CSPC3003

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

MODULE-I

Course Objectives:

- To learn the concepts of Artificial Intelligence
- To learn the methods of solving problems using Artificial Intelligence
- To introduce the concepts of Expert Systems and machine learning

Why AI?

- **Automates tasks** and saves time.
- **Improves accuracy** and reduces errors.
- **Helps in smart decision-making** using data.
- **Personalizes experiences** (e.g., recommendations).
- **Solves complex problems** faster than humans.
- **Drives innovation** in every industry.

In short, AI makes machines smart, work faster, and improve our daily lives.

Several computer systems have been built that can perform tasks such as these. Also there are specially developed computers systems that can diagnose disease, solve quadratic equations, understand human speech and natural language text.

We can say that all such systems possess certain degree of artificial intelligence.

The central point of all such activities and systems is that "**How to think**" OR rather "**How to make system think**". The process of thinking has various steps like **perceive**, **understand**, **predict** and **manipulate** a world that is made up of tiny complex things or situations.

What is AI?

- *"Artificial Intelligence (AI) is the simulation of human intelligence in machines."* These machines are programmed to think, learn, and solve problems like humans, enabling them to perform tasks that typically require human intelligence.
- *"AI is the design and development of systems that can mimic human intelligence."*
- "The field of AI not just attempts to understand but also it builds intelligent entities."
- "Al may be defined as the branch of computer science that is concerned with the automation of intelligent behaviour". (Luger 1993)
- "Systems that thinks like human".
- "Systems that act like humans".

- *“Systems that think rationally”.*
- “The exciting new effort to make computers think machines with minds, in the full and literal sense”. (Hallgeland - 1985)
- “The automation of activities that we associate with human thinking, activities such as decision making, problem solving, learning”. (Bellman 1978)
- "The art of creating machines that perform functions that require intelligence, when performed by people". (Kurzweil - 1990)
- The study of mental faculties through the use of computational models. (Charniak and McDermott – 1985)
- "The study of the computations that make it possible to perceive, reason and act". (Winston 1992)
- ."Computational intelligence is the study of the design of intelligent agents". (Poole et al 1998)
- "AI is concerned with intelligent behaviour in artifacts". (Nilsson - 1998)

Differentiate between Conventional Problem Solving Techniques with Artificial Intelligence Techniques

Features	AI Techniques/Problems	Conventional Techniques/Software
Processing Type	Symbolic	Numeric
Technique Used	Heuristic Search	Algorithmic Search
Solution Steps	Not explicit	Precise
Answer Sought	Satisfactory	Optimal
Control/data Separation	Separate	Intermingled
Knowledge	Imprecise	Precise
Modification	Frequent	Rare
Involve	Large knowledge base	Large database
Process	Inferential	Repetitive

❖ These definitions vary along two main dimensions.

- First dimension is the thought process and reasoning and
- second dimension is the behaviour of the machine.

Some definitions are based on comparisons to human performance where as remaining definitions measure success against an ideal concept of intelligence, which we call rationality.

A system is rational if it does the "right thing" given what it knows.

Historically, there are four approaches that are followed in AI. These four approaches are:

- Thinking Humanly
- Acting Humanly
- Thinking Rationally
- Acting Rationally

❖ Thinking Humanly

- AI systems that attempt to **mimic human thought processes**.
- Focuses on **how humans think**, including emotions, biases, and psychological processes.
- To make machines think like humans (simulate human brain activities).
- Uses Cognitive Science + Psychology.
- **Example:**
  - AI chatbots designed to respond in a human-like manner.
  - Cognitive models of problem-solving that imitate human reasoning steps.
  - Humanoid robots (e.g., Sophia by Hanson Robotics),
  - AI receptionist/concierge robots,
  - AI in games that mimic player behaviour,
  - Voice assistants like Alexa, Siri (when talking casually)

❖ Acting Humanly

- AI that **acts like a human being** (focus on human-like behavior).
- If the system's actions are indistinguishable from a human's, it is considered intelligent.
- **Turing Test –** proposed by Alan Turing in 1950.
- **Characteristics:**
  - Natural Language Processing (to communicate).
  - Knowledge Representation (to store information).
  - Automated Reasoning (to answer questions/make decisions).
  - Machine Learning (to adapt and improve).
- **Example:**
  - Chatbots (like Siri, Alexa) that converse like humans.
  - Humanoid robots mimicking human interaction.
  - ChatGPT (understands & responds like a person),
  - IBM Watson (uses human-like reasoning),
  - Replika (emotional conversation),
  - AI in language translation (e.g., Google Translate)

❖ Thinking Rationally

- AI systems that aim to **think logically and make correct inferences** based on rules and logic.
- Focuses on **what is logically correct** and guarantees an optimal solution.
- To reason according to **formal logic and mathematics**, irrespective of how humans actually think.
- Uses Logic, Mathematics, Philosophy.
- **Example:**
  - Expert systems using logical rules (If-Then statements).
  - AI solving a Sudoku puzzle using logical deduction instead of guessing like humans might.
  - Expert systems (e.g., MYCIN for medical diagnosis),
  - Prolog-based AI,
  - Rule-based fraud detection systems,
  - Theorem provers

❖ Acting Rationally

- AI that acts to achieve the **best outcome** (or the most optimal result) based on logic and environment.
- Rational Agent approach – system acts to maximize performance measure.
- Uses Rationality, Decision Theory, Game Theory.
- **Characteristics:**
  - Always chooses the **best possible action**.
  - Not about imitating humans, but about **acting optimally**.
- **Example:**
  - Autonomous vehicles choosing safest and fastest routes.
  - Chess AI (like Deep Blue, AlphaZero) making the best possible move.
  - Stock trading bots,
  - Smart thermostats (e.g., Nest learning optimal temperature settings

Approach	Goal	Resembles	Focus	Example
Think Humanly	Make machines think like humans	Human brain	Cognitive processes (reasoning, memory, learning)	ChatGPT (understands and replies like a human) Replika chatbot
Act Humanly	Make machines behave like humans	Human behaviour	Interaction, natural responses	Humanoid robot, virtual assistant (e.g., Alexa, Siri)
Think Rationally	Make machines reason using logic	Ideal logic solver	Logical rules, inference, deduction	Expert systems (e.g., MYCIN), theorem provers
Act Rationally	Make machines take the best action to reach goals	Goal-driven agent	Decision-making, outcomes	Self-driving car, AlphaGo, intelligent agent systems

Thinking = What's happening **inside the AI's mind** (reasoning)

Acting = What the AI **does** in the world (behavior)

Game: “Which AI Am I?” – Categorize AI Systems

Classify different AI systems into one of the **four approaches to AI**.

- AI Music Composer
- Automated Theorem Prover
- Customer Service Robot in a Mall
- Stock Trading AI
- Handwriting Recognition System
- GPS Route Optimization System
- Virtual Shopping Assistant Avatar
- Medical Diagnosis AI

Game: “Which AI Am I?” – Categorize AI Systems

- AI Music Composer: Creates songs based on human styles, sometimes even with “errors” like a human composer. → ***Thinking Humanly***
- Automated Theorem Prover: Uses logic rules to prove/disprove mathematical theorems. → ***Thinking Rationally***
- Customer Service Robot in a Mall: Interacts with visitors, greets them, answers basic FAQs in natural language. → ***Acting Humanly***
- Stock Trading AI: Analyzes market data and makes the most profitable trades. → ***Acting Rationally***
- Handwriting Recognition System: Learns to read messy human handwriting by imitating how humans adapt. → ***Thinking Humanly***
- GPS Route Optimization System: Finds the shortest/fastest route mathematically. → ***Thinking Rationally***
- Virtual Shopping Assistant Avatar: Talks with customers online with human-like expressions and gestures. → ***Acting Humanly***
- Medical Diagnosis AI: Suggests the most likely disease and best treatment plan based on evidence. → ***Acting Rationally***

Goals of Artificial Intelligence

- **Automation** – Perform repetitive, laborious, or risky tasks.
- **Mimic Human Intelligence** – Think, reason, and learn like humans.
- **Problem Solving** – Analyze situations and make better decisions.
- **Knowledge Representation** – Store and use knowledge effectively.
- **Learning from Data** – Improve performance using past experience.
- **Natural Interaction** – Communicate via speech, text, or vision.
- **Perception of Environment** – Sense and understand surroundings.
- **Creativity** – Generate new ideas, art, designs, or solutions.
- **Human Assistance** – Enhance human capabilities in daily life.

The Foundations of AI?

The foundation of AI consists of the core principles, disciplines, and techniques that support the development of intelligent systems. These include:

- **Mathematics**
  - **Logic:** For reasoning and decision-making.
  - **Probability & Statistics:** For handling uncertainty and data analysis.
  - **Linear Algebra & Calculus:** For machine learning and deep learning algorithms.
- **Computer Science**
  - **Algorithms & Data Structures:** For efficient processing and learning.
  - **Programming Languages:** Like Python, Java, Lisp, etc., for implementing AI models.
  - **Search & Optimization:** To find solutions in large problem spaces (e.g., A* algorithm).
- **Neuroscience**
  - Studies the human brain structure and function to develop **neural networks**.

The Foundations of AI?

- **Cognitive Science**
  - Understands **how humans think and learn** to model AI behavior accordingly.
  - Inspiration for systems that can mimic human intelligence.
- **Linguistics**
  - Helps in building **Natural Language Processing (NLP)** systems to understand and generate human language.
- **Philosophy**
  - Defines the **nature of intelligence, reasoning, ethics, and consciousness**.
- **Control Theory & Cybernetics**
  - Helps in **robotics and intelligent agents** by maintaining system stability and responses.

Strong AI vs Weak AI

Weak AI (Narrow AI)

Weak AI is designed to perform a specific task. It does not possess real intelligence or consciousness, but it appears intelligent in its function.

Characteristics:

- Task-specific
- No self-awareness
- Cannot generalize beyond its training
- Based on predefined rules or machine learning

Examples:

- **Siri, Alexa** – Voice assistants that follow commands.
- **Google Translate** – Translates text but doesn't understand meaning like a human.
- **Face recognition systems, spam filters, recommender systems (Netflix, YouTube).**

Strong AI (General AI)

Strong AI refers to machines that have human-level intelligence and can perform any intellectual task that a human can do.

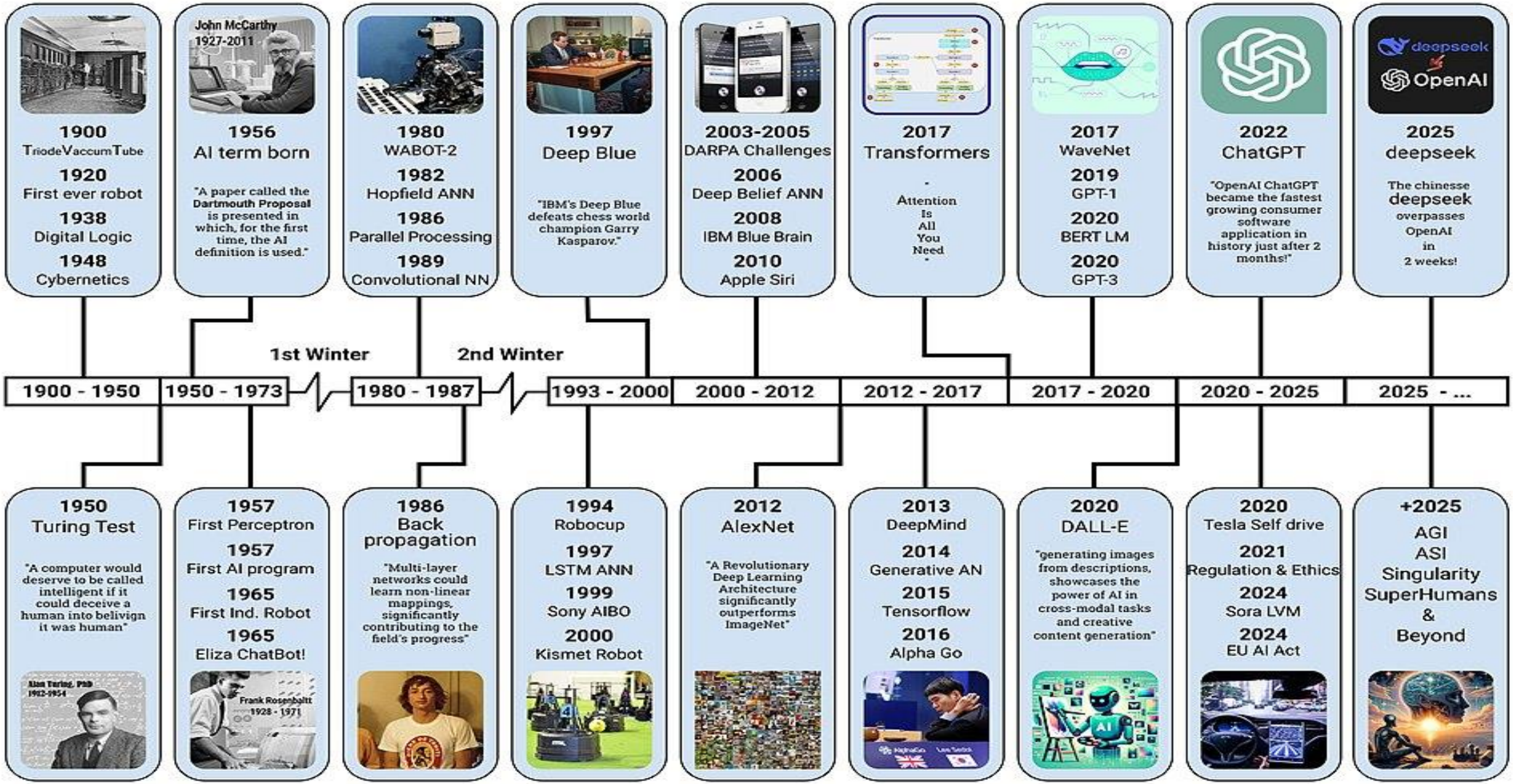
Characteristics:

- Fully autonomous
- Can reason, learn, and make decisions like a human
- Has self-awareness and consciousness (hypothetically)

Examples:

- No real examples yet (as of 2025) – Strong AI is still theoretical.
- Often portrayed in sci-fi movies (e.g., Jarvis from Iron Man, Data from Star Trek, Sophia the robot is a partial example but not true strong AI).

History of AI

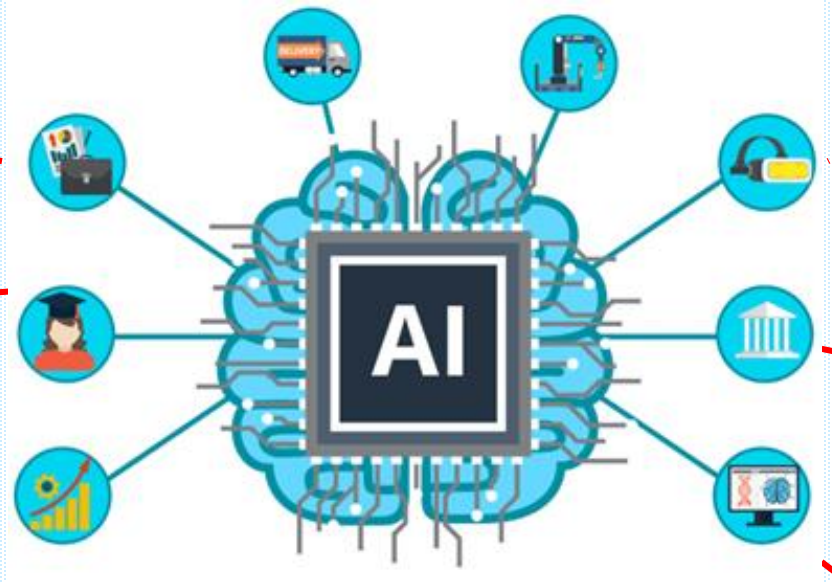


Applications of AI

TRANSPORTATION
- Self-driving cars (autonomous vehicles)
- Route optimization (e.g., Google Maps, Uber)
- Traffic prediction and control
- Smart logistics and delivery drones

MANUFACTURING
- Predictive maintenance of machines
- AI-based quality control
- Robotic process automation
- Smart factories and Industry 4.0

BUSINESS
- Product recommendations (e.g., Amazon, Flipkart)
- Customer service chatbots
- Inventory management
- Virtual shopping assistants



ENTERTAINMENT
- Content recommendations (Netflix, YouTube)
- Deepfake technology
- Game AI for smart opponents
- Virtual reality (VR) and augmented reality (AR)

EDUCATION
- Smart tutoring systems (adaptive learning)
- Automated grading
- AI-based personalized learning apps
- Virtual classrooms and chatbots for student support

GOVERNMENT
- Law Enforcement – Crime prediction, facial recognition.
- E-Governance – Chatbots, document automation, grievance redressal.
- Smart Cities – Waste, water, energy, and urban planning.

FINANCE
- Fraud detection
- Credit scoring and risk assessment
- Algorithmic trading
- Robo-advisors for investment management

HEALTHCARE
- Disease diagnosis (e.g., cancer, diabetes)
- AI-based medical imaging (X-ray, MRI analysis)
- Virtual health assistants and chatbots
- Drug discovery and personalized treatment

Applications of AI

Predictive Analytics

Predictive analytics uses AI/ML to analyze historical data and predict future events or outcomes, helping in smarter decision-making.

❖ Industry (Manufacturing)

- AI analyzes sensor data from machines to predict when a part might fail.
- **Example:** GE and Siemens use predictive maintenance AI in factories to reduce downtime.
- **Benefit:** Saves costs, avoids breakdowns, and improves efficiency.

❖ Market (Business & Finance)

- AI studies customer behavior, market trends, and transaction history to predict demand or detect fraud.
- **Example:** Banks use predictive AI to flag suspicious transactions before fraud occurs.
- **Benefit:** Increases revenue (better demand forecasts) and prevents financial losses.

❖ Medical (Healthcare)

- AI analyzes patient data, genetics, and medical history to predict the risk of diseases.
- **Example:** IBM Watson Health predicts cancer treatment responses.
- **Benefit:** Early disease detection, personalized medicine, and better patient outcomes.

Applications of AI

Predictive Analytics

❖ Transport

- AI predicts traffic congestion and adjusts traffic signals in real time.
- **Example:** Google Maps uses AI to predict travel times and suggest optimal routes.
- **Benefit:** Saves travel time, reduces fuel use, and improves city traffic flow.

❖ Education

- AI predicts student performance, risk of dropouts, and areas needing improvement.
- **Example:** Learning platforms (like Coursera, Byju's) use AI to recommend personalized learning paths.
- **Benefit:** Improves learning outcomes and supports struggling students.

❖ Environment

- AI models predict climate change impacts, natural disasters, or energy consumption.
- **Example:** AI predicts floods, cyclones, and wildfires using satellite and sensor data.
- **Benefit:** Helps in disaster preparedness and resource optimization.

Applications of AI

Natural Language Processing (NLP)

AI that enables machines to understand, process, and generate human language.

Examples:

- Google Translate (language translation).
- ChatGPT (human-like conversations).
- Grammarly (grammar & writing assistant).

Use: Customer service chatbots, document summarization, speech-to-text, sentiment analysis.

Computer Vision

AI that allows computers to interpret and process visual data (images, videos).

Examples:

- Facial recognition systems (used in smartphones & airports).
- Quality inspection in factories.
- Medical image analysis (detecting tumors in X-rays).

Use: Healthcare, surveillance, self-driving cars, manufacturing quality control.

Applications of AI

Robotics

AI-powered robots that can perform tasks autonomously.

Examples:

- Warehouse robots in Amazon for packaging & delivery.
- Surgical robots in hospitals.
- Boston Dynamics robots for logistics & defense.

Use: Manufacturing, healthcare, agriculture, defense.

Game Playing AI

AI that learns strategies and plays games at or above human level.

Examples:

- DeepMind's AlphaGo defeating world champions in Go.
- Chess engines like Stockfish and AlphaZero.

Use: Research in problem-solving, reinforcement learning, and strategy optimization.

Applications of AI

Autonomous Vehicles

AI that enables cars, drones, or ships to drive/fly without human input.

Examples:

- Tesla's self-driving cars.
- Waymo's autonomous taxis.

Use: Transport, delivery, military, space exploration.

Expert Systems

AI systems that mimic human experts by using **knowledge bases + inference engines**.

Examples:

- MYCIN (medical diagnosis).
- DENDRAL (chemical analysis).

Use: Medicine, troubleshooting, engineering, law.

Human Vs Machine

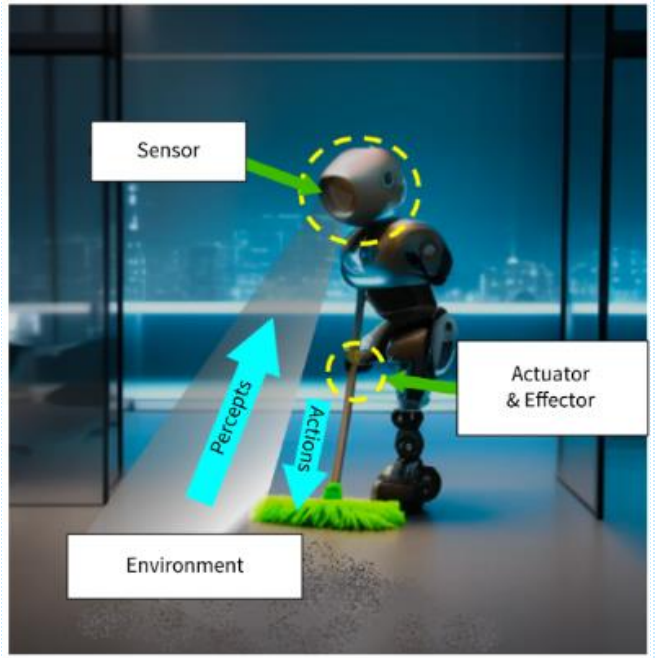
Feature	Human	Machine (AI)
Intelligence	Natural, emotional, creative	Artificial, logical, data-driven
Learning	Based on experience, emotions	Based on data and algorithms
Decision-making	Sometimes emotional or biased	Logical, data-based
Speed	Slower	Much faster
Memory	Limited and prone to forgetting	Large, fast, accurate
Creativity	High (art, innovation)	Limited or imitative
Fatigue	Gets tired	No fatigue
Adaptability	Highly adaptable	Limited to training and updates
Judgment	Moral, ethical reasoning	Lacks true judgment or ethics

Agents

- An **agent** is **anything that can perceive its environment through sensors and act upon that environment using actuators** to achieve a specific goal.
- An **agent** is a system that **observes** (perceives) and **acts** intelligently in an environment to maximize its success.

- For example consider human as agent. Human has eyes, ears and other organs which are sensors. Hands, legs, mouth and other body part work as actuators.

- Lets consider another example of agent Robot. A Robotic agent might have cameras, infrared rangefinders as sensors. Robot can have various motors for actuators.

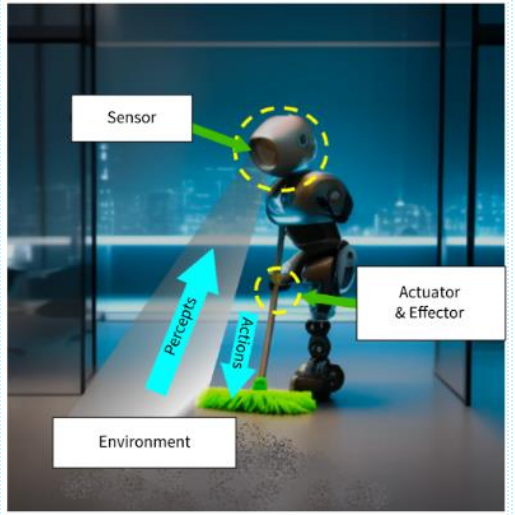


Types of Agent

Agent can be of 3 types:

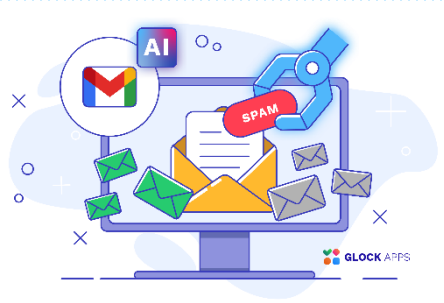
- **Human Agent**

  A human agent has **eyes**, **ears**, and other organs which work for **sensors**, and **hand**, **legs**, etc. work as **actuators**.



- **Robotic Agent**

  A robotic agent can have **camera**, **infrared range**, **NLP (Natural Language Processing)**, etc. as **sensor**, and various motors act as **actuator**.



- **Software Agent**

  Software agent can have **key stroke**, **file**, etc. as sensor input, and **display** output content act as sensor output and display output.

Agents

More examples of agent

- **Software agent:** A **software agent** is a **program** that performs tasks **autonomously** on behalf of a user or another program. It operates in a **digital environment**, senses changes (inputs), and acts to achieve specific goals.

  Sensors: Keystrokes, file contents and network packets.

  Actuator: Screen, writing files, network packet.

  Examples:

  - **Web crawlers** – Automatically search and index web pages (e.g., Googlebot)
  - **Spam filters** – Monitor and classify emails
  - **Chatbots** – Interact with users (e.g., customer support)
  - **Personal assistants** – Like Siri, Google Assistant
  - **Trading bots** – Make decisions in stock markets

- **Internet shopping agent:** An **Internet Shopping Agent** is a smart online assistant that helps users **compare and shop** efficiently by finding the **best products and prices** from multiple sources.

  Sensors: HTML, DHTML, pages (text graphics script)

  Actuators: Forms, display to user, follow URL.

  - **Examples: Google Shopping, Shopzilla, PriceGrabber, Trivago** (for hotel comparison), **Policybazaar** (for insurance comparison)

How do AI Agents work?

AI agents work by following a **sense–think–act** cycle. They continuously observe their environment, make intelligent decisions, and then take actions to achieve specific goals.

❖ Step-by-Step

1. **Perception (Sense)**
   - The agent **receives input** from the environment through **sensors** (or input functions).
   - Example: A robot senses temperature or obstacles; a chatbot receives text input.

2. **Decision-Making (Think)**
   - The agent **processes** the input using **rules, logic, or learning algorithms**.
   - It **evaluates the current situation**, predicts outcomes, and **chooses the best action**.

3. **Action (Act)**
   - The agent uses **actuators** (or output functions) to **interact with the environment**.
   - Example: A robot moves; a chatbot replies; a game character defends or attacks.

4. **Learning (Optional but powerful)**
   - Intelligent agents **learn from experience** (feedback) to **improve future performance**.
   - Learning agents can adapt to new situations over time.

AI Agent's Architectural Components

The **architecture of an agent** refers to the **design and structure** that defines how an AI agent **senses**, **thinks**, **acts**, and optionally **learns** in its environment.

❖ Components of an AI Agent

- **Sensors (Perception)**
  - Used to **observe** the environment.
  - Input devices or data sources.
  - Example: Camera, microphone, keyboard (for physical agents); input data stream (for software agents).

- **Percept Sequence**
  - The complete history of all percepts (inputs) received by the agent.
  - Helps in decision-making based on past inputs.

- **Agent Program / Decision-making Unit**
  - The "brain" of the agent.
  - Processes percepts and chooses an appropriate action.
  - Can use logic, rules, machine learning, or search algorithms.

AI Agent's Architectural Components

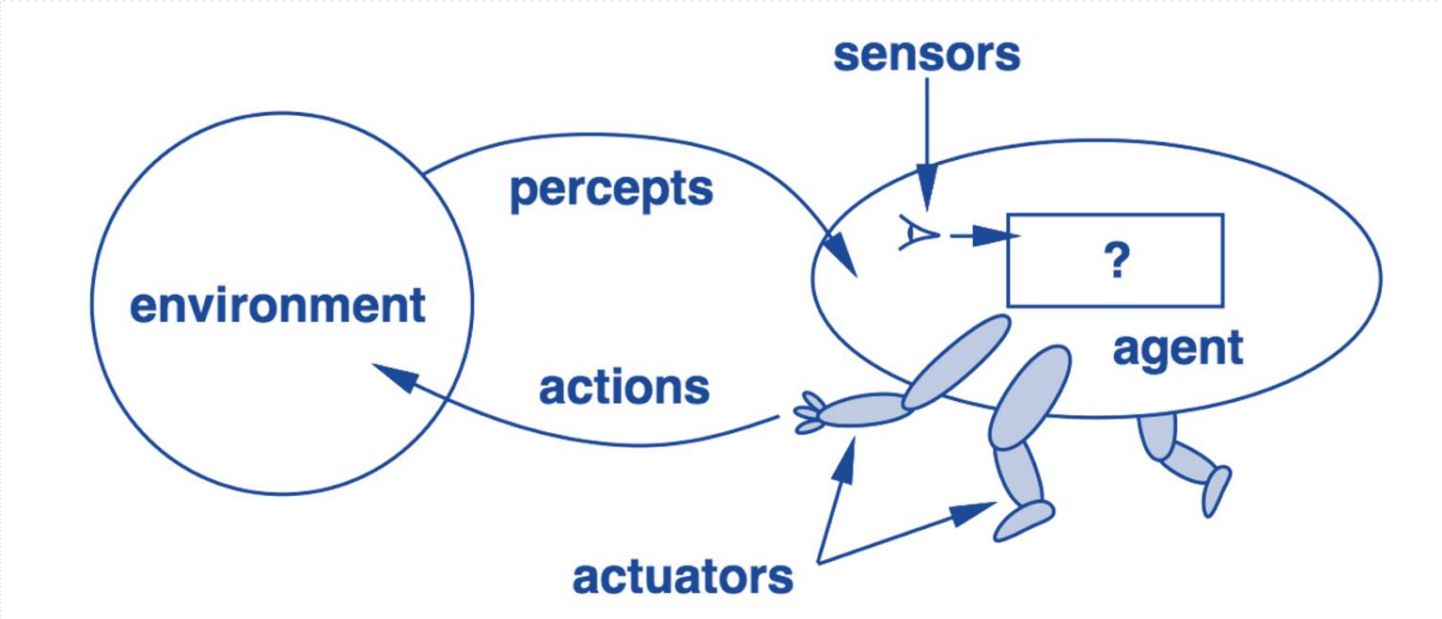
❖ Components of an AI Agent

- **Actuators (Actions)**
  - Allow the agent to **act upon the environment**.
  - Example: Motors, display screen, robotic arms (for physical agents); output APIs (for software agents).

- **Environment**
  - The world in which the agent operates.
  - Could be physical (robots) or digital (web, software).

How It Works (Cycle)

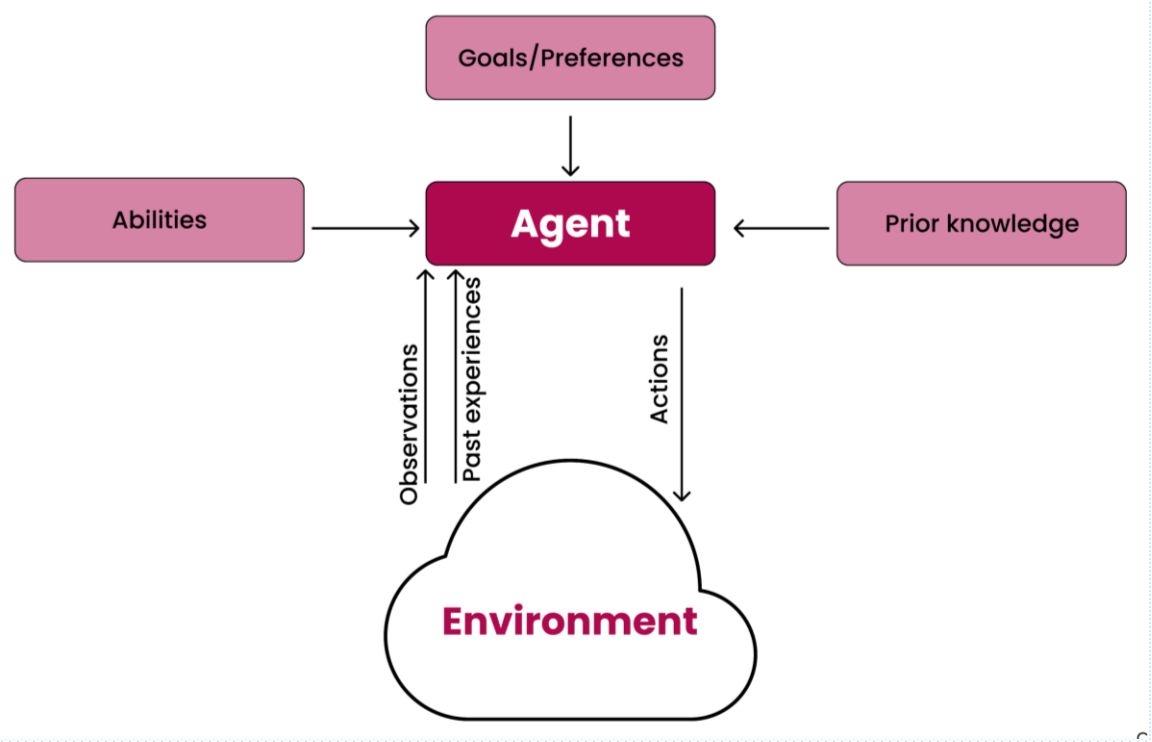
Sensors → Percept → Agent Program → Action → Actuators → Affect Environment



AI Agents Architectural Components

❖ Some other Components of an AI Agent

- **Performance Measure:** Evaluates how well the agent is performing
- **Internal State:** The agent's memory of past inputs and decisions, useful in dynamic environments to track progress or **predict future events.**
- **Learning Component:** Enables improvement over time. The agent learns from success and failure to make better decisions in the future.
- **Knowledge Base:** A repository of facts, rules, and strategies that the agent uses for decision-making.



AI Agent's Function or Program

AI agents **observe**, **decide**, and **act** — sometimes learning from experience — to achieve specific goals intelligently.

❖ **Agent Function**

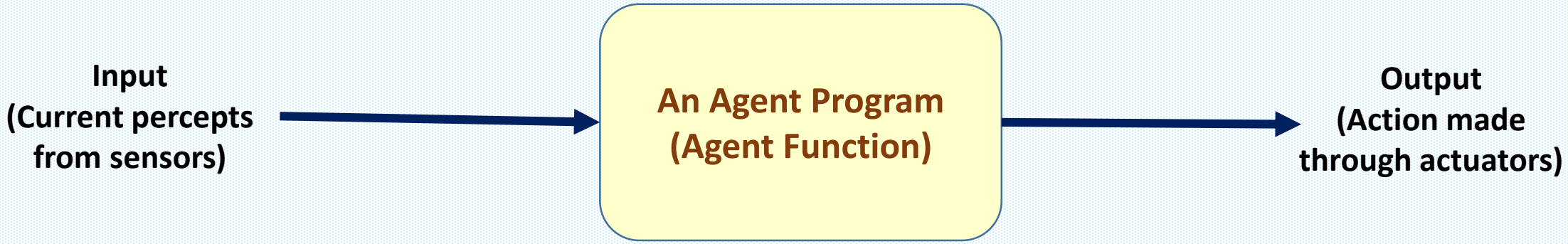
An agent's behavior is defined by an **Agent Function**:

f : Percept → Action

It maps **percepts (inputs)** to **actions**.

❖ An agent program is internally implemented as agent function.

❖ An agent program takes input as the current percept from the sensor and return an action to the effectors (Actuators).



AI Agent's Architectural Components, Function

❖ **Example: Self-Driving Car as an Agent**

- **Sensors**: Cameras, LIDAR, radar, GPS, ultrasonic sensors collect real-time data
- **Percepts:** Lane position, obstacles, speed, traffic signs, nearby vehicles
- **Agent Program**: Uses AI algorithms (computer vision, ML, deep learning) to make decisions (e.g., lane change, braking)
- **Actuators**: Controls steering, throttle, brakes, lights
- **Environment**: Roads, pedestrians, vehicles, traffic signals, weather conditions

❖ **Example: Virtual Personal Assistant (e.g., Siri, Google Assistant)**

A **virtual assistant** is a **software agent** that understands natural language, performs tasks, and interacts with users.

- **Sensors:** Microphone, keyboard (to receive voice or text input)
- **Percepts:** User queries like "What's the weather today?"
- **Agent Program**: Natural Language Processing (NLP), AI models to process the request and find answers
- **Actuators**: Speaker (for voice response), screen (for text/display)
- **Environment**: Smartphone, internet, apps, cloud-based services

Intelligent Agent

- An **Intelligent Agent** is an AI system that **perceives its environment**, **makes decisions**, and **acts** to achieve specific goals — **autonomously** and often with the ability to **learn and adapt**.

- An **intelligent agent** is an entity that uses sensors to perceive its environment and actuators to perform actions that maximize its performance based on some goals.

- **Key Characteristics of an Intelligent Agent:**
  - **Autonomy** – Can operate without human intervention
  - **Perception** – Senses the environment (e.g., via cameras, input data)
  - **Rationality** – Takes actions that are expected to maximize success
  - **Reactivity** – Responds to changes in the environment
  - **Pro-activeness** – Takes initiative to achieve goals
  - **Learning** – Improves its performance based on experience (optional but powerful)

- **How it Works (Agent Loop)**

  Environment → Sensors → Agent (Decision Logic) → Actuators → Environment

Good Agent Vs Bad Agent

❖ In Artificial Intelligence, the **quality of an agent** depends on how **effectively it perceives, decides, and acts** to achieve its goals. We often describe agents as **"good"** or **"bad"** based on their **performance, behavior, and rationality**.

Good Agent	Bad Agent
A good agent is one that: ▪ Performs tasks effectively and accurately ▪ Chooses rational actions based on its knowledge and goals ▪ Adapts to changes in the environment ▪ Improves performance over time (learning) ▪ Maximizes performance measure or utility **Example:** A self-driving car that safely navigates traffic, avoids accidents, and follows rules is a good agent.	A bad agent is one that: ▪ Fails to achieve its goals or performs poorly ▪ Takes irrational or random actions ▪ Does not adapt to the environment ▪ Makes decisions that reduce performance ▪ May cause harm, errors, or inefficiency **Example:** A spam filter that blocks important emails or allows spam through is a bad agent.

Concept of Rationality in AI

❖ **Rationality** in AI refers to an agent's ability to **make the best possible decision or action** to achieve its goals, based on the **information it has**, its **perceptions**, and **available actions**.

❖ What is Rational Agent?

- A **rational agent** is one that **acts to maximize its performance measure**, based on:
  - What it knows about the environment (Knowledge)
  - What it can observe (Percepts),
  - What actions it can take (Actions).

❖ Key Concepts of Rationality

- **Goal-Oriented**: Rationality is judged based on how well the agent achieves its **defined goals**.
- **Performance Measure**: There must be a way to **evaluate success** (e.g., speed, safety, cost, accuracy).
- **Depends on Knowledge**: The more **complete and correct** the agent's knowledge, the better its decisions.
- **Environment Aware**: A rational agent considers the **current state** and possible **outcomes** of its actions.
- **Not Always Perfect**: Rational ≠ Omniscient. A rational agent makes the best decision **with the information available** — even if it turns out wrong later.

Environment in AI

- In Artificial Intelligence, the **environment** is the **external world** or **surroundings** in which an agent operates and interacts.

- It provides **input (percepts)** to the agent via **sensors**, and receives **output (actions)** from the agent via **actuators**.

- Nature of Environment – Key Properties
  - Fully Observable and Partially Observable
  - Deterministic and Stochastic
  - Episodic and Sequential
  - Static and Dynamic
  - Discrete and Continuous
  - Single-agent and Multi-agent

Environment in AI

- Fully Observable and Partially Observable

Fully Observable	Partially Observable
○ If an agent's sensors give it the access to the complete state of the environment at each point of time, then it is fully observable. • **Example:** ◦ Chess, ◦ Tic-Tac-Toe, ◦ Sudoku solver, ◦ Calculator, ◦ Rubik's Cube, ◦ Digital Circuit Simulator, ◦ Warehouse robot with complete sensor coverage	○ In some environment, if there is noise or agent is with inaccurate sensors or may be some states of environment are missing then such environment is partially observable. • **Example:** ◦ Self-driving car (cannot see around corners), ◦ Robot vacuum cleaner, ◦ Poker game, ◦ Medical diagnosis, ◦ Weather forecasting, ◦ Military surveillance, ◦ Drone navigation in fog, ◦ Voice assistants (background noise)

Environment in AI

- Deterministic and Stochastic

Deterministic	Stochastic
○ If next state is predictable from the current state and action then such environment is deterministic. So there is no uncertainty in the environment. • **Example:** ◦ Chess, ◦ Sudoku, ◦ Calculator, ◦ Traffic signal controller, ◦ Industrial robotic arm, ◦ Mathematical equation solver, ◦ Maze with fixed walls	○ An environment is the opposite of a deterministic environment. The next state is totally unpredictable for the agent. So randomness exists in the environment. • **Example:** ◦ Stock market prediction ◦ Medical diagnosis ◦ Self-driving cars ◦ Weather prediction ◦ Poker ◦ Autonomous drones ◦ Online recommendation systems

Environment in AI

- Episodic and Sequential

Episodic	Sequential
○ **Episodic** is an environment where each state is independent of each other. The action on a state has nothing to do with the next state. • **Example:** ◦ Image classification ◦ Medical image classification ◦ Support Bot ◦ Card Games ◦ Spam email detection ◦ Face recognition ◦ Fingerprint verification ◦ Quality inspection in factories ◦ OCR (Character Recognition)	○ **The sequential** environment is an environment where the next state is dependent on the current action. So agent current action can change all of the future states of the environment. • **Example:** ◦ Playing Tennis ◦ Chess ◦ Self-driving car ◦ Robot navigation ◦ GPS navigation ◦ Video games ◦ Warehouse robots ◦ Mars Rover

Environment in AI

- Static and Dynamic

Static	Dynamic
○ The **Static** environment is completely unchanged while an agent is precepting the environment. • **Example:** ◦ Cleaning a room (Environment) by a dry-cleaner reboot (Agent), ◦ 8-queens Puzzle. ◦ Crossword puzzle ◦ Sudoku ◦ Chess (during your turn) ◦ Image classification ◦ Offline data analysis ◦ Mathematical theorem proving	○ **Dynamic** Environment could be changed while an agent is precepting the environment. So agents keep looking at the environment while taking action. • **Example:** ◦ Self-driving car ◦ Traffic control ◦ Stock market ◦ Air traffic control ◦ Robot soccer ◦ Smart home ◦ Online multiplayer games

Environment in AI

- Discrete and Continuous

Discrete	Continuous
○ **Discrete Environment** consists of a finite number of states and agents have a finite number of actions. • **Example:** ◦ moves (action) in a tic-tac game ◦ Crossword Puzzle ◦ 8-queens Puzzle ◦ Chess ◦ Tic-Tac-Toe ◦ Sudoku ◦ ATM Machine ◦ Elevator Controller ◦ Digital Circuit Design	○ While in a **Continuous environment**, the environment can have an infinite number of states. So the possibilities of taking an action are also infinite. • **Example:** ◦ Players' positions in basketball game ◦ Part Picking Robot ◦ Drone flying ◦ Robot arm movement ◦ Self-driving car ◦ Aircraft autopilot ◦ Weather forecasting ◦ Temperature control ◦ Water level control

Environment in AI

- Single-agent and Multi-agent

Single-agent	Multi-agent
○ **Single agent environment** where an environment is explored by a single agent. All actions are performed by a single agent in the environment. • **Example:** ◦ Playing tennis against the ball ◦ Boat driving ◦ Crossword solver ◦ Sudoku solver ◦ Calculator ◦ Route planner ◦ Personal fitness app ◦ Personal robot	○ If two or more agents are taking actions in the environment, it is known as a multi-agent environment. • **Example:** ◦ Playing soccer ◦ Playing maze ◦ football ◦ Chess ◦ Autonomous vehicles ◦ Stock trading ◦ Drone swarm ◦ Smart traffic management ◦ Online multiplayer games

Environment in AI

- More real-life examples and categories into environment groups.

Application	Observable	Deterministic	Episodic/ Sequential	Static/ Dynamic	Discrete/ Continuous	Agents
Chess	Fully	Deterministic	Sequential	Static	Discrete	Multi
Sudoku	Fully	Deterministic	Sequential	Static	Discrete	Single
Self-driving Car	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi
Medical Diagnosis	Partially	Stochastic	Episodic	Dynamic	Continuous	Single
Spam Filter	Fully	Deterministic	Episodic	Static	Discrete	Single
Face Recognition	Fully	Deterministic	Episodic	Static	Discrete	Single
Robot Vacuum	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
GPS Navigation	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Weather Forecasting	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Stock Trading Bot	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi
Drone Delivery	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi
Smart Traffic Signal	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi

Types of AI Agent

AI agents are the foundation of many intelligent systems which helps them to understand their environment, make decisions, and act to achieve specific goals, ranging from simple rule-based to adaptive learning models.

- Operate autonomously with minimal human intervention.
- Used in applications like assistants, automation, and robotics.

There are the following type of AI agents:

1. Simple Reflex Agents

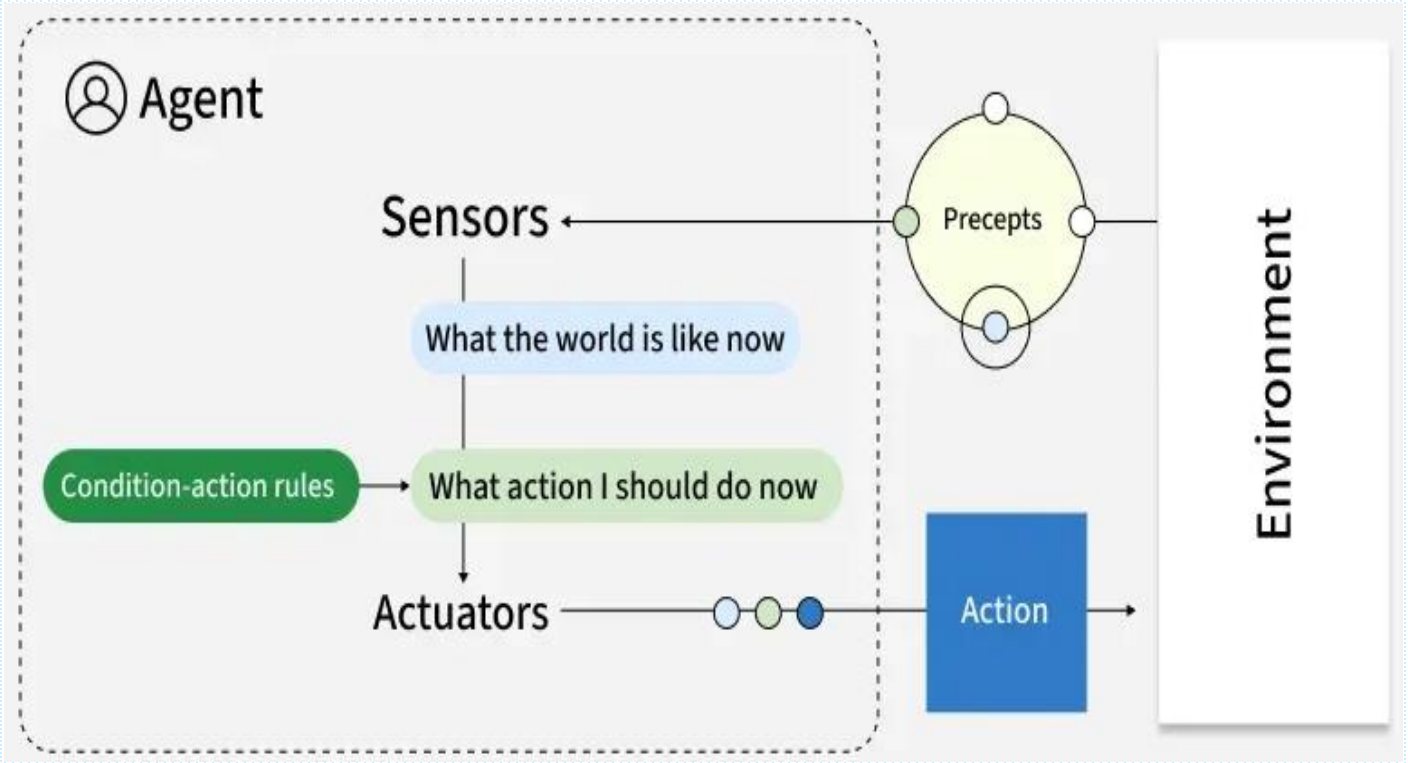
2. Model-Based Reflex Agents

3. Goal-Based Agents

4. Learning Agents

1. Simple Reflex Agent

- A Simple Reflex Agent makes decisions only on the current input using predefined rules.
- It does not remember past events or learn from experience.
- It works well in simple and fully observable environments, but performs poorly in dynamic or partially observable environments.



Simple Reflex Agent

Characteristics:

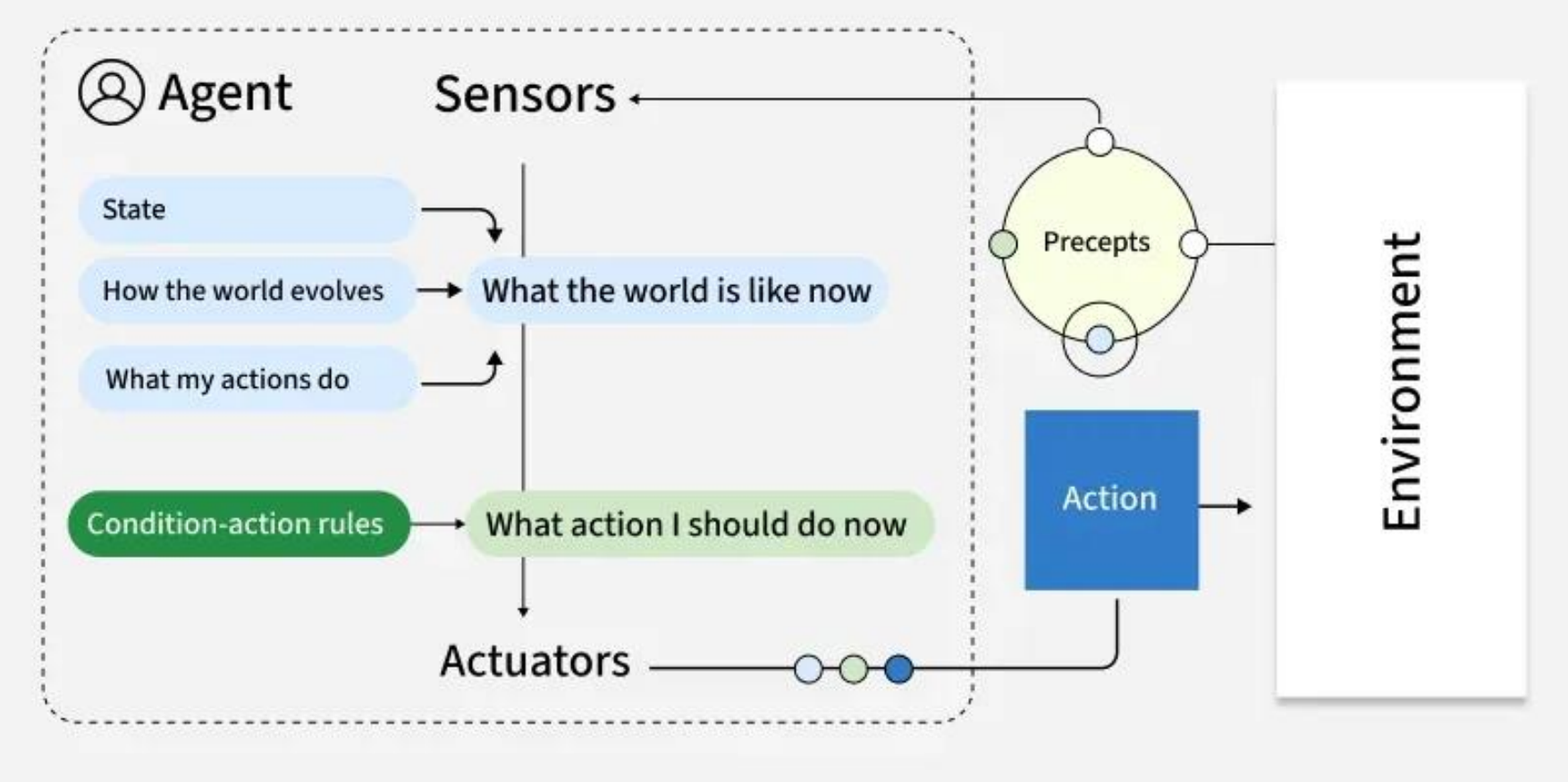
1. **Reactive:** Responds only to the current input.
2. **Rule-Based:** Uses fixed condition–action (if–then) rules.
3. **Fast:** Makes quick decisions because no memory or reasoning is used.
4. **No Memory:** Does not store or use past information.
5. **No Learning:** Cannot improve its performance over time.

Example:

A traffic light controller that changes signals using fixed time intervals.

2. Model-Based Reflex Agent

- A Model-Based Reflex Agent is an improved version of a Simple Reflex Agent.
- It keeps an internal state (memory) of the environment, allowing it to work in partially observable and changing environments.
- It uses both the current input and stored information to make decisions.



Model-based Reflex Agent

Characteristics:

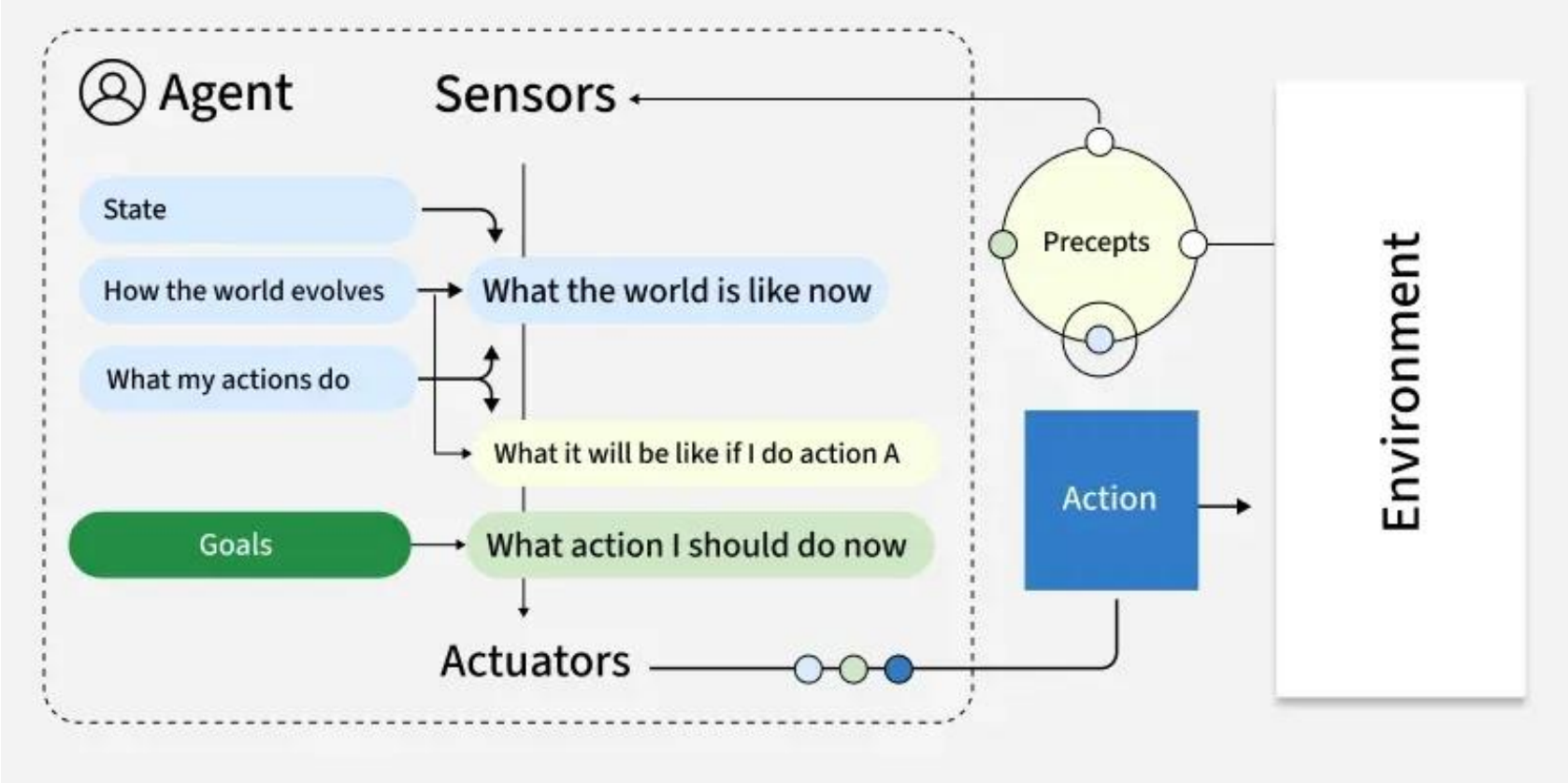
1. **Internal State:** Stores information about the environment.
2. **Adaptive:** Updates its internal state whenever new information is received.
3. **Better Decision-Making:** Uses current input and past information to make better decisions.
4. **Handles Partial Information:** Can work even when the entire environment is not visible.
5. **More Complex:** Requires additional memory and computation.

Example:

A robot vacuum cleaner remembers the room layout and tracks the areas it has already cleaned.

3. Goal-Based Agent

- A Goal-Based Agent makes decisions by choosing actions that help it reach a specific goal.
- Instead of reacting only to the current situation, it plans ahead and selects the best actions to achieve its goal.



Goal Based Agent

Characteristics:

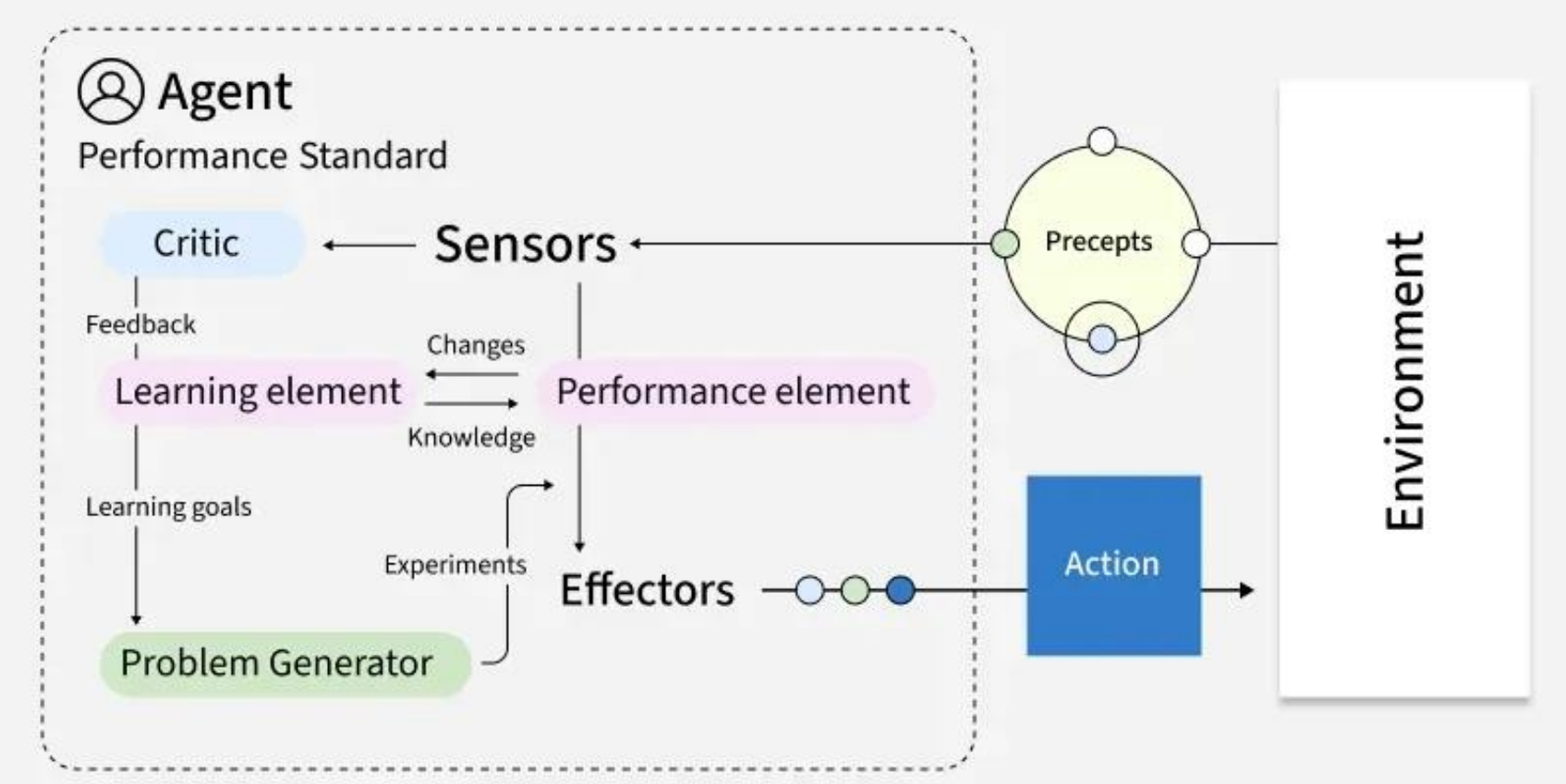
1. **Goal-Oriented:** Chooses actions that help achieve a defined goal.
2. **Planning:** Evaluates different action sequences before making a decision.
3. **Flexible:** Changes its plan when the environment or information changes.
4. **Future-Oriented:** Thinks ahead to select the best possible action.

Example:

A GPS navigation system finds the shortest or fastest route to a destination and updates the route if traffic conditions change.

4. Learning Agent

- A Learning Agent improves its performance by learning from experience.
- It updates its knowledge based on feedback and adapts its behavior to perform better in changing environments.
- A Learning Agent has four extra components that are not present in Simple Reflex, Model-Based Reflex, or Goal-Based Agents.



Learning Agent

These components help the agent learn and improve over time.

1. **Learning Element:** The Learning Element learns from the agent's experiences and feedback. It updates the agent's knowledge so that future decisions become more accurate and efficient.

2. **Critic:** The Critic evaluates the agent's actions by comparing the results with the desired performance. It provides feedback to the Learning Element about what was done correctly or incorrectly.

3. **Performance Element:** The Performance Element decides which action the agent should perform based on the current knowledge and information received from the sensors. It directly interacts with the environment.

4. **Problem Generator:** The Problem Generator encourages the agent to try new actions and explore different possibilities. This helps the agent discover better strategies and improve its learning over time.

Learning Agent

Characteristics:

1. **Learns from Experience:** Improves performance using past experiences and feedback.
2. **Adaptive:** Changes its behavior according to new information.
3. **Explores and Improves:** Tries new actions while also using successful strategies.
4. **Generalizes:** Applies learned knowledge to similar new situations.

Example:

A customer service chatbot learns from previous conversations and gradually provides more accurate and helpful responses to users.

PEAS Model for AI

- The **PEAS model** is a framework used to define the **task environment** for an AI agent. It helps in identifying what the agent needs to do, how it perceives its environment, and how it acts.

- PEAS stands for:
  - **P – Performance Measure**
    - This defines the **success criteria** of the agent.
    - It measures how well the agent is performing in its environment.
  - **E – Environment**
    - The **external context** or world in which the agent operates.
    - The environment provides input (percepts) through sensors.
  - **A – Actuators**
    - The **hardware components** or parts of the agent that perform actions in the environment.
  - **S – Sensors**
    - Sensors are used by the agent to **perceive** or get input from the environment.
    - They collect raw data or signals for the agent to process.

PEAS Model in AI

❖ PEAS Example for a Self-Driving Car:

Component	Description
Performance	Safe driving, obeying traffic rules, reaching destination, comfort
Environment	Roads, traffic, pedestrians, weather
Actuators	Steering, accelerator, brake, signal indicators
Sensors	Cameras, LIDAR, GPS, speedometer, radar

❖ PEAS Model for Vacuum Cleaner (AI Agent)

Component	Description
Performance Measure	Cleanliness of the floor, Coverage area, Battery efficiency, Time taken to clean- Noise level
Environment	The house or room, Floor surface (tile, carpet, wood), Obstacles (furniture, walls, pets)
Actuators	Wheels (for movement), Vacuum motor (for suction), Brushes, Speaker (for alerts)
Sensors	Dirt sensor, Obstacle sensor, Cliff sensor (to avoid stairs), Battery level sensor, Position or mapping sensors (gyroscope, LIDAR)

PEAS Model in AI

❖ PEAS Model for Online Shopping Recommendation System

Component	Description
Performance	Accuracy of recommendations, Click-through rate, User satisfaction
Environment	E-commerce platform, User activity, Product catalog
Actuators	Display of recommended items, Notifications, Email suggestions
Sensors	User click data, Browsing history, Past purchases, Ratings

❖ PEAS Model for Chess-Playing AI

Component	Description
Performance	Winning the game, Minimizing mistakes, Strategy level
Environment	Chessboard, Rules of the game, Opponent moves
Actuators	Move generation (on screen or robot arm)
Sensors	Opponent's move input, Board state

Problem Solving Agent

❖ A **problem-solving** agent is a type of **goal-based agent** that decides what to do by finding sequences of actions that lead to desirable states (goals). It is one of the simplest and most widely studied forms of intelligent behavior in AI.

❖ **Key Characteristics:**

- **Goal-based:** It tries to achieve a specific **goal**.
- **Search Techniques:** Uses **search** techniques to explore possible actions.
- **Plans Ahead:** Applies **planning** to choose the best action sequence.
- **Rational Decision-Making:** Chooses the solution with the lowest cost or highest benefit.

❖ **Components of a Problem-Solving Agent:**

- **Initial State**
  - The starting point of the agent.
  - Example: In a puzzle game, the current layout of pieces.

- **Actions (Operators)**
  - All possible actions that the agent can take.
  - Example: Moving a tile, taking a step, etc.

Problem Solving Agent

❖ Components of a Problem-Solving Agent:

- **Transition Model**
  - Defines what happens when an action is taken (how the state changes).
  - Example: Moving forward changes your position in a maze.
- **Goal Test**
  - Determines whether the current state is a goal state.
  - Example: Reaching the destination, solving the puzzle.
- **Path Cost**
  - A function that assigns a numeric cost to each path.
  - Example: Number of moves, time taken, resources used.

❖ Problem-Solving Process:

1. **Goal Formulation:** Defines what needs to be achieved.
2. **Formulate the problem:** Identifies the initial state, possible actions, and goal state.
3. ***Search for a solution:*** Uses a search algorithms *like BFS, DFS, A** to find a path to the goal.
4. Execute the found solution (action sequence)

Problem Solving Agent

❖ **Example: Robot Maze Solver:** Finding the exit of a maze.

- **Initial State:** Robot is at the maze entrance.
- **Goal State:** Robot reaches the maze exit.
- **Possible Actions:** Move forward, turn left, turn right, or move backward.
- **Search Algorithm:** Explores different paths to find the shortest route.
- **Solution:** Guides the robot safely to the maze exit.

❖ **Example: GPS Navigation System:** Finding the shortest route from **Home** to **College**.

- **Initial State:** Current location (Home).
- **Goal State:** Destination (College).
- **Possible Actions:** Move through different roads and intersections.
- **Search Algorithm:** Finds the shortest or fastest route.
- **Solution:** Guides the user step-by-step to reach the destination.

Searching Algorithms

- **Search Space:** Represents a set of all possible solutions a system may have.
- **Initial State:** The state from where the agent begins the search.
- **Goal State:** A function that checks the current state and determines whether the goal state has been achieved or not.
- **Search Tree**
  - A **tree representation** of the search problem.
  - The **root** of the tree is the **start state**.
  - Nodes represent **states**, and edges represent **actions**.

Transition Model

- Describes what each action **can do**.
- Represented as a **transition function** (resulting state from an action).

4. Path Cost: A function that assigns a **numeric cost** to each path taken.

5. Solution: An **action sequence** leading from the start node to the goal node.

6. Optimal Solution: A solution with the **lowest cost** among all possible solutions.

Searching Algorithms

1. Search

- A step-by-step procedure to **solve or search** a problem in a given search space.
- A search problem has **5 main factors**:

Factor	Description	Example (GPS Navigation)
Initial State	Starting point of the search	Home
State Space	All possible states	All roads and locations
Actions	Possible moves from one state to another	Take left, right, or straight road
Goal Test	Checks if the goal is reached	Arrive at College
Path Cost	Total cost of the solution	Shortest distance or minimum travel time

Searching Algorithms

A search problem has **5 main factors**:

- **Initial State**
  - The starting point of the problem.
  - Describes where the agent begins before searching for a solution.
  - Example: In a maze → starting cell, In GPS navigation → current location.

- **Actions (Operators)**
  - The possible actions or moves the agent can take from each state.
  - Defines the transitions between states.
  - Example: In a chess game → moving a pawn, rook, etc., In a self-driving car → turn left, go straight, stop.

- **Transition Model**
  - Describes the result of applying an action to a state (what the next state will be).
  - Example: If in a maze you are at (2,3) and move "Right" → you go to (2,4), If a car accelerates → new position = old position + speed × time.

Searching Algorithms

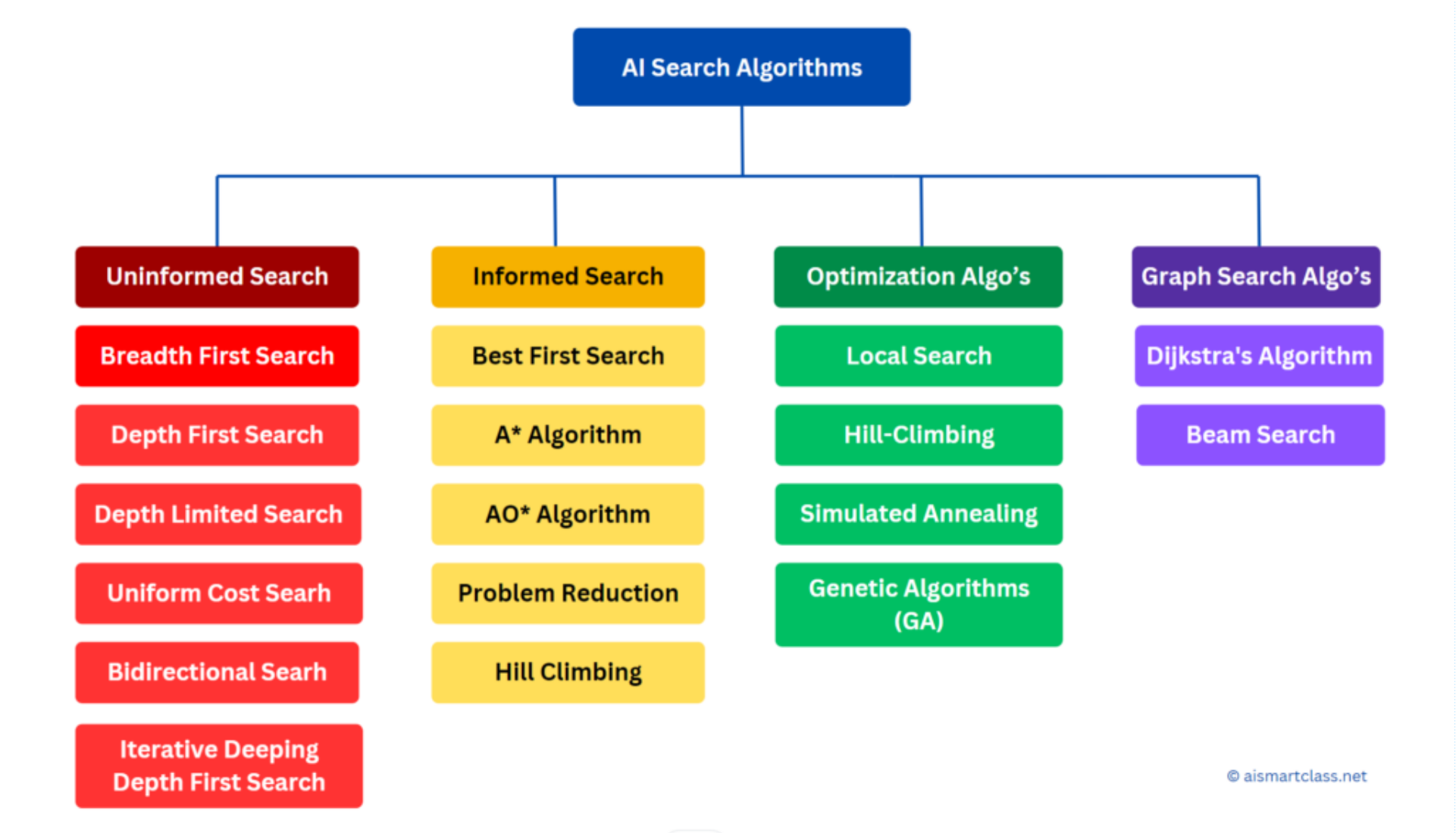
- **Goal Test**
  - A way to check whether the current state is a goal state.
  - Example: In a puzzle → all tiles in order, In navigation → reaching the destination.

- **Path Cost (Cost Function)**
  - A numerical value representing how expensive a path is.
  - The sum of step costs from the initial state to the current state.
  - Example: In GPS → distance in kilometers or travel time, In logistics → cost of fuel or money spent.

Searching Algorithms

Types of Search Algorithms

❖ Based on the problem, search algorithms are classified into **two types**:



Searching Algorithms

Types of Search Algorithms

❖ Based on the problem, search algorithms are classified into **two types**:

Uninformed Search (Blind Search)

❖ **Uninformed Search**, also known as **Blind Search**, is a class of search strategies in Artificial Intelligence that:

❖ It does not contain any **domain knowledge**, such as **closeness** or the **location of the goal**.

❖ Only use the **problem definition** (initial state, actions, goal test, path cost).

❖ Explore the search space blindly until a solution is found.

❖ It operates in a **brute-force** way.

❖ It tries all possible paths blindly without any intelligent direction.

❖ It applies a way in which the agent **searches its successors** until the goal state is achieved.

Searching Algorithms

Key Characteristics:

- ❖ No heuristic or domain knowledge is used.
- ❖ These algorithms treat all nodes uniformly.
- ❖ Suitable for small or well-defined problems.
- ❖ Can be inefficient for large search spaces.

Advantages of Uninformed Search

- ❖ **Simplicity**: Easy to implement since it doesn't require domain-specific knowledge.
- ❖ **Generality**: Can be applied to any kind of search problem where the structure of the problem is known.
- ❖ **Completeness** (for some algorithms like BFS and UCS): Guarantees to find a solution if one exists.
- ❖ **Optimality** (in some cases): Uniform Cost Search is optimal if path costs are non-negative.

Searching Algorithms

Disadvantages of Uninformed Search

- **Inefficiency**: Explores a large number of unnecessary nodes, especially in large search spaces.
- **No Guidance**: Lacks information to direct the search towards the goal effectively.
- **High Time Complexity**: Consumes more time due to exhaustive nature.
- **High Space Complexity**: May require a lot of memory (e.g., BFS stores all nodes in memory).

Examples:

- Breadth-First Search (BFS),
- Depth-First Search (DFS),
- Uniform Cost Search (UCS),
- Depth-Limited Search (DLS),
- Bidirectional Search,
- Iterative Deeping Depth First Search

Searching Algorithms

Informed Search (Heuristic Search)

- **Informed Search** (also called **Heuristic Search**) is a type of search algorithm in Artificial Intelligence that uses **domain-specific knowledge** or **heuristics** to find solutions more efficiently than uninformed (blind) search.

Key Characteristics:

- Makes **intelligent decisions** about which path to explore next.
- Uses a **heuristic function** h(n) that estimates the cost from a node n to the goal.
- Aims to **reduce the number of states explored** by focusing on promising paths.

Important Concepts:

- **Heuristic Function h(n)**
  - A function that provides a **guess** about how close a state is to the goal.
  - Not guaranteed to be correct, but helps guide the search.
  - **Admissible**: Never overestimates the true cost.
  - **Consistent** (monotonic): $h(n) \leq c(n, a, n') + h(n')$

Searching Algorithms

Informed Search (Heuristic Search)

Advantages:

- Much faster than uninformed search.
- **Efficient exploration** of the search space.
- Often finds the **shortest/optimal path** when heuristics are good.

Disadvantages:

- May be **incomplete or non-optimal** if the heuristic is poor.
- Requires **careful design** of the heuristic function.
- Can be **misled** by local minima, plateaus, or incorrect heuristics.

Examples:

- Best-First Search, A* Search, AO* Search, Hill Climbing, Problem Reduction

Searching Algorithms

Difference Between Uninformed Search and Informed Search

Uninformed Search	Informed Search
Search without information	Search with information
No knowledge regarding the search	Uses domain knowledge to find a good solution
More time-consuming	Less time-consuming
More complexity	Less complexity
Best solution not guaranteed	High chances of finding the optimal solution with less time
DFS (Depth First Search), BFS (Breadth First Search)	Best First Search, A* Search, AO* Search

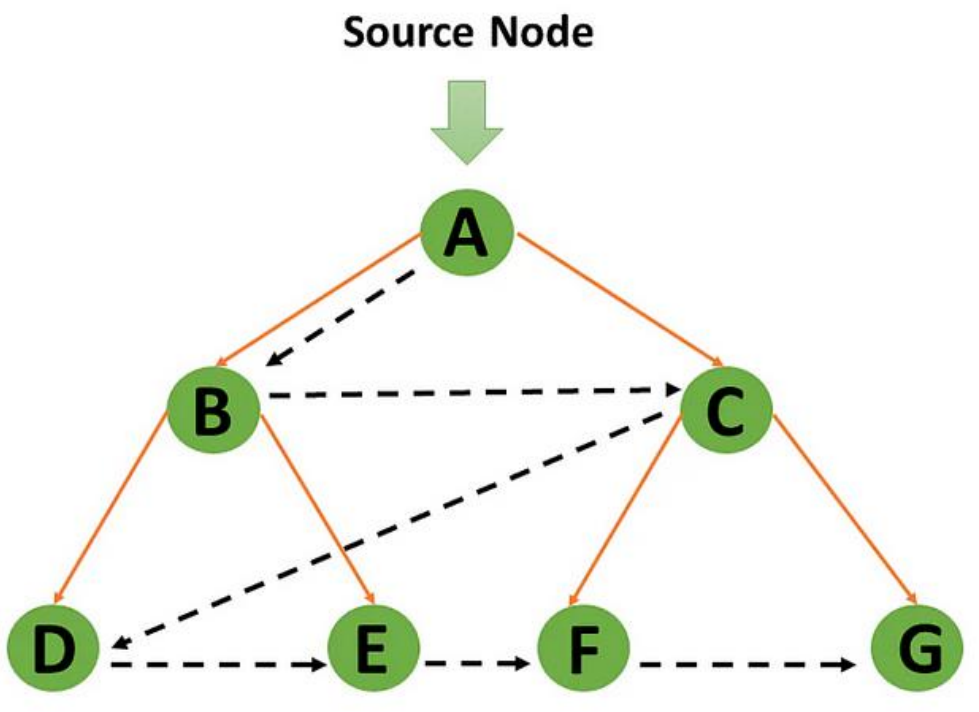
Breadth-First Search (BFS)

Breadth-First Search (BFS) is a graph traversal/search algorithm that explores all the neighbor nodes at the present depth before moving on to the nodes at the next depth level. It uses a **queue** (FIFO – First In First Out) data structure to keep track of the nodes to be explored.

BFS Algorithm Using Queue

Steps:

1. Initialize a queue and add the **starting (root) node** to it.
2. Mark the starting node as **visited**.
3. While the queue is **not empty**:
   - Dequeue a node from the front of the queue.
   - Process that node (visit it).
   - Enqueue all **unvisited adjacent (neighbor) nodes** of the current node.
   - Mark those neighbors as visited.
4. Repeat until the queue is empty or the **goal node is found**.

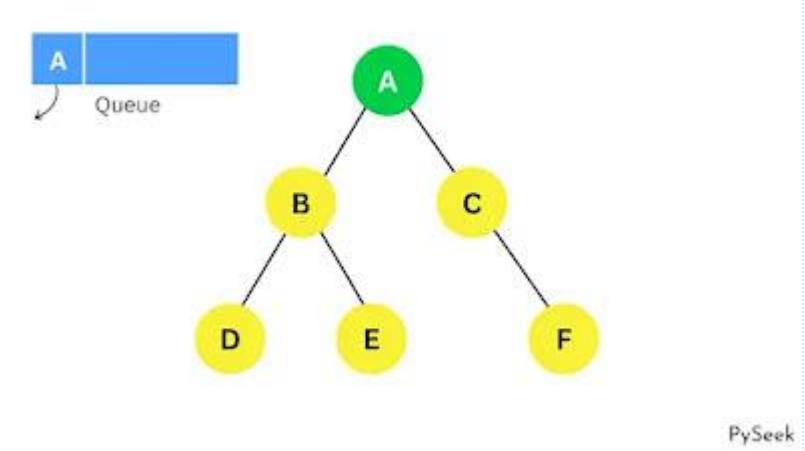


Why a Queue?

- BFS explores level by level.
- The **first node added to the queue is processed first**.
- This ensures **shortest path discovery** in unweighted graphs.

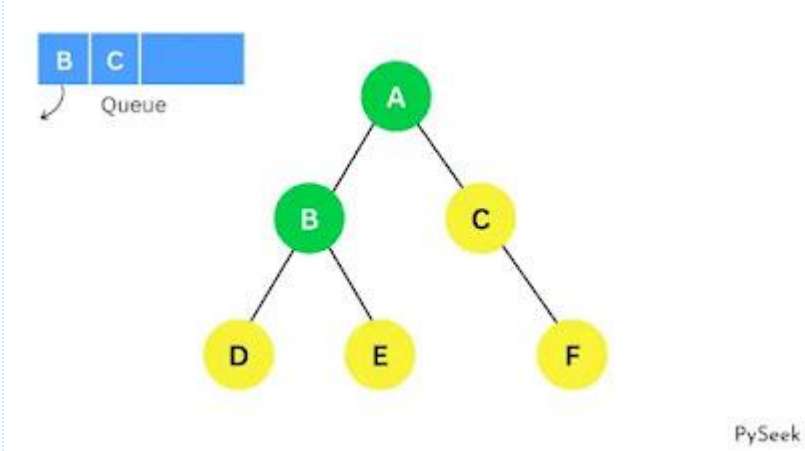
Breadth-First Search (BFS)

Traverse A



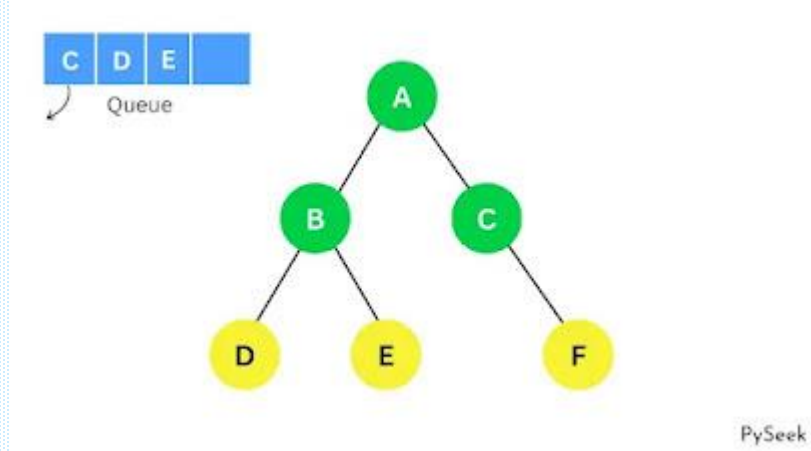
BFS: A

Traverse B



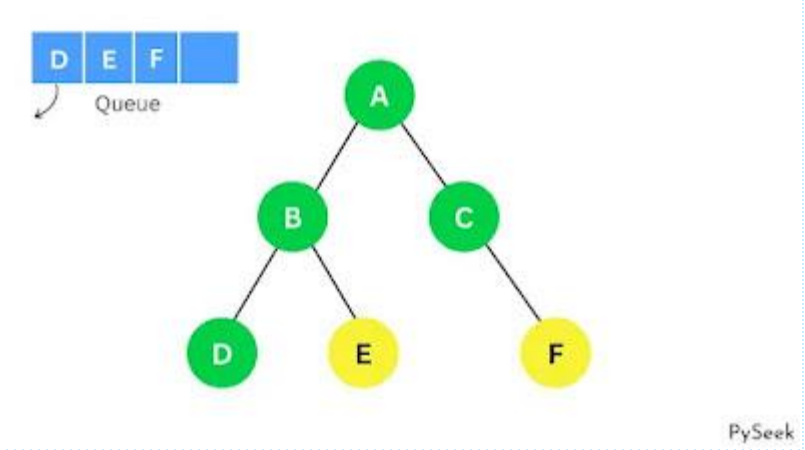
BFS: A, B

Traverse C



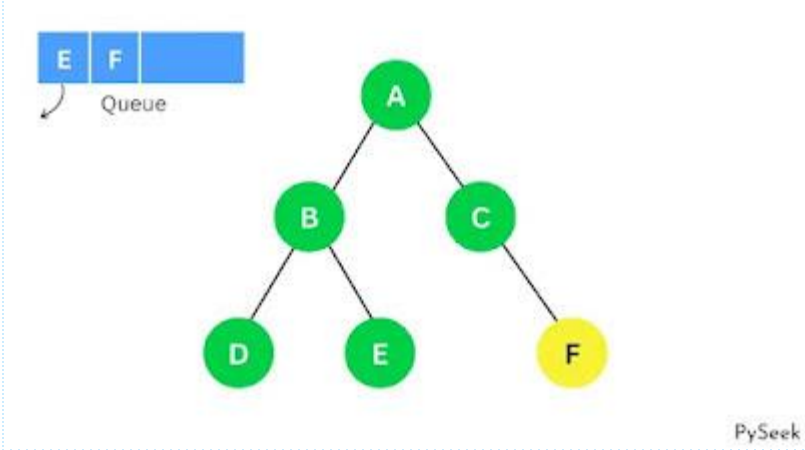
BFS: A, B, C

Traverse D



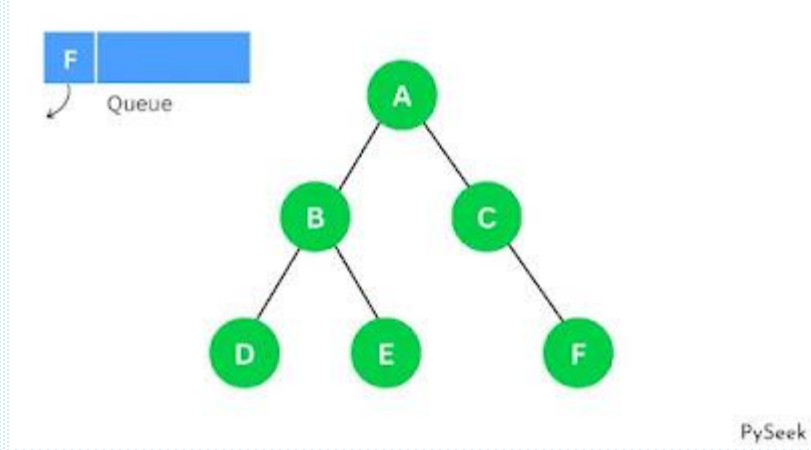
BFS: A, B, C, D

Traverse E



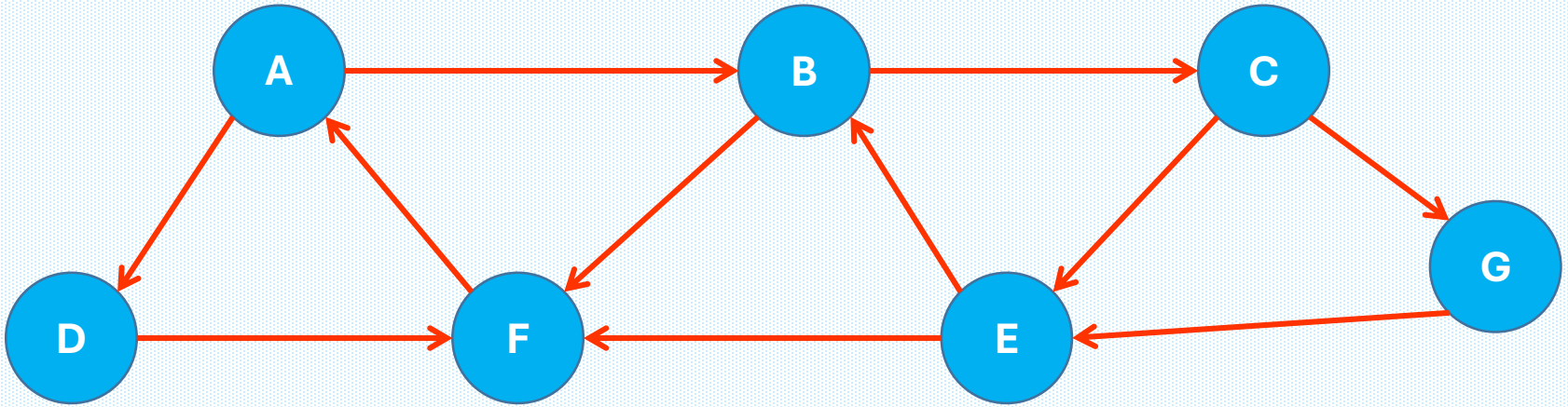
BFS: A, B, C, D, E

Traverse F



BFS: A, B, C, D, E, F

Breadth-First Search (BFS)



Step-by-Step BFS Algorithm (Based on Given Graph)

Given Graph Nodes: Vertices: A, B, C, D, E, F, G

Adjacency List:

A → B, D
B → C, F
C → E, G
D → F
E → B, F
F → A
G →E

Step 1:
Start with node **A.**
Queue: []

Step 2:
Insert starting node **A** into the queue.
Queue: [A]
BFS: A

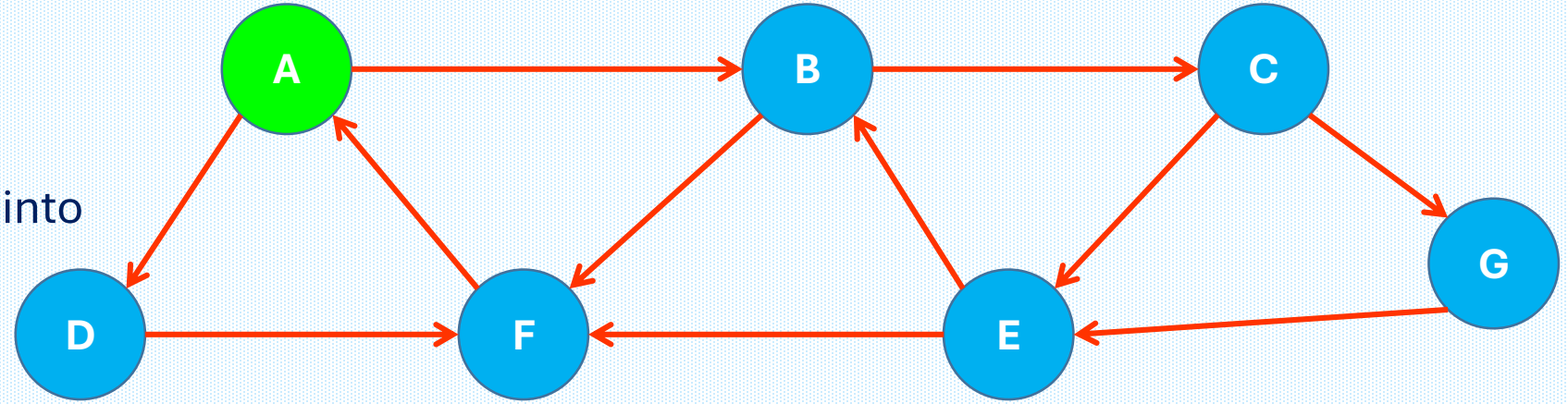
Breadth-First Search (BFS)

Step 3

- Remove A from the queue.
- Insert adjacent nodes of A: B, D into the queue.

  Queue: [B, D]

  BFS: A

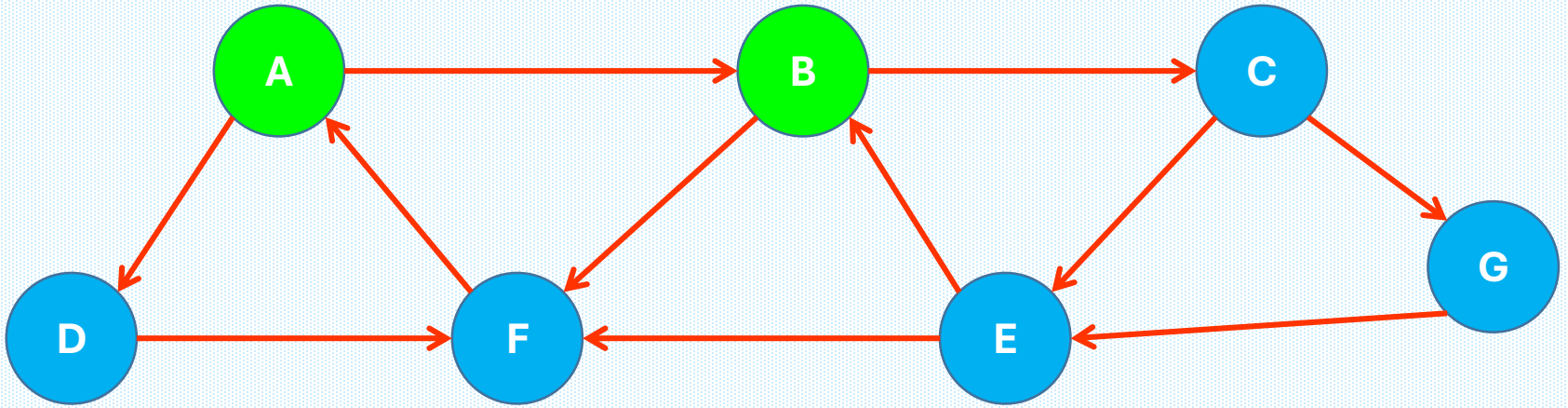


Step 4

- Remove B.
- Adjacent nodes of B: C, F.
- Insert them into the queue.

  Queue: [D, C, F]

  BFS: A, B

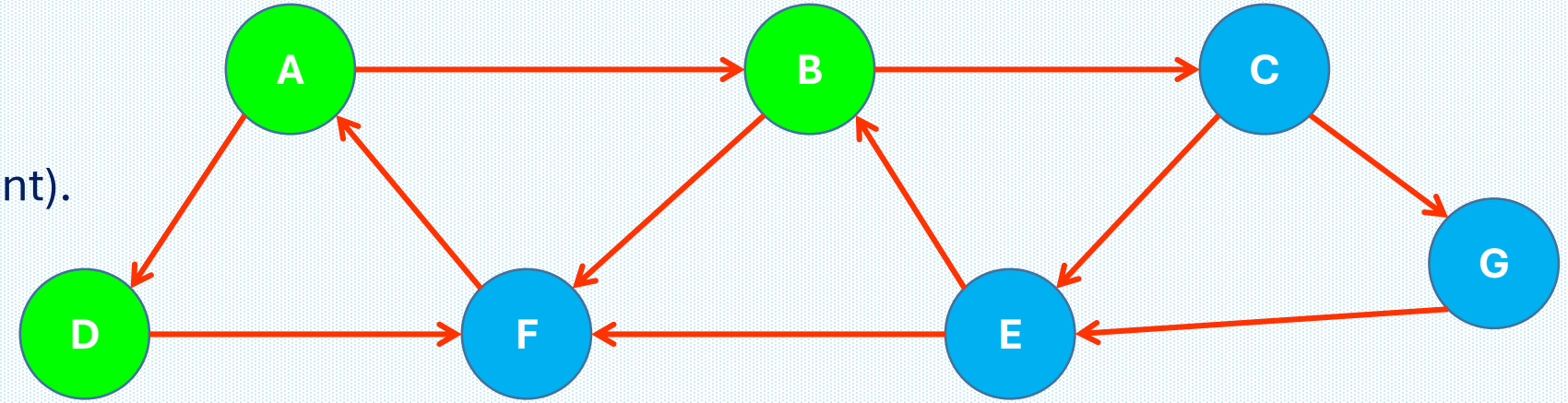


Step 5

- Remove D.
- Adjacent node: F (already present).
- Do not reinsert.

  Queue: [C, F]

  BFS: A, B, D



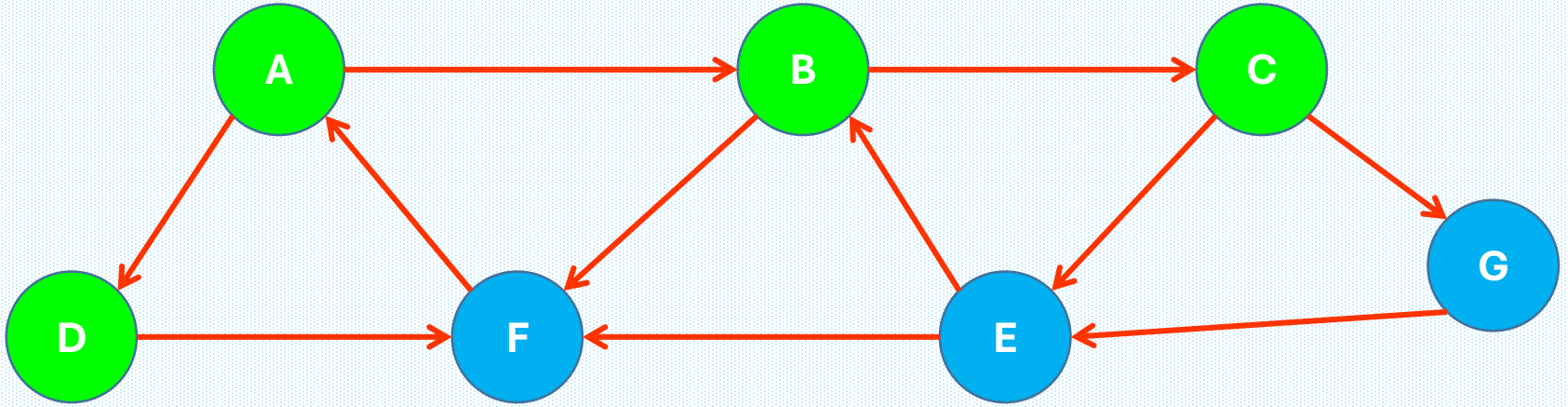
Breadth-First Search (BFS)

Step 6

- Remove C.
- Adjacent node: G, E.
- Insert G, E.

  Queue: [F, G, E]

  BFS: A, B, D, C

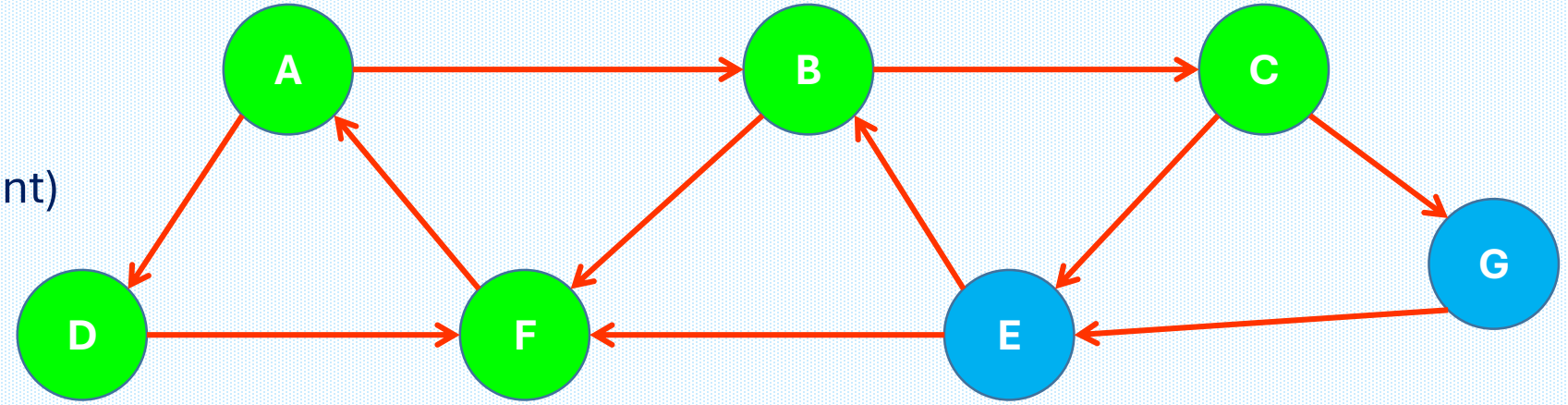


Step 7

- Remove F.
- Adjacent node: A (already present)
- Do not reinsert.

  Queue: [G, E]

  BFS: A, B, D, C, F

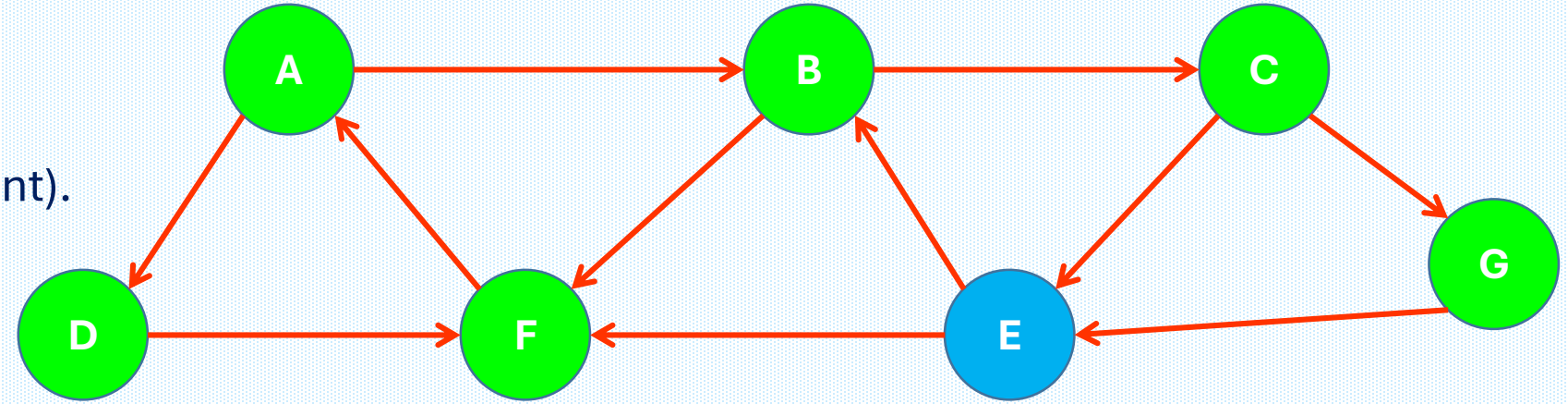


Step 8

- Remove G.
- Adjacent node: E (already present).
- Do not reinsert.

  Queue: [E]

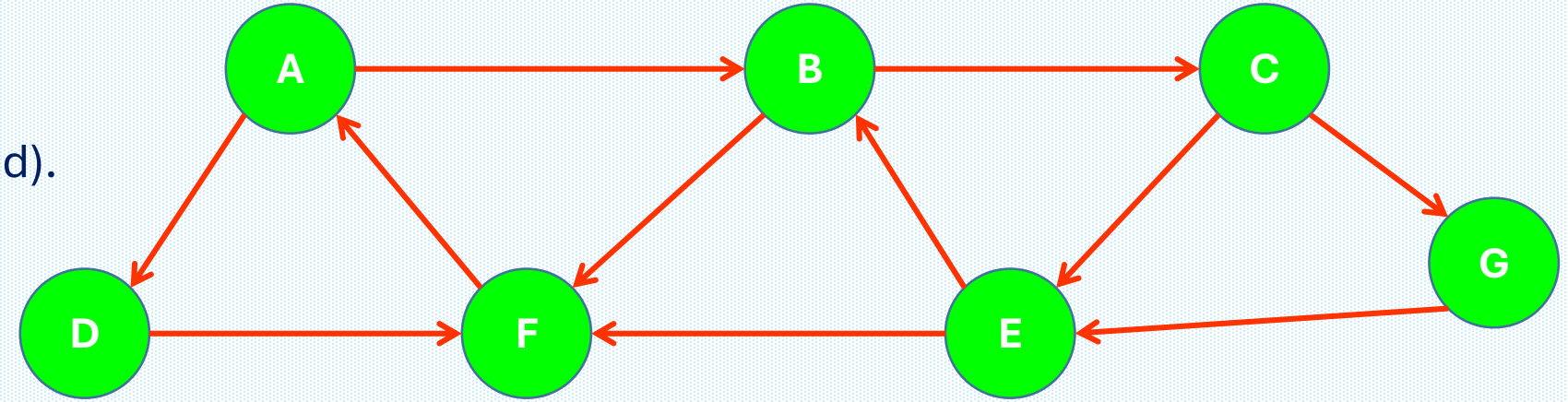
  BFS: A, B, D, C, F, G



Breadth-First Search (BFS)

Step 10

- Remove E.
- Adjacent node: G (already visited).
  Queue: []
  BFS: A, B, D, C, F, G, E



BFS Characteristics:

- **Complete**: Yes (if the solution exists)
- **Optimal**: Yes (for unweighted graphs)
- **Time Complexity**: O(V + E), The graph has |V| vertices and |E| edges.
- **Space Complexity**: O(V)

Solution of Water Jug Problem Using Breadth-First Search (BFS)

The Water Jug Problem is a classic example of a state-space search problem in Artificial Intelligence. We can solve it using Breadth-First Search (BFS) by treating each possible amount of water in the two jugs as a state.

Example Problem

Suppose we have:

- **Jug A:** 4 litres capacity
- **Jug B:** 3 litres capacity
- **Initial state:** Both jugs are empty → **(0, 0)**
- **Goal:** Obtain exactly **2 litres** of water in Jug A.

Here, a state **(x, y)** means:

x = amount of water in 4-litre Jug A
y = amount of water in 3-litre Jug B

Possible Operations

From any state, we can perform six operations:

1. Fill Jug A completely.
2. Fill Jug B completely.
3. Empty Jug A.
4. Empty Jug B.
5. Pour water from Jug A into Jug B.
6. Pour water from Jug B into Jug A.

BFS explores these states **level by level** and therefore finds the solution with the minimum number of operations.

Water Jug Problem (4 Litre Jug and 3 Litre Jug)

Goal: Obtain 2 litres in Jug A

Notation: (A, B)

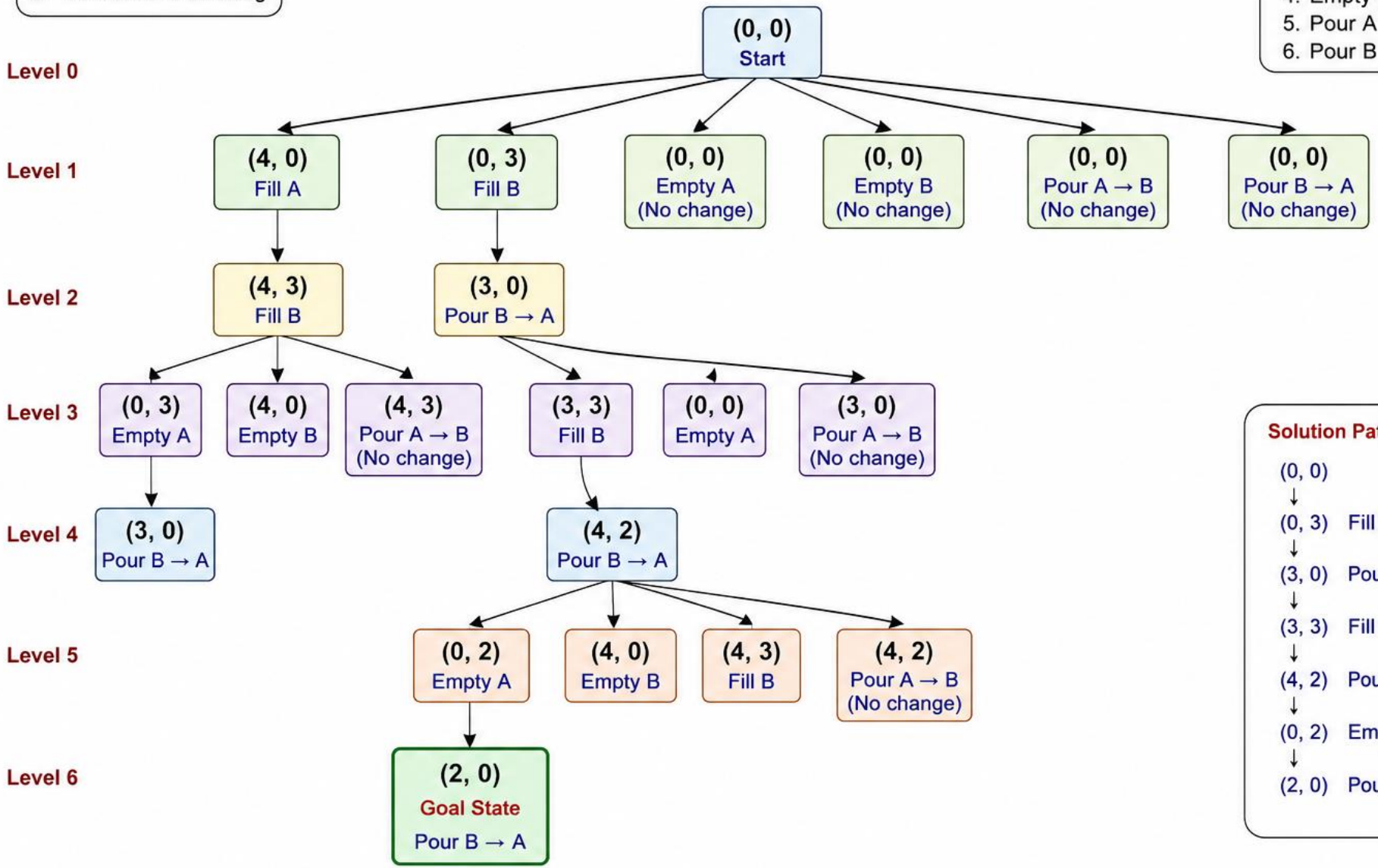
A = Amount in 4 Litre Jug

B = Amount in 3 Litre Jug

Operations

1. Fill A
2. Fill B
3. Empty A
4. Empty B
5. Pour A → B
6. Pour B → A

BFS Tree Representation



Solution Path (Shortest)

(0, 0)
↓
(0, 3) Fill B
↓
(3, 0) Pour B → A
↓
(3, 3) Fill B
↓
(4, 2) Pour B → A
↓
(0, 2) Empty A
↓
(2, 0) Pour B → A (Goal)

Solution of Water Jug Problem Using Breadth-First Search (BFS)

Starting state: **(0, 0)**
A shortest path to the goal is:

Step	State (A, B)	Operation
0	(0, 0)	Initial state
1	(0, 3)	Fill Jug B
2	(3, 0)	Pour B → A
3	(3, 3)	Fill Jug B
4	(4, 2)	Pour B → A
5	(0, 2)	Empty Jug A
6	(2, 0)	Pour B → A

State-Space Representation

The problem can be represented as:

1. **Initial State:** (0, 0)
2. **Goal State:** (2, 0) or any state containing 2 litres, depending on the problem definition.
3. **State:** (x, y), where $0 \le x \le 4$ and $0 \le y \le 3$.
4. **Actions:**
   Fill, empty, and transfer water between the two jugs.
5. **Search Algorithm:**
   Breadth-First Search (BFS)
6. **Solution:**
   (0,0) → (0,3) → (3,0) → (3,3) → (4,2) → (0,2) → (2,0)
7. **Total operations = 6.**

Breadth-First Search (BFS)

Applications of Breadth-First Search (BFS)

- **Shortest Path in Unweighted Graphs:** BFS finds the shortest path (fewest edges) from the source to any node.
- **Peer-to-Peer Networks:** Used in systems like BitTorrent to discover all connected peers.
- **Search Engine Crawlers:** Builds indexes by exploring web pages level by level from a source page.
- **Social Networking Sites:** Finds people within a certain distance (e.g., "friends of friends").
- **GPS Navigation Systems:** Locates all nearby destinations from a current location.
- **Broadcast Networks:** Ensures that data packets reach all nodes efficiently.
- **Garbage Collection (Cheney's Algorithm):** Uses BFS to copy reachable memory blocks, improving memory locality.
- **Cycle Detection:** Helps detect cycles in both directed and undirected graphs.
- **Path Existence:** Checks if a route exists between two vertices.
- **Connected Components:** Identifies all nodes in a connected subgraph.

Depth-First Search (DFS)

Depth First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It can be applied to both **graphs** and **trees**.

Characteristics of Depth First Search (DFS)

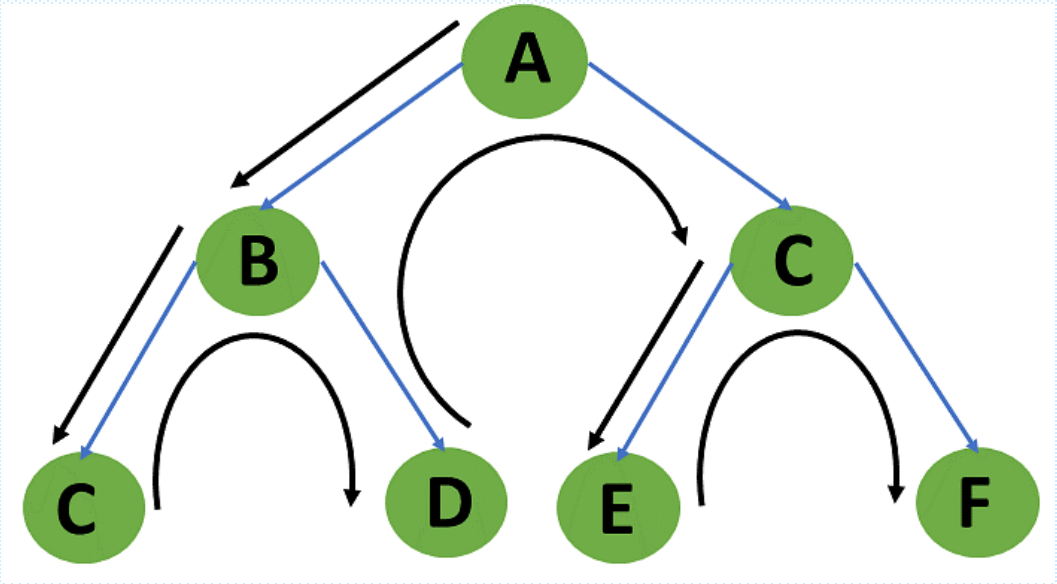
- DFS explores a graph by going as deep as possible along each branch before backtracking.
- DFS uses a **stack** for its iterative implementation or the **call stack** in the recursive version.
- **Time Complexity:** O(V + E) where V = number of vertices, E = number of edges.
- **Space Complexity:** O(V) in the worst case, due to the stack and visited node storage.
- Nodes are visited in a **depthward motion**, which means child nodes are visited before sibling nodes.
- DFS is **not complete**, meaning it may not find a solution if one exists, especially in infinite or cyclic graphs without proper cycle detection.
- DFS is **not optimal** as it doesn't guarantee the shortest path in an unweighted graph.
- Requires tracking of visited nodes to avoid infinite loops in cyclic graphs.
- Can be implemented using **recursion** or **explicit stack-based iteration**.

Depth-First Search (DFS)

DFS Algorithm Using Stack

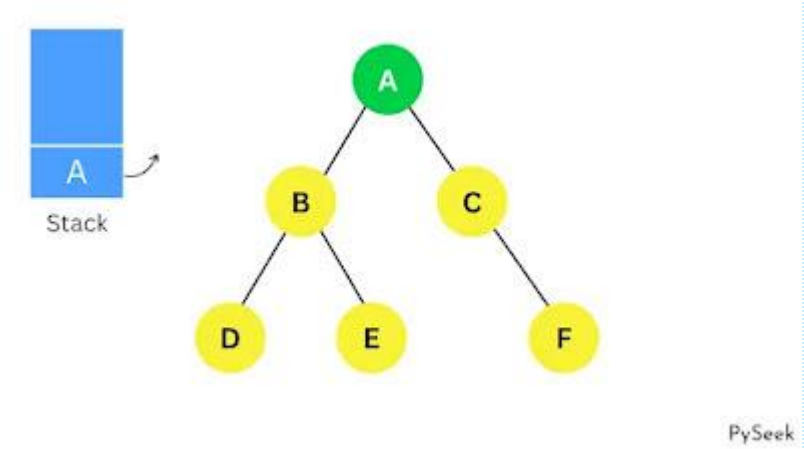
Steps:

1. **Initialize a stack** and push the starting (root) node onto it.
2. **Mark the starting node as visited.**
3. **While the stack is not empty:**
   - Pop a node from the **top** of the stack.
   - **Process** (visit) that node.
   - Push all **unvisited adjacent (neighbor)** nodes of the current node **onto the stack**.
   - **Mark** those neighbors as visited.
4. **Repeat** until the stack is empty or the goal node is found.



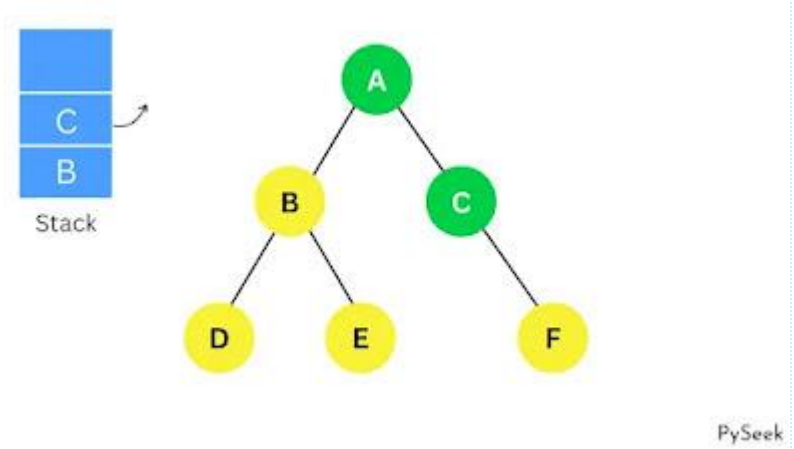
Depth-First Search (DFS)

Traverse A



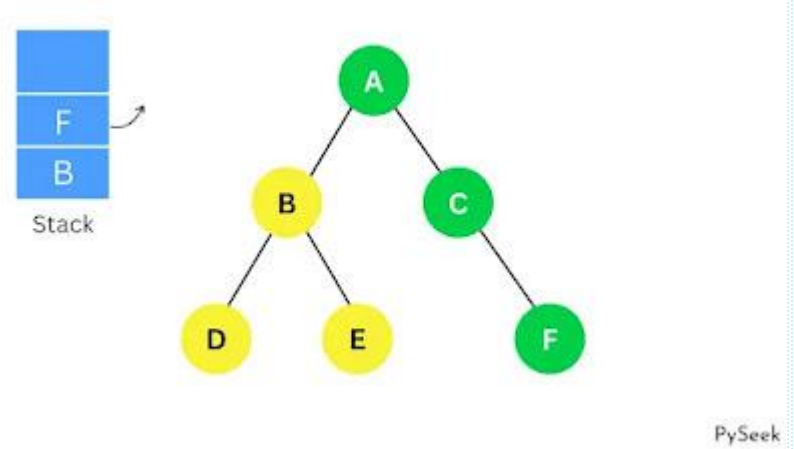
BFS: A

Traverse C



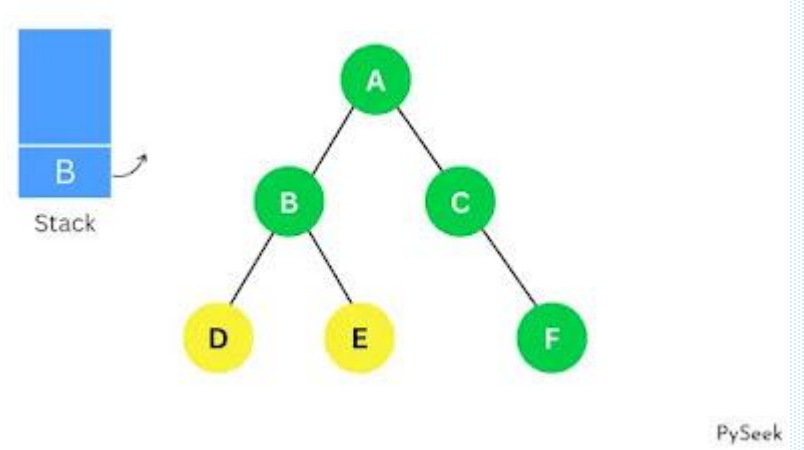
BFS: A, C

Traverse F



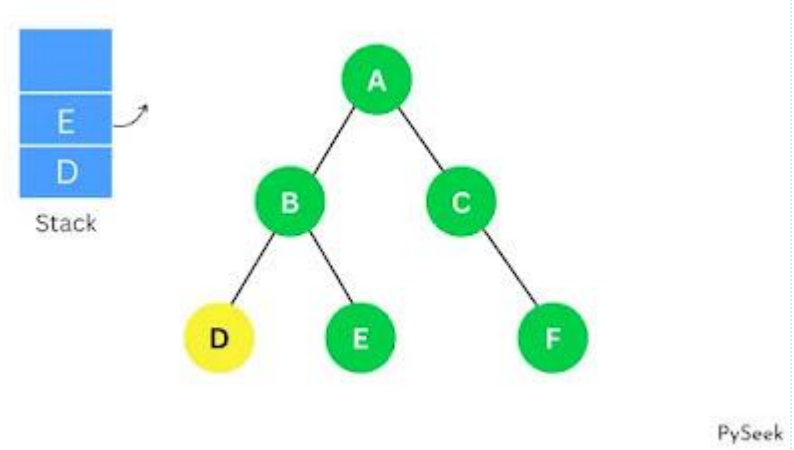
BFS: A, C, F

Traverse B



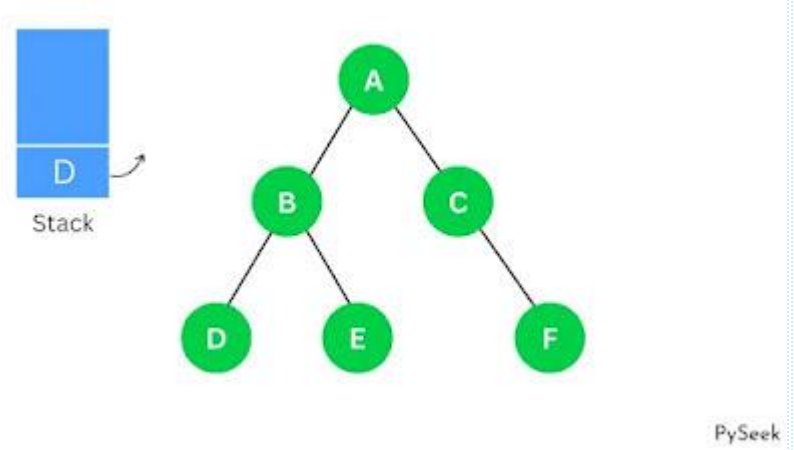
BFS: A, C, F, B

Traverse E



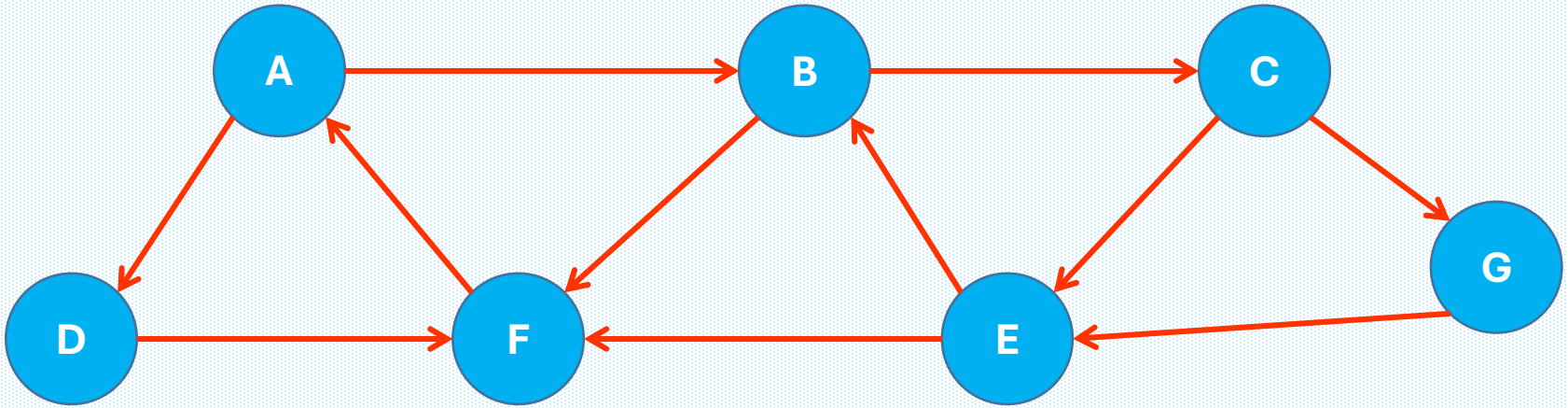
BFS: A, C, F, B, E

Traverse D



BFS: A, C, F, B, E, D

Depth-First Search (DFS)



Step-by-Step DFS Algorithm (Based on Given Graph)

Given Graph Nodes: Vertices: A, B, C, D, E, F, G

Adjacency List:

A → B, D
B → C, F
C → E, G
D → F
E → B, F
F → A
G → E

Step 1:
In the above graph, minimum path **P** can be found by using the **DFS** that will start at node **A** and end at node **G**. Here we required a **stack** to implement the **DFS** algorithm.

Step 2:
Initially, the stack is empty — no element present inside the stack.

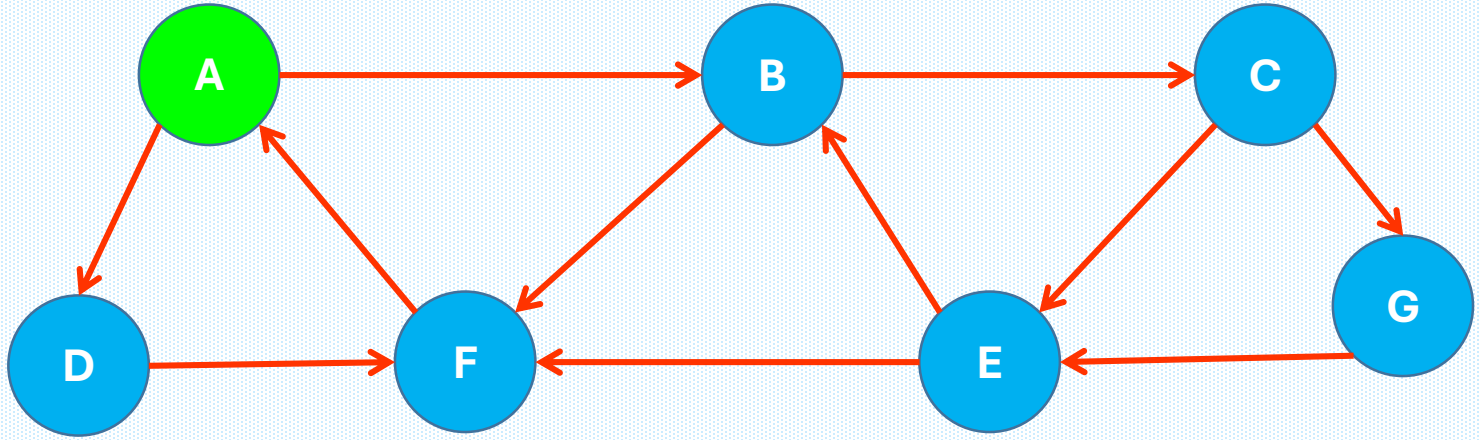
Step 3
Then we have to insert the starting vertex of the graph, which is **A**.

Depth-First Search (DFS)

Step 4

- Now delete **A** & put the adjacent elements of **A** into the stack.
- The adjacent elements of **A** are **B** & **D**. Now in the stack: (**D**, **B**) is present.

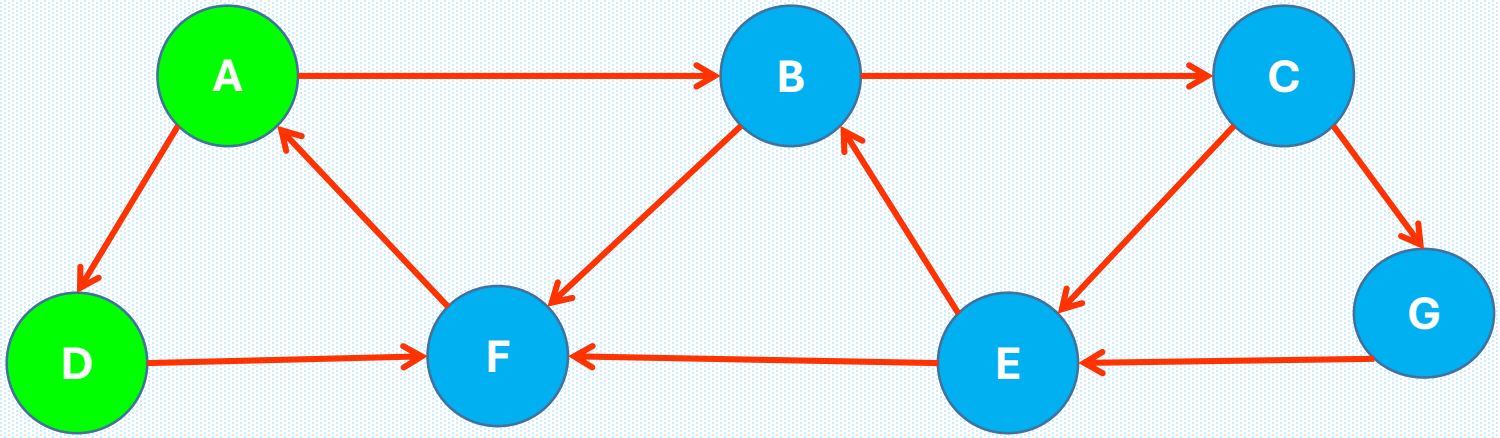
DFS: A



Step 5

- Now **D** is the top of the stack.
- Delete **D** and put the adjacent element of **D**, i.e., **F**, into the stack.
- From the graph, adjacent element of **D** is **F**.
- Now in the stack, **F** is present.

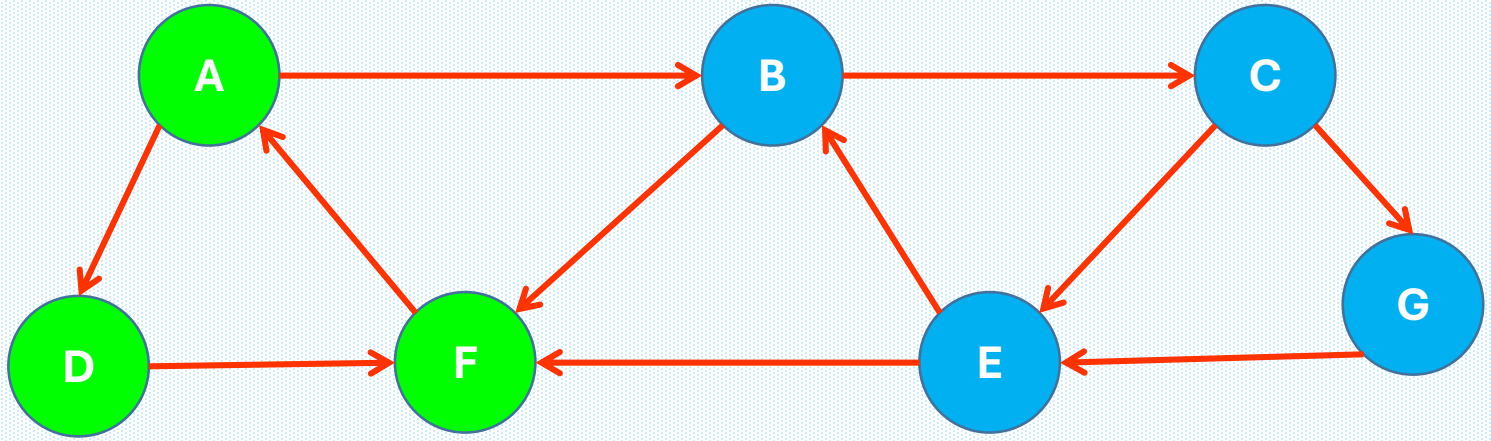
DFS: A, D



Depth-First Search (DFS)

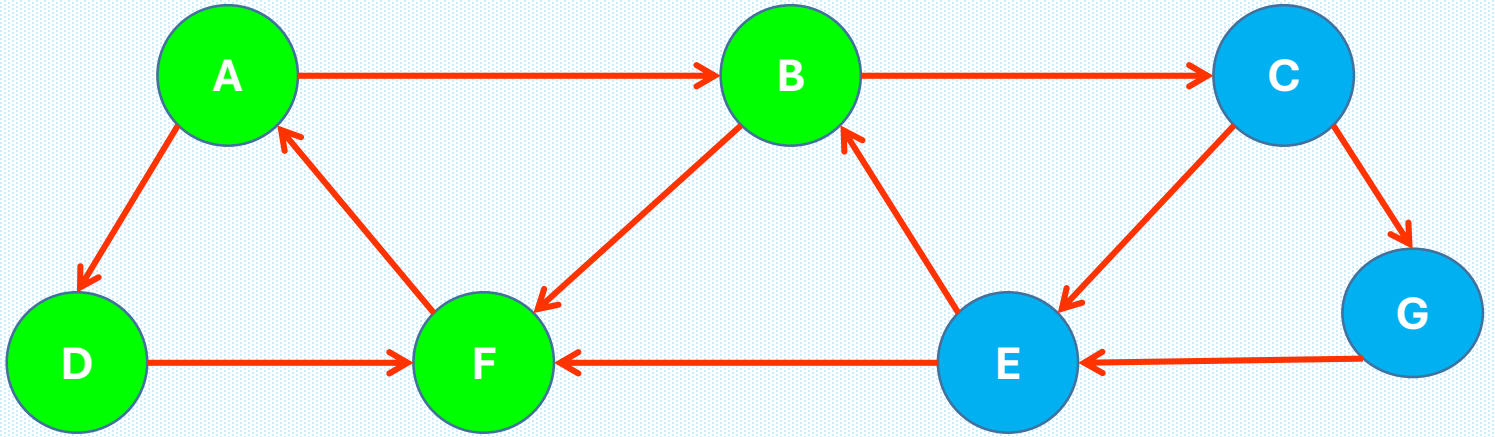
Step 6

- Now **F** is the top of the element of the stack.
- That's why delete **F** and put the adjacent element of **F** (i.e., **A**) into the stack.
- The adjacent element of **F** is **A**, but **A** is already traversed in the stack.
- DFS: A, D, F



Step 7

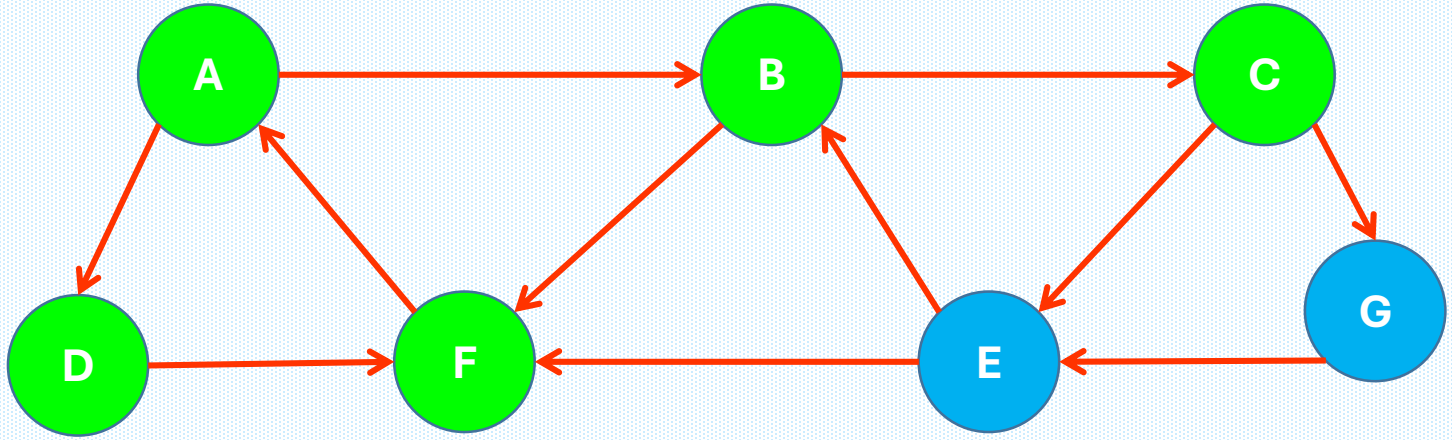
- Now **B** is the top of the element of the stack.
- That's why delete **B** and put the adjacent elements of **B** into the stack.
- The adjacent elements of **B** are **C**, **F**.
- But **F** is already traversed in the stack.
- DFS: A, D, F, B



Depth-First Search (DFS)

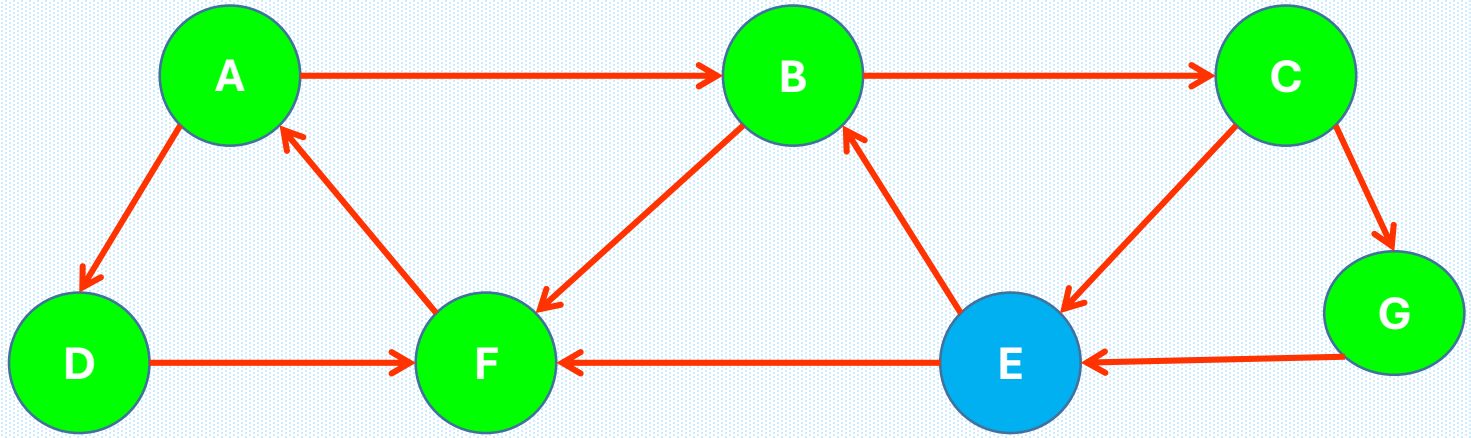
Step 8

- Now **C** is the top of the element of the stack.
- That's why delete **C** and put the adjacent elements of **C** into the stack.
- The adjacent elements of **C** are **E**, **G**.
- **Top of stack = C**
  Delete **C**, push its adjacent nodes **E** and **G**.
- DFS: A, D, F, B, C



Step 9

- **Top of stack = G**
  Delete **G**, push its adjacent node **E** → **Already in the stack**, so skip.
- DFS: A, D, F, B, C, G

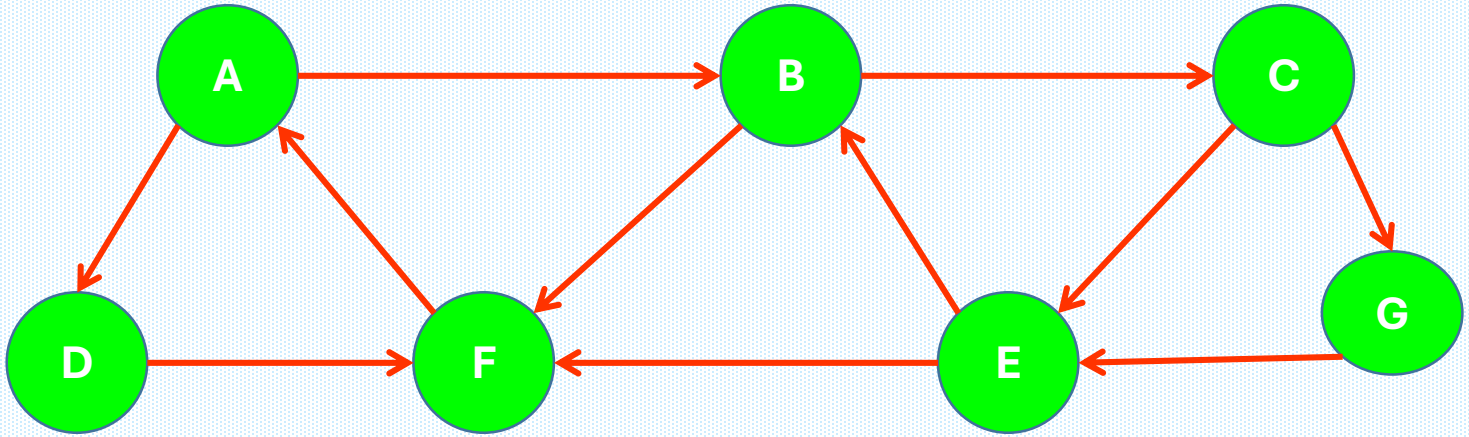


Depth-First Search (DFS)

Step 10

- **Top of stack = E**

  Delete **E**, its adjacent nodes are **D** and **F** → both already visited.
- DFS: A, D, F, B, C, G, E



Depth-First Search (DFS)

Applications of DFS (Depth First Search)

- **Cycle Detection:** Used to detect cycles in **both directed and undirected graphs**.
- **Topological Sorting:** Essential for ordering tasks (vertices) in **Directed Acyclic Graphs (DAGs)**.
- **Finding Connected Components:** Identifies all nodes connected to a particular node in **undirected graphs**.
- **Path Finding:** Can be used to determine whether **a path exists** between two nodes (not necessarily the shortest).
- **Solving Mazes or Puzzles:** Explores all possible paths to find a valid solution (e.g., mazes, N-Queens problem, Sudoku).
- **Backtracking Algorithms:** DFS is the foundation of **backtracking**, used in constraint satisfaction problems.
- **Strongly Connected Components (SCCs):** Used in algorithms like **Kosaraju's** or **Tarjan's** for directed graphs.
- **Artificial Intelligence and Game Solving:** DFS explores **decision trees** and possible moves in games (like chess or tic-tac-toe).
- **Tree Traversals:** In-order, Pre-order, and Post-order traversals of binary trees are DFS-based.
- **Web Crawlers and File Systems:** Used to **recursively explore directories or web links**.

Uniform Cost Search (UCS)

- It is a variant of Dijkstra's algorithm and is particularly useful when all edges of the graph have different weights.
- It is used to find the minimum path from the source node to the destination node around a directed weighted graph.
- This searching algorithm uses a brute force approach, visits all the nodes based on their current weight, and finds the path having minimum cost by repeatedly checking all the possible paths.
- A search algorithm that expands the node with the *minimum path cost* from the start node.
- Uninformed search (no heuristics).
- Always Expands least costly node until goal is reached.
- Uses Priority Queue (min-heap) based on path cost g(n).
- Priority Queue is sorted increasing order of path cost.
- Stops when goal is expanded; Guaranteed to find optimal solution.

Uniform Cost Search (UCS)

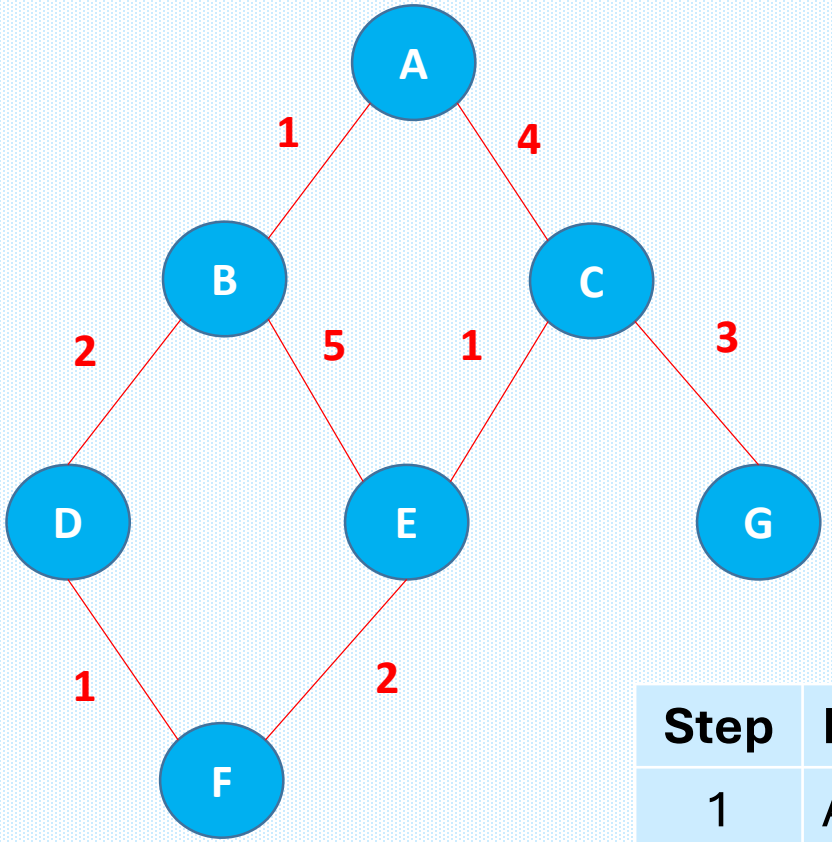
❖ **Algorithm Steps**

1. Initialize a priority queue with the start node, having cost 0.
2. Remove the node with the lowest cost from the queue.
3. If it is the goal node → return the path and cost.
4. Else, expand the node:
    a. For each neighbor, calculate the total path cost from the start node.
    b. If the neighbor hasn't been visited or has a lower cost now, add/update it in the queue.
5. Repeat until the goal is found or the queue is empty.

❖ **Time Complexity:** The worst-case time complexity of UCS is **O(b^d)**, where b is the branching factor of the search tree, and d is the depth of the goal node.

❖ **Space Complexity:** The space complexity of UCS is also **O(b^d)** but can be further reduced to **O(bd)** if we use a binary heap as the priority queue.

Uniform Cost Search (UCS)



Edge Weights:

- A → B = 1
- A → C = 4
- B → D = 2
- B → E = 5
- C → E = 1
- C → G = 3
- D → F = 1
- E → F = 2

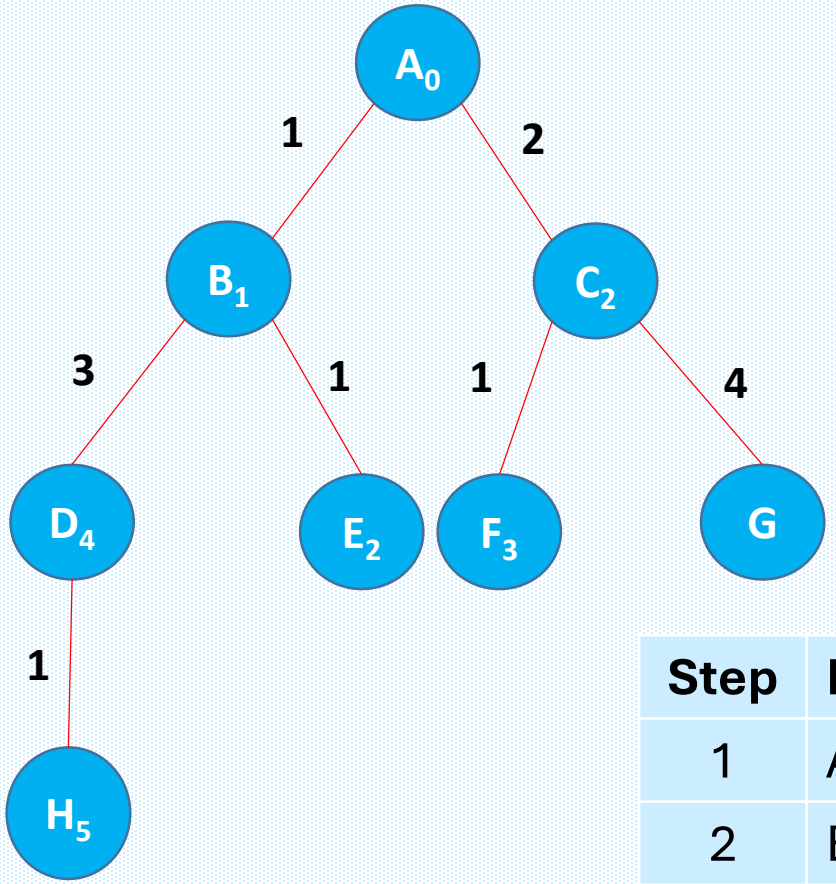
Goal: Find shortest path from **A** to **F**.

Step	Priority Queue (node:cost)	Expand Node	New Paths Added
1	A:0	A	B:1, C:4
2	B:1, C:4	B	D:3, E:6
3	D:3, C:4, E:6	D	F:4
4	F:4, C:4, E:6	F	**Goal Reached**

Result:

- **Actual Path:** A → B → D → F, **Cost:** 4
- **Traversed Path:** A → B → D → F

Uniform Cost Search (UCS)



Edge Weights:

- A → B = 1
- A → C = 2
- B → D = 3
- B → E = 1
- C → F = 1
- C → G = 4
- D → H = 1

Goal: Find shortest path from **A** to **F**.

Step	Priority Queue (node:cost)	Expand Node	New Paths Added
1	A:0	A	B:1, C:2
2	B:1, C:2	B	D:4, E:2
3	C:2, E:2, D:4	C	F:3, G:6
3	E:2, F:3, D:4, G:6	E	-
4	F:3, D:4, G:6	F	**Goal Reached**

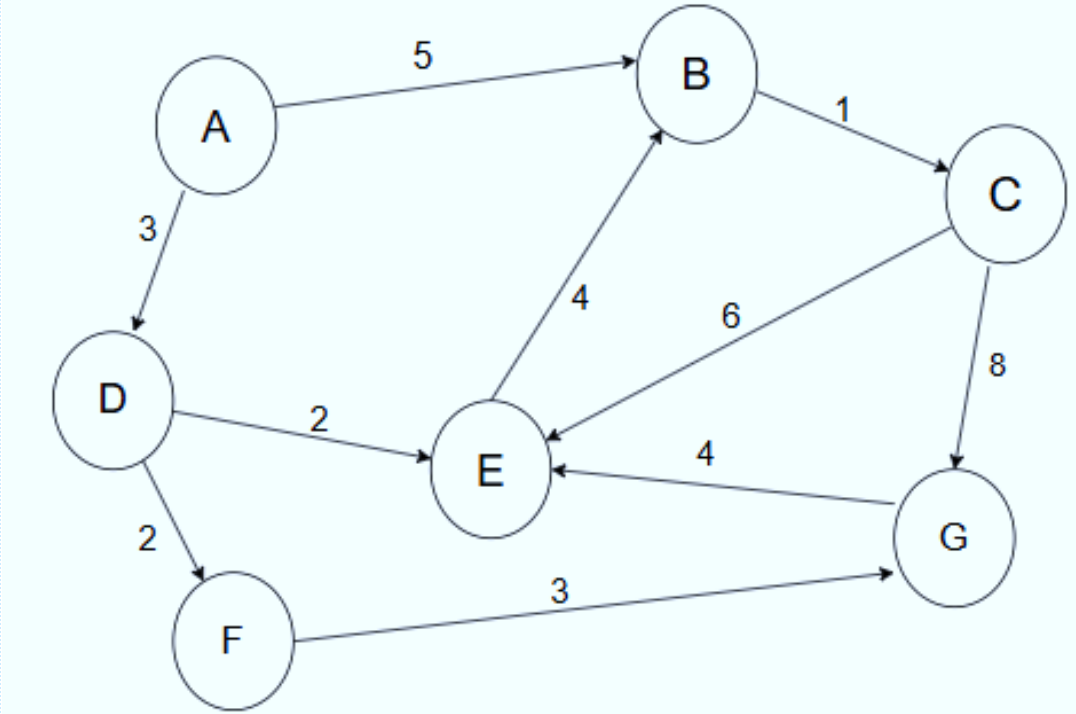
Result: Actual Path: A → C → F, **Cost:** 3

Traversed Path: A → B → C → E → F

Uniform Cost Search (UCS)

S. No.	Priority Queue (node:cost)	Expand Node	New Paths Added
1	{(A:0)}	A	D:3, B:5
2	{(A-D:3), (A-B:5)}	D	E:5, F:5
3	{(A-B:5), (A-D-E:5), (A-D-F:5)}	B	C:6
3	{(A-D-E:5), (A-D-F:5), (A-B-C:6)}	E	B:9
4	{(A-D-F:5), (A-B-C:6), ~~(A-D-E-B:9)~~} Here B is already explored	F	G:8
5	{(A-B-C:6), (A-D-F-G:8)}	C	E:12, G:14
6	{(A-D-F-G:8), ~~(A-B-C-E:12)~~, (A-B-C-G:14)} Here E is already explore	G	E:18 **E already visited**
7	{(A-D-F-G:8)}	Null	**Goal Found**

Finding path from A to G



Hence we get:

- Actual path: A → D → F → G , with Path Cost = 8.
- Traversed path: A → D → B → E → F → C → G

Difference between BFS & DFS

Feature	BFS	DFS
Full Form	Breadth-First Search	Depth-First Search
Search Method	Level by level	Branch by branch
Data Structure	Queue	Stack
Memory Usage	Higher	Lower
Completeness	Complete	Not always complete
Shortest Path	Finds shortest path when all step costs are equal	May not find shortest path
Speed	Can be slower due to high memory usage	Can reach a deep solution quickly
Best Use	Finding the shortest path in an unweighted problem	Problems where solutions may be deep

Best First Search (BFS)

General Concept:

- It's a **search strategy** where you expand the node that appears to be the "best" according to some evaluation function **f(n)**.
- The function **f(n)** can be defined in different ways, for example:
  - **Greedy Best First Search:** f(n) = h(n) → Uses *only heuristic* (estimated cost to goal).
  - **A*** Search: f(n) = g(n) + h(n) → Uses *path cost* + *heuristic*.

So BFS is a **framework** — GBFS & A* are just ways to define "best."

Greedy Best First Search (GBFS)

- GBFS is an informed search algorithm that uses heuristics to decide which node to explore next.
- GBFS is a search algorithm that **expands the node which appears to be closest to the goal** based on a **heuristic function** h(n).
- **Evaluation Function:**

  f(n)=h(n)

  - h(n) = estimated cost from node n to the goal.
  - It **ignores** g(n) (the cost from start to current node).
- Reach the goal quickly by "greedily" following the most promising path according to the heuristic.
- **Important Characteristics**
  - **Informed Search:** Uses heuristic information.
  - **Not Optimal:** May not find the shortest path (since it ignores path cost).
  - **Complete:** Will find a solution if the search space is finite and a path exists.
  - **Speed:** Usually faster than algorithms like A*, but can be misled by poor heuristics.
  - **Risk:** Can get stuck in loops or dead ends without proper checks.

Greedy Best First Search (GBFS)

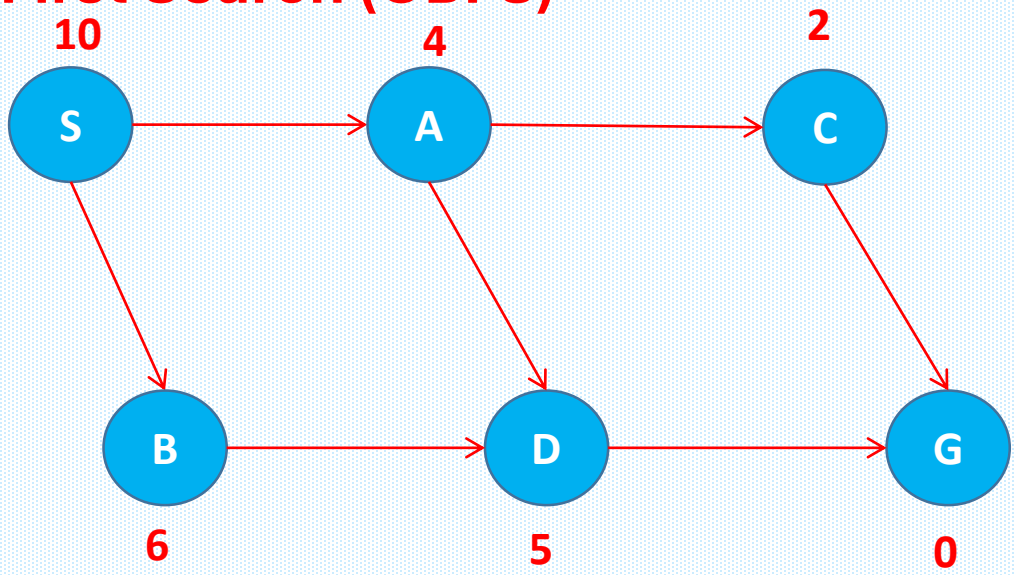
- **Best First Search (BFS)** complexity:
  - **Time Complexity:** O(b^d) in the worst case, where **b** = branching factor and **d** = maximum depth. This is because it may explore many nodes before finding the goal.
  - **Space Complexity:** O(b^d), since it stores all generated nodes in memory (can be large for big graphs).
  - **Optimality:** Not guaranteed — it may find a suboptimal path if the heuristic is not perfect.
  - **Completeness:** Yes, if the branching factor is finite and the goal is reachable.
- **Advantages:**
  - Simple and fast.
  - Useful when the heuristic is accurate.
- **Disadvantages:**
  - Not guaranteed to be optimal.
  - Performance highly depends on heuristic quality.
  - Can follow misleading paths.

Greedy Best First Search (GBFS)

Algorithm Steps

1. Start at the **initial node.**
2. Use h(n) to estimate the distance from each node to the goal.
3. Initialize a **priority queue** with the start node.
4. Mark the start node as visited.
5. While the queue is not empty:
    1. Remove the node with the **lowest h(n)** from the queue.
    2. If it is the **goal node**, stop.
    3. Otherwise, add all **unvisited neighbors** to the queue with their heuristic values.
6. Repeat until the goal is found or no nodes remain.

Greedy Best First Search (GBFS)



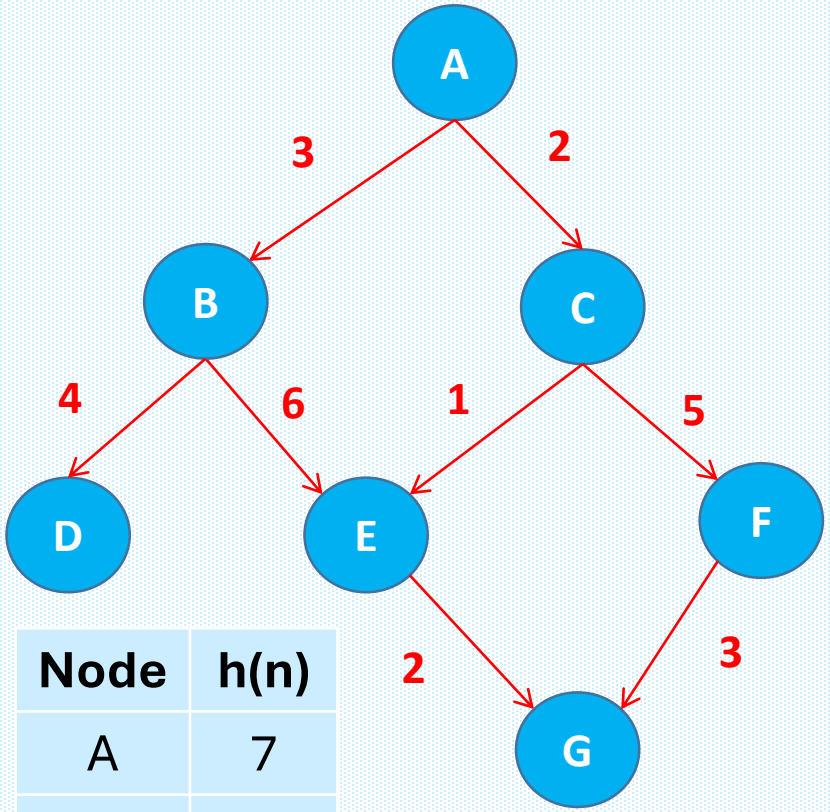
Node	h(n)
S	10
A	4
B	6
C	2
D	5
G	0

Step-by-step GBFS Traversal

1. **Start at S (h=10)** → Queue = [S]
2. Expand S → Neighbors: A (h=4), B (h=6) → Queue = [A(4), B(6)]
3. Expand A (lowest h=4) → Neighbors: C (h=2), D (h=5) → Queue = [C(2), B(6), D(5)]
4. Expand C (lowest h=2) → Neighbor: G (h=0) → Queue = [G(0), B(6), D(5)]
5. Expand G → **Goal found**

Path: S → A → C → G

Greedy Best First Search (GBFS)



Node	h(n)
A	7
B	6
C	2
D	10
E	4
F	6
G	0

Step-by-step GBFS Traversal

1. **Start:** A → h(A)=7

   Queue: {A}

2. **Expand A:** Neighbours B(h=6), C(h=2)

   Queue: {C(2), B(6)} → Pick C (smallest h)

3. **Expand C:** Neighbours E(h=4), F(h=6)

   Queue: {E(4), B(6), F(6)} → Pick E (smallest h)

4. **Expand E:** Neighbour G(h=0)

   Queue: {G(0), B(6), F(6)} → Pick G **Goal reached!**

Path: A → C → E → G

A* Search

- **A* Search** is an informed ***best-first search algorithm*** that efficiently determines the lowest cost path between any two nodes in a directed weighted graph with non-negative edge weights.

- This algorithm is a variant of Dijkstra's algorithm. A slight difference arises from the fact that an evaluation function is used to determine which node to explore next.

- A* (pronounced "A Star") is a **graph search algorithm** that finds the **least-cost path** from a **start node** to a **goal node** by combining the benefits of **Uniform Cost Search** and **Greedy Best First Search**.

A* Search

- It uses evaluation function:

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

  Where:

  g(n) → cost from start node to current node.

  h(n) → heuristic (estimated cost from current node to goal).

  f(n) → estimated total cost of path through node **n** to the goal.

- If the heuristic is **admissible** (never overestimates) and **consistent** (monotonic), A* **always finds the optimal path**.
- Balances between:
  - **UCS** (safe, but can be slow — explores many nodes).
  - **GBFS** (fast, but can give suboptimal paths — ignores path cost so far).

A* Search

Step-by-Step Algorithm

1. **Initialize**: Put the start node in a **priority queue** with priority f(start) = g(start) + h(start)
   - For start node: g = 0, f = h (start)
2. **Repeat until queue is empty**:
   - Remove the node with **smallest f(n)**.
   - If it's the goal → **return path** (this is optimal).
   - For each neighbor:
     - Calculate g(new) = g(current) + cost(current → neighbor)
     - Calculate h(new) = heuristic estimate to goal
     - Calculate f(new) = g(new) + h(new)
     - If neighbor not visited or found with lower f → update and push into priority queue.
3. **End**: If queue empty without finding goal → no path exists.

A* Search

Advantages:

- **Optimal**: This algorithm is optimal as it guarantees the shortest possible path.
- **Efficient**: When an appropriate heuristic is used, it is faster than all other algorithms.
- **Flexible**: The algorithm is available to adapt to various types of graphs and different heuristics.

Disadvantages:

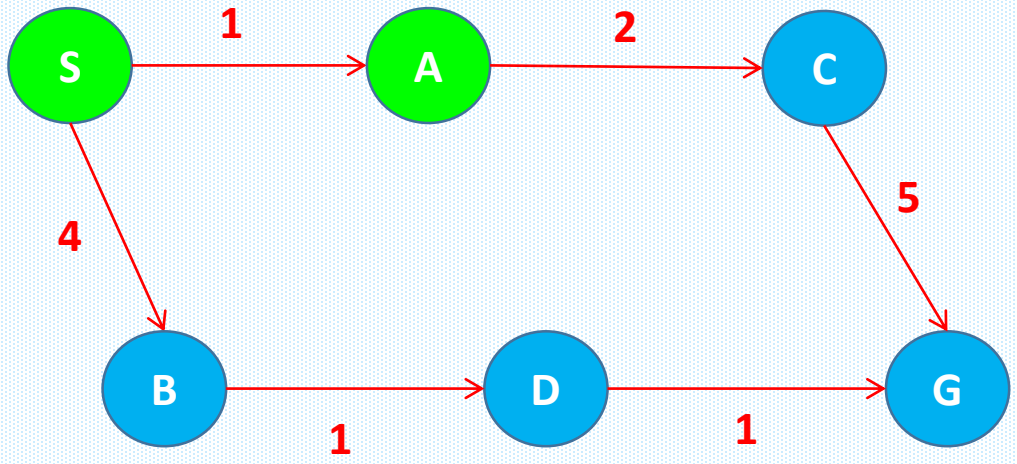
- **Heuristic Dependent**: The performance of this algorithm is heavily dependent on the choice of heuristics.
- **Memory Intensive**: This algorithm stores all the nodes in memory, which is not suitable for large graphs.

A* Search

A* Applications

1. **Pathfinding in Games & Robotics:** Finding the shortest or optimal path in maps, grids, or navigation systems (e.g., NPC movement in video games, robot path planning).
2. **Network Routing:** Used in computer networks (e.g., packet routing, communication protocols) to find least-cost paths.
3. **Puzzle Solving:** Classic problems like **8-puzzle, 15-puzzle, Rubik's cube**, etc., where optimal sequences of moves are needed.
4. **AI Planning:** Automated planning in AI systems (e.g., logistics, scheduling, decision-making).
5. **Speech & Language Processing:** Word segmentation, parsing, and machine translation using search in grammar trees.
6. **Robotics & Autonomous Vehicles:** Path planning for drones, self-driving cars, and delivery robots.

A* Search



Step	Node Expanded	g(n)	h(n)	f(n) = g(n)+h(n)	Queue (f values)
1	S	0	7	7	A(7), B(8)
2	A	1	6	7	B(8), C(5+2=7)
3	C	3	2	5	B(8), G(8)
4	G	8	0	8	— goal reached

Node	h(n)
S	7
A	6
B	4
C	2
D	1
G	0

Init

OPEN = { **S**: g=0, h=7 → **f=7** }

CLOSED = { }

Step 1: Pop S (min f=7), expand

Neighbours:

- **A:** g=0+1=1, h=6 → **f=7** (parent S)
- **B:** g=0+4=4, h=4 → **f=8** (parent S)

OPEN = { **A(f=7)**, B(f=8) }

CLOSED = { S }

Select next: A (smallest f)

Step 2: Pop A (f=7), expand

Neighbors:

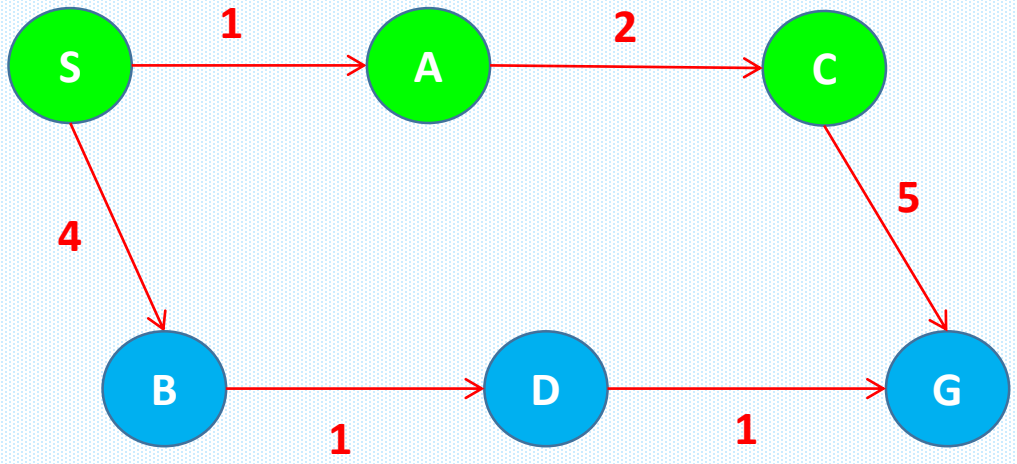
- **C:** g=1+2=3, h=2 → **f=5** (parent A)

OPEN = { **C(f=5)**, B(f=8) }

CLOSED = { S, A }

Select next: C (smallest f)

A* Search



Step	Node Expanded	g(n)	h(n)	f(n) = g(n)+h(n)	Queue (f values)
1	S	0	7	7	A(7), B(8)
2	A	1	6	7	B(8), C(5+2=7)
3	C	3	2	5	B(8), G(8)
4	G	8	0	8	— goal reached

Node	h(n)
S	7
A	6
B	4
C	2
D	1
G	0

Step 3: Pop C (f=5), expand

Neighbors:

- **G:** g=3+5=8, h=0 → **f=8** (parent C)

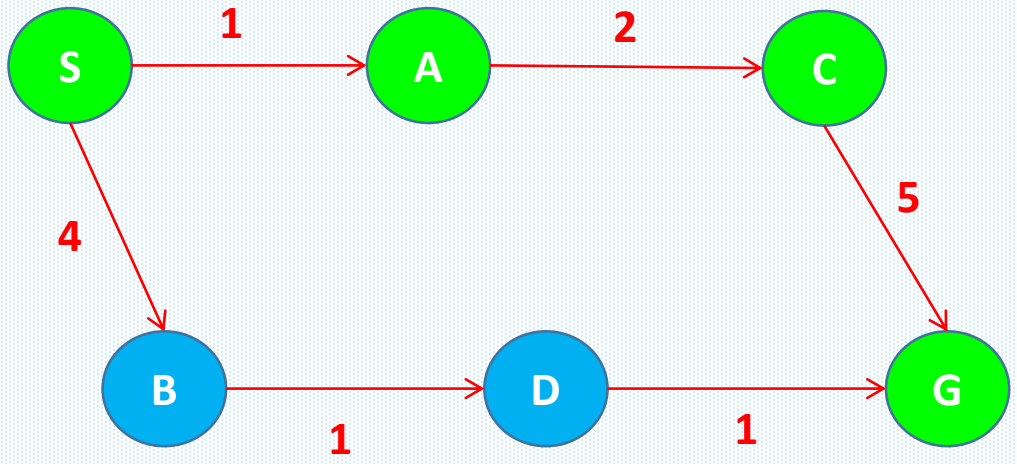
OPEN = { B(f=8), G(f=8) }

CLOSED = { S, A, C }

Now there's a **tie** (both f=8).

At this point, two things can happen depending on tie-breaking:

A* Search



Step	Node Expanded	g(n)	h(n)	f(n) = g(n)+h(n)	Queue (f values)
1	S	0	7	7	A(7), B(8)
2	A	1	6	7	B(8), C(5+2=7)
3	C	3	2	5	B(8), G(8)
4	G	8	0	8	— goal reached

Node	h(n)
S	7
A	6
B	4
C	2
D	1
G	0

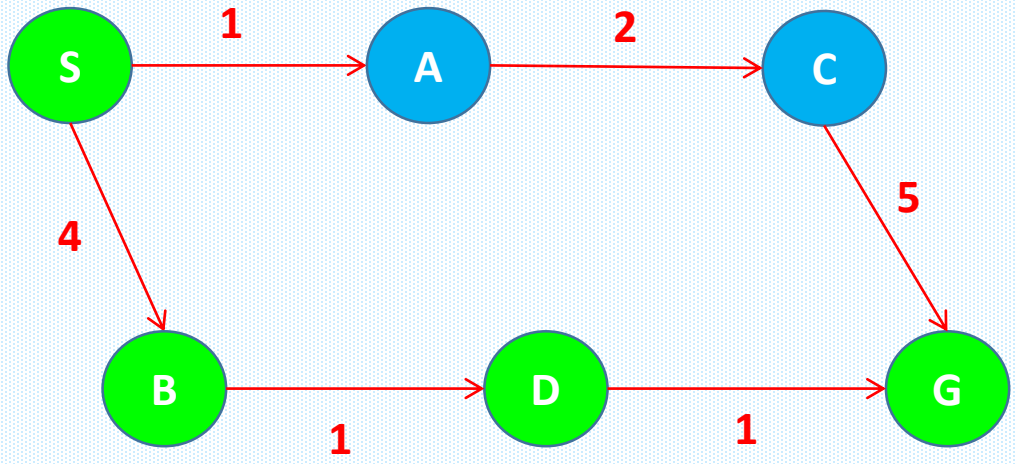
Path outcome depends on tie-break

Case A — Pick G(f=8) next

- Pop **G** → goal reached with **g=8**
- Reconstruct path: **S → A → C → G** (cost **8**)

This is **not** the cheapest possible path in this graph.

A* Search



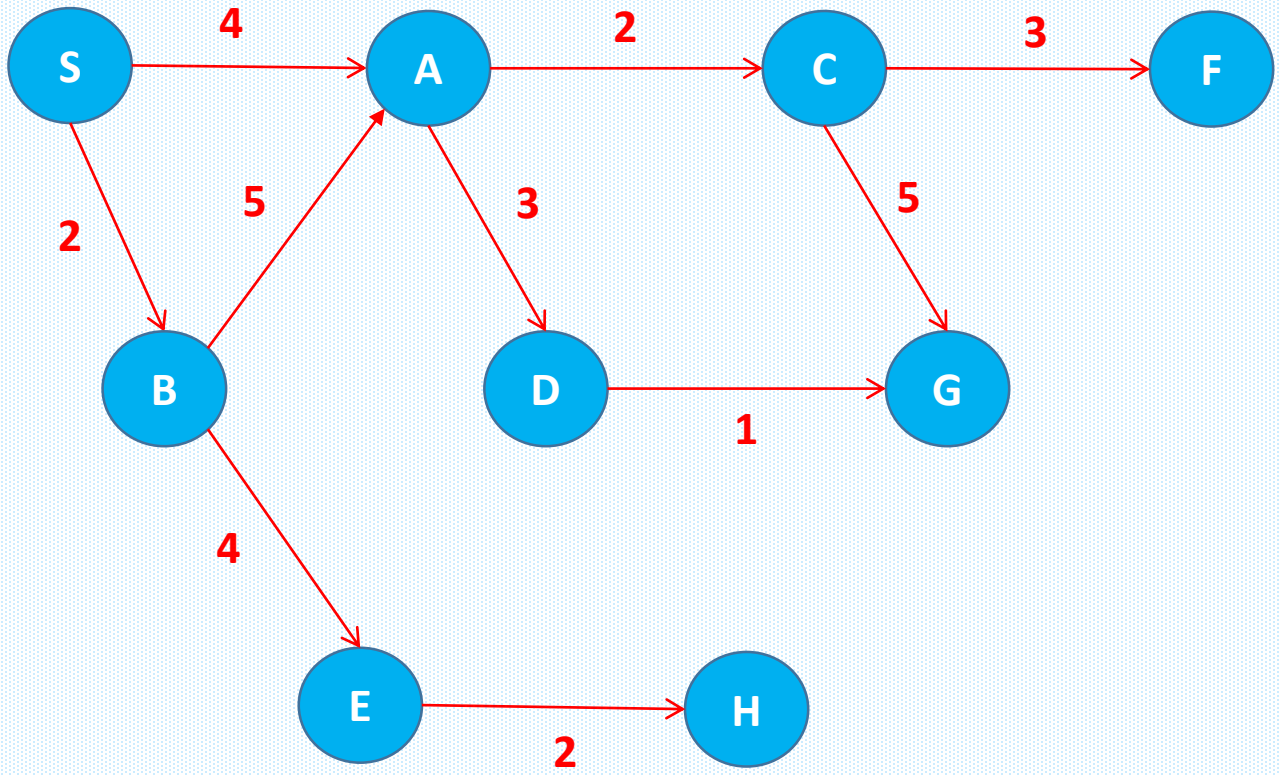
Step	Node Expanded	g(n)	h(n)	f(n) = g(n)+h(n)	Queue (f values)
1	S	0	7	7	A(7), B(8)
2	A	1	6	7	B(8), C(5+2=7)
3	C	3	2	5	B(8), G(8)
4	G	8	0	8	— goal reached

Node	h(n)
S	7
A	6
B	4
C	2
D	1
G	0

Case B — Pick B(f=8) next (common tie-breaks do this)

- Pop **B**, expand:
  - **D:** g=4+1=5, h=1 → **f=6** (parent B)
  - OPEN = { **D(f=6)**, G(f=8) } → pick **D**
- Pop **D**, expand:
  - **G:** g=5+1=6, h=0 → **f=6** (better than old G)
  - OPEN = { **G(f=6)**, (old G f=8 outdated) } → pop **G**
  - Goal reached with **g=6**
- Reconstruct path: **S → B → D → G** (cost **6**) (cheapest)

A* Search



Node	h(n)
S	7
A	6
B	4
C	5
D	2
E	6
F	4
G	0
H	5

Step 1: Start at S

- g(S) = 0
- f(S) = 0 + 7 = **7**

Open list: { S }
Closed: { }

Step 2: Expand S

From S:

- **A**: g = 4, h = 6 ⇒ f = 4 + 6 = 10
- **B**: g = 2, h = 4 ⇒ f = 2 + 4 = **6**

Open list: {B(6), A(10)}
Choose **B** (lowest f).
Closed: { S }

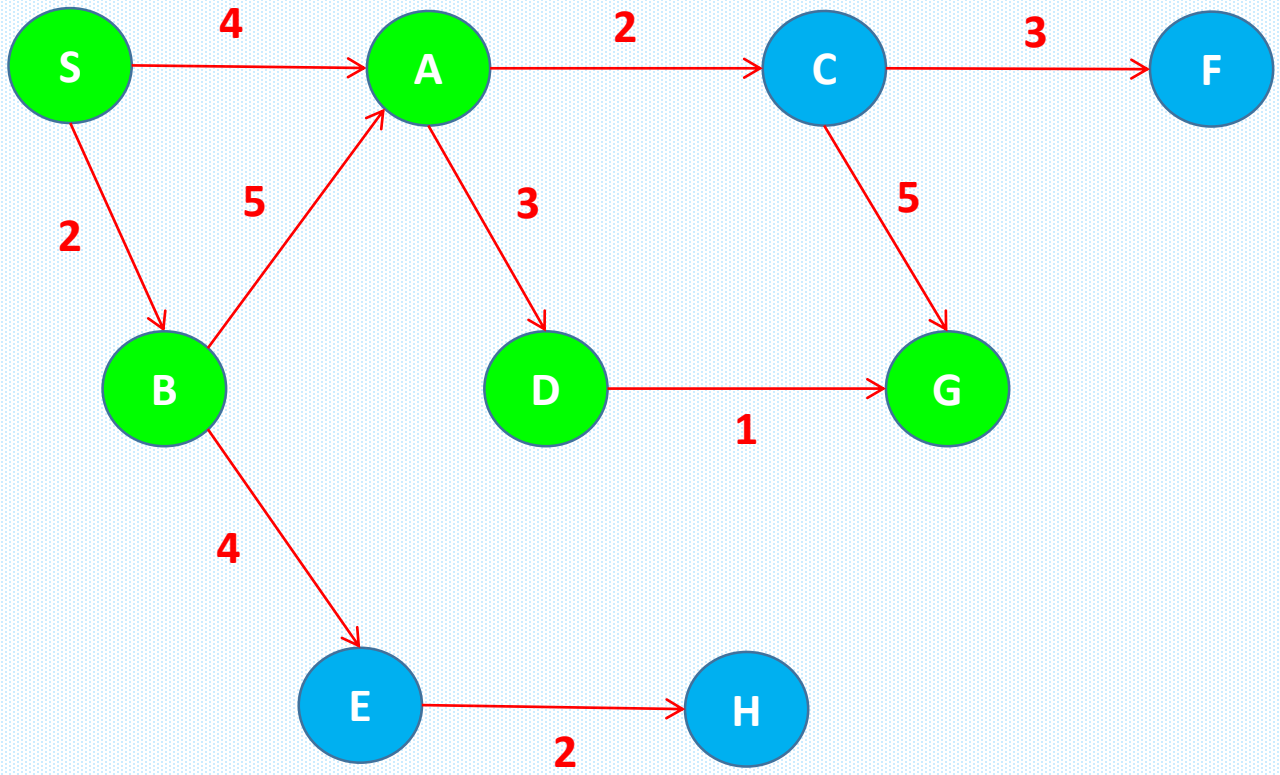
Step 3: Expand B

From B:

- **E**: g = 2 + 4 = 6, h = 6 ⇒ f = 12
- **A** (via B): g = 2 + 5 = 7, h = 6 ⇒ f = 13 (worse than earlier A's 10 ⇒ discard)

Open list: {A(10), E(12)} ⇒ Choose **A** (lowest f).
Closed list: { S, B }

A* Search



Node	h(n)
S	7
A	6
B	4
C	5
D	2
E	6
F	4
G	0
H	5

Step 4: Expand A

From A:

- **C**: g = 4 + 2 = 6, h = 5 ⇒ f = 11
- **D**: g = 4 + 3 = 7, h = 2 ⇒ f = **9**

Open list: {D(9), C(11), E(12)}

Choose **D** (lowest f).

Closed list: { S, B, A }

Step 5: Expand D

From D:

- **G**: g = 7 + 1 = 8, h = 0 ⇒ f = **8**

Open list: {G(8), C(11), E(12)} ⇒ Choose **G**.

Closed list: { S, B, A, D }

Step 5: Goal Test

Expanded node is **G** → **stop**.

Reconstruct path (via parents):

G ← D ← A ← S ⇒ **Path: S → A → D → G**

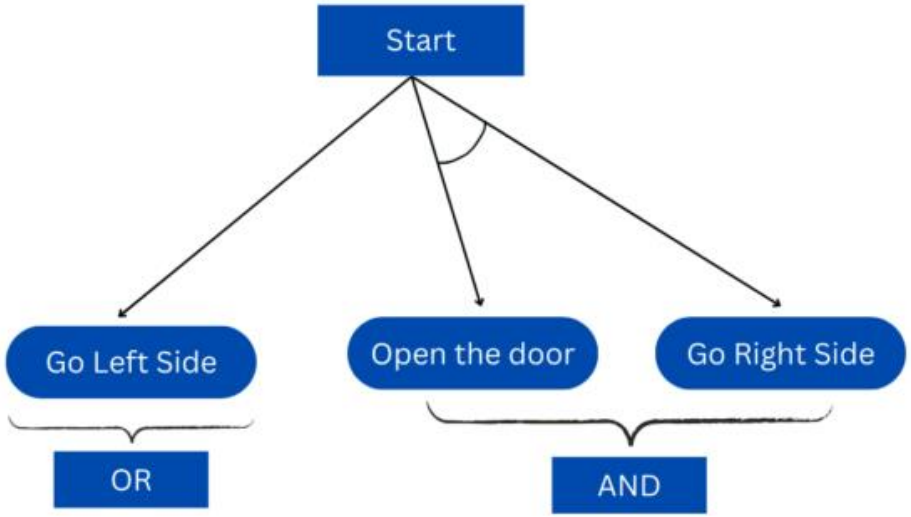
Total cost: g(G) = 8

AO* Search

- The term “AO*” or “AO-Star Algorithm” can indeed stand for “Anytime Repairing A*” or “Anytime Optimistic”, depending on the context and the specific variant of the algorithm being discussed. Both interpretations are valid and are used in different contexts within the field of artificial intelligence and robotics.
- **Break it Down:** AO-Star Algorithm uses a special kind of graph called an AND-OR graph to represent the search space. This graph breaks down the problem into smaller sub-problems, making it easier to handle.
  - **AND Nodes:** These represent subgoals that must all be achieved to reach the final goal. Think of them as checkpoints you need to pass through.
  - **OR Nodes:** These represent different ways to achieve a subgoal. Imagine multiple paths leading to the same checkpoint.
- **Informed Search:** AO* is an informed search algorithm, meaning it uses a heuristic to estimate the cost of reaching the goal from any point. This heuristic is like a good guess, guiding the search towards the most promising paths.

AO* Search

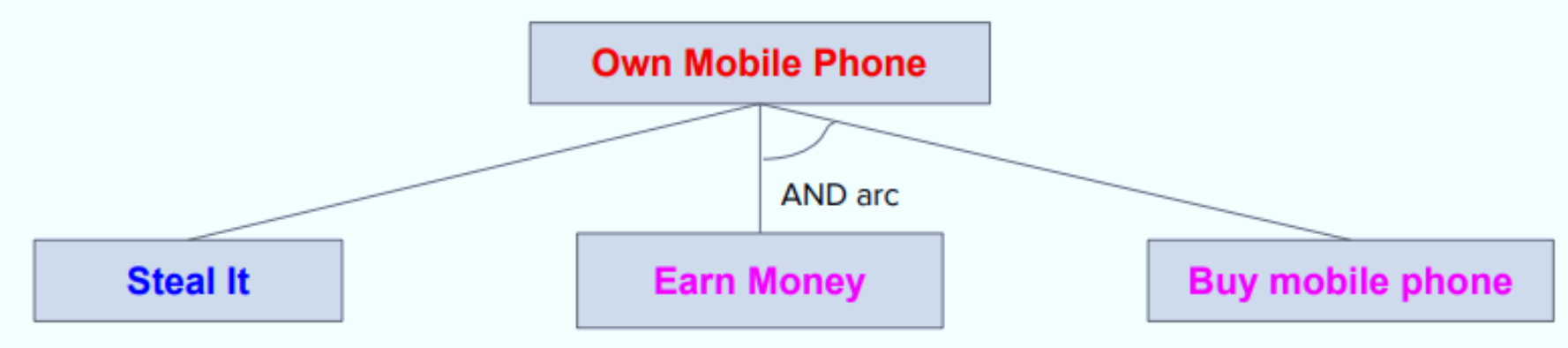
- **Adapting to Change:** The magic of AO* lies in its ability to adapt. If the environment changes (e.g., a wall appears), AO* can re-evaluate the costs of affected paths and adjust its search accordingly. It doesn't have to start from scratch, saving time and effort.
- AO* is often used for the common pathfinding problem in applications such as video games, but was originally designed as a general graph traversal algorithm, It finds applications in diverse problems, including the problem of parsing using stochastic grammars in NLP. Other cases include an Informational search with online learning. It is useful for searching game trees, problem solving etc.



AO* Search

❖ AND-OR Graph

- AND-OR graph is useful for representing the solution of problems that can be solved by decomposing them into a set of smaller problems, all of which must then be solved.



- Node in the graph will point both down to its successors and up to its parent nodes.
- Each Node in the graph will also have a heuristic value associated with it.

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

f(n): Cost function.

g(n): Actual cost or Edge value

h(n): Heuristic/Estimated value of the nodes

AO* Search

Working process

1. Nodes can be **OR** (choose one child) or **AND** (must solve **all** children).
2. Each node has a heuristic estimate h(n).
3. AO* uses f(n)=g(n)+h(n) to rank options, where g(n) is the cost so far.
4. For an **OR** node: cost = minimum cost among its alternative child subgraphs.
5. For an **AND** node: cost = **sum** of the costs of all children (plus incoming edge costs).
6. AO* repeatedly:
    1. Pick the most promising partial solution (lowest f),
    2. Expand it,
    3. Update/fix costs back up the graph,
    4. Repeat until the solution subgraph is fully solved.

AO* Search

Advantages of AO*

- ❖ Handles AND-OR problems unlike A*, making it suitable for complex hierarchical tasks.
- ❖ Optimal solution graph (not just a single path) is found.
- ❖ Dynamic revision of costs (backtracking and updating when better estimates are found).
- ❖ Works well in problem reduction tasks where solutions depend on multiple sub-solutions.

Disadvantages of AO*

- ❖ Computationally expensive – especially with large AND-OR graphs.
- ❖ Requires admissible heuristics (must not overestimate) for optimal results.
- ❖ Memory intensive – needs to store partial solution graphs.
- ❖ More complex to implement compared to A*.
- ❖ If heuristic is poor → performance degrades badly.

AO* Search

AO* is mainly used where problems can be represented as AND-OR graphs, i.e., where solving a problem may require solving multiple subproblems together (AND) or choosing one among alternatives (OR).

1. **Problem Reduction in AI:** Breaking down complex tasks into subproblems (e.g., theorem proving, diagnosis).
2. **Planning & Scheduling:** Task planning where some tasks require multiple subtasks to be completed together.
3. **Expert Systems:** Medical diagnosis, troubleshooting, or decision support where multiple conditions must be satisfied.
4. **Game Playing:** AND-OR game trees where moves depend on multiple submoves or strategies.
5. **Natural Language Processing (NLP):** Parsing where a phrase can be interpreted in different ways (OR), and some interpretations require combined rules (AND).

AO* Search Example 1

Here, in the above example all numbers in brackets are the heuristic value i.e. h(n). Each edge is considered to have a value of 1 by default.

Step-1

Starting from node A, we first calculate the best path.

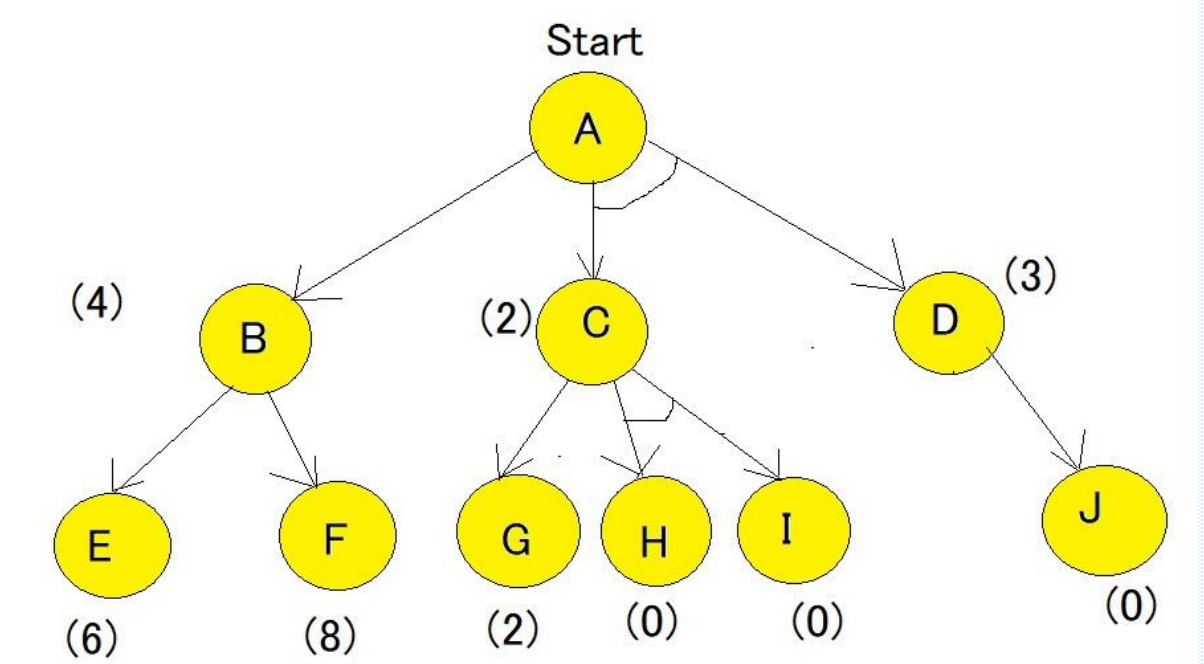
f(A-B) = g(B) + h(B) = 1+4= 5,

where 1 is the default cost value of travelling from A to B and 4 is the estimated cost from B to Goal state.

f(A-C-D) = g(C) + h(C) + g(D) + h(D) = 1+2+1+3 = 7,

here we are calculating the path cost as both C and D because they have the AND-Arc.

The default cost value of travelling from A-C is 1, and from A-D is 1, but the heuristic value given for C and D are 2 and 3 respectively hence making the cost as 7.



The minimum cost path is chosen i.e. A-B.

AO* Search Example 1

Step-2

Using the same formula as step-1, the path is now calculated from the B node,

f(B-E) = 1 + 6 = 7.

f(B-F) = 1 + 8 = 9.

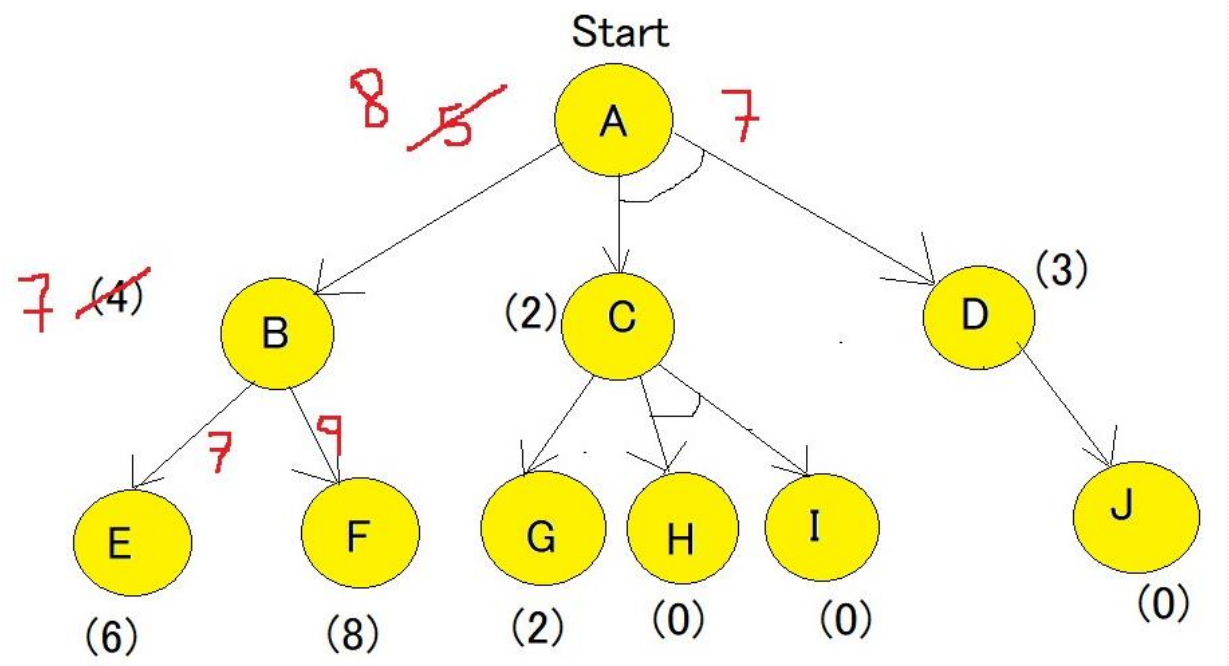
Hence, the B-E path has lesser cost.

Now the heuristics have to be updated since there is a difference between actual and heuristic value of B.

The minimum cost path is chosen and is updated as the heuristic, in our case the value is 7.

And because of change in heuristic of B there is also change in heuristic of A which is to be calculated again.

f(A-B) = g(B) + updated((h(B)) = 1+7=8



AO* Search Example 1

Step-2

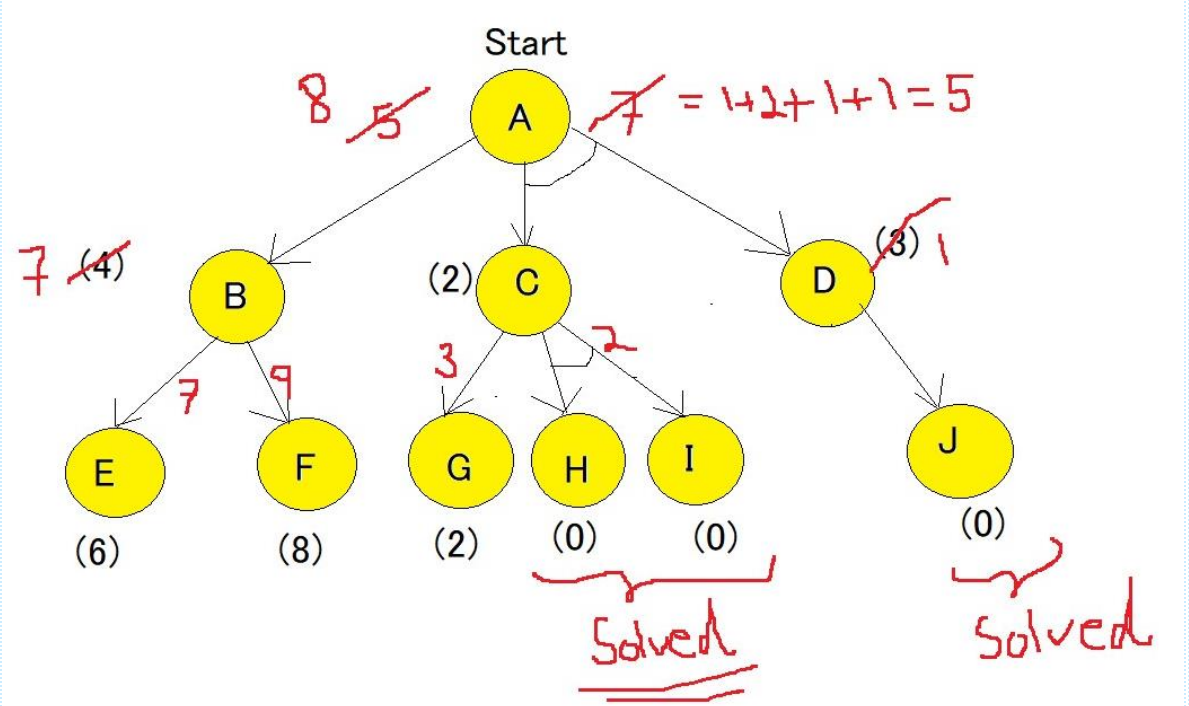
Comparing path of f(A-B) and f(A-C-D) it is seen that f(A-C-D) is smaller. Hence f(A-C-D) needs to be explored.

Now the current node becomes C node and the cost of the path is calculated,

$$f(C\text{-}G) = 1+2 = 3$$

$$f(C\text{-}H\text{-}I) = 1+0+1+0 = 2$$

f(C-H-I) is chosen as minimum cost path, also there is no change in heuristic since it matches the actual cost.



Heuristic of path of H and I are 0 and hence they are solved, but Path A-D also needs to be calculated, since it has an AND-arc.

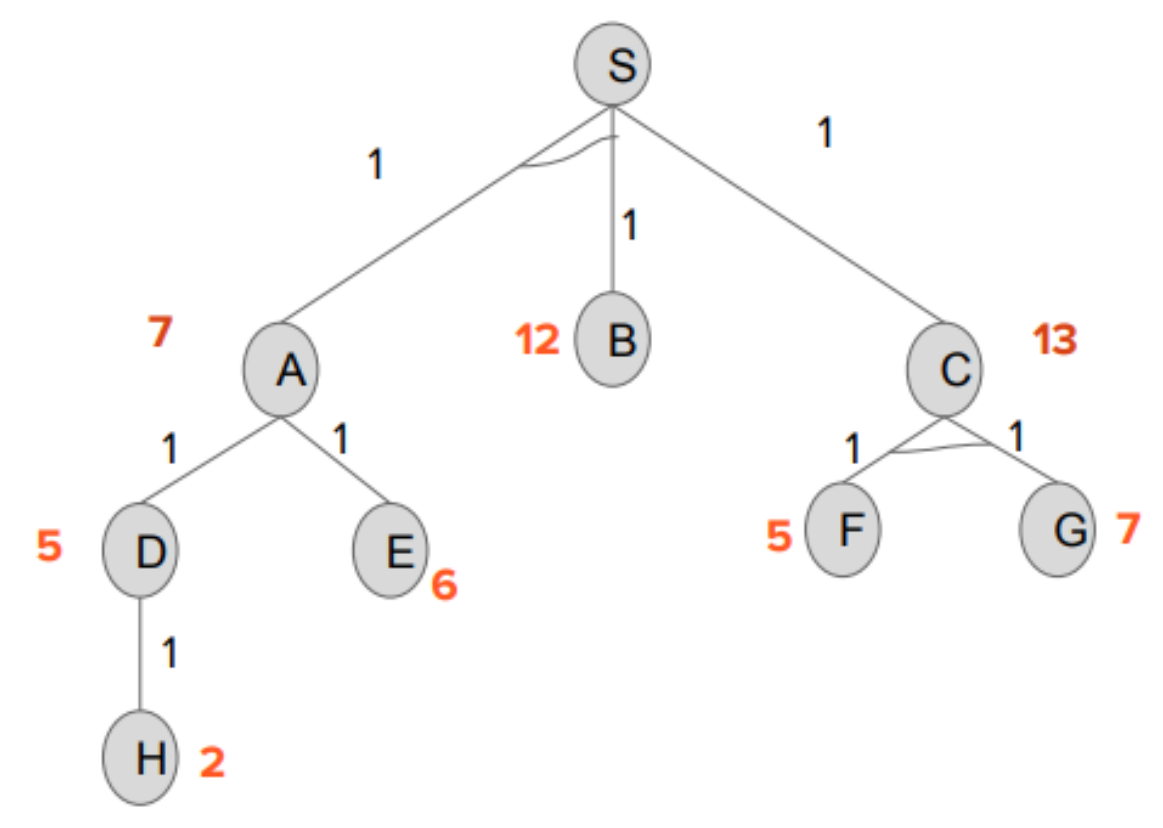
f(D-J) = 1+0 = 1, hence heuristic of D needs to be updated to 1. And finally the f(A-C-D) needs to be updated.

$$f(A\text{-}C\text{-}D) = g(C) + h(C) + g(D) + updated((h(D)) = 1+2+1+1 = 5.$$

As we can see that the solved path is f(A-C-D).

AO* Search Example 2

- Root: **S** (three branches: **A**, **B**, **C**) — each S → child edge cost = 1.
- Subtrees:
  - **A** → children **D** and **E** (each edge cost = 1). D → H (edge cost = 1).
  - **B** is a single child (orange number near B = 12).
  - **C** → children **F** and **G** (each edge cost = 1).
- Leaf heuristic values (the orange numbers next to leaves):
- h(D)=5, h(H)=2, h(E)=6, h(F)=5, h(G)=7, h(B)=12



AO* Search Example 2

Step 1 - Initialization (Compute initial f for top branches)

Compute rough f estimates from S down each branch using the available heuristics.

- **Path via C (first pass):**

  f(S → C) ≈ g(S,C) + h(C) — picture uses an initial h(C)=13 so

  $$f(S \rightarrow C) = 1+13 = 14$$

  (This is an initial estimate before expanding C's children.)

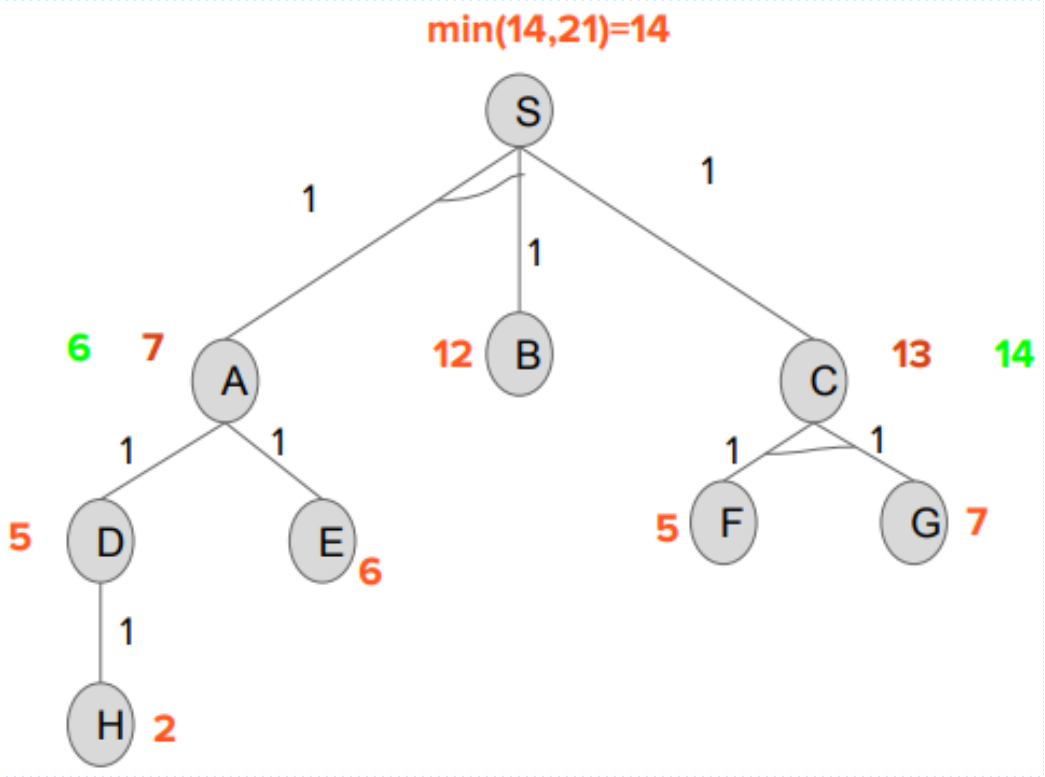
- **Path via A then B (initial):**

  The picture shows a two-step OR-AND path cost (S→A→B) computed as:

  $$f(S \rightarrow A \rightarrow B) = 1+1+7+12 = 21$$

  (that expression captures the g-costs along the path plus the h() shown for intermediate nodes.)

At this point the lowest estimated f from S is 14 (via C), so AO* will expand C first.



AO* Search Example 2

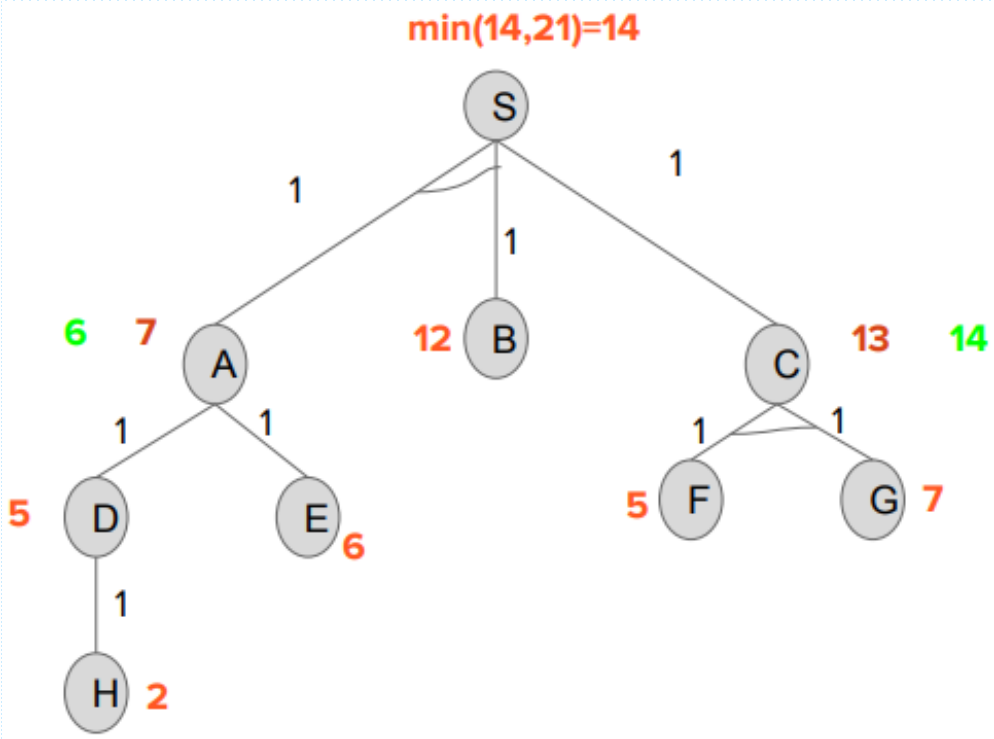
Step 2 - Expand the most promising node: C

C is an **AND node** (it requires solving **both** F and G). To evaluate C properly we must expand both children and compute the combined cost.

Compute the combined cost of the C-subgraph (as shown in the figure): Costs along the C branch and the heuristics at F and G are used:

$$f(C \rightarrow F \rightarrow G) = 1+1+5+7 = 14$$

(the figure writes this as f(C→F→G) = 14.



Now revise the S→C estimate by adding the S→C edge again (depending on how they show g accumulation). The figure reports the revised S→C cost as: revised f(S−C) = 1+14 = 15

(that is the value shown at the top: **Revised cost: 15**.)

So after fully expanding C, the AO* engine discovers the real cost to solve the C-subproblem is higher than the initial crude estimate; we update (backpropagate) that revision to S.

AO* Search Example 2

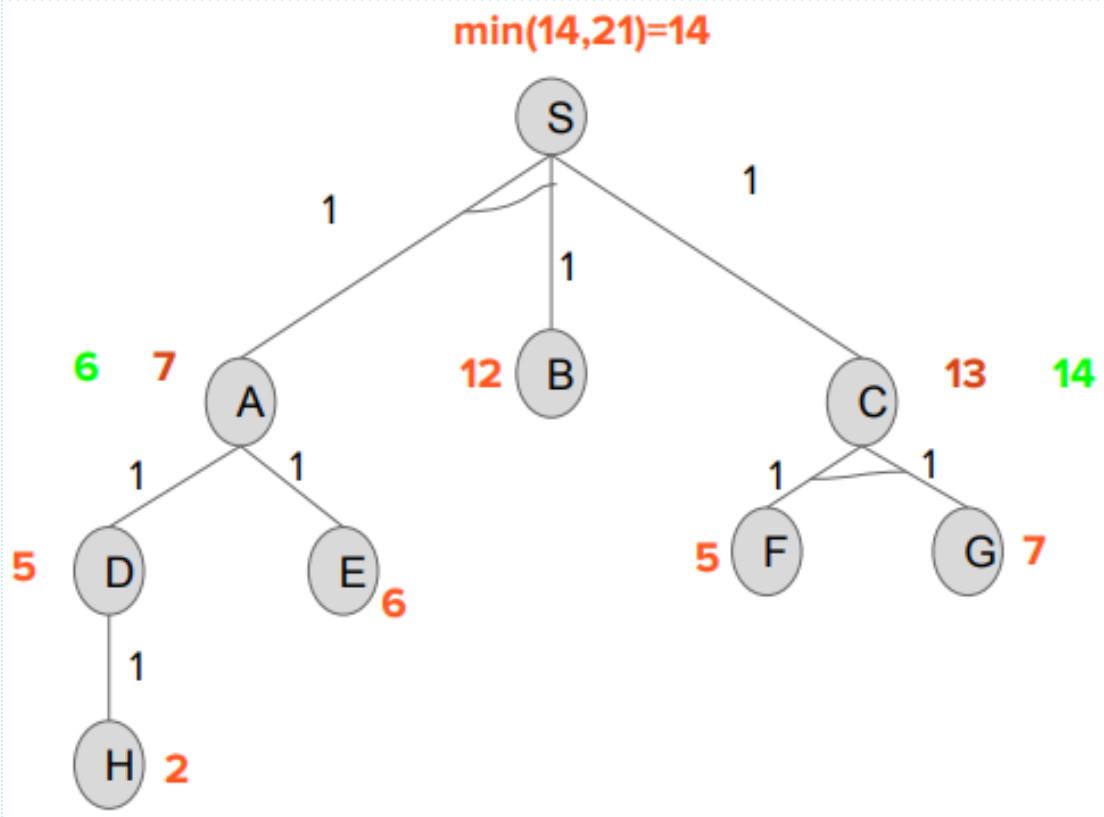
Step 3 - Reevaluate alternatives at S

Now the algorithm compares the revised costs for each top branch:

- **S→C** now has **revised f = 15** (after expansion of C).
- **S→A→B** still has the earlier computed estimate of **21**.

So **S→C (15)** currently remains the best option — but AO* will still check whether exploring the A-subtree can produce an even better solution (because an unexpanded branch may hide a cheaper combination).

Therefore AO* expands the next promising partial node on the incumbent solution graph. The figure continues by exploring **A** (to see if A's subtree can beat 15).



AO* Search Example 2

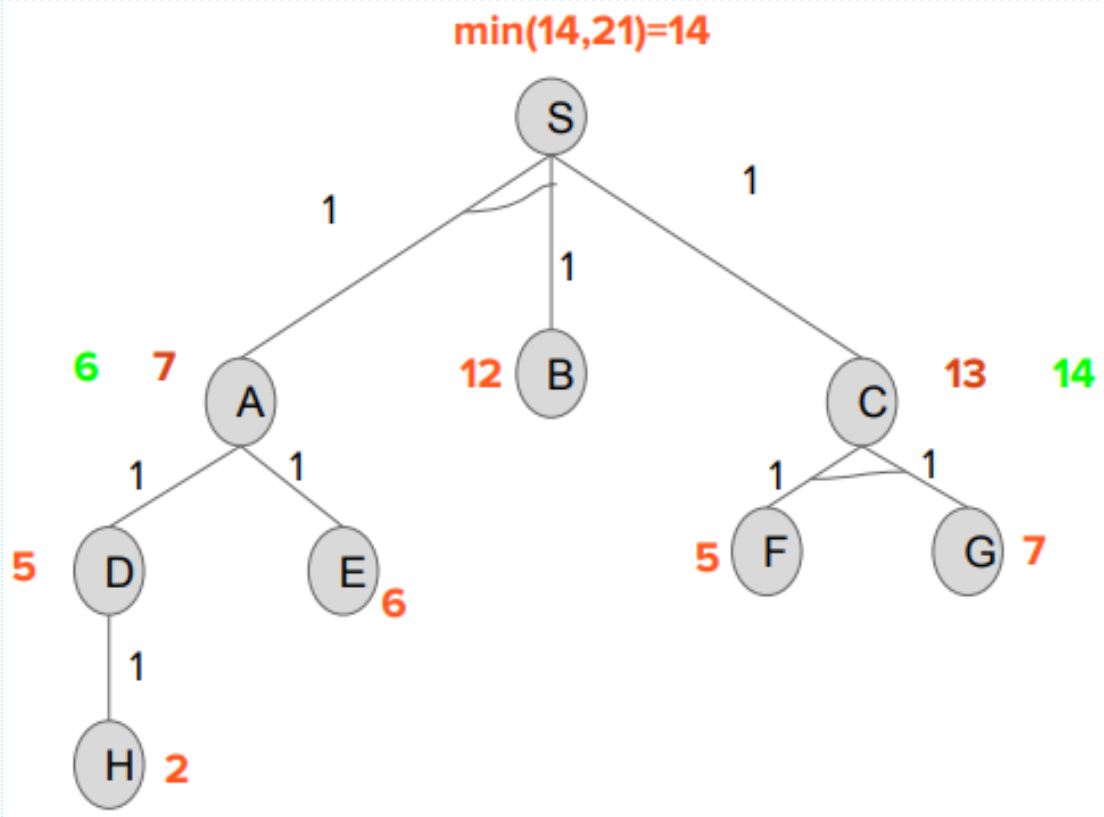
Step 4 - Expand subtree under A

Expand children of **A** to compute f(A) properly:

- Compute costs for A's children:
  - f(A – D) = (A→D) = 1+h(D) = 5 ⇒ 1+5 = 6
  - f(A – E) = (A→E) = 1+h(E) = 6 ⇒ 1+6 = 7
- For an **OR** node A, take the **minimum** of its children:

f(A)=min(6,7)=6

Now recompute the S path that goes through A and then B (if that path required also solving B, the figure shows this attempted recomputation):



- They compute a revised f(S-A-B) (including A's revised cost) and get: revised f(S – A – B) =1+6+1+12=20

Because the **revised S→A→B = 20 > revised S→C = 15**, A's subtree cannot beat the current best S→C solution.

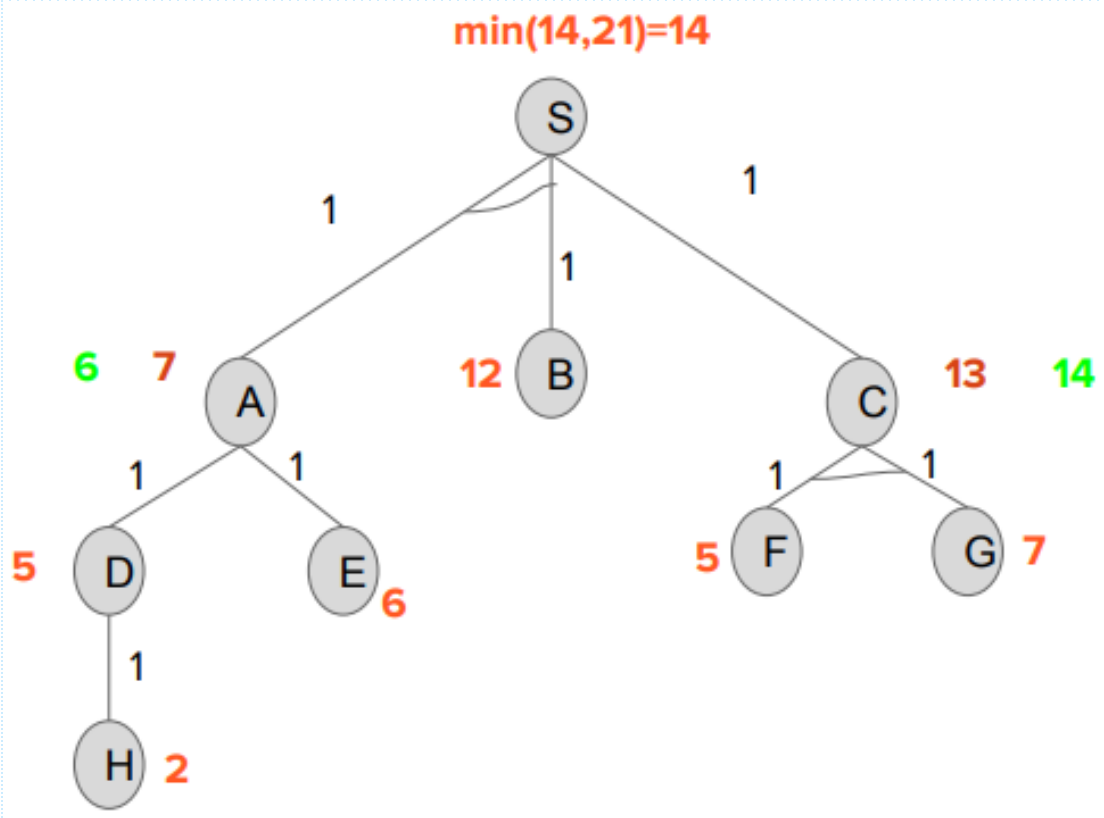
AO* Search Example 2

Step 5: Conclude & produce solution subgraph

At this point:

- S→C (cost = **15**) is the **lowest-cost way** found to solve the root, and further exploration of other branches does not reduce that cost below 15.
- The solution subgraph is therefore **S → C → {F, G}** (i.e., pick the C branch and solve both F and G since C is an AND node).

So AO* halts and returns the solution subgraph with **total revised cost = 15** and the path structure S→C with C's solutions F and G.



AO* Search Example 2 (Short Explanation)

Step 1: Start at root (S)

Two main choices:

Path S→C ≈ 14

Path S→A→B ≈ 21

Best so far = **S→C (14)** → expand C.

Step 2: Expand C (AND node)

Must solve both F and G:

Cost = 1 (S→C) + 1 (C→F) + 5 (h(F)) + 7 (h(G)) = **14**

Revised cost for S→C = 1 + 14 = **15**

Step 3: Check other path (via A and B)

Expand A:

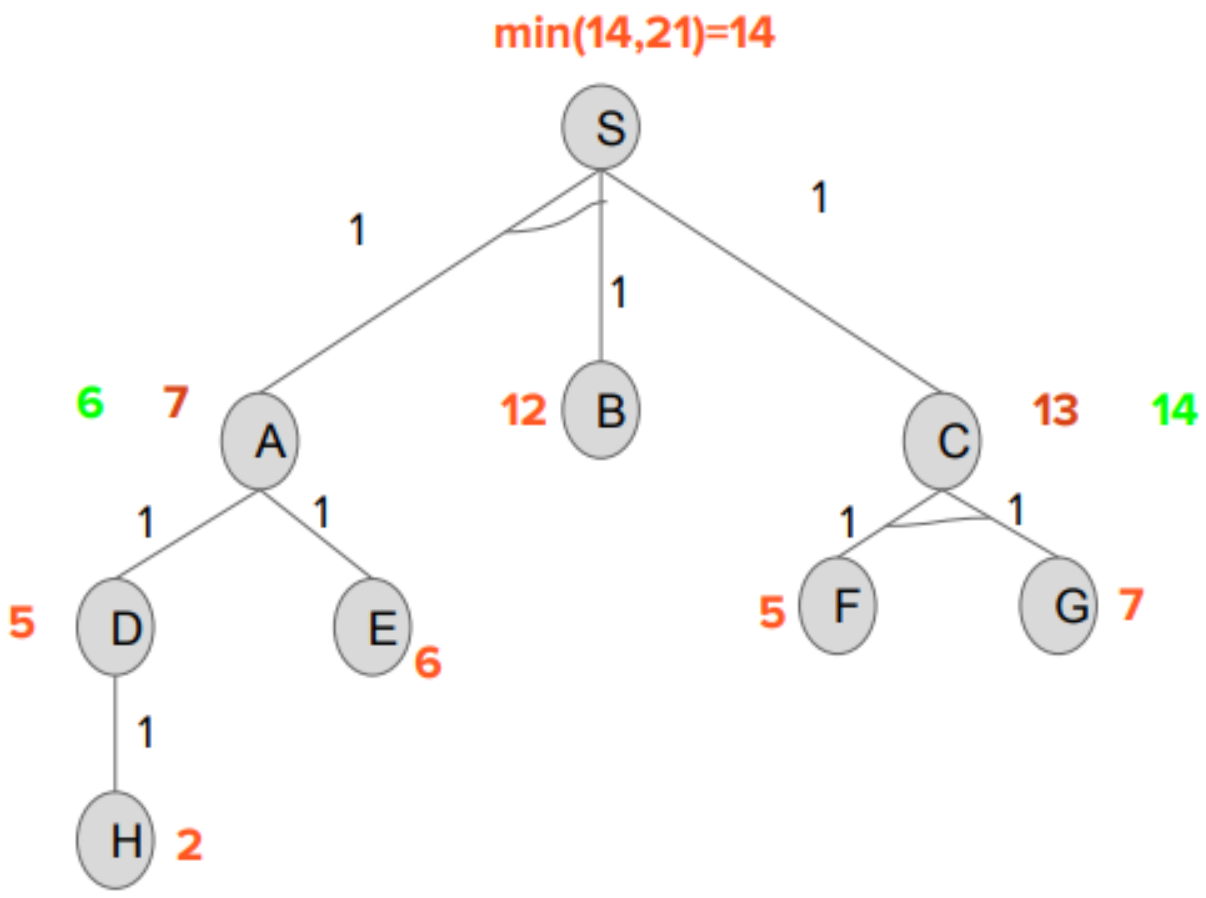
A→D = 1+5=6

A→E = 1+6=7

So A = min(6,7)=6

Then S→A→B = 1 + 6 + 1 + 12 = **20**

This is worse than 15.



Step 5: Decision

Best solution = **S→C**, with children **F and G**.

Final cost = 15

AO* Search Example 3 (Short Explanation)

Given

- Edge costs: A→B = 1, A→C = 1, A→D = 1; B→E = 1, B→F = 1; D→G = 1, D→H = 1.
- Leaf heuristic values: **E = 5**, **F = 7**, **C = 4**, **G = 4**, **H = 4**.

Calculations (child-cost only convention)

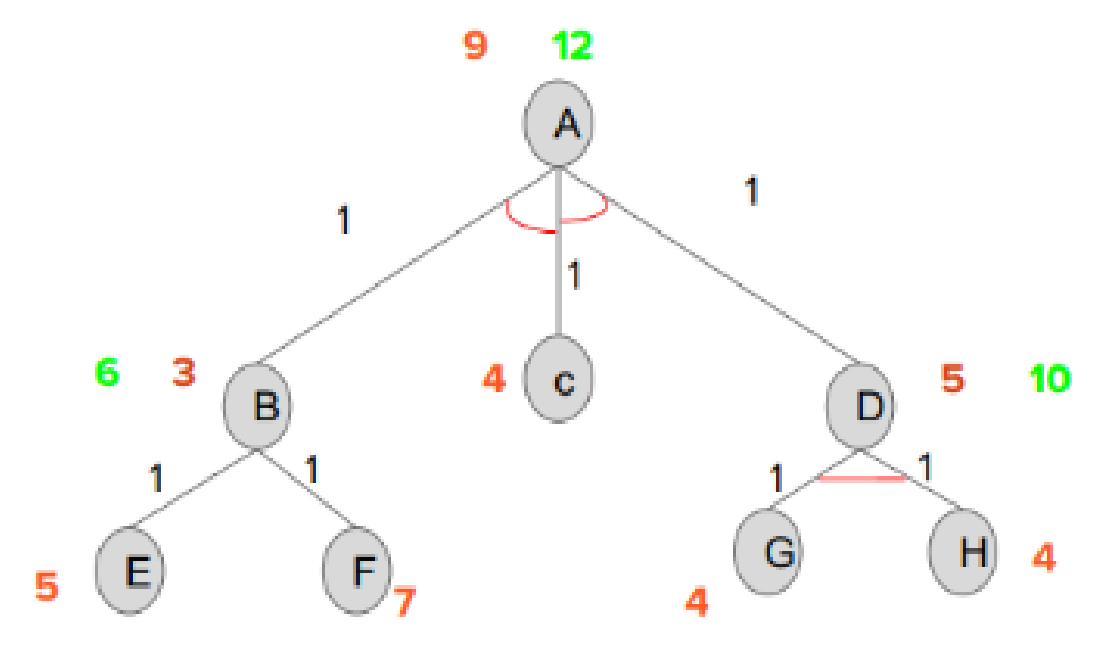
(Here f(node) = cost of solving that node **starting at the node**, not including the parent edge.)

1. **B (OR node)**
   - f(B→E) = 1 + 5 = **6**
   - f(B→F) = 1 + 7 = **8**

     → **f(B) = min(6, 8) = 6**
2. **C (leaf)**

   → **f(C) = 4**
3. **D (AND node)**
   - f(D→G) = 1 + 4 = **5**
   - f(D→H) = 1 + 4 = **5**

     → **f(D) = 5 + 5 = 10**



4. **Compare at A (add the top-edge cost 1 when we choose an option)**
   - Cost via B = A→B(1) + f(B) = 1 + 6 = **7**
   - Cost via C = A→C(1) + f(C) = 1 + 4 = **5**
   - Cost via D = A→D(1) + f(D) = 1 + 10 = **11**

Choose smallest → A → C (total = **5** including the parent edge).

A* Vs AO*

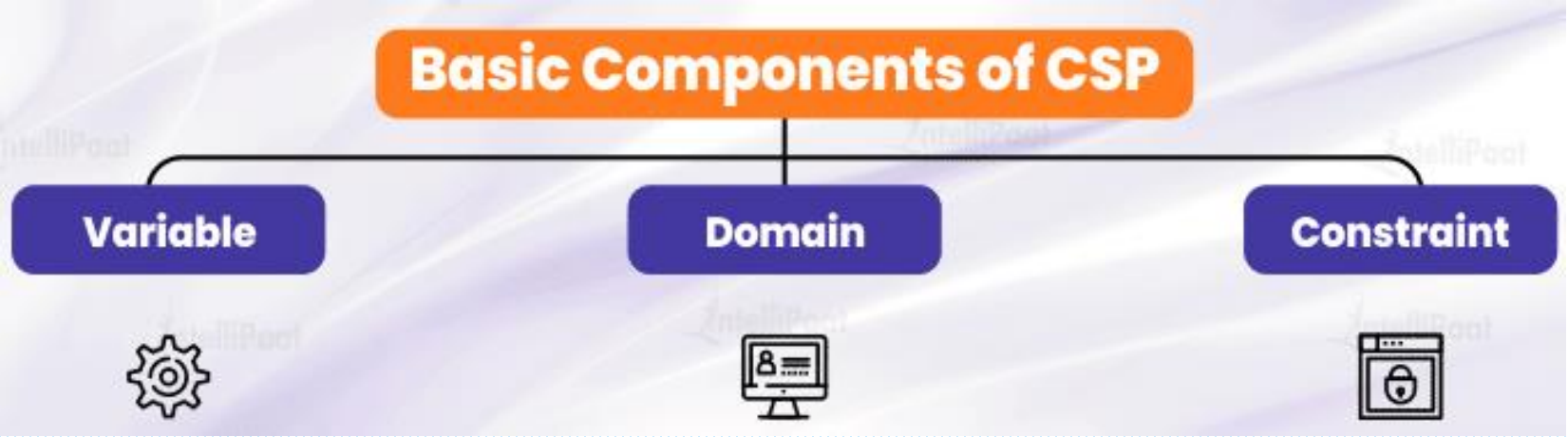
Aspect	A*	AO*
Type of Search	Graph/tree search algorithm (OR-graph)	AND-OR graph search algorithm (can represent multiple subgoals)
Problem Domain	Suitable for problems that can be represented as a single path solution (e.g., pathfinding, shortest path problems)	Suitable for problems requiring decomposition into sub-problems where solutions may involve AND/OR choices (e.g., planning, diagnosis, design problems)
Evaluation Function	Uses f(n) = g(n) + h(n) where g(n) = cost so far, h(n) = heuristic	Uses f(n) = g(n) + h(n) but evaluates costs for AND/OR nodes (must consider combined costs of subgoals)
Heuristic Requirement	Requires admissible heuristic (never overestimates cost) for optimality	Requires admissible heuristic for both AND and OR nodes
Goal Test	Finds a single path from start to goal	Finds a solution graph (may involve solving multiple subproblems simultaneously)
Expansion	Expands nodes with minimum f(n) value	Expands nodes while considering AND (all child nodes must be solved) and OR (only one child needs to be solved) conditions
Output	Returns a single optimal path	Returns an optimal solution graph (network of subgoals and decisions)
Applications	Pathfinding, robotics navigation, routing problems	Problem reduction, AI planning, game playing, theorem proving, Automated Reasoning, Expert Systems
Complexity	Generally less complex than AO* due to linear path nature	More complex due to handling AND-OR structures

Constraint Satisfaction Problem (CSP) in AI

In **Artificial Intelligence (AI)** and **Computer Science**, a **Constraint Satisfaction Problem (CSP)** is defined by:

- A set of **variables**
- A **domain** of possible values for each variable
- A set of **constraints** that specify valid combinations of values

The objective is to assign values to all variables such that **all constraints are satisfied**. In simple terms, CSPs aim to find consistent solutions that meet given rules or limitations.



Constraint Satisfaction Problem (CSP) in AI

Basic Components of CSPs

- **Variables X = {X1, X2, …, Xn}:**

  The unknowns that need values. Examples: Boolean, integer, categorical.

- **Domain D = {D1, D2, …., Dn}:**

  The set of possible values each variable can take.

  - Finite: {1, 2, 3}
  - Continuous: real numbers in [0,1]

- **Constraints C = {C1, C2, …., Cn}:**

  Rules that restrict variable assignments where each can involve any number of variables:

  - **Unary:** involves one variable
  - **Binary:** involves two variables
  - **Higher-order:** involves more than two
  - Examples: Sudoku rules (row, column, box uniqueness)

Constraint Satisfaction Problem (CSP) in AI

Here are some simple examples of constraint satisfaction problems:

Example 1: The n-Queen problem: The local condition is that no two queens attack each other, i.e. are on the same row, or column, or diagonal.

Example 2: A map coloring problem: We are given a map, i.e. a planar graph, and we are told to color it using three colors, green, red, and blue, so that no two neighboring countries have the same color.

Example 3: A cryptography problem: In the following pattern

```
  S E N D
  M O R E
---------
M O N E Y
```

we have to replace each letter by a distinct digit so that the resulting sum is correct.

Constraint Satisfaction Problem (CSP) in AI

In **Constraint Satisfaction Problems (CSPs)**, constraints define relationships between variables and restrict possible value assignments. The main types are:

- **Unary Constraints** – Restrict values of a single variable.

  Example: X1 ≠ 7.

- **Binary Constraints** – Define relationships between two variables.

  Example: X1 < X2.

- **Global Constraints** – Apply to multiple variables, capturing higher-level patterns.
  - **AllDifferent Constraint (AllDiff):** All variables must take unique values.

    Example: AllDiff (X1, X2, X3).
  - **Sum Constraint:** Sum of variables must meet a condition.

    Example: X1 + X2 + X3 = 15.

Constraint Satisfaction Problem (CSP) in AI

In **CSPs**, domains define the possible values for variables. Common categories are:

- **Finite Domains** – Discrete sets of values:
  - **Binary:** {0,1}
  - **Integer:** {1,2,3,4,...}
  - **Enumeration:** {red, green, blue}
- **Continuous Domains** – Real-valued ranges:
  - **Real-valued:** $X \in [0,1]$
  - **Interval:** $X \in [-\pi, \pi]$

CSP – Backtracking Algorithm

CSP Solving Algorithms

- **Backtracking Algorithm** Backtracking is a **systematic trial-and-error method** used to solve CSPs (Constraint Satisfaction Problems).

  It works as follows:

  - Start with an empty assignment.
  - Pick a variable and assign it a value.
  - If the assignment is valid (satisfies constraints), move to the next variable.
  - If no valid assignment is possible, **backtrack** (go back to previous variable and try a new value).
  - Repeat until a solution is found or all possibilities are exhausted.

CSP – Backtracking Algorithm

Example 1: Solving Map Coloring Problem by backtracking

Problem: Color 3 regions (A, B, C) using {Red, Green, Blue}, such that no two neighboring regions have the same color.

- Constraints: A ≠ B, B ≠ C, A ≠ C

Step-by-step Backtracking

1. Assign A = Red. ✓
2. Move to B. Possible values = {Red, Green, Blue}. But **A ≠ B**, so B ≠ Red → Try B = Green. ✓
3. Move to C. Possible values = {Red, Green, Blue}.
    - Constraint: C ≠ A (Red) and C ≠ B (Green) → Only C = Blue is valid. ✓

Solution = {A = Red, B = Green, C = Blue}

If we had chosen B = Red earlier, it would fail at step 3 → backtracking would occur.

CSP – Backtracking Algorithm

Example 1: N-Queens Problem (4 Queens)

Problem: Place 4 queens on a 4×4 chessboard so no two queens attack each other.

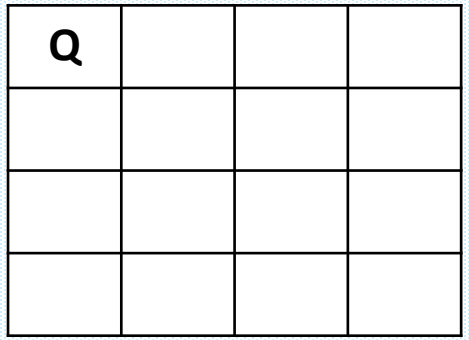
- Constraint: No two queens can be in the same row, column, or diagonal.

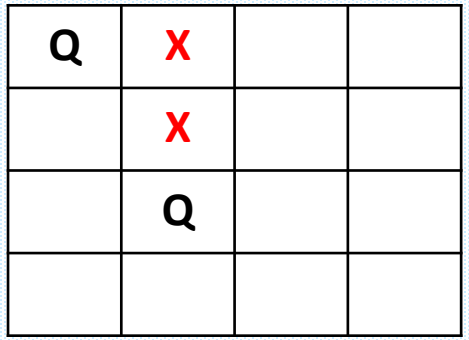
Step-by-step Backtracking: (Column, Row) Representation used

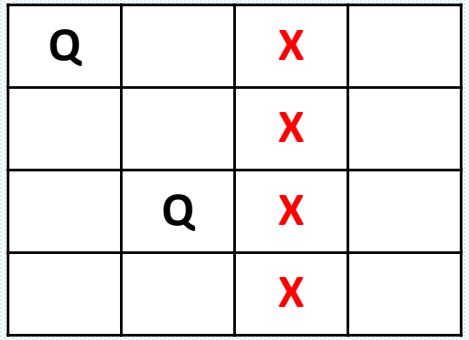
We place one queen per row.

1. **Col 1:** Place queen at (1,1). ✓
2. **Col 2:** Can't place at (2,1) (same column), (2,2) (diagonal clash). Try (2,3). ✓
3. **Col 3:**
   - (3,1) is attacked (same row as (1,1))
   - (3,2) is attacked (diagonal)
   - (3,3) is attacked (same row as (2,3))
   - (3,4) is attacked (diagonal)

No safe place → Backtrack!

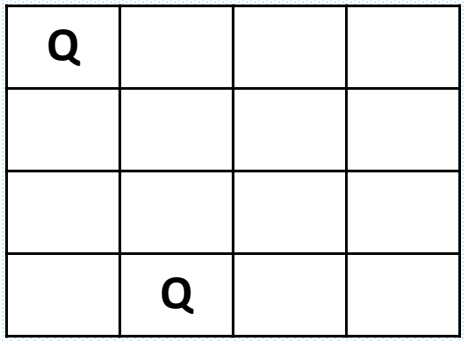




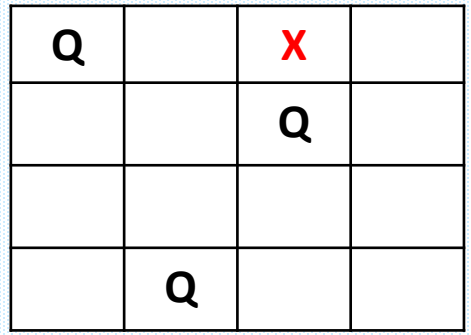


CSP – Backtracking Algorithm

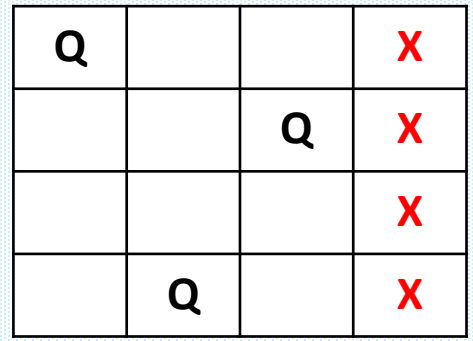
4. Go back to **Col 2**. Move queen from (2,3) → try (2,4). ✓



5. **Col 3:**
   - (1,1) attacked by (1,1)
   - (3,2) is safe → place queen at (3,2). ✓



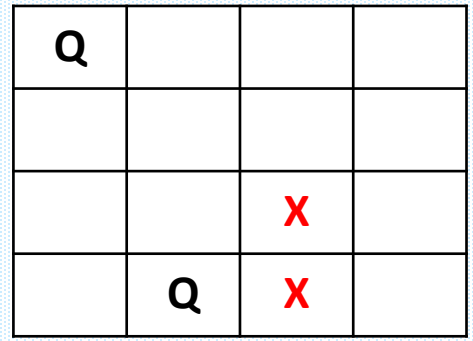
6. **Col 4:**
   - (4,1) is attacked (same row as (1,1))
   - (4,2) is attacked (diagonal)
   - (4,3) is attacked (same row as (3,2))
   - (4,4) is attacked (same row as (2,4))



No safe place → Backtrack!

7. Go back to col 3.

8. Check (3,3) attacked, (3,4) attacked.



CSP – Backtracking Algorithm

9. Go back to **Col 1**. Move queen from (1,1) → try (1,2). ✓

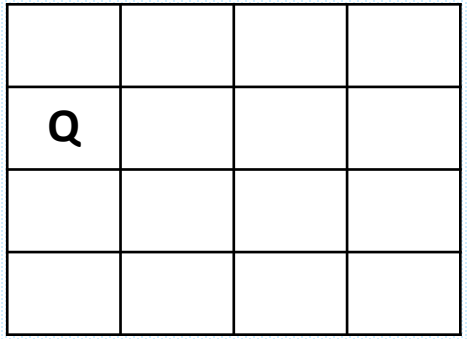
8. Col 2:

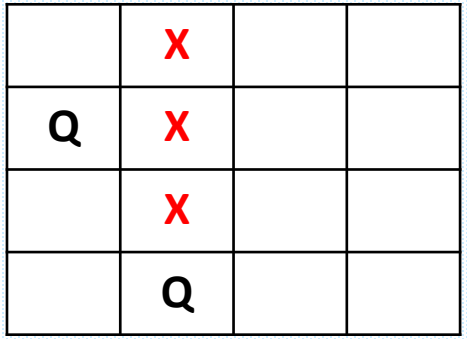
- (2, 1) is attacked.
- (2,2) is attacked.
- (2,3) is attacked.
- (2,4) is safe → place queen at (2,4). ✓

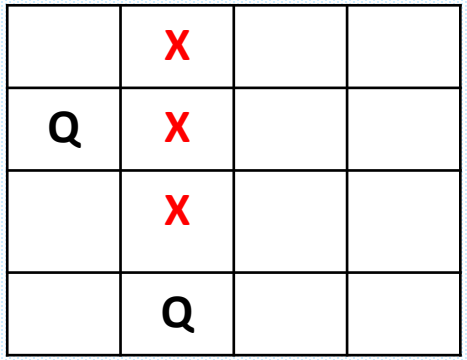
9. Col 3: (3,1) is safe → place queen at (3,1). ✓

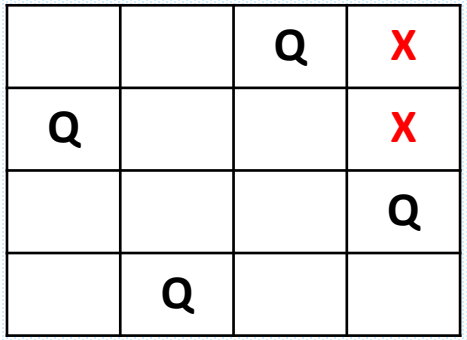
10. Col 4:

- (4,1) is attacked.
- (4,2) is attacked.
- (4,3) is safe → place queen at (4,3). ✓





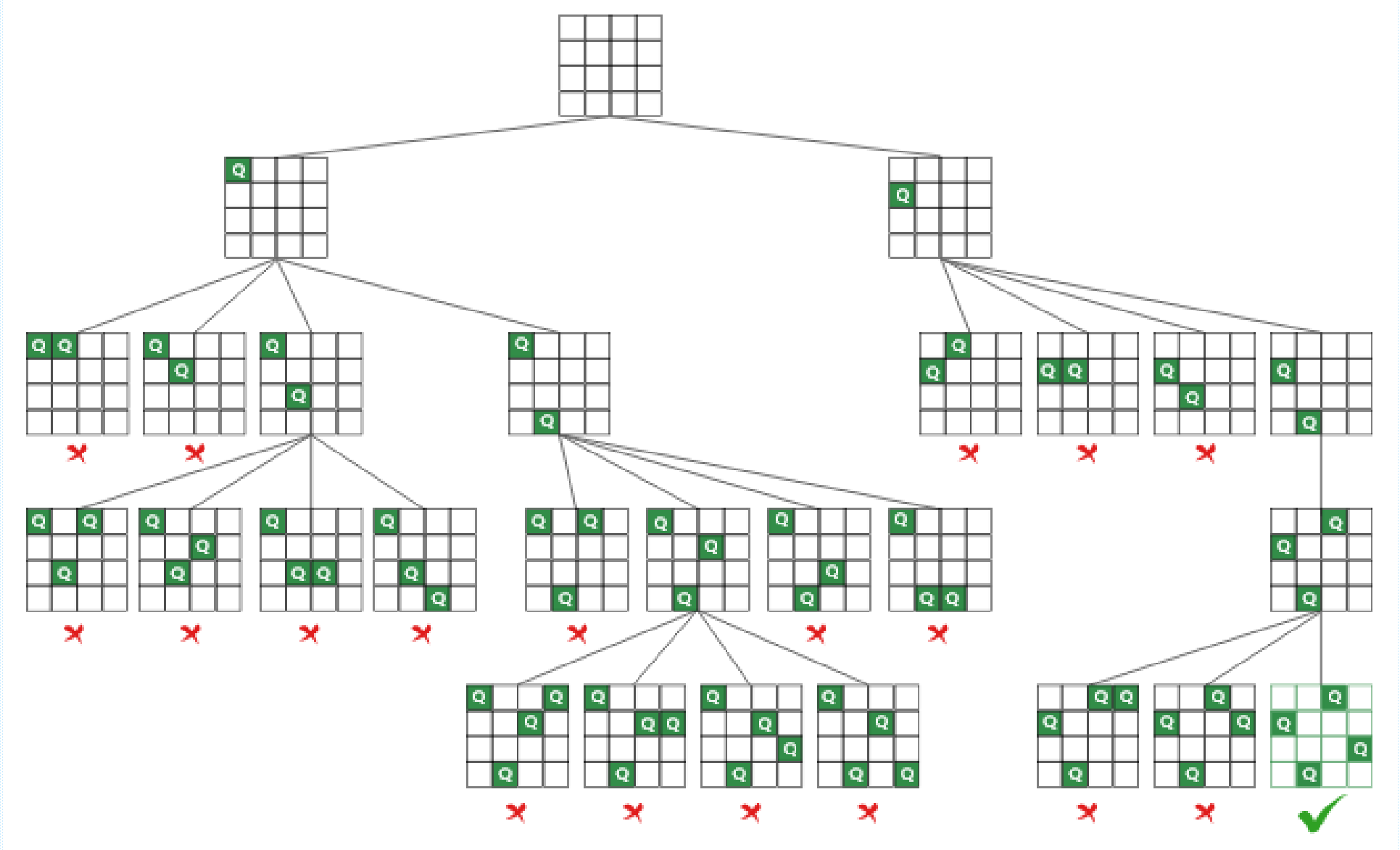




Solution: ***(1, 2) (2,4) (3,1) (4,3)***

CSP – Backtracking Algorithm

4 Queen's Problem using Backtracking



CSP – Variable Ordering

❖ **Variable Ordering**: Choosing the sequence in which variables are assigned values during search. A good ordering reduces the search space and detects conflicts earlier, making solving much faster.

❖ **Types of Variable Ordering**

- **Static Ordering** – fixed before search begins (e.g., alphabetical order of variables).
- **Dynamic Ordering** – chosen **during search**, based on heuristics like:
  - **MRV (Minimum Remaining Values / Most Constrained Variable):** Pick the variable with the fewest legal values left.
  - **Degree Heuristic:** Pick the variable involved in the largest number of constraints (most restrictive).

❖ **Why is Variable Ordering Important?**

- Reduces backtracking.
- Detects infeasibility earlier.
- Cuts down search space.
- Makes otherwise intractable problems solvable.

CSP – Variable Ordering

Example: Cryptarithmetic (SEND + MORE = MONEY)

Variables = {S, E, N, D, M, O, R, Y}, Domains = {0–9} (unique digits, no repetition).

Constraints:

- M = 1 (since result has 5 digits).
- Leading digits (S, M) ≠ 0.
- Column-wise addition must be consistent.

Case A: Naive (Static) Variable Ordering

- Choose variables in alphabetical order: **D → E → M → N → O → R → S → Y**.
- Suppose D = 0, E = 1, M = 2 …
- You may waste time exploring **invalid assignments** (like M ≠ 1).
- Huge branching before realizing contradiction.

Case B: Smart (Heuristic) Variable Ordering

- Use **MRV & problem knowledge**:
- Start with **M** (most constrained → must be 1).
- Then choose **S** (first digit of SEND → can't be 0).
- Next choose **O**, because it appears twice (in MORE & MONEY).
- Then assign E, N, R, D, Y progressively.
- This way, you cut the search tree drastically.

CSP – Value Ordering

❖ Value Ordering

- Once a variable has been chosen (using variable ordering), we must decide which value to try first from its domain.
- Value Ordering decides the sequence of values to assign to a variable.
- A smart value ordering can: ***Reduce backtracking***, ***Help find solutions faster***, ***Avoid exploring "bad" values early.***

❖ Strategies for Value Ordering

- **Naïve Ordering** – pick values in some fixed order (e.g., smallest to largest).
- **Least Constraining Value (LCV) Heuristic** – The Least Constraining Value (LCV) is a heuristic used in Constraint Satisfaction Problems (CSPs) to decide which value to assign to a variable first. Choose the value that rules out the fewest choices for remaining variables.
  - Keeps more flexibility for later assignments.
  - Often used in CSP solvers like Sudoku, Map Coloring, Cryptarithmetic.

CSP – Value Ordering

Example 1: Cryptarithmetic (SEND + MORE = MONEY)

We already applied variable ordering:

- Start with **M = 1** (since MONEY has 5 digits).
- Now suppose we are assigning **O** (next important variable).
- Domain of O = {0–9}, but O ≠ 1 (already taken by M).

Case 1: Naive Value Ordering:

- Try O = 2, then O = 3, then O = 4 … etc.
- It may waste time exploring values that quickly lead to contradiction.

Case 2: Smart Value Ordering (LCV):

- Pick the value for O that: Does not conflict with carry constraints in addition.
- Leaves more possible options for S, E, N, R, D, Y.
- The correct choice is O = 0 (since in 9567 + 1085 = 10652).

  Choosing 0 early avoids wasted search.

CSP – Value Ordering

Example 2: Map Coloring Problem

- Variables: {A, B, C, D} (regions).
- Domain: {Red, Green, Blue}.
- Constraints: Adjacent regions must have different colors.

Suppose we're assigning variable **A** first.

Domain(A) = {Red, Green, Blue}.

Case 1: Naive Ordering:

- Assign Red first → then try Green → then Blue.

Case 2: Smart Value Ordering (LCV):

- If A = Red → many neighbors lose Red from their domains.
- If A = Green → fewer restrictions.
- If A = Blue → even fewer conflicts.

Best value choice: Assign Blue first, because it leaves more freedom for neighbors.

CSP – Forward Checking

Forward Checking: The backtracking technique has been improved using forward checking. It tracks the remaining accurate values of the unassigned variables after each assignment and reduces the domains of variables whose values don't match the assigned ones. As a result, the search space is smaller, and constraint propagation is more effectively accomplished.

In backtracking search for CSPs, **forward checking (FC)** prunes the search *as soon as you make an assignment*.

After assigning a value to a variable X, FC looks at every **unassigned neighbor** Y (variables that share a constraint with X) and **deletes values from Y's domain** that are inconsistent with X's new value.

If any neighbor's domain becomes **empty**, FC **immediately backtracks** (early failure).

FC is lighter-weight than full arc consistency (e.g., AC-3) because it only propagates **one step forward** from the most recent assignment.

CSP – Forward Checking

Algorithm Forward Checking

- Pick an unassigned variable X (often with MRV).
- Try a value $v \in D(X)$.
- For each neighbor Y of X:
  - Remove from D(Y) any value w that violates the constraint with X=v.
- If any D(Y) becomes empty → **undo** the pruning and **backtrack**.
- Recurse; on backtrack, restore pruned values.

CSP – Forward Checking

Example 1:

Problem: We want to color a small map (3 regions: **A, B, C**) such that no two neighboring regions have the same color.

- Variables: A, B, C
- Domain: {Red, Green, Blue}
- Constraints: A ≠ B, B ≠ C, A ≠ C

Forward Checking Steps

1. **Initial Domains**

   A: {R, G, B}, B: {R, G, B}, C: {R, G, B}

2. **Assign A = Red**
   - Now, look ahead (forward check):
     - B cannot be Red (because A ≠ B) → B: {G, B}
     - C cannot be Red (because A ≠ C) → C: {G, B}
   - Updated domains:

     A: {R}, B: {G, B}, C: {G, B}

3. **Assign B = Green**
   - Forward check again:
     - C cannot be Green (because B ≠ C) → C: {B}
   - Updated domains:

     A: {R}, B: {G}, C: {B}

4. **Assign C = Blue**

   All constraints satisfied.

Solution:

A = Red, B = Green, C = Blue

CSP – Forward Checking

Example:

Problem:

- Variables: {S, E, N, D, M, O, R, Y}
- Domain: {0,1,2,3,4,5,6,7,8,9}
- Constraints:
  - All letters must have unique digits.
  - No leading zero → S ≠ 0, M ≠ 0.
  - Arithmetic constraint: SEND + MORE = MONEY.

Step 1: Assign M = 1

Reason: In MONEY, the result is a 5-digit number, so M must be 1.

FC: Remove **1** from all other domains (S,E,N,D,O,R,Y).

Step 2: Assign O = 0

Because in MONEY, O is the second digit, and in addition "S + M" creates this carry → O = 0.

FC: Remove **0** from all other domains.

Step 3: Assign S = 9

Reason: For SEND + MORE = MONEY, S must be the largest unused digit (common heuristic).

FC: Remove **9** from all other domains.

Step 4: Assign E = 5

From carry constraints in column addition.

FC: Remove **5** from others.

Now domains shrink after each assignment:

$D \in \{7, 8\}$, $N \in \{6, 7\}$, $R \in \{8, 7\}$, $Y \in \{2, 3, 4\}$

CSP – Constraint Propagation

- **Constraint Propagation:** Constraint Propagation is a technique used in Constraint Satisfaction Problems (CSPs) to reduce the possible values of variables by using the constraints between them.

- Unlike **forward checking** (which only looks one step ahead), constraint propagation tries to maintain a stronger level of **consistency** across multiple variables.

- Common techniques:
  - **Node consistency** → unary constraints
  - **Arc consistency (AC-3)** → binary constraints
  - **Path consistency** → constraints involving 3 variables

- The main idea: **push the effect of constraints throughout the network** so conflicts are detected as early as possible.

CSP – Constraint Propagation

Example 1: Map Coloring (Arc Consistency – AC-3)

Problem: Suppose we have three regions: **A, B, C.** Available colors: **{Red, Green, Blue},** Constraint: **A ≠ B**

- Initially: A={Red, Green, Blue}, B={Red, Green, Blue}, C={Red, Green, Blue}
- Now suppose we assign: **A = Red,** Because **A and B must have different colors**, B cannot be Red.

  Therefore: **B = {Green, Blue}**

  This is **constraint propagation**.
- **Propagation Can Continue**
  - Suppose there is another constraint: **B ≠ C**
  - Now suppose we choose: **B = Green**
  - Because B and C must be different: **C cannot be Green.**
  - Therefore: **C = {Red, Blue}**

Result: Constraint propagation prevents invalid assignments (like A=Red, B=Red) from ever being considered.

Constraint Propagation

MAP COLORING PROBLEM

MAP COLORING – CONSTRAINT PROPAGATION EXAMPLE

PROBLEM

Color the map of Australia using three colors {Red (R), Green (G), Blue (B)} such that no two adjacent regions have the same color.

Regions:

WA – Western Australia
NT – Northern Territory
SA – South Australia
Q – Queensland
NSW – New South Wales
V – Victoria
T – Tasmania

STEP 1: Initial Domains

Initially each region can have any of the three colors.

Region	Domain (Possible Colors)
WA	{R, G, B}
NT	{R, G, B}
SA	{R, G, B}
Q	{R, G, B}
NSW	{R, G, B}
V	{R, G, B}
T	{R, G, B}

Adjacency (Constraints)

WA – NT, SA
NT – WA, SA, Q
SA – WA, NT, Q, NSW, V
Q – NT, SA, NSW
NSW – SA, Q, V
V – SA, NSW, T
T – V

MAP COLORING PROBLEM

STEP 2: Assign a color and propagate constraints

Let's assign **WA = Red (R)**

Since WA is adjacent to NT and SA, remove Red (R) from their domains.

Region	Domain
WA	{R}
NT	{G, B}
SA	{G, B}
Q	{R, G, B}
NSW	{R, G, B}
V	{R, G, B}
T	{R, G, B}

STEP 3: Continue propagation

Assign **NT = Green (G)**

NT is adjacent to WA, SA and Q. Remove Green (G) from SA and Q.

Region	Domain
WA	{R}
NT	{G}
SA	{B}
Q	{R, B}
NSW	{R, G, B}
V	{R, G, B}
T	{R, G, B}

STEP 4: More propagation

From Step 3, SA = {B} (only one choice left)

So, assign **SA = Blue (B)**

SA is adjacent to Q, NSW and V. Remove Blue (B) from their domains.

Region	Domain
WA	{R}
NT	{G}
SA	{B}
Q	{R}
NSW	{R, G}
V	{R, G}
T	{R, G, B}

STEP 5: Propagate further

From Step 4, Q = {R} → assign **Q = Red (R)**

Q is adjacent to NT, SA and NSW. Remove Red (R) from NSW.

Region	Domain
WA	{R}
NT	{G}
SA	{B}
Q	{R}
NSW	{G}
V	{R, G}
T	{R, G, B}

MAP COLORING PROBLEM

STEP 6: Continue

From Step 5, NSW = {G} → assign **NSW = Green (G)**

NSW is adjacent to SA, Q and V.
Remove Green (G) from V.

Region	Domain
WA	{R}
NT	{G}
SA	{B}
Q	{R}
NSW	{G}
V	{R}
T	{R, G, B}

STEP 7: Final propagation

From Step 6, V = {R} → assign **V = Red (R)**

V is adjacent to SA, NSW and T.
Remove Red (R) from T.

Region	Domain
WA	{R}
NT	{G}
SA	{B}
Q	{R}
NSW	{G}
V	{R}
T	{G, B}

FINAL SOLUTION

One possible coloring after constraint propagation:

WA = Red | **NT = Green** | **SA = Blue** | **Q = Red** | **NSW = Green** | **V = Red** | **T = Green or Blue**

All regions have at least one consistent color and the problem is solved without guessing (only using constraint propagation).

Constraint propagation reduces domains step by step and often solves the CSP without backtracking.

CSP – Constraint Propagation

Example 2: Cryptarithmetic (SEND + MORE = MONEY)

- **Variables:** S,E,N,D,M,O,R,Y
- **Domains:** {0–9}, all different, with S, M≠0

❖ **Constraint propagation:**

- From column 5 (the leftmost): S+M=10+O. Since M is the carry digit to MONEY, M must be **1**.
  - Prune 1 from all other domains.
- From column 4: S+M=O+10. With M=1, only S=9 and O=0 work.
  - Prune {9} from all others and {0} from others.
- These forced values propagate through other columns, reducing possibilities before deep search.

Cryptarithmetic (SEND + MORE = MONEY)

Problem: Each Letter represents a different digit (0-9). Find digits for S, E, N, D, M, O, R, Y such that SEND +MORE = MONEY

Rules:

- All letters are different
- S and M cannot be 0 (leading digits).

Step 1: Write the Addition Column-wise

```
    S   E   N   D
+   M   O   R   E
-------------------
  M   O   N   E   Y
  Th  H   T   O   (1's)
              (carry)
```

Let the carries from right to left be:

C_1 1, C_2, C_3, C_4

C_0 = 0 (initial carry)

Cryptarithmetic (SEND + MORE = MONEY)

Step 2: Start from Units Column

Column (1's): $D + E = Y + 10\ C_1$

Possible digit pairs (D, E) and results:

D	E	Sum	Y C_1
0	1	1	0
0	2	2	0
0	3	3	0
0	4	4	0
0	5	5	0
...	...	...	...
9	0	9	0
9	10	0	1
2	11	1	1
...	...	...	...
9	18	8	1

$Y \neq 0$ because result has 5 digits.

So, C_1 can be 0 or 1.

Step 3: Tens Column

Column (Tens): $N + R + C_1 = E + 10\ C_2$

Case 1: $C_1 = 0$

$N + R = E + 10\ C_2$

Since N, R, E are digits 0–9

N + R can be 0–18

If $N + R \leq 9 \rightarrow C_2 = 0$

If $N + R \geq 10 \rightarrow C_2 = 1$

Case 2: $C_1 = 1$

$N + R + 1 = E + 10\ C_2$

N + R can be 0–18

If $N + R + 1 \leq 9 \rightarrow C_2 = 0$

If $N + R + 1 \geq 10 \rightarrow C_2 = 1$

From this column also, C_2 can be 0 or 1.

Cryptarithmetic (SEND + MORE = MONEY)

Step 4: Hundreds Column

Column (Hundreds): $E + O + C_2 = O + 10\ C_3$

O cancels out from both sides

$\Rightarrow\ E + C_2 = 10\ C_3$

C_2	E	$E + C_2$	C_3
0	0	0	0
0	1	1	0
...	...	...	...
0	9	9	0
1	0	1	0
1	1	2	0
...	...	...	...
1	8	9	0
1	9	10	1

Only when $C_2 = 1$ and $E = 9$, we get $C_3 = 1$

So we must have: $C_2 = 1,\ E = 9,\ C_3 = 1$

Step 5: Thousands Column

Column (Thousands): $S + M + C_3 = N + 10\ C_4$

We know $C_3 = 1$

$S + M + 1 = N + 10\ C_4$

Since S, M, N are digits (0–9)

$S + M + 1$ can be 1–19

If $S + M + 1 \leq 9 \rightarrow C_4 = 0$

If $S + M + 1 \geq 10 \rightarrow C_4 = 1$

C_4 can be 0 or 1.

But in the next column the result has only 5 digits (MONEY), so C_4 must be 1.

Cryptarithmetic (SEND + MORE = MONEY)

Step 6: Ten-Thousands Column

Column (Ten-Thousands): $C_4 = M$

Since $C_4 = 1$

$M = 1$

Propagated information so far:

- $M = 1$
- $C_4 = 1$
- $C_3 = 1$
- $C_2 = 1$
- $E = 9$
- $Y \neq 0$

Used digits:

{1, 9}

Remaining digits:

{0, 2, 3, 4, 5, 6, 7, 8}

Unassigned letters:

{S, N, D, O, R, Y}

Step 7: Continue with Units Column

From Step 2 results with $C_1 = 0$ or 1 and $Y \neq 0$.

Try values (D, E, Y, C_1) using remaining digits and propagate forward.

One consistent possibility found by systematic propagation is:

D = 7	O = 0
E = 9	R = 8
N = 6	Y = 2
S = 5	M = 1

Check with all column equations → Valid!

Cryptarithmetic (SEND + MORE = MONEY)

Step 8: Final Solution

```
  5  9  6  7
+ 1  0  8  9
-------------
 1  0  5  6  2
 M  O  N  E  Y
```

Verification:

- 7 + 9 = 16 → 6, carry 1
- 6 + 8 + 1 = 15 → 5, carry 1
- 9 + 0 + 1 = 10 → 0, carry 1
- 5 + 1 + 1 = 7 → 7, carry 0
- Carry to next place = 1 → M = 1 ✓

By applying **Constraint Propagation** column by column, we significantly reduce the search space.

From the carry values, we get:

$M = 1, E = 9, C_2 = 1, C_3 = 1, C_4 = 1$

Then, by checking the possibilities step by step, we obtain the **unique solution**:

SEND + MORE = MONEY

5967 + 1089 = 10562

Cryptarithmetic Problem

Problem 1:

	T	O
+	G	O
O	U	T

Letter	Digit
T	2
O	1
G	8
U	0

Step 1:

Leftmost digit of answer is always 1. As it is produced by carry of previous column. So, O=1. O =1 will be come after addition of previous column letters T+G. O=1 value will be placed at all places of O.

		1
+		1
1		

Step 2:

At Unit Column, we will sum the value of both O. Now, it will give value 2 for letter T. Also, place T=2 in other places of T in the arithmetic.

	2	1
+	9	1
1	1	2

Step 3:

As we can see that, we have T=2 in the 10^{th} place column, the value of G will be assigned so that it will generate a carry which is already assumed in step 1 for 100^{th} place column. We can take only two values (9, 8) for G to produce a carry.

Case 1: If G=9, then T + G = 9 + 2 = 11. Here, Carry is 1 and the value of U after sum of T & G will be 1 which is not possible as 1 is already assigned to O.

Case 2: We will assign G=8, then we'll get U = 0 which is acceptable. Hence, it is solved.

	2	1
+	8	1
1	0	2

Cryptarithmetic Problem

Problem 1:

c4	c3	c2	c1	
	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

Letter	Digit
S	9
E	5
N	6
D	
M	1
O	0
R	
Y	

Step 1:

Leftmost digit is always 1. As it is produced by carry of previous column. So, M=1. M =1 will be come after addition of previous column S+M. M=1 value will be placed at all places of O.

1				
+	1			
1				

Step 2:

If M=1, S+M >= 10, So S=1. Then, S+M = 10 & O=0.

1				
	9			
+	1	0		
1	0			

Step 3:

N=E+c2 and force c2=1

With O=0 and c3=0:

E+0+c2=N → N=E+c2.

Because N and E are different digits, c2 must be 1, so N=E+1.

We will assign E=5, then N = 5 + 1 = 6.

1	0	1		
	9	5	6	
+	1	0		5
1	0	6	5	

Cryptarithmetic Problem

Problem 1:

c4	c3	c2	c1	
	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

Letter	Digit
S	9
E	5
N	6
D	7
M	1
O	0
R	8
Y	2

Step 4:

R+c1=9 → R=8, c1=1

Tens equation: N+R+c1=E+10*c2.

Substitute N=E+1 and c2=1:

(E+1)+R+c1=E+10⇒R+c1=9.

Digit 9 is already used by S, so **R=8** and **c1=1**.

1	0	1	1	
	9	5	6	
+	1	0	8	5
1	0	6	5	

Step 5:

D+E=Y, Y>=10

Since c1=1: D+E=Y. So D+E ≥ 10.

Need D so D+5 ≥ 10; choose D=7 (as 5, 6, 8 & 9 are already assigned to other variables), then Y=7+5=12 where c1=1 & Y=2 (available).

1	0	1	1	
	9	5	6	7
+	1	0	8	5
1	0	6	5	2

Bibliography

1. Artificial Intelligence by Anamitra Deshmukh, Nimbalkar Technical Publication Pune
2. https://commons.wikimedia.org/wiki/File:AI-History-Timeline-300dpi.jpg

CSPC3003

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

MODULE - 2

Course Objectives:

- To learn the concepts of Artificial Intelligence
- To learn the methods of solving problems using Artificial Intelligence
- To introduce the concepts of Expert Systems and machine learning

Game Theory in AI

AI **Game Theory** was proposed because in many **real-world situations**, agents (humans, machines, or programs) interact with each other, and their **decisions affect one another**. To make good decisions, an AI agent must **predict the actions of others** and choose the best strategy accordingly.

Here's why AI game theory was proposed:

1. Decision Making with Multiple Agents

- In real life, there are often multiple players (agents) competing or cooperating.
- Example: In chess, your move depends on what your opponent might do next.

2. Modeling Rational Behavior

- Game theory provides a **mathematical framework** for reasoning about rational decisions.
- It assumes each player is *rational* and tries to maximize their own payoff (benefit).

Game Theory in AI

3. Handling Competitive & Cooperative Environments

- Some environments are **competitive** (e.g., poker, auctions, stock trading).
- Some are **cooperative** (e.g., AI agents working together in traffic management).
- Game theory helps AI handle both.

4. Optimal Strategy Selection

- AI needs to choose the **best move** considering the possible actions of others.
- Example: In the **Minimax algorithm** (used in chess/checkers), one player maximizes the score while the opponent minimizes it.

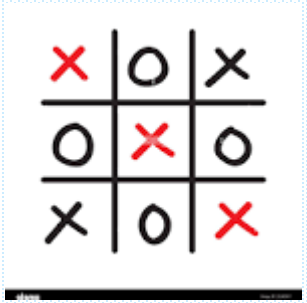
Game Theory Applications in AI

1. **Robotics** → multi-robot coordination.
2. **Economics & Auctions** → AI agents bidding optimally.
3. **Finance, Military Strategy**, and even **Business decision-making**,
4. **Security** → predicting adversary behavior.
5. **Games (Chess, Sudoku, Monopoly, Go, Poker)** → AI competes with humans or other AIs.

Types of Games in AI

There are 4 types of games in AI:

1. **Perfect Information Game**
2. **Imperfect Information Game**
3. **Deterministic Game**
4. **Non-deterministic Game**



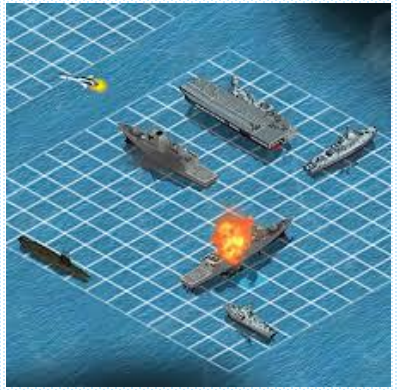
		7			6	5		
	4			7	1			
		9		4		7		6
3		6	8		7	9		2
				5				
2			1		4	6		8
	8	5			3	1		9
		1						
			7		9	4		

1. Perfect Information Game

- A game with perfect information is one in which each agent can look into the complete board.
- The agent has all the information about the game.
- They can see each other's moves also.
- **Examples: Chess** (you see all pieces), **Checkers, Tic-Tac-Toe, Go** (board game), **Sudoku** (all clues are visible)

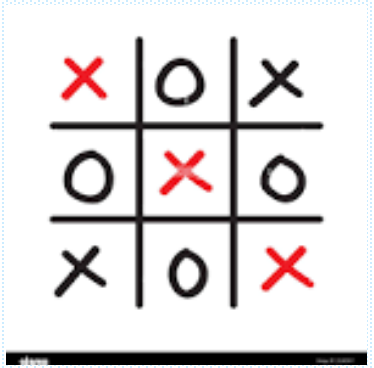
2. Imperfect Information Game

- If in a game, an agent does not have all the information about the game or is not aware of what is going on, such a game is called an **imperfect information game**.
- **Examples: Poker** (opponent's cards are hidden), **Bridge** (card game), **Battleship** (positions of opponent's ships are hidden), **Scrabble** (tiles of the opponent are hidden), **Clue (Detective board game)**.



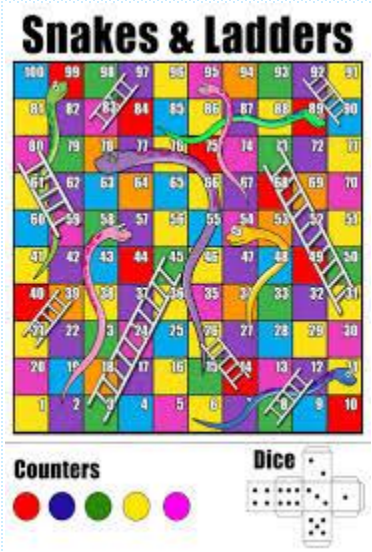
3. Deterministic Game

- Deterministic games follow a strict set of rules and patterns.
- There is **no randomness** involved in the game.
- The next state of the game is completely determined by the current state and actions.
- **Examples:** Tic-Tac-Toe, Chess, Checkers



4. Non-deterministic Game

- Non-deterministic games involve **unpredictable events**.
- They include factors of **chance or luck**, introduced by dice, cards, or other random elements.
- Outcomes are not fully predictable, hence they are also called **stochastic games**.
- **Examples: Monopoly** (dice rolls decide movement), **Ludo** (dice decide moves), **Snakes and Ladders, Poker** (luck of card draw), **Backgammon** (dice rolls matter)



What is Game Theory & How is it Important in AI?

- Game theory is a **logical and scientific study** that models the possible interactions between two or more rational players.
- Here, *rational* means each player thinks that others are just as rational, with the same level of knowledge and understanding.
- In game theory player deals with given **Set of options** is made against situations.
- It means choices of one player affect the choices of the opponent player.
- Game theory and AI are **closely related** and useful to each other.
- To play games (like online mode, laptop games, desktop games, etc.), one has to create algorithms.
- Such game algorithms are developed with the help of **AI**.
- Game theory is widely used to enable **key capabilities** required in multi-agent environments, where multiple agents interact with each other to achieve a goal.

Adversarial Search

- It is a type of search where we **examine the problem** which arises when agents plan ahead of the world, and then agents are planning **against each other**.
- Search in which two or more players with conflicting goals try to explore the same search space for a solution is called **adversarial search**.
- Example: Chess, Tic-Tac-Toe, Checkers.

There are two types of search in adversarial search:

1. **Minimax Algorithm**
2. **Alpha-beta Pruning**

Minimax Algorithm

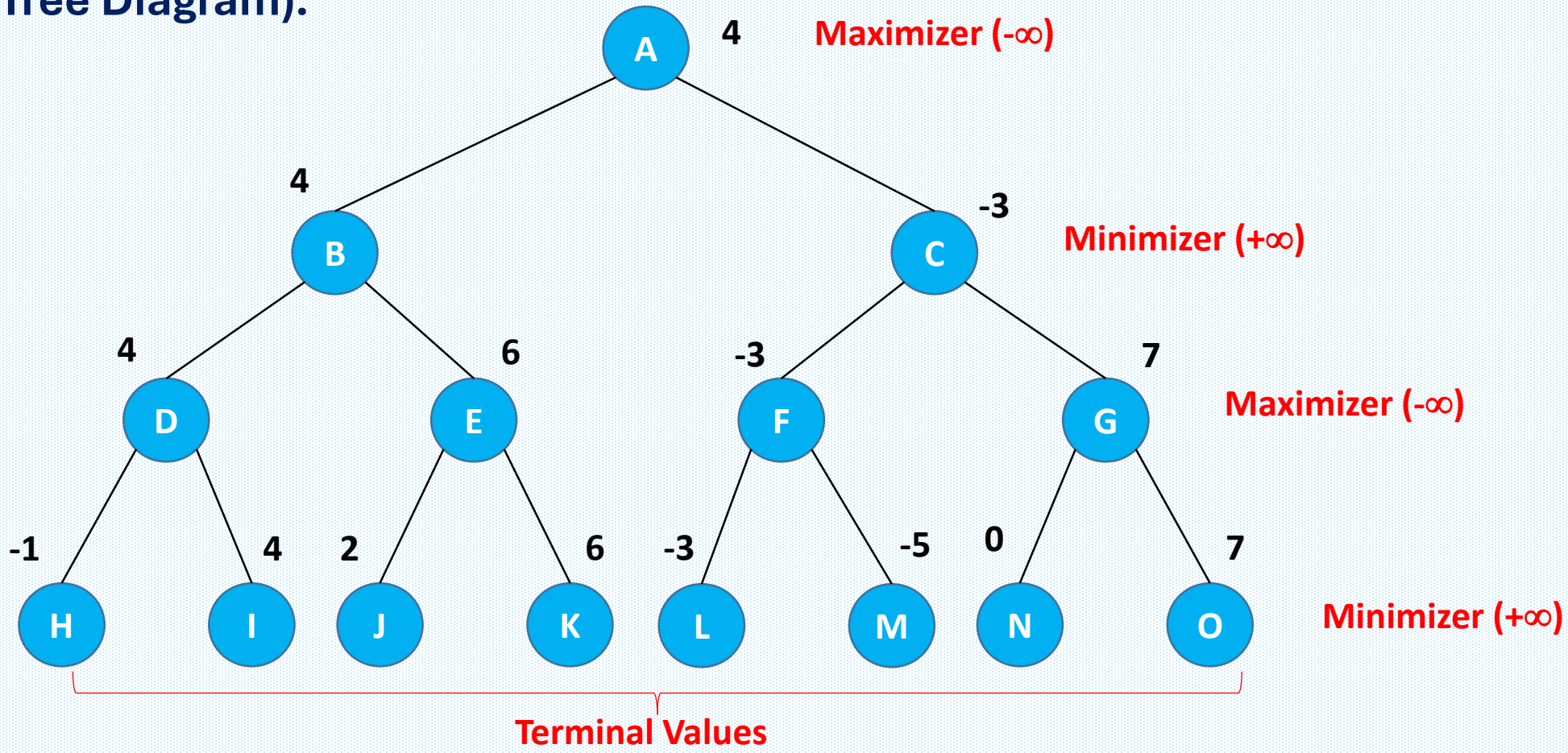
- A **recursive backtracking algorithm** used in decision theory and game theory.
- It provides an **optimal move for the player**, assuming the opponent also plays optimally.
- Minimax is mainly applied in **two-player games**.
- **Minimax algorithm uses recursion** to search through the game tree.
- It is mostly used for game playing in AI such as **chess, checker, tic-tac-toe**, and various two-player games.
- This algorithm computes the **next best move** for the current state.

Minimax Algorithm

- **Key Concepts:**
  - Two players play the game:
    - One is called **MAX**.
    - The other is called **MIN**.
  - Both players fight as opponents:
    - MAX tries to get **maximum benefit**.
    - MIN tries to get **minimum benefit** (opponent's gain).
  - Both are **opponents** of each other:
    - MAX selects the move with the **maximum value**.
    - MIN selects the move with the **minimum value**.

Minimax Algorithm

- The algorithm performs a **DFS (Depth First Search)** of the complete game tree.
- It proceeds **all the way down to the terminal node** of the tree.
- Then it **backtracks** the tree using recursion to assign values.
- **Example (Tree Diagram):**



Minimax Algorithm

- **MAX (Maximizer):**

  Represents the player who wants to **maximize the score** (finds the best/highest value move).

  → Usually called **our agent/AI**.

- **MIN (Minimizer):**

  Represents the **opponent**, who tries to **minimize our/AI score** (chooses the worst/lowest value for us).

- In a two-player game, MAX and MIN alternate at each level of the game tree.

Minimax Algorithm

How to Decide Who is MAX and MIN at Each Node in Game Tree?

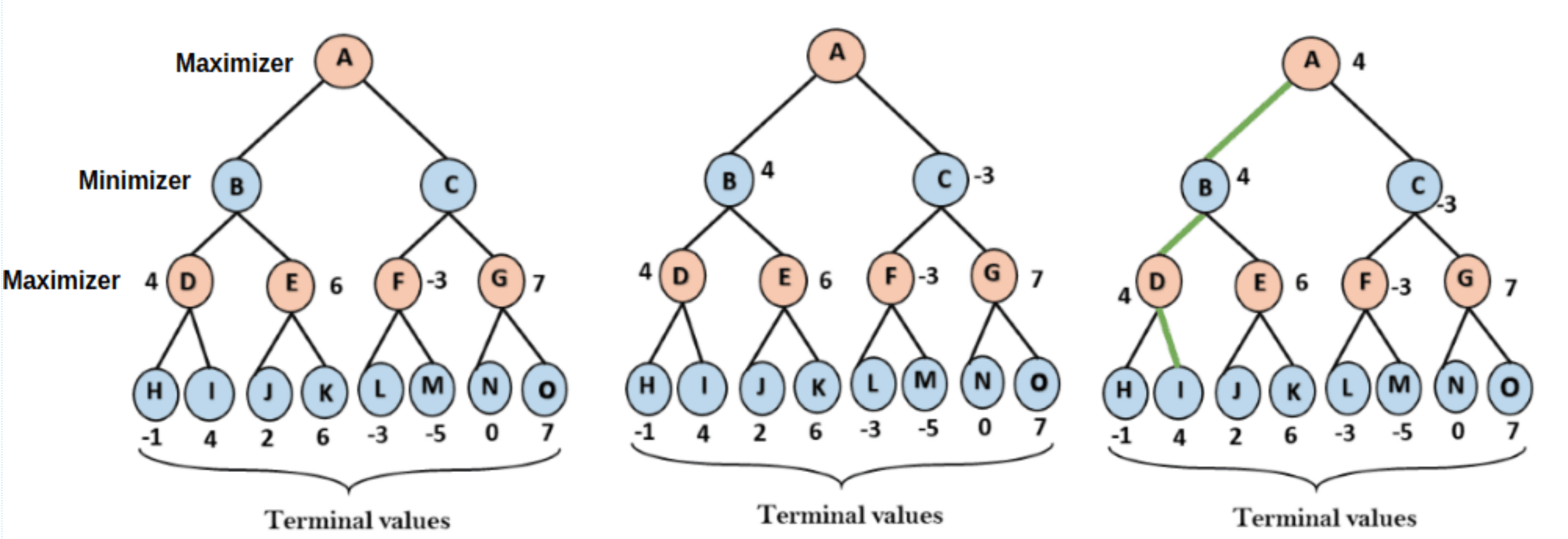
- **Root node (top of the tree):**
  - Always belongs to **the player whose turn it is now**.
  - If it's our AI's turn → Root = MAX.
  - If it's the opponent's turn → Root = MIN.
- **Next level:**
  - After one player makes a move, the **other player's turn comes**.
  - So we alternate between MAX and MIN at each level.
- **Leaf nodes:**
  - Have no MAX or MIN — they only contain the **evaluated scores** (terminal values).

Minimax Algorithm

How to Decide Who is MAX and MIN at Each Node?

- **Root node (top of the tree):**
  - Always belongs to **the player whose turn it is now**.
  - If it's our AI's turn → Root = MAX.
  - If it's the opponent's turn → Root = MIN.
- **Next level:**
  - After one player makes a move, the **other player's turn comes**.
  - So we alternate between MAX and MIN at each level.
- **Leaf nodes:**
  - Have no MAX or MIN — they only contain the **evaluated scores** (terminal values).

Minimax Algorithm



- **Root A (Maximizer)**
  - **B (Minimizer)**
    - D (Max) → H = -1, I = 4 → **D = 4**
    - E (Max) → J = 2, K = 6 → **E = 6**
  - **B = min(4, 6) = 4**
  - **C (Minimizer)**
    - F (Max) → L = -3, M = -5 → **F = -3**
    - G (Max) → N = 0, O = 7 → **G = 7**
  - **C = min(-3, 7) = -3**
- **A = max(4, -3) = 4**
- **Final result = 4 (path: A → B → D → I)**

Tic-Tac-Toe using Minimax Algorithm

What Minimax does (for tic-tac-toe)

- Treats the game as a tree of states: each node = a board; edges = legal moves.
- Two players: **Max** (AI) tries to maximize the score; **Min** (opponent) tries to minimize it.
- Search goes to terminal boards (win/lose/draw) and scores them, then backs those scores up the tree to choose the optimal move.

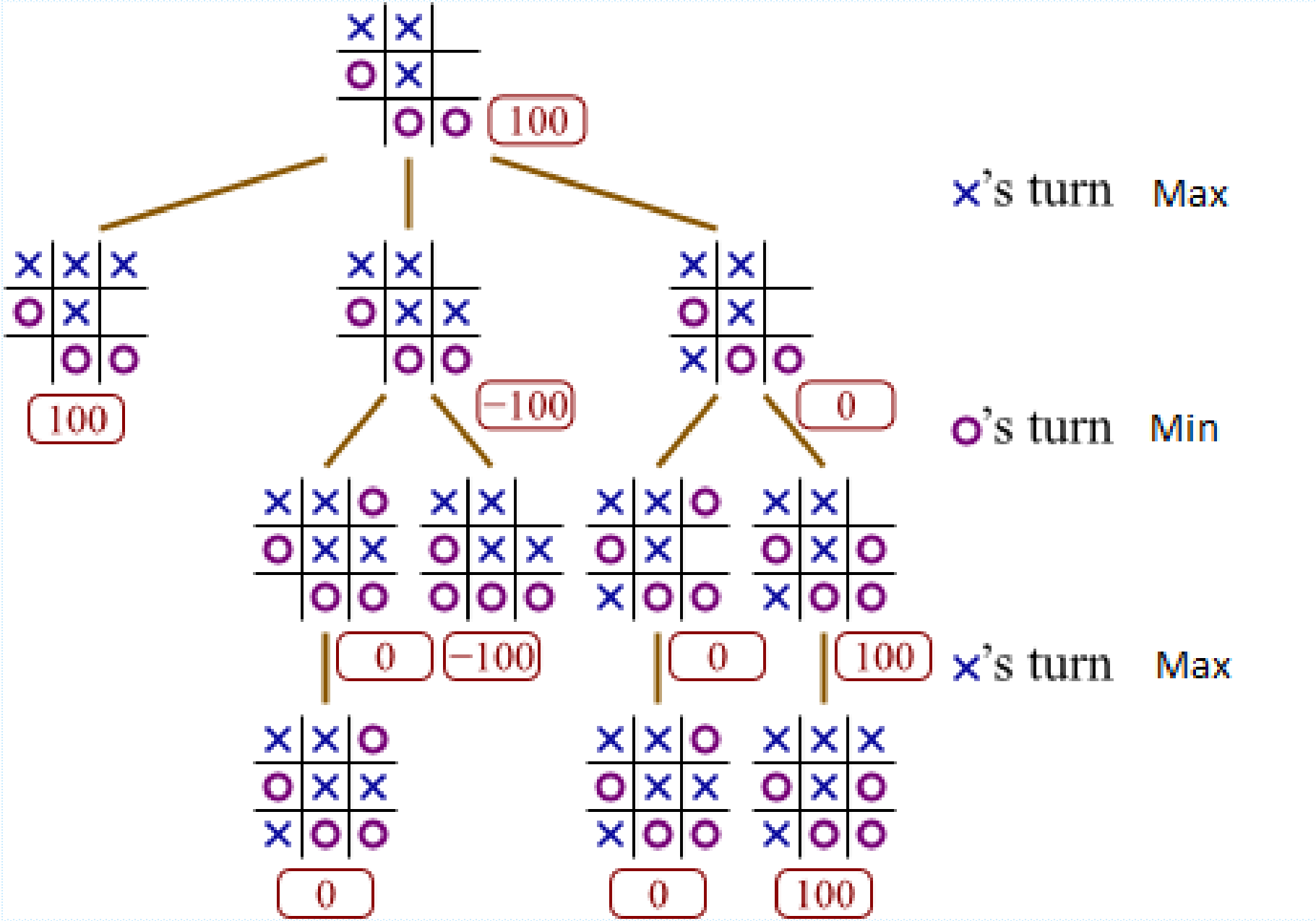
Scoring (simple and effective)

- If AI wins → +100 − depth (prefer faster wins).
- If AI loses → −100 + depth (prefer slower losses).
- Draw → 0.

Turns:

- **Max (X/AI):** chooses the move with the highest score.
- **Min (O/human):** chooses the move with the lowest score.

Tic-Tac-Toe using Minimax Algorithm



Properties of Minimax Algorithm

- **Completeness**: Min-Max is complete i.e. guaranteed to return a move.
- **Time Complexity**: O(b^m), where b is the number of legal moves at each point and m is the maximum depth of the tree.
- **Space Complexity**: O(bm)
- **Optimality**: The MinMax algorithm always finds the optimal move for a player, assuming that the other player is also making optimal moves.

Limitations of Minimax Algorithm

- **High Computational Cost:** Mini-Max explores all moves to terminal nodes, making it expensive for large games like chess.
- **Not Feasible for Complex Games**: Games like Chess or Go have huge branching factors. Exploring the full tree is impractical.
- **No Chance Handling:** It cannot deal with probabilistic events (e.g., dice rolls).
- **High/Exponential Time Complexity:** Time complexity is O(b^m), where b is branching factor and m is depth; impractical without optimizations like Alpha-Beta pruning.
- **Slowness Without Optimization**: Without pruning (like **Alpha-Beta pruning**), it wastes time exploring unnecessary branches.
- **High Space Requirement**: Needs to store many game states, so space complexity is O(bm).

Alpha Beta Pruning in AI

- The Minimax algorithm is an adversarial search algorithm that involves a Depth-First-Search(DFS) on the game tree.
- However, as the depth of the game tree increases, the number of states increases exponentially, leading to high time and space complexity. This is solved by the Alpha Beta Pruning algorithm, an optimization technique of the Minimax algorithm.
- Alpha-beta pruning is an optimization technique applied to the minimax algorithm, which is commonly used in artificial intelligence for game playing.
- Its purpose is to reduce the number of nodes that need to be evaluated in a game tree, thereby improving the efficiency of the search process.
- **Core Idea:** The key concept behind alpha-beta pruning is to eliminate branches of the game tree that are guaranteed not to influence the final decision. This is achieved by maintaining two values:
  1. **Alpha (α):** Represents the best (highest) value that the maximizing player can guarantee so far.
  2. **Beta (β):** Represents the best (lowest) value that the minimizing player can guarantee so far.

Alpha Beta Pruning in AI

How it Works:

The critical points of Alpha Beta Pruning in AI are as follows.

- **The initialization of the parameters**
  - Alpha(α) is initialized with −∞.
  - Beta(β) is initialized with +∞.
- **Updating the parameters**
  - Alpha(α) is updated only by the maximizer at its turn.
  - Beta(β) is updated only by the minimizer at its turn.
- **Passing the parameters**
  - Alpha(α) and Beta(β) are passed on to only child nodes.
  - While backtracking the game tree, the node values are passed to parent nodes.
- **Pruning Condition**
  - The child sub-trees, which are not yet traversed, are pruned if the condition $\alpha \geq \beta$ holds.

Alpha Beta Pruning in AI

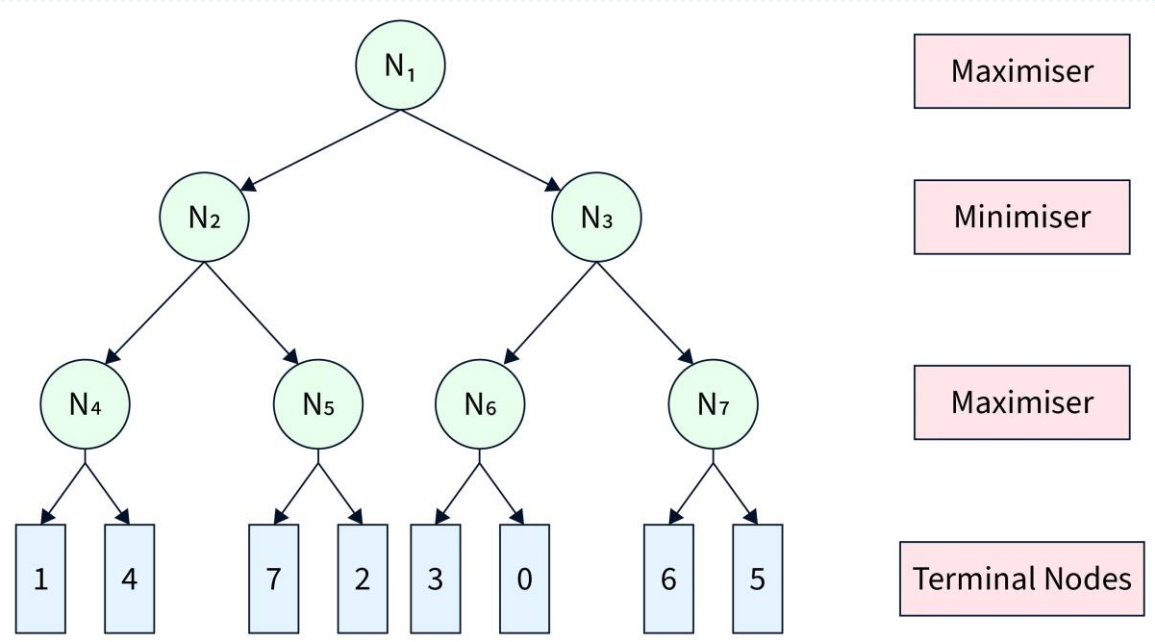
Pruning Condition:

If, at any point during the search, the alpha value becomes greater than or equal to the beta value (α≥β) the current branch can be "pruned" or cut off. This is because:

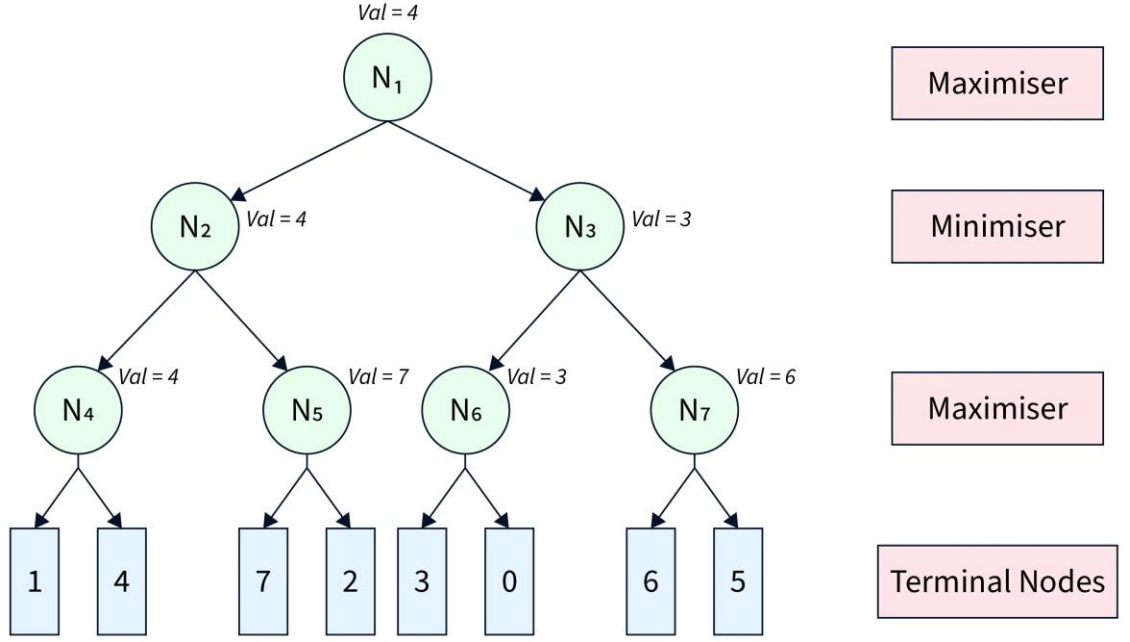
1. If α≥β at a Max node, it means the maximizing player has already found a move that guarantees a score at least as good as what the minimizing player can force in this branch. Therefore, exploring this branch further will not lead to a better outcome for the maximizing player.
2. If α≥β at a Min node, it means the minimizing player has already found a move that guarantees a score at least as bad as what the maximizing player can achieve in this branch. Therefore, exploring this branch further will not lead to a better outcome for the minimizing player.

Alpha Beta Pruning in AI

we use an example game tree shown in Figure-1. The numbers shown in terminal nodes represent their respective utilities.



The Minimax algorithm would have traversed the complete game tree leading to the game tree shown in Figure-2.



Alpha Beta Pruning in AI

Alpha Beta Pruning algorithm restricts the agent from visiting redundant sub-trees leading to faster average search time complexity. To visualize its working, we refer to the same game tree in Figure 1 . We start from node N1 with α=−∞ and β=∞. From N1, we move to its child node N2 with the same *α* and β. Similarly, we move from N2 to its child N4, as shown in Figure-2.

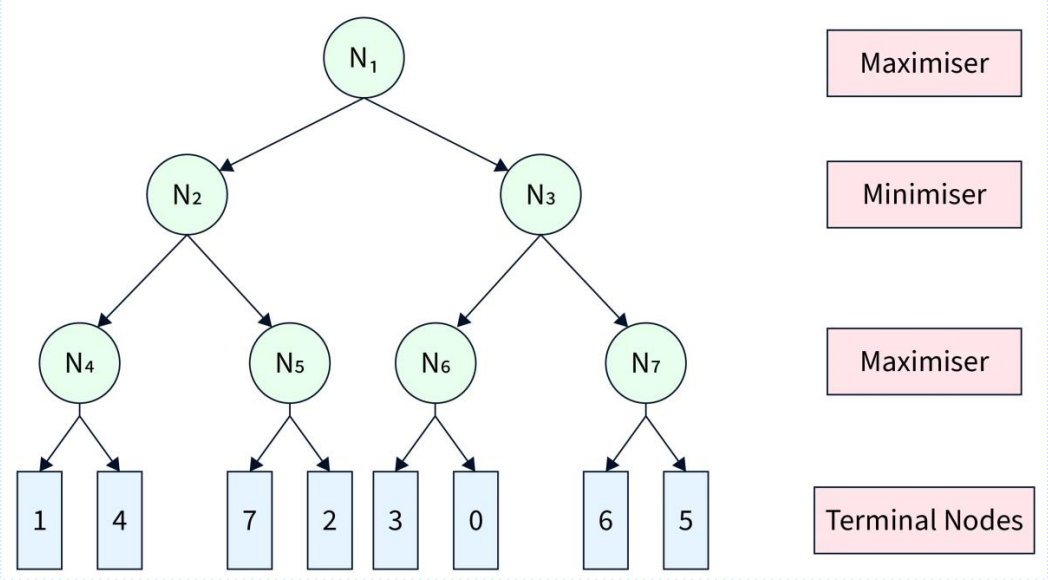


Figure - 1

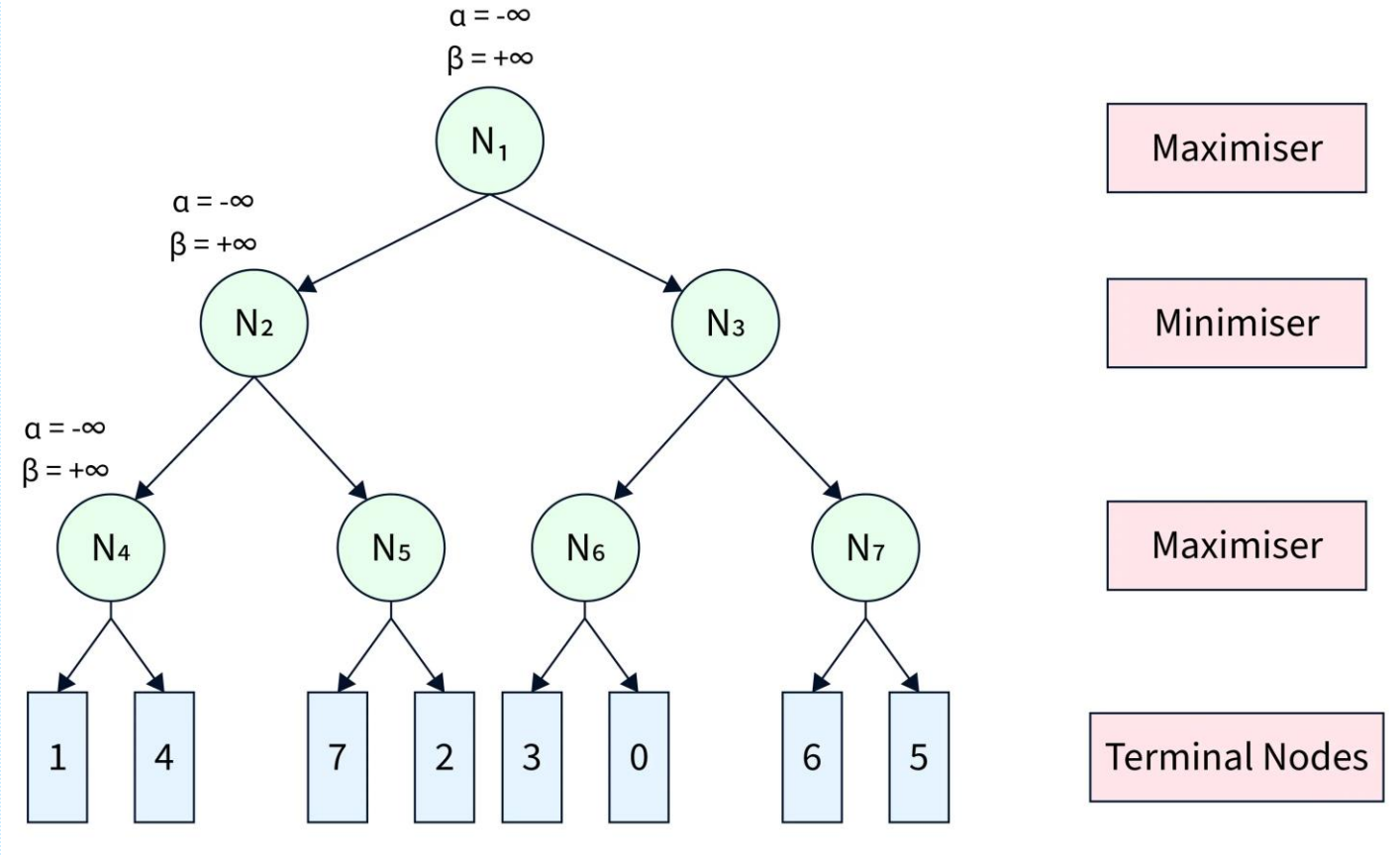
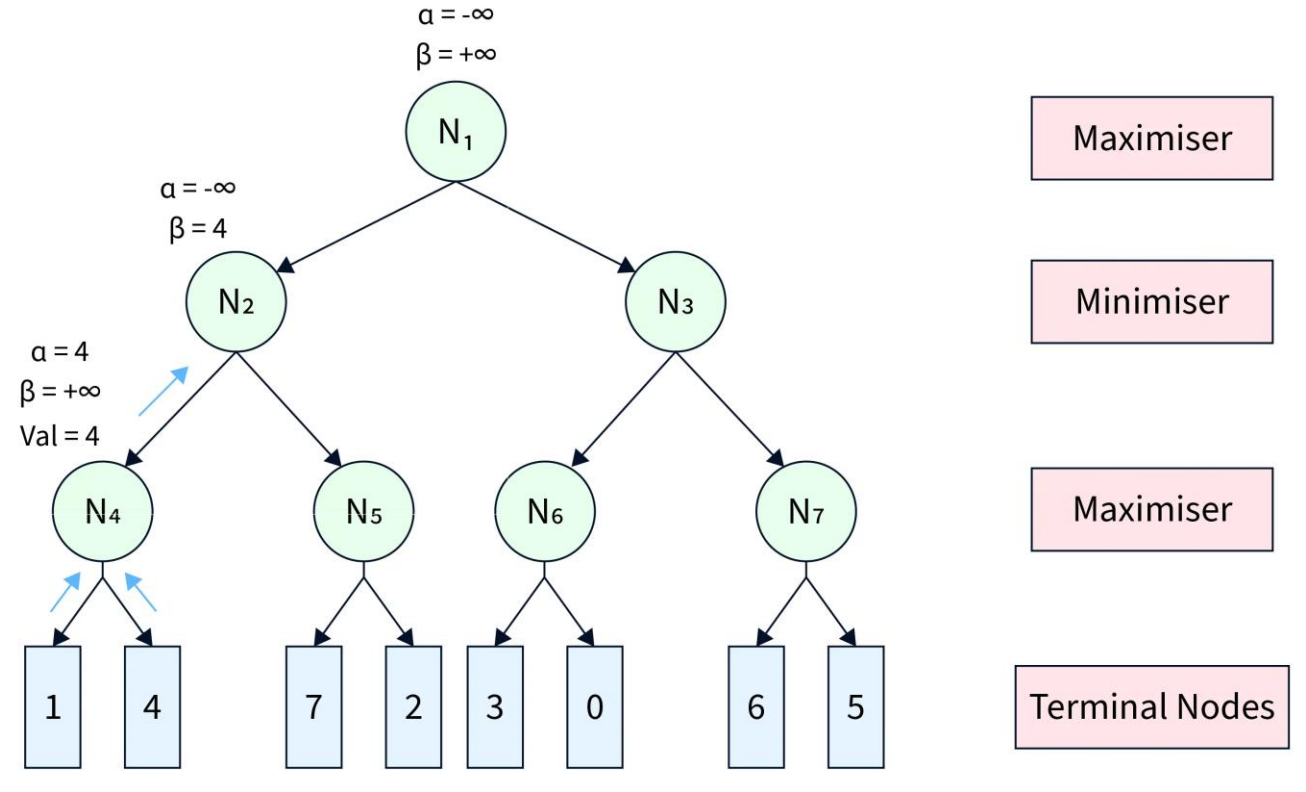


Figure - 2

Alpha Beta Pruning in AI

From N4, we move to its first terminal child and get back its utility as 1. Since N4 is a maximiser, we update α=max(−∞,1)=1. The pruning condition α≥β remains unsatisfied. Therefore, we move to the second terminal child of N4 and get its utility as 4. Further, we update α=max(1,4)=4. After this step, we get the node value of N4 as max(1,4)=4 and return it to its parent N2. Since N2 is a minimiser, it updates β=min(+∞,4)=4. The game tree after this step is shown in Figure-3.

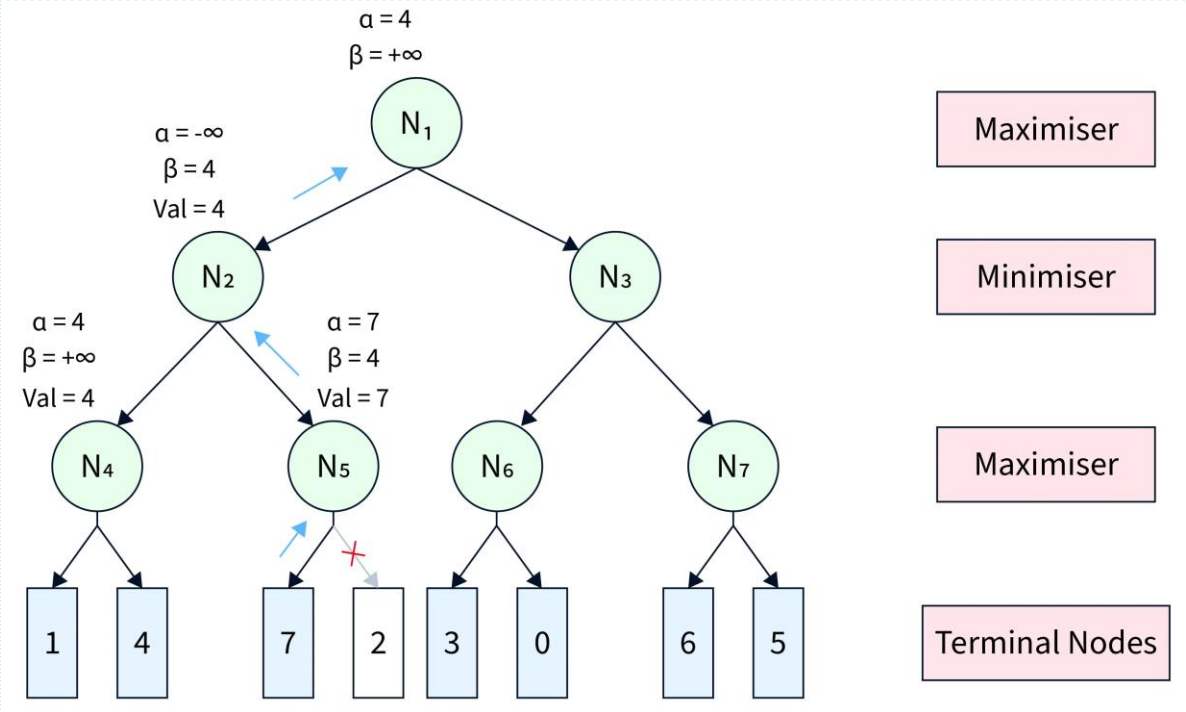
Figure - 3



Alpha Beta Pruning in AI

At N2, the pruning condition α≥β is still unsatisfied. Therefore, we visit its next child node N5 with α=−∞ and β=4. At N5, we get the utility of its first terminal child as 7. Since N5 is a maximiser, we update α=max(−∞,7)=7. Now, the pruning condition gets satisfied (7≥4), leading to the pruning of the other child node. After that, the node utility of N5=7 is further sent back to the parent node N2. N2 updates β=min(4,7)=4. Further, the node value of N2=min(4,7)=4 is returned to its parent N1. At N1, we update α=max(−∞,4)=4. After this step, the game tree looks like Figure-4.

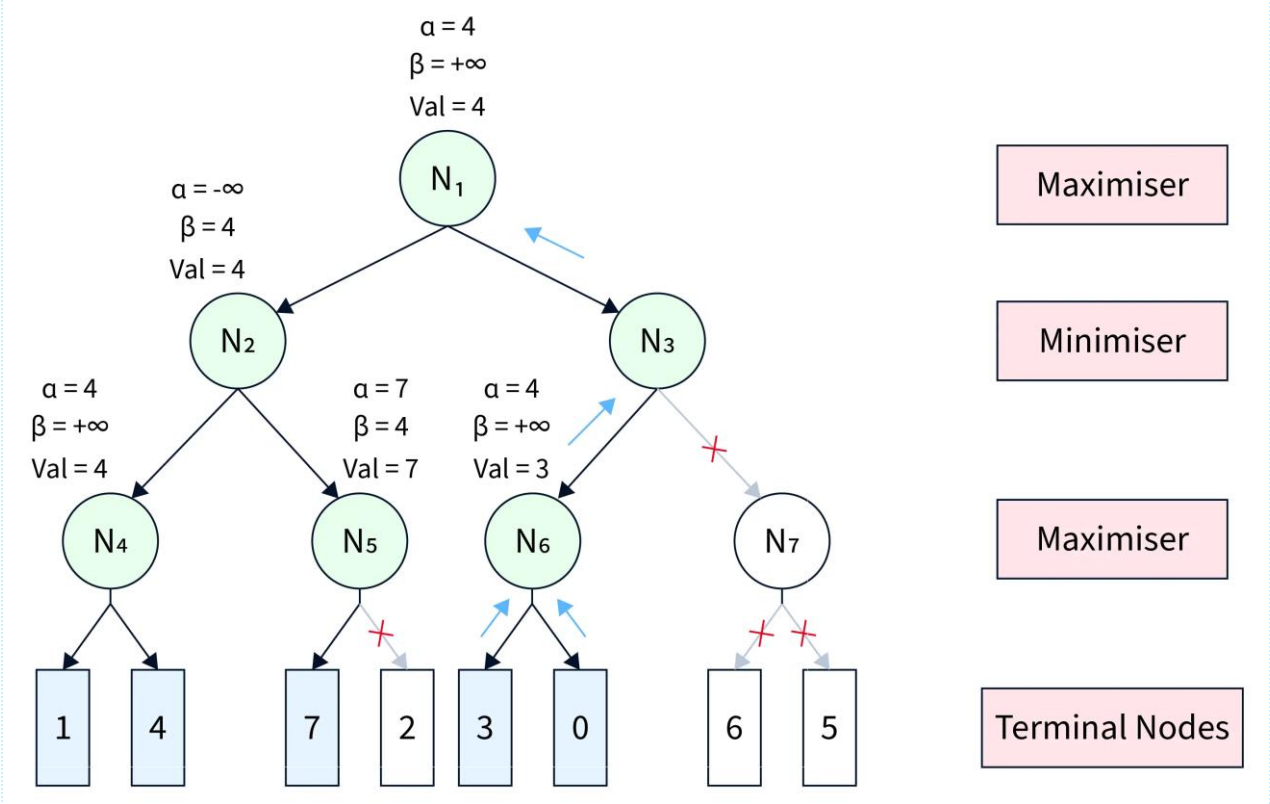
Figure - 4



Alpha Beta Pruning in AI

The pruning condition is still unsatisfactory at N1. Therefore, we continue towards the next child N3, with the same values of α and β. Similarly, we propagate from N3 to its first child N6. At N6, we get back the utility of its first terminal child as 3 and update α=max(4,3)=4. Since the pruning condition is still unsatisfied, we get the utility of the second terminal child as 0 and update α=max(4,0)=4. After this, the node value of N6=max(3,0)=3, is returned to the parent N3.

Figure - 5



Alpha Beta Pruning in AI

Since N3 is a minimiser, we update β=min(+∞,3)=3. Now, the pruning condition gets satisfied (4≥3). Therefore, the other child sub-tree(rooted at N7) of N3 is pruned, and the node value of N3 becomes 3, which is returned to parent N1. At N1, we again update α=max(4,3)=4 and the final utility of N1 becomes 4. All these steps are shown in Figure-6.

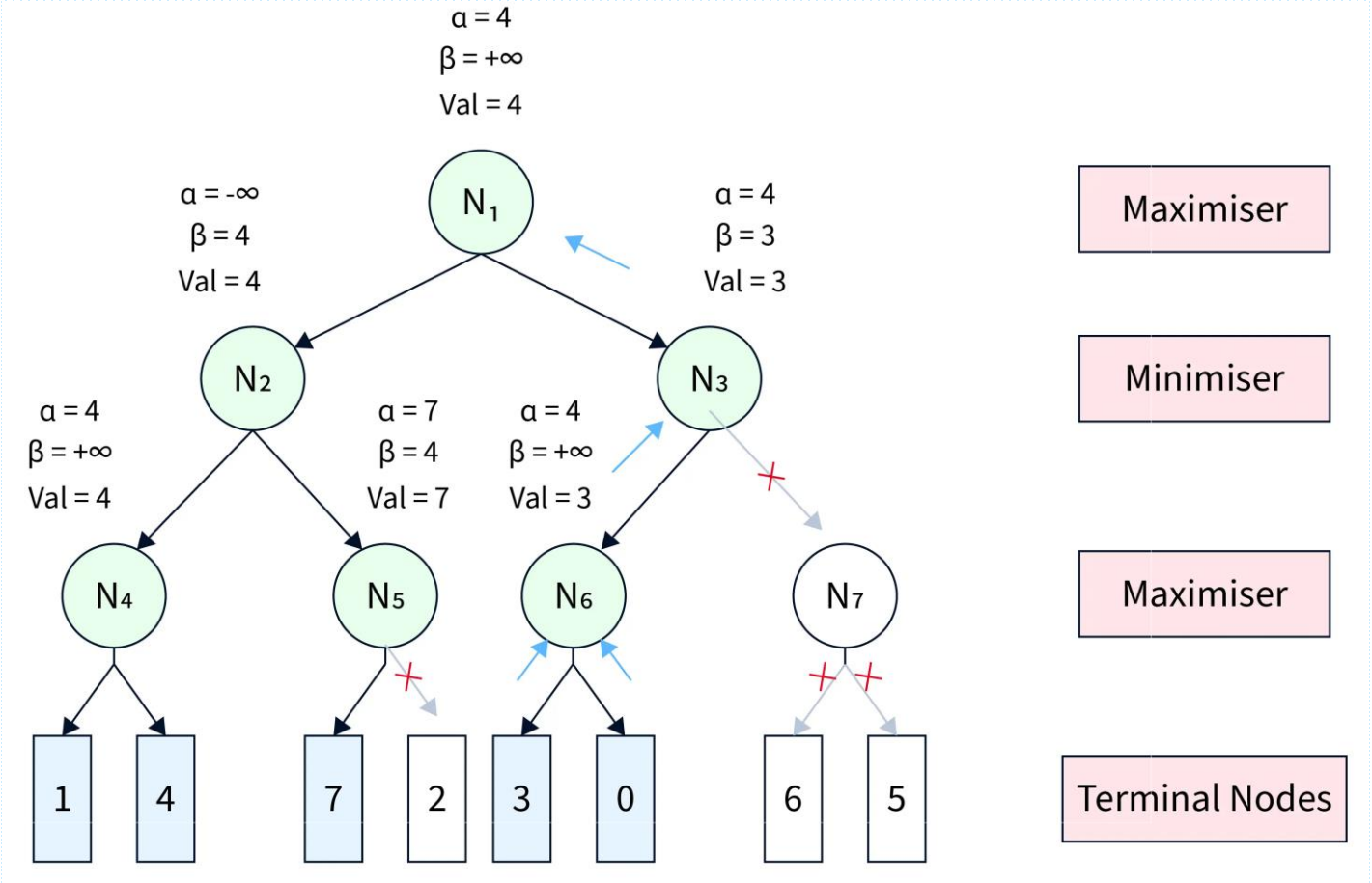


Figure - 6

CSPC3003

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

MODULE - 2

Course Objectives:

- To learn the concepts of Artificial Intelligence
- To learn the methods of solving problems using Artificial Intelligence
- To introduce the concepts of Expert Systems and machine learning

Human Intelligence vs AI

- **Humans** are naturally intelligent:
  - Use **logic, creativity, critical thinking** to solve problems.
  - **Learn from past experiences** and **adapt** to new situations.
  - Handle situations using **knowledge stored in their minds**.

- **Artificial Intelligence (AI):**
  - Tries to replicate human intelligence in machines.
  - **Knowledge-Based Agents (KBA)** are AI systems that store facts, rules, and relationships to make decisions.
  - Unlike humans, KBAs can **process huge volumes of data quickly and accurately**.

Knowledge-Based Agents (KBA)

- KBAs are AI agents that combine **stored knowledge** (facts + rules) and **reasoning techniques** to perform tasks intelligently.
- Knowledge based agents depend on a **knowledge base** which stores facts, rules, and information about the world.
- They use an inference system for reasoning and draw conclusions from the knowledge base.
- KBAs can adapt to new situations by adding or updating their knowledge.
- KBAs are commonly used in problem-solving, decision-making, planning tasks and in domains like healthcare, education, and legal systems.
- KBAs improve the scalability and accuracy of decision-making by handling large amounts of **structured data**.

Knowledge-Based Agents (KBA)

Purpose:

- Solve complex problems.
- Make decisions based on logical reasoning.
- Adapt to new environments by updating knowledge.

Applications:

- **Problem-solving & decision-making** (e.g., expert systems).
- **Planning & scheduling** (e.g., logistics).
- **Healthcare:** suggest diagnosis, detect patterns in patient data.
- **Business Intelligence:** analyze market trends, customer behavior, predict outcomes.
- **Education & Legal Systems:** provide recommendations based on stored knowledge.

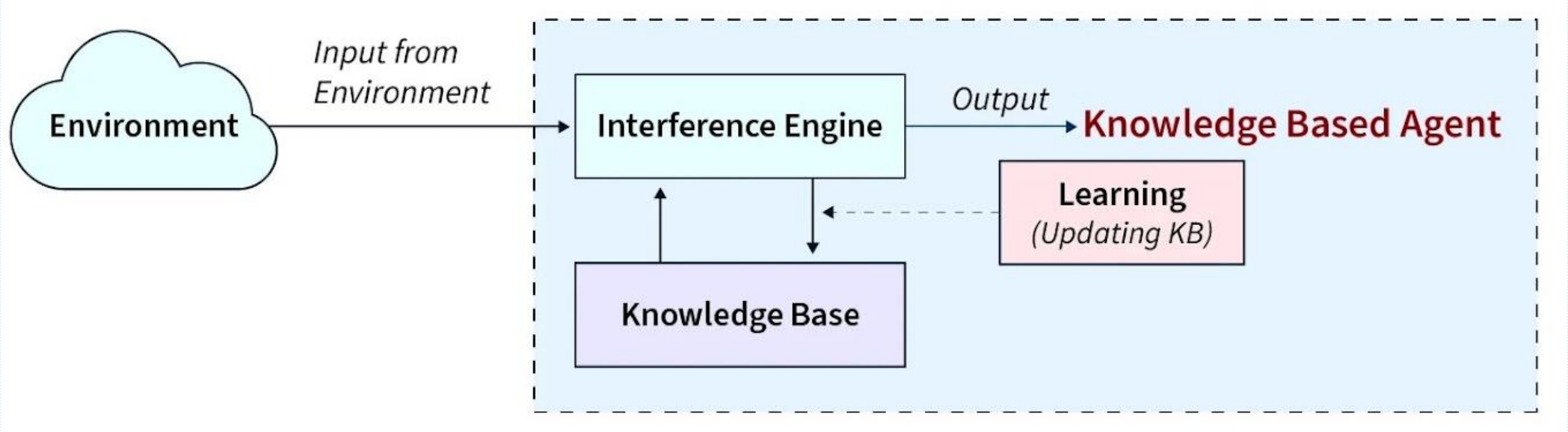
Architecture of Knowledge-Based Agents

Knowledge-based agents (KBAs) are structured to integrate, process, and apply knowledge for intelligent reasoning and decision-making. Their architecture consists of three key components:

a) Knowledge Base (KB)

b) Inference System (Reasoning Engine)

c) Sensors & Actuators



Architecture of Knowledge-Based Agents

a) Knowledge Base (KB)

❖ A central repository that contains all the information an agent needs about the world.

❖ **Content:**

- **Facts:** Statements that are true.
  - Example: *"The Earth orbits the Sun in 365.25 days."*
- **Rules:** Logical relationships between facts.
  - Example: *"If temperature < 0°C → water freezes into ice."*

❖ **Dynamic Nature:**

- KB can be **updated** with new knowledge → improves decision-making over time.
  - Example: Smart thermostat updates:
    - *"Maintain 22°C."*
    - *"If temperature > 25°C → turn ON AC."*

Architecture of Knowledge-Based Agents

b) Inference System (Reasoning Engine)

- Processes knowledge stored in KB.
- Draws **new conclusions** from known facts.
- Determines **what action should be taken next**.
- **Approaches:**
  - **Forward Chaining:**
    - Start with known facts → apply rules → infer new facts → reach conclusion.
    - Example: Soil is dry → infer "Turn ON sprinkler."
  - **Backward Chaining:**
    - Start with a goal → work backward to find which facts must be true.
    - Example: Want sprinkler ON → check if soil is dry.
- Used in expert systems, medical diagnosis, natural language processing, automated reasoning.

Architecture of Knowledge-Based Agents

c) Sensors and Actuators

- **Sensors:** Collect environmental data (temperature, motion, light).
- **Actuators:** Perform actions based on reasoning (turn AC ON, open door).
- **Example:**
  - Robot vacuum uses sensors to detect dirt.
  - Uses actuators to clean and avoid obstacles.

Operations Performed by KBA

- **TELL:** The agent updates its knowledge base with new information from the environment. For example, if a robot sees a door open, it tells the knowledge base, "The door is open."
- **ASK:** The agent asks the knowledge base what decision it should take. For example, it may ask, "Should I close the door or move forward?".
- **PERFORM:** The agent will follow the guidance of the knowledge base, such as closing the door or moving forward. It performs the selected option.

Architecture of Knowledge-Based Agents

Step-by-step process:

1. **Perception:** Sensors receive data from environment.
2. **Knowledge Retrieval:** KB retrieves relevant facts/rules.
3. **Reasoning:** Inference system applies logic (forward/backward chaining).
4. **Decision-Making:** Best action is selected.
5. **Action Execution:** Actuators perform the action.
6. **Learning/Updating:** New knowledge is added to KB for future use.

Levels of Knowledge-Based Agents

1. **Knowledge Level:**
   - Describes **what the agent knows** (facts, rules).
   - Example: "Soil is dry" + "If soil is dry → turn ON sprinkler."
2. **Logical Level:**
   - Describes **how the agent reasons** using logic.
   - Uses propositional logic or first-order logic to derive actions.
3. **Implementation Level:**
   - Physical/software realization of agent (algorithms, sensors, actuators).
   - Example: Moisture sensor + sprinkler + Python code to control system.

Knowledge Representation in AI

- Knowledge Representation is a core concept in Artificial Intelligence. It refers to the way knowledge is organized, stored, and structured inside an AI system so that it can be used for reasoning, problem-solving, and decision-making.

- The main goal of KR is to make AI systems understand and use information like humans do. Without proper representation, data is meaningless to AI. Knowledge must be converted into a form that computers can process logically.

- For example, when you see a cup of hot tea, your brain quickly recalls knowledge that “hot objects can burn the skin.” This prior knowledge warns you not to touch the cup. Similarly, for AI to behave intelligently, it needs structured representations of such rules and facts.

What To Represent in Knowledge Representation

1. **Object Knowledge:** Information about physical objects and their properties, for example, "A laptop has keyboard, mouse, screen". or "A tree has branches, leaves, and roots.“ This helps AI recognize and classify things in its environment.
2. **Event Knowledge:** Knowledge about actions or events, for example, "A traffic light turns red" or "A user clicks a button." This helps AI understand cause and effect.
3. **Performance Knowledge:** Knowledge of how to do tasks, like "How to Bake a Cake" or "How to Troubleshoot a Machine." It gives AI the efficiency to accomplish tasks.
4. **Meta Knowledge:** Knowledge of what the system knows, such as understanding *when to update the knowledge base or determining which information is most useful in a given context.*
5. **Factual Knowledge:** Verifiable statements or facts, such as "Water boils at 100C." It serves as the basis of AI reasoning and decision-making.
6. **Knowledge base:** A centralized source of maintaining facts, rules, and procedures, which enables the AI system to take decisions and address problems based on relevant knowledge.

Knowledge Vs Intelligence

- **Knowledge** is the information, facts, and skills that are acquired through learning or experience.
  - *Example*: Knowing that "water boils at 100°C."
- **Intelligence** is the ability to use that knowledge effectively for reasoning, problem-solving, and adapting to new situations.
  - *Example*: Using the boiling point of water to design an electric kettle.

In AI:

- Knowledge = Stored facts and rules (like a database).
- Intelligence = The ability of AI to use that knowledge to reason, learn, and make decisions.

Types of Knowledge in AI

1. Declarative Knowledge

- Declarative knowledge refers to facts, statements, or information that describe **"what is known"** about a domain. It can be described as static since the information can be represented as either assertions or truths.
- For example, declarative knowledge has facts like **"The sky is blue", "Delhi is capital of India", and "A triangle has three sides".** These statements are declarative knowledge because it is a **fact** that can be directly expressed and documented.

2. Procedural Knowledge

- Procedural knowledge refers to the knowledge, of **how to perform a task**. It involves **procedures, methods, or processes** that needed to achieve a task or solve a problem. It focuses on step-by-step procedures rather than just facts.
- **For example**, solving a quadratic equation is an structured process that includes determining coefficients, applying the quadratic formula, and simplifying the result.

Types of Knowledge in AI

3. Meta-knowledge is a term that refers to **"knowledge about knowledge."**

- It's not the actual fact itself, but information about **how useful, reliable, or when to apply that fact**.
- It allows AI to **understand what it knows**, how reliable the information is, and when to apply it.
- **Example: Autonomous Car**
  - **Knowledge (base level):**
    - "A red traffic light means stop."
    - "If a pedestrian is detected in the crosswalk, yield."
  - **Meta-knowledge (about that knowledge):**
    - "Camera sensor data is less reliable in heavy rain → rely more on radar and lidar in such conditions."
    - "Traffic rules can have exceptions: if an emergency vehicle is approaching, overriding the red light rule may be necessary."
    - "When two rules conflict (e.g., red light vs. emergency vehicle), prioritize safety and legality hierarchy."

Types of Knowledge in AI

4. Heuristic knowledge

- **Heuristic knowledge** means ***"knowledge based on experience, rules of thumb, or practical shortcuts"*** rather than guaranteed, perfect solutions.
- It helps an AI (or a human) make decisions faster when an exact or complete method would be too slow or complex.
- It helps in decision-making when specific rules or formulae are not available.
- **Example: In route-finding (Google Maps):**
  - **Exact method:** Check all possible routes and traffic conditions.
  - **Heuristic knowledge:** "Highways are usually faster than local roads."
- **Example: Medical Diagnosis AI**
  - **Knowledge (exact):** "Run every possible medical test to identify the disease with certainty." (but this is expensive and time-consuming)
  - **Heuristic knowledge (rule of thumb):**
    - "If the patient has a sore throat and fever, first check for common infections like flu or strep throat before rare diseases."

Types of Knowledge in AI

5. Structural Knowledge

- **Structural knowledge** means *knowledge about how different pieces of information are connected to each other*.
- It shows **relationships, hierarchies, and organization** between concepts rather than just listing facts.
- **Example:**
  - **In AI (Knowledge Representation):**
    - **Facts (declarative knowledge):**
      - "A car has wheels."
      - "A car has an engine."
      - "An engine needs fuel."
    - **Structural knowledge:**
      - "A car is made of parts → wheels, engine, body."
      - "The engine is a sub-part of the car."
      - "The engine is connected to the fuel system."

Inference Engine Techniques (Forward Chaining)

Forward Chaining

❖ Forward chaining is a **reasoning method** used in **Artificial Intelligence (AI)** and **Expert Systems** to derive conclusions.

❖ It is a **data-driven approach** — meaning it **starts with the available facts (data)** and **applies rules** to infer new facts until a goal is reached.

❖ How Forward Chaining Works

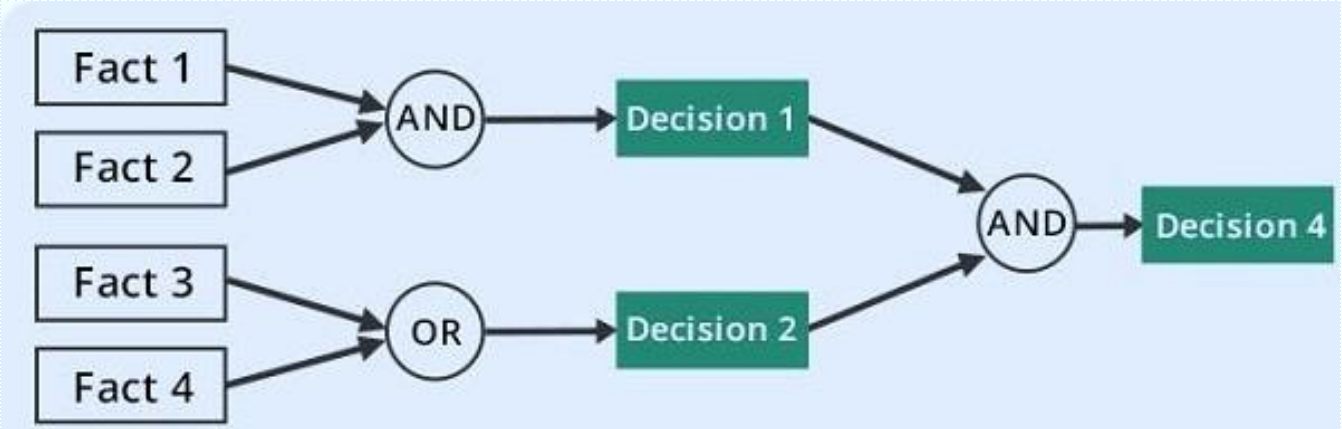
- **Start with Known Facts** – The system begins with what is already true.
- **Match Facts with Rules** – It looks for rules whose conditions are satisfied by the known facts.
- **Apply Rule** – If a rule's conditions are true, the system executes its conclusion (adds new fact).
- **Repeat** – The process continues until:
  - No more rules can be applied, or
  - The desired goal is achieved.

Inference Engine Techniques (Forward Chaining)

Example of Forward Chaining

Let's take a simple **medical diagnosis example:**

- **Knowledge Base (Rules)**
  - **Rule 1:** IF the patient has a fever AND cough → THEN the patient might have flu
  - **Rule 2:** IF the patient has flu → THEN advise rest and fluids
  - **Rule 3:** IF the patient has fever AND rash → THEN the patient might have measles
- **Facts (Known Information)**
  - Patient has fever
  - Patient has cough



Forward Chaining Process

Step 1:

- Known facts = {fever, cough}
- Check Rule 1:
  - Condition: fever AND cough → True
  - So, **conclude:** patient might have flu
- Now facts = {fever, cough, flu}

Inference Engine Techniques (Forward Chaining)

Step 2:

- Check Rule 2:
  - Condition: patient has flu → True
  - So, **conclude:** advise rest and fluids
- Now facts = {fever, cough, flu, advise rest and fluids}

Step 3:

- Check Rule 3:
  - Condition: fever AND rash → False (**rash not in facts**)
  - ***Cannot apply Rule 3***

Inference Engine Techniques (Forward Chaining)

Example: Given

- **Knowledge Base (Rules):**
  - IF A and C → E
  - IF A and E → G
  - IF B → E
  - IF G → D
- **Facts:**
  - A, B
- **Goal:**
  - D

Step 1: Start with Known Facts

Facts = {A, B}

Check rules whose conditions are satisfied:

Rule 3: IF B → E (B is known)

So, **derive E**

Now facts = {A, B, E}

Step 2: Use New Facts

Facts = {A, B, E}

•Check rules again:

- **Rule 2:** IF A and E → G (both are known)
  - So, **derive G**

Now facts = {A, B, E, G}

Step 3: Continue Inference

Facts = {A, B, E, G}

Check rules again:

Rule 4: IF G → D (G is known)

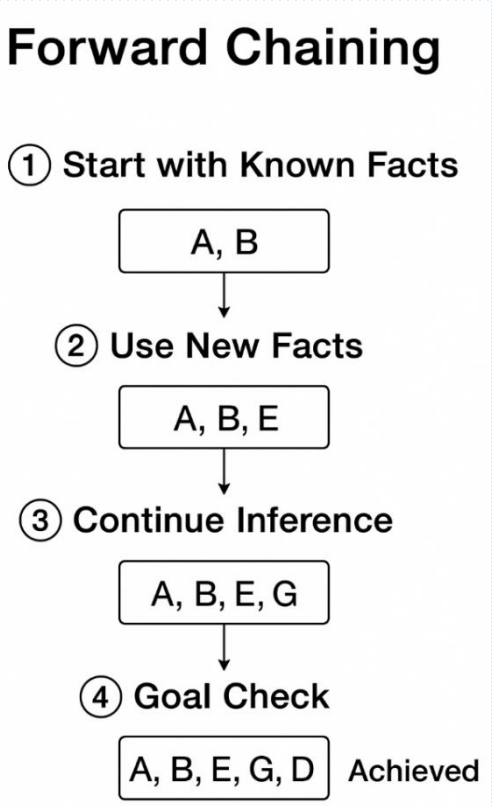
So, **derive D**

Now facts = {A, B, E, G, D}

Step 4: Goal Check

Goal = D → Achieved

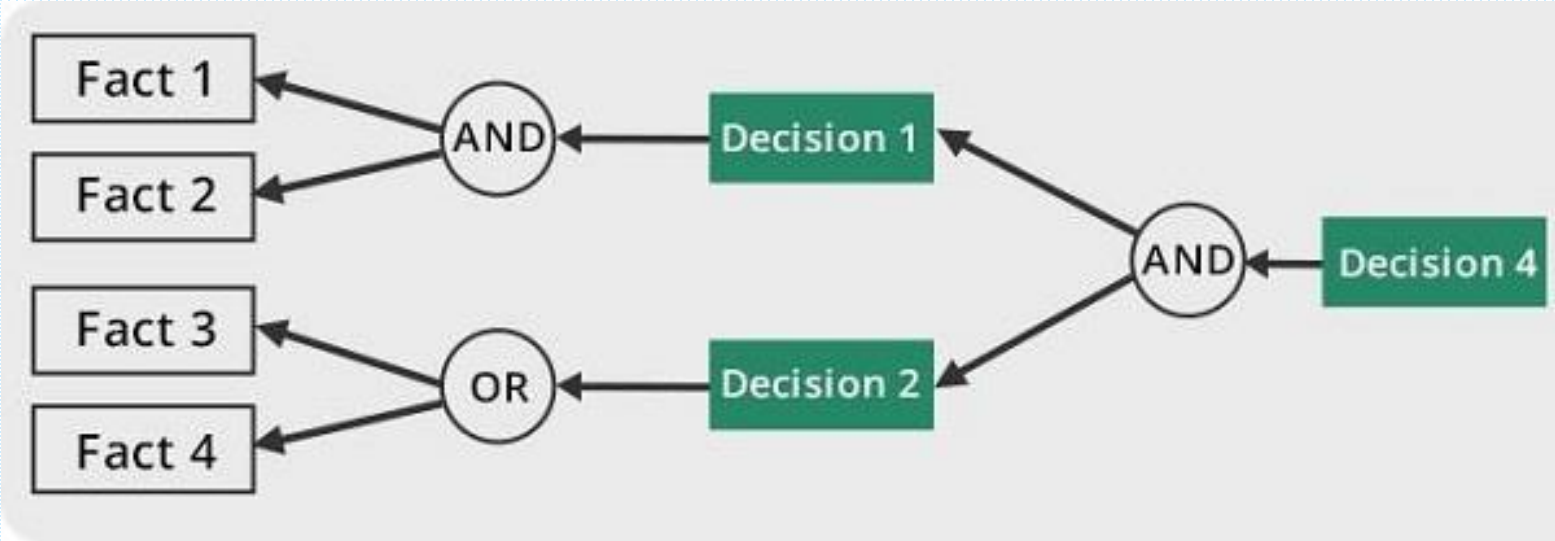
Stop reasoning (no need to check further rules).



Inference Engine Techniques (Backward Chaining)

Backward Chaining

- Backward chaining is a **reasoning method** used in **AI** and **Expert Systems** that starts from the **goal (or hypothesis)** and works **backward** to find supporting facts.
- It is a **goal-driven approach** — meaning it **starts with the conclusion** you want to prove and **looks for rules** that could lead to that conclusion, verifying each condition until it finds enough facts to support it (or fails).



Inference Engine Techniques (Backward Chaining)

❖ How Backward Chaining Works

- **Start with Goal** – Select the goal (what you want to prove).
- **Look for Rules that Conclude Goal** – Find rules where the goal appears in the THEN (conclusion) part.
- **Check Conditions** – For each rule, check whether its IF part (premises) are satisfied:
  - If a condition is already a known fact → good.
  - If not, treat the condition as a **new sub-goal** and repeat the process.
- **Continue Until**:
  - All conditions are satisfied (goal proven), OR
  - No rule can be applied (goal cannot be proven).

Inference Engine Techniques (Forward Chaining)

Example: Given

- **Knowledge Base (Rules):**
  - IF A and C → E
  - IF A and E → G
  - IF B → E
  - IF G → D
- **Facts:**
  - A, B
- **Goal:**
  - D

Step 1:

Goal = D

Look for a rule where conclusion is D → **Rule 4**

IF G → D

So, we need to prove **G** first.

(**New sub-goal: G**)

Step 2: Sub-goal = G

Look for a rule where conclusion is G → **Rule 2**

IF A and E → G

We must check **A and E**

A is already a fact. Need to **prove E (new sub-goal)**

Step 3: Sub-goal = E

Look for a rule where conclusion is E → **Rule 1, Rule 3**

IF A and C → E (need C, but C not known)

IF B → E (B is known)

So, we can derive E using Rule 3. Now E is proven

Step 4:

Go back to G:

A and E → G is proven

Step 5:

Go back to D:

G → D is proven

Inference Engine Techniques (Forward Chaining)

Forward Chaining	Backward Chaining
Starts from known facts and applies inference rules to extract more data until the goal state is reached. (Fact → Goal)	Starts from the goal and works backward using inference rules to find facts that support the goal. (Goal → Fact)
Bottom-up approach	Top-down approach
Known as **data-driven approach** because it uses available data to reach the goal.	Known as **goal-driven approach** because it starts from the goal and tries to satisfy it.
Applies **BFS (Breadth-First Search)** technique	Applies **DFS (Depth-First Search)** technique
Operates in forward direction	Operates in backward direction
Can generate multiple possible conclusions from facts	Can generate multiple possible sets of facts to support a goal
Tests **for all available data**	Tests **for few required facts**
Suitable for **planning, monitoring, control, interpretation** applications	Suitable for **diagnosis, prescription, debugging** applications

Knowledge Representation Techniques in AI

Logical Representation

- Uses **mathematical logic** (propositional logic & predicate logic).
- Knowledge is expressed in the form of **facts, predicates, and logical connectives** (AND, OR, NOT, →).
- Inference is made through logical reasoning (deduction, resolution).
- Very **precise and unambiguous**.
- Allows **proof-based reasoning**.
- Good for **rule-based systems** (mathematical theorems, verification).
- Struggles with **uncertainty, incomplete, or approximate knowledge**.
- **Examples:**

1. **Propositional Logic:**
   - Statement: “It is raining.” R, “The ground is wet.” W
   - Rule: "If it is raining, then the ground is wet." R → W
   - Inference: If R is true, then W is true.
2. **Predicate Logic:**
   - "All humans are mortal." → ∀x (Human(x) → Mortal(x))
   - "Socrates is a human." → Human(Socrates)
   - Conclusion: Mortal(Socrates).

Knowledge Representation Techniques in AI

Semantic Networks

- Represents knowledge as a **graph**.
- **Nodes = concepts/objects, Edges = relationships.**
- Inheritance property: sub-nodes inherit attributes from parent nodes.
- **Intuitive and visual** (easy to understand).
- Supports **hierarchical relationships** ("is-a", "has-a").
- Good for **taxonomies and concept hierarchies**.
- Can become **complex** with many exceptions.
- **Examples:**

Animals:

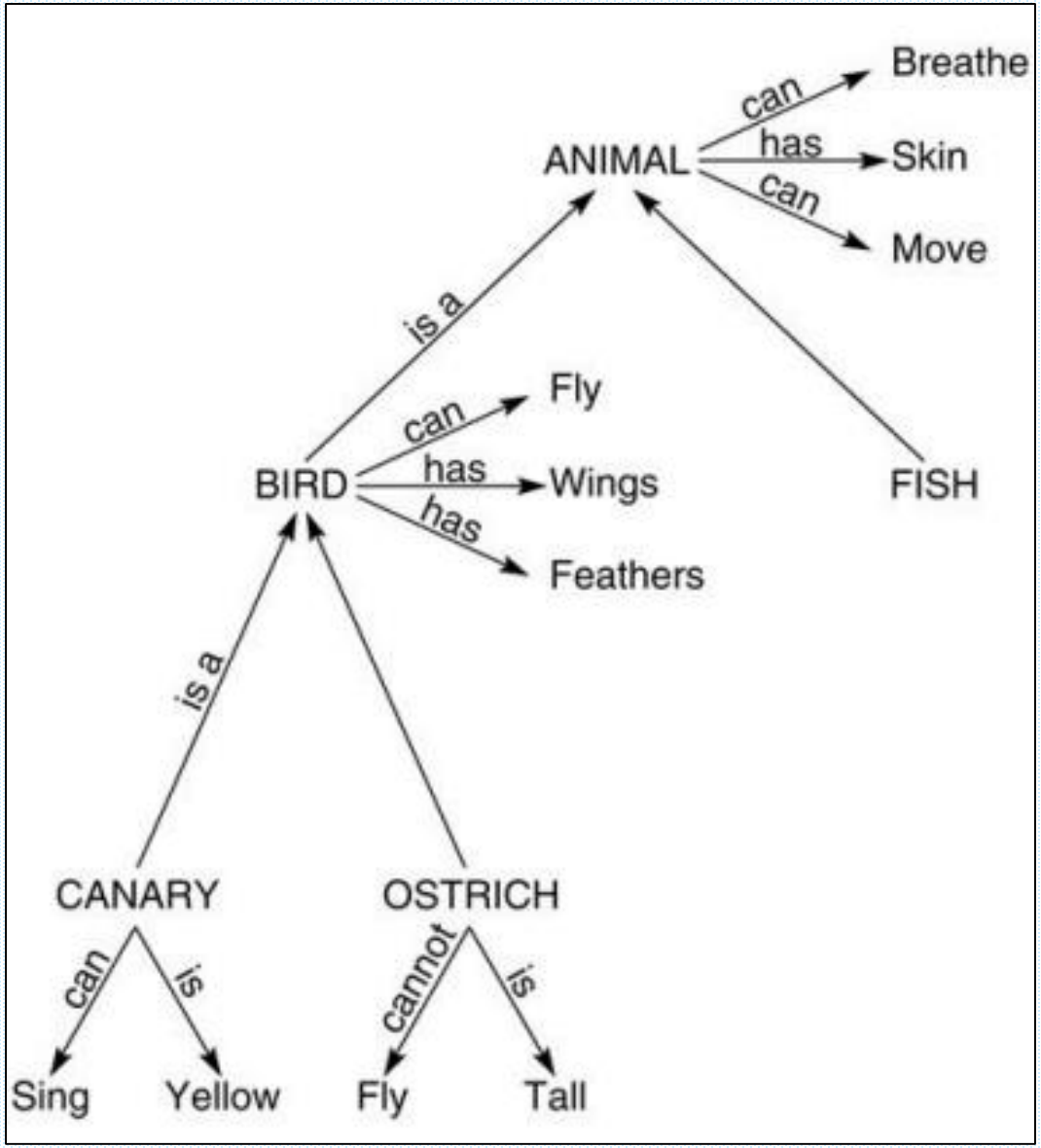
Node: Bird → has property “Can Fly”

Node: Ostrich → is-a Bird, but property overridden “Cannot Fly”

Family Tree:

Node: Person → has relationships (Father-of, Mother-of, Child-of).

Helps represent genealogy knowledge.



Knowledge Representation Techniques in AI

Frame Representation

- Knowledge is stored in **frames** (data structures like objects).
- Each frame has **slots (attributes)** and **values** (specific details).
- It can represents **Objects, Facts and Procedures**.
- It can represents **declarative knowledge (what), procedural knowledge (how).**
- Supports **default values and inheritance**.
- Similar to **Object-Oriented Programming**.
- Good for representing **real-world entities**.
- Not suitable for highly **abstract or dynamic knowledge**.
- **Examples:**

Car Frame:

Frame: Car

Slots: Wheels = 4, Engine = Petrol/Diesel, MaxSpeed = 180 km/h

Frame: Tesla Model 3 (inherits from Car, Engine = Electric, MaxSpeed = 250 km/h).

Hospital Frame:

Frame: Hospital

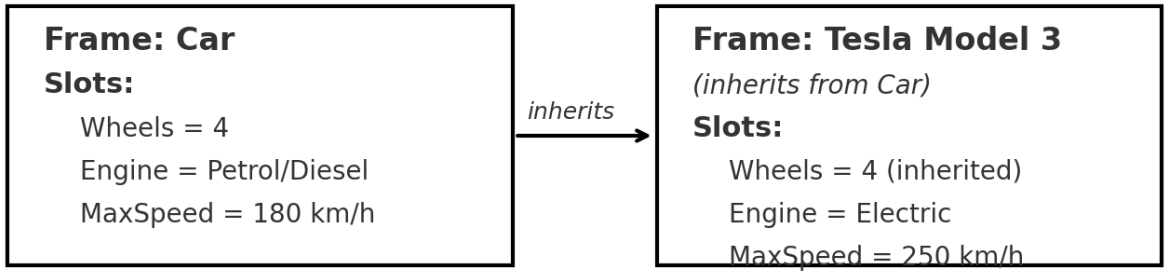
Slots: Doctors = many, Beds = 200, Services = Surgery, OPD

Specific: Apollo Hospital (Beds = 1000, Special Services = Oncology).

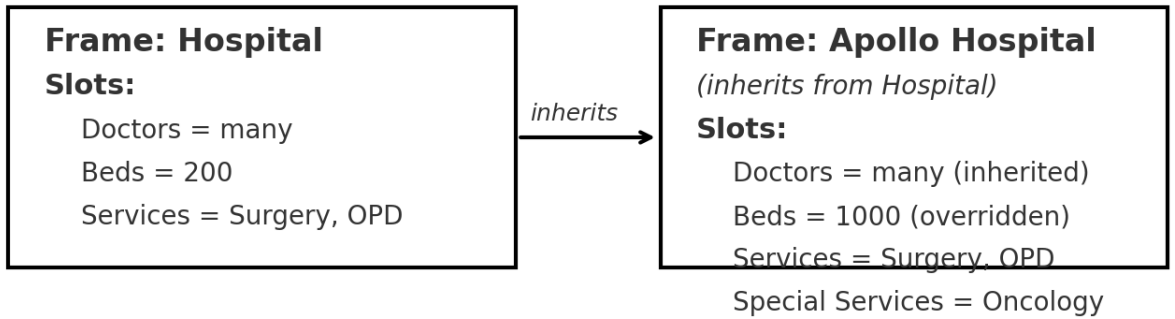
Knowledge Representation Techniques in AI

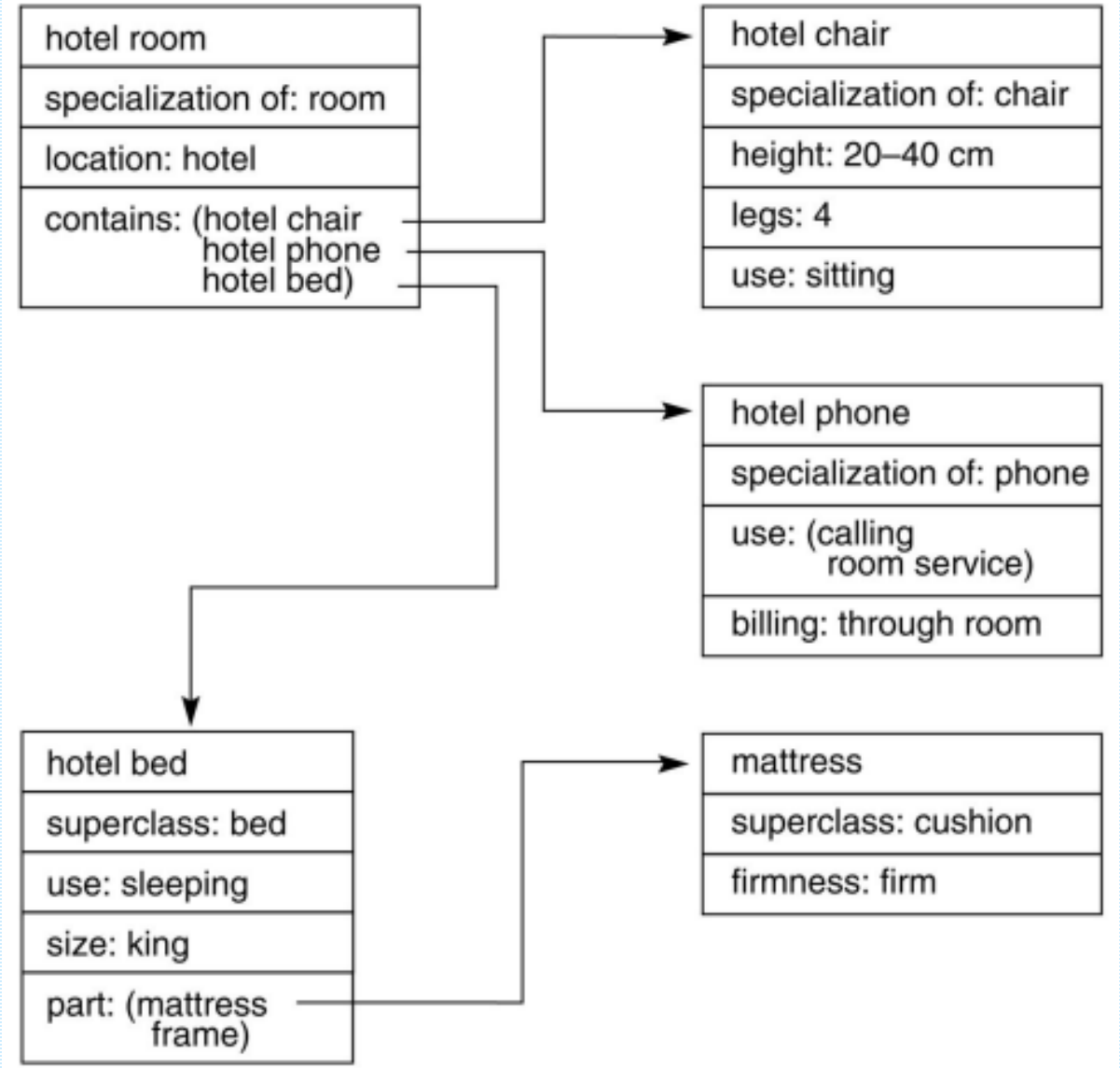
Frame Representation

Frame Representation: Car and Tesla Model 3



Frame Representation: Hospital and Apollo Hospital





Knowledge Representation Techniques in AI

Production Rules

- Knowledge is represented as **IF-THEN rules**.
- Inference engines use **forward chaining** (data → goal) or **backward chaining** (goal → data).
- Widely used in **expert systems**.
- **Simple and modular** (easy to add/remove rules).
- Very effective for **decision-making**.
- Good for **problem-solving, diagnostics, planning**.
- Can lead to **rule explosion** in large systems.
- **Examples:**

Medical Diagnosis Expert System:

Rule 1: IF patient has fever AND cough THEN patient may have flu.

Rule 2: IF patient has rash AND fever THEN patient may have measles.

Cooking Assistant:

Rule: IF user has rice AND vegetables THEN suggest “Fried Rice Recipe.”

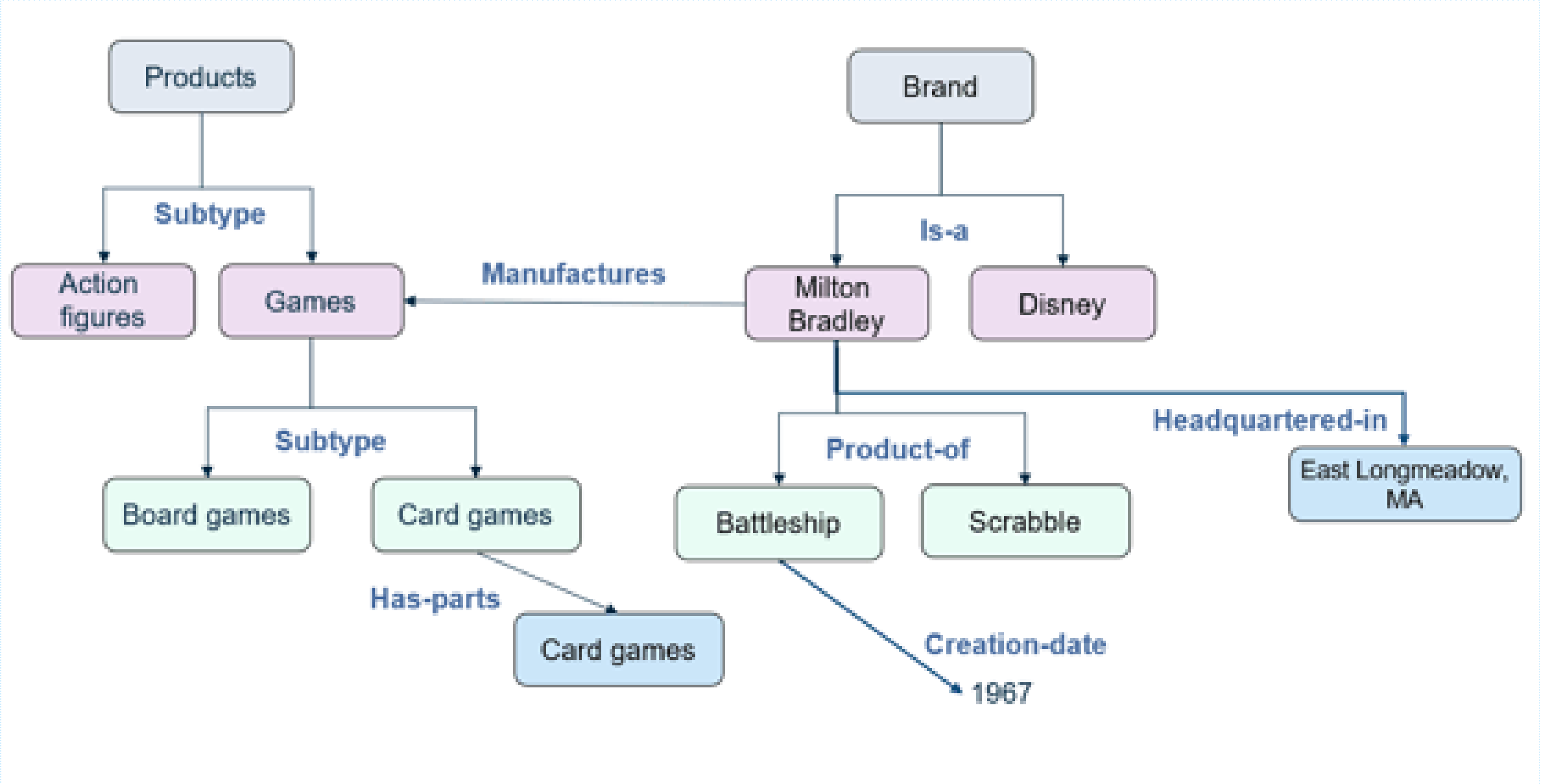
Knowledge Representation Techniques in AI

Ontology (Modern KR)

- A structured **formal representation of knowledge** about a domain.
- Defines **concepts, categories, properties, and relationships**.
- Basis of **Semantic Web, Knowledge Graphs, NLP systems**.
- Provides **shared understanding** across systems.
- Machine-readable (can be processed by AI agents).
- Good for **large-scale systems (healthcare, e-commerce, education)**.
- Complex to **design and maintain**.
- **Examples:**
  - **Healthcare Ontology:**
    - Class: Disease
    - Subclasses: ViralDisease, BacterialDisease
    - Relationship: COVID-19 is-a ViralDisease; hasSymptom(Fever, Cough).
  - **E-commerce Ontology:**
    - Class: Product → subclasses Electronics, Clothing
    - Property: hasPrice, hasBrand
    - Example: iPhone is-a Electronics; hasBrand = Apple; hasPrice = $999.

Knowledge Representation Techniques in AI

Ontology (Modern KR)



CSPC3003

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

MODULE - 2

Course Objectives:

- To learn the concepts of Artificial Intelligence
- To learn the methods of solving problems using Artificial Intelligence
- To introduce the concepts of Expert Systems and machine learning

Logical Knowledge Representation Techniques in AI

We can represent the knowledge logically in 2 ways:

1. Propositional Logic (PL)
2. First Order Propositional Logic (FOPL) / First-Order Predicate Logic (FPL)

1. Propositional Logic (PL)

❖ **Proposition**

- Contains *atomic meaning*.
- Atomic sentence that cannot be broken further.
- We represent the atomic sentence in notation of AI.

❖ **Symbols**: P, Q, X, Y, M, S → used to denote *atomic sentences*.

❖ **Example:**

- X = "It is raining"
- This is an *atomic sentence* (basic, independent, indivisible).
- Symbol of assignment: X = "It is raining"
- Atomic sentences check note (independent of any action) can be denoted by English capital letters.

Propositional Logic in AI

❖ Example:

- X = "It is raining" in **New York**
- Here it **cannot be represented using only X** because additional context is needed.
- **Dependency/relationship**:
  - Changes according to different functions used.
  - X → "Raining“, Y → "New York"
  - Representation:
  - Y(X), X(Y) → 2 possible atomic sentences.

❖ Connectives in PL

- To join two atomic sentences, the connectives are:

∧	→ And
∨	→ Or
¬	→ Not
→	→ Implication
↔	→ If and only if

Propositional Logic in AI

- While writing any sentence, we keep in mind that one is **syntactic representation** and the other one is **semantic**.
  - **Syntax**
    - Arrangement of symbols in a sequence.
  - **Semantics**
    - The *meaningful sentence*.
    - When a sentence has meaning, it is *semantically valid*.
- **Example:**
  - **Sentence:** "It is raining"
  - **Syntax:** valid sentence structure.
  - **Semantics:** meaningful only if it conveys truth.

If it is raining, we get full information and it is meaningful. But if the truth value is not determined, then it is not considered valid.

Propositional Logic in AI

Propositional Logic (PL)

- It is the simplest form of logic where all statements are made by using *propositions*.
- A **proposition** is a *declarative sentence* which is either **true** or **false**.
- It is a technique of knowledge representation in **logical/mathematical form**.
- **Examples of Propositions**
  - "Today is Sunday" → **True proposition**
  - "Sun rises from West" → **False proposition**
  - "3 + 3 = 7" → **False proposition**
  - "7 is a prime number" → **True proposition**

Propositional Logic in AI

Basic Rules of Propositional Logic (PL)

Rule 1: **PL is also called Boolean Logic** as it works on True/False values.

Rule 2: In PL, we use **symbolic variables** to represent the logic. Ex.: We can use any symbol for representing a proposition, such as **A, B, C, P, Q, R, X, etc.**

Rule 3: A proposition can be either **True or False**, but it cannot be both.

Rule 4: PL consists of an **object**, **Relation or function**, and **Logical connectives**.

Rule 5: The connectives are also called **logical operators**.

Rule 6: **Propositions + Connectives** are the basic elements of propositional logic.

Rule 7: **Connectivity** can be said as a logical operator which connects two sentences.

Rule 8: A proposition formula which is always **false** is called a **Contradiction / False**.

Rule 9: A proposition formula which has both **true and false** is called a **Contingency**.

Rule 10: A proposition formula which is always **true** is called a **Tautology** (valid / logically valid statement).

Propositional Logic in AI

Syntax of PL (Propositional Logic)

- The **system of PL** defines the allowable sentence for the knowledge representation.
- There are **two types of proposition**:
  1. **Atomic Proposition**
  2. **Compound Proposition**

1. Atomic Proposition

- Atomic propositions are **simple propositions**.
- It consists of a **single propositional symbol**.
- These are the sentences which must be either **true or false**.
- **Examples:**

  a) "It is an atomic proposition and it is true."

  b) "Sun is cold." → This is also atomic, but it is **false**.

Propositional Logic in AI

2. Compound Proposition

- Constructed by **combining two or more atomic propositions** using **logical connectives** (AND, OR, NOT, →, ↔).
- **Examples:**
  - "It will rain tomorrow **or** it will be sunny."
  - "It is raining today and street is wet."
  - "If you work hard, then you will succeed."
  - "The sky is clear **and** the sun is shining."
  - "You will pass the exam **if and only if** you study."
  - "The sun rises in the west."

Propositional Logic in AI

Logical Connectives

- Logical connectives are used to connect two simple propositions (or representing a sentence logically).
- They help create **compound propositions**.
- There are **5 logical connectives**:
  1. Negation (¬)
  2. Conjunction (∧)
  3. Disjunction (∨)
  4. Implication (→)
  5. Bidirectional (↔)

1. Negation (~, ¬)

- A sentence such as ¬P is called the **negation** of P.
- Literal can be either **positive** or **negative literal**.

Propositional Logic in AI

2. Conjunction (∧)

- A sentence which has an **∧ (AND)** connectivity such as P ∧ Q is called a **conjunction**.
- **Example:**
  - "Rohan is intelligent and hardworking."
  - Let:
    - P = "Rohan is intelligent"
    - Q = "Rohan is hardworking"
  - Then, **P ∧ Q = Rohan is intelligent and hardworking**.

3. Disjunction (OR)

- A **disjunction** is a sentence that uses the logical connective **∨ (OR)**.
- If P and Q are propositions, then:

  P ∨ Q means **"P or Q"** (or both).
- **Example:**
  - P: Ritika is a doctor.
  - Q: Ritika is an engineer.
  - P∨Q: Ritika is a doctor **or** engineer.
  - This is **true** if **either** P or Q (or both) are true.

4. Implication (IF–THEN)

- An **implication** (also called a conditional statement) is written as:
- P→Q This means **"If P, then Q"**.
- **Example:**
  - P: It is raining.
  - Q: The street is wet.
  - P→Q: If it is raining, then the street is wet.
- Implication is **false only when P is true but Q is false** (because that would contradict the "if–then" relationship).

5. Biconditional (IF AND ONLY IF)

- A **biconditional** is written as:
- P↔Q This means **"P if and only if Q"** (P and Q must both be either true or false).
- **Example:**
  - P: I am breathing.
  - Q: I am alive.
  - P↔Q: I am breathing **if and only if** I am alive.
- This is **true** when P and Q have the same truth value (both true or both false).

Propositional Logic in AI

Truth Table

- In propositional logic, we need to know the **truth table** of propositions in all possible scenarios.
- We can combine all possible combinations of the truth value of the representation with the logical connectivity of the representation as the combination in a tabular format so-called **truth table**.

P	¬P
T	F
F	T

Negation

P	Q	P ∧ Q
T	T	T
T	F	F
F	T	F
F	F	F

Conjunction

P	Q	P ∨ Q
T	T	T
T	F	T
F	T	T
F	F	F

Disjunction

P	Q	P → Q
T	T	T
T	F	F
F	T	T
F	F	T

Implication

P	Q	P ↔ Q
T	T	T
T	F	F
F	T	F
F	F	T

Bidirectional

Propositional Logic in AI

Logical equivalence:

- It is one of the features of propositional logic (PL). Two propositions are **logically equivalent** if (if and only if) the columns in the truth table are identical to each other.
- Let's take two propositions A & B. For logical equivalence we can write A⇔B

Truth table

A	B	¬A	¬A ∨ B	A → B
T	T	F	T	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

In the above truth table we can see that the columns for ¬A ∨ B and A → B are identical. Hence ¬A ∨ B ⇔ A → B (they are equivalent).

Propositional Logic in AI

Properties of Operators (Logical Connectives)

- **Commutative Law**

  $P \wedge Q = Q \wedge P$

  $P \vee Q = Q \vee P$

- **Associative Law**

  $(P \wedge Q) \wedge R = P \wedge (Q \wedge R)$

  $(P \vee Q) \vee R = P \vee (Q \vee R)$

- **Identity Law**

  $P \wedge T = P$

  $P \vee F = P$

- **Distributive Law**

  $P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$

  $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$

- **De Morgan's Laws**

  $\neg(P \wedge Q) = \neg P \vee \neg Q$

  $\neg(P \vee Q) = \neg P \wedge \neg Q$

- **Double Negation**

  $\neg(\neg P) = P$

Propositional Logic in AI

Compound Propositions & Truth Tables

We can build a proposition using several variables P,Q,R.

Example 1

Columns: P, Q, R, ¬R, P ∨ Q → ¬R

P	**Q**	**R**	**¬Q**	**PVQ**	**(PVQ)→¬R**
T	T	T	F	T	F
T	T	F	F	T	T
T	F	T	T	T	F
T	F	F	T	T	T
F	T	T	F	T	F
F	T	F	F	T	T
F	F	T	T	F	T
F	F	F	T	F	T

Example 2

Expression: (P∧¬Q)→R∨Q

P	Q	R	¬Q	(P∧¬Q)	(P∧¬Q)→R	(P∧¬Q)→R∨Q
T	T	T	F	F	T	T
T	T	F	F	F	T	T
T	F	T	T	T	T	T
T	F	F	T	T	F	F
F	T	T	F	F	T	T
F	T	F	F	F	T	T
F	F	T	T	F	T	T
F	F	F	T	F	T	T

Conclusion: Because the final column is **not all T**, this is **not** a tautology, but it is a valid statement in specific rows.

Propositional Logic in AI

In logic, a **tautology** is a statement (or compound proposition) that is **always true**, no matter what truth values its individual parts have.

Key points:

- It stays **True** for **every possible combination** of truth values for its variables.
- In a **truth table**, the final column for a tautology contains **only T (True)**.

Examples

- P∨¬P ("P or not-P") – This is always true, because either P is true or P is false.
- (P∧Q)→P – If P and Q are both true, P is true; if not, the implication is still true.

P	¬P	P ∨ ¬P
T	F	T
F	T	T

P	Q	P ∧ Q	(P ∧ Q) → P
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	T

Propositional Logic in AI

Limitations / Disadvantages of Propositional Logic (PL)

- It has **limited expressive power**.
- PL can describe a statement in terms of its **truth values** and logical relations, but not more complex structures.

Propositional Logic in AI

Inference Rules in Propositional Logic

- In AI, we need **intelligent computers** which can create **new logic from old logic** so as to generate conclusions from evidence. This process is called **inference**.
- **Inference rules** are the simplest form of generating valid statements.
- Inference rules are applied to derive **proof**.
  - Proof = a sequence of conclusions that lead to the desired goal.
- In inference rules, the **implication** (→) and other **logical connectives** play an important role.
- **Premises (assumptions) –** statements assumed true.

Propositional Logic in AI

General Inference Rules

1. **Implication (→)** – one of the five logical connectives, written as P → Q. It is a **Boolean expression**.
2. **Converse:** The converse of implication means that the **right-hand side proposition goes to the left-hand side**, and vice versa.

   If P → Q, then converse is Q → P.
3. **Contrapositive:** Negation of converse is called **contrapositive**.

   If P → Q, then contrapositive is ¬Q → ¬P.
4. **Inverse:** Negation of implication is called **inverse**.

   If P → Q, then inverse is ¬P → ¬Q.

Propositional Logic in AI

Types of Inference Rules

Generally, there are many inference rules.

Some important ones are:

1. Modus Ponens

- One of the most important inference rules and It states that **if P is true, and P → Q is true, then Q must also be true**.
- It can be represented as notation:

  P → Q, P

  ∴ Q

P	Q	P → Q	Premises (P and P→Q)	Conclusion (Q)	Valid?
T	T	T	True	T	✓ Valid
T	F	F	False (premise fails)	F	✗ Not applicable
F	T	T	False (P is false)	-	✗ Not applicable
F	F	T	False (P is false)	-	✗ Not applicable

- **Example for Modus Ponens**
  - **Statement 1**: If I am sleepy, then I go to bed. **P→Q**
  - **Statement 2**: I am sleepy. **P**
  - **Conclusion**: I go to bed. **Q**
  - ➢ Hence, we can say that if P→Q is true and P is true, then Q will also be true

2. Modus Tollens

- This rule states that **if P→Q is true and ¬Q is true, then ¬P will be true.**
- Notation: P→Q,¬Q ∴ ¬P
- **Example:**
  - Statement 1: If I am sleepy, then I go to bed. ⇒ **P→Q**
  - Statement 2: I do not go to bed. ⇒ **¬Q**
  - Conclusion: I am not sleepy. ⇒ **¬P**

P	Q	P → Q	¬Q	Premises (P→Q ∧ ¬Q)	Conclusion (¬P)	Valid?
T	T	T	F	False	F	✗ Not applicable
T	F	F	T	False (P→Q fails)	F	✗ Not applicable
F	T	T	F	False (¬Q fails)	T	✗ Not applicable
F	F	T	T	True	T	✓ Valid

Propositional Logic in AI

3. Hypothetical Syllogism

- This rule states that **if P → R is true whenever P → Q is true and Q → R is true**.
- **Example:**
  - **Statement 1**: If you have my home key, then you can unlock my home. ⇒ **P→Q**
  - **Statement 2**: If you can unlock my home, then you can take my money. ⇒ **Q→R**
  - **Conclusion**: If you have my home key, then you can take my money. ⇒ **P→R**

P	Q	R	P → Q	Q → R	P → R	Validity
T	T	T	T	T	T	✓ Valid
T	T	F	T	F	F	✗ Not applicable
T	F	T	F	T	T	✗ Not applicable
T	F	F	F	T	F	✗ Not applicable
F	T	T	T	T	T	✓ Valid
F	T	F	T	F	T	✗ Not applicable
F	F	T	T	T	T	✓ Valid
F	F	F	T	T	T	✓ Valid

- Hypothetical Syllogism **only fails** in cases where the chain of implications is broken (when Q → R is false or P → Q is false).
- If both premises are true, then the conclusion P→R **must** be true.

First Order Logic (FOL) or First Order Predicate Logic (FOPL) in AI

- In the topic of propositional logic, we have seen that how to represent the statement using PL. But unfortunately, propositional logic can only represent the fact which is either true or false.
- PL is not sufficient to represent the complex statement.
- PL has very limited expressive power.
- Consider the following sentence, which we can't represent using PL logic:
  - Some humans are intelligent.
  - Sachin likes cricket.
- To represent the above statement PL logic is not sufficient. So we require some more powerful logic such as First Order Logic (FOL) or First Order Propositional Logic (FOPL).

First Order Logic (FOL) or First Order Predicate Logic (FOPL) in AI

First Order Logic (FOL)

- FOL is another way of knowledge representation in AI. It is an extension to PL.
- FOL is sufficiently expressive to represent the natural language statement on a clear manner and concise way.
- FOL is also known as **Predicate Logic** or **First Order Predicate Logic**.
- FOL is a powerful language that develops information about the object in a more easy way and can express the relationship between the objects.
- FOL doesn't only assume that the world contains facts like PL, but also assumes the following things in the world:
  - **Object:** A, B, people, number, colour, war, theories, square.
  - **Relation:** It can be unary relation such as *is red, round, is adjacent*, or any relation such as *sister of, brother of, has colour, comes between*.
  - **Function:** *Father of, bestfriend of, 3rd son of, end of.*

First Order Logic (FOL) or First Order Predicate Logic (FOPL) in AI

- As in natural language, FOL has two main parts: one is **Syntax** and another one is **Semantic**.
- **Syntax:** Syntax of FOL determines which collection of symbols is a logical sentence.
- Semantics: **meaning** of the symbols and sentences in FOL. It tells us **how to interpret** the constants, variables, predicates, and quantifiers in the real world.

Basic Elements of FOL

1. Constants

- These represent specific objects in the domain.
- They are like proper nouns.
- **Example:**
  - John, Mumbai, 5, A, etc.
  - If our domain is "People", constants can be John, Mary.
  - Example sentence: **Human(John)** → “John is a human.”

First Order Logic (FOL) or First Order Predicate Logic (FOPL) in AI

2. Variables

- Symbols that can take values from the domain.
- They are like placeholders.
- **Example:**
  - x, y, z
  - Sentence: **Human(x)** → "x is a human" (where x could be John, Mary, etc.)

3. Predicates

- Express properties of objects or relationships between objects.
- Predicates are like verbs/adjectives.
- **Example:**
  - Unary predicate (1 argument): **Human(x)** → "x is a human"
  - Binary predicate (2 arguments): **Loves(John, Mary)** → "John loves Mary"
  - Ternary predicate (3 arguments): **Between(x, y, z)** → "x is between y and z"

First Order Logic (FOL) or First Order Predicate Logic (FOPL) in AI

4. Functions

- Map objects from the domain to another object.
- Think of them like mathematical or descriptive functions.
- **Example:**
  - **Father(John)** → returns "the father of John"
  - **Square(5)** = 25
  - Sentence: **Human(Father(John))** → “The father of John is a human.”

5. Connectives (Logical Operators)

- Used to connect sentences.
- Symbols: ∧ (and), ∨ (or), ¬ (not), → (implies), ⇔ (if and only if)
- **Example:**
  - **Human(John) ∧ Mortal(John)** → “John is a human AND John is mortal.”
  - **Loves(John, Mary) → Happy(John)** → “If John loves Mary, then John is happy.”

First Order Logic (FOL) or First Order Predicate Logic (FOPL) in AI

6. Equality (=)

- States that two terms are equal.
- **Example:**
  - **Father(John) = David** → "The father of John is David."
  - **2 + 3 = 5**

7. Quantifiers

- Describe the scope of variables in the domain.
- **Universal Quantifier (∀):** "For all"
  - Example: **∀x (Human(x) → Mortal(x))**
  - Meaning: "For all x, if x is a human, then x is mortal."
- **Existential Quantifier (∃):** "There exists"
  - Example: **∃x Loves(x, Mary)**
  - Meaning: "There exists at least one x such that x loves Mary."

First Order Logic (FOL) or First Order Predicate Logic (FOPL) in AI

Atomic Sentence

- The most basic FOL sentence.
- Formed by a **predicate symbol + arguments**.
- Cannot be broken further.
- **Example:**
  - **Human(John)** → "John is a human."
  - **GreaterThan(5, 3)** → "5 is greater than 3."

Sequence of Terms

- We can represent atomic sentence as: **Predicate($term_1$, $term_2$, …, $term_n$)**

Examples:

Sentence	FOL Representation	Predicate	Terms
Chonky is a cat.	Cat(Chonky)	Cat	Chonky
Ravi and Ajay are brothers.	Brother(Ravi, Ajay)	Brother	Ravi, Ajay
John likes cricket.	Like(John, Cricket)	Like	John, Cricket
Ram is tall.	Tall(Ram)	Tall	Ram

First Order Logic (FOL) or First Order Predicate Logic (FOPL) in AI

Complex Sentence

- A **complex statement** is made by combining two or more **atomic sentences** using **logical connectives** like AND (∧), OR (∨), NOT (¬), IMPLIES (→), and IFF (⇔).

Type of Connective	Atomic Sentences	Complex Statement (FOL)	Meaning in English
Conjunction (AND, ∧)	Human(Socrates), Mortal(Socrates)	Human(Socrates) ∧ Mortal(Socrates)	Socrates is a human and Socrates is mortal.
Disjunction (OR, ∨)	Likes(John, Pizza), Likes(John, Burger)	Likes(John, Pizza) ∨ Likes(John, Burger)	John likes pizza or John likes burger.
Implication (IF...THEN, →)	Rainy(Today), Wet(Road)	Rainy(Today) → Wet(Road)	If it is raining today, then the road will be wet.
Biconditional (IFF, ⇔)	Study(John), Pass(John)	Study(John) ⇔ Pass(John)	John will pass if and only if he studies.
Negation (NOT, ¬)	Happy(John)	¬Happy(John)	John is not happy.

First Order Logic (FOL) or First Order Predicate Logic (FOPL) in AI

Quantifiers in FOPL

- A quantifier is a language element which generates quantification, and quantification specifies the quantity of specimen in the universe of discourse.
- These are the symbols that permit to denote and identify the range and scope of the variables in the logical expression.
- There are 2 types of quantifier, i.e.:
  - **Universal Quantifier (∀):**
    - for all, everyone, everything
  - **Existential Quantifier (∃):**
    - for some, at least one

First Order Logic (FOL) or First Order Predicate Logic (FOPL) in AI

Universal Quantifier

- It is a symbol of logical representation which specifies that within its range is true for everything.
- Universal Quantifier is represented by a symbol **∀**.
- If x is a variable, then ∀x is read as:
  - for all x
  - for each x
  - for every x

Examples (Universal Quantifier):

English Sentence	FOL Representation	Explanation
All men drink coffee.	∀x (Man(x) → Drink(x, Coffee))	For every x, if x is a man, then x drinks coffee.
All birds fly.	∀x (Bird(x) → Fly(x))	For every x, if x is a bird, then x can fly.
Every man respects his parents.	∀x (Man(x) → Respect(x, Parents))	For every x, if x is a man, then x respects his parents.

First Order Logic (FOL) or First Order Predicate Logic (FOPL) in AI

Existential Quantifier

- These are the type of quantifier which express that the statement within its scope is true for **at least one instance** of something. Existential quantifier is represented by the symbol **∃**.
- If x is a variable, then Existential quantifier (∃x) will be read as:
  - there exists an x
  - for some x
  - for at least one x

Examples (Existential Quantifier):

English Sentence	FOL Representation	Explanation
Some boys are intelligent.	∃x (Boy(x) ∧ Intelligent(x))	There exists at least one x such that x is a boy and x is intelligent.
Some boys play cricket.	∃x (Boy(x) ∧ Play(x, Cricket))	There exists at least one x such that x is a boy and x plays cricket.
Not all students like both mathematics and science.	∀x (Student(x) → ¬(Like(x, Mathematics) ∧ Like(x, Science)))	For every x, if x is a student, then it is not true that x likes both mathematics and science.

First Order Logic (FOL) or First Order Predicate Logic (FOPL) in AI

Exercise:

1. All humans are mortal.

$\forall x\,[Human(x) \rightarrow Mortal(x)]$

(For every x, if x is a human then x is mortal.)

2. Some teachers are researchers.

$\exists x\,[Teacher(x) \wedge Researcher(x)]$

(There exists at least one x who is both a teacher and a researcher.)

3. Every student has submitted at least one assignment.

Let Student(x) and Submitted(x, y) mean "x submitted assignment y". Then:

$\forall x\,[Student(x) \rightarrow \exists y\ Submitted(x, y)]$

(For every person x, if x is a student then there exists an assignment y such that x submitted y.)

4. Some students are not honest.

$\exists x[Student(x) \wedge \neg Honest(x)]$.

(There exists a student who is not honest.)

First Order Logic (FOL) or First Order Predicate Logic (FOPL) in AI

Exercise:

5. All birds can fly.

$\forall x\,[Birds(x) \rightarrow Fly(x)]$

(For every x, if x is a bird then x can fly.)

6. Not all birds can fly.

$\neg\,\forall x\,[Birds(x) \rightarrow Fly(x)] \cong \exists x[Bird(x) \wedge \neg Fly(x)]$.

(There exists at least one bird that cannot fly.)

7. Every professor teaches at least one course:

Let Professor(p), Teaches(p,c), constant AI, ML denote course names (or course predicates). Two parts:

$\forall p\,[Professor(p) \rightarrow \exists c\ Teaches(p, c)]$

8. Some professor teaches both AI and ML.

$\exists p[Professor(p) \wedge Teaches(p, AI) \wedge Teaches(p,ML)]$

(There exists some teacher who teaches both AI & ML.)

First Order Logic (FOL) or First Order Predicate Logic (FOPL) in AI

Statement in English	Solution (FOL)	Notes / Explanation
Mary loves everyone. (Assuming domain = all humans)	$\forall x$ love(Mary, x)	Simple universal quantifier, Mary loves each person.
Mary loves everyone. (If domain includes non-humans, and we only want "everyone who is a person")	$\forall x$ (person(x) → love(Mary, x))	Adds a condition restricting to persons.
No one talks.	$\neg\exists x$ talk(x) or $\forall x \neg$talk(x)	Two logically equivalent alternatives.
Everyone loves themselves.	$\forall x$ love(x, x)	Reflexive relation.
Everyone loves everyone.	$\forall x \forall y$ love(x, y)	Double universal.
Everyone loves everyone else (except themselves).	$\forall x \forall y$ ($x \neq y$ → love(x, y))	Adds inequality to avoid self-love.