Introduction To Artificial Intelligence

Introduction:

In 1968, Marvin Minsky defined Artificial Intelligence as:

"The science of making machines do things that would require intelligence if done by men."

So, in other words:

Artificial Intelligence (AI) is the branch of computer science that aims to create machines capable of performing tasks that typically require human intelligence. These tasks include learning from experience, understanding and processing natural language, recognizing patterns and images, solving problems, and making decisions. AI systems use algorithms and data to simulate human cognitive functions, enabling them to adapt and improve over time. By mimicking human thought processes, AI can automate complex activities, from driving cars and diagnosing diseases to personalizing content recommendations and managing financial portfolios.

History of AI:

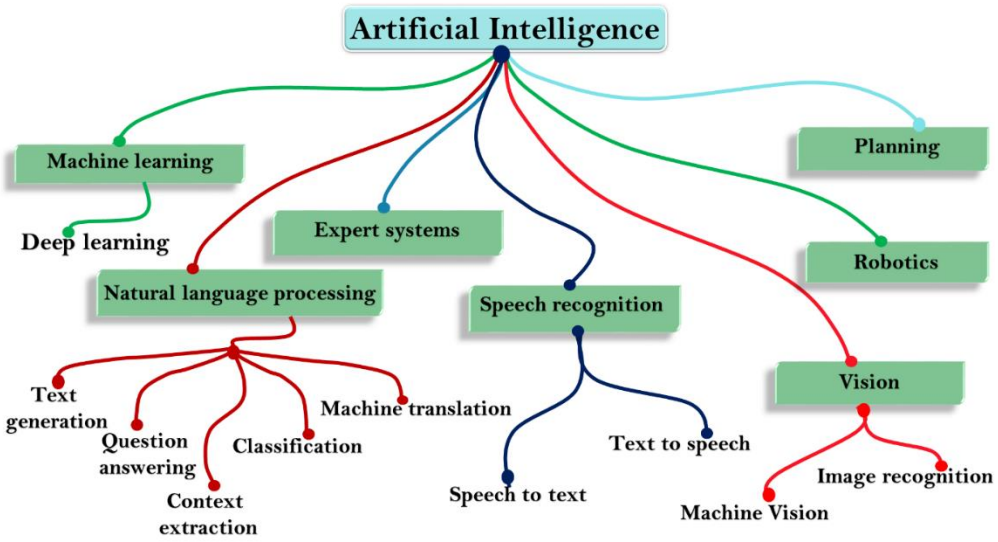
- The concept of Artificial Intelligence has intrigued humans for centuries, but serious research began in the 1950s.
- Alan Turing, a pioneering British mathematician, laid the groundwork with his idea that machines could simulate any human intelligence process, famously posing the question, "Can machines think?" His Turing Test became a fundamental criterion for determining a machine's ability to exhibit intelligent behavior equivalent to, or indistinguishable from, that of a human.
- In the mid-1950s, John McCarthy, Marvin Minsky, Nathaniel Rochester, and Claude Shannon organized the Dartmouth Conference, where the term "Artificial Intelligence" was coined. This event marked the formal beginning of AI as a field of study.
- Early successes included the development of programs that could play chess and solve algebra problems, showcasing machines' potential to mimic human thinking.
- Throughout the following decades, AI research evolved significantly. In the 1980s, the advent of machine learning allowed computers to learn from data and improve their performance over time.

- In the 2010s, deep learning, inspired by the human brain's neural networks, led to breakthroughs in image and speech recognition.
- Today, AI technologies like natural language processing, robotics, and machine vision are integrated into everyday applications. From virtual assistants like Siri and Alexa to self-driving cars and advanced medical diagnostics, AI continues to transform industries and enhance our daily lives.

Importance of AI:

Artificial Intelligence (AI) is crucial because it enables machines to perform tasks that typically require human intelligence. This includes learning from data, recognizing patterns, solving complex problems, and making decisions autonomously. AI is transforming industries by improving efficiency, accuracy, and innovation across various sectors such as healthcare, finance, transportation, and entertainment, ultimately enhancing our daily lives.

Subsets of AI:



Artificial Intelligence (AI) comprises several specialized fields:

- **Machine Learning (ML):** ML enables machines to learn from data and improve their performance without explicit programming. It powers recommendation systems, fraud detection, and personalized medicine.
- **Deep Learning:** Inspired by the human brain's neural networks, Deep Learning focuses on complex patterns and is used for image and speech recognition, autonomous driving, and natural language processing.

- **Natural Language Processing (NLP):** NLP allows computers to understand, interpret, and generate human language. It's essential for virtual assistants, language translation, and sentiment analysis.
- **Expert Systems:** These AI systems mimic human decision-making in specific domains, aiding in medical diagnostics, financial advising, and technical support.
- **Robotics**: Robotics involves designing and programming robots to perform tasks autonomously, from manufacturing to space exploration and healthcare.
- **Machine Vision:** Machine Vision enables machines to interpret visual information, used in quality control, autonomous vehicles, and surveillance systems.
- **Speech Recognition:** AI systems that understand and interpret human speech, used in voice assistants, speech-to-text applications, and accessibility tools.

Applications of AI:

- **Healthcare**: AI aids in diagnostics, personalized treatment plans, and drug discovery.
- **Finance**: Used for fraud detection, algorithmic trading, and customer service automation.
- **Retail**: Powers personalized recommendations, inventory management, and customer service chatbots.
- **Transportation**: Enables autonomous vehicles, traffic management systems, and predictive maintenance for fleets.
- **Manufacturing**: Optimizes production lines, quality control, and predictive maintenance.
- **Education**: Supports personalized learning platforms, student progress tracking, and automated grading systems.
- **Entertainment**: Enhances gaming experiences, content recommendation, and virtual reality simulations.

Physical Symbol System Hypothesis

Introduction:

The Physical Symbol System Hypothesis (PSSH) is a theory proposed by Allen Newell and Herbert A. Simon in 1976. It is foundational in the fields of artificial intelligence and cognitive science.

The Physical Symbol System Hypothesis (PSSH) suggests that systems like computers, when equipped with symbol-processing programs, can demonstrate intelligent behaviour akin(similar) to humans. This hypothesis proposes that human cognition, which relies on manipulating symbols, can be replicated in machines.

However, Newell and Simon acknowledged the complexity involved, noting that producing true intelligence is challenging. They emphasized that debates about the hypothesis's validity rely on real-world evidence—data and observations from experiments and studies—that either support or challenge its claims.

What is a Physical Symbol System?

A physical symbol system is a mechanism that:

- **Uses Symbols**: Symbols are physical patterns that stand for something else, like letters, numbers, or icons.
- **Combines Symbols:** These symbols can be combined to form more complex structures, known as expressions or symbol structures.

How Does a Physical Symbol System Work?

- **Symbol Manipulation:** The system can perform operations on symbols, such as creating new symbols, modifying existing ones, and destroying old ones.
- **Designation**: Symbols can represent objects, actions, or concepts. For instance, the symbol “2” can represent the number two.
- **Interpretation**: The system can understand and act upon symbols. For example, the instruction “add 2 and 3” can be interpreted to perform the addition and produce the result “5”.

Key Components of PSSH

- **Symbols**: Basic units that represent information.
- **Expressions**: Combinations of symbols that form more complex ideas.
- **Operations**: Rules that define how symbols can be manipulated.

- **Interpretation** and Designation: The ability of the system to understand and act on the symbols and expressions.

The Hypothesis Explained

The PSSH posits two main points:

- **Necessity**: Any system that shows intelligent behaviour can be described as a physical symbol system. This means that for a system to be intelligent, it must be capable of manipulating symbols.
- **Sufficiency**: A well-organized physical symbol system has all the necessary components to exhibit general intelligence. This means that such a system can perform any intellectual task that a human can.

Why is PSSH Important?

- **Foundation for AI:** It provides a theoretical basis for building AI systems. By creating machines that can handle and manipulate symbols, we can develop systems that think and solve problems like humans.
- **Cognitive Science:** It offers a model for understanding human intelligence. Since humans use symbols (like language) to think and communicate, studying symbol systems helps us understand human cognition.

Applications of PSSH

- **Expert Systems:** AI programs that simulate the decision-making ability of human experts by using symbols to represent knowledge and rules to process it.
- **Natural Language Processing (NLP):** Systems that use symbols to understand and generate human language, enabling tasks like translation and sentiment analysis.
- **Robotics:** Robots use symbols to represent objects and actions, allowing them to plan and execute tasks like navigation and manipulation.
- **Cognitive Modelling:** Creating models of human thought processes by representing cognitive functions with symbols and their manipulation.
- **Problem Solving:** AI systems solve complex problems (e.g., chess, mathematical proofs) by representing states and operations with symbols.

Production Systems

What is a Production System?

A production system in artificial intelligence is like a set of instructions for a computer. It uses rules to decide how to behave based on what's happening around it. These rules are like "if-then" statements: if a certain condition is met, then the system performs a specific action. This system continuously checks its surroundings, applies the rules, and adjusts its actions accordingly. It's useful for tasks where the computer needs to make decisions and adapt to changes in its environment.

Components of Production System:

The major components of Production System in Artificial Intelligence are:

- **Global Database:** The global database is the central data structure used by the production system in Artificial Intelligence.
- **Set of Production Rules**: The production rules operate on the global database. Each rule usually has a precondition that is either satisfied or not by the global database. If the precondition is satisfied, the rule is usually be applied. The application of the rule changes the database.
- **A Control System:** The control system then chooses which applicable rule should be applied and ceases computation when a termination condition on the database is satisfied. If multiple rules are to fire at the same time, the control system resolves the conflicts.

Features of Production System:

The main features of the production system include:

- **Simplicity**: The structure of each sentence in a production system is unique and uniform as they use the “IF-THEN” structure. This structure provides simplicity in knowledge representation. This feature of the production system improves the readability of production rules.
- **Modularity**: This means the production rule code the knowledge available in discrete pieces. Information can be treated as a collection of independent facts which may be added or deleted from the system with essentially no deleterious side effects.

- **Modifiability**: This means the facility for modifying rules. It allows the development of production rules in a skeletal form first and then it is accurate to suit a specific application.
- **Knowledge-intensive:** The knowledge base of the production system stores pure knowledge. This part does not contain any type of control or programming information. Each production rule is normally written as an English sentence; the problem of semantics is solved by the very structure of the representation.

How Production Systems Function?

In AI, a production system works in cycles:

- **Match**: The system's **Inference Engine** looks at the current facts in its memory and finds rules that apply to those facts.
- **Select**: It picks one rule from the matched ones, often based on factors like how specific or recent the rule is.
- **Execute**: The chosen rule is then carried out. This usually involves updating the memory by adding new facts, changing existing ones, or removing some altogether.

This cycle repeats continuously, allowing the system to adapt and respond to new information or changes in its environment.

Control/Search Strategies:

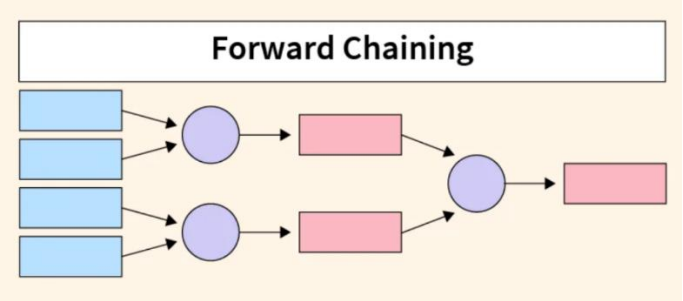
How would you decide which rule to apply while searching for a solution for any problem? There are certain requirements for a good control strategy that you need to keep in mind, such as:

- The first requirement for a good control strategy is that it should **cause motion**.
- The second requirement for a good control strategy is that it should be **systematic**.
- Finally, it must be **efficient** in order to find a good answer.

Control Strategies in AI production systems are like guides that tell the system how to reason and decide things. They control the order in which rules are used and how data is handled. This is crucial for making decisions and solving problems effectively.

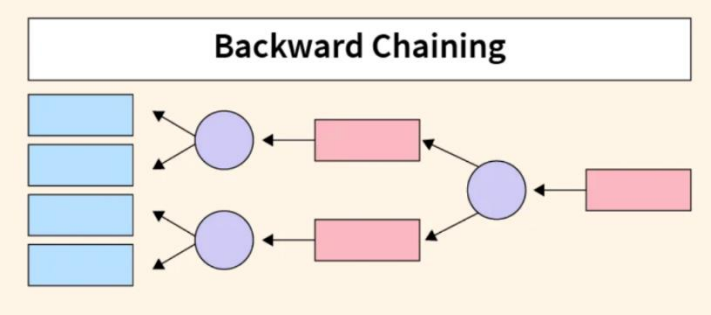
Two primary control strategies are commonly employed:

1. **Forward Chaining:**



Forward chaining in AI production systems begins with existing data and facts. It applies rules to this data step-by-step, deriving new conclusions or facts along the way. This process repeats until it achieves a specific goal or condition. Forward chaining works best when you have data and want to predict possible outcomes or results.

2. **Backward Chaining:**



Backward chaining, also known as goal-driven reasoning, starts with a clear goal or requirement. The system figures out which production rules are needed to reach that goal and works backward, activating rules as necessary until the goal is achieved or no more rules can be applied. This approach is great when you have a specific goal and need to find out what conditions or actions are necessary to achieve it.

Classes/Types Of Production System:

There are four major classes of Production System in Artificial Intelligence:

- **Monotonic Production System**: It's a production system in which the application of a rule never prevents the later application of another rule, that could have also been applied at the time the first rule was selected.
- **Partially Commutative Production System**: It's a type of production system in which the application of a sequence of rules transforms state X into state Y, then any permutation of those rules that is allowable also transforms state x into state Y. Theorem proving falls under the monotonic partially communicative system.

- **Non-Monotonic Production Systems:** These are useful for solving ignorable problems. These systems are important from an implementation standpoint because they can be implemented without the ability to backtrack to previous states when it is discovered that an incorrect path was followed. This production system increases efficiency since it is not necessary to keep track of the changes made in the search process.

- **Commutative Systems:** These are usually useful for problems in which changes occur but can be reversed and in which the order of operation is not critical. Production systems that are not usually not partially commutative are useful for many problems in which irreversible changes occur, such as chemical analysis. When dealing with such systems, the order in which operations are performed is very important and hence correct decisions must be made at the first attempt itself.

Advantages of Production Systems in AI:

- Provides **excellent tools** for structuring AI programs
- The system is highly **modular** because individual rules can be added, removed or modified independently
- Separation of **knowledge** and **Control-Recognises Act Cycle**
- A natural **mapping** onto state-space research data or goal-driven
- The system uses pattern directed control which is more **flexible** than algorithmic control
- Provides opportunities for **heuristic control** of the search
- A good way to model the **state-driven nature** of intelligent machines
- Quite helpful in **a real-time environment** and applications.

Disadvantages of Production Systems in AI:

- It is very **difficult** to analyze the flow of control within a production system
- It describes the operations that can be performed in a search for a solution to the problem.
- There is an **absence of learning** due to a rule-based production system that does not store the result of the problem for future use.
- The rules in the production system should not have any type of **conflict resolution** as when a new rule is added to the database it should ensure that it does not have any conflict with any existing rule.

An Example of Production System in AI:

Water Jug Problem:

Problem Statement: (for proper way of writing the example, refer to the book)

We have two jugs of capacity 5l and 3l (liter), and a tap with an endless supply of water. The objective is to obtain 4 liters exactly in the 5-liter jug with the minimum steps possible.

Production System:

1. Fill the 5-liter jug from the tap.
2. Pour from 5-liter to 3-liter jug
3. Empty 3-liter jug
4. Pour from 5-liter to 3-liter jug
5. Pour from 3-liter to 5-liter jug
6. Empty 5-liter jug
7. Pour from 3-liter to 5-liter jug
8. Fill 3-liter jug
9. Pour from 3-liter to 5-liter jug

This sequence results in exactly 4 liters of water in the 5-liter jug, achieving the goal with minimal steps.

Q. What are Production System Rules?

Ans: Production system rules are fundamental components used in artificial intelligence and expert systems. They consist of conditional statements typically written in the form of "if-then" rules, which govern the behavior or decision-making process of an intelligent agent or system.

1. **Structure**: Each rule has two main parts:

- **Antecedent (IF part):** Specifies a condition or a set of conditions that must be met or satisfied for the rule to be applicable.
- **Consequent (THEN part):** Specifies the action or actions that should be taken if the conditions in the antecedent are true.

2. **Example**: A simple production rule in an expert system might look like this: IF Temperature is High AND Humidity is High THEN Activate Cooling System

3. **Execution**: During operation, the production system evaluates each rule's antecedent against the current state of the system. If all conditions in the antecedent are satisfied, the rule's consequent is executed.

4. **Usage**: Production systems are widely used in fields like artificial intelligence, expert systems, and rule-based programming to model decision-making processes. They allow for the encoding of complex knowledge and reasoning into a set of manageable rules that can be easily updated or modified.

5. **Advantages**: They provide transparency in decision-making, as each rule explicitly states its conditions and actions. They are also modular, allowing for incremental addition or modification of rules without disrupting the entire system.

Heuristic Search Techniques

What Is Heuristic Search Technique?

Heuristic search is a problem-solving technique used in artificial intelligence to find solutions more efficiently by using heuristics. Heuristics are strategies or rules of thumb that help guide the search process towards the goal. Instead of exploring all possible options, heuristic search methods focus on the most promising paths, which can make solving complex problems faster and more practical.

Significance/Need of Heuristic Techniques:

Heuristic techniques are important because they:

- **Improve Efficiency**: By focusing on the most promising paths, heuristic techniques save time and computational resources. This is crucial when dealing with large or complex search spaces.
- **Enhance Problem-Solving:** They allow AI systems to tackle problems that would be too time-consuming or impossible to solve with exhaustive search methods.
- **Facilitate Real-World Applications**: Many practical applications, such as route planning in GPS systems, game AI, and puzzle-solving, rely on heuristic techniques to function effectively.

Components of Heuristic Techniques:

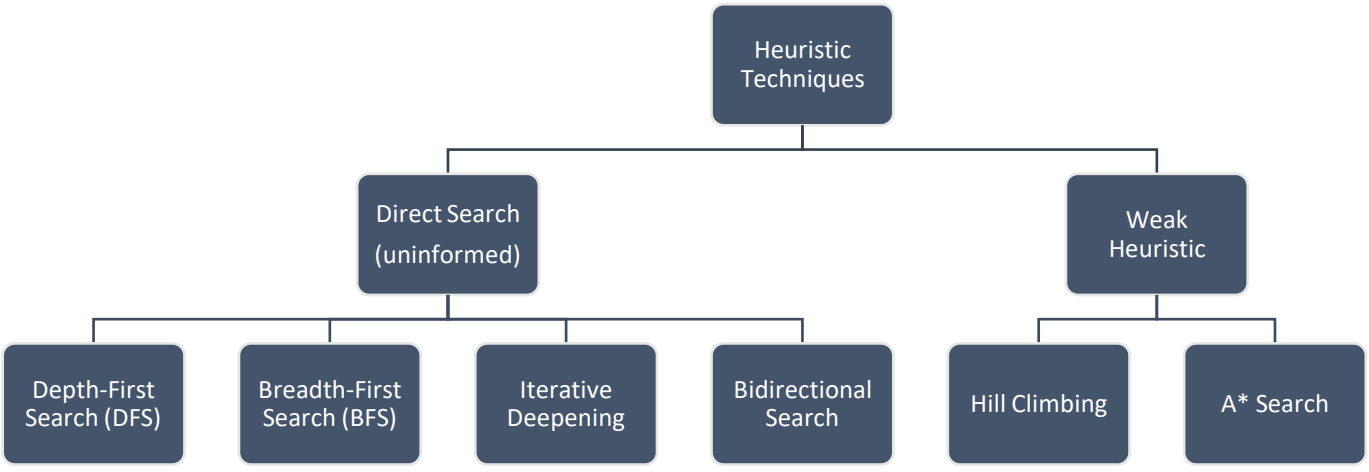
1. **Heuristic Function (h(n)):** This is a function that estimates the cost or distance from the current state to the goal. It helps in evaluating which paths are more promising. For example, in a navigation system, it might estimate the shortest distance to the destination.

2. **Evaluation Function (f(n)):** In some heuristic search methods, the evaluation function combines the cost to reach the current state (g(n)) with the heuristic estimate (h(n)). For example, in the A* algorithm, f(n) = g(n) + h(n), where g(n) is the cost from the start to the current node, and h(n) is the estimated cost from the current node to the goal.

Characteristics of Heuristic Search Techniques

- **Informed**: Uses heuristics to guide the search towards the goal.
- **Efficient**: Reduces the number of nodes explored, speeding up the search.
- **Adaptable**: Can be tailored to different problems by changing the heuristic.
- **Optimality**: Techniques like A* find optimal solutions if the heuristic is admissible.
- **Completeness**: May or may not guarantee a solution depending on the method and heuristic.
- **Heuristic Quality**: Effectiveness depends on the quality of the heuristic function.
- **Memory Usage**: Varies by technique; some use more memory to store nodes.
- **Local Optima**: Some methods, like hill climbing, can get stuck in local optima.
- **Speed**: Generally faster than exhaustive searches but depends on the heuristic.
- **Scalability**: Handles large problems better than uninformed searches, though performance can still vary.

Types of Heuristic Techniques:

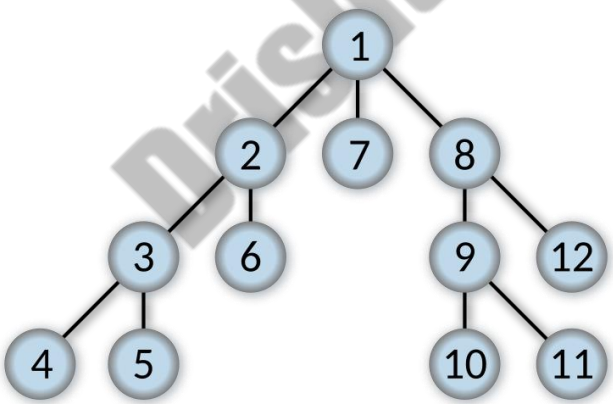


Direct Search (uninformed):

Direct search techniques are methods that explore all possible options to find a solution without using any extra information or shortcuts. These techniques rely only on the structure of the search space. They systematically examine every possible path or choice until they find a solution. Because they don't use any problem-specific knowledge, direct search techniques can be slower and less efficient than methods that use heuristics, but they are straightforward and ensure that all possible options are considered.

1. Depth- First Search:

Depth-First Search (DFS) is a fundamental search algorithm used in artificial intelligence and computer science. It explores a graph or tree structure by starting at the root and diving as deep as possible along each branch before backtracking. This approach makes DFS efficient for solving problems that require examining all possible paths, such as puzzles, maze solving, and pathfinding.



How does DFS work?

DFS works by exploring nodes and branches in a depthward motion. Here's a more detailed and easy-to-understand step-by-step explanation:

I. **Start at the Root Node**: Begin the search at the root or starting node. This is the node from where you want to start your exploration.

- Imagine you are standing at the entrance of a cave system, and this is your starting point.

II. **Explore Deeply**: From the root node, move to the first unvisited node connected to it. Continue moving to connected unvisited nodes as deep as possible.

- In our cave example, this means walking down one tunnel until you can't go any further.
- For instance, if you start at node A and A is connected to B, you move to B. If B is connected to D, you move to D, and so on.

III. **Backtrack**: When you reach a node that has no unvisited neighbors, backtrack to the previous node and explore any remaining unvisited neighbors.

- This is like reaching a dead end in the cave. You then go back to the last intersection and choose a different tunnel to explore.
- For example, if you are at node D and it has no unvisited neighbors, you backtrack to B and check if B has any other unvisited connections.

IV. **Repeat**: Continue the process of exploring deeply and backtracking until all nodes are visited or the target node is found.

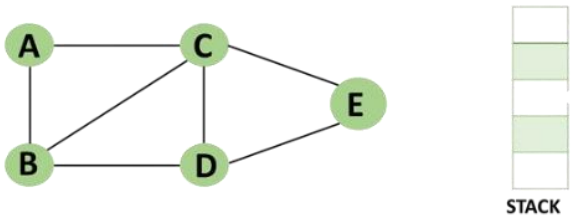
- Keep repeating the process until you have visited every part of the cave, or until you find what you are looking for.
- In our example, after backtracking from D to B, you might find and visit node E, then backtrack again if necessary, and continue this process until all nodes connected to A are visited.

Algorithm for DFS:

- **Initialize Stack:** Create a stack to hold the vertices.
- **Start Traversal**: Choose a starting vertex, mark it as visited, and push it onto the stack.
- **Explore Vertices**: Push any unvisited adjacent vertices of the top vertex onto the stack.
- **Repeat**: Continue exploring until no more adjacent vertices are unvisited.
- **Backtrack**: If no new vertices to visit, pop the top vertex from the stack and backtrack.
- **Continue**: Repeat steps 3, 4, and 5 until the stack is empty.
- **Final Spanning Tree**: Once the stack is empty, form the spanning tree by removing unused edges.

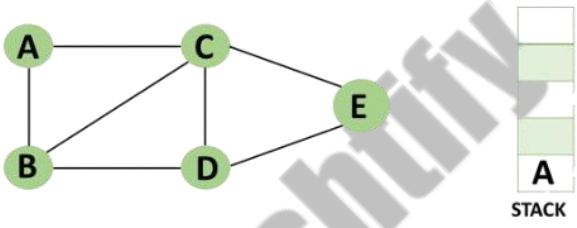
Now we shall look through a suitable example for it:

Example of DFS Algorithm:



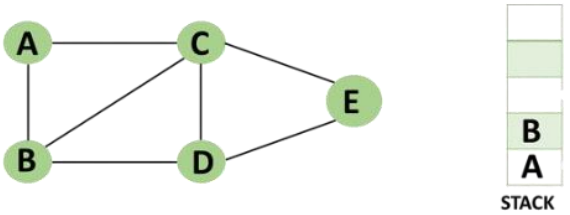
Step 1: Mark vertex A as a visited source node by selecting it as a source node.

- Push vertex A to the top of the stack.



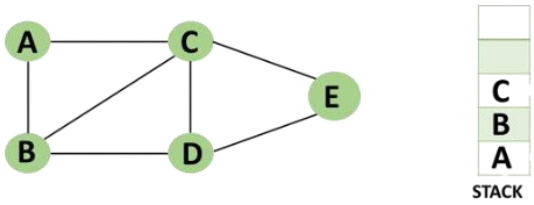
Step 2: Any nearby unvisited vertex of vertex A, say B, should be visited.

- Push vertex B to the top of the stack



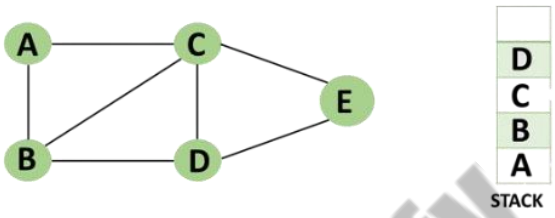
Step 3: From vertex C and D, visit any adjacent unvisited vertices of vertex B. Imagine you have chosen vertex C, and you want to make C a visited vertex.

- Vertex C is pushed to the top of the stack



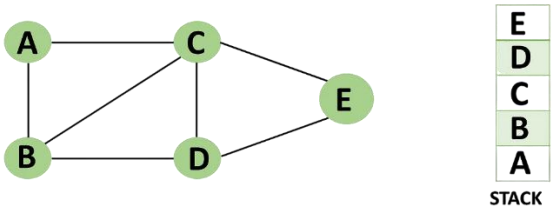
Step 4: You can visit any nearby unvisited vertices of vertex C, you need to select vertex D and designate it as a visited vertex.

- Vertex D is pushed to the top of the stack.

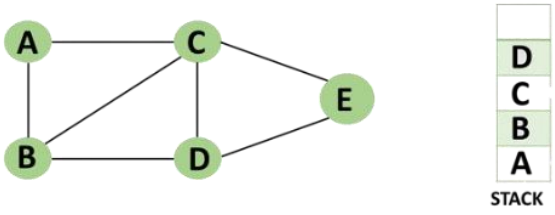


Step 5: Vertex E is the lone unvisited adjacent vertex of vertex D, thus marking it as visited.

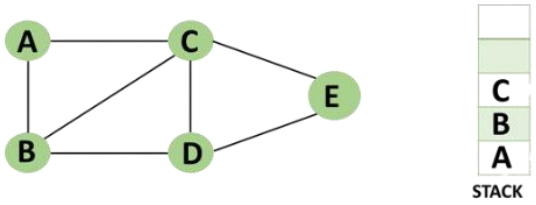
- Vertex E should be pushed to the top of the stack.



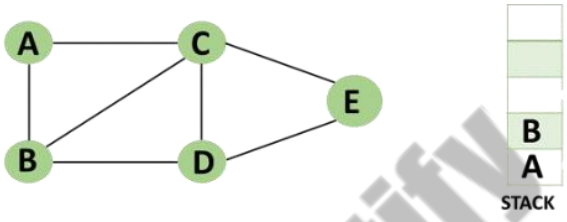
Step 6: Vertex E's nearby vertices, namely vertex C and D have been visited, pop vertex E from the stack.



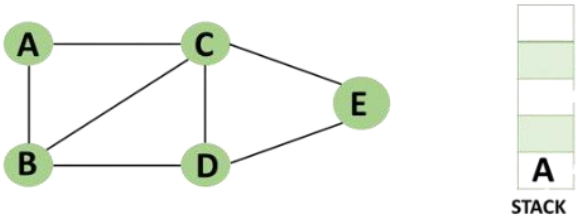
Step 7: Now that all of vertex D's nearby vertices, namely vertex B and C, have been visited, pop vertex D from the stack.



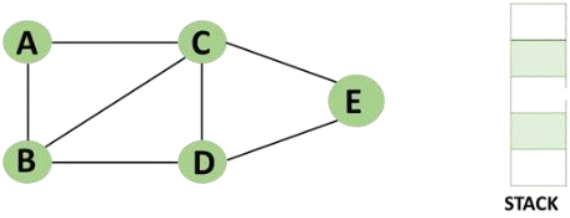
Step 8: Similarly, vertex C's adjacent vertices have already been visited; therefore, pop it from the stack.



Step 9: There is no more unvisited adjacent vertex of b, thus pop it from the stack.



Step 10: All of the nearby vertices of Vertex A, B, and C, have already been visited, so pop vertex A from the stack as well.

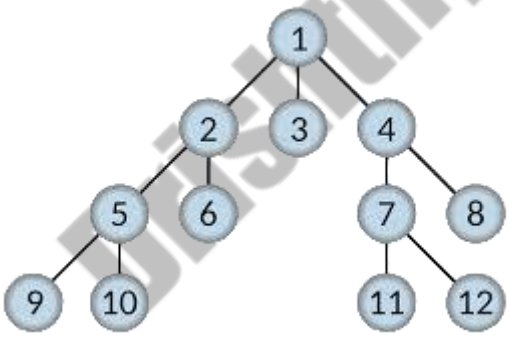


Applications of DFS Algorithm:

- Finding paths between nodes in a graph
- Testing if a graph is bipartite
- Identifying strongly connected components in a graph
- Detecting cycles within a graph

2. Breadth- First Search:

Breadth-First Search (BFS) is a key search algorithm used in artificial intelligence and computer science. It traverses a graph or tree by starting at a given node and exploring all of its neighboring nodes at the present depth before moving on to nodes at the next depth level. This approach is particularly useful for finding the shortest path in unweighted graphs, level order traversal in trees, and discovering all nodes within a connected component. BFS systematically explores each layer of the graph, making it effective for problems that involve finding the shortest path or analyzing the structure of a graph.



How does BFS work?

Breadth-First Search (BFS) works by exploring nodes level by level, ensuring that all nodes at the present depth are visited before moving on to nodes at the next depth level. Here's a step-by-step explanation:

I. **Start at the Root Node:** Begin the search at the root or starting node. This is the node from where you want to start your exploration.

- Imagine you are standing at the entrance of a maze.

II. **Initialize the Queue**: Place the starting node into a queue and mark it as visited. This queue will keep track of nodes to be explored.

- Think of the queue as a list of rooms you need to explore, starting with the room you're in.

III. **Explore Neighbors:** Dequeue a node from the front of the queue. Visit all its unvisited neighbors, mark them as visited, and enqueue them.

- This is like visiting all rooms directly connected to the current room and then adding them to your list of rooms to check next.

IV. **Repeat**: Continue the process of dequeuing nodes and exploring their neighbors until the queue is empty.
- Keep moving through each room, checking all connected rooms before moving to the next one on your list.

V. **Finish**: When the queue is empty, all reachable nodes have been visited, and you have explored the graph level by level.

- You have now visited every room in the maze that was connected to the starting point.

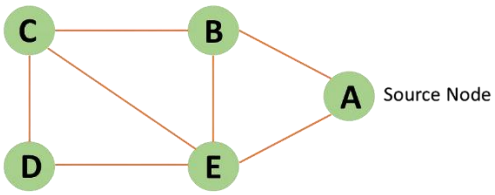
Algorithm for BFS:

- **Initialize Queue:** Create a queue to hold the vertices.
- **Start Traversal:** Choose a starting vertex, mark it as visited, and enqueue it.
- **Explore Neighbors**: Dequeue a vertex from the front of the queue. Visit all its unvisited adjacent vertices, mark them as visited, and enqueue them.
- **Repeat**: Continue the process of dequeuing and exploring neighbors until the queue is empty.
- **Finish**: When the queue is empty, all reachable vertices have been visited.

Now we shall look through a suitable example for it:

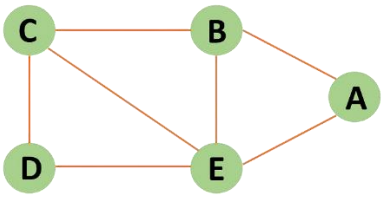
Example of DFS Algorithm:

Step 1: In the graph, every vertex or node is known. First, initialize a queue.



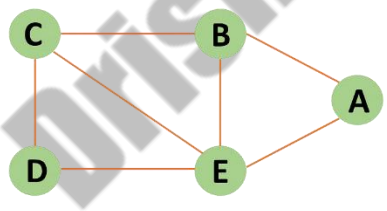
Queue

Step 2: In the graph, start from source node A and mark it as visited.



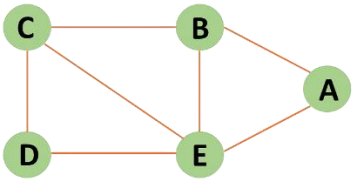
Queue | A

Step 3: Then you can observe B and E, which are unvisited nearby nodes from A. You have two nodes in this example, but here choose B, mark it as visited, and enqueue it alphabetically.



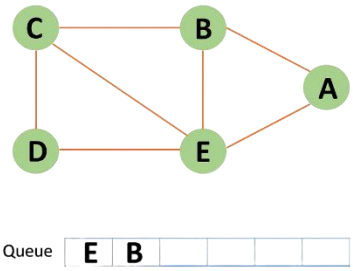
Queue | B | A

Step 4: Node E is the next unvisited neighboring node from A. You enqueue it after marking it as visited.



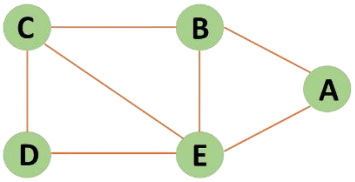
Queue | E | B | A

Step 5: A now has no unvisited nodes in its immediate vicinity. As a result, you dequeue and locate A.



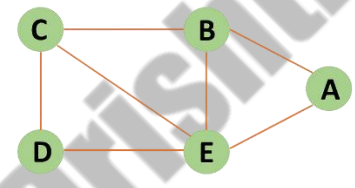
Queue | E | B | | | | |

Step 6: Node C is an unvisited neighboring node from B. You enqueue it after marking it as visited.



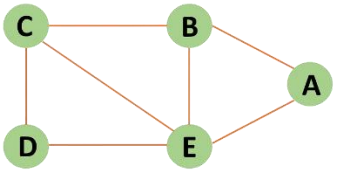
Queue | E | B | C | | | |

Step 7: Node D is an unvisited neighboring node from C. You enqueue it after marking it as visited.



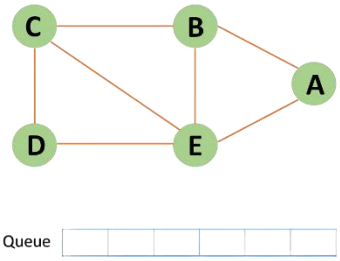
Queue | E | B | C | D | | |

Step 8: If all of D's adjacent nodes have already been visited, remove D from the queue.



Queue | E | B | C | | | |

Step 9: Similarly, all nodes near E, B, and C nodes have already been visited; therefore, you must remove them from the queue.



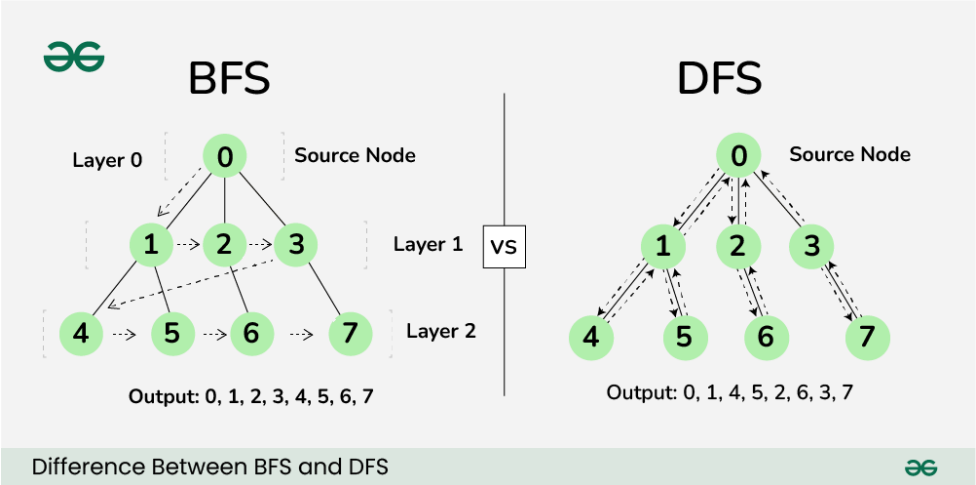
Step 10: Because the queue is now empty, the bfs traversal has ended.

Application of BFS:

- Finding the shortest path between nodes in an unweighted graph
- Level order traversal of trees
- Finding all nodes within one connected component
- Solving puzzles and games that require exploring all possible states

Difference Between DFS & BFS:

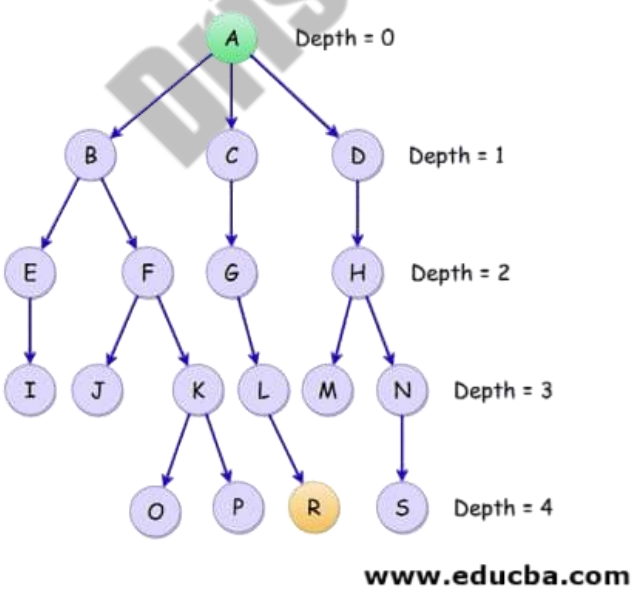
BFS	DFS
BFS stands for Breadth First Search.	DFS stands for Depth First Search.
BFS uses Queue data structure for finding the shortest path.	DFS uses Stack data structure.
BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.	DFS is a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.
BFS builds the tree level by level.	DFS builds the tree sub-tree by sub-tree.
It works on the concept of FIFO (First In First Out).	It works on the concept of LIFO (Last In First Out).
BFS is more suitable for searching vertices closer to the given source.	DFS is more suitable when there are solutions away from source.
BFS is used in various applications such as bipartite graphs, shortest paths, etc.	DFS is used in various applications such as acyclic graphs and finding strongly connected components, etc.



** Look for algorithm/example from above.

3. Iterative Deepening(Iterating Deepening Depth First Search):

Iterative Deepening Depth-First Search (IDDFS) is a search algorithm that combines the space efficiency of Depth-First Search (DFS) with the optimality of Breadth-First Search (BFS). It repeatedly applies DFS with increasing depth limits until the goal is found. This method ensures that the shallowest solution is found first, similar to BFS, while using less memory.



How IDDFS works?

IDDFS works by performing a series of depth-limited DFS searches. Here's a detailed step-by-step explanation:

I. **Start with Depth Limit Zero:** Begin by performing a DFS with a depth limit of zero. This means you only visit the root node.

- Imagine you are standing at the entrance of a building and only checking the first room.

II. **Increase Depth Limit**: Increment the depth limit by one and perform a DFS up to this new depth limit.

- Now, you check the first room and the rooms directly connected to it.

III. **Repeat the Process:** Continue increasing the depth limit and performing DFS until the goal is found or all nodes are visited.

- You keep exploring deeper levels of the building with each new iteration.

IV. **Depth-Limited DFS**: In each iteration, only explore nodes up to the current depth limit. If a node is at the current depth limit, do not explore its children in that iteration.

- This is like deciding not to enter rooms beyond a certain level until the next round.

V. **Optimality and Completeness**: IDDFS ensures that the shallowest goal node is found first, making it optimal like BFS. It is also complete, meaning it will find a solution if one exists.

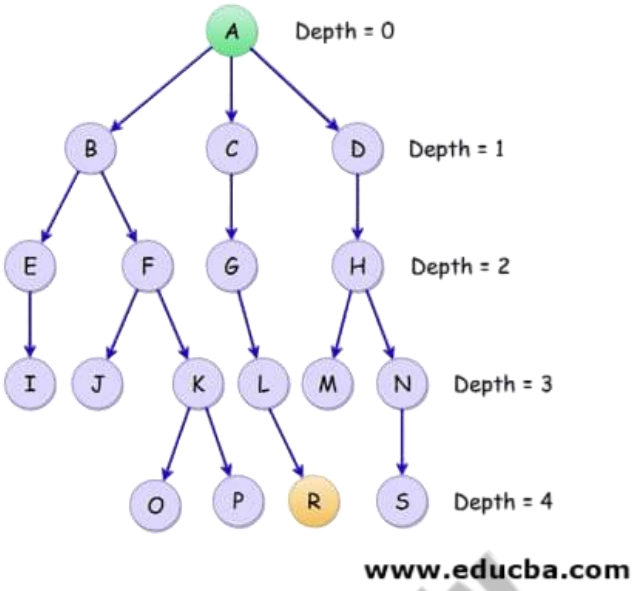
- You systematically explore each level of the building, ensuring you find the nearest exit first.

Algorithm for IDDFS:

- Set initial depth limit to 0.
- For each depth limit, perform depth-limited DFS.
- Choose a starting vertex, mark it as visited, and push it onto the stack.
- Push unvisited adjacent vertices of the top vertex onto the stack if within depth limit.
- Continue until no more vertices are within the depth limit.
- If no new vertices within the limit, pop the top vertex and backtrack.

- Increase depth limit by 1 and repeat until goal is found or all nodes are visited.

Example of IDDFS:



- The starting node is A with an initial depth of 0.
- The goal node is R; we need to find the depth and path to reach it.
- The depth to reach R from A is 4.
- This example uses a finite tree, but the same procedure applies to infinite trees.
- In IDDFS, we first perform DFS up to a specified depth.
- After each iteration, we increase the depth limit.
- This step is part of Depth Limited Search (DLS).
- The following traversal shows the IDDFS search:

The tree can be visited as: A B E F C G D H

DEPTH = {0, 1, 2, 3, 4}

DEPTH LIMITS	IDDFS
0	A
1	A B C D
2	A B E F C G D H
3	A B E I F J K C G L D H M N
4	A B E I F J K O P C G L R D H M N S

Advantages:

- Combines the space efficiency of DFS with the completeness of BFS.
- Finds the shortest path in problems with uniform cost.
- Uses less memory compared to BFS, only storing nodes for the current depth.

Disadvantages:

- Nodes are revisited multiple times, especially at deeper levels.
- Can be slower due to repeated searches.
- Less efficient for problems with very large depths.

4. Bidirectional Search:

Bidirectional search is a technique used to find the shortest path between a start node and a goal node in a graph. It operates by simultaneously searching from both the start and goal nodes until the two searches meet. This approach often leads to faster solutions than searching from one direction only, especially in large graphs.

How Bidirectional Search works?

I. **Initialization**: Begin by initializing two searches: one from the start node and one from the goal node.

- Maintain two separate queues (or lists) for the forward and backward searches.

II. **Simultaneous Search**: Expand nodes from both the start and goal nodes alternately.

- Each search explores its respective direction and keeps track of visited nodes and paths.

III. **Meeting Point**: The searches continue until they intersect or meet at a common node.

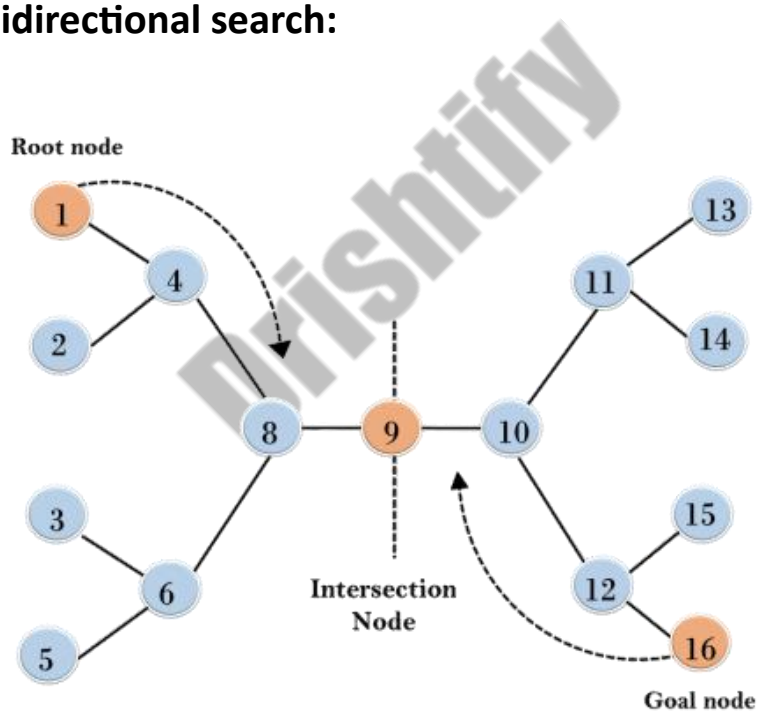
- Once the intersection is found, reconstruct the path from the start node to the goal node.

IV. **Path Construction:** Combine the paths found from the start and goal nodes to form the complete path.

Algorithm for Bidirectional Search:

- Initialize two searches: one from the start node and one from the goal node.
- Create two queues for the forward and backward searches.
- Expand nodes alternately from both queues.
- Mark nodes as visited and record paths for both searches.
- Check for an intersection between the two searches.
- Once an intersection is found, combine the paths from both searches to form the final path.
- Terminate the search after constructing the complete path.

Example of Bidirectional search:



- The starting node is 1 with an initial depth of 0.
- The goal node is 16; we need to find the depth and path to reach it.
- The depth to reach 16 from 1 is 4.
- This example uses a finite tree, but the same procedure applies to infinite trees.
- In Bidirectional Search, we simultaneously perform searches from both the start and goal nodes.
- The search terminates when the two searches meet at an intersection node.

- The following traversal shows the Bidirectional Search process:

Bidirectional Search Solution

DEPTH LIMIT	NODES
0	1, 16
1	1, 4, 8, 9, 10, 12, 16
2	1, 2, 4, 6, 8, 9, 10, 12, 15, 16
3	1, 2, 3, 4, 6, 5, 8, 9, 10, 12, 15, 16
4	1, 2, 3, 4, 6, 5, 8, 11, 13, 9, 14, 10, 12, 15, 16

Forward Search:

- Depth 0: 1
- Depth 1: 1, 4
- Depth 2: 1, 4, 8
- Depth 3: 1, 4, 8, 9
- Depth 4: 1, 4, 8, 9, 10

Backward Search:

- Depth 0: 16
- Depth 1: 12, 16
- Depth 2: 10, 12, 16
- Depth 3: 9, 10, 12, 16
- Depth 4: 8, 9, 10, 12, 16

Intersection Node:

- Node 9

Advantages:

- Often finds the shortest path faster by exploring from both ends.
- Reduces the number of nodes expanded compared to unidirectional search.
- Can be more efficient for large graphs in terms of search space.

Disadvantages:

- Requires significant memory to store nodes for both searches.
- More complex to implement compared to single-direction search algorithms.

- May be less effective if the start and goal nodes are not well-defined or if the graph changes dynamically.

Weak Heuristic Search Techniques:

Weak heuristic search techniques use simple rules or strategies to guide the search process, but they do not guarantee an optimal solution. They provide a way to explore the search space more efficiently than uninformed methods by using basic heuristics or preferences. These techniques rely on general strategies to improve the search, such as favoring certain paths or nodes that appear more promising. While they can be more efficient than direct search methods, they may still struggle with complex problems where more sophisticated heuristics are needed.

1. Hill Climbing:

The hill climbing algorithm is a search method that tries to find the best solution by always moving towards higher values or better states. It stops when it reaches a peak where no adjacent state is better. It's used for optimizing problems, like minimizing travel distance in the Traveling Salesman Problem. Also known as greedy local search, it focuses only on immediate improvements and doesn't consider further possibilities. It works with a single current state and doesn't require managing a search tree or graph, making it efficient when a good heuristic is available.

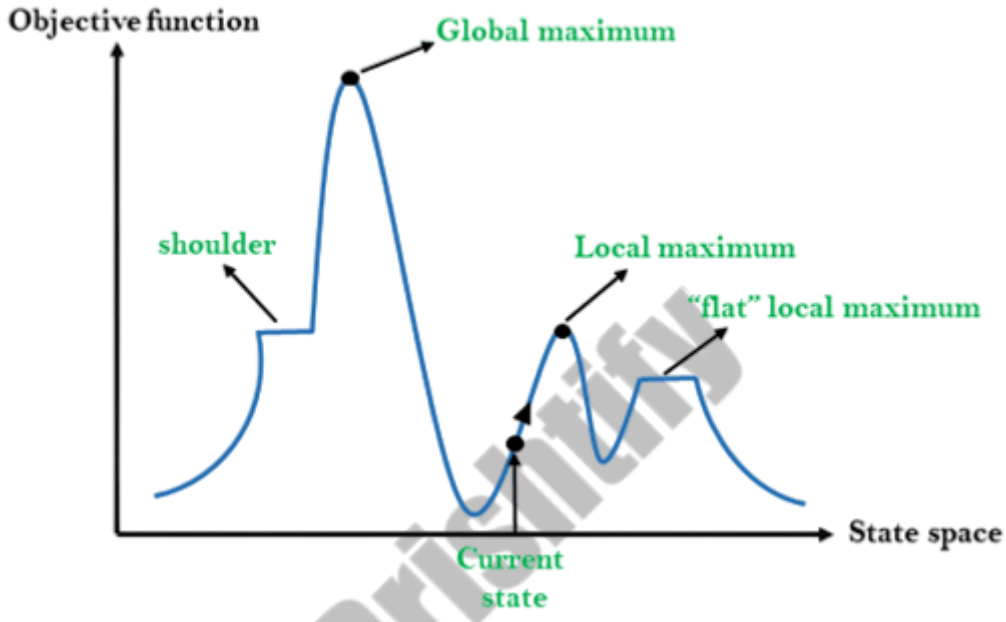
Features of Hill Climbing:

Following are some main features of Hill Climbing Algorithm:

- **Generate and Test variant:** 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.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

State-space Diagram for Hill Climbing:

The state-space landscape is a graph that shows how the hill-climbing algorithm moves through different states to find the best solution. On the Y-axis, it displays the function we're trying to optimize (like cost or objective), and on the X-axis, it shows the different states. If the Y-axis represents cost, the goal is to find the lowest point. If it represents an objective, the goal is to find the highest point.



Different regions in the state space landscape:

- **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.

Types of Hill Climbing Algorithm:

A. Simple Hill Climbing:

Simple hill climbing evaluates one neighbor at a time, moving to the first better state it finds. It checks only one successor and stays if no improvement is found. This method is quick but less optimal and not guaranteed to find the best solution.

Algorithm for Simple Hill Climbing:

i. Evaluate the initial state; if it's the goal, stop.
ii. Loop until a solution is found or no new operators remain.
iii. Apply an operator to the current state.
iv. Check the new state:
- If it's the goal, stop.
- If it's better, update the current state.
- If not, continue.

v. Exit.

B. Steepest-Ascent Hill Climbing:

Steepest-Ascent hill climbing examines all neighbors and chooses the one closest to the goal, consuming more time as it checks multiple neighbors before moving.

Algorithm for Steepest-Ascent Hill Climbing:

i. Evaluate the initial state; if it's the goal, stop; otherwise, make it the current state.
ii. Loop until a solution is found or the current state doesn't change:
- For each operator, generate and evaluate new states.
- If a new state is better than the current, update the current state.

iii. Exit.

C. Stochastic Hill Climbing:

Stochastic hill climbing randomly selects a neighbor and decides whether to move to it, rather than evaluating all neighbors.

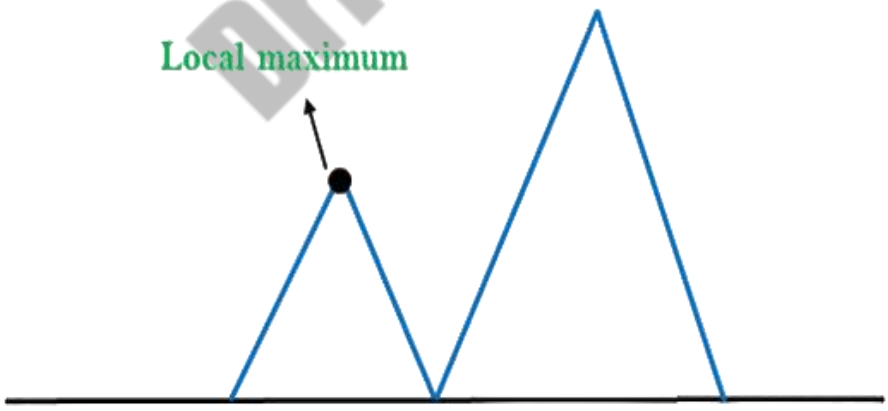
Algorithm for Stochastic Hill Climbing:

i. Start: Check if the initial state is the goal. If yes, stop.
ii. Loop: Randomly pick a neighbor of the current state.
   - If the neighbor is better, move to it.
   - If not, pick another neighbor.
iii. Repeat until the goal is reached or no neighbors are left.
iv. Finish when a solution is found or all options are exhausted.

Problems in Hill Climbing Algorithm:

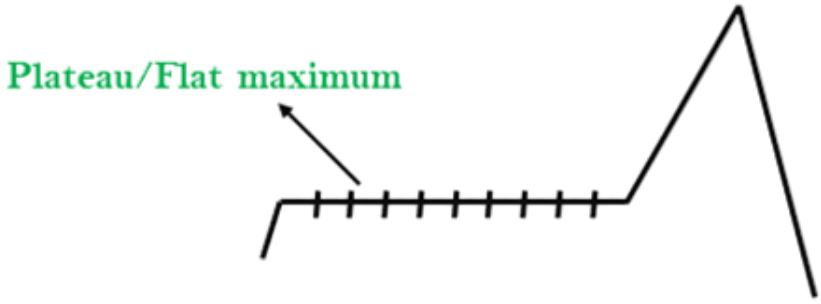
a) **Local Maximum:** A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

Solution: Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.



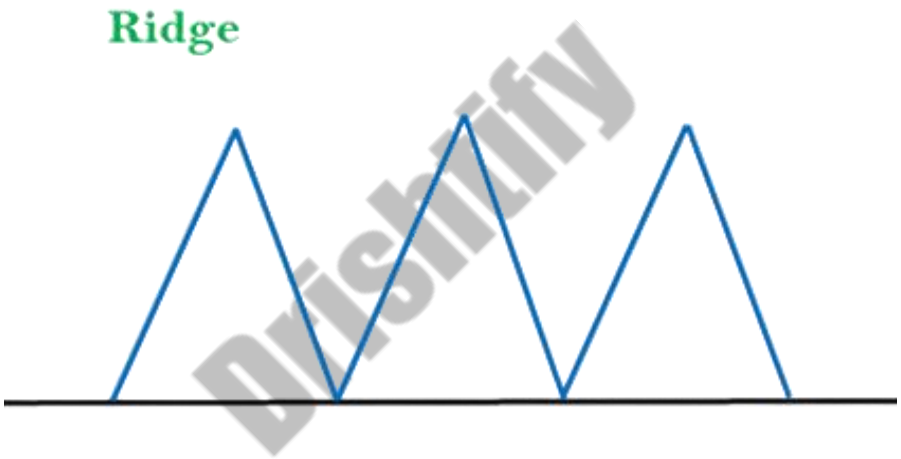
b) **Plateau**: A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

Solution: The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.



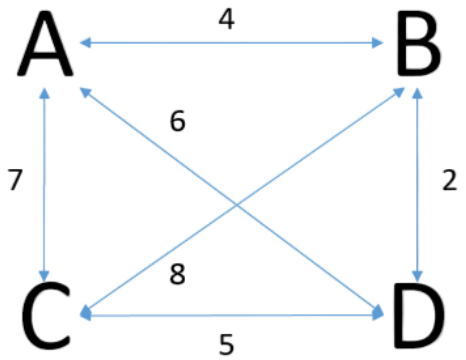
c) **Ridges**: A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.

Solution: With the use of bidirectional search, or by moving in different directions, we can improve this problem.



Write an algorithm for Hill Climbing with example:

Problem statement:
Given four cities (A, B, C, and D) and the distances between each pair of cities as shown in the image, find the shortest possible route that visits each city exactly once and returns to the starting city:



Algorithm:

i. Initial State Evaluation:
- Start at any node (e.g., A).
- If the initial state is the goal state (shortest path), stop; otherwise, make it the current state.

ii. Loop Until Solution is Found:
- For each operator: Generate and evaluate new states (possible paths).
- If a new state is better than the current: Update the current state with the new state (shorter path).
- Repeat the process until no better state is found.

iii. Exit Condition:
- Exit when the shortest path that visits all nodes and returns to the starting node is found.

Solution:

i. Evaluate Possible Routes:
- Calculate the total distance for all possible routes:
  - A → B → C → D → A: Total distance = 21 units
  - A → B → D → C → A: Total distance = 17 units
  - A → C → B → D → A: Total distance = 23 units
  - A → C → D → B → A: Total distance = 19 units
  - A → D → B → C → A: Total distance = 21 units
  - A → D → C → B → A: Total distance = 25 units

ii. Select the Shortest Route:
- The optimal path is A → B → D → C → A.

iii. Conclusion:
- Optimal Path: A → B → D → C → A
- Total Distance: 17 units

2. A* Algorithm:

A* (pronounced "A-star") is a powerful graph traversal and pathfinding algorithm used to find the shortest path between an initial and a final point. It combines elements of Dijkstra's algorithm and Greedy Best-First Search to offer an optimal and efficient solution.

A* evaluates nodes based on two key metrics:

- g(n): The actual cost from the start node to node nnn.
- h(n): The estimated cost from node nnn to the destination.

The evaluation function f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n) helps A* prioritize paths that are both short and promising. Originally designed for graph traversal problems, A* is particularly useful in map traversal, such as in robotics and navigation systems. It searches for shorter paths first, making it both optimal (finding the least cost solution) and complete (exploring all possible solutions).

How A* Algorithm works:

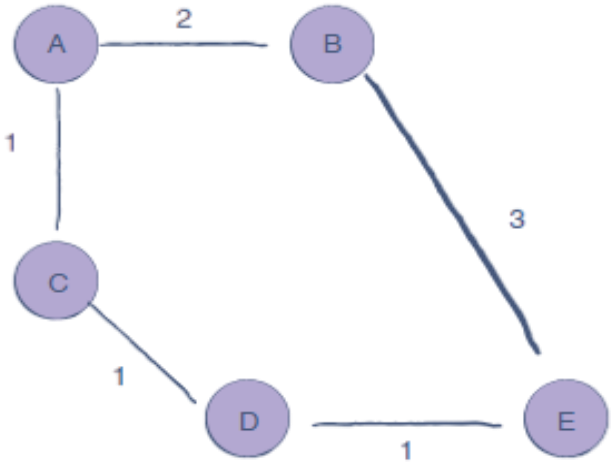
The A* algorithm finds the shortest path from a start node to a destination in a weighted graph. It uses a priority queue where nodes are prioritized based on their total cost, f(n)=g(n)+h(n) Here, g(n) is the actual cost to reach node n, and h(n) is an estimated cost from n to the destination.

Starting with the initial node, A* adds it to the queue with g(n) set to 0. It then repeatedly removes the node with the lowest f(n), expands it by exploring its neighbors, and updates their costs if a shorter path is found. The process continues until the destination node is reached or the queue is empty. A*'s efficiency depends on the heuristic h(n); a good heuristic speeds up the search by prioritizing more promising paths.

Example of A* Algorithm:

Problem Statement:

Given a graph with nodes and edges, apply the A* algorithm to find the shortest path from the initial node A to the goal node E:



Algorithm for the above problem:

- Start at Node A.
- Look at the connected nodes, B and C.
- C is closer (cost = 1), so move to C.
- From C, look at the connected nodes, A and D.
- D is closer (total cost = 2), so move to D.
- From D, look at the connected nodes, C and E.
- E is the goal, so move to E (total cost = 3).
- You've reached the goal node E.

Solution for the above problem:

- Start at A.
- Move to C (cost = 1).
- Move to D (total cost = 2).
- Move to E (total cost = 3).
- Path: A → C → D → E with a total cost of 3.

Applications of A* algorithm:

- A* helps video game characters navigate complex environments.
- In robotics, A* plans optimal movement paths.
- GPS systems use A* to find the shortest route between locations.
- A* aids AI in making decisions in dynamic environments.
- It optimizes data packet paths in networks.
- A* is used in task scheduling and resource allocation.

Advantages of A* algorithm:

- Finds the shortest path efficiently.
- Balances between speed and accuracy.

- Adaptable to various environments and problems.
- Guarantees an optimal path if the heuristic is admissible.

Disadvantages of A* algorithm:

- Can be slow with large search spaces.
- Requires significant memory for storing paths.
- Performance depends on the quality of the heuristic.
- May not be suitable for real-time applications in some cases.