UNIT-1

INTRODUCTION

WHAT IS AI :

Since the invention of computers or machines, their capability to perform various tasks went on growing exponentially. Humans have developed the power of computer systems in terms of their diverse working domains, their increasing speed, and reducing size with respect to time. *Artificial Intelligence* pursues creating the computers or machines as intelligent as human beings.

In today's world, technology is growing very fast, and we are getting in touch with different new technologies day by day.

Here, one of the booming technologies of computer science is Artificial Intelligence which is ready to create a new revolution in the world by making intelligent machines.The Artificial Intelligence is now all around us. It is currently working with a variety of subfields, ranging from general to specific, such as self-driving cars, playing chess, proving theorems, playing music, Painting, etc.

According to the father of Artificial Intelligence, John McCarthy, it is *"The science and engineering of making intelligent machines, especially intelligent computer programs".*

Artificial Intelligence is a way of making a computer, a computer-controlled robot, or a software think intelligently, in the similar manner the intelligent humans think.

So, we can define AI as: "*It is a branch of computer science by which we can create intelligent machines which can behave like a human, think like humans, and able to make decisions*."

Artificial Intelligence exists when a machine can have human based skills such as learning, reasoning, and solving problems

With Artificial Intelligence you do not need to preprogram a machine to do some work, despite that you can create a machine with programmed algorithms which can work with own intelligence, and that is the awesomeness of AI.

It is believed that AI is not a new technology, and some people says that as per Greek myth, there were Mechanical men in early days which can work and behave like humans.

Why Artificial Intelligence?

Before Learning about Artificial Intelligence, we should know that what is the importance of AI and why should we learn it. Following are some main reasons to learn about AI:

1) With the help of AI, you can create such software or devices which can solve real-world problems very easily and with accuracy such as health issues, marketing, traffic issues, etc.

2) With the help of AI, you can create your personal virtual Assistant, such as Cortana, Google Assistant, Siri, etc.

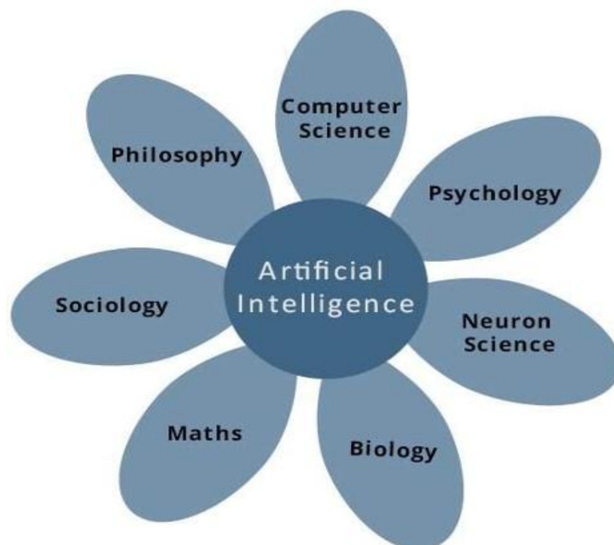
3) With the help of AI, you can build such Robots which can work in an environment where survival of humans can be at risk.

3) AI opens a path for other new technologies, new devices, and new Opportunities.

FOUNDATION OF AI:

Artificial intelligence is a science and technology based on disciplines such as Computer Science, Biology, Psychology, Linguistics, Mathematics, and Engineering. A major thrust of AI is in the development of computer functions associated with human intelligence, such as reasoning, learning, and problem solving.

Out of the following areas, one or multiple areas can contribute to build an intelligent system.

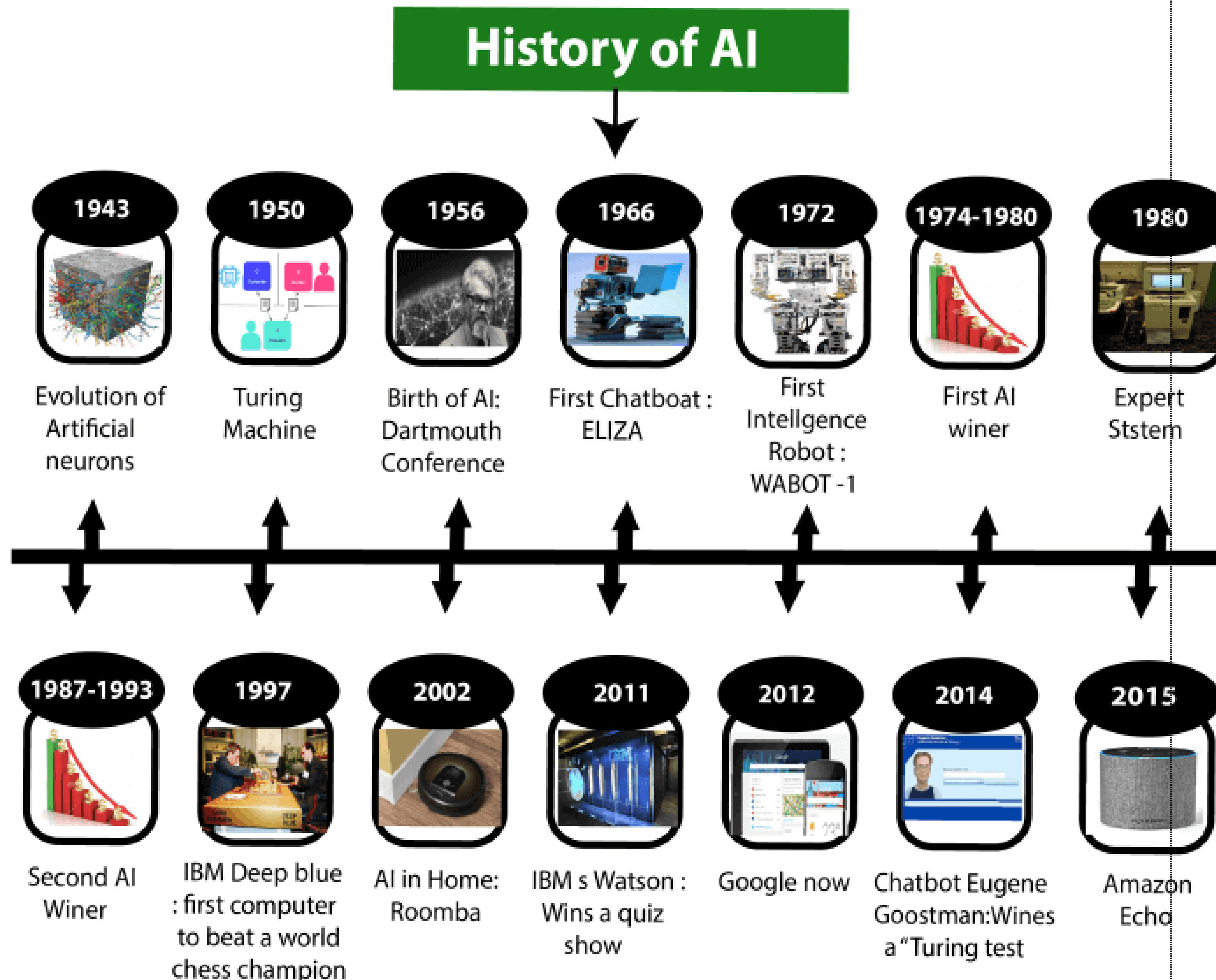


Discipline	**Role in AI**
Computer Science	Programming, algorithms, data structures
Biology	Neural inspiration, adaptation, evolution
Psychology	Cognitive modeling, behavior simulation
Linguistics	Language understanding and processing
Mathematics	Formal models, logic, probability
Engineering	Designing physical machines and systems

HISTORY OF AI :

Artificial Intelligence is not a new word and not a new technology for researchers. This technology is much older than you would imagine. Even there are the myths of

Mechanical men in Ancient Greek and Egyptian Myths. Following are some milestones in the history of AI which defines the journey from the AI generation to till date development.



Maturation of Artificial Intelligence (1943-1952)

- **Year 1943:** The first work which is now recognized as AI was done by Warren McCulloch and Walter pits in 1943. They proposed a model of **artificial neurons**.
- **Year 1949:** Donald Hebb demonstrated an updating rule for modifying the connection strength between neurons. His rule is now called **Hebbian learning**.
- **Year 1950:** The Alan Turing who was an English mathematician and pioneered Machine learning in 1950. Alan Turing publishes **"Computing Machinery and Intelligence"** in which he proposed a test. The test can check the machine's ability to exhibit intelligent behavior equivalent to human intelligence, called a **Turing test**.

The birth of Artificial Intelligence (1952-1956)

- **Year 1955:** An Allen Newell and Herbert A. Simon created the "first artificial intelligence program"Which was named as **"Logic Theorist"**. This program had proved 38 of 52 Mathematics theorems, and find new and more elegant proofs for some theorems.
- **Year 1956:** The word "Artificial Intelligence" first adopted by American Computer scientist John McCarthy at the Dartmouth Conference. For the first time, AI coined as an academic field.

At that time high-level computer languages such as FORTRAN, LISP, or COBOL were invented. And the enthusiasm for AI was very high at that time.

The golden years-Early enthusiasm (1956-1974)

- **Year 1966:** The researchers emphasized developing algorithms which can solve mathematical problems. Joseph Weizenbaum created the first chatbot in 1966, which was named as ELIZA.
- **Year 1972:** The first intelligent humanoid robot was built in Japan which was named as WABOT-1.

The first AI winter (1974-1980)

- The duration between years 1974 to 1980 was the first AI winter duration. AI winter refers to the time period where computer scientist dealt with a severe shortage of funding from government for AI researches.
- During AI winters, an interest of publicity on artificial intelligence was decreased.

A boom of AI (1980-1987)

- **Year 1980:** After AI winter duration, AI came back with "Expert System". Expert systems were programmed that emulate the decision-making ability of a human expert.
- In the Year 1980, the first national conference of the American Association of Artificial Intelligence **was held at Stanford University**.

The second AI winter (1987-1993)

- The duration between the years 1987 to 1993 was the second AI Winter duration.

- Again Investors and government stopped in funding for AI research as due to high cost but not efficient result. The expert system such as XCON was very cost effective.

The emergence of intelligent agents (1993-2011)

- **Year 1997:** In the year 1997, IBM Deep Blue beats world chess champion, Gary Kasparov, and became the first computer to beat a world chess champion.
- **Year 2002:** for the first time, AI entered the home in the form of Roomba, a vacuum cleaner.
- **Year 2006:** AI came in the Business world till the year 2006. Companies like Facebook, Twitter, and Netflix also started using AI.

Deep learning, big data and artificial general intelligence (2011-present)

- **Year 2011:** In the year 2011, IBM's Watson won jeopardy, a quiz show, where it had to solve the complex questions as well as riddles. Watson had proved that it could understand natural language and can solve tricky questions quickly.
- **Year 2012:** Google has launched an Android app feature "Google now", which was able to provide information to the user as a prediction.
- **Year 2014:** In the year 2014, Chatbot "Eugene Goostman" won a competition in the infamous "Turing test."
- **Year 2018:** The "Project Debater" from IBM debated on complex topics with two master debaters and also performed extremely well.
- Google has demonstrated an AI program "Duplex" which was a virtual assistant and which had taken hairdresser appointment on call, and lady on other side didn't notice that she was talking with the machine.

Now AI has developed to a remarkable level. The concept of Deep learning, big data, and data science are now trending like a boom. Nowadays companies like Google, Facebook, IBM, and Amazon are working with AI and creating amazing devices. The future of Artificial Intelligence is inspiring and will come with high intelligence.

GOALS OF ARTIFICIAL INTELLIGENCE:

Following are the main goals of Artificial Intelligence:

1. Replicate human intelligence
2. Solve Knowledge-intensive tasks
3. An intelligent connection of perception and action

4. Building a machine which can perform tasks that requires human intelligence such as:

- Proving a theorem
- Playing chess
- Plan some surgical operation
- Driving a car in traffic

5. Creating some system which can exhibit intelligent behavior, learn new things by itself, demonstrate, explain, and can advise to its user.

ADVANTAGES OF ARTIFICIAL INTELLIGENCE:

Artificial Intelligence offers numerous benefits across various domains due to its ability to simulate human intelligence and automate complex tasks. Some of the key advantages are:

1. **High Accuracy and Reduced Errors**
   AI systems operate based on stored data and algorithms, which reduces the possibility of human errors. This leads to highly accurate results, especially in areas like medical diagnosis, manufacturing, and data analysis.
2. **High-Speed Processing**
   AI-powered machines can process vast volumes of data and perform computations at speeds much faster than humans. This is particularly useful in tasks requiring real-time decision-making, such as fraud detection or autonomous driving.
3. **High Reliability and Consistency**
   Unlike humans, AI systems do not suffer from fatigue and can perform repetitive tasks continuously without compromising accuracy. This makes them highly reliable for continuous operations.
4. **Use in Dangerous Environments**
   AI can be effectively used in situations that are hazardous for humans, such as bomb disposal, space exploration, deep-sea missions, or nuclear facility inspections. These systems reduce human risk in life-threatening scenarios.
5. **Digital Assistance and Automation**
   AI provides personalized digital assistants that can interact with users, schedule tasks, and respond to queries. Examples include voice assistants like Siri, Alexa, and Google Assistant, as well as AI chat bots in customer service.
6. **Applications in Public Utilities**
   AI finds applications in everyday life such as self-driving vehicles, facial recognition systems, automatic translators, and smart home devices, making life safer, more efficient, and convenient.

DISADVANTAGES OF ARTIFICIAL INTELLIGENCE:

Despite its many advantages, AI also comes with limitations and challenges that must be considered:

1. **High Development and Maintenance Cost**
   The creation and maintenance of AI systems require significant investment in terms of hardware, software, and expertise. Updating these systems to adapt to real-world changes adds to the cost.

2. **Lack of Creativity and Originality**
   AI systems operate within the boundaries of their training data and programming. They are incapable of generating genuinely new ideas or exhibiting creative thinking like humans.
3. **Absence of Emotions and Human Touch**
   AI lacks the ability to feel emotions or understand the emotional needs of humans. This makes AI unsuitable for jobs that require human empathy, compassion, or emotional intelligence, such as counseling or care giving.
4. **Dependency on Machines**
   Increased reliance on AI for basic and advanced tasks can make humans overly dependent on technology, resulting in reduced use of critical thinking and problem-solving skills.
5. **Limited Adaptability and Common Sense**
   AI systems perform well within defined tasks but often fail to adapt to unexpected situations or handle problems requiring common sense reasoning. They cannot think “out of the box” or act beyond their programming.

THE STATE OF ART :

In a world in which humanity exercises total power over machines, super robots must respect three fundamental laws:

1. A robot cannot injure a human being or allow a human being to be harmed by remaining inert;
2. A robot must obey an order given by a human, as long as it does not conflict with the first rule;
3. A robot must protect its existence without conflicting with the other two rules.

We can start with the certainty that AI is not just about robots, as movies often show us. Artificial intelligence is in a search algorithm, in a chatbot, in IoT devices, in cars, and even in autonomous weapons, such as missiles that follow their target.

Today, we can identify two main trends in AI:

- **Narrow AI** (or weak artificial intelligence), which deals with single tasks by integrating only one part of the mind;
- **General AI** (AGI or strong artificial intelligence), which pertains to the ability of a machine to understand or learn any intellectual task that a human being can perform.

The increasingly advanced technology provides researchers with new tools that are capable of achieving important goals, and these tools are great starting points in and of themselves. Among the achievements of recent years, the following are some specific domains:

- **Machine learning;**
- **Reinforcement learning;**
- **Deep learning;**
- **Natural language processing.**

INTELLIGENT AGENTS

AGENTS AND ENVIRONMENTS:

An AI system can be defined as the study of the rational agent and its environment. The agents sense the environment through sensors and act on their environment through actuators. An AI agent can have mental properties such as knowledge, belief, intention, etc.

What is an Agent?

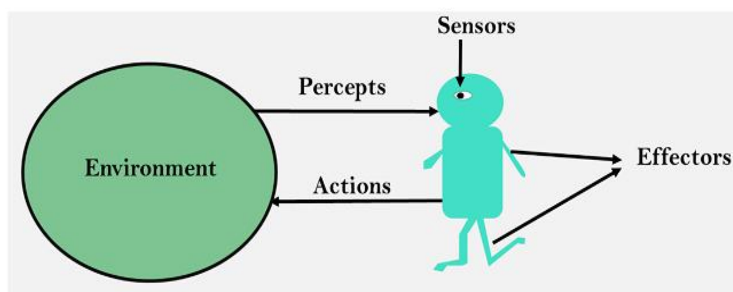
An agent can be anything that perceives environment through sensors and act upon that environment through actuators. An Agent runs in the cycle of **perceiving**, **thinking**, and **acting**. An agent can be:

- **Human-Agent:** A human agent has eyes, ears, and other organs which work for sensors and hand, legs, vocal tract work for actuators.
- **Robotic Agent:** A robotic agent can have cameras, infrared range finder, NLP for sensors and various motors for actuators.
- **Software Agent:** Software agent can have keystrokes, file contents as sensory input and act on those inputs and display output on the screen.

Sensor: Sensor is a device which detects the change in the environment and sends the information to other electronic devices. An agent observes its environment through sensors.

Actuators: Actuators are the component of machines that converts energy into motion. The actuators are only responsible for moving and controlling a system. An actuator can be an electric motor, gears, rails, etc.

Effectors: Effectors are the devices which affect the environment. Effectors can be legs, wheels, arms, fingers, wings, fins, and display screen.



TYPES OF AGENTS :

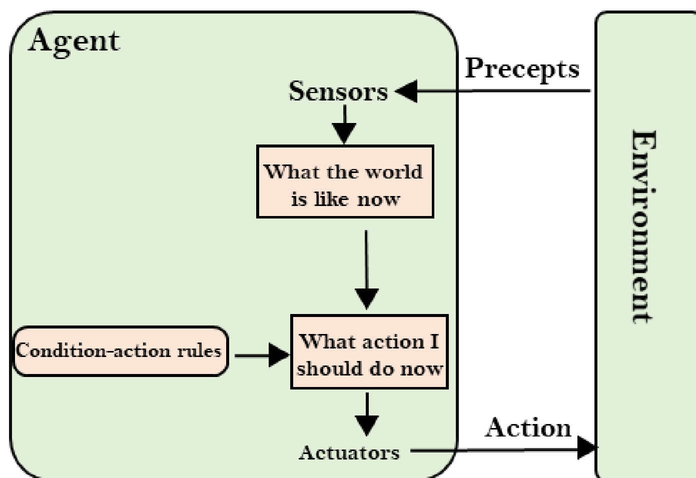
Agents can be grouped into five classes based on their degree of perceived intelligence and capability. All these agents can improve their performance and generate better action over the time. These are given below:

1. Simple Reflex Agent

2. Model-based reflex agent
3. Goal-based agents
4. Utility-based agent
5. Learning agent

1. Simple Reflex agent:

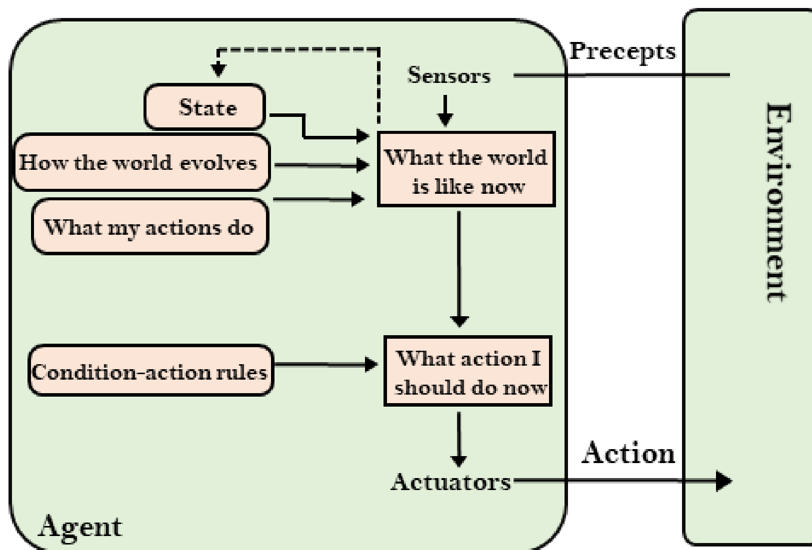
- The Simple reflex agents are the simplest agents. These agents take decisions on the basis of the current percepts and ignore the rest of the percept history.
- These agents only succeed in the fully observable environment.
- The Simple reflex agent does not consider any part of percepts history during their decision and action process.
- The Simple reflex agent works on Condition-action rule, which means it maps the current state to action. Such as a Room Cleaner agent, it works only if there is dirt in the room.
- Problems for the simple reflex agent design approach:
    - They have very limited intelligence
    - They do not have knowledge of non-perceptual parts of the current state
    - Mostly too big to generate and to store.
    - Not adaptive to changes in the environment.



2. Model-based reflex agent

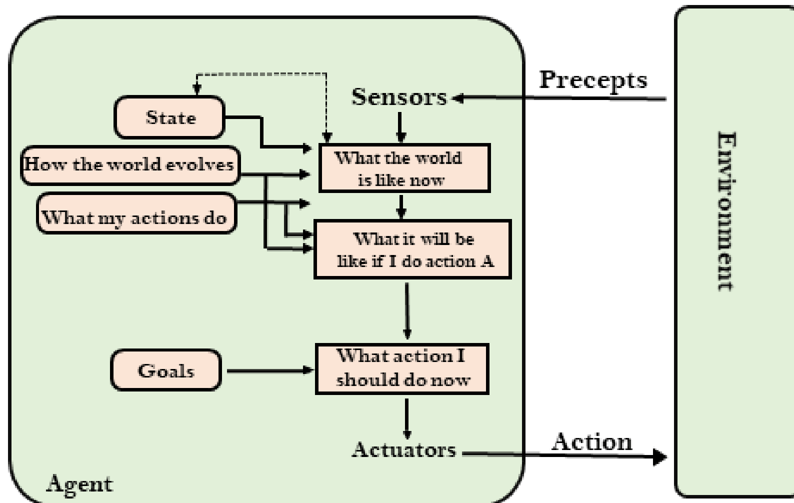
- The Model-based agent can work in a partially observable environment, and track the situation.
- A model-based agent has two important factors:
    - **Model:** It is knowledge about "how things happen in the world," so it is called a Model-based agent.
    - **Internal State:** It is a representation of the current state based on percept history.
- These agents have the model, "which is knowledge of the world" and based on the model they perform actions.

- Updating the agent state requires information about:
  a. How the world evolves
  b. How the agent's action affects the world.



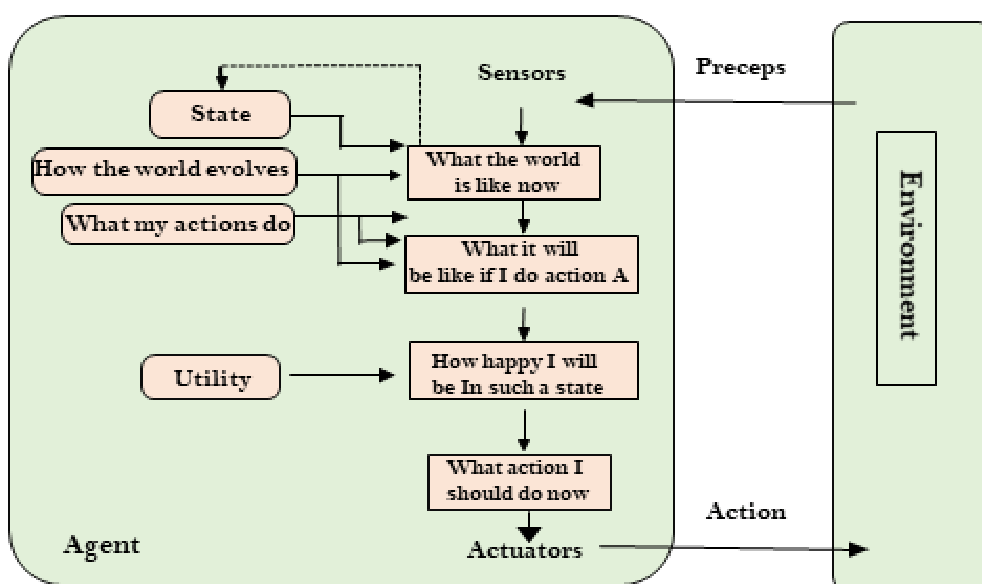
3. Goal-based agents

- The knowledge of the current state environment is not always sufficient to decide for an agent to what to do.
- The agent needs to know its goal which describes desirable situations.
- Goal-based agents expand the capabilities of the model-based agent by having the "goal" information.
- They choose an action, so that they can achieve the goal.
- These agents may have to consider a long sequence of possible actions before deciding whether the goal is achieved or not. Such considerations of different scenario are called searching and planning, which makes an agent proactive.



4. Utility-based agents

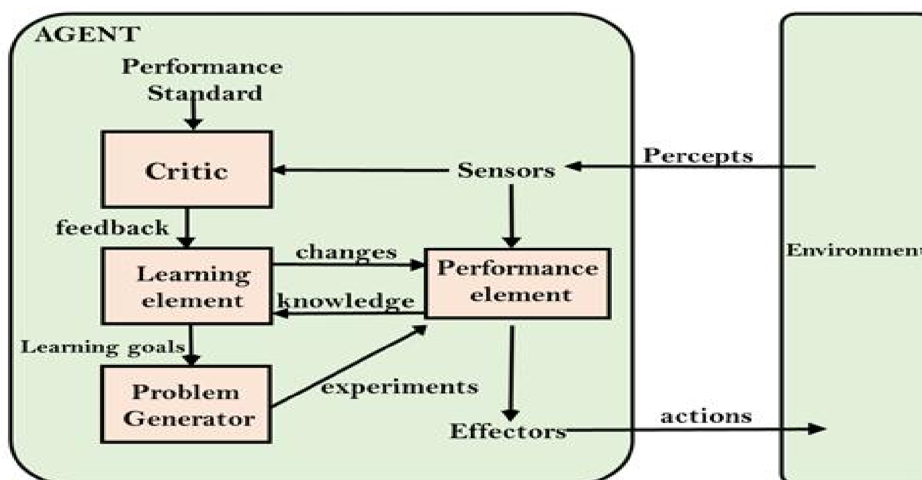
- These agents are similar to the goal-based agent but provide an extra component of utility measurement which makes them different by providing a measure of success at a given state.
- Utility-based agent act based not only goals but also the best way to achieve the goal.
- The Utility-based agent is useful when there are multiple possible alternatives, and an agent has to choose in order to perform the best action.
- The utility function maps each state to a real number to check how efficiently each action achieves the goals.



5. Learning Agents

- A learning agent in AI is the type of agent which can learn from its past experiences, or it has learning capabilities.
- It starts to act with basic knowledge and then able to act and adapt automatically through learning.
- A learning agent has mainly four conceptual components, which are:
  a. **Learning element:** It is responsible for making improvements by learning from environment
  b. **Critic:** Learning element takes feedback from critic which describes that how well the agent is doing with respect to a fixed performance standard.
  c. **Performance element:** It is responsible for selecting external action
  d. **Problem generator:** This component is responsible for suggesting actions that will lead to new and informative experiences.

Hence, learning agents are able to learn, analyze performance, and look for new ways to improve the performance.



Intelligent Agents:

An intelligent agent is an autonomous entity which act upon an environment using sensors and actuators for achieving goals. An intelligent agent may learn from the environment to achieve their goals. A thermostat is an example of an intelligent agent.

Following are the main four rules for an AI agent:

- **Rule 1:** An AI agent must have the ability to perceive the environment.
- **Rule 2:** The observation must be used to make decisions.
- **Rule 3:** Decision should result in an action.
- **Rule 4:** The action taken by an AI agent must be a rational action.

CONCEPT OF RATIONALITY:

The rationality of an agent is measured by its performance measure. Rationality can be judged on the basis of following points:

- Performance measure which defines the success criterion.
- Agent prior knowledge of its environment.
- Best possible actions that an agent can perform.
- The sequence of percepts.

Rational Agent:

- A rational agent is an agent which has clear preference, models uncertainty, and acts in a way to maximize its performance measure with all possible actions.
- A rational agent is said to perform the right things. AI is about creating rational agents to use for game theory and decision theory for various real-world scenarios.
- For an AI agent, the rational action is most important because in AI reinforcement learning algorithm, for each best possible action, agent gets the positive reward and for each wrong action, an agent gets a negative reward.

A rational agent always performs right action, where the right action means the action that causes the agent to be most successful in the given percept sequence. The problem the agent solves is characterized by Performance Measure, Environment, Actuators, and Sensors (PEAS).

> Note: Rational agents in AI are very similar to intelligent agents.

<u>THE NATURE OF ENVIRONMENT:</u>

An environment is everything in the world which surrounds the agent, but it is not a part of an agent itself. An environment can be described as a situation in which an agent is present.The environment is where agent lives, operate and provide the agent with something to sense and act upon it. An environment is mostly said to be non-feministic.

Features of Environment

As per Russell and Norvig, an environment can have various features from the point of view of an agent:

1. Fully observable vs Partially Observable
2. Static vs Dynamic
3. Discrete vs Continuous
4. Deterministic vs Stochastic
5. Single-agent vs Multi-agent
6. Episodic vs sequential
7. Known vs Unknown
8. Accessible vs Inaccessible

1. Fully observable vs Partially Observable:

- If an agent sensor can sense or access the complete state of an environment at each point of time then it is **a fully observable** environment, else it is **partially observable**.
- A fully observable environment is easy as there is no need to maintain the internal state to keep track history of the world.
- An agent with no sensors in all environments then such an environment is called as **unobservable**.

2. Deterministic vs Stochastic:

- If an agent's current state and selected action can completely determine the next state of the environment, then such environment is called a deterministic environment.
- A stochastic environment is random in nature and cannot be determined completely by an agent.
- In a deterministic, fully observable environment, agent does not need to worry about uncertainty.

3. Episodic vs Sequential:

- In an episodic environment, there is a series of one-shot actions, and only the current percept is required for the action.
- However, in Sequential environment, an agent requires memory of past actions to determine the next best actions.

4. Single-agent vs Multi-agent

- If only one agent is involved in an environment, and operating by itself then such an environment is called single agent environment.
- However, if multiple agents are operating in an environment, then such an environment is called a multi-agent environment.
- The agent design problems in the multi-agent environment are different from single agent environment.

5. Static vs Dynamic:

- If the environment can change itself while an agent is deliberating then such environment is called a dynamic environment else it is called a static environment.
- Static environments are easy to deal because an agent does not need to continue looking at the world while deciding for an action.
- However for dynamic environment, agents need to keep looking at the world at each action.
- Taxi driving is an example of a dynamic environment whereas Crossword puzzles are an example of a static environment.

6. Discrete vs Continuous:

- If in an environment there are a finite number of percepts and actions that can be performed within it, then such an environment is called a discrete environment else it is called continuous environment.
- A chess gamecomes under discrete environment as there is a finite number of moves that can be performed.
- A self-driving car is an example of a continuous environment.

7. Known vs Unknown

- Known and unknown are not actually a feature of an environment, but it is an agent's state of knowledge to perform an action.
- In a known environment, the results for all actions are known to the agent. While in unknown environment, agent needs to learn how it works in order to perform an action.
- It is quite possible that a known environment to be partially observable and an Unknown environment to be fully observable.

8. Accessible vs Inaccessible

- If an agent can obtain complete and accurate information about the state's environment, then such an environment is called an Accessible environment else it is called inaccessible.
- An empty room whose state can be defined by its temperature is an example of an accessible environment.
- Information about an event on earth is an example of Inaccessible environment.

THE STRUCTURE OF AGENTS :

The task of AI is to design an agent program which implements the agent function. The structure of an intelligent agent is a combination of architecture and agent program. It can be viewed as:

Agent = Architecture + Agent program

Following are the main three terms involved in the structure of an AI agent:

Architecture: Architecture is machinery that an AI agent executes on.

Agent Function: Agent function is used to map a percept to an action.

$f: P^* \rightarrow A$

Agent program: Agent program is an implementation of agent function. An agent program executes on the physical architecture to produce function f.

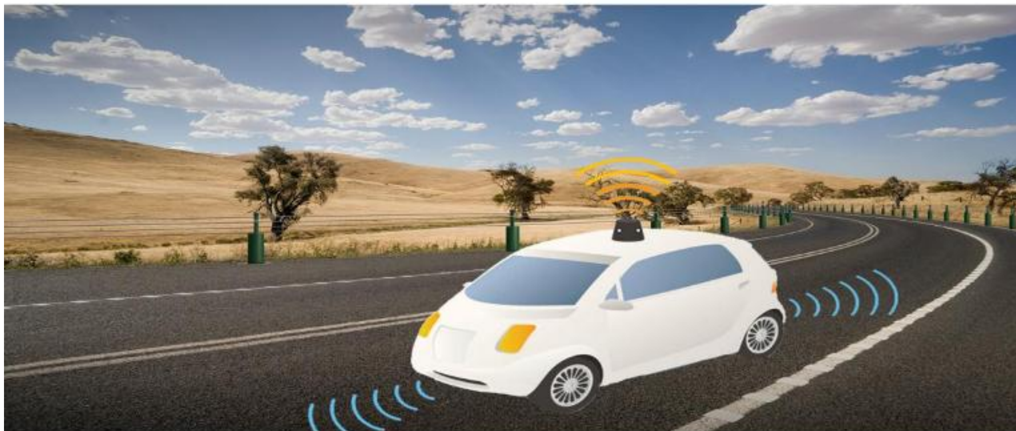
PEAS Representation

PEAS is a type of model on which an AI agent works upon. When we define an AI agent or rational agent, then we can group its properties under PEAS representation model. It is made up of four words:

- **P:** Performance measure
- **E:** Environment
- **A:** Actuators
- **S:** Sensors

Here performance measure is the objective for the success of an agent's behavior.

PEAS for self-driving cars:



Let's suppose a self-driving car then PEAS representation will be:

Performance: Safety, time, legal drive, comfort

Environment: Roads, other vehicles, road signs, pedestrian

Actuators: Steering, accelerator, brake, signal, horn

Sensors: Camera, GPS, speedometer, odometer, accelerometer, sonar.

Example of Agents with their PEAS representation

Agent	Performance measure	Environment	Actuators	Sensors
1. Medical Diagnose	○ Healthy patient ○ Minimized cost	○ Patient ○ Hospital ○ Staff	○ Tests ○ Treatments	Keyboard (Entry of symptoms)
2. Vacuum Cleaner	○ Cleanness ○ Efficiency ○ Battery life ○ Security	○ Room ○ Table ○ Wood floor ○ Carpet ○ Various obstacles	○ Wheels ○ Brushes ○ Vacuum Extractor	○ Camera ○ Dirt detection sensor ○ Cliff sensor ○ Bump Sensor ○ Infrared Wall Sensor
3. Part - picking Robot	○ Percentage of parts in correct bins.	○ Conveyor belt with parts, ○ Bins	○ Jointed Arms ○ Hand	○ Camera ○ Joint angle sensors.

PROBLEM-SOLVING AGENTS AND PROBLEM FORMULATION:

In Artificial Intelligence, Search techniques are universal problem-solving methods. **Rational agents** or **Problem-solving agents** in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result. Problem-solving agents are the goal-based agents and use atomic representation. In this topic, we will learn various problem-solving search algorithms.

The problem-solving agent perfoms precisely by defining problems and its several solutions. According to psychology, "*a problem-solving refers to a state where we wish to reach to a definite goal from a present state or condition.*"

Therefore, a problem-solving agent is a **goal-driven agent** and focuses on satisfying the goal.

Steps performed by Problem-solving agent are:

- **Goal Formulation:** It is the first and simplest step in problem-solving. It organizes the steps/sequence required to formulate one goal out of multiple goals as well as actions to achieve that goal. Goal formulation is based on the current situation and the agent's performance measure (discussed below).
- **Problem Formulation:** It is the most important step of problem-solving which decides what actions should be taken to achieve the formulated goal. There are following five components involved in problem formulation:

  Initial State: It is the starting state or initial step of the agent towards its goal.

  Actions: It is the description of the possible actions available to the agent.

  Transition Model: It describes what each action does.

  Goal Test: It determines if the given state is a goal state.

  Path cost: It assigns a numeric cost to each path that follows the goal. The problem-solving agent selects a cost function, which reflects its performance measure. Remember, **an optimal solution has the lowest path cost among all the solutions.**

Note: Initial state, actions, and **transition model** together define the **state-space** of the problem implicitly. State-space of a problem is a set of all states which can be reached from the initial state followed by any sequence of actions.

The state-space forms a directed map or graph where nodes are the states, links between the nodes are actions, and the path is a sequence of states connected by the sequence of actions.

- **Search:** It identifies all the best possible sequence of actions to reach the goal state from the current state. It takes a problem as an input and returns solution as its output.
- **Solution:** It finds the best algorithm out of various algorithms, which may be proven as the best optimal solution.
- **Execution:** It executes the best optimal solution from the searching algorithms to reach the goal state from the current state.

Properties of Search Algorithms:

Following are the four essential properties of search algorithms to compare the efficiency of these algorithms:

Completeness: A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.

Optimality: If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.

Time Complexity: Time complexity is a measure of time for an algorithm to complete its task.

Space Complexity: It is the maximum storage space required at any point during the search, as the complexity of the problem.

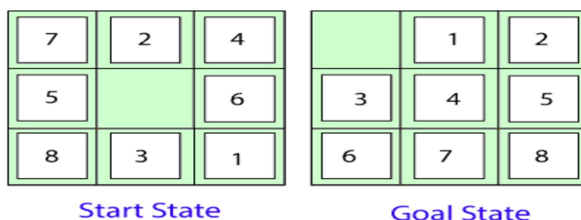
Example Problems

Basically, there are two types of problem approaches:

- **Toy Problem:** It is a concise and exact description of the problem which is used by the researchers to compare the performance of algorithms.
- **Real-world Problem:** It is real-world based problems which require solutions. Unlike a toy problem, it does not depend on descriptions, but we can have a general formulation of the problem.

Some Toy Problems

- **8 Puzzle Problem:** Here, we have a 3×3 matrix with movable tiles numbered from 1 to 8 with a blank space. The tile adjacent to the blank space can slide into that space. The objective is to reach a specified goal state similar to the goal state, as shown in the below figure.
- In the figure, our task is to convert the current state into goal state by sliding digits into the blank space.



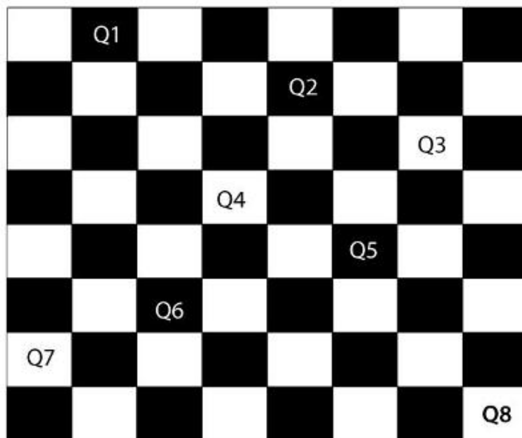
In the above figure, our task is to convert the current(Start) state into goal state by sliding digits into the blank space.

The problem formulation is as follows:

- **States:** It describes the location of each numbered tiles and the blank tile.
- **Initial State:** We can start from any state as the initial state.
- **Actions:** Here, actions of the blank space is defined, i.e., either **left, right, up or down**
- **Transition Model:** It returns the resulting state as per the given state and actions.
- **Goal test:** It identifies whether we have reached the correct goal-state.
- **Path cost:** The path cost is the number of steps in the path where the cost of each step

8-QUEENS PROBLEM*:*

The aim of this problem is to place eight queens on a chessboard in an order where no queen may attack another. A queen can attack other queens either **diagonally or in same row and column.** From the following figure, we can understand the problem as well as its correct solution.



It is noticed from the above figure that each queen is set into the chessboard in a position where no other queen is placed diagonally, in same row or column. Therefore, it is one right approach to the 8-queens problem.

For this problem, there are two main kinds of formulation:

- **Incremental formulation:** It starts from an empty state where the operator augments a queen at each step.

Following steps are involved in this formulation:

States: Arrangement of any 0 to 8 queens on the chessboard.

Initial State: An empty chessboard

Actions: Add a queen to any empty box.

Transition model: Returns the chessboard with the queen added in a box.

Goal test: Checks whether 8-queens are placed on the chessboard without any attack.

Path cost: There is no need for path cost because only final states are counted.
In this formulation, there is approximately **1.8×10^{14}** possible sequence to investigate.

- **Complete-state formulation:** It starts with all the 8-queens on the chessboard and moves them around, saving from the attacks.

Following steps are involved in this formulation

- **States:** Arrangement of all the 8 queens one per column with no queen attacking the other queen.
- **Actions:** Move the queen at the location where it is safe from the attacks.

This formulation is better than the incremental formulation as it reduces the state space from **1.8×10^{14}** to **2057**, and it is easy to find the solutions.

Some Real-world problems

- **Traveling salesperson problem(TSP):** It is a **touring problem** where the salesman can visit each city only once. The objective is to find the shortest tour and sell-out the stuff in each city.

- **VLSI Layout problem:** In this problem, millions of components and connections are positioned on a chip in order to minimize the area, circuit-delays, stray-capacitances, and maximizing the manufacturing yield.

  The layout problem is split into two parts:
- **Cell layout:** Here, the primitive components of the circuit are grouped into cells, each performing its specific function. Each cell has a fixed shape and size. The task is to place the cells on the chip without overlapping each other.
- **Channel routing:** It finds a specific route for each wire through the gaps between the cells.
- **Protein Design:** The objective is to find a sequence of amino acids which will fold into 3D protein having a property to cure some disease.

UNIT - II Searching

Searching for solutions, uniformed search strategies – Breadth first search, depth first Search. Search with partial information (Heuristic search) Hill climbing, A* ,AO* Algorithms, Problem reduction, Game Playing - Adversial search, Games, mini-max algorithm, optimal decisions in multiplayer games, Problem in Game playing, Alpha-Beta pruning, Evaluation functions.

Search: Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:

a. **Search Space:** Search space represents a set of possible solutions, which a system may have.
b. **Start State:** It is a state from where agent begins **the search**.
c. **Goal test:** It is a function which observe the current state and returns whether the goal state is achieved or not.

Search tree: A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.

Actions: It gives the description of all the available actions to the agent.

Transition model: A description of what each action do, can be represented as a transition model.

Path Cost: It is a function which assigns a numeric cost to each path.

Solution: It is an action sequence which leads from the start node to the goal node.

Optimal Solution: If a solution has the lowest cost among all solutions.

Properties of Search Algorithms:

Following are the four essential properties of search algorithms to compare the efficiency of these algorithms:

Completeness: A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.

Optimality: If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.

Time Complexity: Time complexity is a measure of time for an algorithm to complete its task.

Space Complexity: It is the maximum storage space required at any point during the search, as the complexity of the problem.

(i) **Infrastructure for search algorithms:**

▪ Search algorithms require a data structure to keep track of the search tree that is being constructed.

▪ For each node n of the tree, we have a structure that contains four components:

• n.STATE: the state in the state space to which the node corresponds;

• n.PARENT: the node in the search tree that generated this node;

• n.ACTION: the action that was applied to the parent to generate the node;

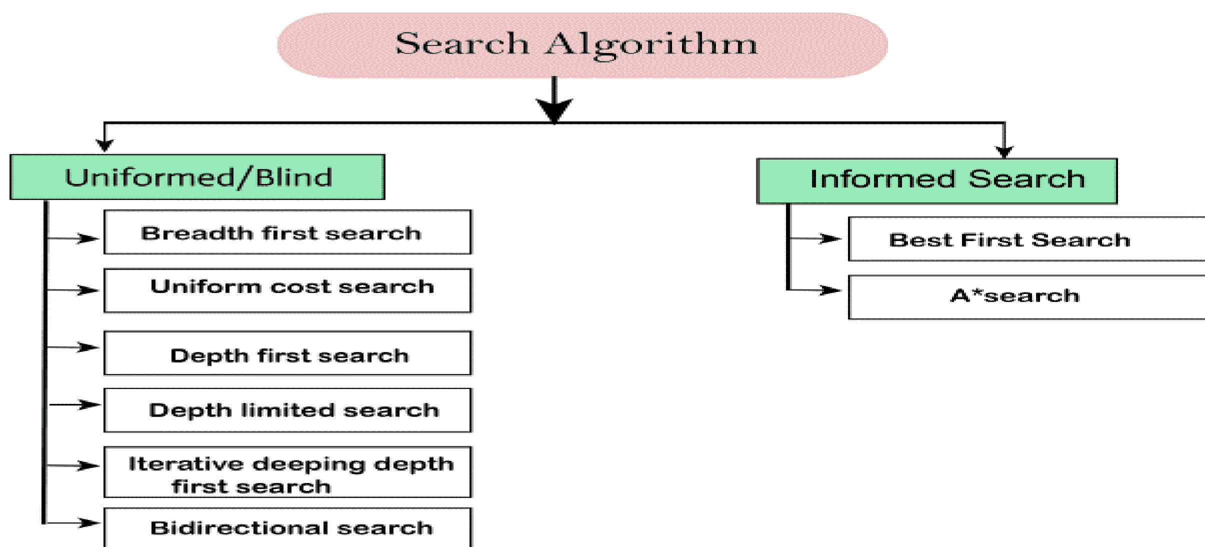
• n.PATH-COST: the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

➢Queue :
▪ Now that we have nodes, we need somewhere to put them.
▪ The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy.
▪ The appropriate data structure for this is a queue.
▪ The operations on a queue are as follows:
• EMPTY?(queue) returns true only if there are no more elements in the queue.
• POP(queue) removes the first element of the queue and returns it.
• INSERT(element, queue) inserts an element and returns the resulting queue.
▪ Queues are characterized by the order in which they store the inserted nodes.

▪ Three common variants are
▪ The first-in, first-out or FIFO queue, which pops the oldest element of the queue;
▪ LIFO QUEUE the last-in, first-out or LIFO queue (also known as a stack), which pops the newest element
▪ PRIORITY QUEUE of the queue; and the priority queue, which pops the element of the queue with the highest priority according to some ordering function.
(ii)Measuring problem-solving performance:
▪ Before we get into the design of specific search algorithms, we need to consider the criteria that might be used to choose among them.

▪ We can evaluate an algorithm's performance in four ways:
→ Completeness: Is the algorithm guaranteed to find a solution when there is one?
→ Optimality: Does the strategy find the optimal solution,
→ Time complexity: How long does it take to find a solution?
→ Space complexity: How much memory is needed to perform the search?

Types of search algorithms



Types of search algorithm

1. Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

Advantages:

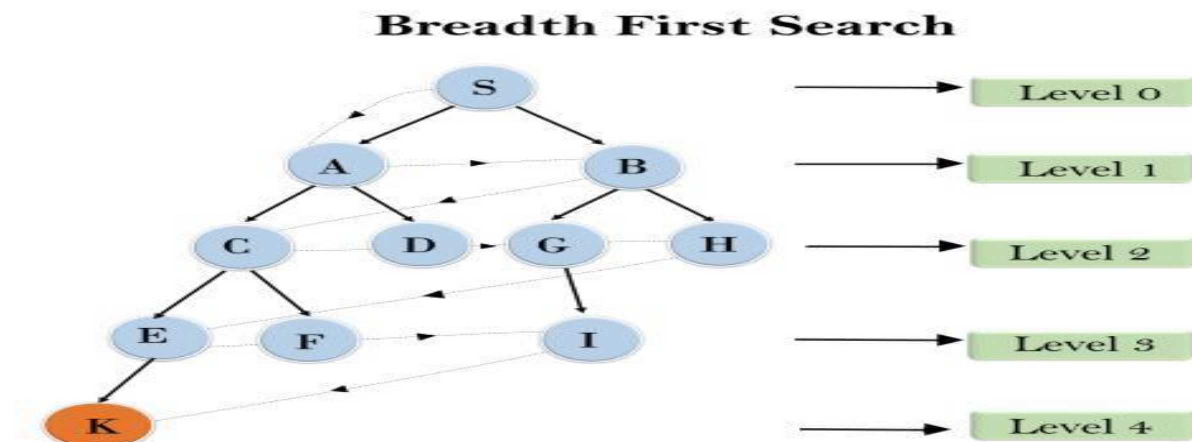
- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.
- It also helps in finding the shortest path in goal state, since it needs all nodes at the same hierarchical level before making a move to nodes at lower levels.
- It is also very easy to comprehend with the help of this we can assign the higher rank among path types.

Disadvantages:

- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.
- It can be very inefficient approach for searching through deeply layered spaces, as it needs to thoroughly explore all nodes at each level before moving on to the next

Example:
In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

1. S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K



Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

$T(b) = 1+b^2+b^3+.......+ b^d= O(b^d)$

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

BFS Algorithm…

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

2. Depth-first Search

- Depth-first search isa recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

Advantage:

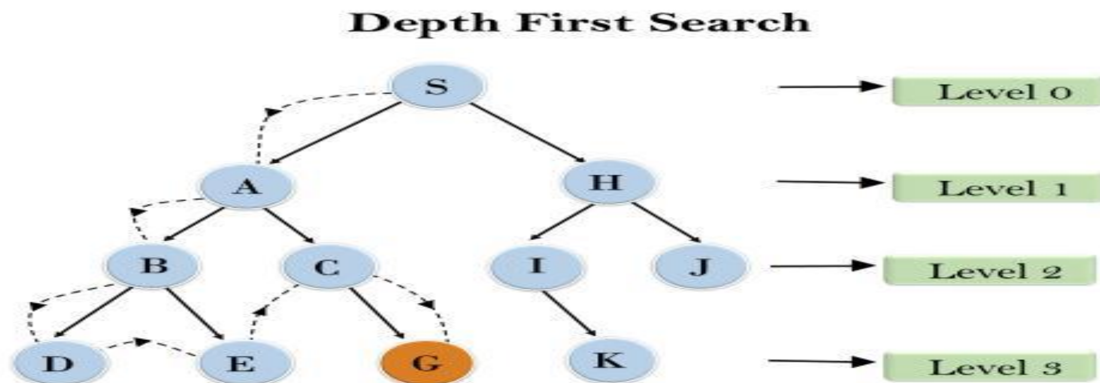
- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).
- With the help of this we can stores the route which is being tracked in memory to save time as it only needs to keep one at a particular time.

Disadvantage:

- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

- The depth-first search (DFS) algorithm does not always find the shorte¬st path to a solution.

Example:
In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:
Root node--->Left node ----> right node.
It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.



Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.
Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:
$T(n)= 1+ n^2+ n^3 +.........+ n^m=O(n^m)$
Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)
Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.
Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

Heuristic Search Techniques in AI

These techniques are essential for tasks that involve finding the best path from a starting point to a goal state, such as in navigation systems, game playing, and optimization problems. This article delves into what heuristic search is, its significance, and the various techniques employed in AI.

Understanding Heuristic Search
Heuristics operates on the search space of a problem to find the best or closest-to-optimal solution via the use of systematic algorithms. In contrast to a brute-force approach, which checks all possible solutions exhaustively, a heuristic search method uses heuristic information to define a route that seems more plausible than the rest. Heuristics, in this case, refer to a set

of criteria or rules of thumb that offer an estimate of a firm's profitability. Utilizing heuristic guiding, the algorithms determine the balance between exploration and exploitation, and thus they can successfully tackle demanding issues. Therefore, they enable an efficient solution finding process.

Components of Heuristic Search

Heuristic search algorithms typically comprise several essential components:

1. State Space: This implies that the totality of all possible states or settings, which is considered to be the solution for the given problem.
2. Initial State: The instance in the search tree of the highest level with no null values, serving as the initial state of the problem at hand.
3. Goal Test: The exploration phase ensures whether the present state is a terminal or consenting state in which the problem is solved.
4. Successor Function: This create a situation where individual states supplant the current state which represent the possible moves or solutions in the problem space.
5. Heuristic Function: The function of a heuristic is to estimate the value or distance from a given state to the target state. It helps to focus the process on regions or states that has prospect of achieving the goal.

Hill Climbing

Hill Climbing can be useful in a variety of optimization problems, such as scheduling, route planning, and resource allocation. However, it has some limitations, such as the tendency to get stuck in local maxima and the lack of diversity in the search space. Therefore, it is often combined with other optimization techniques, such as genetic algorithms or simulated annealing, to overcome these limitations and improve the search results.

Advantages of Hill Climbing algorithm:

1. Hill Climbing is a simple and intuitive algorithm that is easy to understand and implement.
2. It can be used in a wide variety of optimization problems, including those with a large search space and complex constraints.
3. Hill Climbing is often very efficient in finding local optima, making it a good choice for problems where a good solution is needed quickly.
4. The algorithm can be easily modified and extended to include additional heuristics or constraints.

Disadvantages of Hill Climbing algorithm:

1. Hill Climbing can get stuck in local optima, meaning that it may not find the global optimum of the problem.
2. The algorithm is sensitive to the choice of initial solution, and a poor initial solution may result in a poor final solution.
3. Hill Climbing does not explore the search space very thoroughly, which can limit its ability to find better solutions.
4. It may be less effective than other optimization algorithms, such as genetic algorithms or simulated annealing, for certain types of problems.

Hill Climbing is a heuristic search used for mathematical optimization problems in the field of Artificial Intelligence.
Given a large set of inputs and a good heuristic function, it tries to find a sufficiently good solution to the problem. This solution may not be the global optimal maximum.

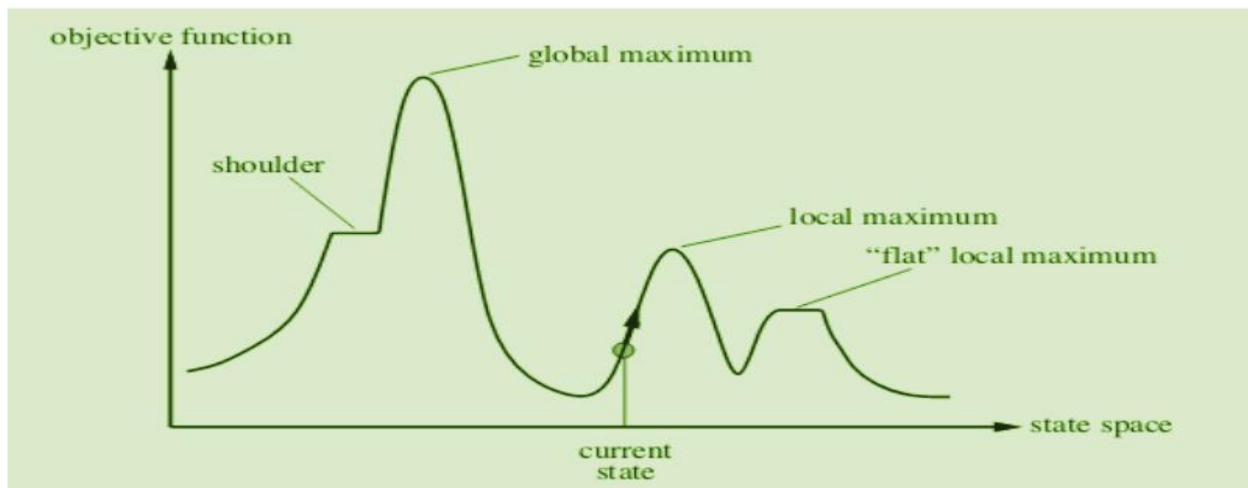
- In the above definition, **mathematical optimization problems** imply that hill-climbing solves the problems where we need to maximize or minimize a given real function by choosing values from the given inputs. Example-Travelling salesman problem where we need to minimize the distance traveled by the salesman.
- 'Heuristic search' means that this search algorithm may not find the optimal solution to the problem. However, it will give a good solution in a **reasonable time.**
- A **heuristic function** is a function that will rank all the possible alternatives at any branching step in the search algorithm based on the available information. It helps the algorithm to select the best route out of possible routes.

State Space diagram for Hill Climbing

The state-space diagram is a graphical representation of the set of states our search algorithm can reach vs the value of our objective function(the function which we wish to maximize).

- **X-axis:** denotes the state space ie states or configuration our algorithm may reach.
- **Y-axis:** denotes the values of objective function corresponding to a particular state.

The best solution will be a state space where the objective function has a maximum value(global maximum).



Different regions in the State Space Diagram:

- **Local maximum:** It is a state which is better than its neighboring state however there exists a state which is better than it(global maximum). This state is better because here the value of the objective function is higher than its neighbors.
- **Global maximum:** It is the best possible state in the state space diagram. This is because, at this stage, the objective function has the highest value.
- **Plateau/flat local maximum:** It is a flat region of state space where neighboring states have the same value.

- **Ridge:** It is a region that is higher than its neighbors but itself has a slope. It is a special kind of local maximum.
- **Current state:** The region of the state space diagram where we are currently present during the search.
- **Shoulder:** It is a plateau that has an uphill edge.

Problems in different regions in Hill climbing

Hill climbing cannot reach the optimal/best state(global maximum) if it enters any of the following regions :

- **Local maximum:** At a local maximum all neighboring states have a value that is worse than the current state. Since hill-climbing uses a greedy approach, it will not move to the worse state and terminate itself. The process will end even though a better solution may exist.
  To overcome the local maximum problem: Utilize the backtracking technique. Maintain a list of visited states. If the search reaches an undesirable state, it can backtrack to the previous configuration and explore a new path.
- **Plateau:** On the plateau, all neighbors have the same value. Hence, it is not possible to select the best direction.
  To overcome plateaus: Make a big jump. Randomly select a state far away from the current state. Chances are that we will land in a non-plateau region.
- **Ridge:** Any point on a ridge can look like a peak because movement in all possible directions is downward. Hence the algorithm stops when it reaches this state.
  To overcome Ridge: In this kind of obstacle, use two or more rules before testing. It implies moving in several directions at once.

Types of Hill Climbing

1. Simple Hill climbing:

It examines the neighboring nodes one by one and selects the first neighboring node which optimizes the current cost as the next node.

2. Steepest-Ascent Hill climbing:

It first examines all the neighboring nodes and then selects the node closest to the solution state as of the next node.

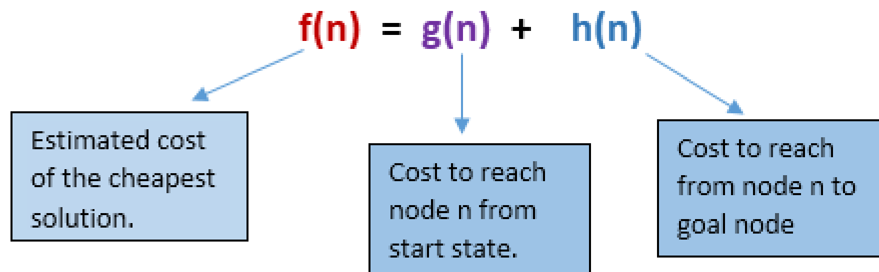
3. Stochastic hill climbing:

It does not examine all the neighboring nodes before deciding which node to select. It just selects a neighboring node at random and decides (based on the amount of improvement in that neighbor) whether to move to that neighbor or to examine another.

A* Search Algorithm:

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).

In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a **fitness number**.



Algorithm of A* search:
Step1: Place the starting node in the OPEN list.
Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise
Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
Step 6: Return to **Step 2**.

Advantages:

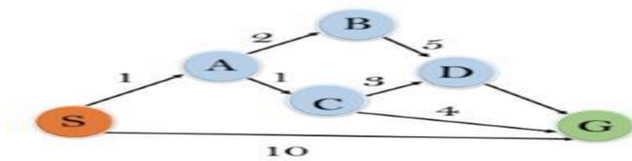
- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

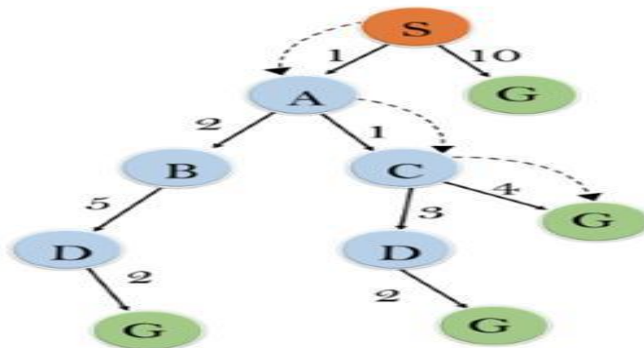
Example:
In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.

Here we will use OPEN and CLOSED list.



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

Solution:



Initialization: {(S, 5)}
Iteration1: {(S--> A, 4), (S-->G, 10)}
Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}
Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}
Iteration 4 will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.

Points to remember:

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n)<="" li="">

Complete: A* algorithm is complete as long as:

- Branching factor is finite.
- Cost at every action is fixed.

Optimal: A* search algorithm is optimal if it follows below two conditions:

- **Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
- **Consistency:** Second required condition is consistency for only A* graph-search.

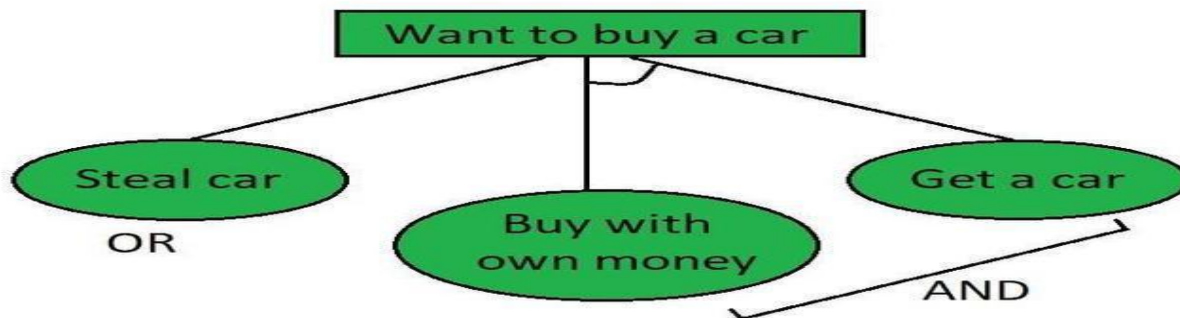
If the heuristic function is admissible, then A* tree search will always find the least cost path.

Time Complexity: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is **O(b^d)**

AO* Algorithms

The AO* method **divides** any given difficult **problem into a smaller group** of problems that are then resolved **using the AND-OR** graph concept. AND OR graphs are specialized graphs that are used in problems that can be divided into smaller problems. The AND side of the graph represents a set of tasks that must be completed to achieve the main goal, while the OR side of the graph represents different methods for accomplishing the same main goal.



In the above figure, the **buying of a car** may be broken down into smaller problems or tasks that can be accomplished **to achieve the main goal** in the above figure, which is an example of a simple AND-OR graph. The other task is to either steal a car that will help us accomplish the main goal or use your own money to purchase a car that will accomplish the main goal. The AND symbol is used to indicate the AND part of the graphs, which refers to the need that all subproblems containing the AND to be resolved before the preceding node or issue may be finished.

The start state and the target state are already known in the knowledge-based search strategy known as the **AO* algorithm**, and the best path is identified by heuristics. The informed search technique considerably reduces the algorithm's **time complexity**. The AO* algorithm is far more effective in searching AND-OR trees **than** the A* algorithm.

Working of AO* algorithm:

The evaluation function in AO* looks like this:

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

f(n) = Actual cost + Estimated cost

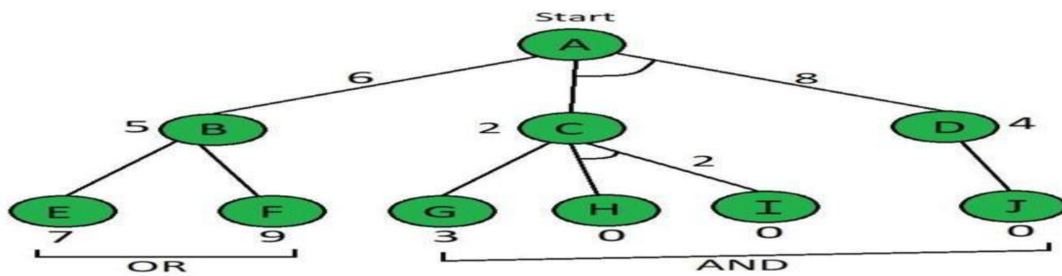
here,

f(n) = The actual cost of traversal.
g(n) = the cost from the initial node to the current node.
h(n) = estimated cost from the current node to the goal state.

Difference between the A* Algorithm and AO* algorithm

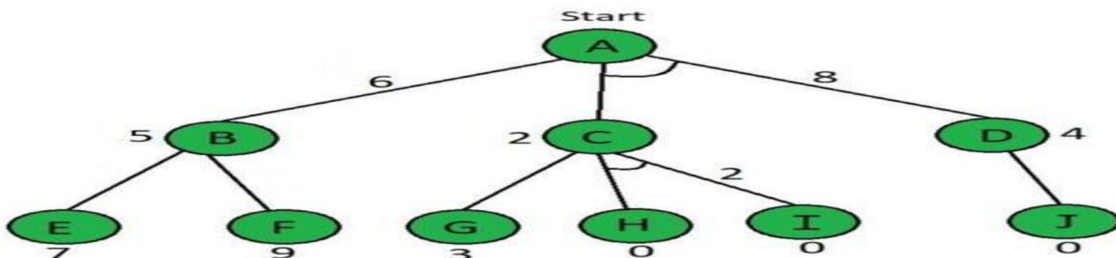
- A* algorithm and AO* algorithm both works on the **best first search**.
- They are both **informed search** and works on given heuristics values.
- **A*** always **gives** the **optimal solution** but AO* doesn't guarantee to give the optimal solution.
- Once AO* got a solution **doesn't explore** all possible paths but A* explores all paths.
- When compared to the A* algorithm, the AO* algorithm uses **less memory.**
- opposite to the A* algorithm, the AO* algorithm cannot go into an endless **loop.**

Example:



Here in the above example below the Node which is given is the heuristic value i.e **h(n)**. Edge length is considered as **1**.

Step 1



With help of **f(n) = g(n) + h(n)** evaluation function,

Start from node A,

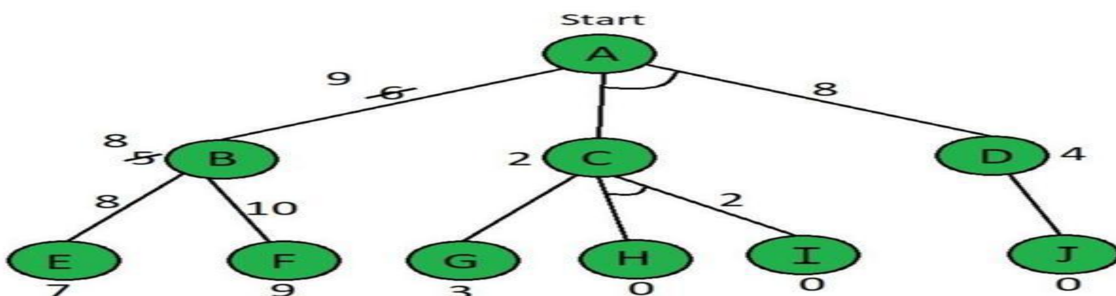
f(A⇢B) = g(B) + h(B)

= 1 + 5 ……here **g(n)=1** is taken by default for path cost
= 6

f(A⇢C+D) = g(c) + h(c) + g(d) + h(d)
= 1 + 2 + 1 + 4 ……here we have added **C & D** because they are in **AND**
= 8

So, by calculation **A⇢B** path is chosen which is the minimum path, i.e **f(A⇢B)**

Step 2



According to the answer of step 1, explore node B

Here the value of E & F are calculated as follows,

f(B⇢E) = g(e) + h(e)

f(B⇢E) = 1 + 7

= 8

f(B⇢f) = g(f) + h(f)

f(B⇢f) = 1 + 9

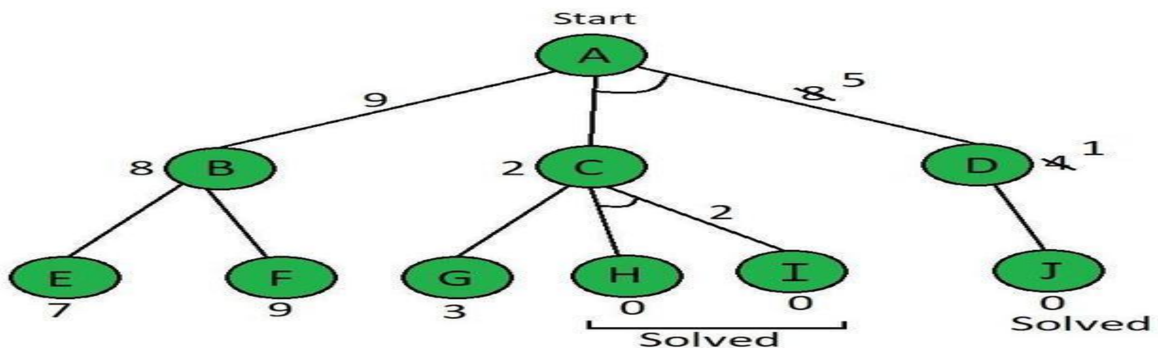
= 10

So, by above calculation B⇢E path is chosen which is minimum path, i.e **f(B⇢E)** because **B's** heuristic value is different from its actual value The heuristic is updated and the minimum cost path is selected. The minimum value in our situation is **8**. Therefore, the heuristic for **A** must be updated due to the change in **B's** heuristic. So we need to calculate it again.

f(A⇢B) = g(B) + updated h(B)
= 1 + 8
= 9

We have Updated all values in the above tree.

Step 3



By comparing **f(A⇢B)**& **f(A⇢C+D)**
f(A⇢C+D) is shown to be **smaller**. i.e 8 < 9
Now explore f(A⇢C+D)
So, the current node is **C**

f(C⇢G) = g(g) + h(g)

f(C⇢G) = 1 + 3
= 4

f(C⇢H+I) = g(h) + h(h) + g(i) + h(i)
f(C⇢H+I) = 1 + 0 + 1 + 0 ……here we have added **H & I** because they are in **AND**
= 2

f(C⇢H+I) is selected as the path with the lowest cost and the heuristic is also left unchanged because it matches the actual cost. Paths H & I are solved because the heuristic for those paths is **0,**
but Path **A⇢D** needs to be calculated because it has an **AND**.

f(D⇢J) = g(j) + h(j)
f(D⇢J) = 1 + 0
= 1
the heuristic of node D needs to be updated to 1.

f(A⇢C+D) = g(c) + h(c) + g(d) + h(d)
= 1 + 2 + 1 + 1
= 5

as we can see that path **f(A⇢C+D)** is get solved and this **tree has become a solved tree** now.
In simple words, the main flow of this algorithm is that we have to find **firstlylevel 1st** heuristic
value and **then level 2nd** and after that **update the values** with going **upward** means towards the root node.
In the above tree diagram, we have updated all the values.

PROBLEM REDUCTION
Problem reduction is a key method in artificial intelligence (AI) that helps solve complex issues. It works by breaking down big problems into smaller, more manageable parts. This approach makes it easier for AI systems to find solutions.

Why Use Problem Reduction?
1. Simplifies complex tasks: By splitting a big problem into smaller pieces, AI can tackle each part separately.
2. Saves time and resources: Solving smaller problems often requires less computing power and time.
3. Improves accuracy: Working on simpler tasks can lead to more precise results.
4. Helps in decision-making: Breaking down problems helps AI make better choices step by step.

Common Problem Reduction Techniques
1. Divide and Conquer This method splits a problem into two or more sub-problems. The AI solves each sub-problem and then combines the results to solve the original problem.

Example: Sorting a large list of numbers can be done by dividing the list, sorting smaller parts, and then merging them.

2. Means-Ends Analysis This technique looks at the difference between the current state and the goal state. It then tries to reduce this difference step by step.

Example: In a puzzle game, AI can compare the current board setup with the desired end result and make moves to get closer to the goal.

3. Problem Abstraction This approach removes unnecessary details from a problem, focusing only on the most important parts.

Example: When planning a route, AI might ignore small side streets and focus only on main roads to find the best path quickly.

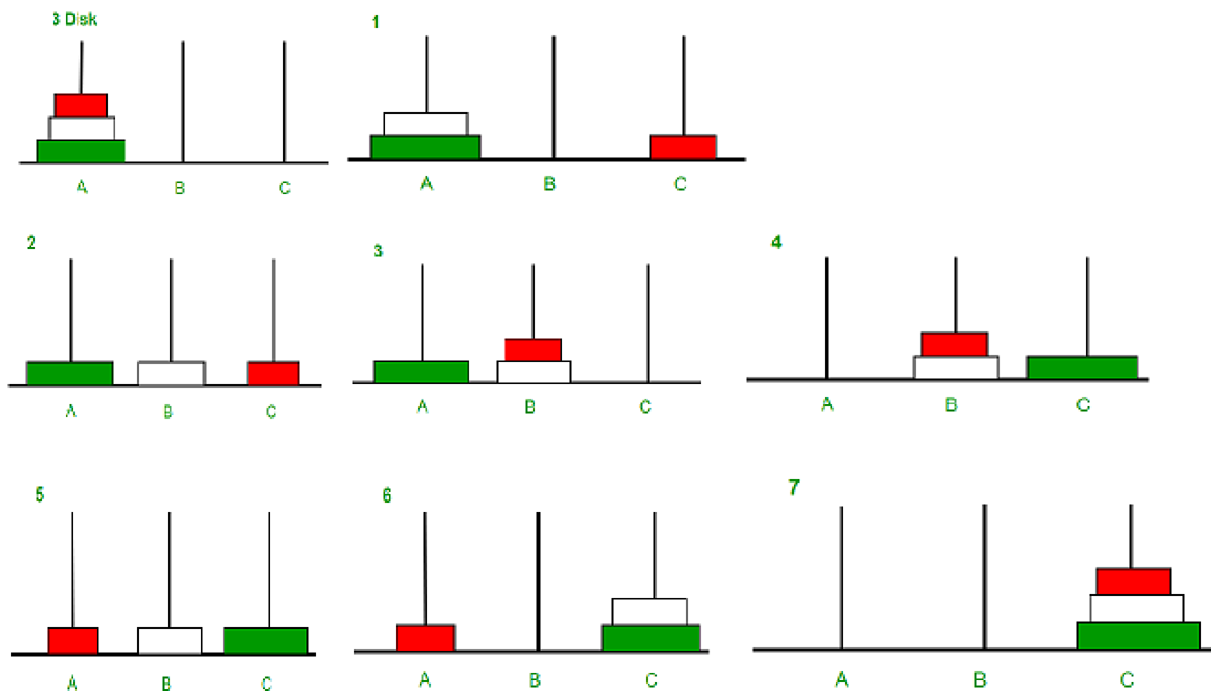
4. Subgoal Decomposition This method breaks down a main goal into smaller, easier-to-achieve subgoals.

Example: To clean a room, AI might set subgoals like "pick up toys," "dust surfaces," and "vacuum floor."

Benefits of Problem Reduction in AI

1. Faster problem-solving
2. Better handling of complex issues
3. More efficient use of resources
4. Improved AI learning and adaptation

Example Towers Of Hanoi



Follow the steps below to solve the problem:

- Create a function **towerOfHanoi** where pass the **N** (current number of disk), **from_rod**, **to_rod**, **aux_rod.**
- Make a function call for N – 1 th disk.

- Then print the current the disk along with **from_rod** and **to_rod**
- Again make a function call for N – 1 th disk.

Game Playing-Adversial search

Adversarial search is a search, where we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us.

Types of Games in AI:

- **Perfect information:** A game with the perfect information is that in which agents can look into the complete board. Agents have all the information about the game, and they can see each other moves also. Examples are Chess, Checkers, Go, etc.
- **Imperfect information:** If in a game agents do not have all information about the game and not aware with what's going on, such type of games are called the game with imperfect information, such as tic-tac-toe, Battleship, blind, Bridge, etc.
- **Deterministic games:** Deterministic games are those games which follow a strict pattern and set of rules for the games, and there is no randomness associated with them. Examples are chess, Checkers, Go, tic-tac-toe, etc.
- **Non-deterministic games:** Non-deterministic are those games which have various unpredictable events and has a factor of chance or luck. This factor of chance or luck is introduced by either dice or cards. These are random, and each action response is not fixed. Such games are also called as stochastic games. Example: Backgammon, Monopoly, Poker, etc.

Zero-Sum Game

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

Zero-sum game: Embedded thinking

The Zero-sum game involved embedded thinking in which one agent or player is trying to figure out:

- What to do.
- How to decide the move
- Needs to think about his opponent as well
- The opponent also thinks what to do

Each of the players is trying to find out the response of his opponent to their actions. This requires embedded thinking or backward reasoning to solve the game problems in AI.

Formalization of the problem:

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

- **Initial state:** It specifies how the game is set up at the start.
- **Player(s):** It specifies which player has moved in the state space.
- **Action(s):** It returns the set of legal moves in state space.
- **Result(s, a):** It is the transition model, which specifies the result of moves in the state space.

- **Terminal-Test(s):** Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called terminal states.
- **Utility(s, p):** A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called payoff function. For Chess, the outcomes are a win, loss, or draw and its payoff values are +1, 0, ½. And for tic-tac-toe, utility values are +1, -1, and 0.

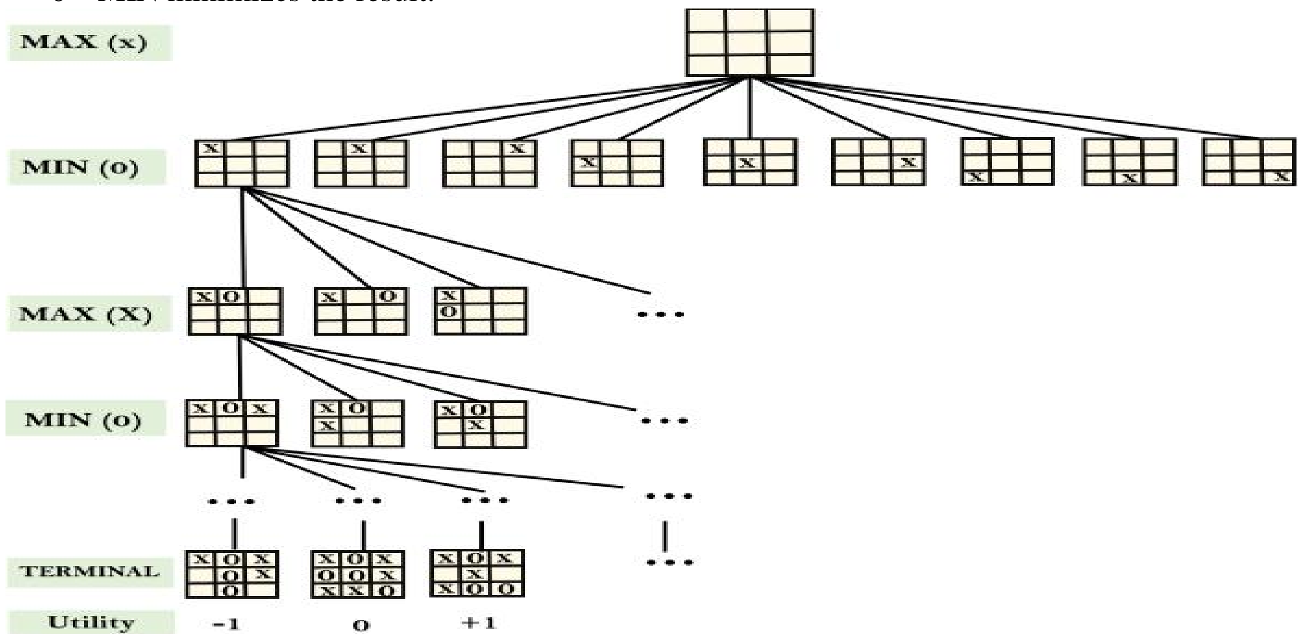
Game tree:

A game tree is a tree where nodes of the tree are the game states and Edges of the tree are the moves by players. Game tree involves initial state, actions function, and result Function.

Example: Tic-Tac-Toe game tree:

The following figure is showing part of the game-tree for tic-tac-toe game. Following are some key points of the game:

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



Example Explanation:

- From the initial state, MAX has 9 possible moves as he starts first. MAX place x and MIN place o, and both player plays alternatively until we reach a leaf node where one player has three in a row or all squares are filled.
- Both players will compute each node, minimax, the minimax value which is the best achievable utility against an optimal adversary.
- Suppose both the players are well aware of the tic-tac-toe and playing the best play. Each player is doing his best to prevent another one from winning. MIN is acting against Max in the game.

- So in the game tree, we have a layer of Max, a layer of MIN, and each layer is called as **Ply**. Max place x, then MIN puts o to prevent Max from winning, and this game continues until the terminal node.
- In this either MIN wins, MAX wins, or it's a draw. This game-tree is the whole search space of possibilities that MIN and MAX are playing tic-tac-toe and taking turns alternately.

Hence adversarial Search for the minimax procedure works as follows:

- It aims to find the optimal strategy for MAX to win the game.
- It follows the approach of Depth-first search.
- In the game tree, optimal leaf node could appear at any depth of the tree.
- Propagate the minimax values up to the tree until the terminal node discovered.

In a given game tree, the optimal strategy can be determined from the minimax value of each node, which can be written as MINIMAX(n). MAX prefer to move to a state of maximum value and MIN prefer to move to a state of minimum value then:

For a state S MINIMAX(s) =

$$\begin{cases} \text{UTILITY}(s) & \text{If TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX(RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX(RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MIN.} \end{cases}$$

Mini-Max Algorithm

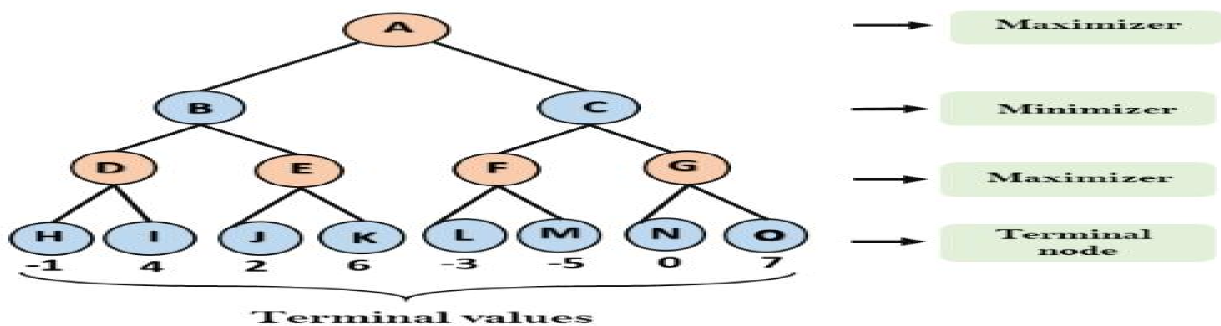
- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Mini-Max algorithm uses recursion to search through the game-tree.
- Min-Max algorithm is mostly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

Initial call:
Minimax(node, 3, true)

Working of Min-Max Algorithm:

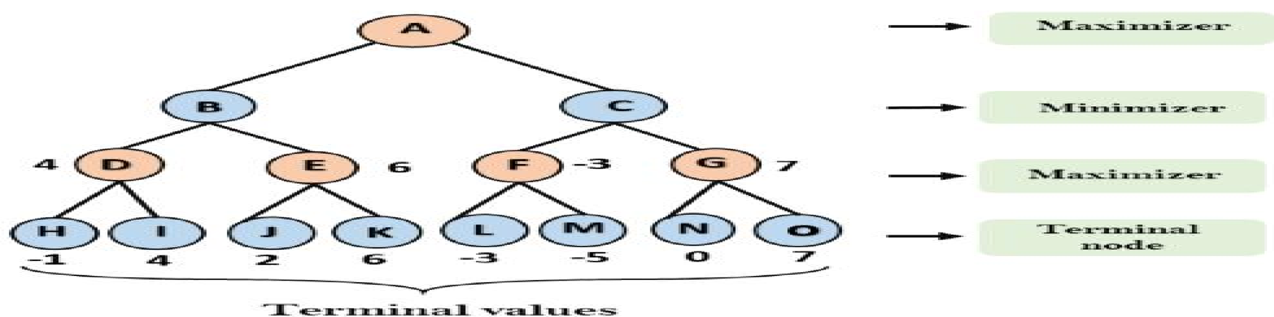
- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.
- At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

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



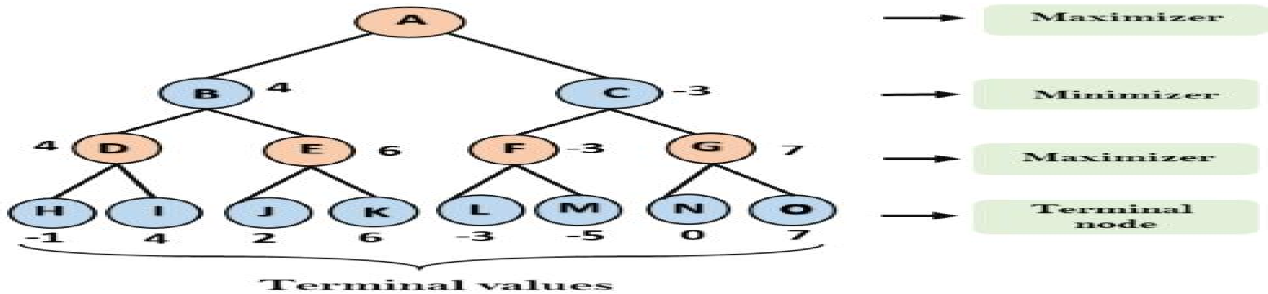
Step 2: Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

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



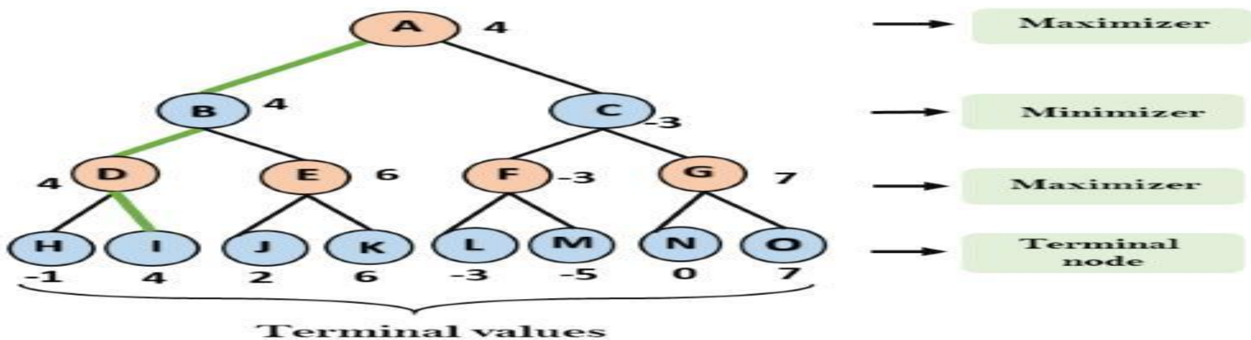
Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

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



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

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



That was the complete workflow of the minimax two player game.

Properties of Mini-Max algorithm:

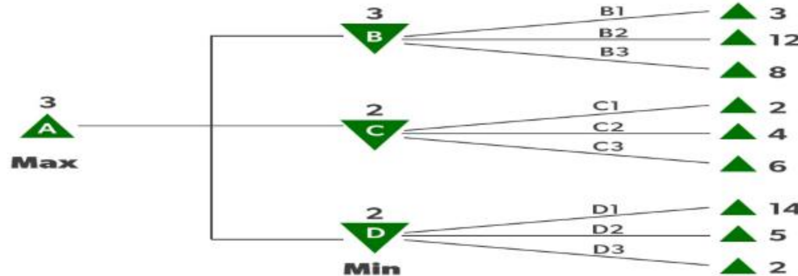
- **Complete-** Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- **Optimal-** Min-Max algorithm is optimal if both opponents are playing optimally.
- **Time complexity-** As it performs DFS for the game-tree, so the time complexity of Min-Max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

Limitation of the minimax Algorithm:

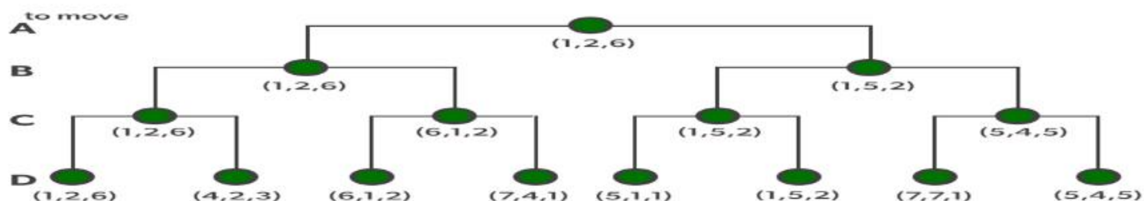
- The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide.

Optimal Decision Making in Multiplayer Games

The optimal solution becomes a contingent strategy when specifies MAX(the player on our side)'s move in the initial state, then Max move to the states resulting for every possible response by MIN. Then MAX's moves in the states resulting from every possible response by MIN to those moves, and so on.



From the current state, the minimax method in the ***Figure above*** computes the minimax choice. It implements the defining equations directly using a simple recursive computation of the minimax values of each successor state. As the recursion unwinds, it progresses all the way down to the tree's leaves, where the minimax values are then backed up through the tree. In the figure below, for example, the algorithm recurses down to the three bottom-left nodes and uses the UTILITY function to determine that their values are 3, 12, and 8, respectively. Then it takes the smallest of these values, 3 in this case, and returns it as node B's backed-up value. The backed-up values of 2 for C and 2 for D are obtained using a similar approach. Finally, we add the maximum of 3, 2, and 2 to get the root node's backed-up value of 3.



The minimax algorithm explores the game tree from top to bottom in depth-first. The temporal complexity of the minimax method is O if the maximum depth of the tree is m and there are b legal moves at each point (bm). For an algorithm that creates all actions at once, the space complexity is O(bm), while for an algorithm that generates actions one at a time, the space complexity is O(m) The time cost is obviously impractical for real games, but this technique serves as a foundation for game mathematics analysis and more practical algorithms.

Disadvantages of Game Playing in Artificial Intelligence:

1. **Limited scope:** The techniques and algorithms developed for game playing may not be well-suited for other types of applications and may need to be adapted or modified for different domains.
2. **Computational cost:** Game playing can be computationally expensive, especially for complex games such as chess or Go, and may require powerful computers to achieve real-time performance.

Alpha-Beta Pruning

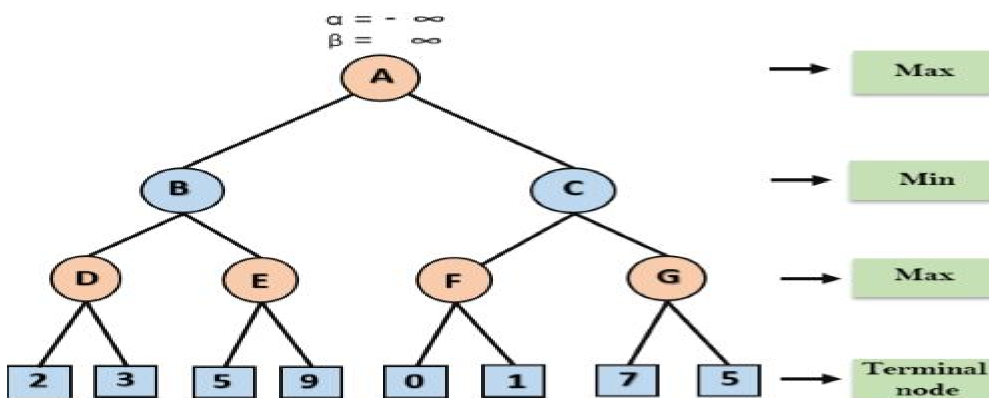
- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**. This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
- The two-parameter can be defined as:
  a. **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is **-∞**.
  b. **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.

The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

Working of Alpha-Beta Pruning:

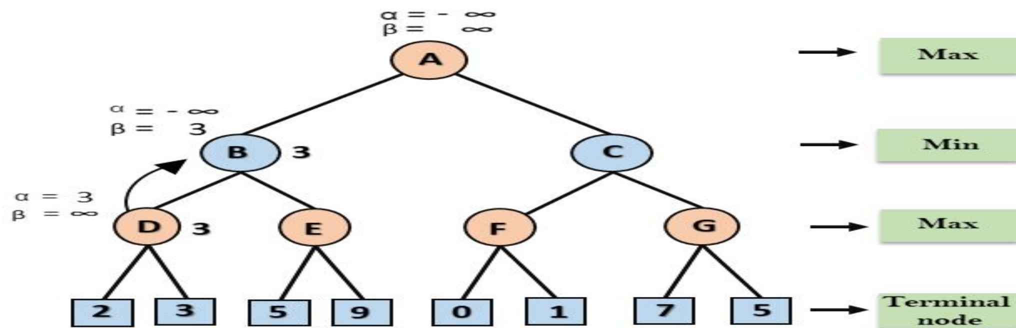
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

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



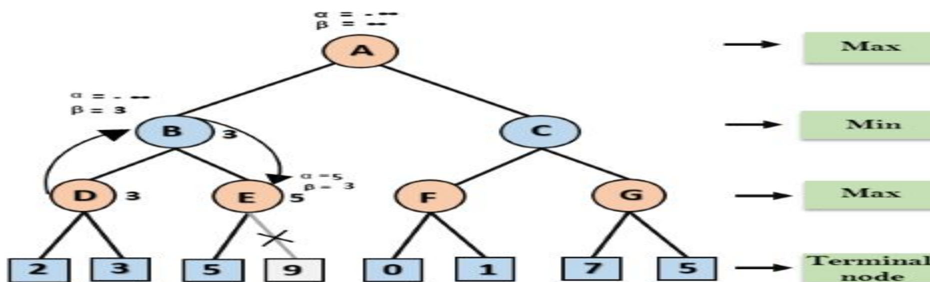
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

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



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

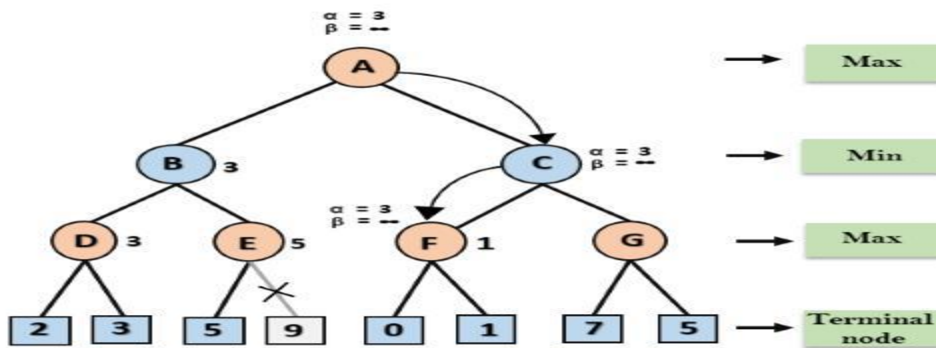
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will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



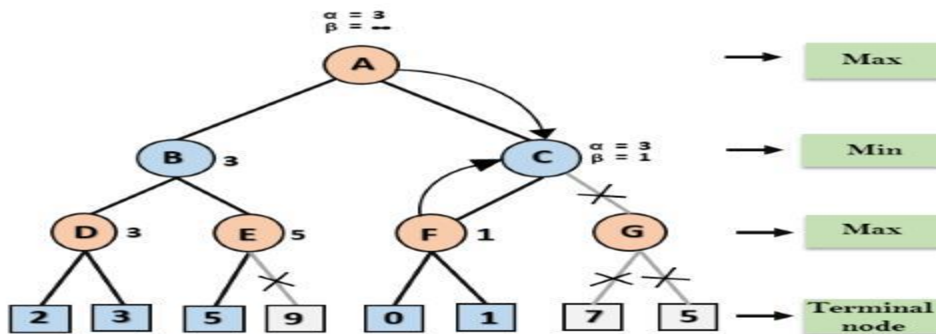
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

At node C, α=3 and β= +∞, and the same values will be passed on to node F.

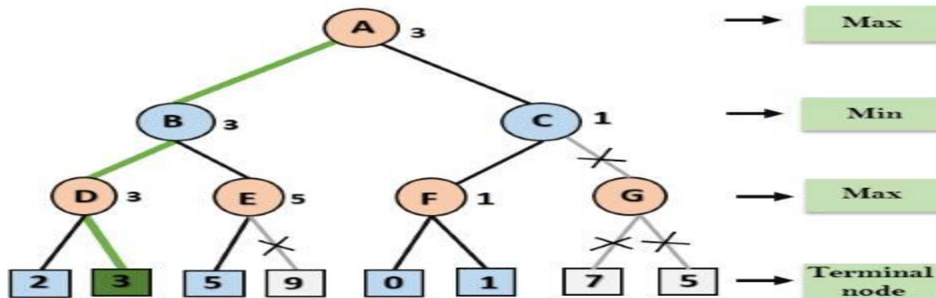
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



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



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



Evaluation functions.

Each leaf node had a value associated with it. We had stored this value in an array. But in the real world when we are creating a program to play Tic-Tac-Toe, Chess, Backgammon, etc. we need to implement a function that calculates the value of the board depending on the placement of pieces on the board. This function is often known as Evaluation Function. It is sometimes also called a Heuristic Function.

The evaluation function is unique for every type of game. In this post, the evaluation function for the game Tic-Tac-Toe is discussed. The basic idea behind the evaluation function is to give a high value for a board if the **maximizer** turn or a low value for the board if the **minimizer** turn.

For this scenario let us consider **X** as the **maximizer** and **O** as the **minimizer**.
Let us build our evaluation function :

- If X wins on the board we give it a positive value of +10.

X	O	O
	X	
		X

+10

- If O wins on the board we give it a negative value of -10.

O	O	O
	X	X
		X

-10

- If no one has won or the game results in a draw then we give a value of +0.

X	O	X
O	X	X
O	X	O

+0

We could have chosen any positive/negative. For the sake of simplicity, we chose 10 and shall use lowercase ‘x’ and lowercase ‘o’ to represent the players and an underscore ‘_’ to represent a blank space on the board.

If we represent our board as a 3×3 2D character matrix, like char board[3][3]; then we have to check each row, each column, and the diagonals to check if either of the players has gotten 3 in a row.

```
# Python3 program to compute evaluation
# function for Tic Tac Toe Game.

# Returns a value based on who is winning
# b[3][3] is the Tic-Tac-Toe board
def evaluate(b):

    # Checking for Rows for X or O victory.
    for row in range(0, 3):

        if b[row][0] == b[row][1] and b[row][1] == b[row][2]:

            if b[row][0] == 'x':
                return 10
            else if b[row][0] == 'o':
                return -10

    # Checking for Columns for X or O victory.
    for col in range(0, 3):

        if b[0][col] == b[1][col] and b[1][col] == b[2][col]:

            if b[0][col]=='x':
                return 10
            else if b[0][col] == 'o':
                return -10

    # Checking for Diagonals for X or O victory.
    if b[0][0] == b[1][1] and b[1][1] == b[2][2]:

        if b[0][0] == 'x':
            return 10
        else if b[0][0] == 'o':
            return -10

    if b[0][2] == b[1][1] and b[1][1] == b[2][0]:

        if b[0][2] == 'x':
            return 10
        else if b[0][2] == 'o':
            return -10
```

```
    # Else if none of them have won then return 0
    return 0

# Driver code
if __name__ == "__main__":

    board = [['x', '_', 'o'],
             ['_', 'x', 'o'],
             ['_', '_', 'x']]

    value = evaluate(board)
    print("The value of this board is", value)
```

Output

```
The value of this board is 10
```

Time Complexity: O(max(row,col))
Auxiliary Space: O(1)

UNIT – III Representation of Knowledge

Knowledge representation issues, predicate logic- logic programming, semantic nets-frames and inheritance, constraint propagation, representing knowledge using rules, rules based deduction systems. Reasoning under uncertainty, review of probability, Bayes' probabilistic interferences and dempstershafer theory.

Knowledge Representation

● Knowledge representation (KR) is an important issue in both cognitive science and artificial intelligence. –

In cognitive science, it is concerned with the way people store and process information and –

In artificial intelligence (AI), main focus is to store knowledge so that programs can process it and achieve human intelligence.

The fundamental goal of knowledge Representation is to facilitate inference (conclusions) from knowledge.
The issues that arise while using KR techniques are many. Some of these are explained below.

1. Important Attributed:
Any attribute of objects so basic that they occur in almost every problem domain?
There are two attributed "instance" and "isa", that are general significance. These attributes are important because they support property inheritance.

2. Relationship among attributes:
Any important relationship that exists among object attributed?
The attributes we use to describe objects are themselves entities that we represent.
The relationship between the attributes of an object, independent of specific knowledge they encode, may hold properties like:

1. Inverse — This is about consistency check, while a value is added to one attribute. The entities are related to each other in many different ways.
2. Existence in an isa hierarchy — This is about generalization-specification, like, classes of objects and specialized subsets of those classes, there are attributes and specialization of attributes. For example, the attribute height is a specialization of general attribute physical-size which is, in turn, a specialization of physical-attribute. These generalization-specialization relationships are important for attributes because they support inheritance.
3. Technique for reasoning about values — This is about reasoning values of attributes not given explicitly. Several kinds of information are used in reasoning, like, height: must be in a unit of length, Age: of a person cannot be greater than the age of person's parents. The values are often specified when a knowledge base is created.

4. Single valued attributes — This is about a specific attribute that is guaranteed to take a unique value. For example, a baseball player can at time have only a single height and be a member of only one team. KR systems take different approaches to provide support for single valued attributes.

3. Choosing Granularity:

At what level of detail should the knowledge be represented?

Regardless of the KR formalism, it is necessary to know:

- At what level should the knowledge be represented and what are the primitives?
- Should there be a small number or should there be a large number of low-level primitives or High-level facts.
- High-level facts may not be adequate for inference while Low-level primitives may require a lot of storage.

Example of Granularity:

- Suppose we are interested in following facts:

John spotted Sue.

This could be represented as

Spotted (agent(John),object (Sue))

Such a representation would make it easy to answer questions such are:

- Who spotted Sue?

Suppose we want to know:

- Did John see Sue?

Given only one fact, we cannot discover that answer.

We can add other facts, such as

Spotted(x, y) -> saw(x, y)

We can now infer the answer to the question.

4. Set of objects:

How should sets of objects be represented?

There are certain properties of objects that are true as member of a set but not as individual;

Example: Consider the assertion made in the sentences:

"there are more sheep than people in Australia", and

"English speakers can be found all over the world."

To describe these facts, the only way is to attach assertion to the sets representing people, sheep, and English.

The reason to represent sets of objects is: if a property is true for all or most elements of a set, then it is more efficient to associate it once with the set rather than to associate it explicitly with every elements of the set.

This is done,

- in logical representation through the use of universal quantifier, and
- in hierarchical structure where node represent sets and inheritance propagate set level assertion down to individual.

5. Finding Right structure:
Given a large amount of knowledge stored in a database, how can relevant parts are accessed when they are needed?
This is about access to right structure for describing a particular situation.
This requires, selecting an initial structure and then revising the choice.
While doing so, it is necessary to solve following problems:

- How to perform an initial selection of the most appropriate structure.
- How to fill in appropriate details from the current situations.
- How to find a better structure if the one chosen initially turns out not to be appropriate.
- What to do if none of the available structures is appropriate.
- When to create and remember a new structure.

There is no good, general purpose method for solving all these problems. Some knowledge representation techniques solve some of these issues.

Predicate Logic - Logic Programming

Predicate Logic or **First-Order Logic (FOL)** is used to represent complex expressions in easier forms using predicates, variables, and quantifiers. The real-world facts can be simply represented as local propositions written as well-formed formulas in propositional logic.

Example of Predicate Logic
Let's see a simple example to understand Predicate Logic in a much better way. Suppose we are given the following statement.

Statement
All white birds are beautiful.

Let's define the predicates and write the expression for this statement.

Predicates

- **IsWhite(x):** Represents the property that x is white.
- **IsBeautiful(x):** Represents the property that x is beautiful.

The following will be the expression for the same.

Expression
$\forall x \, (IsWhite(x) \rightarrow IsBeautiful(x))$

Explanation

The statement **∀ x (IsWhite(x) → IsBeautiful(x))** can be read as "For all x, if x is white, then x is beautiful". The universal quantifier (∀) indicates that the statement applies to all objects (birds) in the domain. The implication (→) connects the properties "IsWhite(x)" and "IsBeautiful(x)," stating that if an object x is white, then it is also beautiful.

Components of Predicate Logic

The three components of Predicate logic are:

Predicates

The symbols which are used to represent the properties or relationships between real-world objects are called **Predicates**. They play an important role in First-Order Logic for knowledge representation.

For example, predicates like **isLocatedIn(Delhi, India)** are used to represent the information.

Variables

The symbols used to represent the objects or entities are called **Variables**. They are used to quantify the objects by providing them with a symbol and passing it to predicates.

For example, in the above example **isLocatedIn(Delhi, India),** the variables are **Delhi** and **India.**

Quantifiers

The symbols in logical statements that are used to represent the scope of variables are called **Quantifiers**. They are used to represent the relationships between objects and properties.

The two main quantifiers are **Universal Quantifier (∀)** and **Existential Quantifier (∃).**

Characteristics of Predicate Logic

The following are the characteristics of Predicate Logic in AI:

- The Predicate Logic makes the representation of complex relationships simpler by representing them using the symbols like Predicates and Variables.
- Predicate Logic is used to represent many complex relationships. It can represent negation, conjunction, disjunction, and many more types of statements.
- Predicate Logic consists of well-defined rules and proper syntax to represent the relationships via valid logical expressions.
- Predicate Logic is used widely in the field of Artificial Intelligence because of its capability to represent facts and relationships in a structured manner.

Before moving towards the example of Predicate Logic, let’s first understand the Connectives and Quantifiers in Predicate Logic.

What are Logical Expressions in Predicate Logic?

Logical expressions in predicate logic are statements that use predicates, variables, and logical operators to express relationships between entities and conditions. These expressions are used to represent and reason about facts, properties, and relationships in a formal, mathematical way.

Here are the key components of logical expressions in predicate logic:

- Predicates: Predicates are functions that take one or more variables as arguments and return a truth value (true or false). They represent properties, relations, or conditions that can be applied to objects or entities. For example, "IsPrime(x)" could be a predicate that checks if a number x is prime.
- Variables: Variables represent entities or objects that can take on various values. In predicate logic, variables are used as placeholders within predicates. For example, in the predicate "IsPrime(x)," 'x' is a variable that can represent any number.
- Quantifiers: Quantifiers are used to specify the scope of variables in logical expressions. There are two main quantifiers in predicate logic.
  Universal Quantifier (∀): Denotes that a statement is true for all values of a variable within a given domain. For example, "∀x IsPrime(x)" means “x is prime for all values of x".
  Existential Quantifier (∃): Denotes that a statement is true for at least one value of a variable within a given domain. For example, "∃x IsPrime(x)" means "there exists a value of x that is prime."
- Logical Operators: Logical operators, such as "AND,""OR,""NOT," and "IMPLIES," are used to combine and manipulate predicates and logical expressions. These operators determine the truth value of compound statements.

Uses of Predicate Logic

- Mathematics: Foundation for formalizing mathematical theories and proofs, allowing precise representation of axioms, theorems, and mathematical structures.
- Computer Science: Used in formal specification languages for software engineering, automated theorem proving, and formal verification of hardware and software systems.
- Artificial Intelligence: Critical for knowledge representation, reasoning, and rule-based systems in AI applications, including expert systems and semantic web technologies.
- Database Management: Utilized in Structured Query Language (SQL) to define and query relational databases, enabling data retrieval and filtering.
- Natural Language Processing: Applied to parse and understand natural language sentences, facilitating tasks like question answering, machine translation, and sentiment analysis in NLP.

What are Logical Connectives in Predicate Logic?

Logical Connectives are symbols that are used to represent more than one statement by combining them to form a complex logical statement. These are used to understand the relationship between the propositions.

The following are the most used logical connectives:

1. **Negation (¬):** Negation is used to negate a statement, which means it reverses the truth value of the expression by changing true to false and vice versa.

For example, if an expression **p** is true, the negation is represented by **¬p.**

2. **Conjunction (∧):** The conjunction is used to combine two statements, and its value is true when both the expression are true. It is basically the representation of logical AND.
   For example, if both **p and q** are true, the value of **p ∧ q** is also true.
   Disjunction (∨): The disjunction is also used to connect two statements, but its value is true when either of the expression is true. It is basically the representation of logical OR.
   For example, if either **p or q** is true**,** the value of **p ∨ q** is also true.

3. **Implication (→):** The implication is also used to connect two statements, but in a way, if one statement, i.e., antecedent, is true, then the other statement, consequent, must have to be true.
   For example, p → q is false only if **p** is true and **q** is false.

4. **Biconditional (↔):** The Biconditional is the logical representation of iff (if and only if) and returns the expression as true if both have the same truth value.
   For example, p ↔ q is true if both **p** and **q** are either true or false.

5. The following is the truth table of the above-discussed Logical Connectives.

p	q	p ∧ q	p ∨ q	p → q	p ↔ q
True	True	True	True	True	True
True	False	False	True	False	False
False	True	False	True	True	False
False	False	False	False	True	True

Quantifiers in Predicate Logic
Quantifiers, as we read above, are the symbols in logical statements that are used to represent the scope of variables. They are used to represent the relationships between objects and properties. The two main quantifiers are:

Universal Quantifier (∀)
The symbol ∀ is used to represent the Universal Quantifier, which basically means **"For All"**. It signifies the expression is true for every object or entity.
For example, "All Boys Like Football" can be written as **∀ x : Boys(x) → Like(x, Football).**

Existential Quantifier (∃)
The symbol ∃ is used to represent the Existential Quantifier, which basically means **"There exists"**. It signifies the expression is true for at least one object or entity.
For example, "Some Boys Like Football" can be written as **∃ x : Boys(x) ∧ Like(x, Football).**

Predicate Logic Vs Propositional Logic

The following are some of the main differences between Predicate Logic and Propositional Logic.

Basis	Predicate Logic	Propositional Logic
Definition	The Predicate Logic is used to represent individual objects and properties and relationships between them.	The Propositional Logic deals with statements that can be either true or false.
Components	Predicates, Variables, Quantifiers	Propositional Variables and Connectives
Symbols	The symbols ∀, ∃, are used to represent Universal and Existential Quantifier, respectively.	The Connectives like ∧, ∨, ¬, →, ↔, etc., are used in Propositional Logic.
Scope	Deals with individual objects and properties	Deals with truth values of statements
Example	∀ x: Girls(x) -> Like(x, Songs)	(p ∧ q) ∨ (¬p ∧ ¬q)

Semantic nets- Frames and Inheritance

Semantic Network:

• Formalism for representing information about objects, people, concepts and specific relationship between them.

• The syntax of semantic net is simple. It is a network of labeled nodes and links. – It's a directed graph with nodes corresponding to concepts, facts, objects etc. and – arcs showing relation or association between two concepts.

The commonly used links in semantic net are of the following types.

– isa → subclass of entity (e.g., child hospital is subclass of hospital)

– inst → particular instance of a class (e.g., India is an instance of country)

prop → property link (e.g., property of dog is 'bark)

Representation of Knowledge in Sem Net "Every human, animal and bird is living thing who breathe and eat. All birds can fly. All man and woman are humans who have two legs. Cat is an animal and has a fur. All animals have skin and can move. Giraffe is an animal who is tall and has long legs. Parrot is a bird and is green in color".

Representation in Predicate Logic

- Every human, animal and bird is living thing who breathe and eat.

  $\forall X$ [human(X) → living(X)]

  $\forall X$ [animal(X) → living(X)]

  $\forall X$ [bird(X) → living(X)]
- All birds are animal and can fly.

  $\forall X$ [bird(X) $\wedge$ canfly(X)]
- Every man and woman are humans who have two legs.

  $\forall X$ [man(X) $\wedge$ haslegs(X)]

  $\forall X$ [woman(X) $\wedge$ haslegs(X)]

  $\forall X$ [human(X) $\wedge$ has(X, legs)]
- Cat is an animal and has a fur.

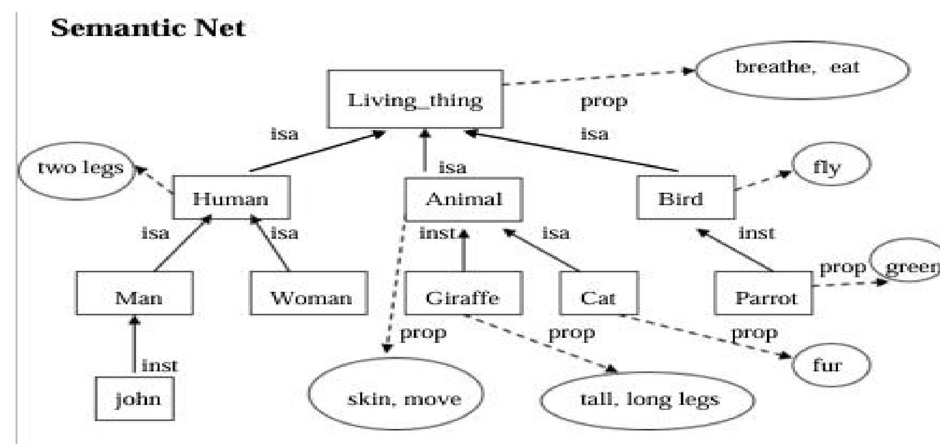
  animal(cat) $\wedge$ has(cat, fur)
- All animals have skin and can move.

  $\forall X$ [animal(X) → has(X, skin) $\wedge$ canmove(X)]
- Giraffe is an animal who is tall and has long legs.

  animal(giraffe) $\wedge$ has(giraffe, long_legs) $\wedge$ is(giraffe, tall)
- Parrot is a bird and is green in color.

  bird(parrot) $\wedge$ has(parrot, green_colour)

Representation in Semantic Net



Inheritance

Inheritance mechanism allows knowledge to be stored at the highest possible level of abstraction which reduces the size of knowledge base.

- It facilitates inferencing of information associated with semantic nets.

– It is a natural tool for representing taxonomically structured information and ensures that all the members and sub-concepts of a concept share common properties.

– It also helps us to maintain the consistency of the knowledge base by adding new concepts and members of existing ones.

Properties attached to a particular object (class) are to be inherited by all subclasses and members of that class.

Coding of Semantic Net in Prolog

Isa facts	Instance facts	Property facts
isa(living_thing, nil).	inst(john, man).	prop(breathe, living_thing).
isa(human, living_thing).	inst(giraffe, animal).	prop(eat, living_thing).
isa(animals, living_thing).	inst(parrot, bird)	prop(two_legs, human).
isa(birds, living_thing).		prop(skin, animal).
isa(man, human).		prop(move, animal).
isa(woman, human).		prop(fur, bird).
isa(cat, animal).		prop(tall, giraffe).
		prop(long_legs, giraffe).
		prop(tall, animal).
		prop(green, parrot).

Inheritance Rules in Prolog

Instance rules:

```
instance(X, Y)      :-   inst(X, Y).
instance (X, Y)     :-   inst(X, Z), subclass(Z,Y).
```

Subclass rules:

```
subclass(X, Y)      :-   isa(X, Y).
subclass(X, Y)      :-   isa(X, Z), subclass(Z, Y) .
```

Property rules:

```
property(X, Y)      :-   prop(X, Y).
property(X, Y)      :-   instance(Y,Z), property(X, Z).
property(X, Y)      :-   subclass(Y, Z), property(X, Z).
```

Knowledge Representation using Frames

Frames are more structured form of packaging knowledge,

– used for representing objects, concepts etc.

● Frames are organized into hierarchies or network of frames.

● Lower level frames can inherit information from upper level frames in network.

● Nodes are connected using links viz.,

– ako / subc (links two class frames, one of which is subclass of other

e.g., science_faculty class is ako of faculty class),

– is_a / inst (connects a particular instance of a class frame

e.g., Renuka is_a science_faculty)

– a_part_of (connects two class frames one of which is contained in other

e.g., faculty class is_part_of department class).

Property link of semantic net is replaced by SLOT fields.

● A frame may have any number of slots needed for describing object.

e.g., – faculty frame may have name, age, address, qualification etc as slot names.

● Each frame includes two basic elements : slots and facets.

– Each slot may contain one or more facets (called fillers) which may take many forms such as:

▪ value (value of the slot),

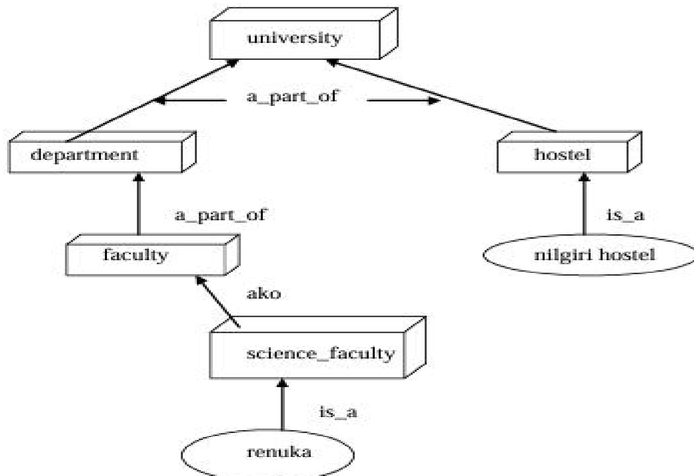
▪ default (default value of the slot),

▪ range (indicates the range of integer or enumerated values, a slot can have),

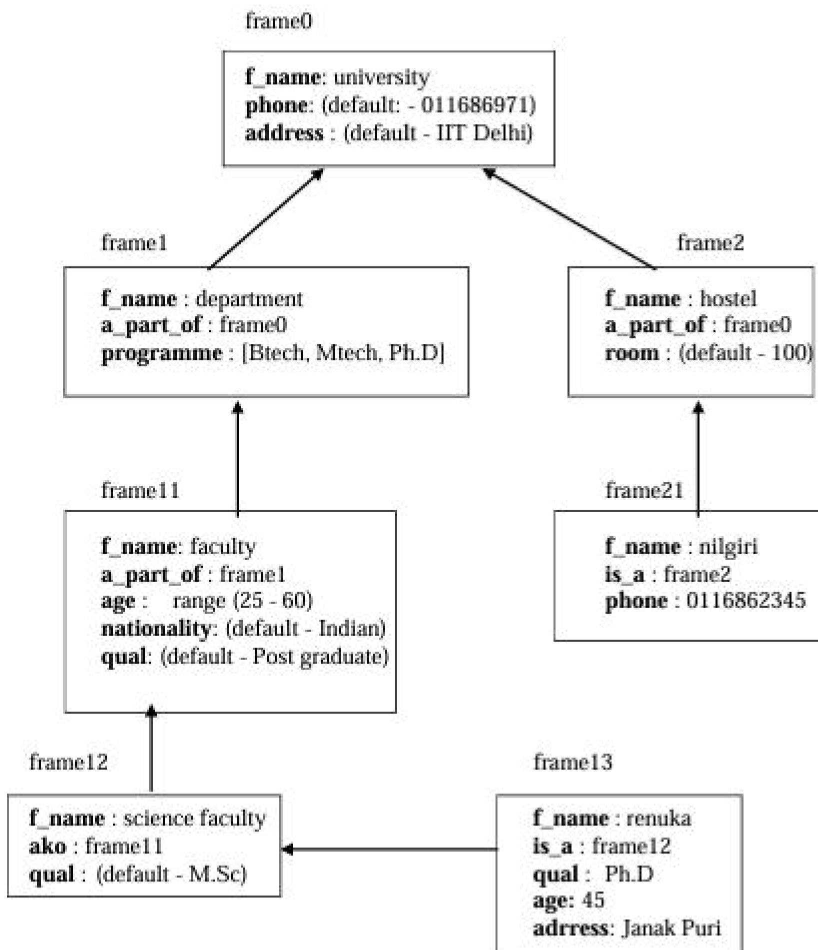
▪ demons (procedural attachments such as if_needed, if_deleted, if_added etc.) and

▪ other (may contain rules, other frames, semantic net or any type of other information).

Frame Network – Example



Detailed Representation of Frame Network



Description of Frames

Each frame represents either a class or an instance.

Class frame represents a general concept whereas instance frame represents a specific occurrence of the class instance.

Class frame generally have default values which can be redefined at lower levels.

If class frame has actual value facet then decedent frames can not modify that value.

Value remains unchanged for subclasses and instances.

Inheritance in Frames

Suppose we want to know nationality or phone of an instance-frame frame13 of renuka.

These informations are not given in this frame.

Search will start from frame13 in upward direction till we get our answer or have reached root frame.

The frames can be easily represented in prolog by choosing predicate name as frame with two arguments.

First argument is the name of the frame and second argument is a list of slot - facet pair.

Constraint propagation:

Artificial Intelligence (AI) encompasses a variety of methods and techniques to solve complex problems efficiently. One such technique is constraint propagation, which plays a crucial role in areas like scheduling, planning, and resource allocation. This article explores the concept of constraint propagation, its significance in AI, and how it is applied in various domains.

Constraint propagation is a fundamental concept in constraint satisfaction problems (CSPs). A CSP involves variables that must be assigned values from a given domain while satisfying a set of constraints. Constraint propagation aims to simplify these problems by reducing the domains of variables, thereby making the search for solutions more efficient.

Key Concepts

1. **Variables**: Elements that need to be assigned values.
2. **Domains**: Possible values that can be assigned to the variables.
3. **Constraints**: Rules that define permissible combinations of values for the variables.

How Constraint Propagation Works

Constraint propagation works by iteratively narrowing down the domains of variables based on the constraints. This process continues until no more values can be eliminated from any domain. The primary goal is to reduce the search space and make it easier to find a solution.

Steps in Constraint Propagation

1. **Initialization**: Start with the initial domains of all variables.
2. **Propagation**: Apply constraints to reduce the domains of variables.
3. **Iteration**: Repeat the propagation step until a stable state is reached, where no further reduction is possible.

Example

Consider a simple CSP with two variables, X and Y, each with domains {1, 2, 3}, and a constraint $X \neq Y$. Constraint propagation will iteratively reduce the domains as follows:

- If X is assigned 1, then Y cannot be 1, so Y's domain becomes {2, 3}.
- If Y is then assigned 2, X cannot be 2, so X's domain is reduced to {1, 3}.
- This process continues until a stable state is reached.

Applications of Constraint Propagation

Constraint propagation is widely used in various AI applications. Some notable areas include:

Scheduling

In scheduling problems, tasks must be assigned to time slots without conflicts. Constraint propagation helps by reducing the possible time slots for each task based on constraints like availability and dependencies.

Planning

AI planning involves creating a sequence of actions to achieve a goal. Constraint propagation simplifies the planning process by reducing the possible actions at each step, ensuring that the resulting plan satisfies all constraints.

Resource Allocation

In resource allocation problems, resources must be assigned to tasks in a way that meets all constraints, such as capacity limits and priority rules. Constraint propagation helps by narrowing down the possible assignments, making the search for an optimal allocation more efficient.

Algorithms for Constraint Propagation

Several algorithms are used for constraint propagation, each with its strengths and weaknesses. Some common algorithms include:

Arc Consistency

Arc consistency ensures that for every value of one variable, there is a consistent value in another variable connected by a constraint. This algorithm is often used as a preprocessing step to simplify CSPs before applying more complex algorithms.

Path Consistency

Path consistency extends arc consistency by considering triples of variables. It ensures that for every pair of variables, there is a consistent value in the third variable. This further reduces the domains and simplifies the problem.

k-Consistency

k-Consistency generalizes the concept of arc and path consistency to k variables. It ensures that for every subset of k-1 variables, there is a consistent value in the kth variable. Higher levels of consistency provide more pruning but are computationally more expensive.

Implementing Constraint Propagation

Step 1: Import Required Libraries

Step 2: Define the CSP Class

Step 3: Consistency Check Method

Step 4: AC-3 Algorithm Method

Step 5: Revise Method

Step 6: Backtracking Search Method

Step 7: Select Unassigned Variable Method

Step 8: Constraint Function

Step 9: Visualization Function

Step 10: Define Variables, Domains, and Neighbors

Step 11: Create CSP Instance and Apply AC-3 Algorithm

Representing knowledge using rules

Procedural and Declarative Knowledge

1. Procedural Knowledge

• The Procedural knowledge is a type of knowledge where the essential control information that is required to use the information is integrated in the knowledge itself.

• It also used with an interpreter to employ the knowledge which follows the instructions given in the knowledge.

• Ex - It can include a group of logical assertions merged with a resolution theorem prove to provide an absolute program for solving problems. Here, the implied income tax of an employee salary can be thought of as a procedural knowledge as it would require a process to calculate it as given below.

So, this is how the tax of an employee is calculated by following a lengthy process instead of just collecting facts.

= GTI (Gross Taxable Income) = Annual Salary of an employee - (Standard deduction + deduction under section 80C)
= Tax computed on GTI (according to slab rate) = A,
= Rebate under section 87A = B;
= Total tax = A - Less B + add: health and education Cess @ 4% on (A-B)

2. Declarative Knowledge

• A Declarative knowledge is where only knowledge is described but not the use to which the knowledge is employed is not provided.

• So, in order to use this declarative knowledge, we need to add it with a program that indicates what is to be done to the knowledge and how it is to be done.

Forward Reasoning

The solution of a problem generally includes the initial data and facts in order to arrive at the solution. These unknown facts and information is used to deduce the result

For example, while diagnosing a patient the doctor first check the symptoms and medical condition of the body such as temperature, blood pressure, pulse, eye colour, blood, etcetera. After that, the patient symptoms are analysed and compared against the predetermined symptoms. Then the doctor is able to provide the medicines according to the symptoms of the patient. So, when a solution employs this manner of reasoning, it is known as forward reasoning.

Steps that are followed in the forward reasoning

1. In the first step, the system is given one or more than one constraints.

2. Then the rules are searched in the knowledge base for each constraint. The rules that fulfill the condition are selected(i.e., IF part). 3. Now each rule is able to produce new conditions from the conclusion of the invoked one. As a result, THEN part is again included in the existing one.

4. The added conditions are processed again by repeating step 2. The process will end if there is no new conditions exist.

Backward Reasoning

• The backward reasoning is inverse of forward reasoning in which goal is analysed in order to deduce the rules, initial facts and data.

• We can understand the concept by the similar example given in the above definition, where the doctor is trying to diagnose the patient with the help of the inceptive data such as symptoms. However, in this case, the patient is experiencing a problem in his body, on the basis of which the doctor is going to prove the symptoms. This kind of reasoning comes under backward reasoning.

Steps that are followed in the backward reasoning

1. Firstly, the goal state and the rules are selected where the goal state reside in the THEN part as the conclusion.

2. From the IF part of the selected rule the sub goals are made to be satisfied for the goal state to be true.

3. Set initial conditions important to satisfy all the sub goals.

4. Verify whether the provided initial state matches with the established states. If it fulfills the condition then the goal is the solution otherwise other goal state is selected.

Rule-Based Deduction Systems

Rule-Based Deduction Systems

The way in which a piece of knowledge is expressed by a human expert carries important information,

example: if the person has fever and feels tummy-pain then she may have an infection.

In logic it can be expressed as follows:

∀x. (has_fever(x) & tummy_pain(x) → has_an_infection(x))

If we convert this formula to clausal form we loose the content as then we may have equivalent formulas like:

(i) has_fever(x) & ~has_an_infection(x) → ~tummy_pain(x)
(ii) ~has_an_infection(x) & tummy_pain(x) → ~has_fever(x)

Notice that:

- (i) and (ii) are logically equivalent to the original sentence
- they have lost the main information contained in its formulation.

Forward production systems

The main idea behind the forward/backward production systems is:

- to take advantage of the implicational form in which production rules are stated by the expert and use that information to help achieving the goal.

- In the present systems the formulas have two forms:

1. rules
2. and facts

Rules are the productions stated in implication form.
Rules express specific knowledge about the problem.
Facts are assertions not expressed as implications.
The task of the system will be to prove a goal formula with these facts and rules.
In a forward production system the rules are expressed as F-rules
F-rules operate on the global database of facts until the termination condition is achieved.
This sort of proving system is a direct system rather than a refutation system.

Facts
Facts are expressed in AND/OR form.
An expression in AND/OR form consists on sub-expressions of literals connected by & and V symbols.
An expression in AND/OR form is not in clausal form.

Steps to transform facts into AND/OR form for forward system:

1. Eliminate (temporarily) implication symbols.
2. Reverse quantification of variables in first disjunct by moving negation symbol.
3. Skolemize existential variables.
4. Move all universal quantifiers to the front an drop.
5. Rename variables so the same variable does not occur in different main conjuncts

-Main conjuncts are small AND/OR trees, not necessarily sum of literal clauses as in Prolog.

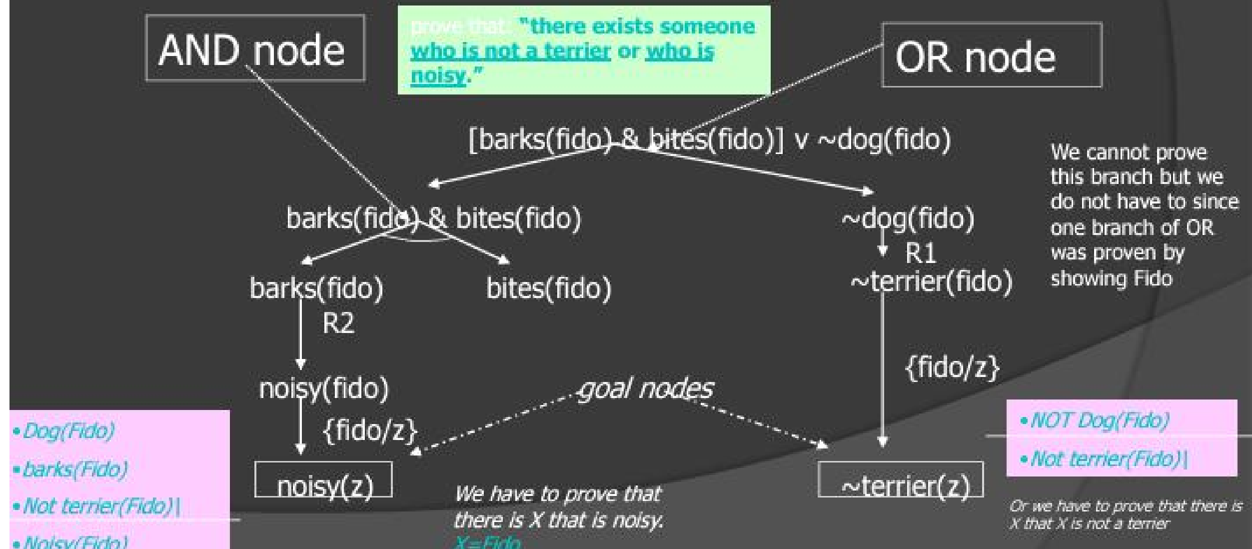
EXAMPLE

Original formula: ∃u. ∀v. {q(v, u) & ~[[r(v) v p(v)] & s(u,v)]}

converted formula: q(w, a) & {[~r(v) & ~p(v)] v ~s(a,v)}

Conjunction of two main conjuncts

Various variables in conjuncts



Reasoning under uncertainty

Knowledge representation using first-order logic and propositional logic with certainty, which means we were sure about the predicates. With this knowledge representation, we might write A→B, which means if A is true then B is true, but consider a situation where we are not sure about whether A is true or not then we cannot express this statement, this situation is called uncertainty.

Causes of uncertainty:

Following are some leading causes of uncertainty to occur in the real world.

1. Information occurred from unreliable sources.
2. Experimental Errors
3. Equipment fault
4. Temperature variation
5. Climate change.

Probabilistic reasoning:

Probabilistic reasoning is a way of knowledge representation where we apply the concept of probability to indicate the uncertainty in knowledge. In probabilistic reasoning, we combine probability theory with logic to handle the uncertainty.

We use probability in probabilistic reasoning because it provides a way to handle the uncertainty that is the result of someone's laziness and ignorance.

In the real world, there are lots of scenarios, where the certainty of something is not confirmed, such as "It will rain today,""behavior of someone for some situations,""A match between two teams or two players." These are probable sentences for which we can assume that it will happen but not sure about it, so here we use probabilistic reasoning.

Need of probabilistic reasoning in AI:

- When there are unpredictable outcomes.
- When specifications or possibilities of predicates becomes too large to handle.
- When an unknown error occurs during an experiment.

In probabilistic reasoning, there are two ways to solve problems with uncertain knowledge:

- **Bayes' rule**
- **Bayesian Statistics**

Probability: Probability can be defined as a chance that an uncertain event will occur. It is the numerical measure of the likelihood that an event will occur. The value of probability always remains between 0 and 1 that represent ideal uncertainties.

$0 \leq P(A) \leq 1$, where P(A) is the probability of an event A.

$P(A) = 0$, indicates total uncertainty in an event A.

$P(A) = 1$, indicates total certainty in an event A.

We can find the probability of an uncertain event by using the below formula.

$$\textbf{Probability of occurrence} = \frac{\text{Number of desired outcomes}}{\text{Total number of outcomes}}$$

- $P(\neg A)$ = probability of a not happening event.
- $P(\neg A) + P(A) = 1$.

Event: Each possible outcome of a variable is called an event.

Sample space: The collection of all possible events is called sample space.

Random variables: Random variables are used to represent the events and objects in the real world.

Prior probability: The prior probability of an event is probability computed before observing new information.

Posterior Probability: The probability that is calculated after all evidence or information has taken into account. It is a combination of prior probability and new information.

Conditional probability:

Conditional probability is a probability of occurring an event when another event has already happened.

Let's suppose, we want to calculate the event A when event B has already occurred, "the probability of A under the conditions of B", it can be written as:

$$P(A|B) = \frac{P(A \wedge B)}{P(B)}$$

Where P($A \wedge B$)= Joint probability of a and B

P(B)= Marginal probability of B.

f the probability of A is given and we need to find the probability of B, then it will be given as:

$$P(B|A) = \frac{P(A \wedge B)}{P(A)}$$

It can be explained by using the below Venn diagram, where B is occurred event, so sample space will be reduced to set B, and now we can only calculate event A when event B is already occurred by dividing the probability of **P(A∧*B*) by P(B)**.

Review of Probability

Probability

Probabilities are used to compute the truth of given statement, written as numbers between 0 and 1, that describes how likely an event is to occur. 0 indicates impossibility and 1 indicates certainly.

1. Tossing a coin

2. Tolling a dice

Probability based reasoning understanding from knowledge how much of uncertainty present in that event.

Probability provides a way of summarizing the uncertainty, that comes from our laziness and ignorance. Toothache problem - an 80 chance , a probability of 0.8 that the patient has a cavity if he or she has a toothache.

The 80 summarizes those cases, but both toothache and cavity are unconnected. The missing 20 summarizes, all other possible causes of toothache, that we are too lazy or ignorant to confirm or deny.

Probabilities between 0 and 1 correspond to intermediate degrees of belief in the truth of the sentence.

The sentence itself is in fact either true or false. It is important to note that a degree of belief is different from a degree of truth.

A probability of 0.8 does not mean "80 true" but rather an 80 degree of belief-that is, a fairly strong expectation. Thus, probability theory makes the same ontological commitment as logic - namely, that facts either do or do not hold in the world. Degree of truth, as opposed to degree of belief, is the subject of fuzzy logic

In probability theory, a sentence such as "The probability that the patient has a cavity is 0.8“, is about the agent's beliefs, not directly about the world. These percepts create the evidence, which are based on probability statements.

All probability statements must indicate the evidence with respect to that probability is being assessed. If an agent receives new percepts, its probability assessments are updated to reflect the new evidence.

Random Variable Referring to a "part" of the world, whose "status" is initially unknown We will use lowercase for the names of values

P(a) = 1 - P(a) Tossing coin : P(h) = 1 - P(h) : (0.5 = 1 - 0.5) Rolling dice : P(n) = 1 – P(n) : (0.16 = 1 - 0.84)

Types of random variables

Boolean random variables Cavity domain (true, false), if Cavity = true then cavity, or if Cavity = false then cavity Discrete random variables – countable domain Weather might be (sunny, rainy, cloudy, snow)

Continuous random variables – finite set real numbers with equal intervals e.g. interval(0.1)

Atomic events

The concept of an atomic event is useful in understanding the foundations of probability theory.

It is a complete specification of the state of the world about which the agent is uncertain.

It can be an assignment of particular values, to all the variables of which the world is composed

Atomic events have some important properties

They are mutually exclusive -at most one can actually be the case.

The set of all possible atomic events is exhaustive – at least one must be the case.

Any particular atomic event entails the truth or falsehood of every proposition, whether simple or complex Any proposition is logically equivalent to the disjunction of all atomic events that required the truth of proposition.

Prior Probability

The unconditional or prior probability associated with a proposition a, is the degree of belief according to the absence of any other information; it is written as P(a).

For example, if the prior probability that one have a cavity is 0.1, then we would write P(Cavity = true) = 0.1 or P(cavity) = 0.1 .

It is important to remember that P(a) can be used only when there is no other information.

we will use an expression P(Weather), which denotes a vector of values, for the probabilities of each individual state of the weather.

P(Weather = sunny) = 0.7 P(Weather = rain) = 0.2 P(Weather = cloudy) = 0.08 P(Weather = snow) = 0.02 .

we may simply write P(Weather) = (0.7, 0.2, 0.08, 0.02) .

This statement defines a prior probability distribution for the random variable Weather.

Conditional Probability

The conditional or posterior probabilities notation is P(a/b), where a and b are any proposition. This is read as "the probability of a, given that all we know is b."

For example, P(cavity/ toothache) = 0.8 if a patient is observed to have a toothache and no other information is yet available, then the probability of the patient's having a cavity will be 0.8.

Conditional probabilities can be defined in terms of unconditional probabilities. The equation is

whenever P(b) 0. This equation can also be written as P(a b) = P(a/ b) P(b) which is called the product rule. Basic Axioms of Probability All probabilities are between 0 and 1. For any proposition a,

Bayes' probabilistic interferences and dempstershafer theory

In probability theory, Bayes' theorem talks about the relation of the conditional probability of two random events and their marginal probability. In short, it provides a way to calculate the value of P(B|A) by using the knowledge of P(A|B).
Bayes' theorem is the name given to the formula used to calculate conditional probability. The formula is as follows:
$P(A|B)=P(A\cap B)/P(B)=(P(A)*P(B|A))/P(B)P(A|B)=P(A\cap B)/P(B)=(P(A)*P(B|A))/P(B)$
where,

- *P(A) is the probability that event A occurs.*
- *P(B) defines the probability that event B occurs.*
- *P(A|B) is the probability of the occurrence of event A given that event B has already occurred.*
- *P(B|A) can now be read as: Probability of event B occurring given that event A occurred.*
- *p(A∩B) is the probability events A and B will happen together.*

Dempster-Shafer Theory was given by Arthur P. Dempster in 1967 and was later developed by his student Glenn Shafer in 1976. This theory was released because of the following reason:-

- Bayesian theory is only concerned about single evidence.
- Bayesian probability cannot describe ignorance.

Dempster Shafer Theory
Dempster Shafer Theory(DST) is an evidence theory, it combines all possible outcomes of the problem. Hence it is used to solve problems where there may be a chance that a piece of different evidence will lead to some different result.

The uncertainty in this model is given by:-

1. Consider all possible outcomes.
2. Belief will lead to belief in some possibility by bringing out some evidence. (What is this supposed to mean?)
3. Plausibility will make evidence compatible with possible outcomes.

Example:

Let us consider a room where four people are present, A, B, C, and D. Suddenly the lights go out and when the lights come back, B has been stabbed in the back by a knife, leading to his death. No one came into the room and no one left the room. We know that B has not committed suicide. Now we have to find out who the murderer is.
To solve these there are the following possibilities:

- Either {A} or {C} or {D} has killed him.
- Either {A, C} or {C, D} or {A, D} have killed him.
- Or the three of them have killed him i.e.; {A, C, D}
- None of them have killed him {o} (let's say).

There will be possible evidence by which we can find the murderer by the measure of plausibility.

Using the above example we can say:
Set of possible conclusion (P): {p1, p2….pn}
where P is a set of possible conclusions and cannot be exhaustive, i.e. at least one (p) I must be true.
(p)I must be mutually exclusive.
Power Set will contain 2n elements where n is the number of elements in the possible set.

For e.g.:-
If P = { a, b, c}, then Power set is given as
{o, {a}, {b}, {c}, {a, d}, {d ,c}, {a, c}, {a, c ,d }}= 23 elements.
Mass function m(K): It is an interpretation of m({K or B}) i.e; it means there is evidence for {K or B} which cannot be divided among more specific beliefs for K and B.

Belief in K: The belief in element K of Power Set is the sum of masses of the element which are subsets of K. This can be explained through an example
Lets say K = {a, d, c}
Bel(K) = m(a) + m(d) + m(c) + m(a, d) + m(a, c) + m(d, c) + m(a, d, c)

Plausibility in K: It is the sum of masses of the set that intersects with K.
i.e.; Pl(K) = m(a) + m(d) + m(c) + m(a, d) + m(d, c) + m(a, c) + m(a, d, c)

Characteristics of Dempster Shafer Theory:

- Uncertainty Representation : The DST is designed to handle situations where there is uncertainty of information and it provides a way to represent and reason incomplete evidence.
- Conflict of Evidence : The DST allows for the combination of multiple sources of evidence. It provides a rule, Dempster's rule of combination, to combine belief functions from different sources.
- Decision-Making Ability : By deriving measures such as belief, probability and plausibility from the combined belief function it helps in decision making.

Advantages of Dempster Shafer Theory:

- As we add more information, the uncertainty interval reduces.
- DST has a much lower level of ignorance.
- Diagnose hierarchies can be represented using this.
- Person dealing with such problems is free to think about evidence.

Disadvantages of Dempster Shafer Theory:

- In this, computation effort is high, as we have to deal with 2n sets

UNIT – IV Logic concepts

First order logic. Inference in first order logic, propositional vs. first order inference, unification & lifts forward chaining, Backward chaining, Resolution, Learning from observation Inductive learning, Decision trees, Explanation based learning, Statistical Learning methods, Reinforcement Learning.

Drawbacks of Propositional Logic

Propositional logic is very simple and declarative, in which knowledge and inference are separate, and inference is entirely domain independent

Propositional logic has lack of data structure in programming.

Propositional logic is not sufficient for complex or natural language sentences.

E.g. Some students in KEC are intelligent

Propositional logic has very limited expressive power

E.g., cannot say "pits cause breezes in adjacent squares" except by writing one sentence for each square

First Order Logic (FOL)

First-order logic is also known as Predicate logic or First-order predicate logic.

First-order logic, like natural language has well defined syntax and semantics.

It assumes the world contains Objects: people, houses, numbers, colors, baseball games, wars, …

Relations: red, round, prime, brother of, bigger than, part of, comes between, …

Functions: father of, best friend, one more than, plus,

Properties of FOL

It has ability to represent facts about some or all of the objects and relations in the universe

Represent law and rules extracted from real world

Useful language for Mathematic , Philosophy and AI

Represent facts in realistic manner rather than just true / false

Makes ontological commitment

- First-order logic (like natural language) does not only assume that the world contains facts like propositional logic but also assumes the following things in the world:

- **Objects:** A, B, people, numbers, colors, wars, theories, squares, pits, wumpus, ...
- **Relations: It can be unary relation such as:** red, round, is adjacent, **or n-any relation such as:** the sister of, brother of, has color, comes between
- **Function:** Father of, best friend, third inning of, end of, ...

- As a natural language, first-order logic also has two main parts:
  - **Syntax**
  - **Semantics**

Syntax of First-Order logic:

The syntax of FOL determines which collection of symbols is a logical expression in first-order logic. The basic syntactic elements of first-order logic are symbols. We write statements in short-hand notation in FOL.

Basic Elements of First-order logic:

Following are the basic elements of FOL syntax:

Constant	1, 2, A, John, Mumbai, cat,....
Variables	x, y, z, a, b,....
Predicates	Brother, Father, >,....
Function	sqrt, LeftLegOf, ...
Connectives	∧, ∨, ¬, ⇒, ⇔
Equality	==
Quantifier	∀, ∃

Atomic sentences:

- Atomic sentences are the most basic sentences of first-order logic. These sentences are formed from a predicate symbol followed by a parenthesis with a sequence of terms.
- We can represent atomic sentences as **Predicate (term1, term2, ..., term n)**.

Example: Ravi and Ajay are brothers: => Brothers(Ravi, Ajay).

Chinky is a cat: => cat (Chinky).

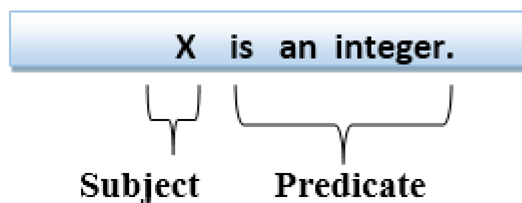
Complex Sentences:

- Complex sentences are made by combining atomic sentences using connectives.

First-order logic statements can be divided into two parts:

- **Subject:** Subject is the main part of the statement.
- **Predicate:** A predicate can be defined as a relation, which binds two atoms together in a statement.

Consider the statement: "x is an integer.", it consists of two parts, the first part x is the subject of the statement and second part "is an integer," is known as a predicate.



Quantifiers in First-order logic:

- 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 determine or identify the range and scope of the variable in the logical expression. There are two types of quantifier:
  - **Universal Quantifier, (for all, everyone, everything)**
  - **Existential quantifier, (for some, at least one).**

Universal Quantifier:

Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing.

The Universal quantifier is represented by a symbol ∀, which resembles an inverted A.

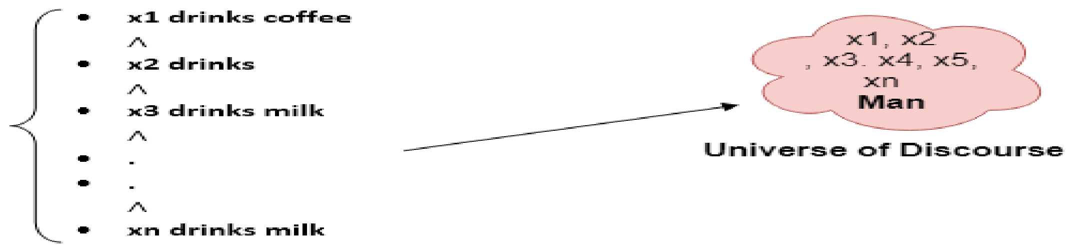
If x is a variable, then ∀x is read as:

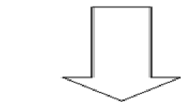
- **For all x**
- **For each x**
- **For every x.**

Example:

All man drink coffee.

Let a variable x which refers to a cat so all x can be represented in UOD as below:





So in shorthand notation, we can write it as :

∀x man(x) → drink (x, coffee).

It will be read as: There are all x where x is a man who drink coffee.

Existential Quantifier:

Existential quantifiers are the type of quantifiers, which express that the statement within its scope is true for at least one instance of something.

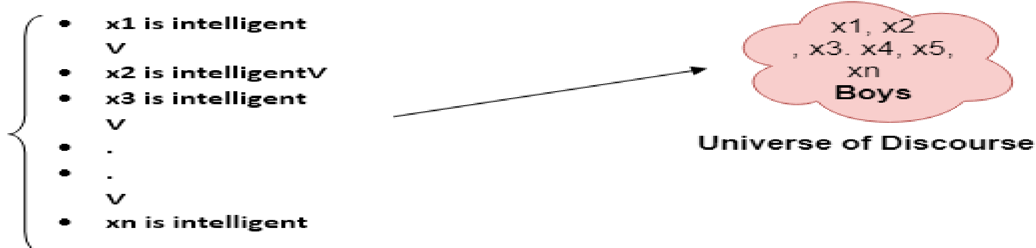
It is denoted by the logical operator ∃, which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.

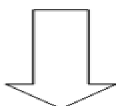
If x is a variable, then existential quantifier will be ∃x or ∃(x). And it will be read as:

- **There exists a 'x.'**
- **For some 'x.'**
- **For at least one 'x.'**

Example:

Some boys are intelligent.





So in short-hand notation, we can write it as:

$\exists x$: boys(x) ∧ intelligent(x)

It will be read as: There are some x where x is a boy who is intelligent.

Properties of Quantifiers:

- In universal quantifier, $\forall x \forall y$ is similar to $\forall y \forall x$.
- In Existential quantifier, $\exists x \exists y$ is similar to $\exists y \exists x$.
- $\exists x \forall y$ is not similar to $\forall y \exists x$.

Some Examples of FOL using quantifier:

1. All birds fly.

In this question the predicate is "fly(bird)."

And since there are all birds who fly so it will be represented as follows.

$\forall x$ bird(x) →fly(x).

2. Every man respects his parent.

In this question, the predicate is "respect(x, y)," where x=man, and y= parent.

Since there is every man so will use ∀, and it will be represented as follows:

$\forall x$ man(x) → respects (x, parent).

3. Some boys play cricket.

In this question, the predicate is "play(x, y)," where x= boys, and y= game. Since there are some boys so we will use ∃, and it will be represented as:

$\exists x$ boys(x) → play(x, cricket).

4. Not all students like both Mathematics and Science.

In this question, the predicate is "like(x, y)," where x= student, and y= subject.

Since there are not all students, so we will use ∀ with negation, so following representation for this:

$$\neg\forall (x)\ [\ student(x) \rightarrow like(x, Mathematics) \land like(x, Science)].$$

5. Only one student failed in Mathematics.

In this question, the predicate is "failed(x, y)," where x= student, and y= subject.

Since there is only one student who failed in Mathematics, so we will use following representation for this:

∃(x) [student(x) → failed (x, Mathematics) ∧∀ (y) [¬(x==y) ∧ student(y) → ¬failed (x, Mathematics)].

Inference in First-Order Logic

Inference in First-Order Logic is used to deduce new facts or sentences from existing sentences. Before understanding the FOL inference rule, let's understand some basic terminologies used in FOL.

Substitution:

Substitution is a fundamental operation performed on terms and formulas. It occurs in all inference systems in first-order logic. The substitution is complex in the presence of quantifiers in FOL. If we write **F[a/x]**, so it refers to substitute a constant "**a**" in place of variable "**x**".

First-Order logic does not only use predicate and terms for making atomic sentences but also uses another way, which is equality in FOL. For this, we can use **equality symbols** which specify that the two terms refer to the same object.

Example: Brother (John) = Smith.

As in the above example, the object referred by the **Brother (John)** is similar to the object referred by **Smith**. The equality symbol can also be used with negation to represent that two terms are not the same objects.

Example: ¬(x=y) which is equivalent to x ≠y.

FOL inference rules for quantifier:

As propositional logic we also have inference rules in first-order logic, so following are some basic inference rules in FOL:

- **Universal Generalization**
- **Universal Instantiation**
- **Existential Instantiation**
- **Existential introduction**

1. Universal Generalization:

- Universal generalization is a valid inference rule which states that if premise P(c) is true for any arbitrary element c in the universe of discourse, then we can have a conclusion as ∀ x P(x).
- It can be represented as: $\frac{P(c)}{\forall x\ P(x)}$.
- This rule can be used if we want to show that every element has a similar property.
- In this rule, x must not appear as a free variable.

Example: Let's represent, P(c): "**A byte contains 8 bits**", so for **∀ x P(x)** "**All bytes contain 8 bits**.", it will also be true.

2. Universal Instantiation:

- Universal instantiation is also called as universal elimination or UI is a valid inference rule. It can be applied multiple times to add new sentences.
- The new KB is logically equivalent to the previous KB.
- As per UI, **we can infer any sentence obtained by substituting a ground term for the variable**.
- The UI rule state that we can infer any sentence P(c) by substituting a ground term c (a constant within domain x) from **∀ x P(x) for any object in the universe of discourse**.
- It can be represented as: $\frac{\forall x\ P(x)}{P(c)}$.

Example:1.

IF "Every person like ice-cream"=>∀x P(x) so we can infer that "John likes ice-cream" => P(c)

Example: 2.

Let's take a famous example,

"All kings who are greedy are Evil." So let our knowledge base contains this detail as in the form of FOL:

∀x king(x) ∧ greedy (x) → Evil (x),

So from this information, we can infer any of the following statements using Universal Instantiation:

- **King(John) ∧ Greedy (John) → Evil (John),**
- **King(Richard) ∧ Greedy (Richard) → Evil (Richard),**

- **King(Father(John)) ∧ Greedy (Father(John)) → Evil (Father(John)),**

3. Existential Instantiation:

- Existential instantiation is also called as Existential Elimination, which is a valid inference rule in first-order logic.
- It can be applied only once to replace the existential sentence.
- The new KB is not logically equivalent to old KB, but it will be satisfiable if old KB was satisfiable.
- This rule states that one can infer P(c) from the formula given in the form of ∃x P(x) for a new constant symbol c.
- The restriction with this rule is that c used in the rule must be a new term for which P(c) is true.
- It can be represented as: $\frac{\exists x\, P(x)}{P(c)}$

Example:

From the given sentence: **∃x Crown(x) ∧ OnHead(x, John),**

So we can infer: **Crown(K) ∧ OnHead(K, John),** as long as K does not appear in the knowledge base.

- The above used K is a constant symbol, which is called **Skolem constant**.
- The Existential instantiation is a special case of **Skolemization process**.

4. Existential introduction

- An existential introduction is also known as an existential generalization, which is a valid inference rule in first-order logic.
- This rule states that if there is some element c in the universe of discourse which has a property P, then we can infer that there exists something in the universe which has the property P.
- It can be represented as: $\frac{P(c)}{\exists x P(x)}$
- **Example: Let's say that,**
  "Priyanka got good marks in English."
  "Therefore, someone got good marks in English."

Generalized Modus Ponens Rule:

For the inference process in FOL, we have a single inference rule which is called Generalized Modus Ponens. It is lifted version of Modus ponens.

Generalized Modus Ponens can be summarized as, " P implies Q and P is asserted to be true, therefore Q must be True."

According to Modus Ponens, for atomic sentences **pi, pi', q**. Where there is a substitution θ such that SUBST **(θ, pi',) = SUBST(θ, pi)**, it can be represented as:

$$\frac{p1', p2', \ldots, pn', (p1 \land p2 \land \ldots \land pn \Rightarrow q)}{\text{SUBST}(\theta, q)}$$

Difference between Propositional and First-Order Logic

Propositional logic, also known as propositional calculus or Boolean logic, is a simple and fundamental form of logic. It deals with propositions, which are statements that can be either true or false. The basic components of propositional logic include:

Propositions: Basic statements that are either true or false.

Logical Connectives: Operators such as AND (∧), OR (∨), NOT (¬), IMPLIES (→), and BICONDITIONAL (↔) used to combine propositions.

Truth Values: Each proposition has a truth value of either true (T) or false (F).

Introduction to First-Order Logic

First-order logic (FOL), also known as predicate logic or first-order predicate calculus, extends propositional logic by introducing quantifiers and predicates. It allows for a more expressive representation of knowledge by dealing with objects, properties, and relationships.

The basic components of first-order logic include:

Constants: Specific objects in the domain (e.g., Alice, Bob).

Variables: Symbols that can represent any object in the domain (e.g., x, y).

Predicates: Functions that map objects to truth values (e.g., Likes(Alice, IceCream)).

Quantifiers: Symbols that indicate the scope of a statement (e.g., ∀ (forall), ∃ (exists)).

Logical Connectives: Same as in propositional logic.

Key Differences Between Propositional Logic and First-Order Logic

Expressiveness

Propositional Logic: Limited to simple true/false statements without the ability to express relationships between objects. Suitable for scenarios where the complexity of relationships is low.

First-Order Logic: More expressive, capable of representing relationships, properties of objects, and quantification. Suitable for complex scenarios involving multiple objects and relationships.

Syntax and Semantics

Propositional Logic: Uses propositions and logical connectives. Each proposition represents a distinct, indivisible truth statement.

First-Order Logic: Uses predicates, constants, variables, and quantifiers in addition to logical connectives. Allows for the construction of more complex statements involving multiple objects and their properties.

Quantification

Propositional Logic: Does not support quantifiers. Statements are either universally true or false.

First-Order Logic: Supports quantifiers ($\forall$ and $\exists$), enabling statements about all or some objects in the domain.

Use Cases

Propositional Logic: Suitable for simple problems like circuit design, troubleshooting, and basic rule-based systems.

First-Order Logic: Suitable for more complex problems involving relationships and properties, such as natural language processing, semantic web, and AI reasoning systems.

Unification and Lifts

- Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process.
- It takes two literals as input and makes them identical using substitution.

- Let Ψ_1 and Ψ_2 be two atomic sentences and ? be a unifier such that, **Ψ_1? = Ψ_2?**, then it can be expressed as **UNIFY(Ψ_1, Ψ_2)**.
- **Example: Find the MGU for Unify{King(x), King(John)}**

Let Ψ_1 = King(x), Ψ_2 = King(John),

Substitution θ = {John/x} is a unifier for these atoms and applying this substitution, and both expressions will be identical.

- The UNIFY algorithm is used for unification, which takes two atomic sentences and returns a unifier for those sentences (If any exist).
- Unification is a key component of all first-order inference algorithms.
- It returns fail if the expressions do not match with each other.
- The substitution variables are called Most General Unifier or MGU.

E.g. Let's say there are two different expressions, **P(x, y), and P(a, f(z))**.

In this example, we need to make both above statements identical to each other. For this, we will perform the substitution.

P(x, y)......... (i)

P(a, f(z))......... (ii)

- Substitute x with a, and y with f(z) in the first expression, and it will be represented as **a/x** and f(z)/y.
- With both the substitutions, the first expression will be identical to the second expression and the substitution set will be: **[a/x, f(z)/y]**.

Conditions for Unification:

Following are some basic conditions for unification:

- Predicate symbol must be same, atoms or expression with different predicate symbol can never be unified.
- Number of Arguments in both expressions must be identical.
- Unification will fail if there are two similar variables present in the same expression.

Unification Algorithm:

Algorithm: Unify(Ψ_1, Ψ_2)

Step. 1: If Ψ_1 or Ψ_2 is a variable or constant, then:

a) If Ψ_1 or Ψ_2 are identical, then return NIL.

b) Else if $\Psi_1$is a variable,

a. then if Ψ_1 occurs in Ψ_2, then return FAILURE

b. Else return { (Ψ_2/ Ψ_1)}.

c) Else if Ψ_2 is a variable,

a. If Ψ_2 occurs in Ψ_1 then return FAILURE,

b. Else return {(Ψ_1/ Ψ_2)}.

d) Else return FAILURE.

Step.2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.

Step. 3: IF Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE.

Step. 4: Set Substitution set(SUBST) to NIL.

Step. 5: For i=1 to the number of elements in Ψ_1.

a) Call Unify function with the ith element of Ψ_1 and ith element of Ψ_2, and put the result into S.

b) If S = failure then returns Failure

c) If S ≠ NIL then do,

a. Apply S to the remainder of both L1 and L2.

b. SUBST= APPEND(S, SUBST).

Step.6: Return SUBST.

Implementation of the Algorithm

Step.1: Initialize the substitution set to be empty.

Step.2: Recursively unify atomic sentences:

1. Check for Identical expression match.
2. If one expression is a variable v_i, and the other is a term t_i which does not contain variable v_i, then:
    1. Substitute t_i / v_i in the existing substitutions
    2. Add t_i /v_i to the substitution setlist.

3. If both the expressions are functions, then function name must be similar, and the number of arguments must be the same in both the expression.

Forward Chaining and backward chaining in AI

In artificial intelligence, forward and backward chaining is one of the important topics, but before understanding forward and backward chaining lets first understand that from where these two terms came.

Inference engine:

The inference engine is the component of the intelligent system in artificial intelligence, which applies logical rules to the knowledge base to infer new information from known facts. The first inference engine was part of the expert system. Inference engine commonly proceeds in two modes, which are:

1. **Forward chaining**
2. **Backward chaining**

Horn Clause and Definite clause:

Horn clause and definite clause are the forms of sentences, which enables knowledge base to use a more restricted and efficient inference algorithm. Logical inference algorithms use forward and backward chaining approaches, which require KB in the form of the **first-order definite clause**.

Definite clause: A clause which is a disjunction of literals with **exactly one positive literal** is known as a definite clause or strict horn clause.

Horn clause: A clause which is a disjunction of literals with **at most one positive literal** is known as horn clause. Hence all the definite clauses are horn clauses.

Example: **($\neg$ p V $\neg$ q V k)**. It has only one positive literal k.

It is equivalent to $p \land q \rightarrow k$.

A. Forward Chaining

Forward chaining is also known as a forward deduction or forward reasoning method when using an inference engine. Forward chaining is a form of reasoning which start with atomic sentences in the knowledge base and applies inference rules (Modus Ponens) in the forward direction to extract more data until a goal is reached.

The Forward-chaining algorithm starts from known facts, triggers all rules whose premises are satisfied, and add their conclusion to the known facts. This process repeats until the problem is solved.

Properties of Forward-Chaining:

- It is a down-up approach, as it moves from bottom to top.
- It is a process of making a conclusion based on known facts or data, by starting from the initial state and reaches the goal state.
- Forward-chaining approach is also called as data-driven as we reach to the goal using available data.
- Forward -chaining approach is commonly used in the expert system, such as CLIPS, business, and production rule systems.

Consider the following famous example which we will use in both approaches:

Example:

"As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American citizen."

Prove that "Robert is criminal."

To solve the above problem, first, we will convert all the above facts into first-order definite clauses, and then we will use a forward-chaining algorithm to reach the goal.

Facts Conversion into FOL:

It is a crime for an American to sell weapons to hostile nations. (Let's say p, q, and r are variables)

American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) ...(1)

Country A has some missiles. ?p Owns(A, p) ∧ Missile(p). It can be written in two definite clauses by using Existential Instantiation, introducing new Constant T1.

Owns(A, T1) ......(2)

Missile(T1) ....(3)

All of the missiles were sold to country A by Robert.

?p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A) ......(4)

Missiles are weapons.

Missile(p) → Weapons (p) ....(5)

Enemy of America is known as hostile.

Enemy(p, America) →Hostile(p) ....(6)

Country A is an enemy of America.

Enemy (A, America) ....(7)

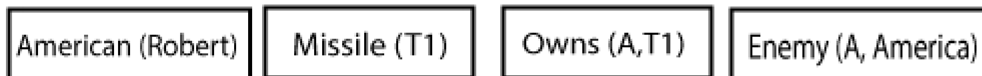
Robert is American

American(Robert). ....(8)

Forward chaining proof:

Step-1:

In the first step we will start with the known facts and will choose the sentences which do not have implications, such as: **American(Robert), Enemy(A, America), Owns(A, T1), and Missile(T1)**. All these facts will be represented as below.



Step-2:

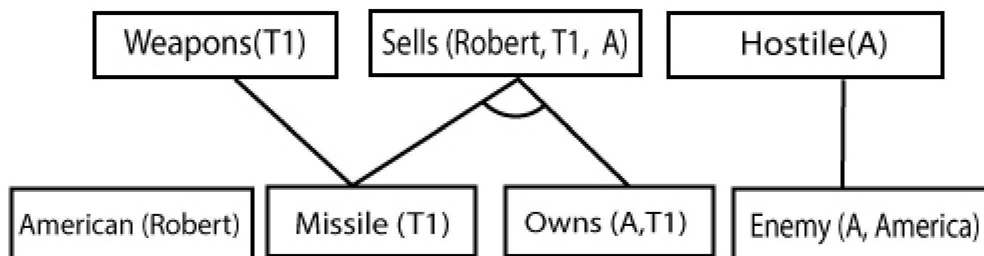
At the second step, we will see those facts which infer from available facts and with satisfied premises.

Rule-(1) does not satisfy premises, so it will not be added in the first iteration.

Rule-(2) and (3) are already added.

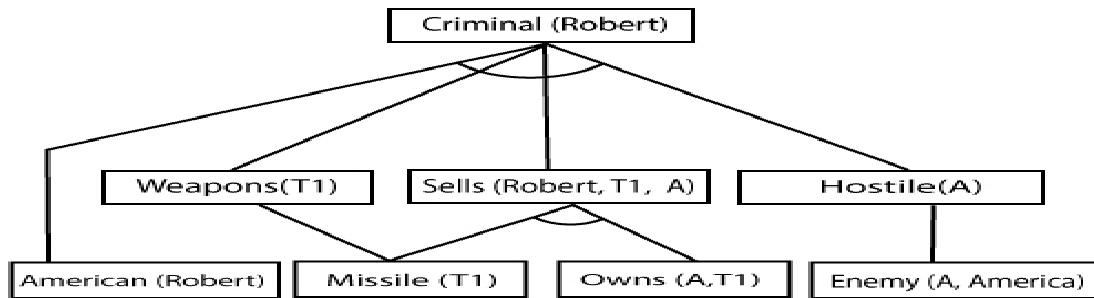
Rule-(4) satisfy with the substitution {p/T1}, **so Sells (Robert, T1, A)** is added, which infers from the conjunction of Rule (2) and (3).

Rule-(6) is satisfied with the substitution(p/A), so Hostile(A) is added and which infers from Rule-(7).



Step-3:

At step-3, as we can check Rule-(1) is satisfied with the substitution **{p/Robert, q/T1, r/A}, so we can add Criminal(Robert)** which infers all the available facts. And hence we reached our goal statement.



Hence it is proved that Robert is Criminal using forward chaining approach.

B. Backward Chaining:

Backward-chaining is also known as a backward deduction or backward reasoning method when using an inference engine. A backward chaining algorithm is a form of reasoning, which starts with the goal and works backward, chaining through rules to find known facts that support the goal.

Properties of backward chaining:

- It is known as a top-down approach.
- Backward-chaining is based on modus ponens inference rule.
- In backward chaining, the goal is broken into sub-goal or sub-goals to prove the facts true.
- It is called a goal-driven approach, as a list of goals decides which rules are selected and used.
- Backward -chaining algorithm is used in game theory, automated theorem proving tools, inference engines, proof assistants, and various AI applications.
- The backward-chaining method mostly used a **depth-first search** strategy for proof.

In backward-chaining, we will use the same above example, and will rewrite all the rules.

American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) ...(1)

Owns(A, T1) ....(2)

Missile(T1)

?p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A) ......(4)

Missile(p) → Weapons (p) ....(5)

Enemy(p, America) →Hostile(p) ....(6)

Enemy (A, America) ....(7)

American(Robert). ....(8)

Backward-Chaining proof:

In Backward chaining, we will start with our goal predicate, which is **Criminal(Robert)**, and then infer further rules.

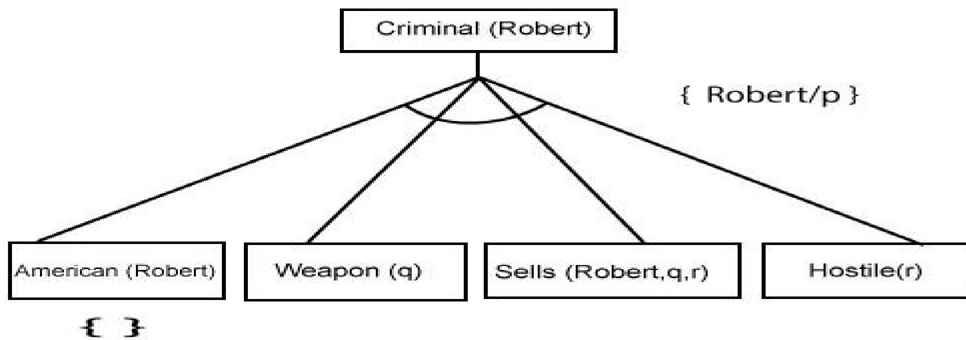
Step-1:

At the first step, we will take the goal fact. And from the goal fact, we will infer other facts, and at last, we will prove those facts true. So our goal fact is "Robert is Criminal," so following is the predicate of it.

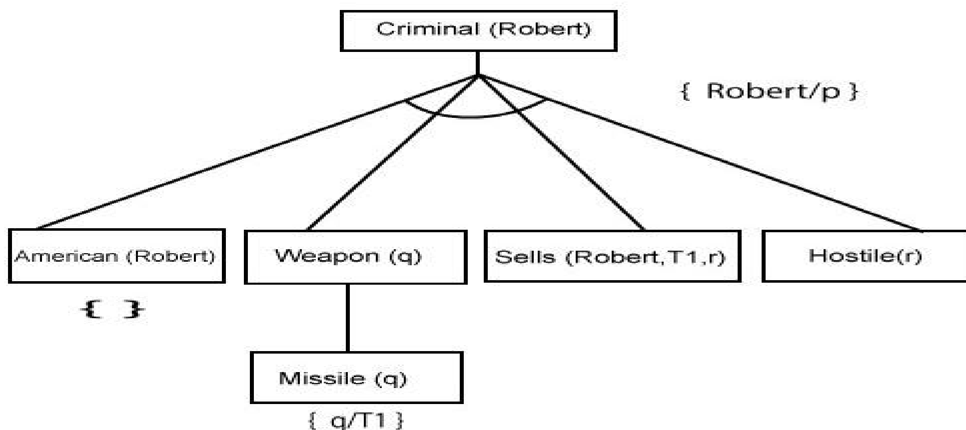
Step-2:

At the second step, we will infer other facts form goal fact which satisfies the rules. So as we can see in Rule-1, the goal predicate Criminal (Robert) is present with substitution {Robert/P}. So we will add all the conjunctive facts below the first level and will replace p with Robert.

Here we can see American (Robert) is a fact, so it is proved here.

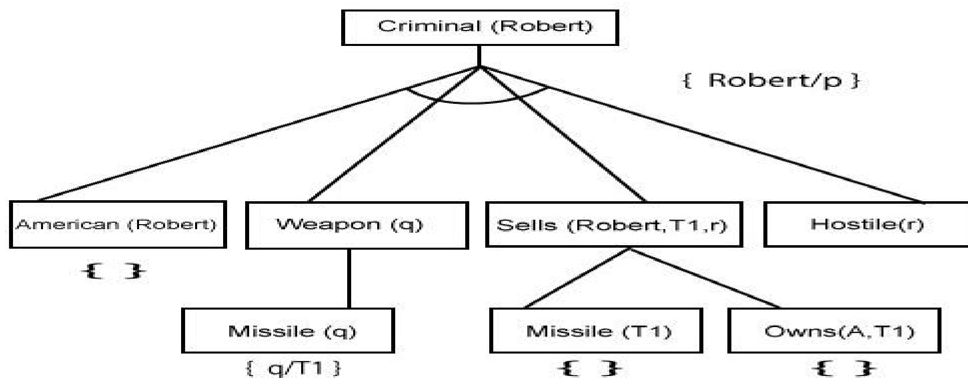


Step-3:t At step-3, we will extract further fact Missile(q) which infer from Weapon(q), as it satisfies Rule-(5). Weapon (q) is also true with the substitution of a constant T1 at q.



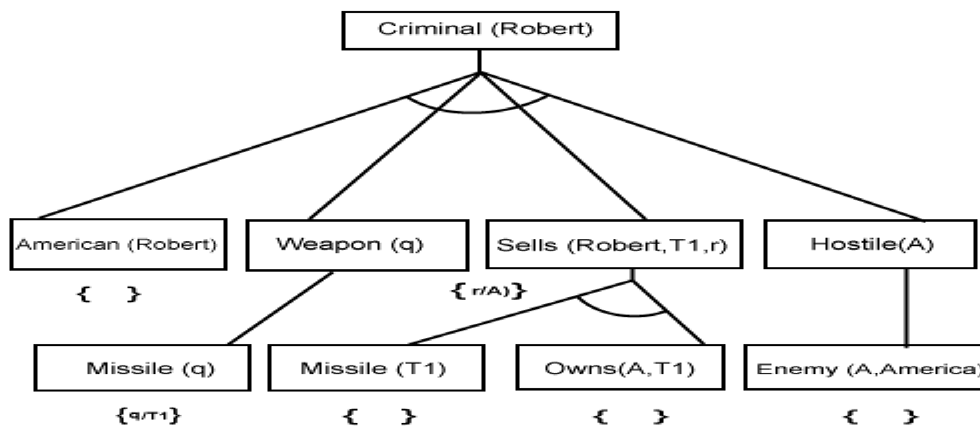
Step-4:

At step-4, we can infer facts Missile(T1) and Owns(A, T1) form Sells(Robert, T1, r) which satisfies the **Rule- 4**, with the substitution of A in place of r. So these two statements are proved here.



Step-5:

At step-5, we can infer the fact **Enemy(A, America)** from **Hostile(A)** which satisfies Rule- 6. And hence all the statements are proved true using backward chaining.



Resolution

Propositional Resolution

1. Convert all the propositions of F to clause form.

2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in step 1.

3. Repeat until either a contradiction is found or no progress can be made:

1. Select two clauses. Call these the parent clauses.

2. Resolve them together. The resulting clause, called the resolvent, will be the disjunction of all of the literals of both of the parent clauses with the following exception: If there are any pairs of

literals L and ¬ L such that one of the parent clauses contains L and the other contains ¬L, then select one such pair and eliminate both L and ¬ L from the resolvent.

3. If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of classes available to the procedure.

Resolution in Predicate Logic

We can now state the resolution algorithm for predicate logic as follows, assuming a set of given statements F and a statement to be proved P: Algorithm: Resolution

1. Convert all the statements of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in 1.
3. Repeat until a contradiction found, no progress can make, or a predetermined amount of effort has expanded.

1. Select two clauses. Call these the parent clauses.
2. Resolve them together. The resolvent will the disjunction of all the literals of both parent clauses with appropriate substitutions performed and with the following exception: If there is one pair of literals T1 and ¬T2 such that one of the parent clauses contains T2 and the other contains T1 and if T1 and T2 are unifiable, then neither T1 nor T2 should appear in the resolvent. We call T1 and T2 Complementary literals. Use the substitution produced by the unification to create the resolvent. If there is more than one pair of complementary literals, only one pair should omit from the resolvent.
3. If the resolvent is an empty clause, then a contradiction has found. Moreover, If it is not, then add it to the set of classes available to the procedure.

Learning from observation

1. Inductive learning

2. Decision trees

3. Explanation based learning

4. Statistical learning methods

5. Reinforcement learning

Inductive learning

Inductive Learning Algorithm (ILA) is an iterative and inductive machine learning algorithm that is used for generating a set of classification rules, which produces rules of the form "IF-THEN", for a set of examples, producing rules at each iteration and appending to the set of rules.

There are basically two methods for knowledge extraction firstly from domain experts and then with machine learning. For a very large amount of data, the domain experts are not very useful and reliable. So we move towards the machine learning approach for this work. To use machine learning One method is to replicate the expert's logic in the form of algorithms but this work is very tedious, time taking, and expensive. So we move towards the inductive algorithms which generate the strategy for performing a task and need not instruct separately at each step.

Why you should use Inductive Learning?

The ILA is a new algorithm that was needed even when other reinforcement learnings like ID3 and AQ were available.

- The need was due to the pitfalls which were present in the previous algorithms, one of the major pitfalls was the lack of generalization of rules.
- The ID3 and AQ used the decision tree production method which was too specific which were difficult to analyze and very slow to perform for basic short classification problems.
- The decision tree-based algorithm was unable to work for a new problem if some attributes are missing.
- The ILA uses the method of production of a general set of rules instead of decision trees, which overcomes the above problems

Basic Requirements to Apply Inductive Learning Algorithm

1. List the examples in the form of a table 'T' where each row corresponds to an example and each column contains an attribute value.
2. Create a set of m training examples, each example composed of k attributes and a class attribute with n possible decisions.
3. Create a rule set, R, having the initial value false.
4. Initially, all rows in the table are unmarked.

Necessary Steps for Implementation

- **Step 1:** divide the table 'T' containing m examples into n sub-tables (t1, t2,…..tn). One table for each possible value of the class attribute. (repeat steps 2-8 for each sub-table)
- **Step 2:** Initialize the attribute combination count ' j ' = 1.
- **Step 3:** For the sub-table on which work is going on, divide the attribute list into distinct combinations, each combination with 'j ' distinct attributes.
- **Step 4:** For each combination of attributes, count the number of occurrences of attribute values that appear under the same combination of attributes in unmarked rows of the sub-table under consideration, and at the same time, not appears under the same combination of attributes of other sub-tables. Call the first combination with the maximum number of occurrences the max-combination ' MAX'.
- **Step 5:** If 'MAX' == null, increase ' j ' by 1 and go to Step 3.
- **Step 6:** Mark all rows of the sub-table where working, in which the values of 'MAX' appear, as classified.
- **Step 7:** Add a rule (IF attribute = "XYZ" –> THEN decision is YES/ NO) to R whose left-hand side will have attribute names of the 'MAX' with their values separated by AND, and its right-hand side contains the decision attribute value associated with the sub-table.
- **Step 8:** If all rows are marked as classified, then move on to process another sub-table and go to Step 2. Else, go to Step 4. If no sub-tables are available, exit with the set of rules obtained till then.

An example showing the use of ILA suppose an example set having attributes Place type, weather, location, decision, and seven examples, our task is to generate a set of rules that under what condition is the decision.

Example no.	Place type	weather	location	decision
1.	hilly	winter	kullu	Yes
2.	mountain	windy	Mumbai	No

Example no.	Place type	weather	location	decision
3.	mountain	windy	Shimla	Yes
4.	beach	windy	Mumbai	No
5.	beach	warm	goa	Yes
6.	beach	windy	goa	No
7.	beach	warm	Shimla	Yes

Subset – 1

s.no	place type	weather	location	decision
1.	hilly	winter	kullu	Yes
2.	mountain	windy	Shimla	Yes
3.	beach	warm	goa	Yes
4.	beach	warm	Shimla	Yes

Subset – 2

s.no	place type	weather	location	decision
5.	mountain	windy	Mumbai	No
6.	beach	windy	Mumbai	No

s.no	place type	weather	location	decision
7.	beach	windy	goa	No

- **At iteration 1** rows 3 & 4 column weather is selected and rows 3 & 4 are marked. the rule is added to R IF the weather is warm then a decision is yes.
- **At iteration 2** row 1 column place type is selected and row 1 is marked. the rule is added to R IF the place type is hilly then the decision is yes.
- **At iteration 3** row 2 column location is selected and row 2 is marked. the rule is added to R IF the location is Shimla then the decision is yes.
- **At iteration 4** row 5&6 column location is selected and row 5&6 are marked. the rule is added to R IF the location is Mumbai then a decision is no.
- **At iteration 5** row 7 column place type & the weather is selected and row 7 is marked. the rule is added to R IF the place type is beach AND the weather is windy then the decision is no.

Finally, we get the rule set:- Rule Set

- **Rule 1:** IF the weather is warm THEN the decision is yes.
- **Rule 2:** IF the place type is hilly THEN the decision is yes.
- **Rule 3:** IF the location is Shimla THEN the decision is yes.
- **Rule 4:** IF the location is Mumbai THEN the decision is no.
- **Rule 5:** IF the place type is beach AND the weather is windy THEN the decision is no.

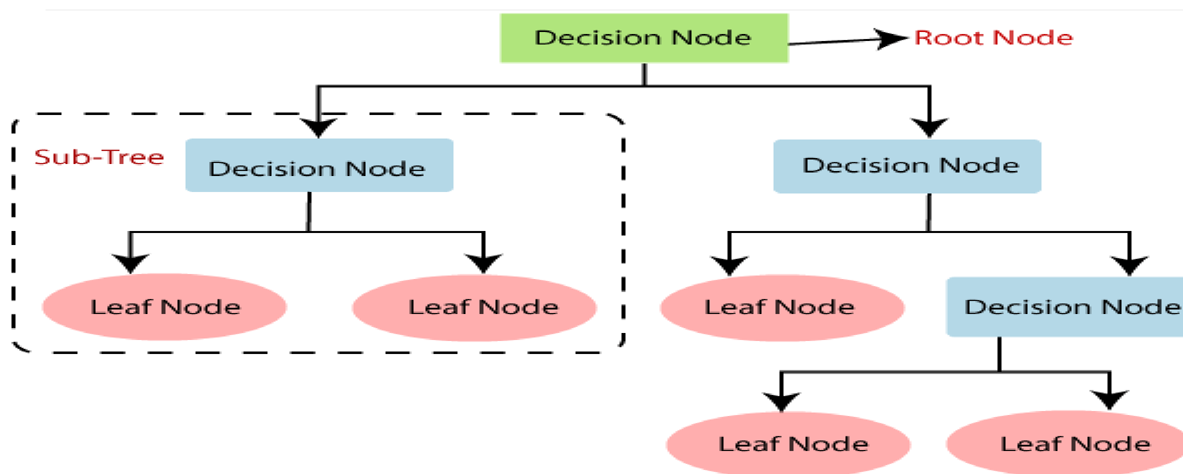
Decision trees

Decision Tree Classification Algorithm

- Decision Tree is a **Supervised learning technique** that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. It is a tree-structured classifier, where **internal nodes represent the features of a dataset, branches represent the decision rules** and **each leaf node represents the outcome.**
- In a Decision tree, there are two nodes, which are the **Decision Node** and **Leaf Node.** Decision nodes are used to make any decision and have multiple branches,

whereas Leaf nodes are the output of those decisions and do not contain any further branches.

- The decisions or the test are performed on the basis of features of the given dataset.
- **It is a graphical representation for getting all the possible solutions to a problem/decision based on given conditions.**
- It is called a decision tree because, similar to a tree, it starts with the root node, which expands on further branches and constructs a tree-like structure.
- In order to build a tree, we use the **CART algorithm,** which stands for **Classification and Regression Tree algorithm.**
- A decision tree simply asks a question, and based on the answer (Yes/No), it further split the tree into subtrees.
- Below diagram explains the general structure of a decision tree:



Decision Tree Terminologies

Root Node: Root node is from where the decision tree starts. It represents the entire dataset, which further gets divided into two or more homogeneous sets.

Leaf Node: Leaf nodes are the final output node, and the tree cannot be segregated further after getting a leaf node.

Splitting: Splitting is the process of dividing the decision node/root node into sub-nodes according to the given conditions.

Branch/Sub Tree: A tree formed by splitting the tree.

Pruning: Pruning is the process of removing the unwanted branches from the tree.

Parent/Child node: The root node of the tree is called the parent node, and other nodes are called the child nodes.

How does the Decision Tree algorithm Work?

- **Step-1:** Begin the tree with the root node, says S, which contains the complete dataset.
- **Step-2:** Find the best attribute in the dataset using **Attribute Selection Measure (ASM).**
- **Step-3:** Divide the S into subsets that contains possible values for the best attributes.
- **Step-4:** Generate the decision tree node, which contains the best attribute.
- **Step-5:** Recursively make new decision trees using the subsets of the dataset created in step -3. Continue this process until a stage is reached where you cannot further classify the nodes and called the final node as a leaf node.

Explanation-Based Learning is a machine learning technique where an AI system learns by analyzing and understanding the underlying structure or reasoning behind a specific example. Unlike traditional learning methods that require numerous examples to generalize, EBL leverages domain knowledge to form a general rule or concept from just one or a few examples. This makes EBL particularly useful in domains where data is scarce or where understanding the rationale behind examples is more critical than just recognizing patterns.

Key Characteristics of EBL

1. **Use of Domain Knowledge:** EBL relies heavily on pre-existing domain knowledge to explain why a particular example is a valid instance of a concept. This knowledge helps the system to generalize the learned concept to new, similar situations.
2. **Focused Learning:** EBL focuses on understanding the essential features of an example that are necessary to achieve a goal or solve a problem. This contrasts with other learning methods that may treat all features equally or rely on statistical correlations.
3. **Efficiency:** Since EBL can learn from a single example by generalizing from it, it is computationally efficient compared to other learning methods that require large datasets for training.

How Explanation-Based Learning Works?

Explanation-Based Learning follows a systematic process that involves the following steps:

1. **Input Example:** The learning process begins with a single example that the system needs to learn from. This example is typically a positive instance of a concept that the system needs to understand.
2. **Domain Knowledge:** The system uses domain knowledge, which includes rules, concepts, and relationships relevant to the problem domain. This knowledge is crucial for explaining why the example is valid.
3. **Explanation Generation:** The system generates an explanation for why the example satisfies the concept. This involves identifying the relevant features and their relationships that make the example a valid instance.
4. **Generalization:** Once the explanation is generated, the system generalizes it to form a broader concept that can apply to other similar examples. This generalization is typically in the form of a rule or a set of rules that describe the concept.
5. **Learning Outcome:** The outcome of EBL is a generalized rule or concept that can be applied to new situations. The system can now use this rule to identify or solve similar problems in the future.

Example of Explanation-Based Learning in AI

Scenario: Diagnosing a Faulty Component in a Car Engine

Context: Imagine you have an AI system designed to diagnose problems in car engines. One day, the system is given a specific example where the engine fails to start. After analyzing the case, the system learns that the failure was due to a faulty ignition coil.

Step 1: Input Example

The system is provided with a scenario where a car engine fails to start. The diagnostic information indicates that the cause is a faulty ignition coil.

Step 2: Use of Domain Knowledge

The AI system has pre-existing domain knowledge about car engines. It knows how the ignition system works, the role of the ignition coil, and the conditions under which an engine would fail to start.

Step 3: Explanation Generation

Using this domain knowledge, the system generates an explanation for why the engine failure occurred:

- **Ignition System Knowledge**: The system understands that the ignition coil is responsible for converting the battery's low voltage to the high voltage needed to create a spark in the spark plugs.
- **Faulty Coil Impact**: It explains that if the ignition coil is faulty, it will fail to generate the necessary high voltage, resulting in no spark, which prevents the engine from starting.

Step 4: Generalization

The system then generalizes this explanation to form a rule:

- **General Rule**: "If the engine fails to start and the ignition coil is faulty, then the cause of the failure is likely due to the ignition coil not providing the necessary voltage to the spark plugs."

Step 5: Learning Outcome

The AI system has now learned a new diagnostic rule that can be applied to future cases:

- **Future Application**: In future diagnostics, if the system encounters a similar scenario where the engine fails to start, it can use this learned rule to quickly check the ignition coil as a potential cause.

Applications of Explanation-Based Learning

Explanation-Based Learning is particularly useful in domains where understanding the reasoning behind decisions is critical.

Some of the notable applications of EBL include:

- **Medical Diagnosis:** EBL can be used in medical diagnosis systems to learn from specific cases and generalize the underlying principles for diagnosing similar conditions in other patients.
- **Legal Reasoning:** In legal systems, EBL can help in understanding the principles behind legal precedents and applying them to new cases with similar circumstances.
- **Automated Planning:** EBL is useful in automated planning systems, where it can learn from successful plans and generalize the steps required to achieve similar goals in different contexts.

- **Natural Language Processing:** EBL can be applied in natural language processing tasks where understanding the structure and meaning behind language is more important than statistical correlations.

STATISTICAL LEARNING METHODS

The goal of learning is prediction. Learning falls into many categories, including:-
Supervised learning,-
Unsupervised learning,-
Semi-supervised learning –
Transfer Learning-
Online learning, and-
Reinforcement learning.

1. **Supervised Learning:**

- Definition: In supervised learning, the algorithm is trained on a labeled dataset, where both the input data and the corresponding desired output (target) are provided. The goal is to learn a mapping or relationship between the input features and the target variable.
- Examples: Regression (predicting a continuous outcome) and classification (predicting a categorical outcome) are common types of supervised learning tasks.

2. **Unsupervised Learning:**

- Definition: Unsupervised learning involves working with unlabeled data, where the algorithm explores the inherent structure or patterns in the input features without explicit guidance on the output. Clustering and dimensionality reduction are examples of unsupervised learning tasks.
- Examples: K-means clustering, hierarchical clustering, and principal component analysis (PCA) are unsupervised learning techniques.

3. **Statistical Models:**

- Definition: Statistical models are mathematical representations that describe the relationships between variables in a dataset. These models can be simple, such as linear regression, or complex, such as ensemble methods or deep neural networks.

- Examples: Linear regression, logistic regression, decision trees, support vector machines, and neural networks are common statistical models used in learning algorithms.

Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning focused on making decisions to maximize cumulative rewards in a given situation. Unlike supervised learning, which relies on a training dataset with predefined answers, RL involves learning through experience. In RL, an agent learns to achieve a goal in an uncertain, potentially complex environment by performing actions and receiving feedback through rewards or penalties.

Key Concepts of Reinforcement Learning

- **Agent:** The learner or decision-maker.
- **Environment:** Everything the agent interacts with.
- **State:** A specific situation in which the agent finds itself.
- **Action:** All possible moves the agent can make.
- **Reward:** Feedback from the environment based on the action taken.

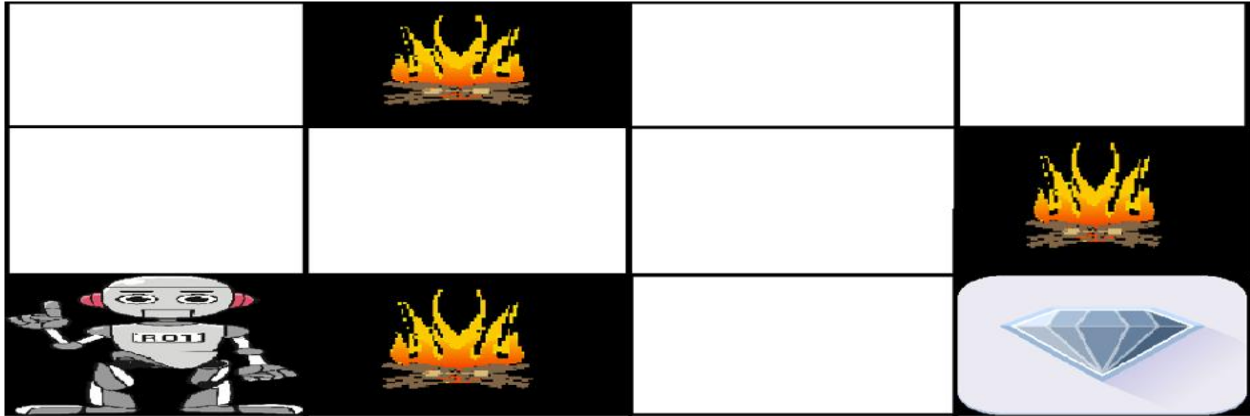
How Reinforcement Learning Works

RL operates on the principle of learning optimal behavior through trial and error. The agent takes actions within the environment, receives rewards or penalties, and adjusts its behavior to maximize the cumulative reward. This learning process is characterized by the following elements:

- **Policy:** A strategy used by the agent to determine the next action based on the current state.
- **Reward Function:** A function that provides a scalar feedback signal based on the state and action.
- **Value Function:** A function that estimates the expected cumulative reward from a given state.
- **Model of the Environment:** A representation of the environment that helps in planning by predicting future states and rewards.

Example: Navigating a Maze

The problem is as follows: We have an agent and a reward, with many hurdles in between. The agent is supposed to find the best possible path to reach the reward. The following problem explains the problem more easily.



The above image shows the robot, diamond, and fire. The goal of the robot is to get the reward that is the diamond and avoid the hurdles that are fired. The robot learns by trying all the possible paths and then choosing the path which gives him the reward with the least hurdles. Each right step will give the robot a reward and each wrong step will subtract the reward of the robot. The total reward will be calculated when it reaches the final reward that is the diamond.

Main points in Reinforcement learning –

- Input: The input should be an initial state from which the model will start
- Output: There are many possible outputs as there are a variety of solutions to a particular problem
- Training: The training is based upon the input, The model will return a state and the user will decide to reward or punish the model based on its output.
- The model keeps continues to learn.
- The best solution is decided based on the maximum reward.

Types of Reinforcement:

- **Positive:** Positive Reinforcement is defined as when an event, occurs due to a particular behavior, increases the strength and the frequency of the behavior. In other words, it has a positive effect on behavior.

Advantages of reinforcement learning are:

Maximizes Performance

- Sustain Change for a long period of time
- Too much Reinforcement can lead to an overload of states which can diminish the results
- **Negative:** Negative Reinforcement is defined as strengthening of behavior because a negative condition is stopped or avoided.

Advantages of reinforcement learning:

Increases Behavior

- Provide defiance to a minimum standard of performance
- It Only provides enough to meet up the minimum behavior

Elements of Reinforcement Learning

i) Policy: Defines the agent's behavior at a given time.

ii) Reward Function: Defines the goal of the RL problem by providing feedback.

iii) Value Function: Estimates long-term rewards from a state.

iv) Model of the Environment: Helps in predicting future states and rewards for planning.

UNIT – V Expert Systems

Architecture of expert systems, Roles of expert systems – Knowledge Acquisition Meta knowledge Heuristics. Typical expert systems – MYCIN, DART, XCON: Expert systems shells.

Architecture of an Expert System

The knowledge base contains the specific domain knowledge that is used by an expert to derive conclusions from facts.

In the case of a rule-based expert system, this domain knowledge is expressed in the form of a series of rules.

The explanation system provides information to the user about how the inference engine arrived at its conclusions. This can often be essential, particularly if the advice being given is of a critical nature, such as with a medical diagnosis system.

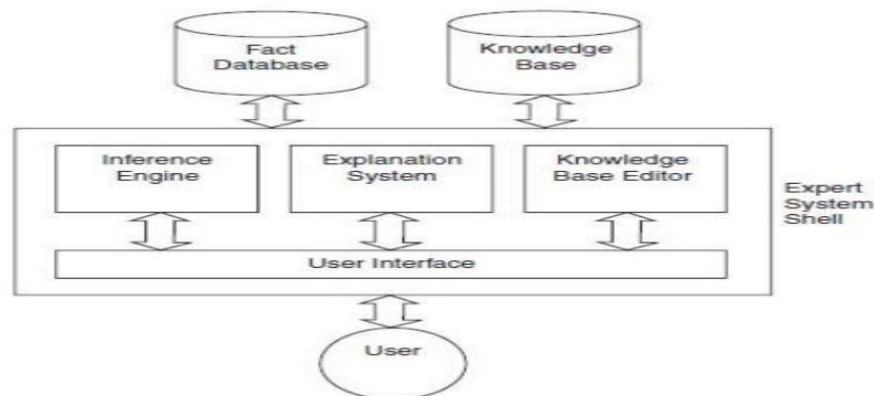


Fig Expert System Architecture

If the system has used faulty reasoning to arrive at its conclusions, then the user may be able to see this by examining the data given by the explanation system.

The fact database contains the case-specific data that are to be used in a particular case to derive a conclusion.

In the case of a medical expert system, this would contain information that had been obtained about the patient's condition.

The user of the expert system interfaces with it through a user interface, which provides access to the inference engine, the explanation system, and the knowledge-base editor.

The inference engine is the part of the system that uses the rules and facts to derive conclusions. The inference engine will use forward chaining, backward chaining, or a combination of the two to make inferences from the data that are available to it.

The knowledge-base editor allows the user to edit the information that is contained in the knowledge base.

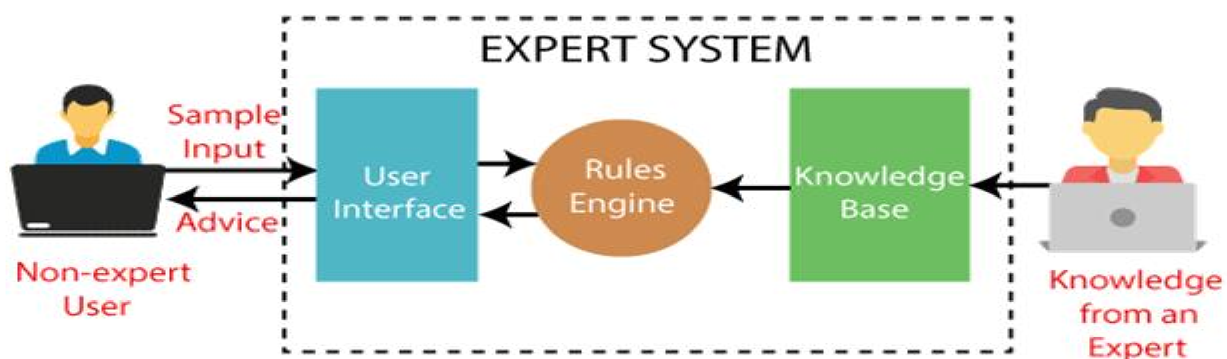
The knowledge-base editor is not usually made available to the end user of the system but is used by the knowledge engineer or the expert to provide and update the knowledge that is contained within the system.

Roles of expert systems

An expert system is a computer program that is designed to solve complex problems and to provide decision-making ability like a human expert. It performs this by extracting knowledge from its knowledge base using the reasoning and inference rules according to the user queries.

The expert system is a part of AI, and the first ES was developed in the year 1970, which was the first successful approach of artificial intelligence. It solves the most complex issue as an expert by extracting the knowledge stored in its knowledge base. The system helps in decision making for compsex problems using **both facts and heuristics like a human expert**. It is called so because it contains the expert knowledge of a specific domain and can solve any complex problem of that particular domain. These systems are designed for a specific domain, such as **medicine, science,** etc.

The performance of an expert system is based on the expert's knowledge stored in its knowledge base. The more knowledge stored in the KB, the more that system improves its performance. One of the common examples of an ES is a suggestion of spelling errors while typing in the Google search box.



Below are some popular examples of the Expert System:

- **DENDRAL:** It was an artificial intelligence project that was made as a chemical analysis expert system. It was used in organic chemistry to detect unknown organic molecules with the help of their mass spectra and knowledge base of chemistry.
- **MYCIN:** It was one of the earliest backward chaining expert systems that was designed to find the bacteria causing infections like bacteraemia and meningitis. It was also used for the recommendation of antibiotics and the diagnosis of blood clotting diseases.
- **PXDES:** It is an expert system that is used to determine the type and level of lung cancer. To determine the disease, it takes a picture from the upper body, which looks like the shadow. This shadow identifies the type and degree of harm.
- **CaDeT:** The CaDet expert system is a diagnostic support system that can detect cancer at early stages.

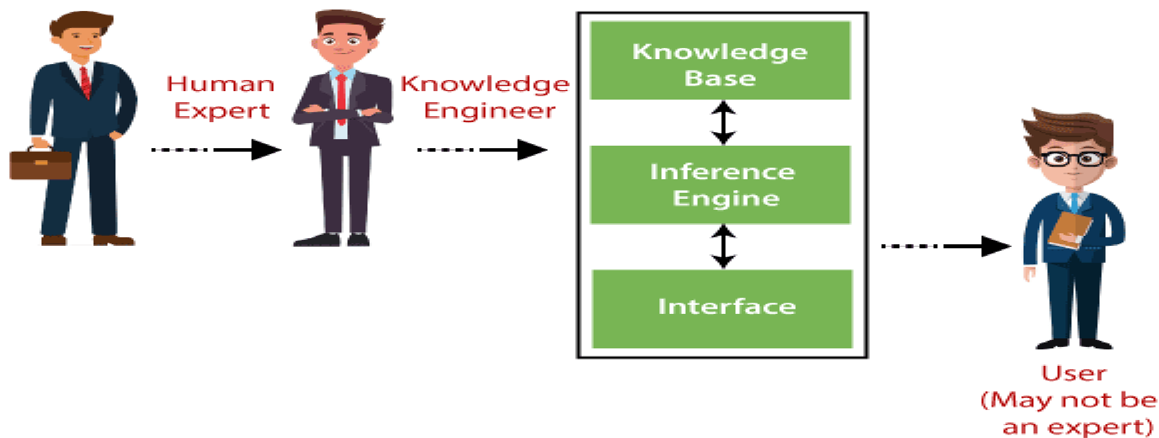
Characteristics of Expert System

- **High Performance:** The expert system provides high performance for solving any type of complex problem of a specific domain with high efficiency and accuracy.
- **Understandable:** It responds in a way that can be easily understandable by the user. It can take input in human language and provides the output in the same way.
- **Reliable:** It is much reliable for generating an efficient and accurate output.
- **Highly responsive:** ES provides the result for any complex query within a very short period of time.

Components of Expert System

An expert system mainly consists of three components:

- **User Interface**
- **Inference Engine**
- **Knowledge Base**



1. User Interface

With the help of a user interface, the expert system interacts with the user, takes queries as an input in a readable format, and passes it to the inference engine. After getting the response from the inference engine, it displays the output to the user. In other words, **it is an interface that helps a non-expert user to communicate with the expert system to find a solution**.

2. Inference Engine(Rules of Engine)

- The inference engine is known as the brain of the expert system as it is the main processing unit of the system. It applies inference rules to the knowledge base to derive a conclusion or deduce new information. It helps in deriving an error-free solution of queries asked by the user.
- With the help of an inference engine, the system extracts the knowledge from the knowledge base.
- There are two types of inference engine:
- **Deterministic Inference engine:** The conclusions drawn from this type of inference engine are assumed to be true. It is based on **facts** and **rules**.
- **Probabilistic Inference engine:** This type of inference engine contains uncertainty in conclusions, and based on the probability.

Inference engine uses the below modes to derive the solutions:

- **Forward Chaining:** It starts from the known facts and rules, and applies the inference rules to add their conclusion to the known facts.
- **Backward Chaining:** It is a backward reasoning method that starts from the goal and works backward to prove the known facts.

---

3. Knowledge Base

- The knowledgebase is a type of storage that stores knowledge acquired from the different experts of the particular domain. It is considered as big storage of knowledge. The more the knowledge base, the more precise will be the Expert System.
- It is similar to a database that contains information and rules of a particular domain or subject.
- One can also view the knowledge base as collections of objects and their attributes. Such as a Lion is an object and its attributes are it is a mammal, it is not a domestic animal, etc.

Applications of Expert System

In designing and manufacturing domain

It can be broadly used for designing and manufacturing physical devices such as camera lenses and automobiles.

In the knowledge domain

These systems are primarily used for publishing the relevant knowledge to the users. The two popular ES used for this domain is an advisor and a tax advisor.

In the finance domain

In the finance industries, it is used to detect any type of possible fraud, suspicious activity, and advise bankers that if they should provide loans for business or not.

In the diagnosis and troubleshooting of devices

In medical diagnosis, the ES system is used, and it was the first area where these systems were used.

Planning and Scheduling

The expert systems can also be used for planning and scheduling some particular tasks for achieving the goal of that task.

Knowledge Acquisition

In artificial intelligence, knowledge acquisition is the process of gathering, selecting, and interpreting information and experiences to create and maintain knowledge within a specific domain. It is a key component of machine learning and knowledge-based systems.

There are many different methods of knowledge acquisition, including rule-based systems, decision trees, artificial neural networks, and fuzzy logic systems. The most appropriate method for a given application depends on the nature of the problem and the type of data available.

Rule-based systems are the simplest form of knowledge-based system. They use a set of rules, or heuristics, to make decisions. Decision trees are another common method, which use a series of if-then-else statements to arrive at a decision.

Artificial neural networks are a more complex form of knowledge-based system, which mimic the way the human brain learns. They are able to learn from data and make predictions based on that data. Fuzzy logic systems are another type of complex knowledge-based system, which use fuzzy set theory to make decisions.

The most important part of knowledge acquisition is the interpretation of information. This is where human expertise is required. Machines are not able to interpret information in the same way humans can. They can only make sense of data if it is presented in a certain way.

Humans need to select the right data and experiences to create knowledge. They also need to interpret that data correctly. This is where artificial intelligence can help. AI systems can automate the process of knowledge acquisition, making it faster and more accurate.

There are a few methods of knowledge acquisition in AI:

1. Expert systems: In this method, experts in a particular field provide rules and knowledge to a computer system, which can then be used to make decisions or solve problems in that domain.
2. Learning from examples: This is a common method used in machine learning, where a system is presented with a set of training data, and it "learns" from these examples to generalize to new data.
3. Natural language processing: This is a method of extracting knowledge from text data, using techniques like text mining and information extraction.
4. Semantic web: The semantic web is a way of representing knowledge on the internet using standards like RDF and OWL, which can be processed by computers.

5. Knowledge representation and reasoning: This is a method of representing knowledge in a formal way, using logic or other formalisms, which can then be used for automated reasoning.

META KNOWLEDGE & HEURISTICS

META KNOWLEDGE

Meta-knowledge is knowledge about knowledge. The term is used to describe things such as tags, models and taxonomies that describe knowledge.

- KNOWLEDGE is a theoretical or practical understanding of a subject or a domain.
- It is the sum of what is currently known and apparently knowledge is power.
- Those who poses knowledge are called the experts.
- Anyone can be considered as a domain expert if they have a deep knowledge and strong practical experience in particular domain.
- The area of domain may be limited.
- An EXPERT is a skillful person who CAN do things that other people CANNOT

RULE AS A KNOWLEDGE REPRESENTATION TECHNIQUE

- The human mental process is internal and it is too complex to be represented as an ALGORITHM
- Most EXPERTS are capable of expressing their knowledge in the form of RULES for problem solving.

IF the traffic light is green

THEN the action is go

IF the traffic light is red

THEN the action is stop

- The term rule in AI, is the most commonlyNOT type of knowledge representation, can be defined as an IF-THEN structure.
- It relates the information or facts in the IF part to some action in the THEN part.
- A rule provides some description of how to solve a problem.
- Rules are relatively easy to create an understand.
- Any rule consists of 2 parts:

1. IF part = ANTECEDANT (Premise or condition)

2. THEN part = CONSEQUENT (conclusion or action)

- The Antecedent of a rule incorporates an object and its value.
- The object & value are linked by an operator.
- The operator identifies the object and assigns the value.
- Operators includes IS, ARE, IS NOT, ARE NOT, etc.

HEURISTICS

- In an expert system, the computer exercises judgement, and it does so by using certain intellectual rules of thumb, a process known as HEURISTICS.
- A broad range of well structured problems embracing forms of diagnosis, catalog, selection and skeletal planning are solved in expert systems by the method of heuristics classification.
- The heuristic classification problem solving model provides a useful framework for characterizing kinds of problems, for designing representational tools and understanding them.

Heuristics Classification Method

1. Simple Classification
2. Data Abstraction

Examples of Heuristic Classification

SACON :

Problem : It selects classes of behavior that should be further investigated by a structural analysis simulation program.

Discussion : It solves 2 problems by classification heuristically analyzing a structure and then using simple classification to select a program.

GRUNDY :

Problem : It is a model of a librarian, selecting books a person might like to read.

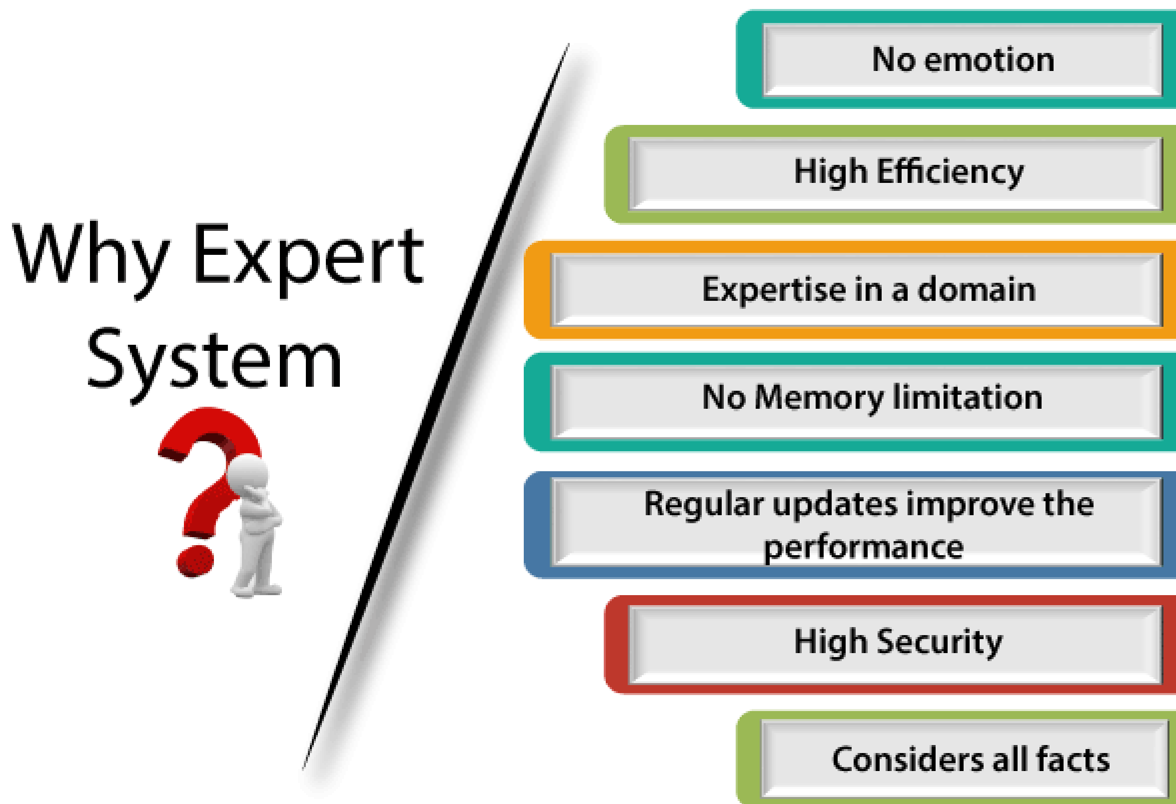
Discussion : It solves 2 classification problem heuristically by classifying a readers personality and then selecting books appropriate to the kind of person in need.

Typical expert systems

Participants in the development of Expert System

There are three primary participants in the building of Expert System:

1. **Expert:** The success of an ES much depends on the knowledge provided by human experts. These experts are those persons who are specialized in that specific domain.
2. **Knowledge Engineer:** Knowledge engineer is the person who gathers the knowledge from the domain experts and then codifies that knowledge to the system according to the formalism.
3. **End-User:** This is a particular person or a group of people who may not be experts, and working on the expert system needs the solution or advice for his queries, which are complex.



Before using any technology, we must have an idea about why to use that technology and hence the same for the ES. Although we have human experts in every field, then what is the need to develop a computer-based system. So below are the points that are describing the need of the ES:

1. **No memory Limitations:** It can store as much data as required and can memorize it at the time of its application. But for human experts, there are some limitations to memorize all things at every time.
2. **High Efficiency:** If the knowledge base is updated with the correct knowledge, then it provides a highly efficient output, which may not be possible for a human.

3. **Expertise in a domain:** There are lots of human experts in each domain, and they all have different skills, different experiences, and different skills, so it is not easy to get a final output for the query. But if we put the knowledge gained from human experts into the expert system, then it provides an efficient output by mixing all the facts and knowledge
4. **Not affected by emotions:** These systems are not affected by human emotions such as fatigue, anger, depression, anxiety, etc.. Hence the performance remains constant.
5. **High security:** These systems provide high security to resolve any query.
6. **Considers all the facts:** To respond to any query, it checks and considers all the available facts and provides the result accordingly. But it is possible that a human expert may not consider some facts due to any reason.
7. **Regular updates improve the performance:** If there is an issue in the result provided by the expert systems, we can improve the performance of the system by updating the knowledge base.

MYCIN Expert System:

MYCIN was an early expert system developed at Stanford University in 1972 to assist physicians in diagnosing and selecting treatment for bacterial and blood infections. It used over 600 production rules encoding the clinical decision criteria of infectious disease experts to diagnose patients based on reported symptoms and test results. While it could not replace human diagnosis due to computing limitations at the time, MYCIN demonstrated that expert knowledge could be represented computationally and established a foundation for more advanced machine learning and knowledge base systems.

The system was actually not used in clinical practice, but it constitutes an excellent early example of a digital expert system and a precursor to much more sophisticated machine learning and knowledge base systems years later. MYCIN, an early expert system, or artificial intelligence (AI) program, designed to assist physicians in the diagnosis of and therapy selection for treating patients with bacterial and blood infections.

The system was actually not used in clinical practice, but it constitutes an excellent early example of a digital expert system and a precursor to much more sophisticated machine learning and knowledge base systems years later.

In 1972 work began on MYCIN at Stanford University in California
The name was chosen after attempts at finding a suitable acronym failed. The name is the common suffix associated with many antimicrobial agents. Examples of aminoglycosides include Gentamicin, tobramycin, neomycin, and streptomycin
attempt to diagnose patients based on reported symptoms and medical test results. If requested,

MYCIN would explain the reasoning that led to its diagnosis and recommendation. The program could request further information concerning the patient, as well as suggest additional laboratory tests, to arrive at a probable diagnosis, after which it would recommend a course of treatment. If requested, MYCIN would explain the reasoning that led to its diagnosis and recommendation.
Much of MYCIN's power derives from the modular, highly stylized nature of these decision rules, enabling the system to dissect its own reasoning and allowing easy modification of the knowledge base.υto the consultation system itself, MYCIN contains an explanation system which can answer simple English questions in order to justify its advice or educate the user.

Much of MYCIN's power derives from the modular, highly stylized nature of these decision rules, enabling the system to dissect its own reasoning and allowing easy modification of the knowledge base. system's knowledge is encoded in the form of some 600 production rules which embody the clinical decision criteria of infectious disease experts.

Users would enter answers to a series of “yes” or “no” questions and short answer questions, and the program would eventually choose a weighted probability for a diagnosis.
he limitation of this early program was simply computing power – because the program was estimated to take up to half an hour to get through in a clinical environment, it was not considered effective enough to replace human diagnosis at the time.

Ethical questions also contributed to the decision not to use Mycin for clinical diagnosis.

proven to be a stepping stone to more modern systems and described in a book on rule-based expert systems by B. G. Buchanan and E. H. Shortliffe as “the granddaddy of them all” in terms of early artificial intelligence for machine learning systems.

Dart Expert System

- These experts are expensive and often not immediately available, and there is inevitably a delay and a loss of working time to the system users. Thus, in some cases, it is necessary to call in someone with expert knowledge about the design of the system. Unfortunately, these diagnostics are not infallible for they do not take into account every fault and combination of faults for every possible system configuration. In anticipation of computer faults, most manufacturers prepare specialized diagnostic aids that allow field engineers without complete knowledge of a system to diagnose the majority of its failures. υfor automated computer fault diagnosis
- The practical goal is the construction of an automated diagnostician capable of pinpointing the functional units responsible for observed malfunctions in arbitrary system configurations. The primary goal of the DART Project is to develop programs that capture the special design knowledge and diagnostic abilities of these experts and to make them available to field engineers. a joint project of the Heuristic Programming Project and IBM that explores the application of artificial intelligence techniques to the diagnosis of computer faults.
- DART achieved logistical solutions that surprised many military planners. It integrates a set of intelligent data processing agents and database management systems to give planners the ability to rapidly evaluate plans for logistical feasibility. By automating evaluation of these processes
- DART decreases the cost and time required to implement decisions. DART uses intelligent agents to aid decision support systems located at the U.S. Transportation and European Commands. an artificial intelligence program used by the U.S. military to optimize and schedule the transportation of supplies or personnel and solve other logistical problems. DART uses intelligent agents to aid decision support systems located at the U.S. Transportation and European Commands. It integrates a set of intelligent data processing agents and database management systems to give planners the ability to rapidly evaluate plans for logistical feasibility. By automating evaluation of these processes DART decreases the cost and time required to implement decisions. DART achieved logistical solutions that surprised many military planners.

XCON is a Rule-based system written in OPS5 in 1978 to assist in the ordering of DEC's VAX computer systems by automatically selecting the computer system components based on the customer's requirements.

In 1975 DEC offered 50 types of central processors with 400 core operations. The estimated possible number of configurations at that time was already in millions. So system configuration was the key process in DEC`s flexibility strategy for it converted a customer order into fully configured system that was designed, checked and ready for delivery. This process involves three separate reviews of each order.

- The first two steps relied upon highly skilled and talented technical editors who learned their craft through a long apprenticeship.
- The final review was FA & T (Fast Assembly and Test) which is nothing but actual assembly of the system prior to delivery.

Elapsed time from signed order to delivery was ten to fifteen weeks, extending at times even up to six months. Computers too were growing more complex, increasing even further the number of configuration options. DEC had to find a new way to configure its order and so XCON DEC`s configuration system was born.

Functions of XCON System:

XCON is used to configure customer orders and to guide the assembly of these orders at the customer site. Using the customer order as input, it provides the following functionality:

- Configures CPUs, memory, cabinets, power supplies, disk, tapes and so on.
- Determines and list cabling information.
- List components ordered with configuration related comments.
- Generate warning messages on issues affecting technical validity.

Scope of XCON

The initial purpose of XCON was to assist manufacturing plant personnel in validating the technical correctness of system orders about to be filled. It is now used by a broad set of users across the company's major functions like sales and marketing, manufacturing and production, field service and engineering. The users of these systems perform functions that span Digital`s

complete order flow and manufacturing cycle. Thus they are involved with many different business processes. Each has different needs and takes a different perspective on the configuration information provided.

- Sales uses the configuration systems as an integral part of the automated process to generate quotations for customer
- Manufacturing uses the information to verify buildability of all incoming orders.
- Field service has the perspective of assembling the order in the customer's unique environment.
- Manufacturing and engineering benefits from the configuration system`s focus on system integration and can identify potential problems in system-level design and manufacturability.

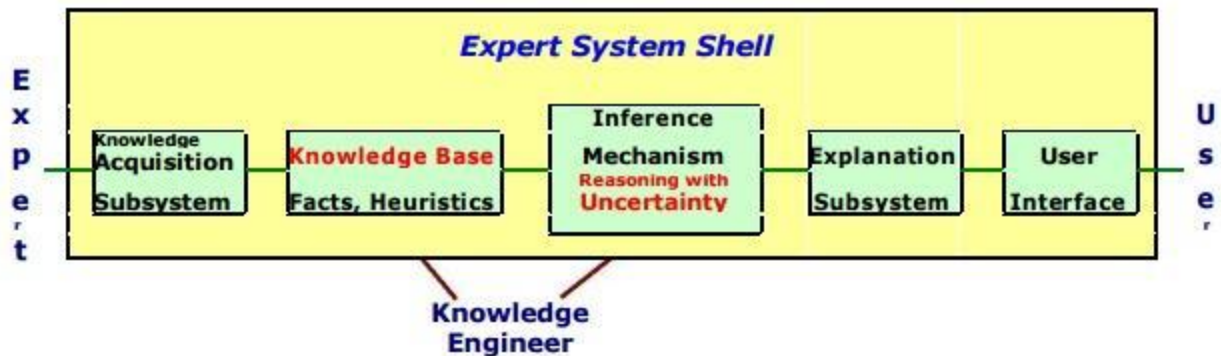
Using XCON DEC was able to eliminate FA & T and to reduce average shipping time to three or four weeks, sometimes even to only two or three days. As the computer market changed in the 1980 a rapid turnaround of orders became a requirement for survival. Because of XCON, DEC was ready. These system allowed DEC to remain competitive and even to thrive in the 1980s as DEC`s sales mushroomed as the market demanded shorter response time. DEC is highly dependent on its configuration system. It considers them to be a solid success for the company by contributing to customer satisfaction, lower costs and higher productivity.

Expert System Shells

An Expert system shell is a software development environment. It contains the basic components of expert systems. A shell is associated with a prescribed method for building applications by configuring and instantiating these components.

Shell components and description

The generic components of a shell : the knowledge acquisition, the knowledge Base, the reasoning, the explanation and the user interface are shown below. The knowledge base and reasoning engine are the core components.



All these components are described below.

■ Knowledge Base

A store of factual and heuristic knowledge. Expert system tool provides one or more knowledge representation schemes for expressing knowledge about the application domain. Some tools use both Frames (objects) and IF-THEN rules. In PROLOG the knowledge is represented as logical statements.

■ Reasoning Engine

Inference mechanisms for manipulating the symbolic information and knowledge in the knowledge base form a line of reasoning in solving a problem. The inference mechanism can range from simple modus ponens backward chaining of IF-THEN rules to Case-Based reasoning.

Knowledge Acquisition subsystem

A subsystem to help experts in build knowledge bases. However, collecting knowledge, needed to solve problems and build the knowledge base, is the biggest bottleneck in building expert systems.

Explanation subsystem

A subsystem that explains the system's actions. The explanation can range from how the final or intermediate solutions were arrived at justifying the need for additional data.

User Interface

A means of communication with the user. The user interface is generally not a part of the expert system technology. It was not given much attention in the past. However, the user interface can make a critical difference in the pe eived utility of an Expert system.