INTRODUCTION TO ARTIFICIAL INTELLIGENCE

UNIT 01

SCOPE OF AI

Intelligence:

- The ability to learn, understand, and make judgments or have opinions that are based on reason is known as intelligence.
- Intelligence is a multifaceted concept encompassing the ability to learn from experience, adapt to new situations, understand and handle abstract concepts, and utilize knowledge to manipulate one's environment. It's not limited to a single domain but spans various cognitive abilities.

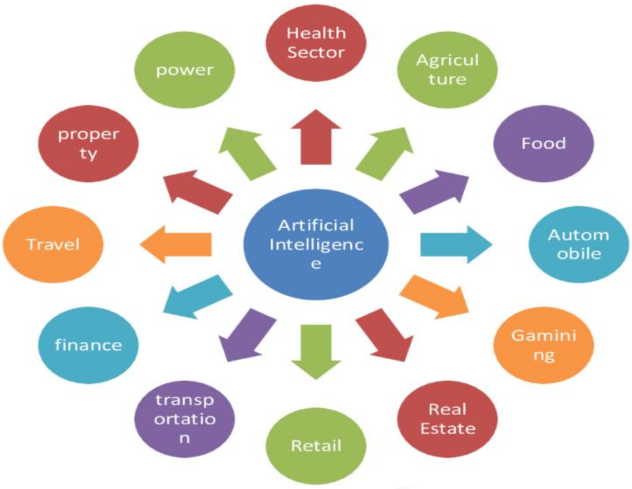
Artificial Intelligence (AI):

- AI is the branch of computer science which deals with helping machines find solutions to complex problems in a more human-like fashion.
- AI is techniques that help machines mimic human behaviour.
- AI is the branch of computer science with which we can create intelligent machine which can think like human, able to take decision and behave like human.
- The term was coined by John McCarthy in 1956.
- Artificial Intelligence (AI) is the branch of computer science dedicated to creating systems capable of performing tasks that typically require human intelligence. These tasks include learning from experience, recognizing patterns, understanding natural language, making decisions, and solving problems.
- AI systems achieve this through various techniques and algorithms, enabling them to process information, adapt to new situations, and carry out complex operations autonomously.
- The ultimate goal of AI is to develop machines that can mimic cognitive functions such as reasoning, perception, and action, thereby enhancing human capabilities and automating routine tasks.

Examples:

- Gmail's spam filter: Uses AI to detect and filter out unwanted and harmful emails.
- Smart assistants like Alexa, Siri: Utilize AI to understand voice commands and perform tasks or answer questions.
- Recommendation systems like Netflix, Spotify: Leverage AI to suggest content based on user preferences and behavior.
- Smartphone features like face recognition, autocorrect, voice to text: Employ AI to enhance security, improve typing accuracy, and convert spoken words to text.
- Navigation and traffic prediction: Use AI to provide real-time route planning and traffic updates.
- Social media: Implement AI to personalize feeds, detect fake news, and filter inappropriate content.
- Online shopping: Utilize AI to recommend products, optimize search results, and personalize the shopping experience.
- Photo organization: Uses AI to categorize and tag images based on content and facial recognition.
- Smart home devices: Employ AI to automate and control home appliances, improving convenience and energy efficiency.

Applications of AI:



- **Health Sector**:
  - AI enhances diagnostic accuracy through medical imaging analysis and predictive analytics.
  - It enables personalized treatment plans and improves patient outcomes by analyzing large datasets from electronic health records.
  - AI-powered chatbots and virtual health assistants provide 24/7 patient support and streamline administrative tasks.
- **Agriculture**:
  - AI optimizes crop management by analyzing data from sensors and drones to monitor soil health, weather conditions, and crop growth.
  - It helps in pest and disease prediction, allowing farmers to take proactive measures to protect crops.
  - Automated machinery and robotics driven by AI improve planting, harvesting, and sorting processes.
- **Food**:
  - AI improves food safety by detecting contaminants and ensuring compliance with safety standards through real-time monitoring.
  - It enhances supply chain efficiency by predicting demand, reducing waste, and optimizing inventory.
  - Personalized nutrition plans are developed using AI to analyze individual dietary needs and health data.
- **Automobile**:
  - AI powers autonomous driving systems, enabling vehicles to navigate and make decisions without human intervention.
  - It enhances driver assistance features such as collision avoidance, lane-keeping, and adaptive cruise control.
  - AI optimizes manufacturing processes and supply chain logistics in the automotive industry.
- **Gaming**:
  - AI creates more realistic and responsive non-player characters (NPCs), enhancing the overall gaming experience.
  - It personalizes gaming environments and difficulty levels based on player behavior and preferences.
  - AI-driven analytics help developers understand player engagement and improve game design.
- **Real Estate**:
  - AI predicts market trends and property values by analyzing historical data and current market conditions.
  - It automates property management tasks such as tenant screening, lease management, and maintenance scheduling.
  - Virtual reality and AI-powered tours provide immersive property viewing experiences for potential buyers.
- **Retail**:
  - AI personalizes shopping experiences by recommending products based on customer behavior and preferences.

- It optimizes inventory management by predicting demand and ensuring timely restocking of popular items.
- AI-powered chatbots and virtual assistants enhance customer service by providing instant support and resolving queries.

- **Transportation**:
  - AI optimizes routing and scheduling for logistics companies, reducing delivery times and operational costs.
  - It predicts maintenance needs for vehicles and infrastructure, improving safety and reliability.
  - AI enhances traffic management systems, reducing congestion and improving urban mobility.
- **Finance**:
  - AI detects fraudulent transactions by analyzing patterns and anomalies in financial data.
  - It automates trading strategies, making real-time investment decisions based on market data.
  - Personalized financial advice is provided by AI, helping individuals and businesses manage their finances effectively.
- **Travel**:
  - AI personalizes travel recommendations and itineraries based on user preferences and past behaviors.
  - It optimizes booking processes by predicting flight prices and availability, ensuring the best deals for travelers.
  - AI-powered virtual assistants provide real-time support for travelers, enhancing their overall experience.
- **Property**:
  - AI assists in property valuation by analyzing market trends, historical data, and property characteristics.
  - It predicts future market conditions and investment opportunities, helping buyers and investors make informed decisions.
  - Automated property management solutions streamline tasks such as rent collection, maintenance requests, and tenant communication.
- **Power**:
  - AI optimizes energy consumption in smart grids by predicting demand and adjusting supply accordingly.
  - It improves the efficiency of renewable energy sources by forecasting weather conditions and energy production.
  - AI predicts equipment maintenance needs, preventing failures and ensuring a reliable power supply.

AI techinques:

- It refers to a set of methods and algorithms used to develop intelligent systems that can perform tasks requiring human-like intelligence.
- AI techniques are the methods and approaches used to enable machines to simulate human intelligence and perform tasks that require cognitive functions.
- These techniques encompass a wide range of algorithms and methodologies that facilitate learning, reasoning, perception, and decision-making.

Here are some AI techniques:

1. Machine Learning (ML)

- Machine Learning is a subset of AI that involves the development of algorithms that enable computers to learn from and make predictions or decisions based on data.

- It is a branch of artificial intelligence based on the idea that systems can learn from data, identify patterns and make decisions with minimal human intervention.
- This machine learning process starts with feeding them good quality data and then training the machines by building various machine learning models using the data and different algorithms.
- The choice of algorithms depends on what type of data we have and what kind of task we are trying to automate.
- This approach involves the building of algorithms to learn patterns in data and make predictions based on it.
  Ex: Personalized Recommendations

ML is categorized into Four main types:

1. **Supervised Learning**:
   - Algorithms are trained on labeled data, meaning the input comes with corresponding output.
   - A combination of an input data set and the intended output is inferred from the training data. AI systems learn from a labelled dataset, where each data point is associated with a known outcome.
   - For instance, it enables email spam filters to distinguish between spam and legitimate emails based on learned patterns.
   - Common algorithms include Linear Regression, Decision Trees, and Support Vector Machines (SVM).
   - Example: Predicting house prices based on historical data of house sales.
2. **Unsupervised Learning**:
   - Algorithms are used on data without labeled responses, aiming to find hidden patterns or intrinsic structures.
   - AI systems analyse unlabelled data, where no predefined outcomes are provided.
   - The objective is to uncover inherent structures or patterns within the data without any prior knowledge.
   - For instance, it can group similar customer behaviour data to identify customer segments for targeted marketing strategies.
   - Common techniques include Clustering (e.g., K-Means) and Association Rules.
   - Example: Market basket analysis to find product groupings in retail.
3. **Semi-supervised Learning**:
   - It is a method that uses a small amount of labelled data and a large amount of unlabelled data to train a model.
   - The goal of semi-supervised learning is to learn a function that can accurately predict the output variable based on the input variables, similar to supervised learning.
   - However, unlike supervised learning, the algorithm is trained on a dataset that contains both labelled and unlabelled data.
4. **Reinforcement Learning**:
   - Algorithms learn by interacting with an environment, receiving rewards or penalties based on actions taken.
   - In RL, the data is accumulated from ML systems that use a trial-and-error method to learn from outcomes and decide which action to take next. After each action, the algorithm receives feedback that helps it determine whether the choice it made was correct, neutral or incorrect.
   - It performs actions with the aim of maximizing rewards, or in other words, it is learning by doing in order to achieve the best outcomes.

  - It involves agents that make sequences of decisions.
  - Example: Training a robot to navigate a maze by rewarding it for reaching the exit.

2. Deep Learning (DL)

- Deep Learning is a subset of ML that uses neural networks with many layers (hence "deep") to model complex patterns in large datasets. It excels in tasks like image and speech recognition.
- It is a method in AI that teaches computers to process data in a way that is inspired by the human brain.
  Deep learning models can recognize complex patterns in pictures, text, sounds, and other data to produce accurate insights and predictions.
- This transformative field has propelled breakthroughs across various domains, from computer vision and natural language processing to healthcare diagnostics and autonomous driving.
  Ex: Image classification
- **Neural Networks**:
  - Composed of layers of interconnected nodes (neurons) that process input data to recognize patterns.
  - Variants include Convolutional Neural Networks (CNNs) for image processing and Recurrent Neural Networks (RNNs) for sequential data.

3. Natural Language Processing (NLP)

- NLP is a field of AI focused on the interaction between computers and humans through natural language.
- NLP is a machine learning technology that gives computers the ability to interpret, manipulate, and comprehend human language.
- NLP is the ability of a computer program to understand human language as it's spoken and written referred to as natural language.
- NLP is widely used in Text classification, text extraction, machine translation, NL generation.
- Real world examples are, Customer feedback analysis, customer service automation, automatic translation, academic research and analysis, categorization of healthcare record, plagiarism detection etc..

4. Computer Vision

- Computer Vision is an AI field that enables machines to interpret and make decisions based on visual data.
- A field of AI that deals with the processing and analysis of visual information using computer algorithms.
- It is a field of AI that trains computers to capture and interpret information from image and video data.
- By applying ML models to images, computers can classify objects and respond—like unlocking your smartphone when it recognizes your face.
- If AI enables computers to think, computer vision enables them to see, observe and understand.

5. Expert Systems

- Expert Systems emulate the decision-making abilities of a human expert.

- It is a computer program that is designed to solve complex problems and to provide decision-making ability like a human expert.
- An expert system's job is to resolve the trickiest problems in a specific field.
- It performs this by extracting knowledge from its knowledge base using the reasoning and inference rules according to the user queries.
- Popular examples are,
  - ❖ 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.
- **Rule-Based Systems**: Use predefined rules to make inferences and solve specific problems.
- **Inference Engines**: Apply logical rules to the knowledge base to deduce new information.

6. Robotics

- Robotics combines AI with physical robots to perform tasks autonomously or semi-autonomously.
- It involves integrating AI technologies into robotic systems to enhance their capabilities and enable them to perform more complex tasks.
- AI in robotics allows robots to learn from experience, adapt to new situations, and make decisions based on data from sensors.
- Robotics deals with the design, construction, operation, and use of robots and computer systems for their control, sensory feedback, and information processing.
- A robot is a unit that implements this interaction with the physical world based on sensors, actuators, and information processing.
- Robotics is used in defence sector, medical sector, industrial sector, entertainment, mining industry and so on.
- **Path Planning**: Algorithms that determine the optimal path for a robot to take.
- **Control Systems**: Methods for managing the movements and actions of robots.

7. Fuzzy Logic

Fuzzy Logic deals with reasoning that is approximate rather than fixed and exact.

- **Fuzzy Sets**: Mathematical representations of vagueness, allowing for degrees of membership.
- **Fuzzy Inference Systems**: Systems that map inputs to outputs using fuzzy logic principles.

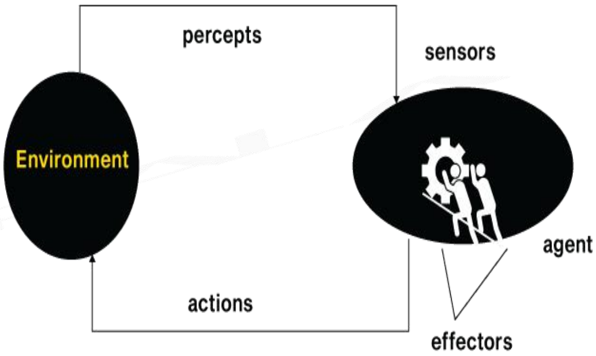
8. Genetic Algorithms

Genetic Algorithms are optimization techniques inspired by natural selection.

- **Selection**: Choosing the best-fit individuals from a population.
- **Crossover**: Combining pairs of individuals to create offspring.
- **Mutation**: Randomly altering individuals to maintain genetic diversity.

AI AGENTS

- An agent can be some independent program or entity that perceive environment through sensor and act upon that environment through actuators.
- An AI agent can be defined as a program that makes decisions and takes action based on the decisions.
- In AI, an intelligent agent (IA) is an independent entity which observes and operates upon an environment and directs its activity towards accomplishing goals.
- In short, an intelligent agent is an entity that interacts with its surroundings via:
    - perception through sensors.
    - actions through effectors or actuators(which operate).
- **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.
- **Effectors:** Effectors are the devices which affect the environment. Effectors can be legs, wheels, arms, fingers, wings, fins, and display screen.



Agents can of be 3 types

- ✓ **Human agent:** Humans contain sensors like their eyes, ears and other organs, as well as actuators like their hands, legs, mouth and other bodily parts.
- ✓ **Robot agent:** These agents feature a variety of high quality motors, grippers, wheels, lights, speakers etc that serve as actuators, as well as cameras, infrared range finders, bumper etc that serve as sensors.
- ✓ **Software agent:** This agent acts on sensory inputs(Data given as input to functions in the structure of encoded bit strings or symbols) such as file contents and network packets it has received, by acting on those inputs and with the result on the screen.

Rules for AI agents

Rule 01: An AI agent needs to have an environment perception.
Rule 02: Decisions must be based on observations of the environment.
Rule 03: All the action should be based on decisions.
Rule 04: The AI agent's actions have to be logical.

Types of Agents based on their degree of perceived intelligence & capacity

1. **Simple Reflex agent**
2. **Model based agent**
3. **Goal based agent**

4. **Utility based agent**
5. **Learning agent**

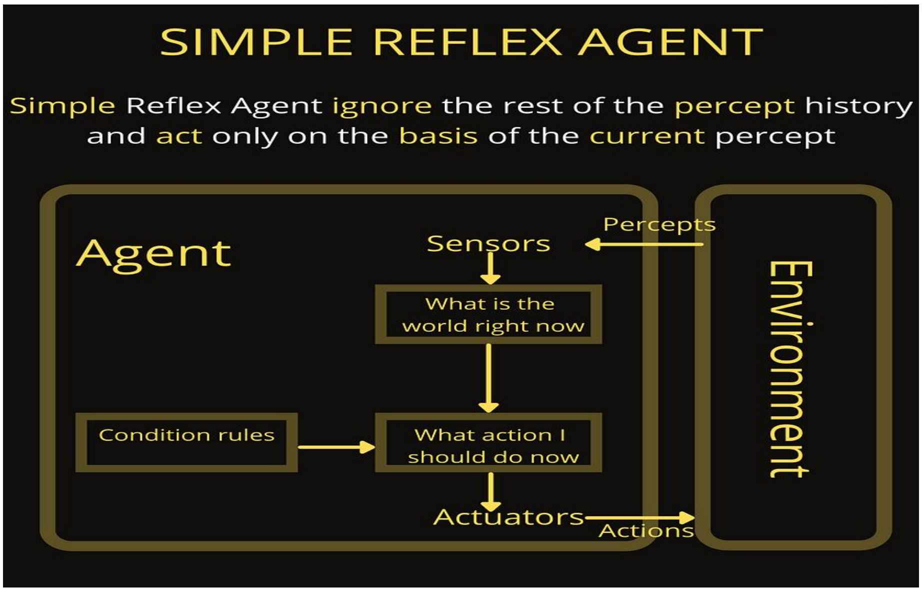
1. Simple reflex agents

- These are the simplest agents.
- These agents take decisions on the basis of the current percepts and ignore the rest of the percept history.
- This type is agent is based upon the condition action rule

If the condition is true
Action is action,
Else NOT

Problems for the simple reflex agent design approach:

- They have very limited intelligence
- They do not have knowledge of non-perceptual (absence of perceiving) parts of the current state

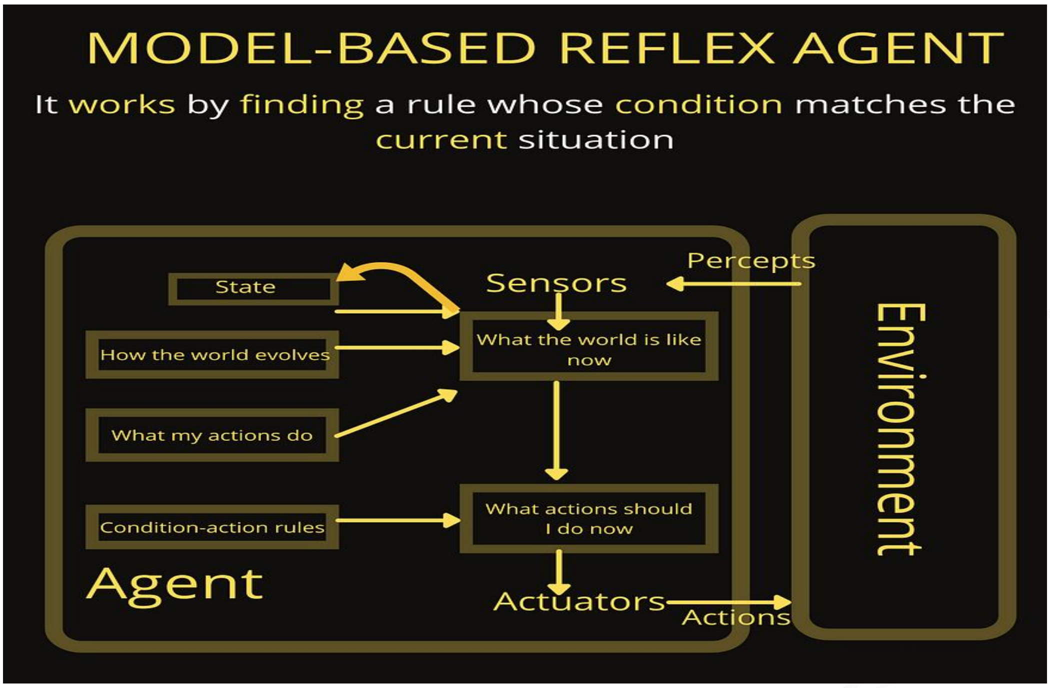


2. Model-based agent

A model-based agent has two important factors:

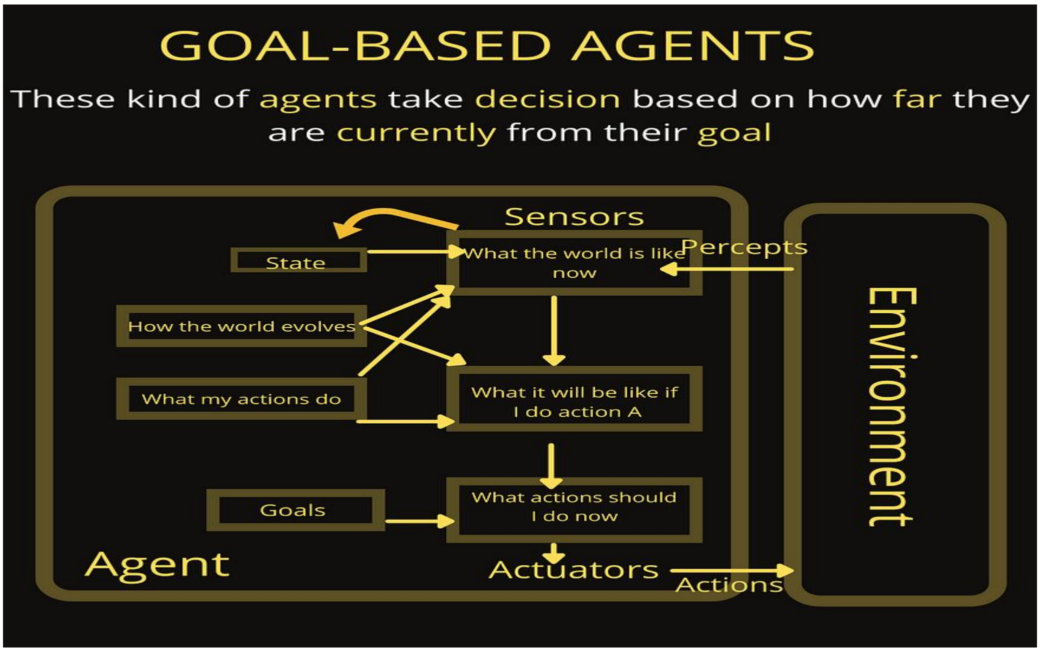
- **Model:** It is knowledge about "how things happen in the world," so it is called a Model-based agent.
- **Internal State:** It is a representation of the current state based on percept history.

- It works by finding a rule whose condition matches the current situation
- It can handle partially observable environment
- Updating the state requires information about
  - How the world evolves
  - How the agent's action affects the world.



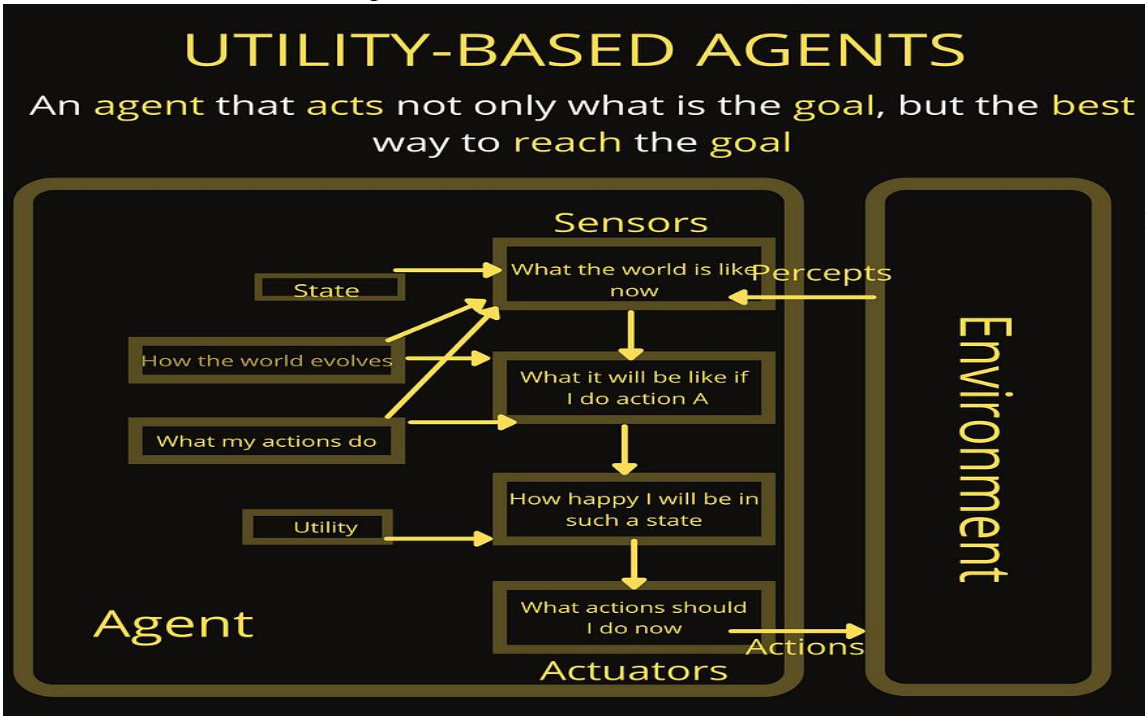
3. Goal-based agents

- 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.
- The goal based agent focuses only on reaching the goal set and hence the decision look by the agent is based on how far it is currently from their goal desired or desired state
- Their every action is intended to minimise their distance from the goal
- This agent is more flexible and the agent develop the decision making skill by choosing the right.



4. Utility-based agent

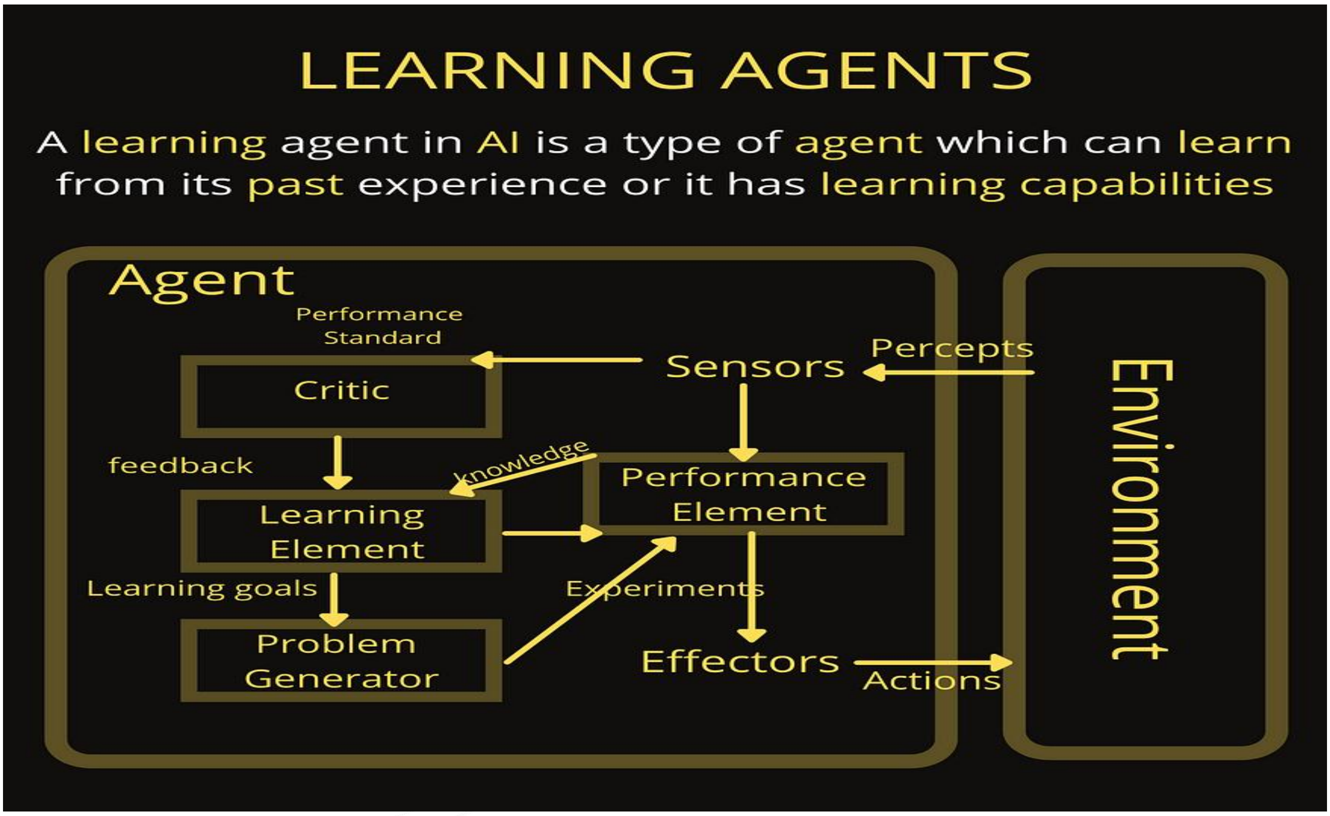
- Utility-based agents used the optimal path which lead to their goal. They choose the best path which leads to their goal among multiple paths.
- These agents are similar to the goal-based agent but provide an extra component of utility measurement which makes them different by providing a measure of success at a given state.
- Utility-based agent act not only based on 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.



5. Learning agent

- 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:
  - **Learning element:** It is responsible for making improvements by learning from environment
  - **Critic:** Learning element takes feedback from critic which describes that how well the agent is doing with respect to a fixed performance standard.
  - **Performance element**: It is responsible for selecting external action
  - **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.



Example : Robotic Vacuum Cleaner

1. Simple Reflex Agent: This basic robotic vacuum cleaner operates on immediate sensory input only. It moves in a straight line until it encounters an obstacle, then changes direction randomly. While simple to implement and quick to respond, it lacks efficiency and may repeatedly clean the same areas while missing others entirely. It has no memory or ability to improve its performance over time.

2. Model-Based Reflex Agent: This vacuum creates and maintains an internal map of its environment. It uses this model to navigate more efficiently, avoiding known obstacles and planning shorter paths. While more effective than the simple reflex agent, it still lacks the ability to set and pursue specific goals or optimize its performance beyond basic navigation.

3. Goal-Based Agent: This type of vacuum cleaner sets a specific goal, such as cleaning the entire floor area. It plans its path to achieve complete coverage, resulting in more systematic and thorough cleaning. However, it may struggle to adapt quickly to changes in the environment and could be computationally more demanding than simpler models.

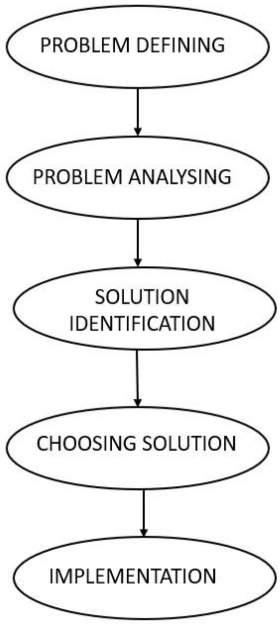
4. Utility-Based Agent: This advanced vacuum considers multiple factors like cleaning efficiency, battery life, and noise levels. It uses a utility function to balance these aspects, optimizing its behavior accordingly. For instance, it might clean high-traffic areas more thoroughly or reduce speed to conserve battery when needed. While highly efficient, designing an appropriate utility function can be complex.

5. Learning Agent: The most sophisticated type, this vacuum improves its performance over time through experience. It adapts its cleaning strategy based on past results, learning the layout and identifying areas that need more frequent attention. While highly adaptable and continually improving, it requires significant computational resources and may exhibit unpredictable behavior during its learning phase.

Problem solving in AI

- Problem-solving is commonly known as the method to reach the desired goal or find a solution to a given situation.
- In computer science, problem-solving refers to various techniques such as forming efficient algorithms, heuristics, and performing root cause analysis to find desirable solutions.
- Problem-solving in AI usually refers to researching a solution to a problem by performing logical algorithms, utilizing polynomial and differential equations, and executing them using modeling paradigms.
- There can be various solutions to a single problem, which are achieved by different heuristics.
- Also, some problems have unique solutions. It all rests on the nature of the given problem.

Steps of Problem Solving in AI



1. **Defining The Problem:**

The problem must be defined precisely, including both initial and desired final states. This step sets the boundaries and goals of the problem-solving process.
Example: In a navigation system, the problem definition might be:
- Initial state: Current location (e.g., 123 Main St)
- Final state: Destination (e.g., 456 Oak Ave)
- Acceptable solution: A route that minimizes travel time while avoiding traffic

2. **Analyzing The Problem:**

This step involves a deep dive into the problem's characteristics, constraints, and potential impacts. It's crucial to identify key features that could significantly affect the solution.
Example: For the navigation problem, analysis might include:
- Available transportation methods (car, public transit, walking)
- Time of day and typical traffic patterns
- Road conditions and construction
- User preferences (e.g., avoid highways, prefer scenic routes)

3. **Identification Of Solutions:**

Generate a range of possible solutions to the problem. This phase should produce multiple viable options within the problem's constraints.
Example: For the navigation problem, identified solutions might include:
- Shortest distance route
- Fastest route based on current traffic
- Route with least turns
- Route prioritizing highways
- Route avoiding toll roads

4. **Choosing a Solution:**

Evaluate the identified solutions based on their effectiveness, efficiency, and alignment with the problem's goals. Select the best solution from the options.

Example: For the navigation problem, choosing might involve:
- Comparing estimated arrival times for each route
- Considering user preferences (e.g., avoid tolls)
- Evaluating fuel efficiency of different routes
- Assessing the reliability of each option based on real-time data

5. **Implementation:**

Put the chosen solution into action. This may involve creating a plan, allocating resources, and executing the solution step by step.

Example: For the navigation problem, implementation would include:
- Displaying the chosen route on the user's device
- Providing turn-by-turn directions
- Offering real-time updates on traffic conditions
- Recalculating if the user deviates from the route

Throughout this process, it's important to note that problem-solving in AI often involves iteration. If the implemented solution doesn't yield satisfactory results, the process may loop back to earlier steps for refinement or to explore alternative solutions.

In complex AI systems, these steps might be automated or augmented by machine learning algorithms. For instance, a reinforcement learning system might continuously refine its problem-solving approach based on feedback from previous solutions, effectively learning to make better choices over time.

Problem solving techniques

Heuristics

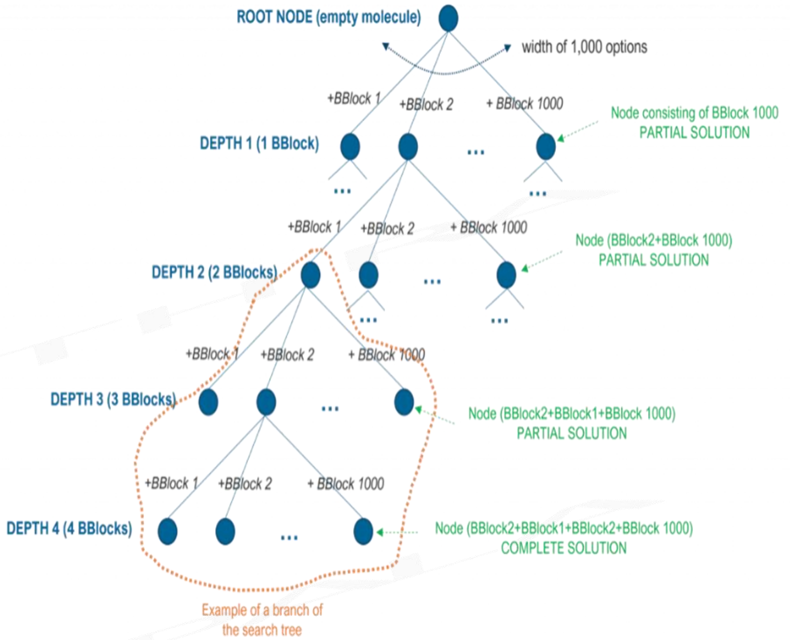
- The heuristic method helps comprehend a problem and devises a solution based purely on experiments and trial and error methods.
- However, these heuristics do not often provide the best optimal solution to a specific problem.
- Instead, these undoubtedly offer efficient solutions to attain immediate goals.
- Therefore, the developers utilize these when classic methods do not provide an efficient solution for the problem.
- Since heuristics only provide time-efficient solutions and compromise accuracy, these are combined with optimization algorithms to improve efficiency.
- Ex: Traveling salesman problem

Searching Algorithms

- Searching is one of the primary methods of solving any problem in AI.
- These problem-solving agents are often goal-based and utilize atomic representation.
- Moreover, these searching algorithms possess completeness, optimality, time complexity, and space complexity properties based on the quality of the solution provided by them.
- A search problem can have three main factors:
  - Search Space: Search space represents a set of possible solutions, which a system may have.
  - Start State: It is a state from where agent begins the search.
  - Goal state: It is a function which observe the current state and returns whether the goal state is achieved or not.

Terms associated:

- **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.
- **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

1. 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.
- This property ensures that if a solution exists, the algorithm will eventually find it.
- Complete algorithms are crucial for problems where finding any solution is critical.
- Example: Breadth-First Search (BFS) is complete for graphs with finite branching factors.

2. 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 is said to be an optimal solution.
- Optimal algorithms ensure finding not just any solution, but the best possible one.
- This is particularly important in resource-constrained environments or when solution quality is critical.
- Example: Dijkstra's algorithm is optimal for finding the shortest path in a weighted graph with non-negative edges.

3. Time Complexity:

Time complexity is a measure of time for an algorithm to complete its task.
- It's typically expressed in Big O notation, indicating how the algorithm's runtime grows with input size.
- Lower time complexity generally indicates faster algorithms, but this can vary based on input characteristics.
- Time complexity helps in comparing algorithms and choosing the most efficient one for a given problem size.
- Example: Quick Sort has an average time complexity of O(n log n), making it efficient for large datasets.

4. Space Complexity:

It is the maximum storage space required at any point during the search, as the complexity of the problem.
- Space complexity is crucial when working with limited memory resources.

- It's also expressed in Big O notation, indicating how memory usage grows with input size.
- Some algorithms trade increased space usage for improved time complexity.
- Example: Depth-First Search (DFS) has a space complexity of O(h), where h is the maximum depth of the search tree.

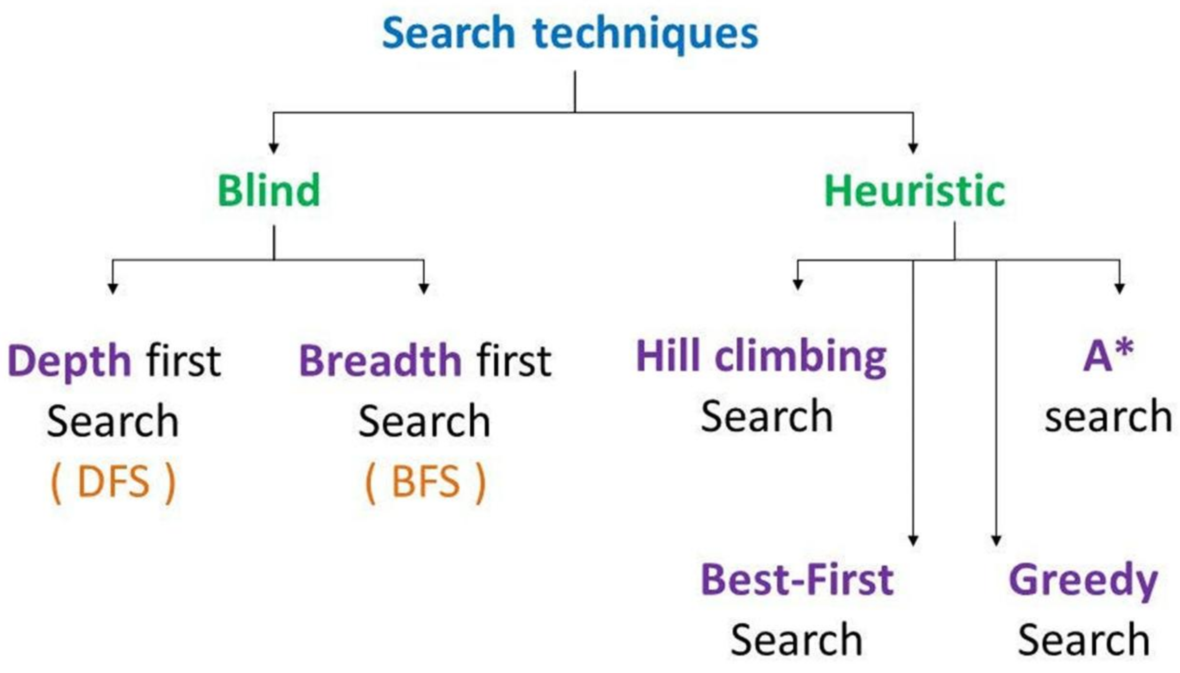
These properties are essential considerations when choosing or designing search algorithms for specific problems. The importance of each property may vary depending on the problem context:

- In some cases, guaranteeing a solution (completeness) might be more important than finding the optimal solution quickly.
- In other scenarios, finding the best possible solution (optimality) might be crucial, even if it takes more time.
- For large-scale problems, time and space complexity become critical factors in determining an algorithm's practicality.

Understanding these properties allows developers and researchers to make informed decisions about which search algorithms to use in different situations, balancing the trade-offs between completeness, optimality, time efficiency, and memory usage.

There are following two main types of searching algorithms:

- **Uninformed Search (Blind)**
- **Informed Search (Heuristics)**



Uninformed/Blind Search

- It is a search algorithm that explores a problem space without any specific knowledge or information about the problem other than the initial state and the possible actions to take.
- It lacks domain-specific heuristics or prior knowledge about the problem.
- Uninformed search applies a way in which search tree is searched without any information, so it is also called **blind search.**

- It examines each node of the tree until it achieves the goal node.

Informed search

- It uses domain knowledge.
- In an informed search, problem information is available which can guide the search.
- Informed search strategies can find a solution more efficiently than an uninformed search strategy.
- Informed search is also called a **Heuristic search**.
- A heuristic is a way which might not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.
- Informed search can solve much complex problem which could not be solved in another way.

Informed Search	Uninformed Search
It is also known as Heuristic Search.	It is also known as Blind Search.
It uses knowledge for the searching process.	It doesn't use knowledge for the searching process.
It finds a solution more quickly.	It finds solution slow as compared to an informed search.
It may or may not be complete.	It is always complete.
Cost is low.	Cost is high.
It consumes less time because of quick searching.	It consumes moderate time because of slow searching.
There is a direction given about the solution.	No suggestion is given regarding the solution in it.
It is less lengthy while implemented.	It is more lengthy while implemented.
It is more efficient as efficiency takes into account cost and performance. The incurred cost is less and speed of finding solutions is quick.	It is comparatively less efficient as incurred cost is more and the speed of finding the Breadth-Firstsolution is slow.
Computational requirements are lessened.	Comparatively higher computational requirements.
Having a wide scope in terms of handling large search problems.	Solving a massive search task is challenging.
Ex: Greedy Search, A* Search, AO* Search, Hill Climbing Algorithm	Ex: Depth First Search (DFS), Breadth First Search (BFS), Branch and Bound

Difference between Informed and uninformed search

Breadth-First Search (BFS)

- BFS is a graph traversal algorithm that systematically explores a graph level by level.
- It starts at a specific node and visits all its immediate neighbors before moving to the next level.
- BFS is widely used in various AI applications, including pathfinding, shortest path problems, and graph analysis.
- A graph is a data structure used to represent relationships between entities.
- It consists of nodes (vertices) representing entities and edges connecting these nodes.

- Imagine a map where cities are nodes and roads are edges.

How BFS Works

1. Start at a specific node (source).
2. Visit all its immediate neighbors (level 1).
3. Visit all neighbors of the level 1 nodes (level 2), and so on.
4. BFS uses a queue data structure to keep track of nodes to be visited.

BFS explores the graph level by level, ensuring all nodes at a specific level are visited before moving to the next.

1. Initialize:
    - Create a queue data structure.
    - Mark all nodes as unvisited.
2. Start at the source node:
    - Add the source node to the queue.
    - Mark it as visited.
3. Iterate until the queue is empty:
    - Dequeue a node from the queue.
    - For each of its unvisited neighbors:
        - Add the neighbor to the queue.
        - Mark it as visited.

General python code for bfs:

```
def bfs(graph, start):
  queue = []
  visited = set()
  queue.append(start)
  visited.add(start)
  while queue:
    node = queue.pop(0)
    for neighbor in graph[node]:
      if neighbor not in visited:
        queue.append(neighbor)
        visited.add(neighbor)
```

Initialize: Create an empty queue and a set for visited nodes.

Start: Add the starting node to the queue and mark it as visited.

Loop: While the queue is not empty, repeat the following steps:

Dequeue: Remove a node from the front of the queue.

Explore Neighbors: For each neighboring node of the current node:

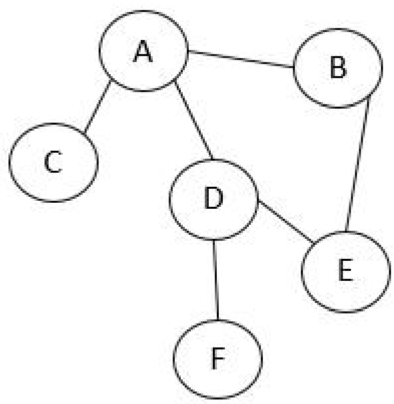
If the neighbor hasn't been visited yet:

Add it to the queue.

Mark it as visited.

This process ensures that each node is visited level by level, starting from the given start node.

BFS Traversal:



1. **Start at node A:**
    i. Add A to the queue.
    ii. Mark A as visited.
2. **Visit A's neighbors:**
    i. Dequeue A from the queue.

  ii. Visit B, C, and D.
  iii. Mark B, C, and D as visited.
  iv. Add B, C, and D to the queue.

3. **Visit B's neighbors:**
  i. Dequeue B from the queue.
  ii. Visit E.
  iii. Mark E as visited.
  iv. Add E to the queue.

4. Visit C's neighbors:
  i. Dequeue C from the queue.
  ii. No unvisited neighbors.

5. Visit D's neighbors:
  i. Dequeue D from the queue.
  ii. Visit F.
  iii. Mark F as visited.
  iv. Add F to the queue.

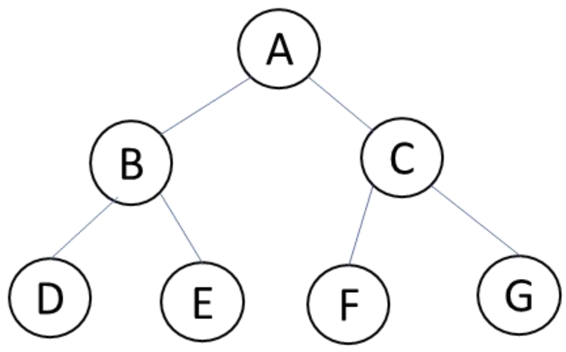
6. Visit E's neighbors:
  i. Dequeue E from the queue.
  ii. No unvisited neighbors.

7. Visit F's neighbors:
  i. Dequeue F from the queue.
  ii. No unvisited neighbors.

Output:
BFS Traversal Order: A, B, C, D, E, F

Example program:

```
def bfs(graph, start):
    queue = []
    visited = set()
    queue.append(start)
    visited.add(start)
    while queue:
        node = queue.pop(0)
        print(node, end=' ')
        for neighbor in graph[node]:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': [],
    'E': [],
    'F': [],
    'G': []
}
start_node = 'A'
```



Output:

BFS traversal starting from node 'A': A B C D E F G

```
print("BFS traversal starting from node 'A':", end=' ')
bfs(graph, start_node)
```

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. O is Order of growth

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

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.

Applications of BFS

- Pathfinding: Finding the shortest path between two points in a map or game.
- Shortest Path Problems: Identifying the most efficient route in various scenarios.
- Network Routing: Optimizing data flow in communication networks.
- Graph Search and Analysis: Exploring and analyzing social networks and other complex structures.

Examples:

- A robot navigating a maze uses BFS to find the shortest path to the exit.
- A social network platform can use BFS to recommend friends to users based on their connections.

Advantages:

- Guaranteed to find the shortest path (in terms of edges) if one exists.
- Easy to understand and implement.

Limitations:

- May not be the most efficient for deep searches (e.g., finding the deepest node in a graph).
- Can be memory-intensive for large graphs.

Depth-First Search (DFS)

- Depth-first search is a 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.
-

The step by step process to implement the DFS traversal is given as follows -

1. Initialize the stack and visited list
2. Choose a starting vertex: Select a vertex as the starting point and push it onto the stack. Mark it as visited.
3. While the stack is not empty, do the following:

- *Look at the vertex on the top of the stack.*
- *Find **an** unvisited adjacent vertex to the vertex on the top of the stack.*
- *If an unvisited adjacent vertex is found:*

- Push this vertex onto the stack.
- Mark it as visited.

- *If no adjacent vertex is found:*

- Pop the vertex from the stack.
4. Repeat steps 3 until the stack is empty.

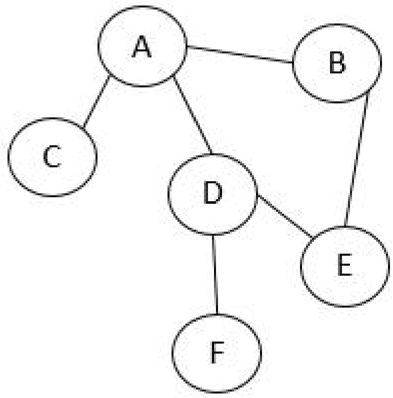
General python code:

```
def dfs(graph, start):
    stack = []
    visited = set()
    stack.append(start)
    while stack:
        node = stack.pop()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            for neighbor in reversed(graph[node]):
                if neighbor not in visited:
                    stack.append(neighbor)
```

DFS Traversal:

- **Start at node A**:

Add A to the stack.



- **Visit A**:

Pop A from the stack.
Mark A as visited.
Visit A's neighbors in reversed order
(to maintain typical DFS order):
Add D, C, B to the stack.

- **Visit B**:

Pop B from the stack.
Mark B as visited.
Visit B's neighbors:
Add E to the stack.

- **Visit E**:

Pop E from the stack.
Mark E as visited.
No unvisited neighbors.

- **Visit C**:

Pop C from the stack.
Mark C as visited.
No unvisited neighbors.

- **Visit D**:

Pop D from the stack.
Mark D as visited.
Visit D's neighbors:
Add F to the stack.

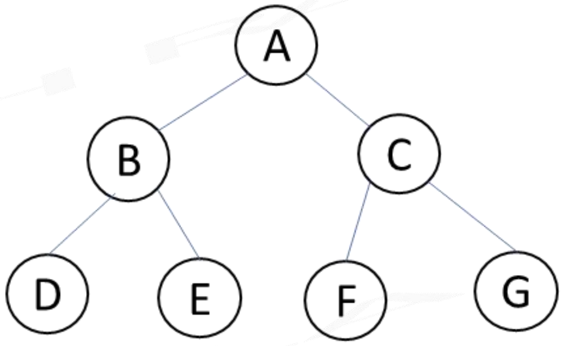
- **Visit F**:

Output:

DFS Traversal Order: A, B, E, C, D, F

Pop F from the stack.
Mark F as visited.
No unvisited neighbors.

Example program:




```
def dfs(graph, start):
    stack = []
    visited = set()
    stack.append(start)
    while stack:
        node = stack.pop()
        if node not in visited:
            print(node, end=' ')
            visited.add(node)
            for neighbor in reversed(graph[node]):
                if neighbor not in visited:
                    stack.append(neighbor)
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': [],
    'E': [],
    'F': [],
    'G': []
}
start_node = 'A'
print("DFS traversal starting from node 'A':", end=' ')
dfs(graph, start_node)
```

Output:

DFS traversal starting from node 'A': A B D E C F G

- **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.

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).

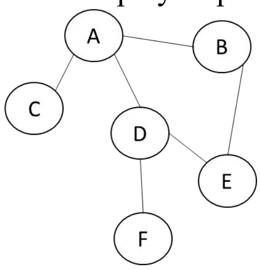
Disadvantage:

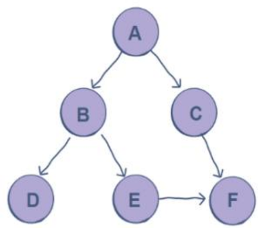
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Do it yourself:

Perform the depth first traversal for the following graphs.

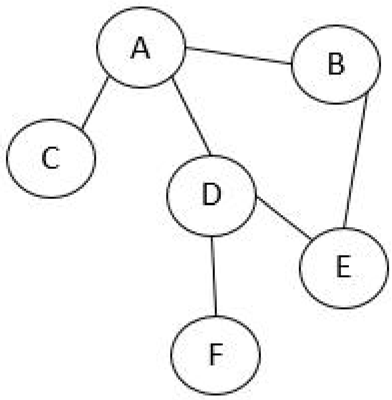
- Write the python code for BFS and DFS for the graph provided.
- Write the step by step traversal.





#Program to find bfs and dfs

```
from collections import deque
def bfs(graph, start):
    queue = deque([start])
    visited = set([start])
    while queue:
        node = queue.popleft()
        print(node, end=" ")
        for neighbor in graph[node]:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)
def dfs(graph, start):
    visited = set()
    def dfs_recursive(node):
        visited.add(node)
        print(node, end=" ")
        for neighbor in graph[node]:
            if neighbor not in visited:
                dfs_recursive(neighbor)
    dfs_recursive(start)
graph = {
    'A': ['B', 'C', 'D'],
    'B': ['E'],
    'C': [],
    'D': ['E', 'F'],
    'E': [],
    'F': []
}
print("BFS traversal:")
```



Output:

BFS traversal:

A B C D E F

DFS traversal:

A B E C D F

```
bfs(graph, 'A')
print("\nDFS traversal:")
dfs(graph, 'A')
```

Difference:

BFS	DFS
BFS stands for Breadth First Search.	DFS stands for Depth First Search.
BFS uses a Queue to find the shortest path.	DFS uses a Stack to find the shortest path.
BFS is better when target is closer to Source.	DFS is better when target is far from source.
As BFS considers all neighbor so it is not suitable for decision tree used in puzzle games.	DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the conclusion, we won.
BFS is slower than DFS.	DFS is faster than BFS.
Time Complexity of BFS = O(V+E) where V is vertices and E is edges.	Time Complexity of DFS is also O(V+E) where V is vertices and E is edges.
BFS requires more memory space.	DFS requires less memory space.
In BFS, there is no problem of trapping into finite loops.	In DFS, we may be trapped into infinite loops.
BFS is implemented using FIFO (First In First Out) principle.	DFS is implemented using LIFO (Last In First Out) principle.

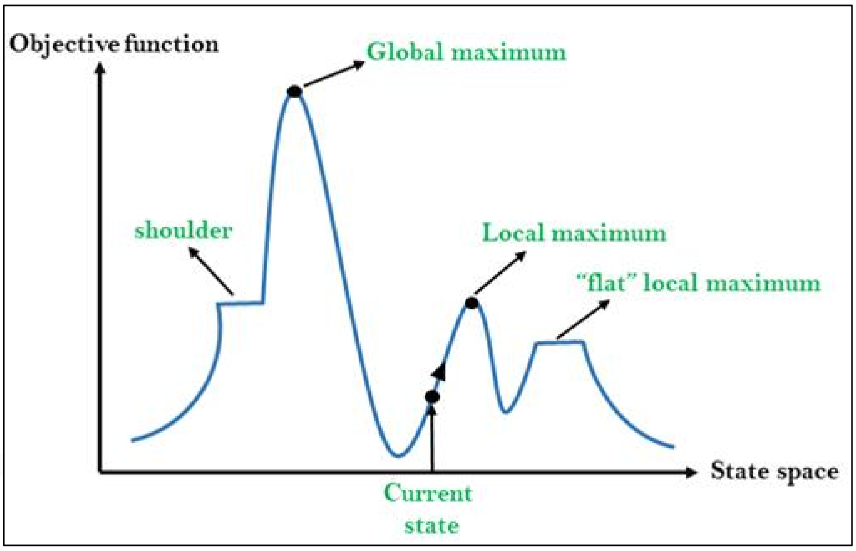
Hill climbing Search

- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.
- It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.
- A node of hill climbing algorithm has two components which are state and value.
- Hill Climbing is mostly used when a good heuristic is available.
- Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- It does not backtrack the search space, as it does not remember the previous states.

State-space Diagram for Hill Climbing:

Local Maximum: Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.
Global Maximum: Global maximum is the best possible state of state space landscape. It has the highest value of objective function.
Current state: It is a state in a landscape diagram where an agent is currently present.
Flat local maximum: It is a flat space in the landscape where all the neighbor states of current states have the same value.

Shoulder: It is a plateau region which has an uphill edge.



Problem: Find the highest point on the curve y = x^2, where x is a real number and max value for y is 16.

1. **Start at a Point:** Let's start at x = 2. So, our initial solution is x = 2.
2. **Look Around:** At x = 2, we calculate the curve's value: y = 2^2 = 4. Now, we check nearby points: x = 1 and x = 3. Calculating their values, we get y = 1^2 = 1 and y = 3^2 = 9. Since 9 is the highest, we move in the direction of x = 3.
3. **Take a Step:** We move to x = 3 and calculate y = 3^2 = 9.
4. **Repeat:** Now, at x = 3, we check the nearby points again: x = 2 and x = 4. The values are y = 4 and y = 4^2 = 16. We see that y = 16 is higher, so we move to x = 4.
5. **Stop:** At x = 4, we calculate y = 4^2 = 16. Now, when we check nearby points (x = 3 and x = 5), both give y = 9 and y = 25, respectively. Neither of these points is higher than the current point (x = 4, y = 16).

Types of Hill Climbing Algorithm:

- **Simple hill Climbing**
- **Steepest-Ascent hill-climbing**
- **Stochastic hill Climbing**

Simple Hill Climbing

Step 1: Evaluate the initial state, if it is goal state then return success and Stop.
Step 2: Loop Until a solution is found or there is no new operator left to apply.
Step 3: Select and apply an operator to the current state.
Step 4: Check new state:

- If it is goal state, then return success and quit.
- Else if it is better than the current state then assign new state as a current state.
- Else if not better than the current state, then return to step2.

Step 5: Exit.

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.

Step 1: Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.

Step 2: Loop until a solution is found or the current state does not change.

- Let SUCC be a state such that any successor of the current state will be better than it.
- For each operator that applies to the current state:
  - Apply the new operator and generate a new state.
  - Evaluate the new state.
  - If it is goal state, then return it and quit, else compare it to the SUCC.
  - If it is better than SUCC, then set new state as SUCC.
  - If the SUCC is better than the current state, then set current state to SUCC.

- **Step 5:** Exit.

- **Simple Hill Climbing**: Here, at each iteration, the algorithm generates a single neighboring solution and moves to it if it's better than the current solution. It doesn't necessarily choose the best neighboring solution; it just takes the first one that leads to improvement.

- **Steepest Ascent Hill Climbing**: Steepest ascent hill climbing takes a more cautious approach. It examines all the neighboring solutions and selects the one that has the steepest increase in the objective function value. It's more systematic in evaluating potential improvements.

Stochastic hill climbing:

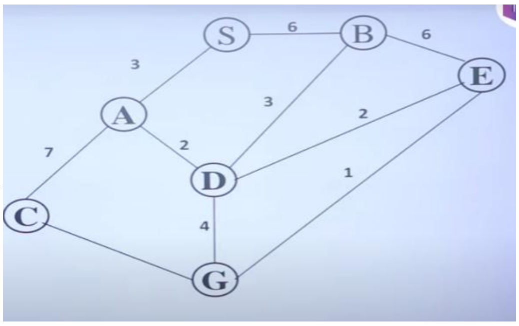
- Stochastic hill climbing does not examine for all its neighbor before moving.
- Rather, this search algorithm selects one neighbor node at random and decides whether to choose it as a current state or examine another state.

1. **Initialization:** Start from an initial point in the search space.
2. **Evaluation:** Evaluate the current solution by calculating the value of the objective function.
3. **Neighbour Generation:** Generate neighbouring solutions by making small modifications to the current solution.
4. **Selection with Probability:** Instead of always selecting the best neighbouring solution, Stochastic Hill Climbing accepts the new solution with a certain probability. The probability of accepting a worse solution is determined by a parameter (often called the "temperature") that decreases over time.
5. **Comparison**: Compare the objective function values of the current solution and the selected neighboring solution.
6. **Update**: If the selected neighboring solution is better, always move to that solution. If it's worse, decide whether to accept it based on the probability determined by the temperature.
7. **Temperature Update**: Decrease the temperature according to a predefined schedule. This reduces the probability of accepting worse solutions over time, making the algorithm more focused on exploiting better solutions as it progresses.
8. **Termination:** The algorithm terminates based on a predefined stopping criterion, such as reaching a certain number of iterations or when the temperature becomes very low.

Branch and Bound

- Branch and Bound is an algorithmic technique which finds the optimal solution by keeping the best solution found so far.
- Branching is the processes of generating sub problems
- Bounding refer to ignoring solution that can not be better than current best solution
- It eliminates those parts of a search space which does not contain better solution
- Branch and Bound search is a way to combine space saving of depth first search with heuristic info.
- Idea in Branch and Bound search is to maintain lowest cost path to a goal found so far and its cost.

- **Step 1:** Traverse the root node.
- **Step 2:** Traverse any neighbor of the root node that is maintaining least distance from the root node.
- **Step 3:** Traverse any neighbor of the neighbor of the root node that is maintaining least distance from the root node.
- **Step 4:** This process will continue until we are getting the goal node.



A* search algorithm

- A* search algorithm finds the shortest path through the search space using the heuristic function.
- It uses h(n) i.e. search heuristic and g(n) i.e. cost to reach the node n from the start state. Hence we combine both costs as,
  f(n)= g(n) + h(n) where f(n) is the estimated cost of the cheapest solution.
- This algorithm expands less search tree and provides optimal result faster.
- This algorithm is complete if the branching factor is finite and every action has fixed cost.
- A* requires heuristic function to evaluate the cost of path that passes through the particular state.
- **Graph Search:** Operates on nodes and edges with associated costs.
- **Heuristic Function (h(n)):** Estimates the cost to reach the goal from node n.
- **Cost Function (g(n)):** Represents the actual cost from the start node to node n.
- **Evaluation Function (f(n)):** Combines g(n) and h(n) as f(n) = g(n) + h(n).

Algorithm Steps

1. Initialize the open list with the start node.
2. Initialize the closed list as empty.
3. Repeat until the open list is empty:
    - Select the node n with the lowest f(n) from the open list.
    - If n is the goal, reconstruct the path and return it.
    - Move n from the open list to the closed list.
    - For each neighbor of n:
        - If the neighbor is in the closed list, skip it.

    - If the neighbor is not in the open list or has a lower f(n), update its g(n) and f(n) and add it to the open list.

Advantages

- Guaranteed to find the shortest path if the heuristic is admissible. It is complete and optimal.
- It is used to solve very complex problems.
- Efficient in both time and space for many practical problems.

Limitations

- Performance depends heavily on the quality of the heuristic.
- Can be computationally expensive for large graphs with many nodes.
- This algorithm is complete if the branching factor is finite and every action has fixed cost.
- The speed execution of A* search is highly dependant on the accuracy of the heuristic algorithm that is used to compute h (n).
- It has complexity problems.

AO* search algorithm

- AO* (AND-OR) search is a best-first search algorithm for solving problems with AND-OR graphs.
- Used for finding the optimal solution by exploring AND-OR graphs which include multiple possible paths and subproblem.
- Often used in decision-making, planning, and problem-solving.
- Nodes:
    - OR Nodes: Represents a choice between multiple alternatives.
    - AND Nodes: Represents a set of subproblems that need to be solved collectively.
    - Edges: Connect nodes and represent the relationship between decisions and subproblems.

Components of AO* Search

- **Open List:** Nodes that are generated but not yet expanded.
- **Closed List:** Nodes that have already been expanded.
- **Heuristic Function (h):** Estimates the cost from the current node to the goal.
- **Cost Function (g):** Actual cost from the start node to the current node.
- **Evaluation Function (f):** Combines g and h to guide the search (f = g + h).

Steps of AO* Search

- **Initialization:** Add the start node to the open list.
- **Selection:** Select the most promising node (lowest f value) from the open list.
- **Expansion:** Expand the selected node by generating its successors.
- **Evaluation:** Compute the cost for each successor and update the parent node's cost.
- **Backtracking:** If an AND node's successors are all solved, mark it as solved and update its parent.
- **Repetition:** Repeat the process until the goal is reached or no more nodes can be expanded.

Advantages:

- Efficient in solving AND-OR graphs.
- Finds the optimal solution if the heuristic is admissible.
- Can handle complex decision trees and problem-solving scenarios.

Disadvantages:

- Relies on the quality of the heuristic for efficiency.

- Computationally expensive for large graphs.

Applications:

- Search Expert Systems: Used in decision-making processes.
- Robotics: Path planning and navigation.
- Game Playing: Solving game trees with AND-OR structures.
- AI Planning: Complex problem-solving involving subproblems.

Compare and contrast:

Feature	A* Algorithm	AO* Algorithm
Optimality	Guaranteed to find optimal solution if heuristics are consistent	Does not guarantee optimality; provides approximate solutions
Completeness	Complete: Will find a solution if one exists	Complete: Will converge to a solution over time
Memory Requirement	Requires memory to store entire search tree/graph	More memory efficient; does not require storing entire search tree/graph
Time Complexity	Can have exponential time complexity	Typically converges faster than A*, but may sacrifice optimality
Resource Allocation	Fixed computational resources	Adapts resource allocation, can be interrupted at any time
Iterative Improvement	Does not iteratively improve solution	Iteratively improves solution over time
Interruptibility	Not interruptible during computation	Interruptible at any time, providing a solution at any point
Solution Quality Control	Produces optimal solutions if possible	Provides solutions of varying quality depending on resources
Application	Well-suited for scenarios where finding the optimal solution is crucial, e.g., pathfinding in robotics or games	Useful for real-time systems or resource-constrained scenarios