UNIT 1: Introduction to Artificial Intelligence (AI)

1. Concept of AI

Definition of AI

Artificial Intelligence (AI) refers to the ability of machines to perform tasks that typically require human intelligence. It involves **learning**, **reasoning**, **problem-solving**, **perception**, **decision-making**, and **language understanding**. AI enables computers to **simulate human thinking and automate complex processes**.

Definitions by Experts:

- **John McCarthy (1956):** "AI is the science and engineering of making intelligent machines."
- **Alan Turing:** "AI refers to machines that can mimic human intelligence and pass the Turing Test."
- **Norvig & Russell:** "AI is the study of agents that receive precepts from the environment and take actions that maximize their chances of achieving their goals."

Characteristics of AI

1. **Problem-Solving Ability:** AI can **analyse data, recognize patterns, and solve complex problems** faster than humans.
2. **Automation:** AI systems automate **repetitive or time-consuming** tasks (e.g., robots in manufacturing).
3. **Adaptability & Learning:** AI learns from experience and improves over time (e.g., Netflix recommendations).
4. **Reasoning & Decision-Making:** AI **evaluates multiple possibilities** before selecting the best action (e.g., self-driving cars).
5. **Perception & Interaction:** AI interacts with the real world through sensors (e.g., speech recognition, image recognition).
6. **Speed & Efficiency:** AI can **process vast amounts of data** quickly and accurately.
7. **Scalability:** AI can handle **huge datasets** and adapt to different scenarios (e.g., fraud detection in banking).

Branches of AI

AI is classified into various domains based on its capabilities and applications:

1. **Machine Learning (ML):**

- Enables computers to learn from data and make predictions without explicit programming.
- **Example:** Spam filters in email services.

2. **Deep Learning (DL):**
   - Uses artificial neural networks (ANNs) to process complex patterns in data.
   - **Example:** AI in self-driving cars recognizing pedestrians.
3. **Natural Language Processing (NLP):**
   - AI understands and processes human language.
   - **Example:** Chatbots, Google Translate.
4. **Computer Vision:**
   - AI interprets and analyses images or videos.
   - **Example:** Face detection in security cameras.
5. **Robotics:**
   - AI controls robots for automation.
   - **Example:** AI-driven robotic arms in industries.
6. **Expert Systems:**
   - AI mimics human expert decision-making.
   - **Example:** AI diagnosing diseases in medical systems.
7. **Fuzzy Logic:**
   - AI handles uncertain or imprecise data.
   - **Example:** AI in washing machines adjusting water levels based on load.

2. History of AI

Era	Milestone
1950s	Alan Turing proposed the **Turing Test** to evaluate AI intelligence.
1956	John McCarthy coined the term **Artificial Intelligence** at the Dartmouth Conference.
1960s	AI programs like **ELIZA** (the first chatbot) were developed.
1980s	Expert systems became popular in industries.
1997	IBM's **Deep Blue** defeated world chess champion Garry Kasparov.

2010s	AI-powered **virtual assistants (Siri, Alexa, Google Assistant)** have become common.
2020s	AI is widely used in **self-driving cars, healthcare, automation, and deep learning applications**.

3. Current Status of AI

Artificial Intelligence (AI) is currently transforming various industries, making processes more efficient and improving decision-making. AI-powered systems can analyse large amounts of data, recognize patterns, and automate tasks, helping businesses and individuals in numerous ways.

Industries Where AI is Used Today

1. **Healthcare**
   - AI helps in **disease diagnosis**, medical imaging, and drug discovery.
   - Machine learning models analyse patient data to detect diseases like **cancer and COVID-19** at an early stage.
   - AI-powered **chatbots and virtual assistants** assist doctors and patients with medical inquiries.
   - **Example:** IBM Watson helps doctors in diagnosing diseases by analysing medical records.
2. **Finance**
   - AI plays a major role in **fraud detection** by identifying suspicious transactions in real time.
   - It is used in **algorithmic trading**, where AI predicts market trends and executes trades.
   - AI-driven **chatbots** assist customers in banking services.
   - **Example:** PayPal uses AI to detect fraudulent activities in online transactions.
3. **Education**
   - AI enables **personalized learning**, adapting lessons based on student performance.
   - AI-powered **grading systems** evaluate exams and assignments automatically.
   - Virtual AI tutors assist students by answering their questions.
   - **Example:** Duolingo uses AI to personalize language-learning experiences.
4. **Retail**

- AI-driven **recommendation systems** suggest products based on user preferences.
- AI is used in **customer service chatbots** to handle inquiries
- AI-powered **inventory management** predicts stock levels and demand.
- **Example:** Amazon and Netflix use AI to recommend products and movies based on user behaviour.

5. **Transportation**
    - AI is improving self-driving cars and **autonomous vehicles**.
    - AI-based **traffic optimization systems** help reduce congestion and accidents.
    - AI is used in **predictive maintenance** for vehicles, reducing breakdowns.
    - **Example:** Tesla's Autopilot system assists in self-driving capabilities.

4. Scope of AI

AI has a vast future, with increasing applications across multiple fields. The advancements in AI will continue to impact our lives in significant ways.

1. Automation

- AI is being used to automate **manufacturing, logistics, and customer service**.
- AI-powered robots are used in industries for assembling products and performing repetitive tasks.
- AI chatbots handle customer queries, reducing the need for human intervention.
- **Example:** Amazon uses AI-powered robots in warehouses to manage inventory and packaging.

2. Medical Research

- AI helps in **drug discovery**, speeding up the process of finding cures for diseases.
- AI models analyse vast medical data to **predict diseases** before symptoms appear.
- AI is used in **robot-assisted surgeries**, improving precision and reducing human errors.
- **Example:** Google's DeepMind developed an AI that can detect eye diseases from retina scans.

3. Smart Cities

- AI optimizes **traffic management**, reducing congestion using real-time data.
- AI-powered **waste management systems** improve recycling and garbage collection efficiency.
- AI helps in **energy conservation**, reducing electricity wastage in buildings.
- **Example:** AI-based traffic lights adjust signals in real time to manage traffic efficiently.

4. Cybersecurity

- AI detects **cyber threats** and prevents hacking attempts in real-time.
- AI-powered **fraud detection systems** identify unauthorized transactions.
- AI helps companies improve **network security** by identifying weak points.
- **Example:** AI in antivirus software detects malware and suspicious activity automatically.

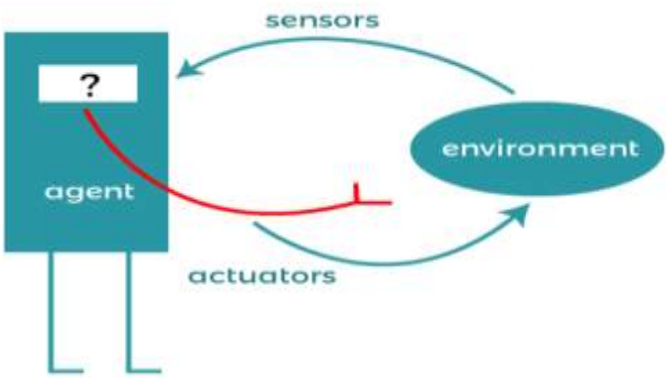
5. Gaming & Entertainment

- AI creates **realistic gaming environments** and intelligent NPCs (non-playable characters).
- AI helps in **film editing, animation, and video effects** automation.
- AI is used in **music and content creation**, generating realistic voices and soundtracks.
- **Example:** AI-powered deepfake technology can create realistic virtual actors in movies.

5. AI Agents

An **AI Agent** is a system that interacts with its surroundings through **sensors** (to collect information) and **actuators** (to take actions) to achieve specific goals. AI agents are used in various domains like robotics, gaming, self-driving cars, and virtual assistants.

Components of an AI Agent



1. Sensors- Sensors collect raw data from the environment, allowing the AI agent to perceive its surroundings. The type of sensor used depends on the agent's purpose.

Functions:

- Gather real-time input.
- Convert physical signals into digital data.

Examples: A self-driving car uses cameras to recognize traffic signs, GPS to navigate, and LIDAR to detect nearby objects.

2. Actuators - Actuators are responsible for carrying out actions based on the AI agent's decisions. They convert digital commands into real-world movements or responses.

Functions:

- Execute physical or virtual actions.
- Enable interaction with the environment.

Examples: A robotic vacuum cleaner uses actuators to move around a room, avoiding obstacles while cleaning.

3. Perception Unit - The perception unit processes data collected by sensors and extracts meaningful information for decision-making.

Functions:

- Recognizes patterns, objects, and speech.
- Converts raw data into structured information.

Examples: A self-driving car's perception unit identifies a red traffic light, signalling the car to stop.

4. Reasoning Engine - The reasoning engine acts as the AI agent's brain. It analyses data and decides the best action based on predefined rules or AI models.

Functions:

- Processes information and makes logical decisions.
- Optimizes actions for specific goals.

Examples: A chess-playing AI analyses board positions and selects the most strategic move.

5. Learning Unit - The learning unit allows AI agents to improve over time by adapting based on experience and data analysis.

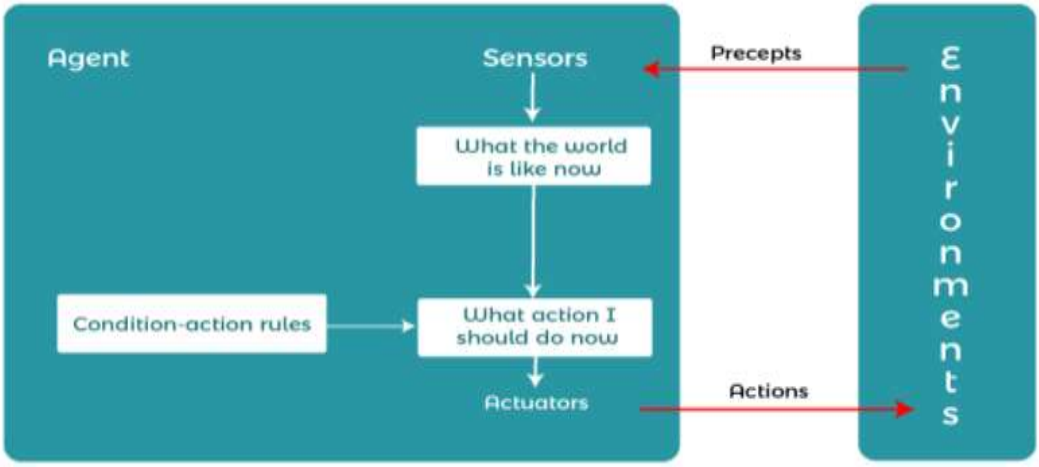
Functions:

- Uses Machine Learning (ML) to enhance performance.
- Adjusts decision-making based on feedback.

Examples: A spam filter initially lets some spam emails pass, but after user feedback, it learns to block similar emails more effectively.

Types of AI Agents

1. Simple Reflex Agents - These agents operate on predefined **"if-then" rules** and react purely to **current** conditions. They do not have memory and cannot learn from past experiences.



How They Work:

- Observe the current environment.
- Follow pre-programmed rules.
- Take action without considering past events.

Example Use Cases:

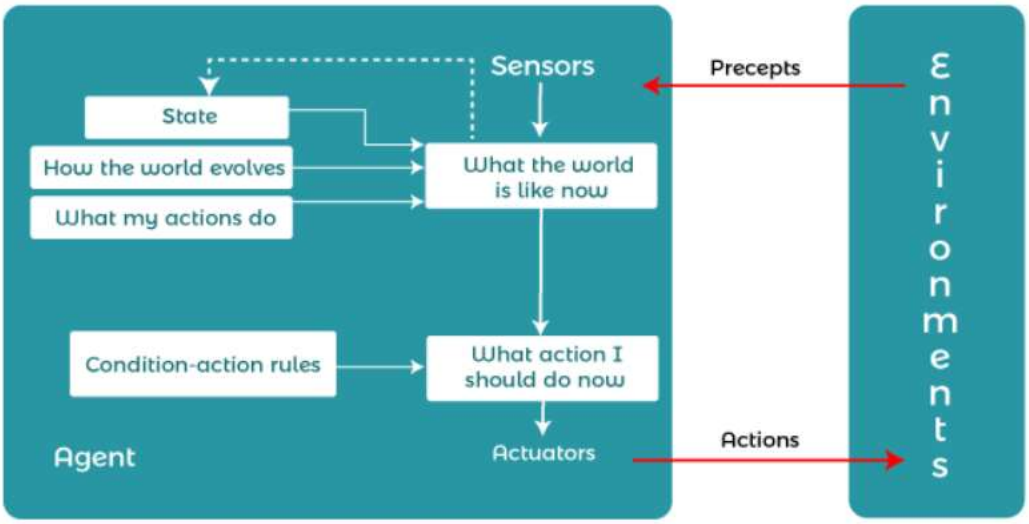
- **Thermostat** – Activates the heater if the temperature drops below a certain level.
- **Motion-Sensing Lights** – Turn on when motion is detected.

Limitation: These agents struggle in dynamic environments because they cannot adapt or learn.

2. Model-Based Reflex Agents - These agents are an improvement over simple reflex agents because they **store past information** and **maintain an internal model of the world** to make better decisions.

How They Work:

- Observe the environment and recall past experiences.
- Use a simplified "mental model" of how the world works.
- Make decisions based on both current and past data.

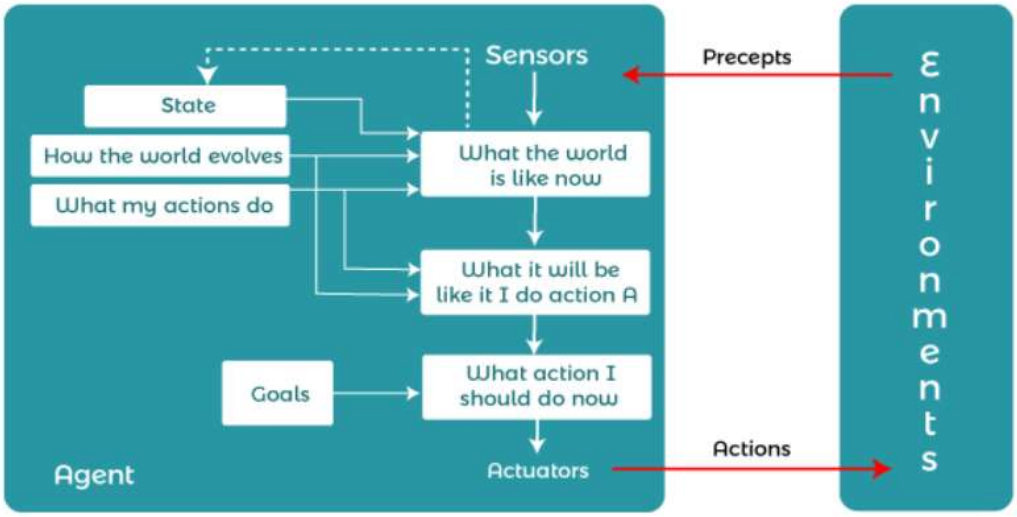


Example Use Cases:

- **Self-Driving Cars** – Remember road conditions, traffic signals, and obstacles.
- **AI-powered Robotic Arms** – Adjust their grip based on past handling experiences.

Advantage: These agents handle more complex tasks because they can adapt based on stored information.

3. Goal-Based Agents - Unlike reflex agents that only react, goal-based agents **think ahead** and take actions that help them achieve a specific goal.



How They Work:

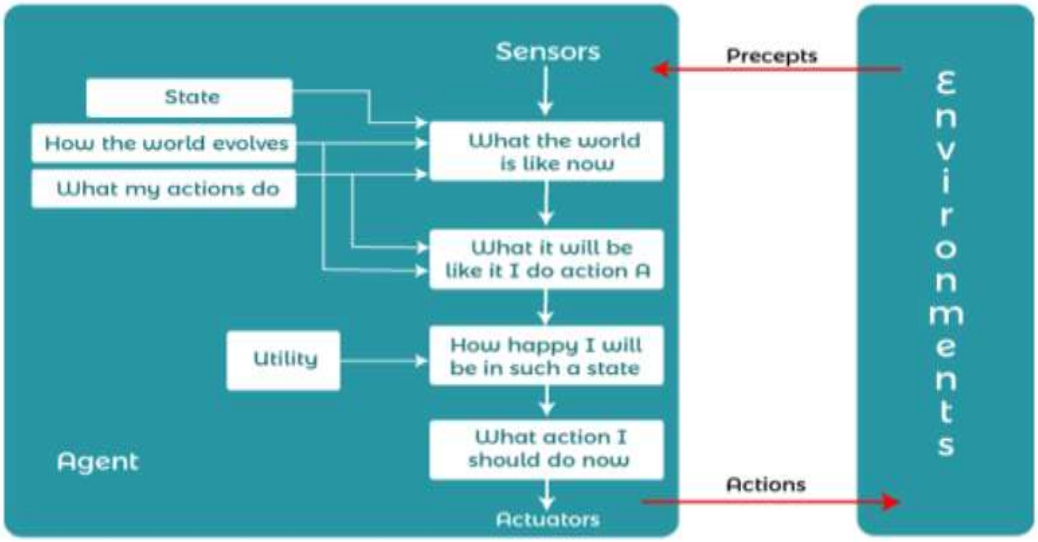
- Define a goal.
- Evaluate possible actions to reach the goal.
- Choose the best course of action.

Example Use Cases:

- **GPS Navigation Systems** – Find the fastest route to a destination.
- **Chess AI** – Plans multiple moves ahead to checkmate the opponent.

Advantage: These agents are more flexible and strategic in decision-making.

4. Utility-Based Agents - These agents go beyond just reaching a goal; they evaluate different options and choose the **best possible** action based on a **utility function** (a way to measure success).



How They Work:

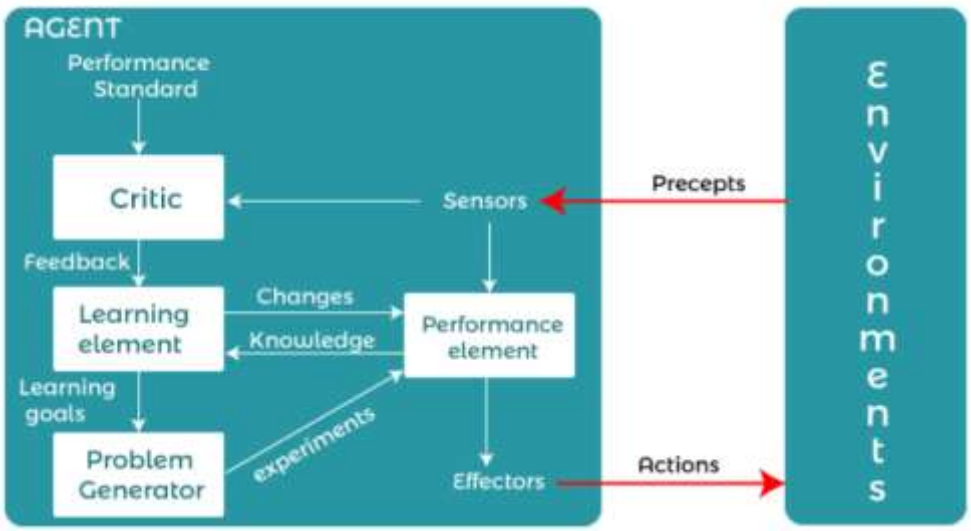
- Assign values (utilities) to different outcomes.
- Compare possible actions.
- Pick the one that maximizes success.

Example Use Cases:

- **Stock Market AI** – Analyses trends and picks the most profitable stocks.
- **Autonomous Drones** – Choose the safest and most efficient delivery route.

Advantage: They not only achieve goals but also optimize performance, making them more intelligent.

6. **Learning Agents -** These are the **most advanced AI agents** because they **learn from past experiences**, adapt to changes, and improve their decision-making using **Machine Learning (ML)** techniques.



How They Work:

- Gather experience from interactions.
- Use learning algorithms to improve future decisions.
- Adapt to new or unforeseen situations.

Example Use Cases:

- **Spam Filters** – Improve in detecting spam emails over time.
- **AI in Video Games** – Adjusts difficulty based on player performance.

Advantage: These agents can handle dynamic environments and continuously evolve for better performance.

6. AI Environments

An **AI environment** is the external world in which an **AI agent** operates. It determines how the agent perceives and interacts with its surroundings. AI environments vary based on different factors, such as observability, determinism, interaction with other agents, and changes over time. Understanding these environments helps in designing AI systems that function efficiently in real-world situations.

1. Fully Observable vs. Partially Observable Environments

Fully Observable Environment

- The AI agent has complete access to all relevant information about the environment at any given time.
- There is no need to store past information since the current state provides everything required for decision-making.

- These environments are easier to manage because the AI can make informed choices without uncertainty.

Examples:

- **Chess AI** – The entire chessboard, including the positions of all pieces, is visible to both players. The AI can calculate the best move based on complete knowledge.
- **Tic-Tac-Toe AI** – The AI knows the entire game board and can predict all possible moves.

Partially Observable Environment

- The AI does not have full access to the environment's current state.
- The agent must store past data or predict missing information to make decisions.
- These environments require AI to handle uncertainty and make decisions based on probability or estimation.

Examples:

- **Self-Driving Cars** – The AI does not always know what is around the corner or if there is an obstacle in a blind spot. It must rely on sensors and past observations.
- **Poker AI** – The AI does not know the opponent's cards and must predict their possible hands based on betting behaviour.

Key Difference: Fully observable environments provide all the necessary information for decision-making, while partially observable environments require the AI to estimate and predict missing data.

2. Deterministic vs. Stochastic Environments

Deterministic Environment

- The outcome of an action is fully predictable—there is no randomness.
- The same action in the same state always leads to the same result.
- AI can plan actions perfectly because there is no uncertainty.

Examples:

- **Chess** – The movement of a piece follows fixed rules. Given a specific board state, the AI can determine all possible outcomes.
- **Tic-Tac-Toe** – Each move follows a strict pattern, and the AI can predict the entire game tree.

Stochastic Environment

- The environment includes randomness, and the outcome of an action is uncertain.

- The AI must use probability and statistical models to predict outcomes.
- AI decision-making relies on handling uncertainty and making the best choices based on available data.

Examples:

- **Self-Driving Cars** – The movement of other vehicles and pedestrians is unpredictable, making the environment stochastic.
- **Stock Market Prediction** – AI cannot fully predict market trends due to external factors like economic changes and investor behaviour.

Key Difference: Deterministic environments have predictable outcomes, while stochastic environments introduce uncertainty and randomness.

3. Competitive vs. Collaborative Environments

Competitive Environment

- AI agents compete against each other to achieve their goals.
- The success of one agent often means failure for another.
- These environments require AI to develop strategies to outperform opponents.

Examples:

- **Chess AI** – Two AIs or an AI and a human player compete to checkmate the opponent's king.
- **Online Multiplayer Games** – AI bots compete with players to win matches.

Collaborative Environment

- Multiple AI agents work together to achieve a common goal.
- Cooperation and communication between agents are necessary for success.

Examples:

- **Self-Driving Cars in Traffic** – Cars communicate with each other to avoid collisions and ensure smooth traffic flow.
- **Robotics in Manufacturing** – Multiple robotic arms work together to assemble products efficiently.

Key Difference: Competitive environments involve rivalry between agents, whereas collaborative environments require agents to work together toward a shared objective.

4. Single-Agent vs. Multi-Agent Environments

Single-Agent Environment

- Only one AI agent interacts with the environment.

- The AI's actions do not affect other agents.

Examples:

- **Maze Solver** – An AI navigates a maze alone to find the exit.
- **Google Assistant** – The AI provides responses based on user input without interacting with other AI systems.

Multi-Agent Environment

- Multiple AI agents exist in the same environment, either competing or collaborating.
- Agents must account for the presence and actions of others.

Examples:

- **Football AI** – A team of AI players competes against another team, making real-time decisions.
- **Stock Market AI** – Multiple AI trading bots react to market conditions and influence each other's decisions.

Key Difference: Single-agent environments involve isolated AI decision-making, while multi-agent environments require AI to consider the actions of other agents.

5. Dynamic vs. Static Environments

Dynamic Environment

- The environment changes continuously, even if the AI does not take any action.
- AI must constantly adapt to new conditions.

Examples:

- **Self-Driving Cars** – Traffic conditions change in real-time, requiring constant adjustments.
- **Online Games** – The game world changes based on player actions and interactions.

Static Environment

- The environment remains unchanged unless acted upon by the AI.
- AI can take its time to analyse and make the best decision.

Examples:

- **Sudoku Solver** – The puzzle remains the same until the AI fills in numbers.
- **Board Games like Chess** – The game state only changes when a player makes a move.

Key Difference: Dynamic environments constantly evolve, whereas static environments remain unchanged unless acted upon.

6. Discrete vs. Continuous Environments

Discrete Environment

- The number of possible actions and states is **finite** and clearly defined.
- AI can evaluate all possible moves and outcomes.

Examples:

- **Chess** – The board has a fixed number of pieces and moves.
- **Tic-Tac-Toe** – There are only a limited number of positions and possible moves.

Continuous Environment

- The number of possible actions and states is **infinite** or extremely large.
- AI must deal with real-world complexities, such as movement, speed, and physics.

Examples:

- **Self-Driving Cars** – The car's position, speed, and steering angles have infinite possible values.
- **Robotics** – A robotic arm moving through space has countless possible positions.

Key Difference: Discrete environments have a limited set of actions, whereas continuous environments deal with infinite possibilities

7. PEAS Framework in AI

The **PEAS** framework is used in Artificial Intelligence (AI) to define an agent's interaction with its environment. It consists of four key components: **Performance Measure, Environment, Actuators, and Sensors**.

PEAS Representation in AI

The **PEAS** framework provides a structured way to design and evaluate intelligent agents by specifying how they interact with their surroundings.

- **Performance Measure:** Defines success criteria for the agent's actions.
- **Environment:** The surroundings in which the agent operates.
- **Actuators:** The parts that execute actions in the environment.
- **Sensors:** The components that collect information from the environment.

This framework ensures AI systems are **well-defined, effective, and capable of performing their tasks efficiently**.

P: Performance Measure

- A **performance measure** is a unit that defines the success of an agent.
- It helps the agent decide what actions to take to achieve the best outcome.
- The measure varies depending on the agent's task and goal.

Example:

- A self-driving car's performance measure includes **safety (avoiding accidents), efficiency (minimizing travel time), and comfort (smooth driving experience).**

E: Environment

- The **environment** is everything surrounding the agent.
- It **keeps changing** as the agent interacts with it.

Example:

- A **smart thermostat** operates in a home environment that includes **temperature changes, weather conditions, and the presence of people.**

A: Actuators

- **Actuators** are the components of an agent that perform actions.
- They **change** or **affect** the environment based on the agent's decisions.

Example:

- A **smart thermostat** has actuators like **heating and cooling systems, which adjust the temperature.**

S: Sensors

- **Sensors** help the agent **perceive** its environment.
- They **collect data** and send it to the agent for decision-making.

Example:

- A **smart thermostat** uses sensors like **temperature sensors, humidity sensors, and motion detectors to collect information about its surroundings.**

Importance of PEAS in AI

The **PEAS** framework helps in designing effective AI agents by:

- **Defining clear goals** – Ensures the agent works toward specific objectives.
- **Understanding the environment** – Helps the agent handle different situations.
- **Improving interactions** – Ensures the agent has the right sensors and actuators to function properly.

Example: PEAS Framework for a Self-Driving Car

PEAS Component	Details
Performance Measure	Safety, fuel efficiency, travel time, comfort, and following traffic rules.
Environment	Roads, traffic, pedestrians, weather conditions.
Actuators	Steering wheel, brakes, accelerator, indicators.
Sensors	Cameras, GPS, LiDAR, radar, ultrasonic sensors.

Advantages of PEAS in AI

- **Provides a structured design** – Helps in defining AI agents clearly.
- **Works for many AI systems** – Can be applied to robots, chatbots, and self-driving cars.
- **Ensures goal-oriented AI** – AI agents can focus on measurable objectives.
- **Helps in systematic development** – Makes AI development organized and efficient.

Disadvantages of PEAS in AI

- **Complex to implement** – Dynamic environments require extensive programming.
- **May oversimplify real-world problems** – Not all AI scenarios fit into the PEAS model.
- **Resource-intensive** – Needs significant computing power and sensors.
- **Limited adaptability** – Some AI agents struggle with unexpected changes.

8. Problem Formulation in AI

Problem formulation is the process of defining a problem in a structured way so that an AI system can analyse and find the best solution.

Key Steps in Problem Formulation

1. **Define the Initial State**
    - The starting condition before any action is taken.
    - **Example**: In a maze-solving AI, the initial state is the agent's starting position.
2. **Define the Goal State**
    - The final outcome the AI aims to achieve.

- **Example**: In chess, the goal state is achieving checkmate.

3. **Define the Set of Possible Actions**
   - The moves or operations AI can perform to transition between states.
   - **Example**: In a self-driving car, actions include turning, accelerating, or stopping.
4. **Define the State of Space**
   - All possible states that can be reached from the initial state.
   - **Example**: In tic-tac-toe, the state space includes all board configurations.
5. **Define the Cost Function**
   - A measure to evaluate the efficiency of a solution.
   - **Example**: In GPS navigation, the cost function could be the shortest travel time or least fuel consumption.

Examples of AI Problems

1. **Pathfinding**
   - **Problem**: Finding the shortest route from one location to another.
   - **AI Usage**: Google Maps finds the best route considering traffic and distance.
2. **Game Playing**
   - **Problem**: Choosing the best move in a strategic game.
   - **AI Usage**: Chess AI like AlphaZero uses Minimax and Alpha-Beta Pruning to make decisions.
3. **Puzzle Solving**
   - **Problem**: Solving logical puzzles like Sudoku or the 8-puzzle problem.
   - **AI Usage**: Uses search algorithms like A* to efficiently reach the goal.

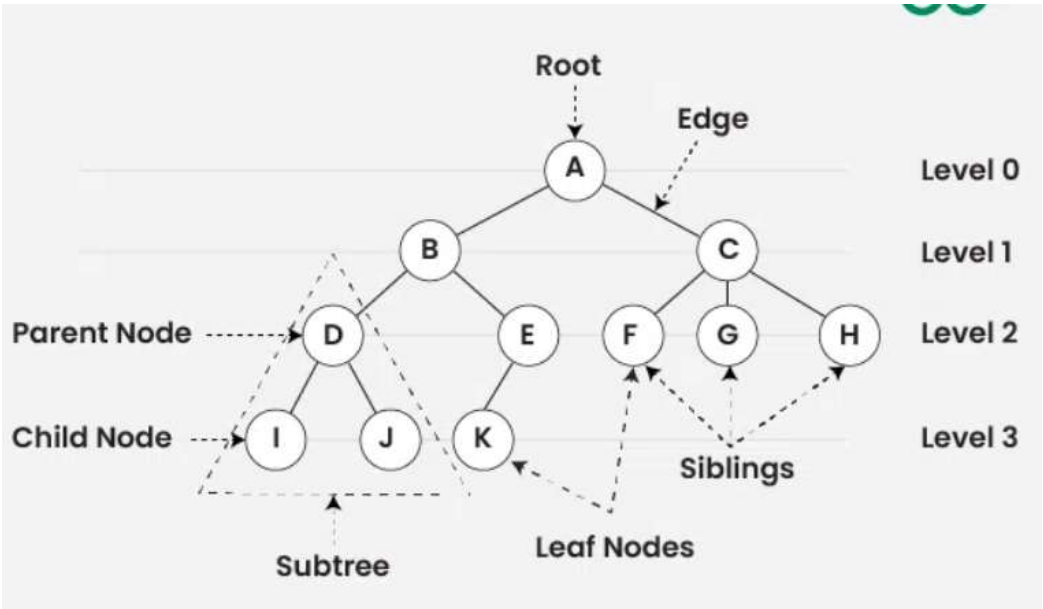
9. Review of Tree and Graph Structures

AI uses **tree and graph structures** for searching, decision-making, and optimization problems.

Tree Structure in AI - A tree data structure is a hierarchical structure used to represent data in a parent-child relationship and is widely used in AI for tasks such as search algorithms (e.g., decision trees, game trees), knowledge representation, and hierarchical clustering.

Basic Terminologies in Tree Data Structure

1. **Parent Node** – Immediate predecessor of a node (e.g., B is the parent of D, E).
2. **Child Node** – Immediate successor of a node (e.g., D, and E are children of B).
3. **Root Node** – The topmost node with no parent (e.g., A).
4. **Leaf Node** – Nodes with no children (e.g., I, J, K, F, G, H).
5. **Ancestor** – Any predecessor from the root to a node (e.g., A and B are ancestors of E).
6. **Descendant** – A node is a descendant of another node if the latter is its ancestor.
7. **Sibling** – Nodes with the same parent (e.g., D, and E are siblings).
8. **Level** – Number of edges from the root to a node (Root is at level 0).
9. **Internal Node** – A node with at least one child.
10. **Neighbour** – Parent or child nodes of a node.
11. **Subtree** – Any node and its descendants.



Properties of Trees

1. **Number of Edges** – A tree with **N nodes has (N-1) edges**.
2. **Depth of a Node** – Number of edges from the **root to the node**.
3. **Height of a Node** – Longest path from the node to a **leaf node**.
4. **Height of a Tree** – Longest path from the **root to a leaf node**.
5. **Degree of a Node** – Number of children a node has.
6. **Degree of a Tree** – Maximum degree of any node in the tree.

Graph Structure in AI - A graph consists of **nodes (vertices)** and **edges (connections)** between them. AI uses graphs for **pathfinding, recommendation systems, and network analysis**.

Key Components of a Graph:

1. **Vertices (Nodes)** – Represent possible states in a problem.
2. **Edges (Links)** – Show possible transitions between states.
3. **Directed vs. Undirected Graphs** – Directed graphs have edges with direction (one-way streets), while undirected graphs have bidirectional edges (two-way streets).

Example: AI in Pathfinding

- AI uses **graph search algorithms** like **Dijkstra's Algorithm** and *An Algorithm** to find the shortest path in a city or video game.
- **Example:** Google Maps represents locations as nodes and roads as edges, using AI to find the shortest route.

10. State Space Representation - State Space Representation is a fundamental concept in Artificial Intelligence (AI) where all possible states and actions of a problem are represented as a graph. It helps AI explore different possibilities to find an optimal solution. This concept is widely used in search algorithms, problem-solving, and decision-making in AI.

Key Components of State Space Representation

1. **States (Nodes)** – Represent different situations or configurations of the problem.
2. **Actions (Edges)** – Define the possible moves from one state to another.
3. **Initial State** – The starting configuration of the problem.
4. **Goal State** – The desired final state AI wants to reach.
5. **State Transition** – The process of moving from one state to another based on actions.

Example: 8-Puzzle Problem

The **8-puzzle** consists of a **3×3 grid** with **8 numbered tiles** and **one space**. By sliding the tiles into the empty space, the goal is to arrange them in a specific order.

State Space Representation of the 8-Puzzle

- **State Space**: All possible tile arrangements in the 3×3 grid.
- **Initial State**: The given tile arrangement at the start.
- **Goal State**: The correct tile arrangement AI aims to achieve.
- **Actions**: Moving tiles **left, right, up, or down** into the empty space.

- **State Transition**: Changing from one tile arrangement to another by acting.

How AI Uses State Space in 8-Puzzle?

- AI **searches through the state space** by applying actions.
- It follows a **search strategy** (e.g., Breadth-First Search, Depth-First Search, A* Algorithm) to find the best sequence of moves.
- The AI **stops when it reaches the goal state**.

11. Search Graph and Search Tree

1. Search Tree - A search tree is a hierarchical structure where each node represents a state, and edges represent actions leading to new states. It is generated dynamically as the search progresses.

Characteristics:

- Each state appears only once in the tree, even if it can be reached through different paths.
- No mechanism to check for repeated states, which may lead to redundant computations.
- Commonly used in problems where exploring every possibility is feasible.

Example:

- **Game Trees**: Used in chess, tic-tac-toe, and other strategic games where each move generates a new state.

Advantages:

- Simple to implement.
- Useful for small or structured problems.

Disadvantages:

- Can lead to unnecessary expansion of duplicate states.
- Inefficient for large problems due to exponential growth.

2. Search Graph - A search graph is a directed graph where nodes represent states, and edges represent transitions between states. Unlike search trees, graphs track visited states to avoid redundancy.

Characteristics:

- Keeps a record of explored states to prevent revisiting them.
- More memory-efficient than search trees as it avoids redundant paths.

- Commonly used in complex problems where multiple paths can lead to the same state.

Example:

- **Pathfinding Algorithms:** Used in GPS navigation (e.g., Google Maps) to find the shortest route between locations**.**

Advantages:

- Avoids unnecessary exploration of repeated states.
- More efficient for large state spaces.

Disadvantages:

- Requires additional memory to store visited states.
- More complex to implement than a search tree.

Feature	**Search Tree**	**Search Graph**
Structure	Hierarchical (tree-like)	Network-based (graph-like)
Redundant States	Yes	No
Cycle Handling	Does not track cycles	Avoids cycles by tracking visited states
Efficiency	Less efficient	More efficient
Memory Usage	Lower	Higher due to state tracking

12. Turing Test and Features Required for AI

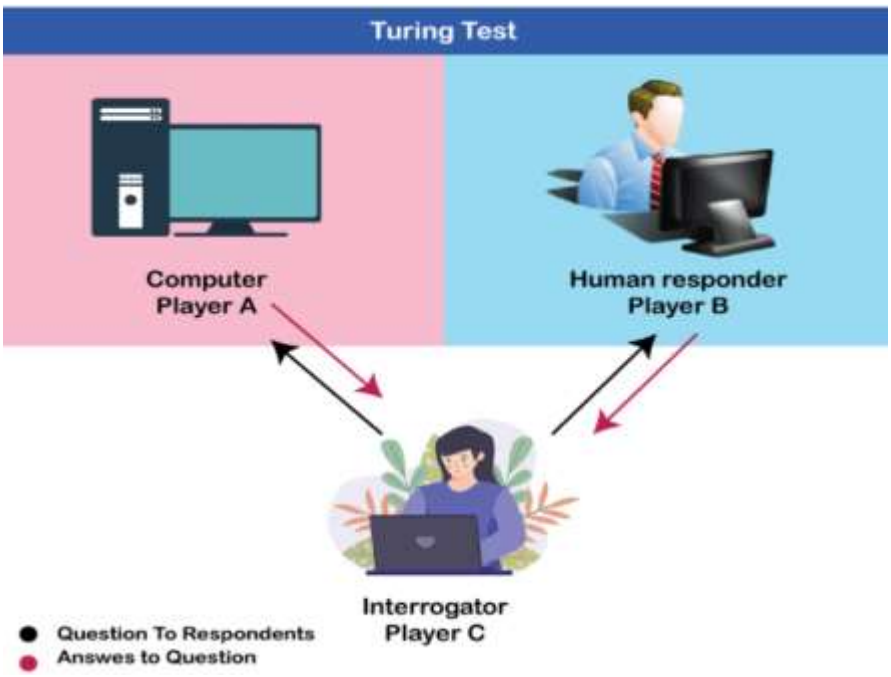
The **Turing Test** was introduced by **Alan Turing** in 1950 to determine whether a machine can exhibit human-like intelligence. In his paper *"Computing Machinery and Intelligence,"* Turing proposed that if a machine could mimic human responses convincingly, it could be considered intelligent.

How the Turing Test Works

The test is based on a modified version of the **"Imitation Game"** and involves three participants:

1. **Computer (Player A)** – The machine being tested.
2. **Human Responder (Player B)** – A real person.

3. **Human Interrogator (Player C)** – The evaluator who determines which participant is the machine.



- The interrogator asks questions to both the human and the computer.
- Responses are exchanged through a **keyboard and screen** to prevent speech-based identification.
- If the interrogator **cannot reliably distinguish** between the human and the machine, the computer is said to have passed the test.

Variations of the Turing Test

Several modified versions of the test have been proposed to improve its accuracy:

1. **Total Turing Test**
   - Assesses not only text-based conversations but also **visual and physical** interactions.
   - AI must recognize objects and respond to actions.
2. **Reverse Turing Test**
   - The **machine becomes the interrogator**, trying to differentiate between humans and other AIs.
3. **Multimodal Turing Test**
   - Tests AI's ability to handle **multiple forms of communication** (text, speech, images, etc.).

Requirements for a Machine to Pass the Turing Test

1. Natural Language Processing (NLP)

- The AI must understand and generate responses in human language (e.g., English, Hindi).
- It should process grammar, sentence structure, and meaning correctly.
- Example: If asked, *"What is your favourite movie?"*, the AI should respond in a natural way, like *"I like science fiction movies."*

2. Knowledge Representation

- The AI must store and retrieve factual information efficiently.
- It should remember previous interactions and provide context-aware responses.
- Example: If asked, *"Who is the President of the USA?"*, it should retrieve and provide the correct answer.

3. Automated Reasoning

- The AI should think logically and use stored knowledge to answer questions.
- It must deduce, infer, and analyse like a human.
- Example: If asked, *"If today is Monday, what will be the day after tomorrow?"*, it should logically conclude *"Wednesday."*

4. Machine Learning

- AI should improve over time by learning from conversations and experiences.
- It must detect patterns in human responses and adjust accordingly.
- Example: If a chatbot often gets asked about football, it should learn and improve its responses on that topic.

5. Vision (For Total Turing Test)

- The AI must process and understand images and objects shown by the interrogator.
- It should recognize facial expressions, hand gestures, or written text.
- Example: If shown a picture of a cat, the AI should identify and respond with, *"This is a cat."*

6. Motor Control (For Total Turing Test)

- The AI should be able to interact physically with objects if required.
- It must move, touch, or manipulate items as a human would.

- Example: If an interrogator asks the AI to *"pick up a ball,"* it should act using a robotic arm.

Limitations of the Turing Test

1. **Not a True Measure of Intelligence**
    - A machine can simulate responses **without real understanding** (as per the Chinese Room argument).
2. **Limited Scope**
    - The test mainly evaluates **text-based** conversations and may not fully assess **complex real-world interactions**.

UNIT 2: Search Algorithms

1. Relational Agent

A **Relational Agent** is an artificial intelligence system designed to build and maintain long-term **social relationships** with users. These agents interact with humans using **natural communication methods**, such as speech, gestures, and emotions, to create a more engaging and human-like interaction.

Key Features:

- **Social Interaction:** Engages in long-term conversations and builds relationships.
- **Personalization:** Adapts to users' behaviours and preferences.
- **Multimodal Communication:** Uses speech, facial expressions, and gestures.
- **Emotional Intelligence:** Detects and responds to user emotions.

2. Search Algorithm in AI

Search algorithms in AI help in finding optimal solutions to problems by simulating different scenarios and evaluating alternatives. These algorithms assist AI agents in navigating complex problems efficiently, ensuring that the best possible solution is identified.

How Search Algorithms Work in AI

- AI systems **process the initial state** and attempt to reach the expected goal state.
- AI agents **analyse multiple possible solutions** to determine the most effective path.
- The **efficiency of an AI system** depends on how well the search algorithm performs.

Role of AI Agents in Search Algorithms

- AI agents help make AI interfaces more accessible, even without advanced software knowledge.
- These agents execute actions step by step, evaluating different choices.
- The goal is to **develop an action plan** that leads to mission completion.
- By continuously assessing available alternatives, AI agents **identify the best possible approach** to a problem.

Search Algorithm's terminologies

1. Search - Searching is a step-by-step method used to find the best solution in a given search space. The process involves evaluating different possibilities and choosing the most effective path.

a. **Search Space**
   - The search space is the collection of all possible solutions that an AI system can consider.
   - It includes every possible way the problem can be solved.
   - A large search space makes finding a solution more difficult.
   - **Example:** In a chess game, the search space consists of all possible board positions.

b. **Initial State**
   - The initial state is the starting point of the search process.
   - The search begins from this state to find the desired solution.
   - **Example**: In a GPS, the initial state is the user's current location.

c. **Goal State**
   - The goal state is the final desired outcome or the solution the AI is looking for.
   - A goal test is used to check if the AI has reached the correct solution.
   - **Example:** In a maze-solving problem, the goal state is reaching the exit.

2. Search Tree - A search tree is a representation of the search problem, where each node represents a state. The root node is the initial state, and the branches represent possible actions leading to different states.

- **Example:** In a chess game, each possible move creates a new branch in the search tree.

3. Actions - Actions define all the possible moves an AI agent can take from a given state**.**

- **Example:** In a robot vacuum, actions include moving forward, turning left, or stopping.

4. Transition Model - The transition model describes how one state changes into another when an action is performed.

- **Example: I**n a self-driving car, turning left moves the car from its current lane to the left lane.

5. Path Cost - Path cost is a numeric value assigned to a path to measure how efficient it is. AI aims to find the path with the lowest cost.

- **Example:** In Google Maps, path cost is based on factors like distance, time, and traffic.

6. Solution - A solution is a sequence of actions that leads from the start state to the goal state.

- **Example:** In a game, solving a level means performing the right moves in the correct order.

7. Optimal Solution - An optimal solution is the best possible solution with the lowest cost among all available options.

- **Example:** Finding the shortest route in a GPS navigation system.

3. Properties of a Search Algorithm

A good search algorithm should efficiently find a solution while considering factors like **accuracy, speed, and memory usage**. The key properties of a search algorithm are:

1. Completeness

- A search algorithm is **complete** if it always finds a solution whenever one exists.
- If no solution is possible, it should confirm that as well.
- **Example**: A maze-solving algorithm is complete if it always finds a way out when an exit exists.

2. Optimality

- An algorithm is **optimal** if it finds the best possible solution with the lowest cost.
- It ensures the most efficient path rather than just any valid solution.
- **Example**: A GPS is optimal if it suggests the shortest route instead of a longer one.

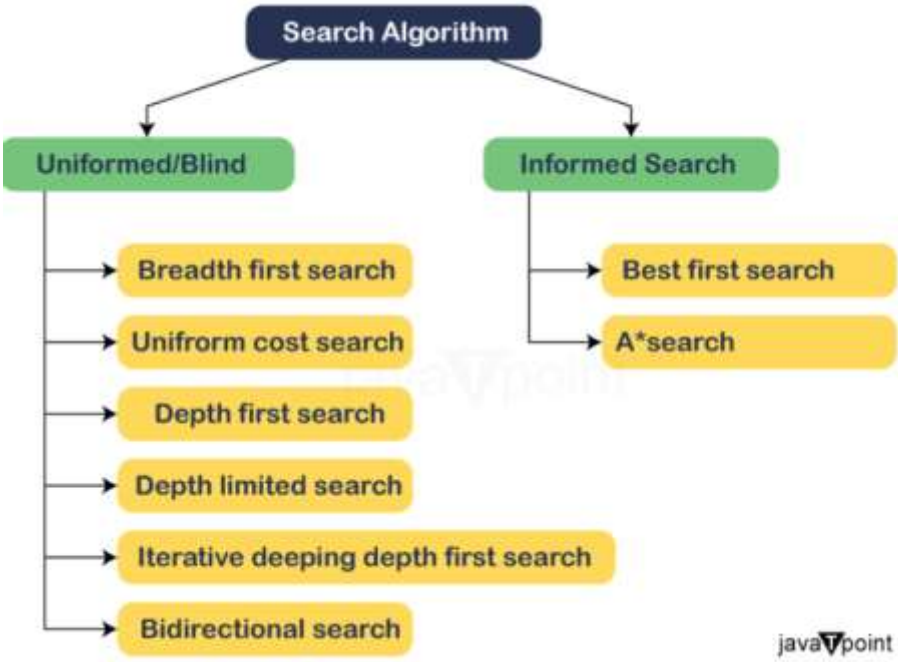
3. Time Complexity

- This measures **how long** the algorithm takes to complete the search.
- A good search algorithm should solve problems **quickly**, especially for large search spaces.
- **Example**: Searching for a name in a phone directory takes longer if the list is not sorted.

4. Space Complexity

- This refers to the **amount of memory** the algorithm uses while searching.
- Some search methods require storing multiple states, which increases memory use.
- **Example**: A game AI may store multiple possible moves, consuming more memory.

4. Search Algorithms and its types in AI



1. Uninformed Search (Blind Search)

An uninformed search does not have any prior knowledge about the search space. It works in a brute-force manner, examining each possible option until it finds the goal state. This method does not use any heuristics or problem-specific knowledge.

Key Features

- **Lacks domain knowledge**: It does not use additional information to guide the search.
- **Explores all possible paths**: It systematically checks all options until it finds the solution.
- **Works in a structured manner**: It follows a predefined search strategy without shortcuts.

How It Works

- The search starts from the **initial state** and explores the entire **search space**.
- It expands nodes systematically without knowing which direction leads to the **goal state**.
- The search continues until the **goal test** confirms that the solution is found.

Types

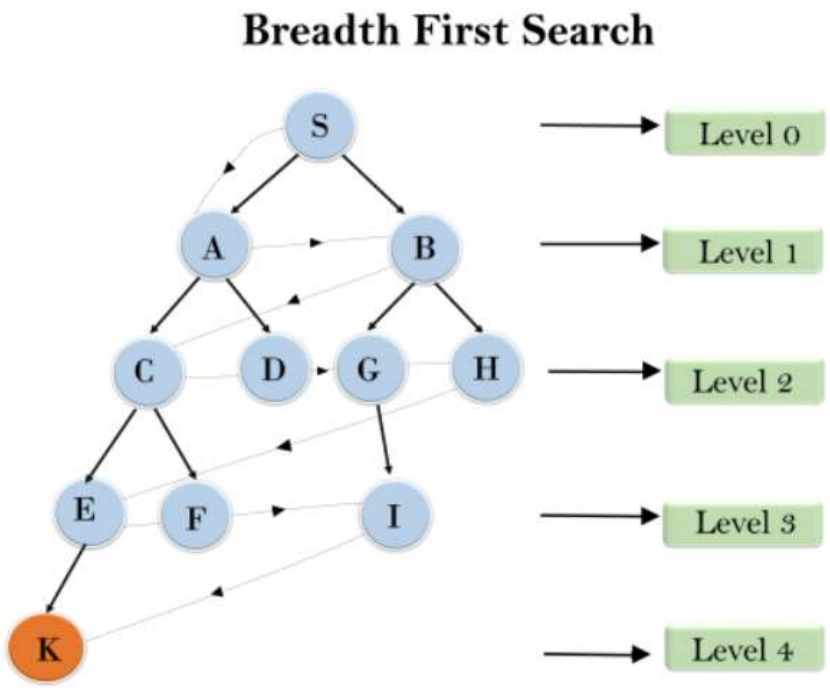
1. Breadth-First Search (BFS) Algorithm - Breadth-First Search (BFS) is a fundamental search algorithm used to traverse or search through a **tree** or **graph**. It explores all nodes at the current depth level before moving on to the next level, making it a level-wise traversal method.

How BFS Works

1. BFS starts at the **root node (initial state)**.
2. It **expands** all its immediate child nodes before moving to the next level.
3. The process continues **level by level** until it reaches the **goal state**.
4. It uses a **FIFO (First In, First Out) queue** to keep track of nodes that need to be visited next.

Example of BFS Traversal

Consider the following tree:



If BFS is applied, it will explore nodes in the following order: **S → A → B → C → D → E → F → G → H → I → K**

Advantages of BFS

1. Guarantees finding a solution if one exists.
2. Finds the shortest path because it explores all possible paths **level by level**, ensuring that the first solution found is the shortest (in terms of the number of steps).
3. Simple to implement using a queue data structure.

Disadvantages of BFS

1. Requires a lot of memory since it stores all nodes at the current level before moving deeper.
2. Slow when the solution is far from the root node, as it expands all nodes at each level before moving deeper.
3. Inefficient for deep search spaces, as it thoroughly explores all nodes at each level before moving on.

Performance Analysis of BFS

1. **Time Complexity**
   - The worst-case time complexity of BFS depends on the number of nodes in the search tree.
   - If **b** is the branching factor (average number of child nodes per parent) and **d** is the depth of the shallowest solution, the total nodes explored are: **T (b) = $1+b^2+b^3+.....+ b^d$= O (b^d)**
   - The time required grows exponentially with depth.
2. **Space Complexity**
   - BFS stores all nodes at a level before moving deeper. This requires: O(b^d).
   - Memory usage is high because all nodes at one level must be stored before moving to the next.
3. **Completeness**
   - BFS is **complete**, meaning it always finds a solution if one exists.
   - Even if the solution is at a deeper level, BFS will reach it eventually.
4. **Optimality**
   - BFS is **optimal** when the cost of moving from one node to another is uniform (e.g., in an unweighted graph).
   - If the cost increases with depth, BFS may not be optimal because it only considers the number of steps, not the actual path cost.

When to Use BFS?

1. When the **shortest path** is required.
2. When the **search space is not too large**.
3. When **all steps have equal cost** (e.g., unweighted graphs).
4. When memory is **not a constraint**.

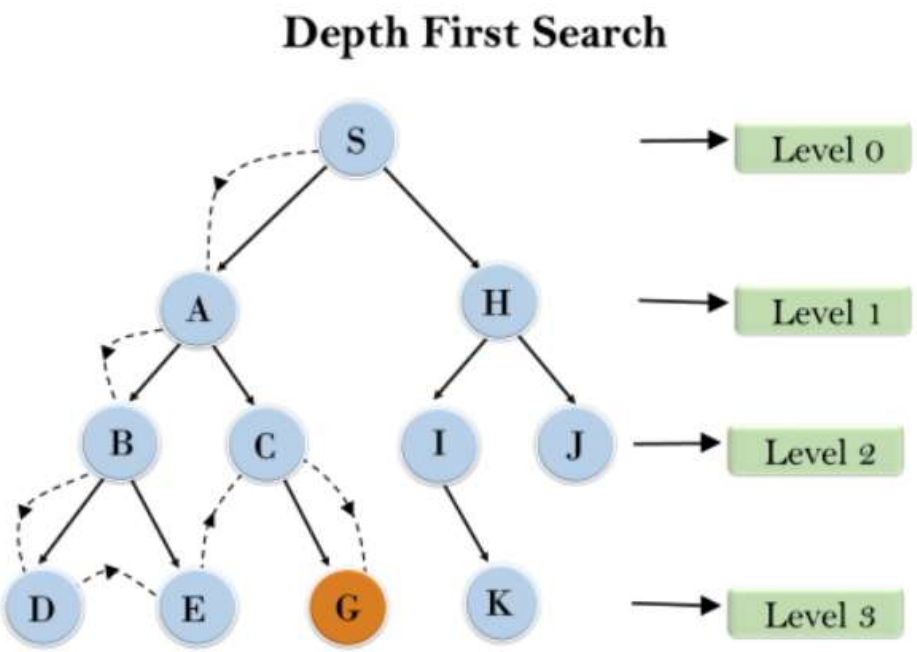
2. Depth-First Search (DFS) Algorithm - Depth-First Search (DFS) is a recursive algorithm used to **traverse or search a tree or graph**. Unlike BFS, which explores level by level, DFS goes **deep into a path** before backtracking and exploring other paths.

How DFS Works

1. DFS **starts at the root node** (or any starting node).
2. It follows **one path as deep as possible** until it reaches a dead end or a goal node.
3. If it reaches a dead end, it **backtracks** to explore the next available path.
4. DFS uses a **stack** (either explicitly using a data structure or implicitly via recursion) to keep track of the nodes being explored.

Example of DFS Traversal

Consider the following tree:



If DFS is applied, it will explore nodes in the following order: **S → A → B → D → E → F → C → G**

DFS goes **deep first**, reaching **E and F** before moving to **C and G**.

Advantages of DFS

1. **Requires less memory** since it only stores nodes along the current path instead of all nodes at a level.
2. **Faster than BFS** in some cases, as it quickly explores a path and may reach the goal sooner.
3. **Efficient for large search spaces** where a solution may be found deep in the tree.

Disadvantages of DFS

1. **May enter an infinite loop** if there are cycles and no mechanism to track visited nodes.
2. **Not guaranteed to find the shortest path** because it follows one path fully before exploring others.
3. **Can be inefficient** in deep trees or graphs where the goal is at a shallower level.

Performance Analysis of DFS

1. **Time Complexity**
   - The worst-case time complexity of DFS depends on the number of nodes and depth of the tree.
   - If **n** is the number of nodes and **m** is the maximum depth, DFS time complexity is: **$O(n^m)$**
   - This can be much larger than BFS if **m** is very deep.
2. **Space Complexity**
   - DFS only needs to store the **current path** and backtracking information.
   - The worst-case space complexity is: O(bm)
   - Here, **b** is the branching factor, and **m** is the maximum depth.
3. **Completeness**
   - DFS is **complete only if the state space is finite**.
   - If there are infinite paths, DFS may **never reach the goal**.
4. **Optimality**
   - DFS is **not optimal** because it does not guarantee the shortest path.
   - It may find a solution that is longer or costlier than the optimal one.

When to Use DFS?

1. When **memory is limited**, as DFS requires less space.
2. When the **search tree is deep**, and the goal node is not necessarily the shortest path.
3. When solving problems that require **backtracking**, such as puzzles and pathfinding in large mazes.

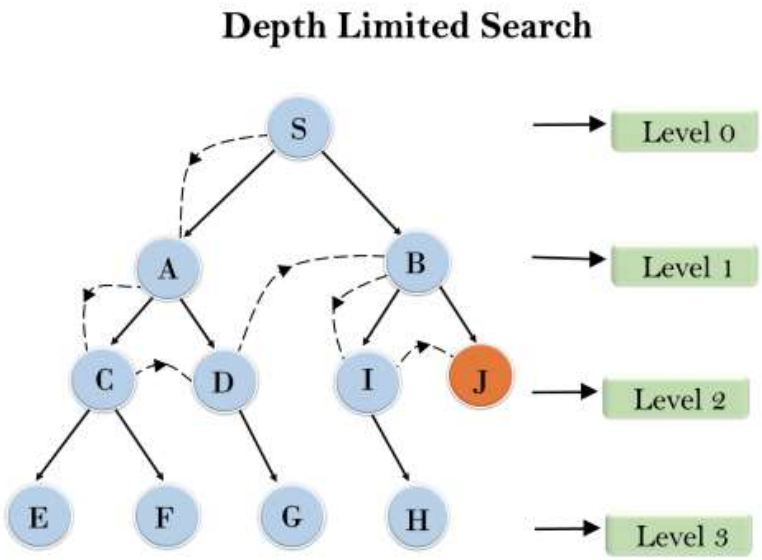
3. Depth-Limited Search (DLS) Algorithm - Depth-Limited Search (DLS) is a variation of the Depth-First Search (DFS) algorithm that introduces a depth limit to **overcome the infinite path problem** in DFS. It stops searching beyond a predefined depth and treats nodes at that level as if they have no successors.

How DLS Works

1. It starts at the **root node** and explores depth-first up to a fixed depth limit.
2. If it reaches the **depth limit**, it stops exploring further down that path.
3. If a goal node is found within the depth limit, the search **succeeds**.
4. If no goal is found, it returns either a **standard failure** (no solution exists) or a **cutoff failure** (solution might exist but is beyond the depth limit).

Example of DLS Traversal

Consider the following tree with a **depth limit of 2**:



If the goal is **F**, DLS will explore nodes **A → B → D → E → C**, but will **not** reach F because it is beyond the depth limit. If the goal was **D or E**, it would be found successfully.

Advantages of DLS

1. **Reduces memory usage** compared to BFS and Iterative Deepening DFS (IDDFS).
2. **Prevents infinite loops** in graphs with cycles.
3. **Efficient for large trees**, where full DFS may be impractical due to depth.

Disadvantages of DLS

1. **Incomplete** if the depth limit is too low, as it may fail to find a solution.
2. **Not optimal**, meaning it does not guarantee the shortest path to the goal.
3. **Performance depends on the depth limit**—setting it too high may waste resources, while setting it too low may miss the solution.

Performance Analysis of DLS

1. **Completeness**
   - DLS is **complete only if the solution exists within the depth limit**.
   - If the goal is beyond the limit, it **fails to find the solution**.
2. **Time Complexity**
   - Given a **branching factor (b)** and a **depth limit (ℓ)**, the time complexity is: **$O(b^{\ell})$**
   - This is similar to DFS but with a restricted depth.
3. **Space Complexity**
   - Since DLS **stores only the current path**, space complexity is: O**(b×ℓ)**
   - This is better than BFS, which requires storing all nodes at a level.
4. **Optimality**
   - DLS is **not optimal**, even if the depth limit is greater than the solution depth.
   - A solution may exist at a lower cost, but DLS does not guarantee finding the best one.

When to Use DLS?

1. When **DFS is preferred** but needs **protection against infinite paths**.
2. When searching a **large space** and memory usage needs to be minimized.
3. When the **maximum solution depth is known**, allowing a reasonable depth limit to be set.

4. Uniform-Cost Search (UCS) Algorithm - Uniform-Cost Search (UCS) is a graph traversal algorithm used when each edge has a different weight. Its primary goal is to find the path with the **lowest cumulative cost** from the starting node to the goal node. Unlike Depth-First Search (DFS) or Breadth-First Search (BFS), UCS expands nodes based on the **total path cost** rather than depth or order of discovery.

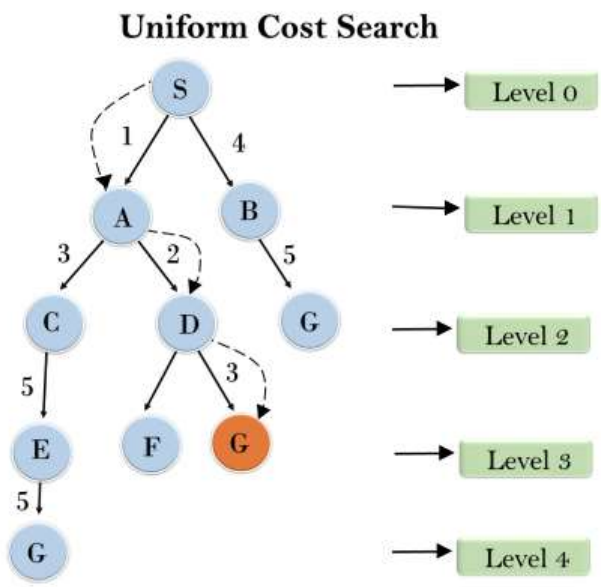
UCS is implemented using a **priority queue**, where nodes with the **least path cost** are given the highest priority. If all edges have the same weight, UCS behaves similarly to BFS.

How UCS Works

1. **Start at the root node** and assign it a cost of **zero**.
2. **Push the node into the priority queue**, where priority is determined by path cost.
3. **Expand the node with the least cost**, adding its neighbours to the priority queue.
4. **Update the path cost** of each neighbour if a lower-cost path is found.
5. **Repeat until the goal node is reached**.

Example of UCS Traversal

Consider the following graph with edge costs:



- Start from node **A** (cost = 0).
- Expand **C** first (cost = 2), then **B** (cost = 4).
- Expand **D** (from C) with a cost of **2 + 1 = 3**.
- The path **A → C → D (cost 3)** is found before **A → B → D (cost 5)**, ensuring the lowest-cost path is chosen.

Advantages of UCS

1. **Optimal Solution** – UCS always finds the path with the lowest cost.
2. **Complete Algorithm** – If a solution exists, UCS will find it.
3. **Works for Weighted Graphs** – Unlike BFS, UCS accounts for different edge costs.
4. **Efficient for Small Edge Weights** – It explores paths in an order that ensures the shortest path is found early.

Disadvantages of UCS

1. **Ignores Step Count** – UCS only focuses on cost and may explore long paths if they have a lower cost.
2. **Requires Knowing Edge Weights** – It must have complete information about edge costs to function.
3. **High Memory Usage** – The priority queue must store all discovered nodes, making UCS memory-intensive for large graphs.
4. **May Get Stuck in Cycles** – If a graph contains cycles with low-cost edges, UCS can keep exploring unnecessary paths.

Performance Analysis of UCS

1. **Completeness**
   - UCS is **complete**, meaning if a solution exists, UCS will **always find it**.
2. **Time Complexity**
   - Let **C*** be the cost of the optimal solution, and **ε** be the smallest step cost.
   - The number of steps required is $C^*/\varepsilon+1$
   - The worst-case time complexity is: $\mathbf{O(b^{1+[C^*/\varepsilon]})}$
   - Here, **b** is the branching factor, meaning how many children each node has on average.
3. **Space Complexity**
   - The same formula applies to space complexity: $\mathbf{O(b^{1+[C^*/\varepsilon]})}$.
   - This is because UCS keeps track of all paths in the priority queue.
4. **Optimality**
   - UCS is **always optimal**, as it only selects paths with the lowest total cost.

When to Use UCS?

- When finding the **shortest or least-cost path** is more important than the number of steps.
- When edge weights **vary** and **cannot be ignored**.
- When searching **small to medium-sized graphs**, UCS can be memory-intensive for large graphs.

5. Iterative Deepening Depth-First Search (IDDFS) - The Iterative Deepening Depth-First Search (IDDFS) is a graph traversal algorithm that **combines the advantages of Depth-First Search (DFS) and Breadth-First Search (BFS)**. It is particularly useful when the search space is **large** and the **depth of the goal node is unknown**.

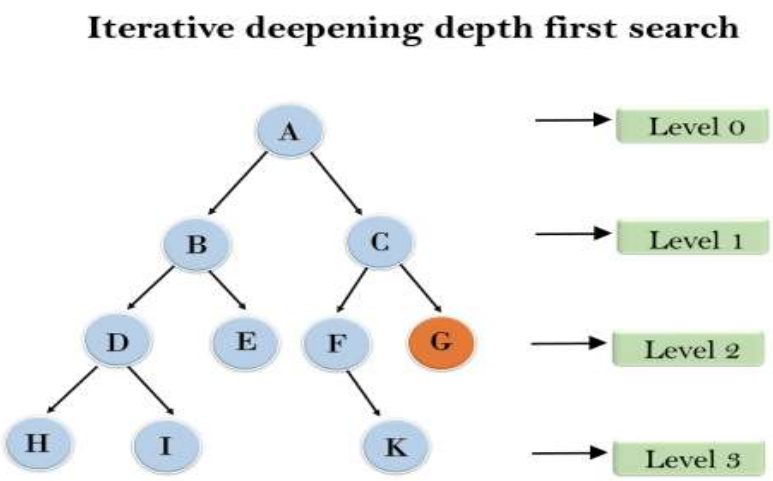
IDDFS works by **repeatedly running a depth-limited DFS**, increasing the depth limit at each iteration until the goal is found.

How IDDFS Works

1. Start with **depth limit = 0**.
2. Perform **Depth-First Search (DFS)** up to the depth limit.
3. If the goal is found, **return the result**.
4. If the goal is **not found**, increase the **depth limit** and repeat.
5. If the goal is not found and the maximum depth is reached, **terminate the search**.

This method ensures that the search **does not get stuck in infinite paths** and can explore the entire graph efficiently.

Example of IDDFS



- **Iteration 1**: A (Depth limit = 0)
- **Iteration 2**: A → B, C (Depth limit = 1)
- **Iteration 3**: A → B → D, E → C → F (Depth limit = 2)
- **Iteration 4**: A → B → D → H, I → E → C → F → K (Depth limit = 3)

If **K is the goal**, it will be found in the **fourth iteration**.

Advantages of IDDFS

1. **Combines the Benefits of BFS and DFS**
    - Uses the **low memory** of DFS.
    - Ensures **optimal search,** like BFS (if edge costs are uniform).
2. **Complete Algorithm**
    - If a **solution exists**, IDDFS **will find it**.
3. **Memory Efficiency**
    - Unlike BFS, it does **not** store the entire tree in memory.
    - Only keeps track of the **current path**, making it efficient.
4. **Guaranteed to Find the Optimal Solution**
    - If all edges have the **same cost**, IDDFS finds the **shortest path**.
5. **Straightforward to Implement**
    - Uses a simple modification of **DFS** to include **depth limits**.

Disadvantages of IDDFS

1. **Repeats Work from Previous Iterations**
    - Each iteration re-explores previously visited nodes.
    - This increases the **computational cost** slightly.

Performance Analysis of IDDFS

1. **Completeness**
    - IDDFS is **complete** if the **branching factor is finite**.
2. **Time Complexity**

- If **b** is the branching factor and **d** is the depth, the worst-case time complexity is **$O(b^d)$**.
- This is similar to BFS in an uninformed search scenario.

3. **Space Complexity**
   - Since IDDFS **only keeps track of the current path**, its space complexity is **O(bd)**.
   - This is much **lower than BFS**, which stores all nodes at each level.
4. **Optimality**
   - IDDFS is **optimal** if the **path cost does not decrease as depth increases**.

When to Use IDDFS?

- When the **search space is large** and the **depth of the goal is unknown**.
- When **memory efficiency** is needed.
- When **uniform path cost** ensures optimality.

6. Bidirectional Search Algorithm - Bidirectional Search is an efficient graph traversal algorithm that **runs two simultaneous searches**:

1. **Forward Search** - Starts from the initial state.
2. **Backward Search** - Starts from the goal state.

These searches progress **simultaneously** until they **meet in the middle**, reducing the number of nodes expanded compared to a single-directional search.

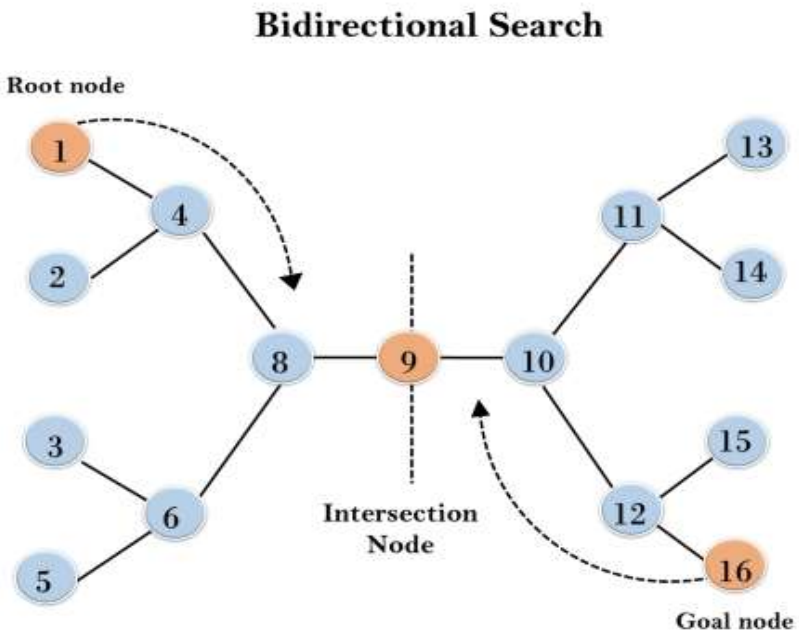
How Bidirectional Search Works

1. Begin **two searches**:
   - One **from the initial node**.
   - One **from the goal node**.
2. Expand the nodes alternately from both directions.
3. **Stop when the two searches meet**, indicating a path has been found.

Instead of searching from **start to goal** in one go, it **reduces the number of expanded nodes**, making it more efficient.

Example of Bidirectional Search

Consider the following search tree:



1. **Forward Search**: Expands from node **1 → 2 → 3 → 4 → 5**.
2. **Backward Search**: Expands from **16 → 15 → 14 → 13 → 9**.
3. The searches **meet at node 9**, so the algorithm terminates successfully.

Advantages of Bidirectional Search

1. **Faster than a single search**
   - Reduces the number of nodes expanded.
   - Instead of searching through the **entire graph**, it searches **only half**.
2. **Requires less memory**
   - Stores **two small search trees** instead of one large tree.
3. **Useful for large graphs**
   - If a graph is too large to explore fully, bidirectional search **reduces the number of steps** needed.
4. **Minimizes node expansion cost**
   - If expanding nodes is expensive, searching in **both directions** helps reduce the total number of expansions.

Disadvantages of Bidirectional Search

1. **Difficult to implement**
    - Requires careful coordination between **both searches**.
2. **The goal state must be known**
    - Unlike **uninformed searches**, this requires a **predefined goal**.
3. **Checking for a meeting point can be complex**
    - Requires an efficient way to **detect when the two searches intersect**.

Performance Analysis of Bidirectional Search

1. **Completeness**
    - The algorithm is **complete** if BFS is used in both searches.
2. **Time Complexity**
    - The time complexity of bidirectional search using BFS is **$O(b^d)$**.
3. **Space Complexity**
    - Space complexity of bidirectional search using BFS is **$O(b^d)$**.
4. **Optimality**
    - Bidirectional search is **optimal** if both searches use BFS.

When to Use Bidirectional Search?

- When the **goal node is known** in advance.
- When the **search space is large**, a **direct BFS or DFS would be slow**.
- When **memory constraints** require a more efficient approach.

2. Informed Search (Heuristic Search)

Informed search algorithms help AI find solutions faster and more efficiently using heuristic functions. A heuristic function estimates how close a certain state is to the goal. These algorithms are better than uninformed search methods because they guide the search process toward the solution instead of blindly exploring all possibilities. These algorithms are widely used in real-world AI applications such as navigation, game playing, and problem-solving.

Heuristic Function

A heuristic function helps an AI algorithm decide which path to explore next by estimating a given state's distance from the goal state. It improves the efficiency of searching by avoiding unnecessary paths.

Types of Heuristic Functions

1. Admissible Heuristic

- A heuristic function is admissible if it never overestimates the cost of reaching the goal.
- It ensures that the algorithm always finds the optimal (best) solution.
- Example: In a map navigation system, if the straight-line distance from one city to another is used as a heuristic, it never overestimates the actual travel cost, making it admissible.

2. Non-Admissible Heuristic

- A heuristic function is non-admissible if it overestimates the cost of reaching the goal.
- This may lead to suboptimal solutions, but in some cases, it makes the search process faster.
- Example: If a navigation system considers traffic conditions but overestimates the delay, it might skip some possible routes even if they are better.

Comparison of Admissible and Non-Admissible Heuristics

Feature	**Admissible Heuristic**	**Non-Admissible Heuristic**
Overestimates cost?	No	Yes
Guarantees the best solution?	Yes	No
Search efficiency	Can be slower	Can be faster
Example	Straight-line distance in maps	Estimated traffic delay in maps

Types of Informed Search Algorithms

1. Best First Search Algorithm (Greedy Search)

- Best First Search selects the next node based only on the heuristic function.
- It always picks the path that appears to be closest to the goal.

Steps to perform Best First Search:

1. Add the start node to the open list.
2. While the open list is not empty:
    - Remove the node with the lowest heuristic value.
    - If it is the goal node, return the solution.
    - Expand the node's children and add them to the open list.
3. If the open list is empty, there is no solution.

Advantages:

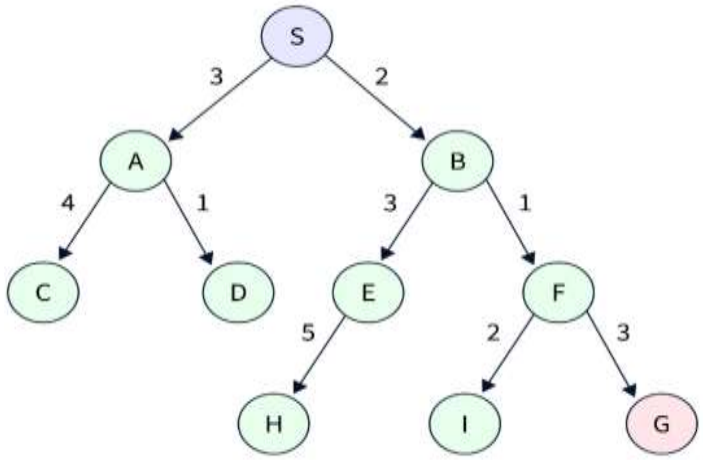
- Faster than uninformed search methods.
- Works well if a good heuristic function is available.

Disadvantages:

- Can get stuck in local minima (a point that seems optimal but is not the best overall).
- Does not guarantee the best solution.

Example:

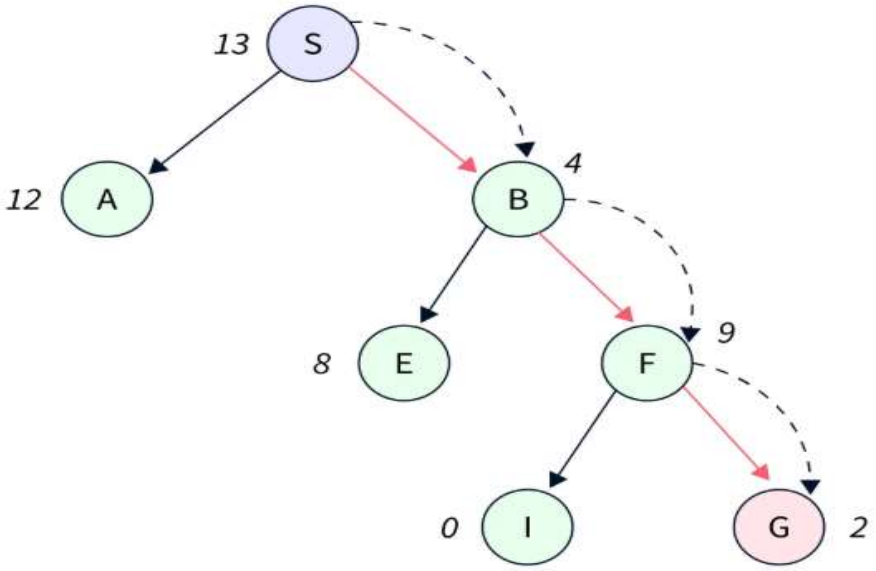
The algorithm selects the node with the lowest heuristic value as the next node to expand. Let's consider the following graph:



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

In this search example, we are using two lists, the OPEN and CLOSED Lists. The iterations for traversing the aforementioned example are listed below.

Solution:



Expand the nodes of S and put them in the CLOSED list

- Initialization: Open [A, B], Closed [S]
- Iteration 1: Open [A], Closed [S, B]
- Iteration 2: Open [E, F, A], Closed [S, B] -> Open [E, A], Closed [S, B, F]
- Iteration 3: Open [I, G, E, A], Closed [S, B, F] -> Open [I, E, A], Closed [S, B, F, G]

Hence, the final solution path will be: S->B->F->G.

2. A Search Algorithm*

A* Search is a combination of:

- Greedy Best First Search, which uses a heuristic function.
- Uniform Cost Search, which considers the cost to reach a node.

A* uses the formula: Where:

- g(n) = Cost from the start node to node n.
- h(n) = Estimated cost from node n to the goal.

Steps of A* Algorithm:

1. Start from the initial node with g(n) = 0.
2. Select the node with the lowest f(n) value.
3. If it is the goal node, return the path.
4. Expand the node's neighbours and calculate their f(n) values.
5. Repeat the process until the goal node is found or no path exists.

Advantages:

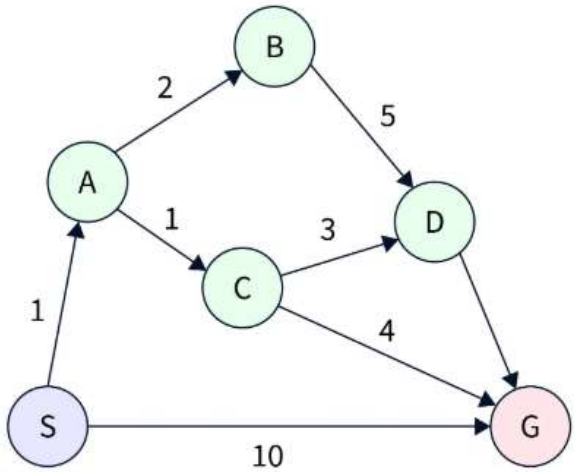
- Guarantees the best solution if the heuristic function is admissible.
- Efficient in finding the shortest path.

Disadvantages:

- Can consume a lot of memory.
- If the heuristic function is poorly chosen, the algorithm may be slow.

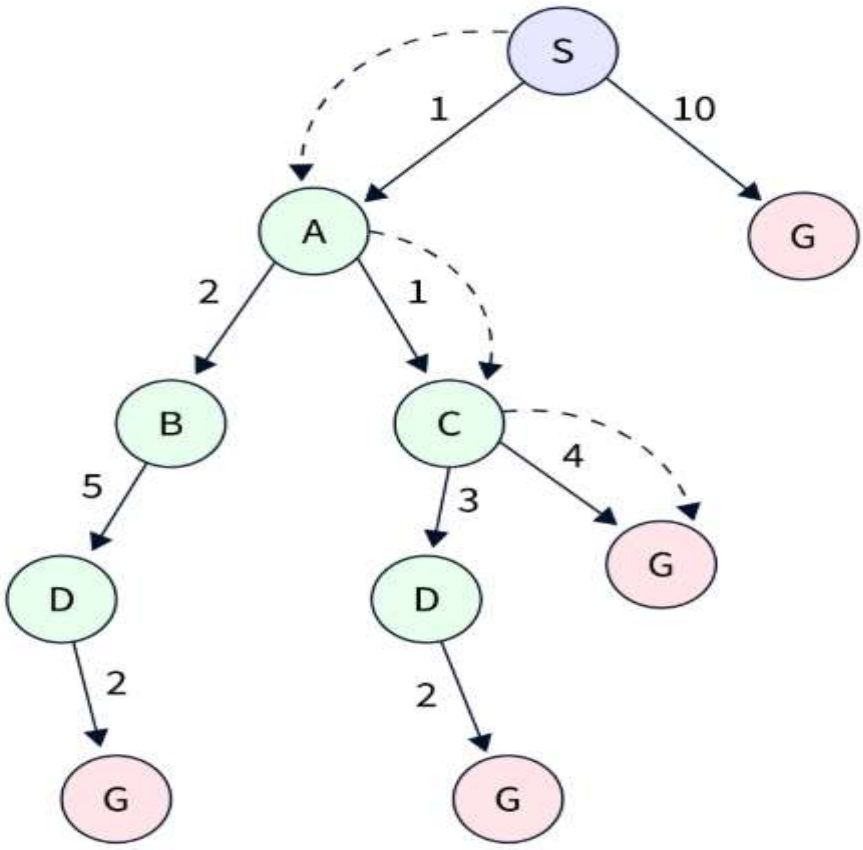
***Example**:*

Consider the following graph:



State	h (n)
S	5
A	3
B	4
C	2
D	6
G	0

Solution:
In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the table so we will calculate the f(n)$f(n)$ of each state using the formula f(n)=g(n)+h(n)$f(n)=g(n)+h(n)$, where g(n)$g(n)$ is the cost to reach any node from the start state.



- Initialization: {(S, 5)}
- Iteration1: {(S--> A, 4), (S-->G, 10)}
- Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}
- Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Iteration 4 will give the final result, as S--->A--->C--->G provides the optimal path with cost 6.

Let's say we want to identify the best route from node A to node F. The straight-line distance can be used as our heuristic function. From node A to node F, the straight line distance is 5.

The procedure begins at node A, where the f-value is 0 + 5, and moves on to node C, where the f-value is 2 + 3 and hence 5. The algorithm then advances to the destination node, node F, with an f-value of 4 + 1 = 5. A -> C -> F is the route from A to F, costing 4.

3. AO* Algorithm

- AO* (And-Or Star) is a **best-first search** algorithm used to solve problems represented as a **Directed Acyclic Graph (DAG)**.
- It evaluates **multiple paths simultaneously**, unlike A*, which explores a single path.
- Used for problems involving **AND-OR nodes**:

- **AND Nodes**: All child nodes must be satisfied.
- **OR Nodes**: At least one child node must be satisfied.

Example of AO* Algorithm

- **Buying a Car**:
  - **AND Section**: Secure funds **and** apply for a loan (both needed).
  - **OR Section**: Buy with **cash** OR **finance** the car (one is enough).

- AO* identifies the best path using **heuristics**, reducing time complexity.

Working of AO* Algorithm

1. Node Types

- **AND Nodes**: All child nodes must be completed.
- **OR Nodes**: Only one child node needs to be completed.

2. Heuristic Function

- **h(n)**: Estimated cost to reach the goal from node **n**.
- **Helps select the best path**, making the search efficient.

3. Search Process

- **For OR Nodes**: Select the child with the **lowest cost**.
- **For AND Nodes**: Consider the **maximum cost** among children.

4. Total Estimated Cost Calculation

- **Formula**:

  $f(n)=C(n)+h(n)$
  - **C(n)** → Actual cost to reach node **n**.
  - **h(n)** → Estimated cost from node **n** to the goal.

***Comparison: A* vs AO** Algorithm**

Aspect	A* Algorithm	AO* Algorithm
Search Type	Best-first search	Best-first search
Path Exploration	Explores all paths	Stops once a solution is found
Solution Optimality	Always optimal	It may not always be optimal
Memory Usage	High	Low
Possibility of Infinite Loop	Possible without checks	Avoids infinite loops

Real-Life Applications of AO* Algorithm

1. **Vehicle Routing Problem**:
    - Determines **shortest routes** for a fleet of vehicles.
    - Minimizes **total distance and time**.
2. **Portfolio Optimization**:
    - Helps in selecting the **best investments** to **maximize returns** and **minimize risks**.

Advantages of AO* Algorithm

- **Efficient**: Evaluates multiple paths at once.
- **Flexible**: Works with **AND-OR nodes**, making it useful for many problems.
- **Near-Optimal**: Uses heuristics to find the best solution.

Limitations of Ao* Algorithm

- **High Complexity**: Can be expensive for large graphs.
- **Memory Usage**: Needs storage for multiple nodes.
- **Heuristic Dependency**: Poor heuristics **reduce efficiency**.

4. Game Search

Game search in **Artificial Intelligence (AI)** is the process of analysing possible moves in a game and selecting the **best possible action** based on the current state. AI uses **search algorithms** to predict future game scenarios and decide optimal strategies.

AI-based game search is commonly used in:

- **Board games** (Chess, Go, Checkers, Tic-Tac-Toe)

- **Card games** (Poker, Rummy, Blackjack)
- **Strategy games** (Dota 2, StarCraft, Civilization)
- **Puzzle games** (Sudoku, Rubik's Cube)
- **Multiplayer games** (Call of Duty, FIFA, Ludo)

Game-playing AI is built using **search trees, heuristics, and probability models** to evaluate moves and strategies.

Types of Games in AI

1. Based on Information Availability

- **Perfect Information Games**
  - Every player has **full knowledge** of the game state.
  - No **hidden elements** or **uncertainty**.
  - **Example:** Chess, Checkers, Tic-Tac-Toe, Go.
- **Imperfect Information Games**
  - Some information is **hidden** from players (e.g., opponent's cards in Poker).
  - AI must **estimate unknown information** based on probabilities.
  - **Example:** Poker, Ludo, Battleship, Strategi.

2. Based on Determinism

- **Deterministic Games**
  - No randomness; **only player moves affect outcomes**.
  - AI can **fully predict** future moves.
  - **Example:** Chess, Checkers, Go, Tic-Tac-Toe.
- **Stochastic (Random) Games**
  - Game involves **random elements** (e.g., dice rolls, shuffled cards).
  - AI must plan for **uncertainty and randomness**.
  - **Example:** Ludo, Poker, Monopoly, Backgammon.

3. Based on Player Outcomes

- **Zero-Sum Games**
  - **One player's gain = Another player's loss**.

- AI aims to **maximize its gain while minimizing the opponent's gain**.
- **Example:** Chess, Rock-Paper-Scissors, Tic-Tac-Toe.

◆ **Non-Zero-Sum Games**

- Players **can cooperate** (mutual benefit possible).
- AI **balances competition and cooperation**.
- **Example:** Economic simulations, cooperative board games.

Game Search Techniques in AI

1. Minimax Algorithm (Basic AI Decision-Making)

The **Minimax algorithm** is a decision-making algorithm for **two-player games**.

- **Player 1 (MAX)** tries to **maximize the score**.
- **Player 2 (MIN)** tries to **minimize MAX's score**.

How Minimax Works?

- AI **builds a game tree** by exploring all possible moves.
- **MAX (AI) selects the best move** to maximize its score.
- **MIN (Opponent) selects the worst move** for MAX.
- The algorithm explores **all possible outcomes** and selects the **optimal move**.

Example: Minimax in Tic-Tac-Toe

- **Step 1:** AI evaluates all possible moves.
- **Step 2:** The opponent responds with the worst move for AI.
- **Step 3:** AI selects the best move that guarantees the highest score.

Minimax Time Complexity

- **$O(b^d)$** → where **b = branching factor** (possible moves per turn), and **d = depth of the game tree**.
- **Higher depth = More calculations = Slower algorithm**.

UNIT 3: Probabilistic Reasoning

1. Probability

Introduction to Uncertainty in AI

- **Uncertain Knowledge**: An agent is said to have uncertain knowledge when it **lacks complete or perfect information** about its environment or the outcomes of actions.
- **Importance**: Even with uncertainty, agents **must make decisions** to function effectively.
- **Examples**:
  - An **autonomous car** must decide whether to slow down when it **might** have seen a pedestrian.
  - A person wears a **seatbelt** not because they know an accident will happen, but because there's a **chance** of it.

Why Use Probability?

- **Definition**: Probability is a **numerical representation** of an agent's belief about the truth of a proposition under uncertainty.
  - The **real world is uncertain**, but decisions still need to be made.
  - Helps agents **quantify risk** and avoid blind decision-making.
  - Probabilities can be **learned from data**.
  - **Bayes' Rule** combines **prior beliefs** with **new evidence**.
  - Reflects an agent's **degree of belief**, based on:
    - Prior knowledge
    - Current observations

Representing Belief Numerically

- **Belief = Probability**:
  - A proposition **f** is assigned a probability **P(f)** in the range **[0, 1]**.
  - Interpretation:
    - **P(f) = 1** → definitely true
    - **P(f) = 0** → definitely false
    - **0 < P(f) < 1** → uncertain
  - Probability measures **belief**, not truth.
  - It captures **ignorance or uncertainty**, not logical certainty.

Random Variables and Propositions

- **Random Variable**: A symbol (e.g., X, Y) representing an unknown value from a **set of possible outcomes**.
- **Range(X)**: Set of values X can take (e.g., {true, false}, {sunny, cloudy, rainy}).
- **Multiple Variables**: A group like (X_1, X_2, ..., X_n) can represent **complex situations**.

- **Assignment**:
  - X = x means X has the value x.
  - Example proposition: X = 5 AND Y = true.

These are the rules that all probabilities must follow:

1. **Non-Negativity**: (Probability is never negative)

$$P(a) \geq 0$$

2. **Normalization**: (A universally true statement must have probability 1)

$$P(true)=1$$

3. **Additivity** (for mutually exclusive a and b): (If a and b can't both be true, their combined probability is the sum)

$$P(a \vee b)=P(a)+P(b)$$

Uses of Probability in AI

1. **Handling Uncertainty**: Helps AI make decisions when information is incomplete or unclear.
2. **Predicting Outcome**: Used to guess future events, like weather or user behaviour.
3. **Machine Learning**: Algorithms like Naive Bayes use probability to learn from data.
4. **Decision Making**: AI chooses the best action based on the highest probability of success.
5. **Natural Language Processing**: Predicts the next word or sentence in speech and text.
6. **Sensor Data Interpretation**: Understands noisy or unclear data from sensors using probabilities.
7. **Bayesian Networks**: Model relationships between variables and reasons for them.
8. **Belief Updating**: Updates knowledge when new data is received (using Bayes' theorem).
9. **Game Playing & Planning**: Helps AI plan moves and predict opponents in uncertain situations.
10. **Risk Assessment:** Assesses risks in tasks like driving or medical diagnosis.
11. **Reinforcement Learning**: Learns the best actions over time using reward probabilities.

2. Uncertainty

- In earlier knowledge representation (like propositional and first-order logic), we dealt with **certain knowledge** where we were sure whether statements were true or false.
- Example of certain logic: **A → B** (If A is true, then B is true)
- But in many real-world situations, we are **not sure** whether A is true. This kind of situation is called **uncertainty**.
- **Uncertainty** means we don't have complete or reliable information about the truth of a statement or outcome.
- **Causes of Uncertainty**

Uncertainty can happen due to many real-world factors, such as:

1. **Unreliable Sources**: Information may come from sources that cannot be trusted.
2. **Experimental Errors**: Mistakes during experiments can cause inaccurate data.
3. **Faulty Equipment**: Sensors or machines might give wrong readings.
4. **Temperature Variations**: Physical conditions can affect results.
5. **Climate Change**: Environmental factors may lead to unpredictable behaviour.

3. Probabilistic Reasoning

- **Probabilistic reasoning** is a method of representing and dealing with uncertain knowledge by using **probability theory**.
- It helps to **combine logic with probability** to reason even when information is incomplete or unreliable.
- **Purpose**: To make intelligent decisions in situations where outcomes are not certain.
- **Why Use Probability in Reasoning?**
  - **Probability** provides a way to **measure and manage uncertainty**.
  - It helps when:
    - We are not 100% sure of the outcome.
    - Complete data is not available.
    - The world behaves in unpredictable ways.
  - **Real-world examples**:
    - "It might rain today."
    - "Player A has a higher chance to win."
    - "This medicine is likely to work."
- **Need for Probabilistic Reasoning in AI**
  1. **Outcomes are unpredictable**.
  2. **Predicates are too many** to represent individually.
  3. **Unknown errors** occur during experiments or observations.
- **Methods Used in Probabilistic Reasoning**
  1. **Bayes' Rule** – Used to update beliefs based on new evidence.
  2. **Bayesian Statistics** – Used for statistical inference and decision-making under uncertainty.

4. Important Terms in Probabilistic Reasoning

Probability

- A measure of how likely an uncertain event is to occur.
- Always lies between 0 and 1.

$$0 \leq P(A) \leq 1$$

  - **P(A) = 1** → Complete certainty
  - **P(A) = 0** → Complete uncertainty

Complement Rule

- Probability of an event **not happening** is: **P(¬A)=1−P(A)**

Event

- A specific outcome or result.
- Example: Getting a 6 when rolling a die is an event.

Sample Space

- The set of **all possible events** or outcomes.
- Example: For a coin toss → {Heads, Tails}

Random Variables

- Symbols used to represent uncertain events or objects.
- Example: Let X represent the result of rolling a die. X can be 1 to 6.

Prior Probability

- The probability **before** seeing any new evidence.
- Example: Chance of raining today based on last week's pattern.

Posterior Probability

- The probability **after** considering new evidence.
- Combines prior knowledge and new information.
- Example: After observing dark clouds, we revise the chance of rain.

5. Conditional Probability

Conditional probability refers to the **likelihood of one event occurring**, **given** that another event has already occurred. It is denoted by**: P(A | B)**

This concept is essential when the outcome or occurrence of one event **affects or provides information** about another.

- $P(A \mid B) = \frac{P(A \cap B)}{P(B)}, \; where \; P(B) > 0$

  Where:

  P(A|B): Probability of A given B

  P(A ∩ B): Joint probability of both A and B happening

  P(B): Probability of B occurring (marginal probability)

- When event **B has occurred**, the **sample space is limited** to event B.
- We are interested in the **fraction of B** where event A also happens.
- Conditional probability is used when information about one event influences our estimation of another.
- **Example – Medical Diagnosis**

  Let:

  - A = Patient has a particular disease
  - B = Patient tests positive in a diagnostic test

We aim to find:

$$P(Disease \mid Positive\ Test) = \frac{P(Disease\ and\ Positive\ Test)}{\mathrm{P(Positive\ Test)}}$$

This is used in real-world diagnosis to determine **how likely a person is to have the disease**, given a positive test result. This is more informative than just looking at the test outcome in isolation.

6. Bayes' Theorem in Artificial Intelligence (AI)

Introduction

- **Bayes' Theorem** is also known as Bayes' Rule, Bayes' Law, and Bayesian Reasoning.
- It is used to determine the probability of an event when there is incomplete or uncertain information.
- It helps update predictions as new data becomes available.
- It is a fundamental part of **Bayesian statistics** and **probabilistic reasoning** in AI.

What Bayes' Theorem Does

- It relates to **conditional probability** and **marginal probability**.
- Calculates the probability of a hypothesis **A**, given some observed evidence **B**.
- Useful in **decision-making**, **inference**, and **prediction** tasks in AI.

Bayes' Theorem Formula

Using the **product rule** of probability:

- $P(A \wedge B) = P(A \mid B) \cdot P(B)$
- $P(A \wedge B) = P(B \mid A) \cdot P(A)$

Equating both:

$$P(A \mid B) = \frac{P(B \mid A)\ .\ P(A)}{\mathrm{P(B)}}$$

Applications

1. Robotics: A robot is trying to figure out whether it has reached a hallway based on its sensor detecting a long, empty path.

- **A**: Robot is in a hallway (hypothesis).
- **B**: Sensor shows a long space (evidence).
- **P(A|B)**: Probability that it's in a hallway **given** the sensor reading.

2. Weather Forecasting: Predicting if it will rain based on cloud data.

- **A**: It will rain today.
- **B**: The sky is cloudy (evidence).

- **P(A|B)**: Probability of rain **given** it is cloudy now.

3. Medical Diagnosis: Predicting whether a patient has a disease based on symptoms.

- **A**: The Patient has the disease (hypothesis).
- **B**: Patient shows a specific symptom (evidence).
- **P(A|B)**: Probability that the patient has the disease **given** that they show the symptom.

4. Spam Email Detection: Deciding whether an email is spam based on certain words in it.

- **A**: Email is spam.
- **B**: Email contains the word "free" (evidence).
- **P(A|B)**: Probability that the email is spam **if** it contains "free".

5. Naive Bayes Classifier (In Machine Learning): Classifying text or images (like positive or negative sentiment).

- **A**: Text is "positive" in sentiment.
- **B**: The text includes the word "great".
- **P(A|B)**: Probability that the text is positive **if** it contains the word "great".

6. Voice Assistant Understanding Commands: Your AI assistant (like Siri or Alexa) tries to understand whether you said "Play Music" or "Call Mom" based on the sound input.

- **A**: The user said "Play Music" (hypothesis).
- **B**: The assistant received a certain sound pattern (evidence).
- **P(A|B)** tells us: What's the chance that the user said "Play Music" given this sound pattern?

7. Self-Driving Cars – Recognizing a Stop Sign: The camera of a self-driving car detects a red octagon. It wants to know if this is a stop sign.

- **A**: The object is a stop sign.
- **B**: The camera detects a red octagonal shape.
- **P(A|B)** tells us: What's the chance this object is a stop sign based on the camera's input?

8. Fraud Detection in Online Payments: A system detects a large transaction from a new device. Is it fraud?

- **A**: The transaction is fraudulent.
- **B**: The transaction is large and from a new device.
- **P(A|B)** tells the system: What's the chance this is fraud **given** the new device and large amount?

9. Email Auto-Categorization: An email filtering system tries to decide if a message belongs to the "Promotions" folder.

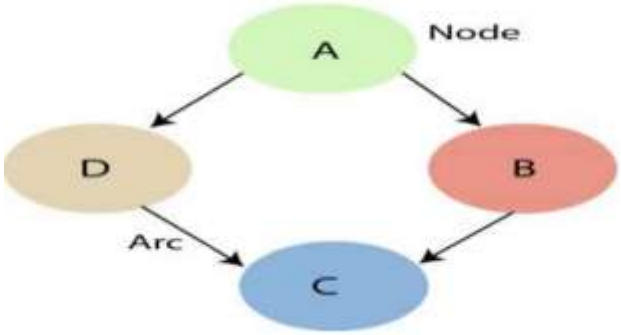
- **A**: The email is promotional.
- **B**: The email has words like "Sale," "Offer," and "Discount."
- **P(A|B)**: Probability that the email is promotional **given** it contains promotional keywords.

10. Online Shopping Recommendation System: A system wants to predict whether a user will buy a product after seeing an ad.

- **A**: The User will buy the product.
- **B**: User clicks on the product ad.
- **P(A|B)**: Probability that the user will buy the product **after** clicking the ad.

7. Bayesian Networks- representation, construction, and inference

- A **Bayesian Belief Network (BBN)** is a **probabilistic graphical model**.
- It represents a set of **variables** and their **conditional dependencies** using a **Directed Acyclic Graph (DAG)**.
- Also known as: **Bayes Network**, **Belief Network**, **Decision Network**, or **Bayesian Model**.
- Useful for reasoning under **uncertainty**.



Why Bayesian Network?

- Real-world problems are **uncertain and probabilistic**.
- BBNs model these problems by **quantifying uncertainty** and **representing relationships** between variables.
- Applied in:
  - Prediction
  - Anomaly detection
  - Diagnostics
  - Automated reasoning
  - Time series prediction
  - Decision making under uncertainty

Components of a Bayesian Network

1. **Directed Acyclic Graph (DAG)**:
   - **Nodes**: Represent random variables (can be discrete or continuous).
   - **Edges/Arrows**: Represent conditional dependencies or causal relationships.
   - If Node A → Node B, then A is the **parent** of B and influences it.

2. **Conditional Probability Tables (CPT)**:
   - For each node, there is a table that defines the probability of that node **given its parent(s)**.
   - For a variable with k binary parents, the CPT contains $\mathbf{2^k}$ **entries**.

Basics of Probability Used in Bayesian Networks

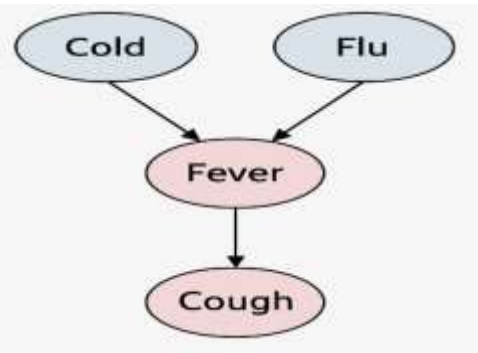
1. **Joint Probability**:
   - Shows the probability of multiple variables happening together.
   - Written as:
     $P(A, B, C) = P(A \mid B, C) \times P(B \mid C) \times P(C)$
     (This is called the **Chain Rule** of probability.)
2. **Conditional Independence**:
   - Two variables A and B are **conditionally independent** given C if:
   - $P(A, B \mid C) = P(A \mid C) \times P(B \mid C)$
   - Meaning: Once C is known, A and B do not influence each other.

Structure of Bayesian Networks

- The network is a **Directed Acyclic Graph (DAG)**.
- Each **node** is a variable.
- An **edge A → B** means that B depends on A, i.e., $P(B \mid A)$ must be known.
- Every node has a **Conditional Probability Table (CPT)** that shows the probability of that variable given its parent(s).

Example

- **Cold** and **Flu** both independently increase the chance of **Fever**.
- **Fever** alone increases the chance of **Cough**.



Nodes:

- **C** = Cold (True/False)
- **F** = Flu (True/False)
- **Fe** = Fever (True/False)
- **Co** = Cough (True/False)

1. Prior Probabilities (no parents):

Variable	True	False
P(Cold)	0.3	0.7
P(Flu)	0.2	0.8

2. Conditional Probability Table for Fever (Fe)

(Depends on: Cold, Flu)

Cold	Flu	P(Fever = True)	P(Fever = False)
T	T	0.95	0.05
T	F	0.7	0.3
F	T	0.8	0.2
F	F	0.1	0.9

3. Conditional Probability Table for Cough (Co)(Depends on: Fever)

Fever	P(Cough = True)	P(Cough = False)
T	0.85	0.15
F	0.2	0.8

Uses of Bayesian Networks

- Medical Diagnosis
- Spam Filtering
- Machine Learning
- Fraud Detection
- Weather Forecasting
- Fault Detection in Systems
- Decision Support Systems

Advantages of Bayesian Networks

- Helps **reduce computation** by only focusing on relevant variables.
- Efficient in dealing with **uncertainty** and incomplete data.
- Good for **reasoning and prediction** in AI systems.

Inference in Bayesian Networks

Inference means **finding unknown probabilities** based on **known facts (evidence)**.

1. Joint Probability Inference

- Used to calculate the **probability of a specific combination of values**.
- It follows the chain rule:

  P(A, B, C) = P(A | B, C) × P(B | C) × P(C)

Example:

Find **P(Cold = True, Flu = False, Fever = True, Cough = True)**

Using values from the CPT (Conditional Probability Tables):

- P(Cold = T) = 0.3
- P(Flu = F) = 0.8
- P(Fever = T | Cold = T, Flu = F) = 0.7
- P(Cough = T | Fever = T) = 0.85

Now multiply them: **P = 0.3 × 0.8 × 0.7 × 0.85 = 0.1428**

So, the joint probability is 0.1428 or 14.28%

2. Conditional Probability Inference

- Used to compute the probability of some variables **given evidence**.
- Formula:

  P(x | e) = P(x, e) / P(e) = α × P(x, e)

  Where:

  - x = variables of interest
  - e = known evidence
  - α = normalization constant (1 / total probability of evidence cases)

Example: Find **P(Cold = True | Cough = True)**

Steps:

1. Find P(Cold = T and Cough = T) = 0.19125
2. Find total P(Cough = T) = 0.33405
3. Use the formula: **P(Cold = T | Cough = T) = 0.19125 / 0.33405 ≈ 0.57**

So, the answer is 57%

Challenges with Inference

- In large networks, exact inference is **very slow** because many variables need to be summed out.
- Two types of methods are used:

  A. Exact Inference: Variable Elimination

  - Factors involve fewer variables.
  - Summation is done smartly to reduce extra calculations.

  B. Approximate Inference: MCMC (Markov Chain Monte Carlo)

  - Used when exact methods are too slow.

- It gives an **estimate** of the probability.
- Much **faster** than exact methods in large networks.

8. Temporal Models

What are Temporal Models?

- Used in AI to model situations that **change over time**.
- Help agents make decisions in **uncertain and dynamic environments**.
- Since data is often **noisy or partial**, agents rely on **probabilities** to estimate the current state.

Main Sections of Temporal Models

1. **Time and Uncertainty**
    - States and Observations
    - Stationary Processes and the Markov Assumption
2. **Inference in Temporal Models**

1. Time and Uncertainty

When the environment changes over time, we use models that:

- Represent the **state of the world at each time step**.
- Include **uncertainty** due to noisy or incomplete data.

a) States and Observations

- Think of the world as a series of **time slices (snapshots)**.
- Each time slice has:
    - **State variables (X_t)** – hidden/internal conditions (we can't directly see).
    - **Evidence variables (E_t)** – observable data (what we can measure).
- At each time t, the agent sees evidence $E_t = e_t$ and tries to guess the hidden state X_t.

Example: Umbrella and Rain

- You want to know if it's raining, but you're underground.
- You **observe** whether the director carries an umbrella (evidence).
- You **infer** if it's raining or not (hidden state).

b) Stationary Processes and the Markov Assumption

Once we define states and observations, we need to model **how they are connected over time**.

a. Stationary Processes

- A process is **stationary** if the **rules don't change over time**.
- Example: If the probability of rain today, given it rained yesterday, is always the same, we call it stationary.

b. **Markov Assumption**
   - Simplifies modelling by saying the **current state depends only on recent history**.
   - **First-order Markov**: current state depends **only on the previous state**.
     - $P(X_t \mid X_{t-1})$
   - **Second-order Markov**: current state depends on the **last two states**.
     - $P(X_t \mid X_{t-2}, X_{t-1})$

c. **Why Markov is Useful**
   - Reduces complexity by limiting how much past information is needed.
   - Makes it easier to calculate probabilities.

2. Inference in Temporal Models

Inference means **estimating the hidden state** of a system over time using observed data.

There are **4 main types** of inference:

1. Filtering (Monitoring)
- **Findings:** Current state
- **Based on:** All observations **up to now**
- **Example:** Is it raining **today**, given the umbrella history?

2. Prediction
- **Findings:** Future state
- **Based on:** All **past and current** observations
- **Example:** Will it rain **tomorrow**?

3. Smoothing
- **Finds:** A **past** state
- **Based on: All** observations (past, current, and future)
- **Example:** Did it rain **3 days ago**?

4. Most Likely Explanation (MLE)
- **Find:** The **most probable sequence** of states
- **Example:** What is the most likely sequence of rain/no-rain days this week?

9. Hidden Markov Model (HMM)

What is a Hidden Markov Model?

- A **Hidden Markov Model (HMM)** is a statistical model used for systems with **hidden states** that evolve.
- **"Hidden"** means the state of the system is not directly observable. Instead, the observer can only see some **output** that depends on the state.
- HMMs are based on the **Markov Property**, which states that the future state depends only on the current state, not the history of previous states.

Components of HMM

HMMs consist of the following components:

- **States:** The possible conditions of the system, but these are not directly visible.
- **Observations:** The visible outputs are probabilistically dependent on the hidden states.
- **Transition Probabilities:** The probability of transitioning from one state to another.
- **Emission Probabilities:** The probability of observing a specific output given a hidden state.
- **Initial State Probabilities:** The probability of starting in each hidden state at the beginning.

These components are described using:

- A **transition matrix** (defining the transition probabilities).
- An **emission probability distribution** for each state.
- An **initial state distribution**.

Applications of HMMs

HMMs are widely applied in many fields, particularly for temporal and sequential data:

- **Speech Recognition:** Modelling the sequence of sounds in speech.
- **Bioinformatics:** Gene prediction, protein sequence modelling, and biological sequence alignment.
- **Natural Language Processing (NLP):** Part-of-speech tagging based on the context of words in a sentence.
- **Finance:** Modelling market conditions and predicting stock prices or market regimes.

Limitations of HMMs

Despite their versatility, HMMs have some limitations:

- **Markov Property Limitation:** The assumption that future states depend only on the current state might not hold for all systems, especially for complex ones.
- **Computational Complexity:** As the number of states increases, the model becomes computationally expensive.
- **Real-World Assumptions:** The model may not perform well if the actual system doesn't fit the assumptions of the HMM.

Examples:

Hidden States (Weather States):

- S: Sunny
- R: Rainy

Observations (Observable Events):

- U: Umbrella (Yes)
- N: No Umbrella (No)

1. Transition Probability Table

These are the probabilities of transitioning from one state to another between two consecutive time steps.

Current State	Next State (Sunny)	Next State (Rainy)
Sunny (S)	0.7	0.3
Rainy (R)	0.4	0.6

2. Emission Probability Table

These represent the probability of observing an Umbrella (U) or No Umbrella (N) given the current weather state.

State/Observation	Umbrella (U)	No Umbrella (N)
Sunny (S)	0.2	0.8
Rainy (R)	0.8	0.2

3. Initial State Probability Table

This table represents the initial probability of the weather being either Sunny or Rainy at the start.

Initial State	Probability
Sunny (S)	0.6
Rainy (R)	0.4

UNIT 4: Markov Decision Process

1. Markov Decision Process (MDP)

A **Markov Decision Process (MDP)** is a mathematical framework used to model decision-making in situations where outcomes are partially random and partially controlled by the decision-maker. It is a central concept in **Reinforcement Learning (RL)**, where an agent learns to make decisions to maximize cumulative rewards through interaction with the environment.

Key Components of MDP

1. **States (S)**:
   - Represents all possible situations or configurations the system (or agent) can be in.
   - Example: In a grid world, states could be the positions of the agent on the grid.
2. **Actions (A)**:
   - These are the choices available to the agent at each state.
   - Example: In the grid world, actions could include moving up, down, left, or right.
3. **Transition Model (T)**:
   - Describes the probability of moving from one state to another after taking an action.
   - Denoted as **P(s' | s, a)**, which is the probability of transitioning from state **s** to state **s'** after taking action **a**.
   - Example: If the agent is in state **S1** and takes action **A1**, the transition model specifies the probability of ending up in **S2**.
4. **Rewards (R)**:
   - Specifies the immediate reward the agent receives after taking an action in a state.
   - Denoted as **R(s, a)**, which gives the reward after the agent takes action **a** in state **s**.
   - Example: In a maze, reaching the goal could give a positive reward, while hitting an obstacle might give a negative reward.
5. **Discount Factor (γ)**:
   - A value between **0** and **1** that determines the importance of future rewards.
   - A higher discount factor (close to 1) means the agent values future rewards more, while a lower discount factor (close to 0) prioritizes immediate rewards.
   - The total value of future rewards is computed as:

$$V(s) = R(s) + \gamma \sum_{s'} P(s'|s,a)V(s')$$

   - A discount factor of **0** means the agent only cares about immediate rewards, while **1** means the agent values future rewards equally.

6. **Policy (π)**:
    - A strategy or plan that specifies the action the agent should take in each state.
    - Can be **deterministic** (a fixed action for each state) or **stochastic** (a probability distribution over actions).
    - Example: A policy could instruct an agent to always move right in a maze or select actions probabilistically.

Markov Property

- The **Markov Property** states that the future state depends only on the current state and action, not on the history of past states or actions.
- Formally, it is expressed as:

$$P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_t)$$

This means that the future is independent of the past, given the present.

Objective in MDP

The goal of an agent in an MDP is to find a **policy (π)** that maximizes the expected sum of rewards over time. This is represented as:

$$J(\pi) = E\left[\sum_{t=0}^{T} \gamma^t R(s_t, a_t)\right]$$

Where:

- **T** is the time horizon.
- **γ** is the discount factor.
- **R(s_t, a_t)** is the reward at time step **t**.

The agent aims to maximize the **expected return** by balancing immediate rewards with potential future rewards.

Bellman Equation

The **Bellman Equation** is a recursive relationship that helps in solving for the optimal policy. It relates the value of a state to the expected return of taking an action in that state and following the optimal policy thereafter.

1. **Bellman Equation for State-Value Function (V(s))**:

$$V(s) = \max_a \left[R(s, a) + \gamma \sum_{s'} P(s'|s, a)V(s')\right]$$

2. **Bellman Equation for Action-Value Function (Q(s, a))**:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q(s',a')$$

   - **V(s)** is the value of state **s** under the optimal policy.
   - **Q(s, a)** represents the expected return from taking action **a** in state **s** and then following the optimal policy.

Applications of MDPs

- **Robotics**: In autonomous navigation, MDPs help robots decide the best action based on the current environment and desired goal.
- **Game AI**: MDPs are used in strategy games, such as chess or tic-tac-toe, where agents make decisions to achieve optimal outcomes.
- **Finance**: MDPs help in stock market prediction, assisting in making investment decisions to maximize long-term returns.
- **Healthcare**: MDPs are used for medical decision-making, optimizing treatment plans for better patient outcomes.

Markov Reward Process (MRP)

An **MRP** is a simplified version of an MDP, where each state transition is associated with a reward. It is defined by:

(S, P, R,γ)

Where:

- **S** is the set of states.
- **P** is the state transition probability.
- **R** is the reward function.
- **γ** is the discount factor.

MDP in Grid World Example

Consider an agent in a 3x4 grid with the following characteristics:

- The agent starts at position (1,1).
- The goal is to reach the blue diamond at (4,3).
- The agent must avoid the fire at (4,2).
- The agent can move **UP, DOWN, LEFT, and RIGHT**.
- The agent receives small rewards for each step and large penalties for falling into undesirable states.

The transition model specifies:

- 80% chance of moving in the intended direction.
- 10% chance of moving randomly left or right.

The agent's task is to find the shortest path to the goal, maximizing its rewards while avoiding obstacles.

2. Utility Theory

What is Utility Theory?

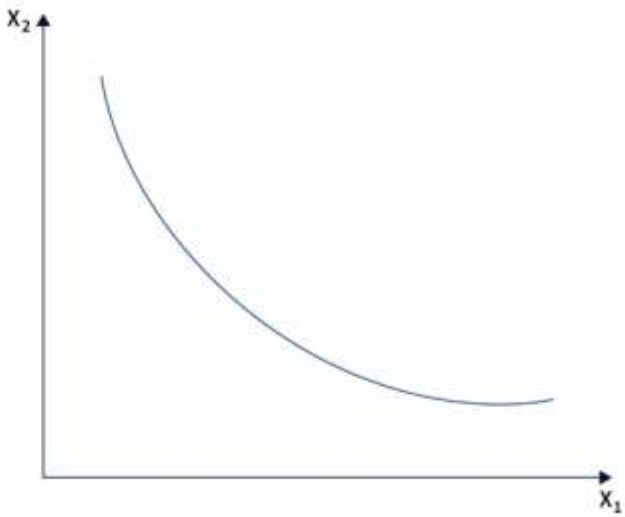
- Utility theory is a way to help AI systems **make decisions**, especially when there's **uncertainty** involved.
- It assigns **values (called utility)** to outcomes depending on how good or useful those outcomes are.
- The goal is to **choose the action that gives the best result** (highest utility).
- It is used in areas like **game theory, robotics, recommendation systems**, and more.

What is Utility?

- Utility means **how much satisfaction or benefit** an outcome gives.
- It's **different for each system or person**.
- For example, a movie recommendation system gives **higher utility** to movies that match your taste.

What is a Utility Function?

- A **utility function** is a mathematical formula that gives a **number (utility value)** to each possible outcome.
- Denoted as **U(x),** where x is the outcome, and U(x) is how good it is.



- Used by AI to **compare and rank different choices**.
- Higher U(x) means the outcome is **better or more preferred**.

What is Expected Utility?

- In uncertain situations, we use **expected utility**.
- It combines the **utility of each outcome** and the **chance (probability)** of that outcome happening.
- AI chooses the option with the **highest expected utility**.

Example

1. Self-Driving Car

- A self-driving car must decide the **best route**.
- It calculates utility for each route using factors like:
  - Time taken
  - Safety
  - Fuel use
- It then picks the **route with the highest expected utility**, considering traffic and other conditions.

2. Movie Recommendation

- A system recommends movies based on:
  - Genre
  - Actor
  - Director
- Movies that match your interests get **higher utility**.
- It recommends the ones with the **highest utility** first.

Notation in Utility Theory

Let us define some basic notation commonly used in utility theory:

- Let *x* represent an outcome or option.
- Let U(x) denote the utility function, which maps x to its utility value.
- Let p(x) denote the probability of outcome x*x* occurring.
- Let E[U(x)] denote the expected utility of outcome x, which is the sum of the utility values of all possible outcomes weighted by their respective probabilities.

The E[U(x)] is calculated using the following expression:

$$E[U(x)]=\sum iP(xi)\cdot U(xi)$$

In this formula, the E[U(x)] represents the expected utility of a decision or action, which is the sum of the product of the probability of each outcome *xi* (denoted as *P*(*xi*)) and its corresponding utility value (denoted as *U*(*xi*)). The summation is taken over all possible outcomes i.

MEU (Maximum Expected Utility)

- AI uses this rule to **make the best decision**.
- It **picks the option** that gives the **highest expected utility**.

Axioms of Utility Theory

Utility theory follows some **rules (axioms)** to make decision-making logical and fair.

1. Orderability

- You should be able to **compare** any two outcomes.
- If U(x) > U(y), then **x is better than y**.

2. Transitivity

- If x > y and y > z, then it must be that x > z.
- Ensures **consistent** preferences.

3. Continuity

- Small changes in chances should not cause big changes in decisions.
- Keeps decisions **stable and smooth**.

4. Substitutability

- If two outcomes are **equally good**, then replacing one with the other **should not change the decision**.

5. Monotonicity

- If the **chance of a better outcome increases**, the **expected utility should also increase**.

6. Decomposability

- Utility can be broken down into **different parts** (like time, cost, etc.).

3. Utility Function

- A **utility function** is a mathematical representation of an agent's **preferences** over a set of possible outcomes.
- It assigns a **numerical value** (called utility) to each outcome, indicating how **desirable** or **satisfactory** that outcome is for the agent.
- Utility functions help agents make **rational decisions** by choosing actions that maximize their overall utility.

Key Characteristics:

- **Quantifies Preferences:** Converts qualitative goals into measurable numeric values.
- **Agent-Specific:** Each agent may have its utility function based on its goals or environment.
- **Supports Decision-Making:** Helps in selecting the best possible action among alternatives.
- **Used in Uncertain Environments:** Especially useful when outcomes are uncertain or partially known.

Example:

- Consider a robot with two possible outcomes:
  - Reaching the goal = Utility 1
  - Failing to reach = Utility 0
- The robot will prefer the action that leads to the outcome with utility 1.

Mathematical Representation:

- A utility function is typically written as:
  U: O → R

- **O** is the set of possible outcomes.
- $\mathbb{R}$ represents real numbers.
- **U(o)** gives the utility value of outcome **o**.

Representations of Utility Function

1. Numerical Utility Assignment:

- Assigns fixed numbers to different outcomes or states.
- Example:

Outcome	Utility Value
Task Completed	100
Partial Success	50
Failure	0

2. Mathematical Functions:

- Utility can be defined using a formula involving parameters like cost, time, reward, or distance.
- Example:
  - **U(x) = log(x)** or **U(x) = √x**, where **x** is some measurable factor such as profit or performance.

3. Machine Learning-Based Representation:

- When the utility function is too complex to define manually, it can be learned using machine learning.
- Common in environments like:
  - Recommender systems
  - Adaptive game agents
  - Reinforcement learning agents
- Utility values are learned based on previous data, actions, and feedback.

Importance of Utility Functions in AI:

- Help in building **rational agents** that make optimal choices.
- Used in **game theory**, **decision theory**, **reinforcement learning**, and **multi-agent systems**.
- Allow agents to make **trade-offs** and manage **uncertainty** effectively.
- Crucial for applications where goals are **not binary** but have different levels of success or satisfaction.

4. Value Iteration

- **Value Iteration** is a **dynamic programming algorithm** used to compute the optimal policy and value function in **Markov Decision Processes (MDPs)**.
- It helps an **AI agent** decide the **best action** to take in each state to maximize long-term rewards.

Purpose of Value Iteration:

- To determine the **optimal value function** V(s) for each state s.
- Once the optimal values are computed, the **optimal policy** (what action to take in each state) can be easily derived.
- It helps the agent learn **how valuable each state is**, based on expected future rewards.

Key Concepts Involved:

- **MDP (Markov Decision Process):** A framework for decision-making where outcomes are partly random and partly under the control of the agent.
- **States (S):** Possible situations the agent can be in.
- **Actions (A):** Choices the agent can make in each state.
- **Transition Model (P):** Probability of moving from one state to another after taking an action.
- **Reward Function (R):** Immediate reward received after taking an action.
- **Discount Factor (γ):** A value between 0 and 1 that reduces the importance of future rewards.

Bellman Equation (Already covered)

How Value Iteration Works:

1. **Initialize** V(s) to 0 for all states.
2. **Repeat** the following until convergence:
    - For each state s:
        - Update V(s) using the Bellman equation.
3. Once value function V(s) converges (i.e., changes very little in further iterations), derive the **optimal policy**:
    - For each state s, choose the action a that maximizes the expected value.

Pseudocode of Value Iteration:

```
Initialize V(s) = 0 for all states s
Repeat:
    For each state s:
        V(s) = max over actions a of [ sum over s' of P(s'|s,a) * (R(s,a,s') + γ * V(s')) ]
Until V(s) converges for all s
```

Example:

Suppose an agent is navigating a 3-cell world: A → B → C

Let rewards be:

- Reaching C = +10
- All other transitions = 0

The agent uses value iteration to estimate the value of each state and decides the best path to reach C.

Advantages of Value Iteration:

- Finds optimal policies for MDPs.
- More efficient than exhaustive search.
- Useful in environments with **stochastic** (random) transitions.

Disadvantages:

- Can be slow for **large state spaces** (computationally expensive).
- Needs a complete **transition and reward model** of the environment.

Applications of Value Iteration:

- **Robotics:** Path planning
- **Game AI:** Decision making in uncertain environments
- **Reinforcement Learning:** Solving grid worlds or planning problems
- **Navigation Systems:** Finding optimal routes

5. Policy Iteration

- Policy Iteration is a **dynamic programming algorithm** used to compute the **optimal policy** in a **Markov Decision Process (MDP)**.
- It finds the best way (policy) for an agent to act in every state to maximize its **long-term reward**.
- It involves two main steps: **policy evaluation** and **improvement**, repeated until convergence.

Key Terms

- **Policy (π)**: A function that maps each state to an action. It defines the agent's behaviour.
- **Value Function (Vπ)**: Expected total reward when following policy π from a given state.
- **Optimal Policy (π*)**: A policy that results in the highest possible value from each state.
- **Bellman Equations**: Mathematical equations that describe the relationship between values and policies.

Steps in Policy Iteration

Step 1: Policy Evaluation

- Evaluate how good the current policy is by computing the value function **Vπ**.
- Use the **Bellman Expectation Equation** to compute Vπ(s) for all states:

$$V^{\pi}(s) = \sum_{s'} P(s'|s, \pi(s)) \left[R(s, \pi(s), s') + \gamma V^{\pi}(s')\right]$$

- Continue until values converge (i.e., changes are below a small threshold).

Step 2: Policy Improvement

- Improve the current policy by choosing better actions based on the current value function.
- For each state, find an action that gives the maximum expected utility:

$$\pi_{\text{new}}(s) = \arg\max_{a} \sum_{s'} P(s'|s, a) \left[R(s, a, s') + \gamma V^{\pi}(s')\right]$$

- Replace the old policy with the new one.

Repeat

- Continue alternating between policy evaluation and improvement until the policy becomes stable (i.e., does not change).

Pseudocode

```
Initialize π randomly
Repeat:
    1. Evaluate current policy π → compute Vπ
    2. Improve π using the updated Vπ
Until π converges to optimal policy π*
```

Advantages of Policy Iteration

- **Guaranteed convergence** to the optimal policy for finite MDPs.
- Often **converges faster** (fewer iterations) than value iteration in small or moderate-sized problems.
- Provides clear separation between policy evaluation and improvement.

Limitations

- **Policy evaluation** can be **computationally expensive**, especially in large state spaces.
- Not suitable for **very large or continuous** state/action spaces without approximations.
- Requires full knowledge of **transition probabilities and reward functions** (i.e., model-based).

Comparison: Policy Iteration vs Value Iteration

Feature	Policy Iteration	Value Iteration
Process	Separate policy evaluation and improvement	Combines value updates and policy changes
Iterations	Fewer iterations, but each more complex	More iterations, simpler updates
Speed (small problems)	Generally faster	Slightly slower
Flexibility	More structured	More flexible, easier to implement

Applications of Policy Iteration

- **Robotics**: For optimal movement planning.
- **Game AI**: For strategy generation in board games or video games.
- **Autonomous systems**: Like self-driving cars or drones for decision-making.
- **Finance**: Investment strategies based on future reward predictions.
- **Operations research**: Optimal planning in logistics and supply chains.

6. Partially Observable Markov Decision Processes (POMDPs)

- A **Partially Observable Markov Decision Process (POMDP)** is an extension of a standard MDP where the agent **cannot directly observe the true state** of the environment.
- It models **decision-making under uncertainty**, both in the outcomes of actions and in state observations.
- Useful when the agent has **limited or noisy information** about the environment.

Key Components of a POMDP

A POMDP is defined as a 7-tuple:

$$(S, A, T, R, \Omega, O, \gamma)$$

- **S**: Set of all possible states.
- **A**: A Set of all possible actions.
- **T**: Transition probability function, $T(s, a, s') = P(s' \mid s, a)$, the probability of moving to state s' after taking action a in state s.
- **R**: Reward function, $R(s, a)$, the reward received after taking action a in state s.
- **Ω (Omega)**: Set of all possible observations.
- **O**: Observation probability function, $O(o \mid s', a)$, the probability of observing o given the new state s' and action a.
- **γ (gamma)**: Discount factor ($0 \le \gamma < 1$) that reduces future rewards' value.

Belief State

- Since the agent **cannot see the true state**, it maintains a **belief state**.
- A **belief state** is a probability distribution over all possible states (i.e., what the agent believes the current state might be).
- The belief is **updated** using:
  - The previous belief
  - The action taken
  - The observation received

Working of a POMDP

- The agent **starts with an initial belief** about the environment.
- At each time step:
  - It chooses an action based on the belief.
  - The environment transitions to a new state (unknown to the agent).
  - The agent receives an observation.
  - The agent **updates its belief** using the observation and transition probabilities.
  - This process continues to maximize the **expected cumulative reward**.

Challenges in Solving POMDPs

- **Belief space is continuous**, making exact solutions computationally expensive.
- Requires **approximation techniques** or **heuristic methods** in real-world applications.
- Finding optimal policies is **PSPACE-complete**, hence not always feasible for large problems.

Techniques to Solve POMDPs

- **Value Iteration in Belief Space**: Compute the optimal value function over the belief states.
- **Policy Trees**: Represent decision-making plans over time.
- **Point-Based Value Iteration (PBVI)**: Approximates value iteration by sampling belief points.
- **Monte Carlo Methods**: Used for approximate belief updating and policy evaluation.

Applications of POMDPs

- **Robotics**: Navigation and interaction in uncertain or partially observable environments.
- **Medical Diagnosis**: Making decisions when the patient's state is only partially known.
- **Natural Language Processing**: Dialogue systems where the user's intention is not fully observable.
- **Autonomous Vehicles**: Driving decisions with incomplete sensor data.
- **Game AI**: Opponent modelling when the full state is unknown.

Differences Between MDP and POMDP

Aspect	MDP	POMDP
Observability	Full observability of states	Partial observability of states
State Knowledge	The agent knows the exact current state	Agent maintains a belief over states
Complexity	Simpler to solve	More complex due to uncertainty
Representation	Uses the current state	Uses belief state (distribution)

UNIT 5: Reinforcement Learning

1. Introduction to Reinforcement Learning (RL)

Reinforcement Learning (RL) is a type of machine learning where an **agent** learns how to make decisions by interacting with an **environment**. The agent takes actions and receives feedback in the form of **rewards** or **penalties**. The goal is for the agent to learn a policy (strategy) that maximizes its cumulative reward over time.

Key Characteristics of Reinforcement Learning

1. **No Supervision:**
   - Unlike supervised learning (which uses labelled data), RL does not have labelled data.
   - The agent learns from **trial and error**, receiving feedback based on its actions.
2. **Delayed Rewards:**
   - Rewards may not be immediate; actions can affect future rewards.
   - The agent needs to plan and consider the long-term consequences of its actions.
3. **Exploration vs Exploitation:**
   - The agent faces a trade-off between **exploring** new actions (to discover more about the environment) and **exploiting** known actions (to maximize immediate reward).
4. **Sequential Decision Making:**
   - Decisions are made over time in a sequence of steps, where the outcome of an action can influence future actions and states.

Applications of Reinforcement Learning

1. **Games:**
    - RL is used in AI for games like **Chess, Go**, and **video games** (e.g., **AlphaGo**, **Dota 2**).
    - Example: In **AlphaGo**, RL enabled the agent to learn strategies by playing against itself.
2. **Robotics:**
    - RL is applied in robots for tasks such as **walking**, **grasping objects**, and **autonomous navigation**.
    - Example: **Boston Dynamics' robots** use RL to learn dynamic movements.
3. **Recommendation Systems:**
    - RL helps optimize the recommendation process in **personalized content** (e.g., Netflix, YouTube).
    - It adapts the system to provide better recommendations based on user feedback over time.
4. **Autonomous Vehicles:**
    - RL is used in self-driving cars to make decisions such as stopping at red lights or avoiding obstacles.

Key Components in Reinforcement Learning

1. **Agent:**
    - The learner or decision maker.
    - The agent interacts with the environment, taking actions based on its policy to maximize its rewards.
2. **Environment:**
    - The external system with which the agent interacts.
    - The environment reacts to the agent's actions and provides feedback (in the form of state and reward).
3. **State (S):**
    - A snapshot of the environment at any given time.
    - Represents all relevant information about the current situation.
    - Example: In a chess game, the state could represent the positions of all pieces on the board.

4. **Action (A):**
    - The decisions or moves the agent makes in the environment.
    - Each action changes the state and may lead to a different set of rewards.
    - Example: In chess, the action could be moving a piece.
5. **Reward (R):**
    - The feedback received by the agent after taking an action.
    - It is a scalar value, typically positive (for good actions) or negative (for bad actions).
    - The goal of RL is to maximize cumulative rewards.
    - Example: In a game, winning may result in a positive reward (e.g., +10), while losing may result in a negative reward (e.g., -10).
6. **Policy (π):**
    - A strategy or mapping from states to actions that the agent follows.
    - Defines the agent's behaviour and decision-making process.
    - **Deterministic Policy:** For every state, there's a single action to take.
    - **Stochastic Policy:** For every state, there's a probability distribution over possible actions.
7. **Utility/Value Function (V):**
    - Represents the expected reward or utility that the agent can achieve starting from a given state and following the policy.
    - It estimates how good a state is in terms of future rewards.
    - Mathematically, **V(s)** represents the expected sum of rewards starting from state **s** and following the policy.
8. **Q-value (Q):**
    - The expected future reward of taking action **A** in state **S** and following the best policy thereafter.
    - **Q-value** helps the agent decide which action to take based on the expected cumulative reward.
    - Mathematically, **Q(s, a)** represents the expected reward for action **a** in state **s**, considering the best future actions.
    - The Q-learning algorithm aims to learn the Q-value for every state-action pair.

RL Workflow

- **Step 1: Initialization**
  - The agent starts with an initial state (S) in the environment.
- **Step 2: Decision Making**
  - The agent takes an action (A) based on its policy.
- **Step 3: Environment Feedback**
  - The environment responds to the agent's action, transitioning to a new state (S') and providing a reward (R).
- **Step 4: Update Policy**
  - Based on the reward and new state, the agent updates its policy to improve future decision-making.
- **Step 5: Repeat**
  - This process repeats until the agent achieves the goal or converges to an optimal policy.

Types of Reinforcement Learning Algorithms

1. **Model-free RL:**
   - The agent doesn't build a model of the environment.
   - **Q-learning** is a model-free algorithm where the agent learns the value of state-action pairs directly from interactions.
2. **Model-based RL:**
   - The agent builds a model of the environment and uses it to predict future states and rewards.
3. **Value-based Methods:**
   - The agent learns to estimate the value function (e.g., Q-learning).
4. **Policy-based Methods:**
   - The agent learns a direct mapping from states to actions without using value functions.
   - **The REINFORCE algorithm** is an example.
5. **Actor-Critic Methods:**
   - Combines value-based and policy-based methods.

- The **actor** updates the policy, while the **critic** evaluates the policy by estimating the value function.

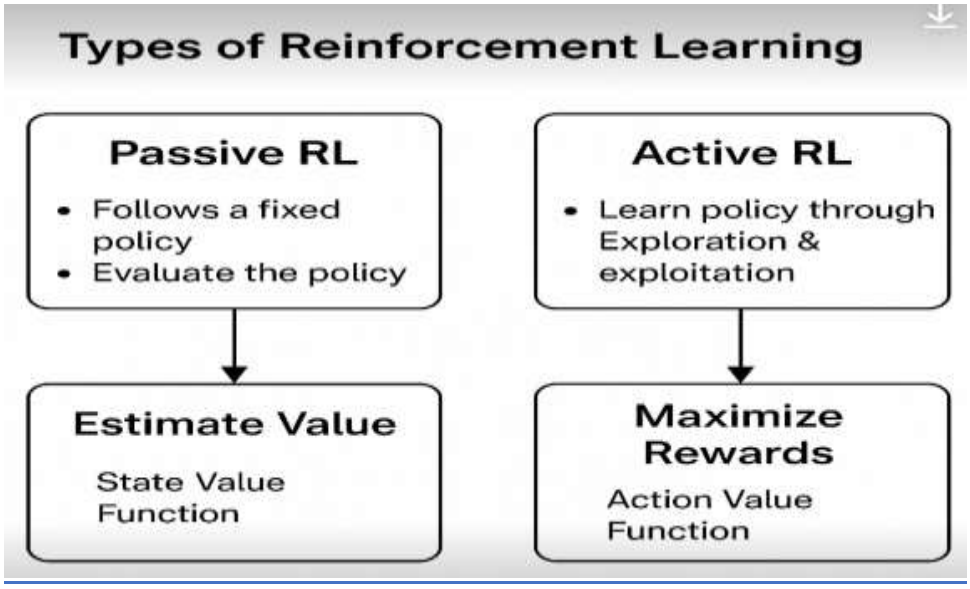
Exploration vs Exploitation Dilemma

- **Exploration:** The agent tries new actions to discover potentially better strategies.
- **Exploitation:** The agent chooses actions that have already been proven to yield high rewards.
- The agent needs to balance between exploring new actions and exploiting the known good actions.

Challenges in Reinforcement Learning

1. **Exploration-Exploitation Trade-off:**
   - Balancing between exploring new actions and exploiting known actions is a key challenge.
2. **Sparse Rewards:**
   - In some environments, rewards may be sparse or hard to obtain, making learning slower.
3. **Delayed Feedback:**
   - Delayed rewards make it harder for the agent to associate actions with rewards, especially when multiple actions contribute to a single reward.
4. **Scalability:**
   - RL can become computationally expensive and slow, especially in environments with large state and action spaces.

2. Types of Reinforcement Learning



a. Passive Reinforcement Learning

In **Passive Reinforcement Learning**, the agent follows a fixed, predefined policy and doesn't change or optimize the policy during learning. It focuses on **evaluating** the existing policy.

Key Features of Passive RL:

1. **Fixed Policy:**
   - The agent follows a fixed policy, meaning the actions it takes are determined by a set strategy, and it does not modify this policy during the learning process.
   - The policy is usually provided or predefined, and the agent is not trying to optimize it.
2. **Goal: Evaluate the Policy:**
   - The main aim is not to find a better policy, but to **evaluate** the effectiveness of the current policy.
   - The agent works on estimating the utility or **value** of each state under the fixed policy.
3. **Learning Process:**
   - The agent simply interacts with the environment, receives feedback (rewards or penalties), and updates its **value function**.
   - The agent's focus is on evaluating how good each state is in terms of future rewards without changing its behaviour.
4. **Policy Evaluation:**
   - The agent's objective is to estimate the **value function (V)**, which represents the expected cumulative reward starting from a given state, assuming it follows the fixed policy.
5. **No Exploration/Exploitation Dilemma:**
   - Since the policy is fixed and the agent is not changing it, there is no need to consider the exploration vs exploitation dilemma.

Applications of Passive RL:

- **Learning from Expert Demonstrations:** Passive RL can be used in situations where an expert demonstrates the correct behaviour (e.g., a teacher guiding a student). The agent then evaluates the policy based on the expert's actions.
- **Markov Decision Processes (MDPs):** Passive RL can be applied to evaluate policies in simple environments represented by MDPs where the environment's dynamics are known.

b. Active Reinforcement Learning

In **Active Reinforcement Learning**, the agent learns the best possible policy through direct interaction with the environment. It explores different actions, receives feedback, and adjusts its actions based on this feedback.

Key Features of Active RL:

1. **Dynamic Policy Learning:**
    - Unlike passive RL, the agent doesn't follow a fixed policy. Instead, it **actively learns** the best policy by interacting with the environment.
    - The goal is to improve and **optimize** the policy over time to maximize cumulative rewards.
2. **Exploration vs Exploitation:**
    - A central challenge in active RL is balancing **exploration** (trying out new actions to learn more about the environment) with **exploitation** (choosing the best-known action to maximize rewards).
    - The agent tries new actions to explore, while also relying on the best-known actions to exploit learned knowledge.
3. **Learning Through Trial and Error:**
    - The agent continuously tries different actions in various states and updates its knowledge of which actions lead to higher rewards.
    - Over time, the agent refines its policy based on past experiences.
4. **Reward Maximization:**
    - The agent's objective is to **maximize the cumulative reward** it receives over time by learning the best sequence of actions (policy).
    - The agent might update the policy after each action based on the received reward and the new state.
5. **Real-Time Decision Making:**
    - Active RL often requires **real-time decision-making**, where the agent adjusts its strategy continuously based on the most recent experiences and environment feedback.

Applications of Active RL:

- **Autonomous Systems:** Active RL is widely used in training autonomous systems like **self-driving cars**, **robotic control**, and **navigation** systems, where the system must learn the best policy through exploration and interaction with the environment.

- **Game AI:** RL is used in games (e.g., **AlphaGo**, **Dota 2**) to optimize strategies and improve decision-making in complex environments.

Key Differences Between Passive and Active RL

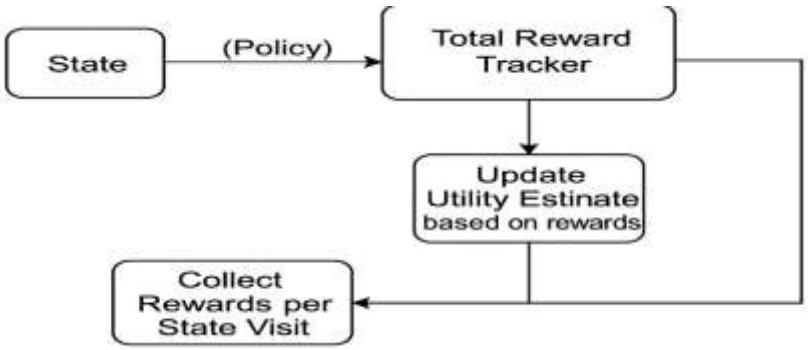
Aspect	Passive RL	Active RL
Policy	Follows a fixed policy	Learns and updates its policy
Goal	Evaluate the fixed policy	Learn the best policy
Exploration	Not required	Balances exploration and exploitation
Reward Optimization	Not a focus	Focused on maximizing cumulative rewards
Learning Process	State evaluation based on fixed policy	Trial and error to improve decision-making
Applications	Learning from expert demos, MDPs	Robotics, Game AI, Autonomous Vehicles

3. Direct Utility Estimation (DUE)

Direct Utility Estimation (DUE) is a technique used in **Passive Reinforcement Learning (RL)**.In Passive RL, the agent is not actively learning from its experiences to improve the policy. Instead, it follows a fixed policy and estimates the utility (or value) of each state based on the rewards received during multiple episodes of interaction with the environment.

Process of DUE

- The agent interacts with the environment, following a fixed policy.
- Over multiple episodes, the agent collects rewards by visiting different states.
- **Utility of a state** is then estimated by averaging the rewards obtained from that state, which gives a measure of the desirability or value of being in that state.



Mathematical Formula

The utility **U(s)** of a state **s** is calculated as:

$$U(s) = \frac{1}{N(s)} \sum_{i=1}^{N(s)} \text{TotalReward}(s)$$

Where:

- **U(s)**: Utility of state **s** (how valuable the state is for the agent).
- **N(s)**: Number of times state **s** was visited.
- **TotalReward(s)**: Sum of the rewards received when the agent visits state **s**.

1. **N(s)** represents how many times the agent visits a particular state **s** during the episodes.
2. **Total Reward(s)** is the accumulated sum of all rewards the agent receives every time it is in state **s**.
3. By dividing the total rewards by the number of visits, we get the **average reward** for that state, which serves as the **utility**.

Advantages of DUE

- **Simplicity**: Direct Utility Estimation is relatively simple to implement and understand.
- **No Need for Model**: It doesn't require knowledge about the environment's transition model, which is useful when the environment is unknown or difficult to model.
- **Passive Learning**: Works well when the agent has a fixed policy and is not trying to learn from its experience actively.

Disadvantages of DUE

- **Inefficiency**: It requires many visits to each state to estimate the utility accurately. This is inefficient, especially in large environments.
- **Slow Convergence**: Since the utility estimate is based on averages over multiple trials, it can take a long time to converge to an accurate estimate, particularly in large state spaces.
- **Limited to Passive RL**: It can only be used in scenarios where the agent is following a fixed policy (not in active learning, where the policy is updated based on the agent's experience).

Steps Involved in DUE

1. **Initialize Utility Estimates**: Begin by assigning a random or initial utility to each state.

2. **Follow the Fixed Policy**: The agent follows a fixed policy to interact with the environment and reaches various states.
3. **Track Rewards**: For each state visited, the agent tracks the rewards accumulated.
4. **Update Utility Estimates**: After multiple episodes, the utility for each state is updated by calculating the average reward received from that state.
5. **Repeat**: The process repeats until the utility values stabilize or converge.

Real-World Use Cases

- **Inventory Management**: Estimating the utility of different stock levels in a warehouse, where each state corresponds to a specific inventory level, and the reward might be based on the costs or profits associated with that level.
- **Game AI**: In some simple games, DUE can be used to estimate the utility of different game states by following a fixed strategy and calculating the average rewards from each state over several plays.

4. Adaptive Dynamic Programming (ADP)

Adaptive Dynamic Programming (ADP) is an approach used in **passive Reinforcement Learning (RL)**. In ADP, the **agent** builds a model of the environment and uses this model to **calculate utilities** for each state in the environment. These utilities help in evaluating the best possible actions to take, without directly interacting with the environment for every decision.

Model Components in ADP:

1. **Transition Probabilities (P(s' | s, a)):**
    - Transition probabilities represent the likelihood of moving from one state to another, given a specific action.
    - It is represented as **P(s' | s, a)**, where:
        - **s**: Current state
        - **s'**: Next state
        - **a**: Action taken
    - Example: If the agent is in state **A** and takes action **X**, what is the probability of transitioning to state **B**?
2. **Reward Function (R(s)):**
    - The reward function defines the immediate reward the agent gets for being in a particular state.
    - **R(s)** represents the reward associated with state **s**.

- Example: If the agent is in state **A**, then the reward for being in **A** could be **R(A)**.

3. **Utility (U(s)):**
   - **Utility (U(s))** refers to the value or desirability of being in a state **s**. It's the expected future reward the agent can achieve starting from that state.
   - ADP aims to estimate the utilities for all possible states in the environment.
4. **Discount Factor (γ):**
   - The **discount factor γ** (gamma) determines how much importance the agent gives to future rewards compared to immediate rewards.
   - If **γ = 1**, future rewards are considered as important as immediate rewards.
   - If **γ < 1**, future rewards are discounted, meaning the agent prefers immediate rewards over long-term ones.

Advantages of ADP:

1. **Faster than Direct Utility Estimation:**
   - ADP is more efficient than directly estimating the utility of each state by exploring the environment extensively.
   - It uses a model of the environment, which speeds up the process of utility calculation.
2. **More Accurate Utility Estimates:**
   - By using the transition probabilities and reward functions, ADP provides more accurate estimates of utility compared to trial-and-error methods.
3. **Effective in Large State Spaces:**
   - ADP can handle large state spaces more effectively as it uses the environment model to predict utilities, rather than exploring all states individually.

Disadvantages of ADP:

1. **Requires Storing and Updating the Environment Model:**
   - The agent must maintain a model of the environment, which includes transition probabilities and reward functions. This can be complex and require substantial memory, especially in large environments.
2. **Works Best in Stable and Predictable Environments:**
   - ADP assumes that the environment is stable and predictable. In highly dynamic or uncertain environments, the utility estimates may become less reliable.

3. **Complexity in Building the Model:**
    - The agent needs to learn the transition probabilities and reward function accurately, which can be a challenging task in certain environments.

Applications of ADP:

- **Robotics**: ADP can be used for motion planning, where the robot learns to calculate optimal paths using transition models.
- **Game Playing**: In games, the agent can learn optimal strategies by using ADP to evaluate different states and actions.
- **Control Systems**: ADP can be used in systems where control actions must be taken based on an estimated model of the system's behaviour.

5. Temporal Difference Learning (TD)

Temporal Difference Learning (TD) is a reinforcement learning technique that **combines direct utility estimation and Adaptive Dynamic Programming (ADP)**. Unlike ADP, **TD learning does not require a model** of the environment (transition probabilities or reward function). Instead, TD learning learns directly from **experiences** as they occur, using **temporal difference errors** to update the utility estimates. TD learning is particularly useful in situations where an agent cannot fully model the environment or when **real-time learning** is required.

TD Learning Update Rule:

The key to TD learning is the **temporal difference error**, which measures the difference between the current utility estimate and the next utility estimate. This error is used to update the utility of the current state.

The update rule in TD learning is as follows:

Where:

- **U(s)**: Current utility estimate of state **s**.
- **R(s)**: Reward received after taking an action in state **s**.
- **U(s')**: Estimated utility of the next state **s'**.
- **α (alpha)**: **Learning rate**, which controls how much the new information (update) should affect the old utility value.
- **γ (gamma)**: **Discount factor**, which determines the importance of future rewards.
- **Temporal Difference Error**: R(s)+γU(s′)−U(s)R(s) + \gamma U(s') - U(s)R(s)+γU(s′)−U(s), which measures the difference between the current estimate and the updated estimate.

How TD Learning Works:

1. **Initial Setup**:
   - Start with an initial estimate of the utility for all states, typically set to zero or some arbitrary value.
2. **Experience**:
   - The agent interacts with the environment, takes actions, and moves from one state to another.
   - For each state **s** encountered, the agent receives a reward **R(s)** and moves to the next state **s'**.
3. **Utility Update**:
   - The agent updates the utility estimate for the current state **s** using the TD update rule:
   - This update uses the reward **R(s)** and the estimated utility of the next state **U(s')**.
4. **Iterative Process**:
   - This process is repeated over time as the agent continues to explore the environment, updating the utility estimates after each transition.

Advantages of Temporal Difference (TD) Learning:

1. **Learns from Experience**
   - TD does not need a model of the environment; it learns directly from interactions.
2. **Online Learning**
   - It updates utility values while the agent is still running, supporting real-time learning.
3. **No Need for Environment Model**
   - Unlike ADP, TD doesn't require knowledge of transition probabilities or rewards.
4. **Scalable to Large Environments**
   - TD works well even when the state space is large or continuous.
5. **Useful in Real-Time Systems**
   - Suitable for areas like robotics and games where quick decisions are needed.

Disadvantages of TD Learning:

1. **Learning Instability**
   - If the learning rate (α) is too high or too low, learning may become unstable or very slow.
2. **Sensitive to Parameters**
   - TD depends heavily on tuning learning rate (α) and discount factor (γ).
3. **Slow in Complex Environments**
   - Can take time to converge, especially in large or unpredictable settings.
4. **Affected by Noisy Rewards**
   - If rewards vary too much or are rare, learning can become less effective.

Applications of TD Learning:

- **Gaming** – Improves strategy in games like chess by learning from positions.
- **Robotics** – Helps robots learn movement based on feedback.
- **Self-Driving Cars** – Learns better driving actions using real-time rewards.
- **Healthcare** – Assists in learning treatment plans based on patient feedback.

6. Q-Learning

Q-learning is a type of **Reinforcement Learning** algorithm. It is **model-free** (does not need transition probabilities or reward functions). It is **off-policy**, meaning it learns the optimal action policy regardless of the agent's current actions. The goal is to learn the **optimal policy** by estimating **Q-values** (action-value function).

What is Q-Value?

- A **Q-value** is the expected total reward for taking a particular action **a** in a state **s**, and then following the **best possible policy** afterward.
- It is written as: **Q(s , a)**

Q-Learning Update Rule

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma \cdot \max Q(s',a') - Q(s,a)]$$

Where:

- **Q(s, a)**: Current Q-value for state-action pair (s, a)
- **R**: Reward received after taking action **a** in state **s**

- **s'**: New (next) state
- **max Q(s', a')**: Maximum Q-value of the next state over all possible actions
- **α (alpha)**: Learning rate (0 < α ≤ 1)
- **γ (gamma)**: Discount factor (0 ≤ γ < 1), represents importance of future rewards

Final Policy from Q-Learning

- The best policy is formed by choosing the action with the highest Q-value for a given state:

$$\pi(s) = \arg\max_a Q(s, a)$$

This means In each state, choose the action that has the **maximum Q-value**.

Advantages of Q-Learning

1. **Model-Free Learning**
   - No need for the agent to know transition probabilities or reward functions.
2. **Learns from Raw Experience**
   - The agent learns by exploring the environment through trial and error.
3. **Finds Optimal Policy**
   - With enough exploration and training, Q-learning converges to the best policy.
4. **Works in Unknown or Complex Environments**
   - Can be used in dynamic, partially known, or large environments.
5. **Can Be Combined with Neural Networks**
   - Forms the base for **Deep Q-Learning**, used in real-time applications like games and robotics.

Disadvantages of Q-Learning

1. **Slow Convergence**
   - Needs a large number of trials and exploration to converge to the optimal policy.
2. **Sensitive to Hyperparameters**
   - Learning rate (α) and discount factor (γ) must be chosen carefully. Bad values can lead to poor learning.
3. **Exploration vs Exploitation Issue**

- The agent must balance between trying new actions (exploration) and using known best actions (exploitation).

4. **Large State Spaces**
    - For environments with too many states or actions, storing and updating the Q-table becomes difficult.

Applications of Q-Learning

- **Game Playing** – e.g., learning to play board/video games.
- **Robotics** – decision making in uncertain environments.
- **Autonomous Vehicles** – for route planning and obstacle avoidance.
- **Finance** – making investment decisions based on state-action rewards.
- **Recommendation Systems** – learning which product to suggest based on user behaviour.