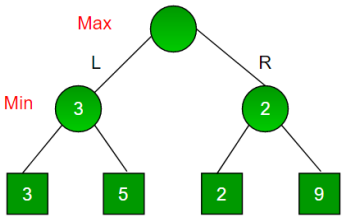
- In **adversarial search problems**, the environment includes **one or more opponents**.
- These problems are typically modeled as **games**, e.g., chess, tic-tac-toe, checkers.
- **Goal:** Choose moves that **maximize** the agent's chance of winning while **minimizing** the opponent's.

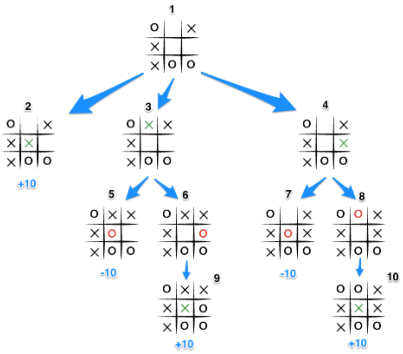
Game Playing

Types of Games

1. **Deterministic vs. Stochastic**
    - Deterministic: No chance elements (e.g., chess, tic-tac-toe).
    - Stochastic: Involves randomness (e.g., backgammon, card games).
2. **Perfect vs. Imperfect Information**
    - Perfect Information: All players have complete knowledge (e.g., chess).
    - Imperfect Information: Some information is hidden (e.g., poker).
3. **Zero-sum vs. Non-zero-sum**
    - Zero-sum: One player's gain is another's loss.
    - Non-zero-sum: Players may have partially aligned interests.

Game Tree





- A **game tree** represents all possible moves in a game.
- **Nodes** = Game states
- **Edges** = Moves
- **Root node** = Current state
- **Leaf nodes** = Terminal states (win/lose/draw).

Minimax Algorithm

Concept:

- Two players:
    - **MAX**: Tries to maximize the score.
    - **MIN**: Tries to minimize the score.
- The algorithm explores the game tree to determine the optimal move.

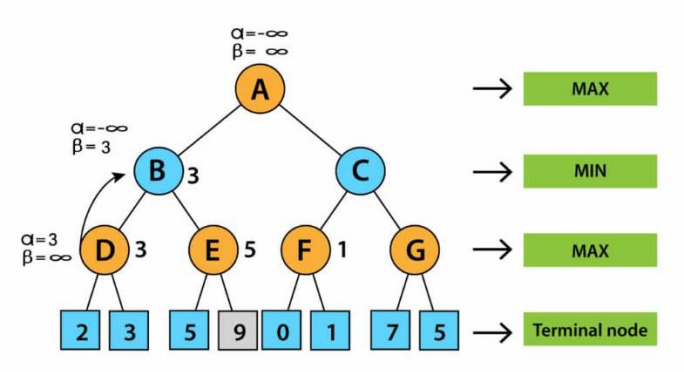
Minimax Value:

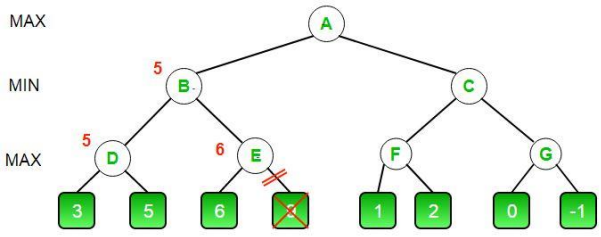
$$\text{Minimax}(n) = \begin{cases} \text{Utility}(n) & \text{if n is terminal} \\ \max\limits_{s \in Successors(n)} \text{Minimax}(s) & \text{if n is MAX} \\ \min\limits_{s \in Successors(n)} \text{Minimax}(s) & \text{if n is MIN} \end{cases}$$

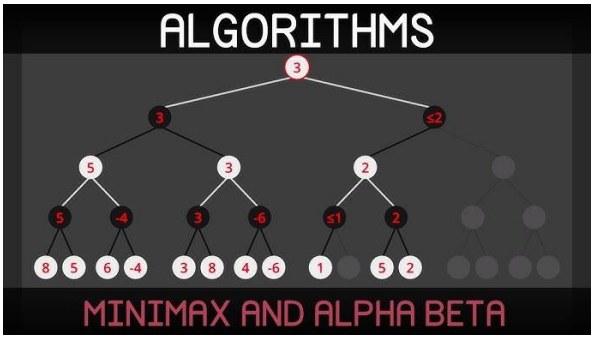
Pseudocode:

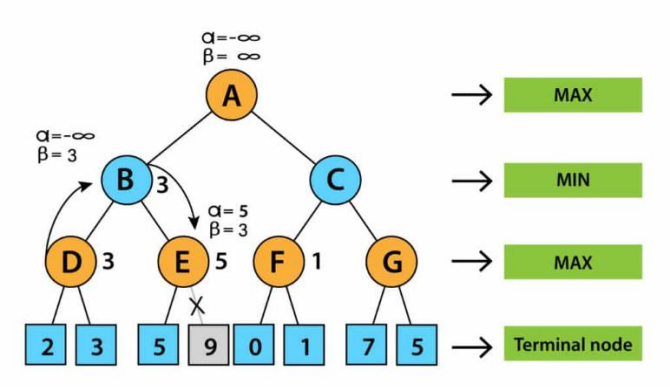
```
function MINIMAX-DECISION(state):
   return argmax_a MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state):
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for each a in ACTIONS(state):
      v ← max(v, MIN-VALUE(RESULT(state, a)))
   return v

function MIN-VALUE(state):
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← +∞
   for each a in ACTIONS(state):
      v ← min(v, MAX-VALUE(RESULT(state, a)))
   return v
```

Alpha–Beta Pruning









- **Improves efficiency** of minimax by pruning branches that **don't affect** the final decision.
- **Alpha (α)** = Best value that MAX can guarantee so far.
- **Beta (β)** = Best value that MIN can guarantee so far.

Pruning Rule:

- If at any point, $\beta \leq \alpha$, prune the remaining branches of that node.

Pseudocode:

```
function MAX-VALUE(state, α, β):
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for each a in ACTIONS(state):
      v ← max(v, MIN-VALUE(RESULT(state, a), α, β))
      if v ≥ β then return v
      α ← max(α, v)
   return v

function MIN-VALUE(state, α, β):
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← +∞
   for each a in ACTIONS(state):
      v ← min(v, MAX-VALUE(RESULT(state, a), α, β))
      if v ≤ α then return v
```

β ← min(β, v)
return v

Evaluation Functions

- Used when searching entire game tree is not feasible.
- Estimate the **"goodness" of a position**.
- Typically a linear weighted function of features:

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + ... + w_n f_n(s)$$

Examples:

- Number of pieces, positions, mobility, king safety, etc. in chess.

Stochastic Games and Expectiminimax

For games with randomness (e.g., dice rolls):

- Introduce **chance nodes** in the tree.
- Use **expectation over probabilities** at chance nodes.

$$\text{Expectiminimax}(n) = \begin{cases} \text{Utility}(n) & \text{if terminal} \\ \max(\text{children}) & \text{if MAX node} \\ \min(\text{children}) & \text{if MIN node} \\ \sum p(s)\text{Expectiminimax}(s) & \text{if Chance node} \end{cases}$$

Aspect	Single-Agent Search	Adversarial Search
Environment	Deterministic	Competitive
Optimal Strategy	Path cost	Minimax / Alpha-Beta
Solution	Optimal path	Optimal strategy
Complexity	Moderate	High due to opponent moves

Benefits of Alpha–Beta Pruning:

- Produces the same result as minimax.
- Significantly reduces the number of nodes to be evaluated.
- Makes it feasible to search deeper in the tree in the same amount of time.

Module II

Knowledge and Reasoning:

A Knowledge Based Agent

One of the most powerful ways to build intelligent systems is to give them the ability to **use knowledge** to make decisions. An agent that can **represent, store, and reason** with knowledge about the world is called a **knowledge-based agent**.

Unlike simple reflex agents that act only on the basis of current percepts, a knowledge-based agent uses **previous knowledge**, **inference**, and **logical reasoning** to make decisions. This makes it more **flexible**, **adaptable**, and **intelligent**.

Components of a Knowledge-Based Agent

A knowledge-based agent has two main components:

1. **Knowledge Base (KB):**
    - A **central repository** of facts about the world.
    - Consists of a set of **sentences** expressed in a formal language (e.g., propositional logic, first-order logic).
    - Example sentence: "If it rains, the ground will be wet."
2. **Inference Engine:**
    - A mechanism that **derives new information** from the knowledge base using logical reasoning.
    - It can answer queries and help the agent decide what to do.

Additional Functional Parts:

- **Percept sequence:** Information received from the environment.
- **Update mechanism:** Adds new percepts to the KB.
- **Query mechanism:** Asks questions to the KB to make decisions.
- **Action mechanism:** Executes actions based on conclusions drawn.

Knowledge-Based Agent – Working Cycle

The functioning of a knowledge-based agent can be summarized as follows:

1. **Perceive:** The agent receives percepts from the environment through its sensors.
2. **Tell:**The agent adds these percepts as **new facts** to the knowledge base.
3. **Ask:** The agent queries the knowledge base to infer **what actions** should be taken.
4. **Act:** Based on the inferred knowledge, the agent performs the appropriate action.

Representation of Knowledge

The effectiveness of a knowledge-based agent depends on how knowledge is represented. A **good knowledge representation** must be:

- **Expressive:** Able to represent all required knowledge.
- **Unambiguous:** Easy to interpret and understand.
- **Efficient:** Easy to store and retrieve.
- **Supports Inference:** Allows drawing logical conclusions.

Common Forms of Knowledge Representation:

- **Propositional Logic** – Represents facts as simple statements.
- **First-Order Logic (Predicate Logic)** – More expressive; allows representation of objects, relations, and properties.
- **Semantic Networks and Frames** – Represent structured relationships.
- **Rules and Production Systems** – If-Then rules used in expert systems.
- **Ontologies** – Structured knowledge domains.

Types of Knowledge

1. **Declarative Knowledge:**
    - Describes **facts** about the world.
    - Example: "Paris is the capital of France."
2. **Procedural Knowledge:**
    - Describes **how to do things** (procedures or operations).
    - Example: "How to ride a bicycle."
3. **Meta-Knowledge:**
    - Knowledge about knowledge (e.g., knowing which strategy to use for reasoning).
4. **Heuristic Knowledge:**
    - Based on experience or practical rules of thumb.
    - Example: "If traffic is heavy, take an alternative route."
    - **Reasoning and Inference**

Reasoning is the process of drawing conclusions from known facts.
A **knowledge-based agent** uses an **inference engine** to perform reasoning and update its knowledge.

Types of Reasoning:

- **Deductive Reasoning:**
  Derives logically certain conclusions from given facts.
  Example:
    - Premise 1: All birds can fly.
    - Premise 2: Sparrow is a bird.
    - Conclusion: Sparrow can fly.

- **Inductive Reasoning:**
  Derives generalized conclusions from specific examples. (Less certain but useful for learning.)
- **Abductive Reasoning:**
  Infers the most likely explanation for observed data.

Inference Mechanisms

- **Forward Chaining:** Start with known facts and apply inference rules to reach a conclusion.
  (Data-driven reasoning.)
- **Backward Chaining:** Start with a goal and work backward to determine what facts support it.
  (Goal-driven reasoning.)

Example (Rule):
IF it is raining THEN the ground is wet.

- Forward chaining: Given "it is raining" → infer "ground is wet".
- Backward chaining: To prove "ground is wet" → check if "it is raining".

Advantages of Knowledge-Based Agents

- **Intelligent Behavior:** Acts based on reasoning, not just reflex.
- **Adaptability:** Can handle new situations by inferring solutions.
- **Explainability:** Can provide reasons for its actions.
- **Modularity:** Knowledge can be updated without redesigning the entire system.
- **Learning Capability:** Can grow more intelligent with experience.

Applications of Knowledge-Based Agents

Knowledge-based agents are widely used in:

- **Expert Systems** – Medical diagnosis, legal advisory.
- **Robotics** – Intelligent decision-making in uncertain environments.
- **Natural Language Processing** – Understanding and responding to human language.
- **Intelligent Tutoring Systems** – Personalized learning experiences.
- **Information Retrieval and Recommendation Systems.**

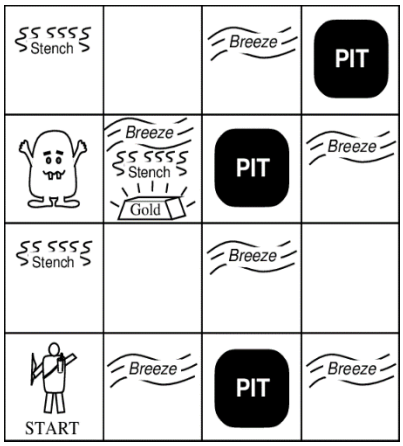
WUMPUS WORLD Environment

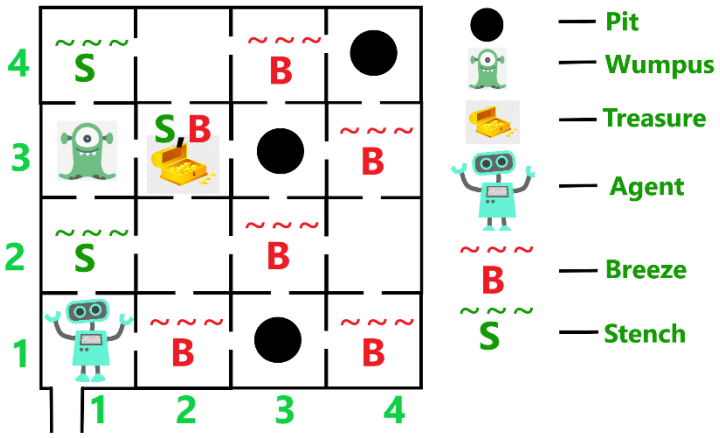
The Wumpus World is a grid-based environment used in artificial intelligence to demonstrate how knowledge-based agents perceive their surroundings and make logical decisions. It is a classic example from the field of logical agents and reasoning with uncertainty.

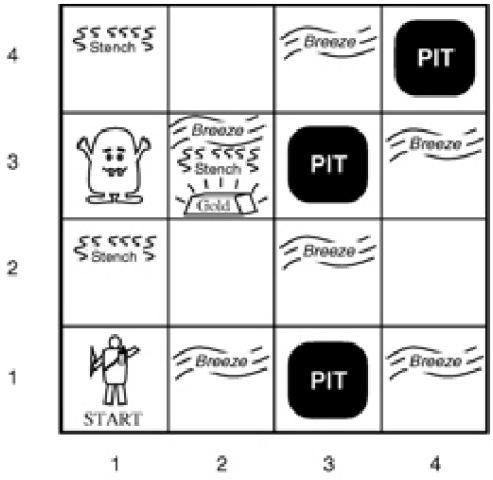
This world consists of rooms connected in a grid, typically a 4 × 4 layout. The agent moves through the rooms to find a gold treasure, while avoiding pits and a dangerous monster called the Wumpus.

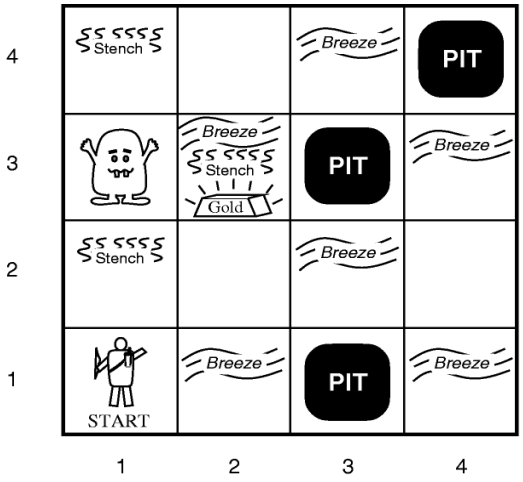
The Wumpus World is particularly useful because:

- It requires **perception** of indirect clues.
- The agent must use **inference and reasoning** rather than direct observation.
- It shows how **knowledge-based agents** can behave intelligently in uncertain environments.









Structure of Wumpus World

- The world is represented as a **4 x 4 grid** of rooms.
- Each room can contain:
  - A **pit** (deadly if entered),
  - The **Wumpus** (a dangerous creature),
  - The **gold** (the treasure),
  - Or be empty (safe).
- The **agent starts** at the bottom-left corner (1,1).
- The Wumpus does not move.
- There can be multiple pits but only **one Wumpus** and **one gold**.

Percepts in Wumpus World

The agent receives **percepts** from the environment in each room, which give **indirect clues** about nearby dangers.

Percept	Meaning	Indicates
Breeze	There is a pit in one of the **adjacent** rooms	Danger nearby (pit)
Stench	The Wumpus is in one of the **adjacent** rooms	Danger nearby (Wumpus)
Glitter	Gold is in the **current** room	Goal found
Bump	Agent has hit a wall	Boundary reached
Scream	Wumpus has been killed (if the agent shoots successfully)	

Agent's Capabilities

The agent can perform the following **actions**:

- **Move Forward:** Move in the direction the agent is facing.
- **Turn Left / Turn Right:** Change direction without moving.
- **Grab:** Pick up the gold if in the same room.
- **Shoot:** Fire an arrow in the direction the agent is facing (can kill the Wumpus if aligned).
- **Climb:** Exit the cave (from the starting point) with the gold.

Rules of the Game

- If the agent enters a room with a pit or the Wumpus → it dies.
- If the agent picks up the gold and climbs out safely → it **wins**.
- Each action has a **cost**.
- Killing the Wumpus earns **bonus points**, but the primary goal is **survival and gold collection**.

Perception and Knowledge

The agent **cannot see** the Wumpus or pits directly.
It must **infer** the location of dangers using percepts:

- If there is a **breeze** in a room, the agent infers that **at least one adjacent room** contains a pit.
- If there is a **stench**, the agent infers the Wumpus is in one of the adjacent rooms.
- If there is **no breeze and no stench**, the adjacent rooms are safe.

Example:
If the agent is in room (1,1) and perceives a breeze, then either (1,2) or (2,1) has a pit. The agent must reason which path is safe.

Knowledge Base and Logical Inference

The agent maintains a **knowledge base (KB)** of facts and uses **propositional logic** to infer new facts.

- Percepts are **added to the KB** as logical statements.
- Inference rules are applied to **deduce**:
    - Which rooms are safe.
    - Possible locations of pits or the Wumpus.
    - Best next action.

Example Logical Rules:

- If there is **no breeze** in (1,1), then **(1,2)** and **(2,1)** have no pits.
- If there is a **stench** in (2,1), then Wumpus is in (2,2) or (3,1).

Example

Imagine the agent at (1,1) with no breeze or stench:

- It infers that (1,2) and (2,1) are safe.
- It moves to (1,2) and perceives breeze.
- It infers a pit must be either at (1,3) or (2,2).
- If it then moves to (2,1) and perceives no breeze, it can deduce:
    - (2,2) is safe,
    - Therefore, the pit must be at (1,3).

The performance of an agent in Wumpus World is typically evaluated by:

- Gold collected (reward),
- Penalties for each action,
- Survival,
- Bonus for killing Wumpus.

Typical Scoring:

- +1000 for climbing out with gold,
- –1000 for dying,
- –1 per move,
- –10 for shooting arrow.

A **rational agent** maximizes its performance by balancing risk and reward.

Propositional Logic

Propositional Logic (also called **Sentential Logic** or **Boolean Logic**) is the simplest form of logic used in Artificial Intelligence to represent facts and reason about them. It deals with **propositions**, which are statements that can either be **True (T)** or **False (F)**.

Example:

- "It is raining." → True or False
- "The light is on." → True or False

Components of Propositional Logic

- **Propositions:** Basic statements (e.g., P, Q, R).
- **Logical Connectives:**
  - **¬ (NOT)**: Negation — reverses truth value.
    - ¬P (not P)
  - **∧ (AND)**: Conjunction — true if both are true.
    - P ∧ Q
  - **∨ (OR)**: Disjunction — true if at least one is true.
    - P ∨ Q
  - **⇒ (IMPLIES)**: Implication — if P then Q.
    - P ⇒ Q
  - **⇔ (BICONDITIONAL)**: Equivalence — true if both sides have the same truth value.
    - P ⇔ Q

Syntax and Semantics

- **Syntax** specifies the rules for forming valid sentences in propositional logic.
  - Example: (P ∧ Q) ⇒ R is a valid sentence.
- **Semantics** defines the meaning of these sentences.
  - Truth values are assigned to propositions to evaluate the sentence.
  - Example: If P = T, Q = T, then (P ∧ Q) = T.

Truth Tables

A truth table lists all possible truth values of propositions and shows the resulting value of a complex expression.

P	**Q**	**P ∧ Q**	**P ∨ Q**	**P ⇒ Q**	**P ⇔ Q**
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

Knowledge Representation Using Propositional Logic

- **Knowledge Base (KB):** A collection of propositional logic sentences that describe the environment.
- Example (in Wumpus World):
  - $P_{1,2} \Rightarrow Breeze_{1,2}$ (If there is a pit at (1,2), there will be breeze at (1,2))
  - $\neg Pit_{1,1}$ (No pit at location (1,1))
- KB can be used by inference mechanisms to **derive new knowledge**.

Inference in Propositional Logic

- **Inference** is the process of deriving new sentences from the knowledge base.
- Common inference methods:
  - **Modus Ponens:**
    - If $(P \Rightarrow Q)$ and P are true, then Q is true.
  - **Modus Tollens:**
    - If $(P \Rightarrow Q)$ and $\neg Q$ are true, then $\neg P$ is true.
  - **Resolution:**
    - A rule of inference used in automated theorem proving.

- **Entailment ($\models$):**

$KB \models \alpha$ means that α logically follows from KB.

- **Logical Equivalence ($\equiv$):**

Two sentences are logically equivalent if they have the same truth value in all models.

Forms-

Negation Normal Form (NNF): Negations only appear directly in front of propositions.

Conjunctive Normal Form (CNF): A conjunction of disjunctions of literals (used in resolution).

- Example: $(P \vee Q) \wedge (\neg P \vee R)$

Applications of Propositional Logic in AI

- Knowledge-based systems for reasoning and decision-making.
- Game playing and planning problems.
- Expert systems (e.g., medical diagnosis).
- Robotics and agent-based systems**.**
- Natural language understanding (logical form representations).
- Automated theorem proving and SAT solvers.

First Order Predicate Logic (FOPL)

- **First Order Predicate Logic (FOPL)** — also called **First Order Logic (FOL)** — is an extension of **propositional logic** that allows us to represent **relationships between objects**, **properties of objects**, and **quantified statements**.

- While propositional logic can only express "facts," FOPL can express **general statements** like:

  - "All humans are mortal."
  - "There exists a student who studies AI."

- FOPL is more **expressive and powerful** than propositional logic and is widely used in knowledge-based AI systems.

Elements of FOPL

Element	Description	Example
Constants	Represent specific objects	John, India, 5
Variables	Represent arbitrary objects	x, y, z
Predicates	Represent properties or relationships between objects	Human(x), Loves(John, x)
Functions	Map objects to objects	Father(John)
Quantifiers	Used to express generality	∀ (for all), ∃ (there exists)
Connectives	Logical connectives	¬, ∧, ∨, ⇒, ⇔

Syntax of FOPL

- **Atomic Sentence:** Predicate with terms.
  Example: `Loves(John, Mary)` means "John loves Mary."
- **Complex Sentence:** Built using logical connectives and quantifiers.
  Example: `∀x (Human(x) ⇒ Mortal(x))` means "For all x, if x is a human, then x is mortal."
- **Terms:** Constants, variables, or functions.
- **Literals:** An atomic sentence or its negation.

Quantifiers

a) Universal Quantifier (∀)

- Denotes "for all".
- Example:
  - ∀x Human(x) ⇒ Mortal(x)
  - Meaning: "All humans are mortal."

b) Existential Quantifier (∃)

- Denotes "there exists".
- Example:
  - ∃x Student(x) ∧ Studies(x, AI)

- Meaning: “There exists at least one student who studies AI.”

Semantics of FOPL

- **Domain:** Set of all objects under consideration.
- **Interpretation:** Assigns meaning to constants, predicates, and functions.
- **Assignment:** Assigns values to variables.
- A sentence is **true** under an interpretation if it holds for all assignments of variables in the domain.

Examples

1. **Statement:** “Every student is intelligent.”
   - FOPL: $\forall x$ (Student(x) $\Rightarrow$ Intelligent(x))
2. **Statement:** “There exists a student who loves mathematics.”
   - FOPL: $\exists x$ (Student(x) $\wedge$ Loves(x, Mathematics))
3. **Statement:** “John likes everyone.”
   - FOPL: $\forall y$ (Person(y) $\Rightarrow$ Likes(John, y)

Conversion to Normal Form

For automated reasoning, FOPL sentences are often converted to Prenex Normal Form (PNF) and Conjunctive Normal Form (CNF)**:**

Steps:

1. Eliminate implications ($\Rightarrow$)
2. Move negations inward (using De Morgan’s Laws)
3. Standardize variables
4. Move quantifiers to the front (Prenex form)
5. Skolemization (remove existential quantifiers)
6. Drop universal quantifiers
7. Convert to CNF (conjunction of disjunctions)

Inference in FOPL

Unlike propositional logic, inference in FOPL involves **unification** and **resolution**.

- **Unification:** Process of making different logical expressions look identical by finding a substitution.
  - Example: Loves(John, x) and Loves(John, Mary) unify with x = Mary.
- **Resolution:** Rule of inference used for automated theorem proving.
  - If P(x) and $\neg$P(x) $\vee$ Q, then infer Q.
- **Generalized Modus Ponens:**
  - If $\forall x$ (P(x) $\Rightarrow$ Q(x)) and P(a) then Q(a).

Applications of FOPL in AI

- Knowledge-based systems and expert systems (e.g., diagnosis, decision support).
- Natural language understanding and processing (semantic representation).
- Automated reasoning and theorem proving**.**
- Robotics (reasoning about the world).
- Planning and problem-solving (goal-directed reasoning).
- Used in logic programming languages like Prolog.

Forward and Backward Chaining

In knowledge-based AI systems, inference is the process of deriving new facts or conclusions from a set of known facts and rules. Two important inference methods are:

- **Forward Chaining** (Data-driven reasoning)
- **Backward Chaining** (Goal-driven reasoning)

These techniques are widely used in expert systems, automated reasoning, diagnosis, and decision support systems.

Knowledge Representation in Rule-Based Systems

- Knowledge is often represented in the form of **IF–THEN rules**:

  IF <condition> THEN <action/conclusion>

  IF temperature is high AND patient has fever THEN patient is sick

The **facts** represent the **current state of knowledge**, and the **inference engine** applies rules to derive new knowledge.

Forward Chaining

Definition

- **Forward Chaining** is a **data-driven** inference technique.
- It starts from **known facts** and applies inference rules to **derive new facts** until a **goal or conclusion** is reached.

How It Works

1. Start with a set of known facts.
2. Check the rules to see which rules' premises (IF part) are satisfied.
3. Apply those rules to infer new facts.
4. Add the new facts to the knowledge base.
5. Repeat the process until the desired goal is derived or no more rules can be applied.

Algorithm (Steps)

1. Initialize known facts.
2. While goal not reached:
    - For each rule:
        - If the condition is satisfied by the facts:
            - Add the conclusion to the fact base.
    - Stop if no new fact is added.
3. End.

Example
Knowledge Base:

- Rule 1: IF A THEN B
- Rule 2: IF B THEN C
- Rule 3: IF C THEN D

Initial Fact:

- A is True

Forward Chaining:

- From A ⇒ infer B (Rule 1)
- From B ⇒ infer C (Rule 2)
- From C ⇒ infer D (Rule 3)

Goal D achieved**.**

Applications of Forward chaining

- Medical diagnosis systems
- Fault detection systems
- Real-time monitoring and decision systems
- Expert systems (e.g., MYCIN)

Backward Chaining

Definition

- **Backward Chaining** is a **goal-driven** inference technique.
- It starts from a **goal (hypothesis)** and works **backwards** to determine if there are facts that support it.

How It Works

1. Start with a **goal**.
2. Check if the goal is already a known fact:

    - If yes → Success.
    - If no → Find rules whose conclusion matches the goal.
3. For each rule:
    - Check if all conditions in the rule can be proved (recursively apply backward chaining).
4. If all premises are proved, the goal is true.

Algorithm (Steps)

1. Initialize goal.
2. If goal in fact base:
    - Return true.
3. Else for each rule:
    - If conclusion matches goal:
        - Try to prove all conditions.
        - If success → add to fact base and return true.
4. Else → return false.

Example
Knowledge Base:

- Rule 1: IF A THEN B
- Rule 2: IF B THEN C
- Rule 3: IF C THEN D

Goal:

- Prove D

Backward Chaining:

- Goal D → look for rule with conclusion D → Rule 3
- Need to prove C → Rule 2
- Need to prove B → Rule 1
- Need to prove A → Fact base has A ✓
- So A → B → C → D

  Goal D achieved**.**

Applications

- Diagnosis systems (medical, fault diagnosis)
- Problem-solving in expert systems
- Game playing and planning
- Intelligent tutoring systems

Comparison Between Forward and Backward Chaining

Aspect	Forward Chaining	Backward Chaining
Type	Data-driven	Goal-driven
Direction of Reasoning	From facts to conclusion	From goal to facts
Best suited for	Situations with many facts and few goals	Situations with few goals and many rules
Execution	Applies rules on existing facts	Searches rules to prove a specific goal
Speed	May derive many irrelevant facts	Focused reasoning, more efficient for single goal
Example use	Real-time monitoring, expert systems	Diagnostic systems, goal checking

Advantages

Forward Chaining:

- Good for situations where **all data are known** in advance.
- Useful for **automated monitoring** and **prediction**.
- Simple and systematic reasoning process.

Backward Chaining:

- Efficient for **goal-directed problem solving**.
- Avoids unnecessary rule evaluation.
- Works well when **goal is clear** but data are incomplete.

Drawback

- May be inefficient in large knowledge bases.
- Difficult to handle uncertain or incomplete knowledge directly.
- Requires **accurate and complete rules** for reliable results.
- Can get stuck in loops without proper control strategies.

Applications of Forward & Backward Chaining in AI

- Expert systems for medical and technical diagnosis.
- Decision support systems.
- Planning systems in robotics and automation.
- Knowledge-based systems in finance, security, and healthcare.
- Natural language understanding and theorem proving.

Forward Chaining starts from facts and moves toward conclusions (data-driven).

- Backward Chaining starts from goals and works backward to find supporting facts (goal-driven).
- Both are key reasoning strategies in rule-based AI systems and form the backbone of many intelligent agents.

Expert Systems:

Introduction to Expert Systems

- Expert System is an AI-based computer application that uses knowledge and inference procedures to solve complex problems that usually require human expertise.
- It mimics the decision-making abilities of a human expert in a particular domain (such as medicine, engineering, finance, etc.).

Example:

- MYCIN – for medical diagnosis of bacterial infections.
- DENDRAL – for chemical analysis.
- XCON – for configuring computer systems.

Definition

"An **Expert System** is a computer system that emulates the decision-making ability of a human expert by reasoning through bodies of knowledge, represented mainly as **if–then rules**."

- Unlike conventional programs, expert systems **reason** about data rather than just compute it.

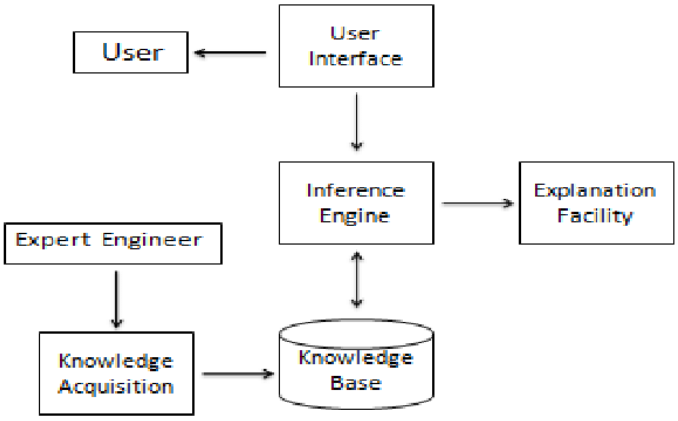
Characteristics of Expert Systems

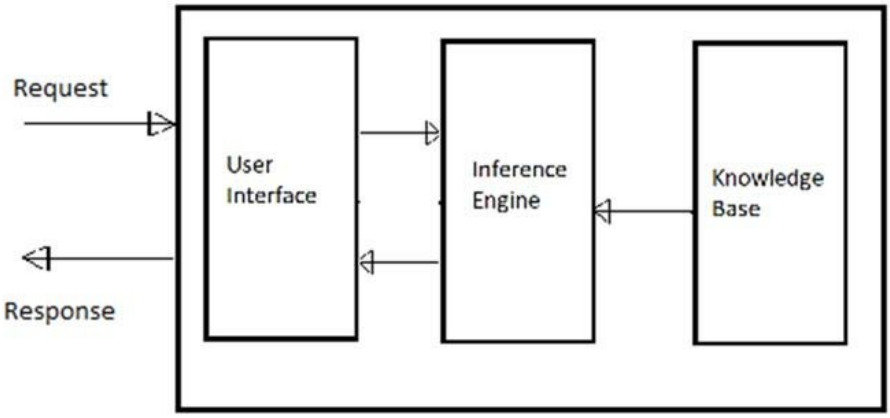
- High performance in specialized areas.
- Knowledge-based reasoning.
- Explanation capability – can explain how a conclusion was reached.
- Rule-based decision-making (IF–THEN rules).
- Ability to handle uncertainty (to some extent).
- User-friendly interfaces.

Examples of Expert Systems

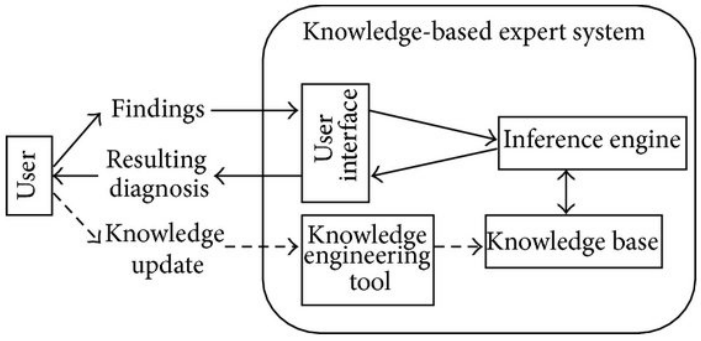
Domain	Expert System Example	Purpose
Medicine	MYCIN	Diagnosis of infectious diseases
Chemistry	DENDRAL	Identifying molecular structures
Computer Systems	XCON	Configuring VAX computer systems
Finance	PROSPECTOR	Mineral exploration and financial analysis
Education	Intelligent tutoring systems	Personalized learning support

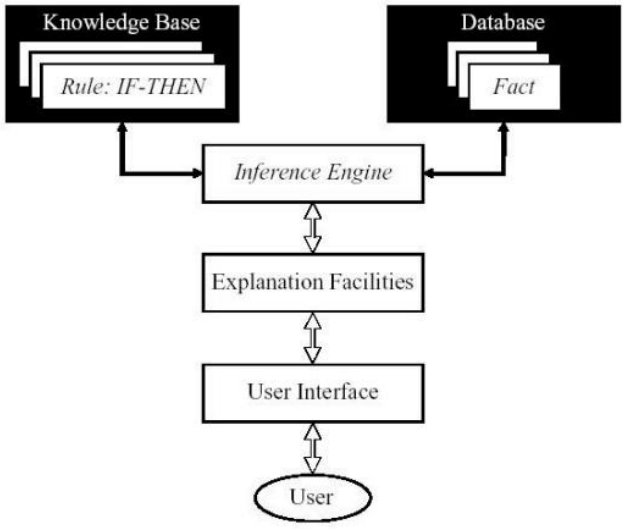
Architecture of an Expert System

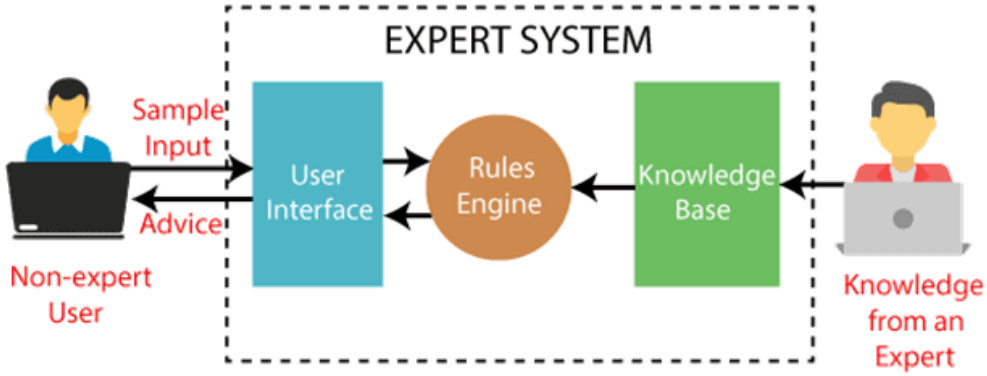




Typical Structure of Expert system







The basic structure of an expert system consists of the following components:

a) Knowledge Base

- The core of the expert system**.**
- Contains domain-specific facts and rules (e.g., IF–THEN rules).
- Example Rule:

  IF patient has fever AND sore throat THEN possible infection is flu.

b) Inference Engine

- The "brain" of the expert system.
- Uses reasoning techniques to apply rules to known facts to derive new facts or conclusions.
- Uses methods like:
  - Forward chaining (data-driven)
  - Backward chaining (goal-driven)

c) Knowledge Acquisition Module

- Helps in extracting knowledge from domain experts and converting it into rules for the knowledge base.

d) User Interface

- Allows interaction between the user and the system.
- Users can input facts or queries and receive explanations or solutions.

e) Explanation Facility

- Explains how the system arrived at a conclusion or why a particular question was asked.
- Increases transparency and trust.

f) Knowledge Refining / Learning Component

- Helps the system improve and update its knowledge base over time.

Design of Expert systems

The design process of an expert system involves several well-defined stages:

Step 1: Problem Identification
- Identify and define the domain where expertise is required.
- Example: medical diagnosis, mineral exploration, financial analysis.

Step 2: Knowledge Acquisition
- Gather knowledge from human experts, books, research papers, and databases.
- Techniques used:
    - Interviews
    - Observation
    - Protocol analysis
    - Machine learning (in modern systems)

Step 3: Knowledge Representation
- Organize knowledge in a structured way suitable for reasoning.
- Common techniques:
    - Rule-based representation
    - Semantic networks
    - Frames
    - Logic (FOPL)

Step 4: Development of Inference Engine
- Choose reasoning strategies:
    - Forward chaining (from facts to conclusions)
    - Backward chaining (from goal to facts)

    - Hybrid approaches
- Build mechanisms to handle uncertainty and conflicting rules**.**

Step 5: User Interface Design
- Create an interface for non-technical users.
- Ensure the system can ask and answer questions in natural language.

Step 6: Testing and Validation
- Test the system with real-world cases.
- Validate its reasoning accuracy with domain experts.

Step 7: Maintenance and Updating
- Update the knowledge base as new information becomes available.
- Improve the inference strategies for better performance.

Advantages of Expert Systems

- Consistency in decision-making.
- Faster problem solving than human experts.
- Availability 24×7 — unlike human experts.
- Can store and reuse expertise.
- Useful in training and education**.**

Limitations of Expert Systems

- Cannot reason beyond its knowledge base**.**
- Difficult and expensive to acquire expert knowledge**.**
- Lacks common sense and intuition.
- Poor at handling ambiguous **or** uncertain data without probabilistic support.
- Maintenance can be challenging.

Applications of Expert Systems in AI

Field	**Application Example**
Medicine	Diagnosis, prescription, clinical decision support
Engineering	Fault diagnosis, system control
Finance	Credit evaluation, stock prediction
Education	Intelligent tutoring, adaptive learning
Agriculture	Crop disease identification, pest control
Robotics	Intelligent control systems

Module III

Introduction MLP

Introduction to Machine Learning (ML)

- **Machine Learning (ML)** is a **subset of Artificial Intelligence (AI)** that enables computers to **learn automatically from experience (data)** without being explicitly programmed.
- In ML, systems **identify patterns**, **make decisions**, and **improve performance** as they are exposed to more data.

Definition (Tom M. Mitchell, 1997):
"A computer program is said to learn from experience **E** with respect to some class of tasks **T** and performance measure **P**, if its performance at tasks in **T**, as measured by **P**, improves with experience **E**."

🔻 Example:

- **Task (T):** Predicting whether an email is spam or not.
- **Experience (E):** Training data of labeled emails.
- **Performance (P):** Accuracy in predicting new emails correctly.

Introduction to MLP (Multi-Layer Perceptron)

- **MLP (Multi-Layer Perceptron)** is a class of **artificial neural networks (ANNs)** that consists of multiple layers of neurons — an **input layer**, one or more **hidden layers**, and an **output layer**.
- It is widely used for **pattern recognition**, **classification**, and **function approximation**.

Structure of MLP:

1. **Input Layer:** Accepts input features (e.g., pixels, data values).
2. **Hidden Layers:** Perform nonlinear transformations using activation functions.
3. **Output Layer:** Produces the final prediction or decision.

Mathematical Representation:

Each neuron performs:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

Where:

- xix_ixi: Input values
- wiw_iwi: Weights
- bbb: Bias
- fff: Activation function (e.g., Sigmoid, ReLU, Tanh)

Training:

- MLPs are trained using backpropagation, a supervised learning algorithm that minimizes the error between actual and predicted output.

Type of Human Learning

Before understanding how machines learn, it's useful to see how humans learn:

Type of Human Learning	Description	Example
Habituation	Learning to ignore repeated, irrelevant stimuli	Ignoring a ticking clock
Classical Conditioning	Learning by association between stimuli	Pavlov's dog salivates at sound of bell
Operant Conditioning	Learning by rewards and punishments	Student studies to get good grades
Observational Learning	Learning by observing others	Learning to cook by watching
Cognitive Learning	Using reasoning, thinking, and understanding	Solving puzzles logically

Machine Learning mimics these human learning styles through algorithms that **adapt**, **learn patterns**, and **improve decisions**.

Type of Machine Learning:

Machine Learning techniques are broadly classified into three main types:

1. Supervised Learning
2. Unsupervised Learning
3. Reinforcement Learning

Supervised Learning

In supervised learning, the model is trained on a labeled dataset, meaning both input and correct output are provided. The goal is to learn a mapping from input to output.

Process:

1. Provide input data (features) and output labels.
2. The algorithm learns from examples.
3. Use the trained model to predict output for unseen data.

Examples:

- Predicting house prices based on area and location.
- Classifying emails as spam or not spam.

Common Algorithms:

- Linear Regression
- Logistic Regression
- Decision Trees
- Support Vector Machines (SVM)
- k-Nearest Neighbors (kNN)
- Neural Networks (MLP)

Unsupervised Learning

In unsupervised learning, the data has no output labels. The system tries to discover hidden structures, patterns, or groupings in the input data.

Process:

1. Provide raw data without labels.
2. The algorithm finds patterns or clusters in the data.

Examples:

- Grouping customers with similar buying behavior.
- Market segmentation.
- Topic modeling of documents.

Common Algorithms:

- k-Means Clustering
- Hierarchical Clustering
- Principal Component Analysis (PCA)
- Autoencoders

Reinforcement Learning

Reinforcement Learning (RL) is based on the idea of learning through interaction with an environment. The agent learns to take actions that maximize reward and minimize penalty over time.

Process:

1. Agent observes the current state of the environment.
2. Takes an action.
3. Receives a reward or penalty.
4. Updates its strategy (policy) to maximize future rewards.

Examples:

- Self-driving cars (learning to navigate roads).
- Game playing (e.g., Chess, Go, Atari).

- Robotics (path planning, control).

Common Algorithms:
- Q-Learning
- Deep Q-Networks (DQN)
- Policy Gradient Methods

Comparison of Learning

Feature	**Supervised Learning**	**Unsupervised Learning**	**Reinforcement Learning**
Input data	Labeled	Unlabeled	State/action/reward
Goal	Predict output	Find structure	Maximize cumulative reward
Feedback	Provided	None	Reward or penalty
Example	Spam detection	Customer segmentation	Game-playing agent
Human supervision	High	Low	Low

General Model of Learning Agents

A Learning Agent is an intelligent system that improves its performance through experience.
Components of a Learning Agent:

1. Learning Element
    - Responsible for improving the agent's performance based on experience.
    - Adjusts the knowledge or rules.
2. Performance Element
    - Responsible for selecting actions based on the current knowledge.
    - It acts within the environment.
3. Critic
    - Evaluates the actions taken by the performance element and provides feedback.
    - Determines whether the outcome was good or bad.
4. Problem Generator
    - Suggests new exploratory actions to improve future performance.
    - Encourages learning by exploration.

Working of Learning Agent:

1. The Performance Element interacts with the environment and takes actions.
2. The Critic observes the outcomes and gives feedback.
3. The Learning Element uses this feedback to improve the internal model or rules.
4. The Problem Generator explores new strategies or actions.

Autonomous Driving Agent

Component	**Function**
Performance Element	Drives the car using current knowledge
Learning Element	Improves driving strategy using data
Critic	Evaluates driving quality (safety, smoothness)
Problem Generator	Tries new routes or technology

Module IV

Supervised Learning:

Definition:

> Supervised learning is a type of machine learning where the model is trained using labeled data. Input data X and corresponding output labels Y are provided. The goal is to learn a mapping function $f : X \rightarrow Y$

Example: Predicting house prices based on size, location, and other factors.

Steps:

1. Collect and prepare labeled data.
2. Split data into training and testing sets.
3. Train the model on the training set.
4. Test the model on unseen data to evaluate performance.

A. Holdout Method

The **Holdout Method** is a **simple and widely used technique** for evaluating the performance of a supervised learning model. In this method, the available dataset is **split into two (or sometimes three) subsets**:

1. **Training set** – used to train the model.
2. **Test set** – used to evaluate the model's performance.

Sometimes a third **validation set** is used to tune model parameters before final testing.

- **Concept:** Divide dataset into two parts — training and testing.
- **Typical split:** 70% training, 30% testing.
- **Advantage:** Simple and fast.
- **Limitation:** May give biased estimates if data is not evenly distributed.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total predictions}}$$

Purpose

- To **assess the generalization ability** of a model — how well it performs on **unseen data**.
- Helps prevent **overfitting**, where the model performs well on training data but poorly on new data.

Steps are-

1. **Split the dataset**: Typically 70%-30% or 80%-20% for training and testing.
2. **Train the model**: Use the training set to fit the algorithm.
3. **Test the model**: Evaluate performance on the test set using metrics like:
    - Accuracy (for classification)
    - Mean Squared Error (MSE) (for regression)
    - Precision, Recall, F1-score (for classification)

Example

- Suppose you have a dataset of 1000 patient records to predict disease risk.
- Split:
    - Training Set: 700 records → used to train the model
    - Test Set: 300 records → used to check accuracy
- Train a decision tree on the 700 records and then evaluate its predictions on the 300 test records.

Advantages

- Simple and fast to implement.
- Works well with large datasets**.**
- Requires no complicated computation.

Limitations

- Biased performance estimate**:** If the split is unlucky (not representative), evaluation may be misleading.
- Sensitive to dataset size: Works poorly on small datasets.
- No multiple validations**:** Unlike K-fold cross-validation, results can vary depending on the split.

K-fold cross-validation method

K-Fold Cross-Validation (CV**)** is a resampling technique used to evaluate the performance of a supervised learning model more reliably. Unlike the simple holdout method, which splits the data once, K-fold CV splits the dataset into K equal-sized "folds" and performs training and testing K times.

Purpose

- To assess the generalization ability of a model more accurately.
- Helps reduce overfitting and provides a more stable performance estimate.
- Particularly useful when datasets are small.

Steps are-

1. Split the dataset into K folds of roughly equal size.
2. Iterate K times:

- In each iteration, select one fold as the test set.
- Use the remaining K-1 folds as the training set.
- Train the model and evaluate performance on the test fold.

3. Aggregate results:
   - Compute the average performance metric (accuracy, MSE, F1-score, etc.) over all K iterations.

Example

Suppose you have 1000 samples and choose K = 5:

- Split the dataset into 5 folds of 200 samples each.
- Iteration 1: Train on folds 2–5, test on fold 1.
- Iteration 2: Train on folds 1, 3–5, test on fold 2.
- … continue until each fold has been used as a test set.
- Calculate the average accuracy over the 5 iterations.

Advantages

- Provides a more reliable estimate of model performance.
- Reduces bias from a single train-test split.
- Efficient use of data, especially important for small datasets.
- Works with most supervised learning algorithms.

Limitations

- Computationally expensive: Training is done K times.
- Choice of K affects the result:
  - Small K (e.g., 2-5) → faster but less reliable.
  - Large K (e.g., N, leave-one-out) → more reliable but very slow.
- Can still be affected by data imbalance if folds are not stratified.

Comparison with Holdout Method

Aspect	**Holdout Method**	**K-Fold Cross-Validation**
Data Usage	Only one train-test split	Every data point used for training and testing
Bias	Can be high (depends on split)	Lower bias, more reliable estimate
Computational Cost	Low	High (train K times)
Suitable For	Large datasets	Small to medium datasets

Boot strapping

- **Bootstrapping** is a **resampling technique** in statistics and machine learning where multiple **datasets are created by sampling with replacement** from the original dataset.
- It is primarily used to **estimate the accuracy, variance, or confidence intervals** of a model when the dataset is **small or limited**.

The term "bootstrap" comes from the phrase **"pulling oneself up by one's bootstraps"**, meaning generating results from the dataset itself without additional information.

Purpose

- To assess model performance when the original dataset is too small to split reliably.
- To reduce variability in performance estimates.
- To create ensemble models (like Bagging in Random Forests).

How Bootstrapping Works

1. Original dataset: Suppose you have NNN data points.
2. Sampling with replacement: Randomly select N data points from the dataset, allowing some points to appear multiple times.
3. Train the model on this new bootstrapped dataset.
4. Repeat the process multiple times (e.g., 100 or 1000 iterations).
5. Aggregate results: Average the predictions or calculate statistics (e.g., confidence intervals, standard errors).

Example

- Original dataset: [A,B,C,D,E]
- Bootstrapped sample 1: [B,C,B,E,A]
- Bootstrapped sample 2: [D,A,E,E,C]
- Train the model separately on each sample.
- Combine results to estimate performance or reduce variance.

Advantages

- Works well with small datasets.
- Provides robust estimates of model accuracy, variance, and bias.
- Forms the basis of ensemble methods like Bagging and Random Forest.
- Can generate confidence intervals for predictions.

Limitations

- Computationally expensive for large datasets or many iterations.
- May over-represent some data points and ignore others in a sample.
- Works best for independent and identically distributed (i.i.d.) data.

Applications

- Model evaluation (estimating accuracy, MSE, or error rates).
- Confidence interval estimation for predictions.
- Ensemble learning techniques like Bagging and Random Forests.
- Variance reduction in prediction models.

Simple-regression method

Simple Regression is a supervised learning technique used to predict a continuous numeric outcome based on one independent variable. It establishes **a** linear relationship between **the** input (independent variable X) and the output (dependent variable Y)**.**

Purpose

- To model the relationship between input and output.
- To predict future values of the dependent variable.
- To understand trends and correlations in data.

Mathematical Representation

The linear regression equation is:

$$Y = a + bX + \epsilon$$

Where:

- Y = Dependent variable (target to predict)
- X = Independent variable (predictor)
- a = Intercept (value of Y when X = 0)
- b = Slope (change in Y for a unit change in X)
- ϵ = Error term (difference between actual and predicted Y)

Steps are-

1. Collect data for X and Y.
2. Plot data to check for linear relationship.
3. Compute coefficients (a and b) using the least squares method:
    - Minimize the sum of squared errors (differences between actual and predicted Y).
4. Build regression model: Y=a+bX
5. Evaluate model performance using:
    - Mean Squared Error (MSE)
    - R-squared (R^2)value (explained variance)
6. Predicting **student exam score (Y)** based on **hours studied (X)**.

Data:

Hours Studied (X)	**Score (Y)**
2	50
4	60
6	70
8	80

Fit a line: Y=40+5X

Interpretation: Each additional hour studied increases score by 5 marks.

Advantages
- Simple and easy to interpret.
- Works well for linear relationships.
- Fast and requires less computational resources.

Limitations
- Only captures linear relationships; cannot model non-linear patterns.
- Sensitive to outliers which can distort the model.
- Assumes independence of errors and constant variance (homoscedasticity).

Applications
- Predicting sales based on advertising expenditure.
- Predicting crop yield based on rainfall.
- Predicting house prices based on area.
- Forecasting demand, revenue, or performance trends.

Unsupervised Learning:

- Unsupervised Learning is a type of machine learning where the algorithm learns patterns, structures, or relationships from unlabeled data.
- There is no target output provided; the model discovers hidden insights by itself.

Goal: Find structure or regularities in the data.

Applications
- Market segmentation
- Customer behavior analysis
- Pattern recognition
- Anomaly detection

Unsupervised Learning Methods

A. Clustering

- Clustering is the process of grouping similar data points together based on similarity or distance measures.
- Each group is called a cluster.

Features
- Similar points → same cluster
- Dissimilar points → different clusters
- Distance measures: Euclidean, Manhattan, Cosine similarity

Popular Algorithms

1. K-Means Clustering – Divides data into K clusters by minimizing variance within clusters.
2. Hierarchical Clustering – Builds a tree of clusters (dendrogram).
3. DBSCAN – Density-based clustering, identifies clusters of arbitrary shape.

Example

- Customer segmentation in marketing: Group customers into clusters based on buying patterns.
- Students clustered based on exam scores or study habits.

B. **Association**

- Association learning discovers rules or relationships between variables in large datasets.
- Often used in market basket analysis to find co-occurring items.

Key Concepts

- Support – Frequency of occurrence of an item or item set.
- Confidence – Likelihood of occurrence of one item given another.
- Lift – Strength of association beyond chance.

Popular Algorithm

- Apriori Algorithm – Finds frequent item sets and generates association rules.

Example

- "If a customer buys bread, they are likely to buy butter."
- Used in retail sales optimization and recommendation systems.

Reinforcement learning model.

- Reinforcement Learning is a goal-directed learning paradigm where an agent interacts with an environment and learns to make optimal decisions by trial and error.
- The agent receives rewards or penalties based on its actions.

Key difference from unsupervised learning: RL uses feedback (reward), whereas unsupervised learning does not.

Components of RL

1. **Agent** – The learner or decision maker.
2. **Environment** – The world in which the agent operates.
3. **State (s)** – Current situation of the agent.
4. **Action (a)** – Choice made by the agent.
5. **Reward (r)** – Feedback received after action.
6. **Policy (π)** – Strategy to select actions based on states.

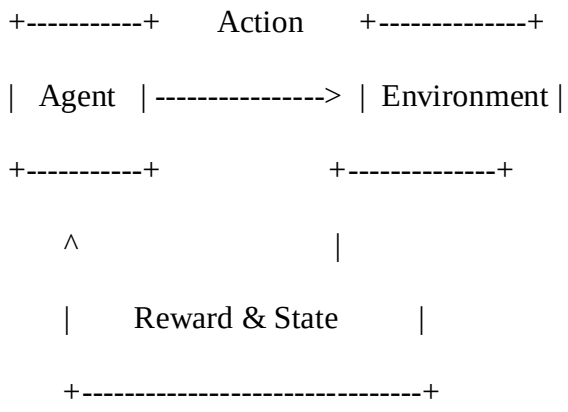
7. **Value Function (V)** – Expected cumulative reward from a state.

How RL Works

1. The agent observes the current state of the environment.
2. Selects an action based on its policy.
3. Receives a reward from the environment.
4. Updates the policy to maximize future rewards.
5. Repeats until an optimal strategy is learned.

Example

- **Game playing AI:** Agent learns to play chess or Go by receiving rewards for winning and penalties for losing.
- **Robotics:** Robot learns optimal path to reach a destination while avoiding obstacles.
- **Autonomous vehicles:** Learn driving behavior to maximize safety and efficiency.



Learning Type	Data Type	Goal	Example Applications
Clustering	Unlabeled	Group similar data points	Customer segmentation, pattern recognition
Association	Unlabeled	Discover rules and relationships	Market basket analysis, recommendations
Reinforcement Learning	Interactive	Learn optimal actions via rewards/penalties	Robotics, game playing, autonomous driving