INTRODUCTION TO ARTIFICIAL INTELLIGENCE

QUESTION BANK ANSWERS

UNIT-1

1. a. What are the types of Agents?

Answer:- Agents can be grouped into four classes based on their degree of perceived intelligence and capability :

- ✓ Simple Reflex Agents
- ✓ Model-Based Reflex Agents
- ✓ Goal-Based Agents
- ✓ Utility-Based Agents

❖ What are the types of Intelligent System?

- ✓ Systems that think like humans
- ✓ Systems that think rationally
- ✓ Systems that behave like humans
- ✓ Systems that behave rationally

b. What is the space complexity of BFS Algorithm?

Answer:- Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$. Where the d= depth of shallowest solution and b is branch factor.

c. Write down the properties name of Search Algorithm.

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

- ✓ **Completeness**
- ✓ **Cost Optimality**
- ✓ **Time complexity**
- ✓ **Space complexity**

d. What is the time complexity of DFS algorithm?

Answer:- 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 n is maximum number of child node of any node (branch factor).

e. Define problem solving agent.

Answer:- Problem Solving Agents decide what to do by finding a sequence of actions that leads to a desirable state or solution. An agent may need to plan when the best course of action is not immediately visible. They may need to think through a series of moves that will lead them to their goal state. Such an agent is known as a **problem solving agent.**

2. a. Explain Iterative deepening depth-first Search.

Answer:- The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.

This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.

This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

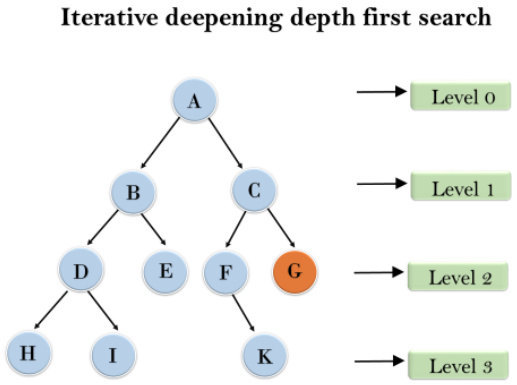
The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Here are the steps for Iterative deepening depth first search algorithm:

- Set the depth limit to 0.
- Perform DFS to the depth limit.
- If the goal state is found, return it.
- If the goal state is not found and the maximum depth has not been reached, increment the depth limit and repeat steps 2-4.
- If the goal state is not found and the maximum depth has been reached, terminate the search and return failure.

Example:

Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:



1'st Iteration-----> A
2'nd Iteration----> A, B, C
3'rd Iteration------>A, B, D, E, C, F, G
4'th Iteration------>A, B, D, H, I, E, C, F, K, G

In the fourth iteration, the algorithm will find the goal node.

b. Explain about different Environment Types in AI.

Answer:-

1. Accessible vs. inaccessible or Fully observable vs Partially Observable:

If an agent sensor can sense or access the complete state of an environment at each point of time then it is a fully observable environment, else it is partially observable.

2. Deterministic vs. Stochastic:

If the next state of the environment is completely determined by the current state and the actions selected by the agents, then we say the environment is deterministic

3. Episodic vs. non-episodic:

The agent's experience is divided into "episodes." Each episode consists of the agent perceiving and then acting. The quality of its action depends just on the episode itself, because subsequent episodes do not depend on what actions occur in previous episodes.

Episodic environments are much simpler because the agent does not need to think ahead.

4. Static vs. dynamic.

If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise it is static.

5. Discrete vs. continuous:

If there are a limited number of distinct, clearly defined percepts and actions we say that the environment is discrete. Otherwise, it is continuous.

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

Figure 2.6 Examples of task environments and their characteristics.

c. Name the Search Algorithm Terminologies and explain them.

Answer:- Different search algorithm terminologies in AI are-

- **Search:** Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:
  - 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 test: It is a function which observe the current state and returns whether the goal state is achieved or not.
- **Search tree:** A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
- **Actions:** It gives the description of all the available actions to the agent.
- **Transition model:** A description of what each action do, can be represented as a transition model.

- **Path Cost:** It is a function which assigns a numeric cost to each path.
- **Solution:** It is an action sequence which leads from the start node to the goal node.
- **Optimal Solution:** If a solution has the lowest cost among all solutions.

d. Explain BFS with suitable example.

Answer:-

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

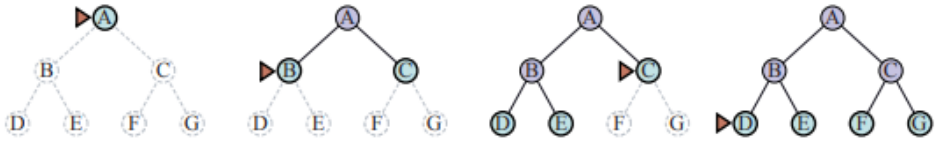


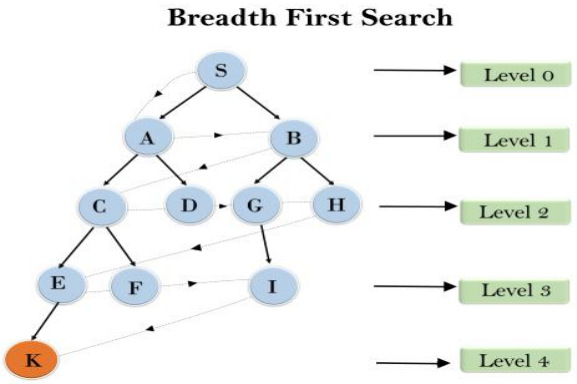
Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

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

Example:

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

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

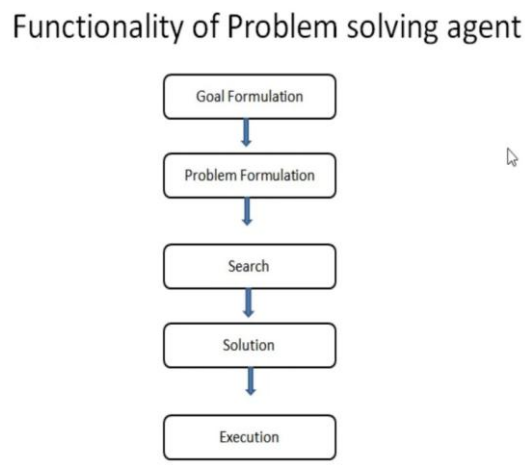


e. Explain Problem Solving Agents in brief.

Answer:- Problem Solving Agents decide what to do by finding a sequence of actions that leads to a desirable state or solution.
An agent may need to plan when the best course of action is not immediately visible. They may need to think through a series of moves that will lead them to their goal state. Such an agent is known as a **problem solving agent**, and the computation it does is known as a **search**.

The problem solving agent follows this four phase problem solving process:

1. **Goal Formulation:** This is the first and most basic phase in problem solving. It arranges specific steps to establish a target/goal that demands some activity to reach it. AI agents are now used to formulate goals.
2. **Problem Formulation:** It is one of the fundamental steps in problem-solving that determines what action should be taken to reach the goal.
3. **Search:** After the Goal and Problem Formulation, the agent simulates sequences of actions and has to look for a sequence of actions that reaches the goal. This process is called **search,** and the sequence is called a **solution**. The agent might have to simulate multiple sequences that do not reach the goal, but eventually, it will find a solution, or it will find that no solution is possible. A search algorithm takes a problem as input and outputs a sequence of actions.
4. **Execution:** After the search phase, the agent can now execute the actions that are recommended by the search algorithm, one at a time. This final stage is known as the execution phase.



Problem Formulation: A formal definition of a problem consists of five components:

1. Initial State
2. Actions
3. Transition Model
4. Goal Test
5. Path Cost

1. Initial State

It is the agent's starting state or initial step towards its goal. For example, if a taxi agent needs to travel to a location(B), but the taxi is already at location(A), the problem's initial state would be the location (A).

2. Actions

It is a description of the possible actions that the agent can take. Given a state s, **Actions**(**s**) returns the actions that can be executed in s. Each of these actions is said to be appropriate in s.

3. Transition Model

It describes what each action does. It is specified by a function **Result**(**s, a**) that returns the state that results from doing action an in state s.

The initial state, actions, and transition model together define the **state space** of a problem, a set of all states reachable from the initial state by any sequence of actions. The state space forms a graph in which the nodes are states, and the links between the nodes are actions.

4. Goal Test

It determines if the given state is a goal state. Sometimes there is an explicit list of potential goal states, and the test merely verifies whether the provided state

is one of them. The goal is sometimes expressed via an abstract attribute rather than an explicitly enumerated set of conditions.

5. **Path Cost**

It assigns a numerical cost to each path that leads to the goal. The problem solving agents choose a cost function that matches its performance measure. Remember that the optimal solution has the lowest path cost of all the solutions.

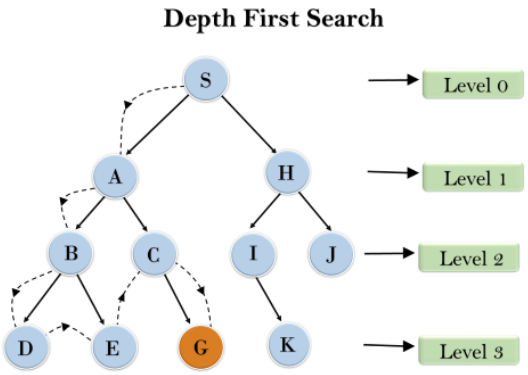
3. a. Explain DFS with suitable example.

Answer:-

- Depth-first search is a recursive algorithm for traversing a tree or graph data structure. It generally starts by exploring the deepest node in the frontier. Starting at the root node, the algorithm proceeds to search to the deepest level of the search tree until nodes with no successors are reached. Suppose the node with unexpanded successors is encountered then the search backtracks to the next deepest node to explore alternative paths.
- 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 (LIFO) for its implementation.
- 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).

Example:

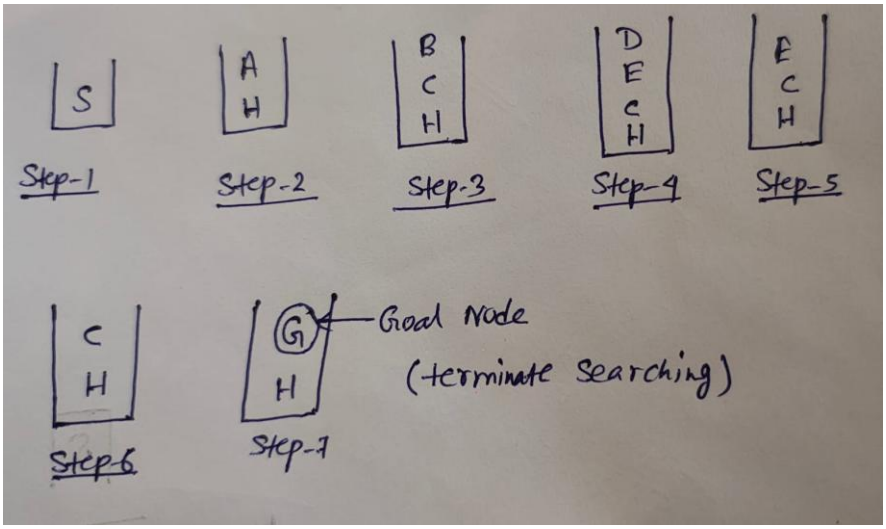
- In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:
- Root node--->Left node ----> right node.
- It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor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



Order of Visited Nodes:

$$S \rightarrow A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow G$$

Visual Representation:



b. Explain about A* search algorithm.

Answer:- A* Search algorithm is one of the best and popular informed search technique used in path-finding and graph traversals.

In the A* algorithm, we consider both path cost and heuristics. In A* the f(n) function comprises two components: path cost [g(n)] and heuristic value ([h(n)]. The f(n) value for node 'a' can be calculated as follows:

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

$f(n)_a$ = Evaluation value at the particular node (node 'a')
$g(n)_a$ = Total path cost from start node to particular node(node 'a')
$h(n)_a$ = The heuristic value of the particular node(node 'a')

Now, we will find the best path according to the A* search algorithm for our previous problem. We have to find the f(n) value for each node and select the child node with the least f(n) value and traverse until meeting the goal node. In the example, only the f(n) values of goals in the path are mentioned.

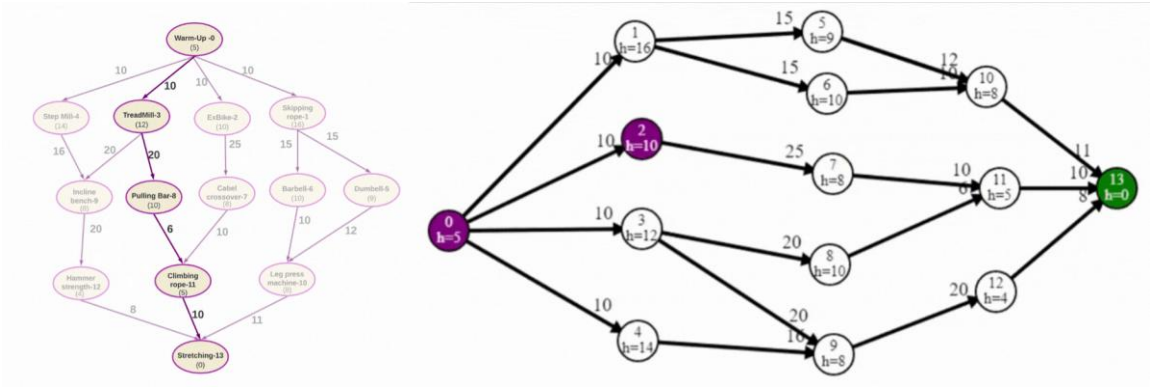


Figure 13: State-Space (left) and state-space traversal (right) in A* search

At Node-0 :

$f(n)_0 = 0 + 5 = 5$

At Node-3:

$f(n)_3 = 10 + 12 = 22$

At Node-8:

$f(n)_8 = 30 + 10 = 40$

At Node-11:

$f(n)_{11} = 36 + 5 = 41$

At Node-13:

$f(n)_{13} = 46 + 0 = 46$

$f(n)_{total} = f(n)_0 + f(n)_3 + f(n)_8 + f(n)_{11} + f(n)_{13} = 154$

Path: 0, 3, 8, 11, 13

A* algorithm is complete since it checks all the nodes before reaching to goal node/end of the state space. It is optimal as well because considering both path cost and heuristic values, therefore the lowest path with the lowest heuristics can be found with A*. One drawback of A* is it stores all the nodes it processes in the memory. Therefore, for a state space of branching factor 'b', and the depth 'd' the space and time complexities of A* are denoted by $\mathbf{O(b^d)}$. The time complexity of A* depends on the heuristic values.

c. What are the types of AI agent? Explain them with neat diagram.

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

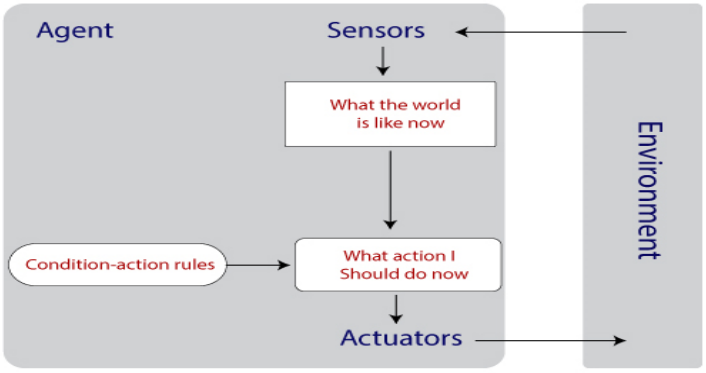
Types of agents:

Agents can be grouped into four classes based on their degree of perceived intelligence and capability :

- ✓ Simple Reflex Agents
- ✓ Model-Based Reflex Agents
- ✓ Goal-Based Agents
- ✓ Utility-Based Agents

Simple reflex agents:

- Simple reflex agents ignore the rest of the percept history and act only on the basis of the current percept.
- The agent function is based on the condition-action rule.
- If the condition is true, then the action is taken, else not. This agent function only succeeds when the environment is fully observable.

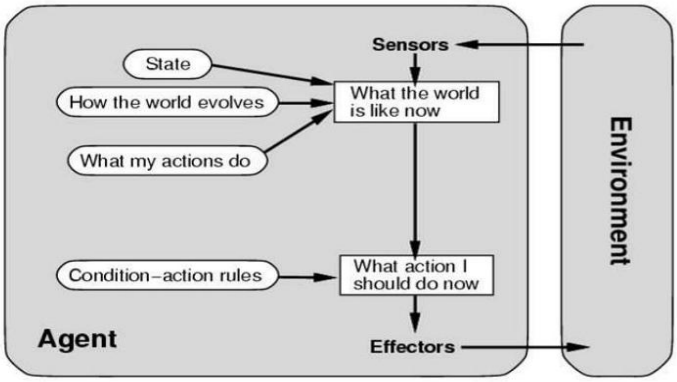


Schematic diagram of a simple-reflex agent

Model-based reflex agents:

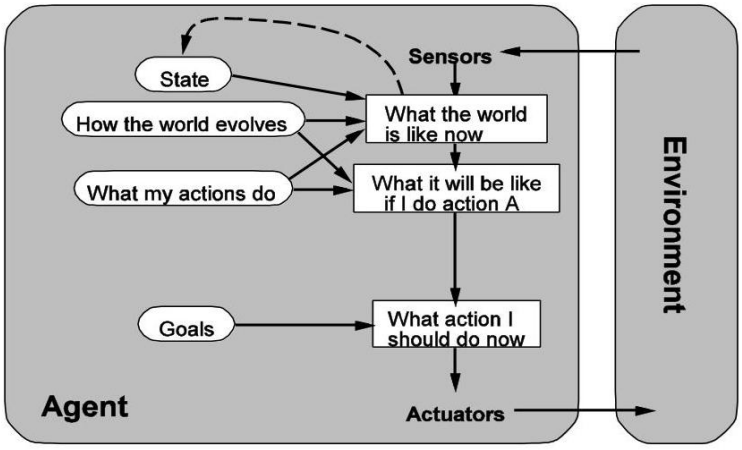
- The Model-based agent can work in a partially observable environment, and track the situation.
- A model-based agent has two important factors:
- Model: It is knowledge about "how things happen in the world," so it is called a Model-based agent.

- Internal State: It is a representation of the current state based on percept history.



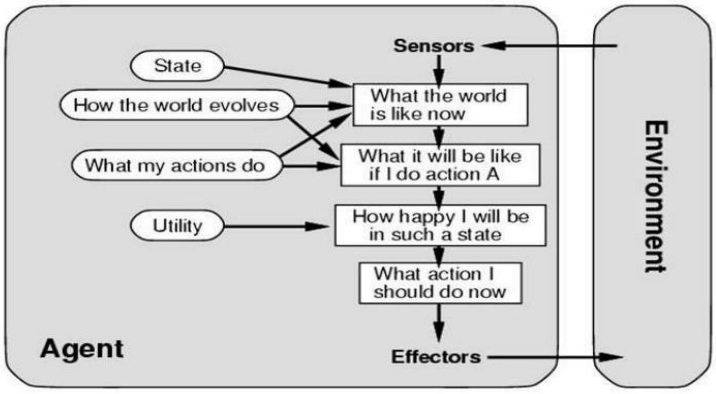
Goal-based agents:

- A goal-based agent has an agenda.
- It operates based on a goal in front of it and makes decisions based on how best to reach that goal.
- A goal-based agent operates as a search and planning function, meaning it targets the goal ahead and finds the right action in order to reach it.
- Expansion of model-based agent.



Utility-based agents:

- utility-based agent is an agent that acts based not only on what the goal is, but the best way to reach that goal.
- The Utility-based agent is useful when there are multiple possible alternatives, and an agent has to choose in order to perform the best action.
- The term utility can be used to describe how "happy" the agent is.



d. Write a short note on State Space Algorithm.

Answer:- The state space representation forms the basis of most of the AI methods.

State-space consists of all the possible states together with actions to solve a specific problem. In the graphical representation of state space (such as trees), the states are represented by nodes while the actions are represented by arcs. Any state space has an initial 'Start' node and an ending 'Goal' node or multiple Goal States. The path from the initial start node to the final goal node is known as the 'Solution' for the particular problem.

Formal Description of the problem:

1. Define a state space that contains all the possible configurations of the relevant objects.

2. Specify one or more states within that space that describe possible situations from which the problem solving process may start (initial state)

3. Specify one or more states that would be acceptable as solutions to the problem. (goal states) Specify a set of rules that describe the actions (operations) available.

For a clear understanding of state space, consider the following problem [Figure 2].

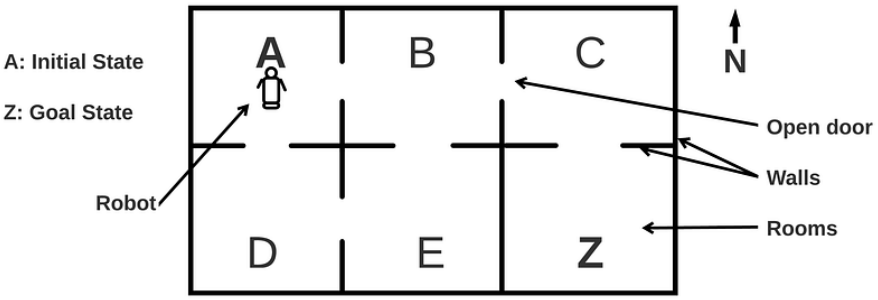


Figure 2: Example Problem

Imagine there is a robot in room ‘A’ (initial state), and it needs to go to room ‘Z’ (goal state). We can draw a state space in terms of a tree if we consider all the possible movements of the robot in each room (node). For example, when the robot is at initial state A, he can either go to B or D. When the robot moved to the next state B, he can move to C, E, or back to A [Figure 3].

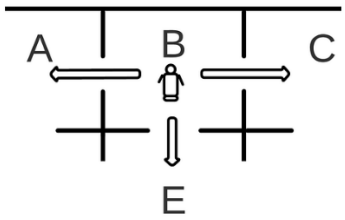


Figure 3: Possible paths for robot at state B

Based on all the possible movements of the robot at each state we can draw the state space for the above scenario as follows:

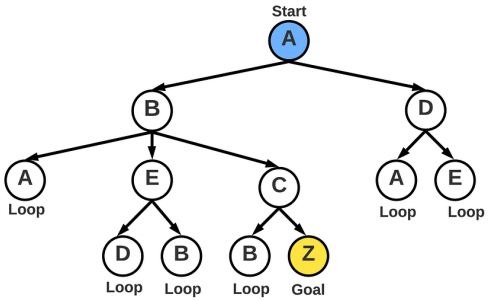


Figure 4: State-space diagram

We can identify many paths beside the direct path A, B, C, Z.

Ex: A B C Z
A B A B C Z
A D E B C Z
A D E B A B C Z
....

It can be observed that some paths are shorter while others are longer. The real problem arises here. We can figure out the best/shortest path in cases like the above example where the state space is small. But imagine a state-space with hundreds of thousands of nodes. How can we explore such a state-space?

The solution is nothing but search algorithms.

Since the state spaces are very large in general, we employ search algorithms to navigate through the state-space effectively and efficiently. A search algorithm defines how to find the path (solution) from the initial state to the goal state. Different algorithms define different methods to move from the current state (node) to the next state (node). Some algorithms provide just the systematic ways to explore the state space while others tell how to explore effectively and efficiently.

e. Write about Greedy Search Algorithm.

Answer:- In greedy search, the heuristic values of child nodes are considered. The path is determined by calculating the path with the nodes with the lowest heuristic values. Another fact to be noticed is that usually the initial node has the highest heuristic value and the goal node has the lowest. But there can be exceptions like getting a mid-range value for the initial node also.

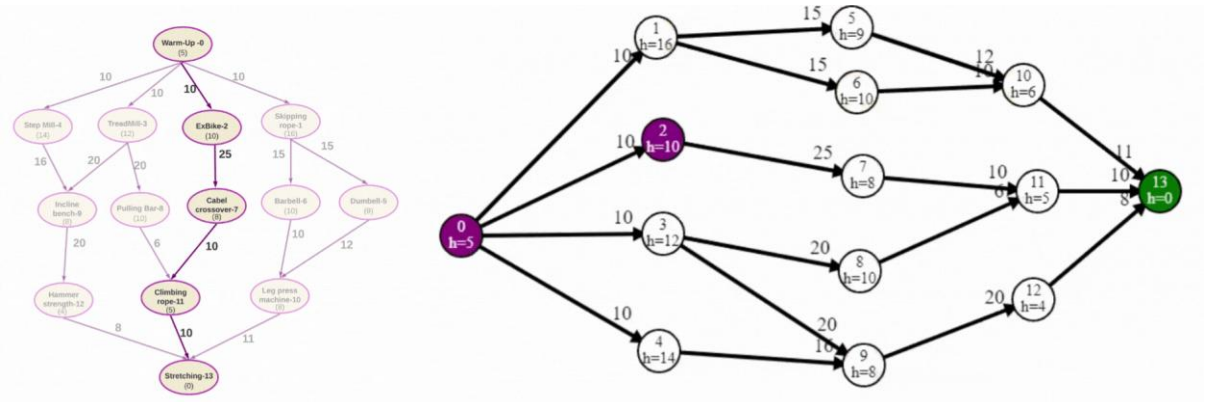


Figure 12: State-Space (left) and state-space traversal (right) in greedy search

f(n) = h(n)

= Summation (Heuristic values of nodes)

= Node0 + Node2 + Node7 + Node11 + Node13

= 5 + 10 + 8 + 5 + 0

= 28

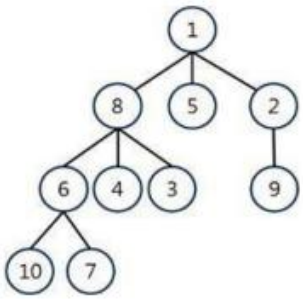
Path: 0, 2, 7, 11, 13

In greedy search, you can see we do not visit all the child nodes of a particular node. We find heuristics of each child node only the one with the lowest value is inserted to the OPEN list to be processed. Therefore greedy search is not complete. As well as this is not the best path (not the shortest path - we found this path in uniform cost search). Therefore greedy search is not optimal, also. As a solution, we should consider not only the heuristic value but also the path cost. Here, comes the A* algorithm.

4. a. Discuss the following search Technique with the help of a given tree. Also discuss the benefits and Application of each.

i. Breadth First Search

ii. Depth First Search



Answer:-

i. Breadth First Search

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.

- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.

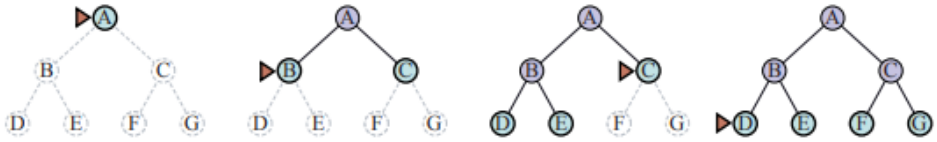
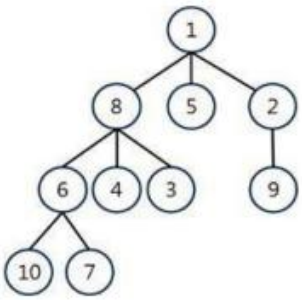


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

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

Example:



In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node 1 to the bottom of the tree. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

1---> 8--->5---->2--->6---->4--->3--->9---->10---->7

Advantages of BFS:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

- It also helps in finding the shortest path in goal state, since it needs all nodes at the same hierarchical level before making a move to nodes at lower levels.
- It is also very easy to comprehend with the help of this we can assign the higher rank among path types.

Applications Of Breadth-First Search Algorithm

- ✓ **GPS Navigation systems:** Breadth-First Search is one of the best algorithms used to find Neighbouring locations by using the GPS system.
- ✓ **Broadcasting:** Networking makes use of what we call as packets for communication. These packets follow a traversal method to reach various networking nodes. One of the most commonly used traversal methods is Breadth-First Search. It is being used as an algorithm that is used to communicate broadcasted packets across all the nodes in a network.

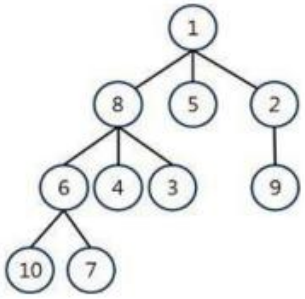
ii. Depth First Search

- Depth-first search is a recursive algorithm for traversing a tree or graph data structure. It generally starts by exploring the deepest node in the frontier. Starting at the root node, the algorithm proceeds to search to the deepest level of the search tree until nodes with no successors are reached. Suppose the node with unexpanded successors is encountered then the search backtracks to the next deepest node to explore alternative paths.
- 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 (LIFO) for its implementation.
- 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).

Example:

- In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:
- Root node--->Left node ----> right node.
- Let's consider 9 is the goal node.
- It will start searching from root node 1, and traverse 8, then 6, then 10 and 7, after traversing 7, it will backtrack to 6 then 8 the tree as 7 has no other successor.

- After backtracking it will traverse node 4 and then 3, after traversing 3, it will backtrack to 8 then 1 the tree as 3 has no other successor.
- After backtracking it will traverse node 2 and then 9 and here it will terminate as it found goal node.



Order of Visited Nodes:

$1 \rightarrow 8 \rightarrow 6 \rightarrow 10 \rightarrow 7 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow 2 \rightarrow 9$

Advantage of DFS:

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

Applications Of Depth-First Search Algorithm

- ✓ **Finding Strongly Connected Components of a graph:** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.
- ✓ **Web crawlers:** Depth-first search can be used in the implementation of web crawlers to explore the links on a website.
- ✓ **Model checking:** Depth-first search can be used in model checking, which is the process of checking that a model of a system meets a certain set of properties.

b. Define heuristic search? Explain different types of heuristic search with suitable examples.

Answer:- A heuristic search/ informed search is a strategy used in AI to optimize the search process by using a heuristic function to estimate the cost of

reaching a goal. Instead of exhaustively exploring all possible paths, a heuristic search uses this estimate to prioritize the most promising paths, reducing computational complexity and speeding up decision-making.

In informed search algorithms additional information is used to make the search more efficient and effective. That additional information is called **heuristics**. Heuristics are not theories but some common sense experience like information.

In informed search algorithms, to find the best node to be visited next, we use an *evaluation function* **f(n)** which assists the child node to decide on which node to be visited next. Then we traverse to the next node with the **least f(n)** value. Depending on the f(n), we have two informed search algorithms as greedy search and A* search algorithms.

2.1 Greedy Search Algorithms

In greedy search, the heuristic values of child nodes are considered. The path is determined by calculating the path with the nodes with the lowest heuristic values. Another fact to be noticed is that usually the initial node has the highest heuristic value and the goal node has the lowest. But there can be exceptions like getting a mid-range value for the initial node also.

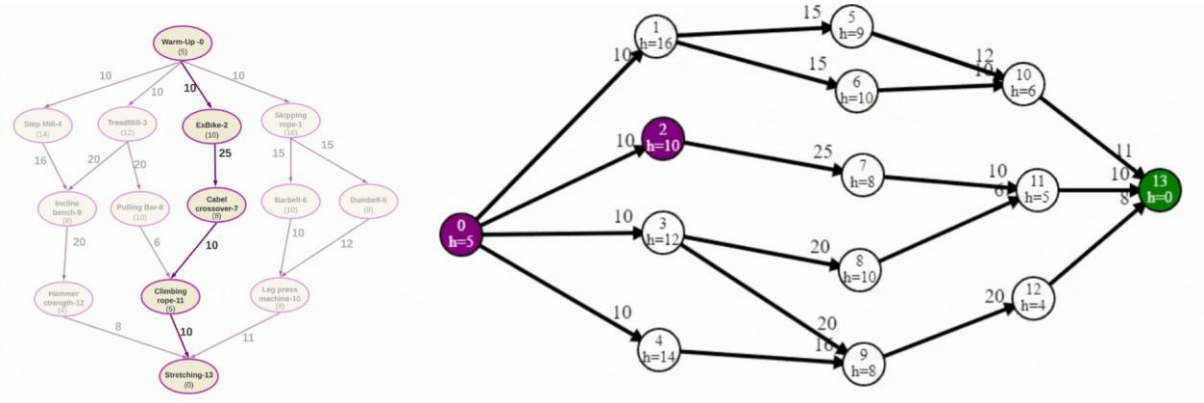


Figure 12: State-Space (left) and state-space traversal (right) in greedy search

f(n) = h(n)

= Summation (Heuristic values of nodes)

= Node0 + Node2 + Node7 + Node11 + Node13

= 5 + 10 + 8 + 5 + 0

= 28

Path: 0, 2, 7, 11, 13

In greedy search, you can see we do not visit all the child nodes of a particular node. We find heuristics of each child node only the one with the lowest value is inserted to the OPEN list to be processed. Therefore greedy search is not complete. As well as this is not the best path (not the shortest path - we found this path in uniform cost search). Therefore greedy search is not optimal, also. As a solution, we should consider not only the heuristic value but also the path cost. Here, comes the A* algorithm.

2.2 A* Search Algorithms

In the A* algorithm, we consider both path cost and heuristics. In A* the f(n) function comprises two components: path cost [g(n)] and heuristic value ([h(n)]. The f(n) value for node ‘a’ can be calculated as follows:

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

$f(n)_a$ = Evaluation value at the particular node (node 'a')
$g(n)_a$ = Total path cost from start node to particular node(node 'a')
$h(n)_a$ = The heuristic value of the particular node(node 'a')

Now, we will find the best path according to the A* search algorithm for our previous problem. We have to find the f(n) value for each node and select the child node with the least f(n) value and traverse until meeting the goal node. In the example, only the f(n) values of goals in the path are mentioned.

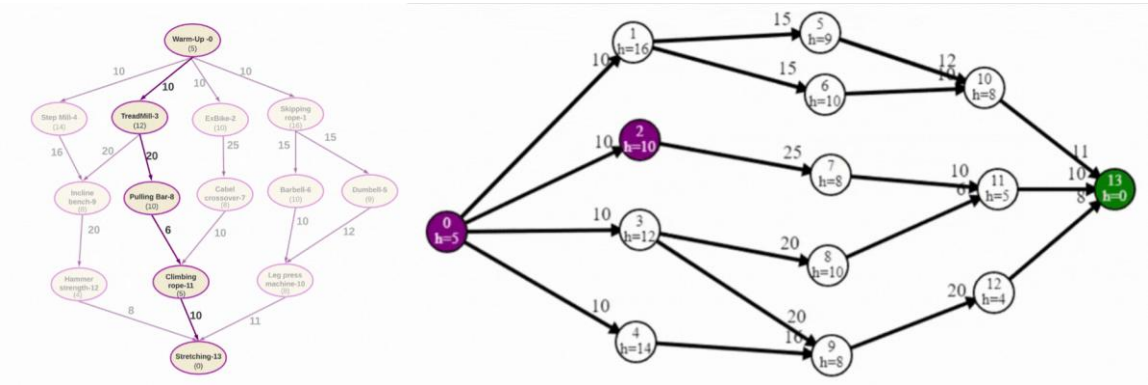


Figure 13: State-Space (left) and state-space traversal (right) in A* search

At Node-0 :

$f(n)_o = 0 + 5 = 5$

At Node-3:

$f(n)_3 = 10 + 12 = 22$

At Node-8:

$f(n)_8 = 30 + 10 = 40$

At Node-11:

$f(n)_{11} = 36 + 5 = 41$

At Node-13:

$f(n)_{13} = 46 + 0 = 46$

$f(n)_{total} = f(n)_0 + f(n)_3 + f(n)_8 + f(n)_{11} + f(n)_{13} = 154$

Path: 0, 3, 8, 11, 13

A* algorithm is complete since it checks all the nodes before reaching to goal node/end of the state space. It is optimal as well because considering both path cost and heuristic values, therefore the lowest path with the lowest heuristics can be found with A*. One drawback of A* is it stores all the nodes it processes in the memory. Therefore, for a state space of branching factor 'b', and the depth 'd' the space and time complexities of A* are denoted by **O(b^d)**. The time complexity of A* depends on the heuristic values.

c. Briefly explain Hill Climbing search and Simulated annealing algorithm with suitable example.

Answer:-

Local search algorithms are essential tools in artificial intelligence and optimization, employed to find high-quality solutions in large and complex problem spaces. Key algorithms include Hill-Climbing Search, Simulated Annealing, Local Beam Search, Genetic Algorithms, and Tabu Search.

Each of these methods offers unique strategies and advantages for solving optimization problems.

1. 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 neighbour has a higher value.

- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance travelled by the salesman.
- It is also called greedy local search as it only looks to its good immediate neighbour 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.
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

Features of Hill Climbing:

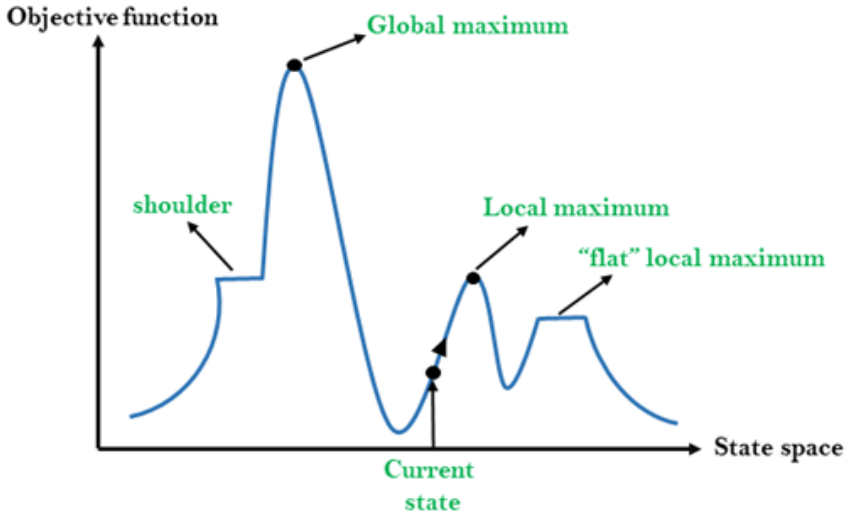
Following are some main features of Hill Climbing Algorithm:

- **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.
- **Deterministic Nature:**
  Hill Climbing is a deterministic optimization algorithm, which means that given the same initial conditions and the same problem, it will always produce the same result. There is no randomness or uncertainty in its operation.
- **Local Neighbourhood:**
  Hill Climbing is a technique that operates within a small area around the current solution. It explores solutions that are closely related to the current state by making small, gradual changes. This approach allows it to find a solution that is better than the current one although it may not be the global optimum.

State-space Diagram for Hill Climbing:

The state-space landscape is a graphical representation of the hill-climbing algorithm which is showing a graph between various states of algorithm and Objective function/Cost.

On Y-axis we have taken the function which can be an objective function or cost function, and state-space on the x-axis. If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum. If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.



Different regions in the state space landscape:

Local Maximum: Local maximum is a state which is better than its neighbour 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.

Example:

To illustrate hill climbing, we will use the 8-queens problem (Figure 4.3). We will use a complete-state formulation, which means that every state has all the components of a solution, but they might not all be in the right place. In this case every state has 8 queens on the board, one per column. The initial state is chosen at random, and the successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has

8×7=56 successors). The heuristic cost function h is the number of pairs of queens that are attacking each other; this will be zero only for solutions. (It counts as an attack if two pieces are in the same line, even if there is an intervening piece between them.) Figure 4.3(b) shows a state that has h=17. The figure also shows the h values of all its successors.

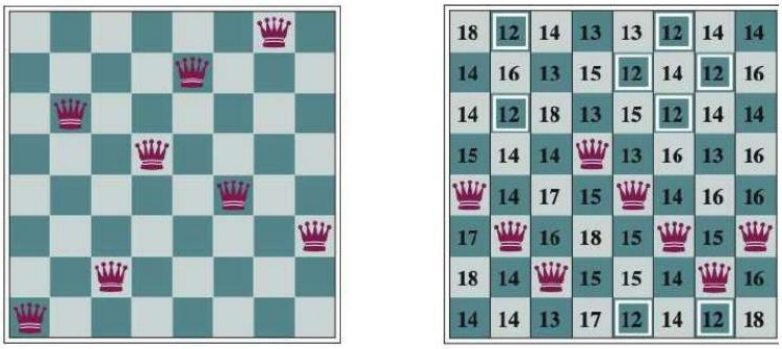


Figure 4.3 (a) The 8-queens problem: place 8 queens on a chess board so that no queen attacks another. (A queen attacks any piece in the same row, column, or diagonal.) This position is almost a solution, except for the two queens in the fourth and seventh columns that attack each other along the diagonal. (b) An 8-queens state with heuristic cost estimate $h=17$. The board shows the value of h for each possible successor obtained by moving a queen within its column. There are 8 moves that are tied for best, with $h=12$. The hill-climbing algorithm will pick one of these.

2. **Simulated Annealing Search**

A hill-climbing algorithm that never makes —downhill moves towards states with lower value (or higher cost) is guaranteed to be incomplete, because it can stuck on a local maximum. In contrast, a purely random walk –that is, moving to a successor chosen uniformly at random from the set of successors – is complete, but extremely inefficient. Simulated annealing is an algorithm that combines hill-climbing with a random walk in some way that yields both

Efficiency and completeness.

simulated annealing algorithm is quite similar to hill climbing. Instead of picking the best move, however, it picks the random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1.

The probability decreases exponentially with the —badness of the move – the amount E by which the evaluation is worsened. The probability also decreases as the "temperature" T goes down: "bad moves are more likely to be allowed at

the start when temperature is high, and they become more unlikely as T decreases. One can prove that if the schedule lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

Simulated annealing was first used extensively to solve VLSI layout problems. It has been applied widely to factory scheduling and other large-scale optimization tasks.

SA is a metaheuristic optimization technique introduced by Kirkpatrick et al. in 1983 to solve the Travelling Salesman Problem (TSP).

The SA algorithm is based on the annealing process used in metallurgy, where a metal is heated to a high temperature quickly and then gradually cooled. At high temperatures, the atoms move fast, and when the temperature is reduced, their kinetic energy decreases as well. At the end of the annealing process, the atoms fall into a more ordered state, and the material is more ductile and easier to work with.

Similarly, in SA, a search process starts with a high-energy state (an initial solution) and gradually lowers the temperature (a control parameter) until it reaches a state of minimum energy (the optimal solution).

Advantages of Simulated Annealing

- **Ability to Escape Local Minima**: One of the most significant advantages of Simulated Annealing is its ability to escape local minima. The probabilistic acceptance of worse solutions allows the algorithm to explore a broader solution space.
- **Simple Implementation**: The algorithm is relatively easy to implement and can be adapted to a wide range of optimization problems.
- **Global Optimization**: Simulated Annealing can approach a global optimum, especially when paired with a well-designed cooling schedule.
- **Flexibility:** The algorithm is flexible and can be applied to both continuous and discrete optimization problems.

Unit-2

1.

a. Define Minimax Algorithm.

Answer:- The Mini-Max algorithm is a decision-making algorithm used in artificial intelligence, particularly in game theory and computer games. It is designed to minimize the possible loss in a worst-case scenario (hence "min") and maximize the potential gain (therefore "max").

b. What are the Challenges in Optimal Decision-Making?

Answer:-

- ✓ Complexity of Game Environments
- ✓ Real-Time Decision Making
- ✓ Incomplete Information
- ✓ Dynamic and Evolving Environments

c. Define Alpha and Beta.

- **Answer:- Alpha:** Alpha represents the best score that the maximizing player has found so far in a particular branch of the game tree. It is the highest score the maximizing player can achieve up to this point. Essentially, it tracks the maximizer's best-known option.
- **Beta:** On the other hand, Beta represents the best score that the minimizing player has found in a specific branch. It is the lowest score the minimizing player can allow up to this point. Beta tracks the minimizer's best-known option.

The main condition which required for alpha-beta pruning is:

- $\alpha >= \beta$

d. What are types of Constraint Satisfaction Problems?

Answer:-

- ✓ Binary CSPs
- ✓ Non-Binary CSPs
- ✓ Hard and Soft Constraints

e. State De Morgan's Laws.

Answer:- These laws describe how **negations** distribute over **AND (∧)** and **OR (∨)** operations:

- **First Law:**
  - ¬(P∧Q)≡(¬P∨¬Q)

This means that the negation of a conjunction is equivalent to the disjunction of the negated propositions.

- **Second Law:**
  - ¬(P∨Q)≡(¬P∧¬Q)

This means that the negation of a disjunction is equivalent to the conjunction of the negated propositions.

2.

a. Write a short note on Minimax algorithm with example.

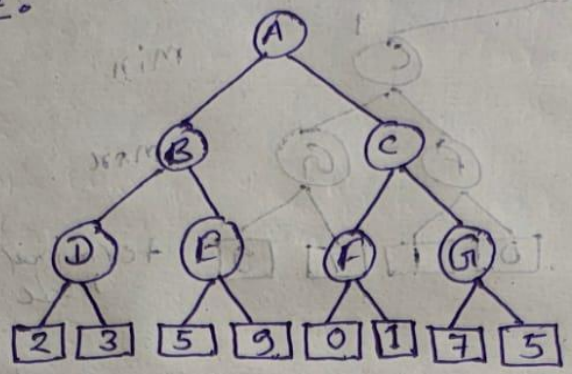
Answer:- The Min Max algorithm is a decision-making algorithm used in the field of game theory and artificial intelligence. It is used to determine the optimal strategies in games like chess, checkers, and tic-tac-toe. In a two-player game, one player takes on the role of the maximizer, seeking the best move to maximize their chances of winning, while the other player acts as the minimizer, attempting to minimize the maximizer's chances.

Working of Min-Max Algorithm:

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

Example:

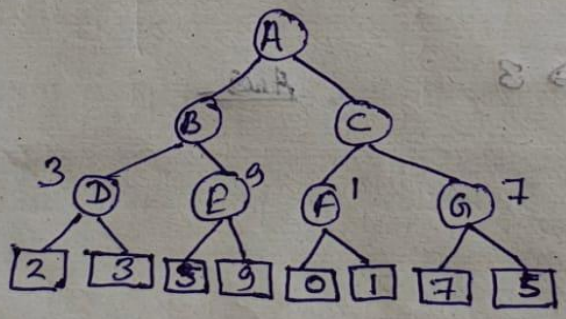
Example:



→ Maximizer
→ Minimizer
→ Maximizer
→ Terminal node

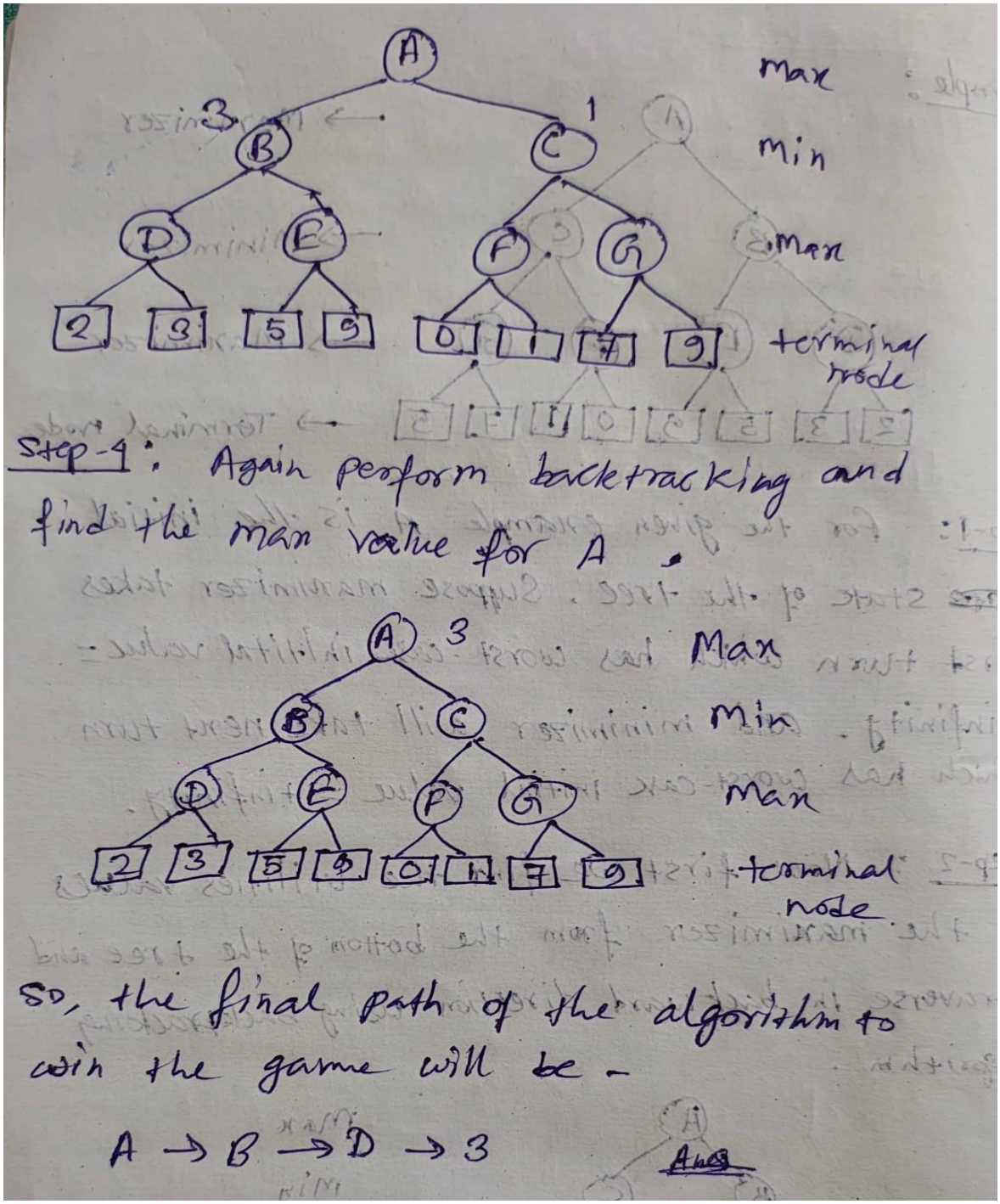
Step-1: For the given example A is the initial ~~state~~ state of the tree. Suppose maximizer takes first turn which has worst-case initial value = -infinity. and minimizer will take next turn which has worst-case initial value = +infinity.

Step-2: Now, first we find the utilities values of the maximizer from the bottom of the tree and traverse in backward direction using backtracking algorithm.



Max
Min
Max
terminal node

Step-3: Again, we have to perform backtracking and find the min value for B and C.



b. Discuss about Constraint Satisfaction Problems (CSP).

Answer:- Constraint Satisfaction Problems (CSP) represent a class of problems where the goal is to find a solution that satisfies a set of constraints. These problems are commonly encountered in fields like scheduling, planning, resource allocation, and configuration.

A **Constraint Satisfaction Problem** is a mathematical problem where the solution must meet a number of constraints. In a CSP, the objective is to assign values to variables such that all the constraints are satisfied. CSPs are used

extensively in artificial intelligence for decision-making problems where resources must be managed or arranged within strict guidelines.

Components of Constraint Satisfaction Problems

CSPs are composed of three key elements:

1. **Variables:** The things that need to be determined are variables. Variables in a CSP are the objects that must have values assigned to them in order to satisfy a particular set of constraints. Boolean, integer, and categorical variables are just a few examples of the various types of variables, for instance, could stand in for the many puzzle cells that need to be filled with numbers in a sudoku puzzle.
2. **Domains:** The range of potential values that a variable can have is represented by domains. Depending on the issue, a domain may be finite or limitless. For instance, in Sudoku, the set of numbers from 1 to 9 can serve as the domain of a variable representing a problem cell.
3. **Constraints:** The guidelines that control how variables relate to one another are known as constraints. Constraints in a CSP define the ranges of possible values for variables. Unary constraints, binary constraints, and higher-order constraints are only a few examples of the various sorts of constraints. For instance, in a sudoku problem, the restrictions might be that each row, column, and 3×3 box can only have one instance of each number from 1 to 9.

Common applications of CSPs include:

- **Scheduling:** Assigning resources like employees or equipment while respecting time and availability constraints.
- **Planning:** Organizing tasks with specific deadlines or sequences.
- **Resource Allocation:** Distributing resources efficiently without overuse.

c. With a neat diagram explain about the architecture of knowledge-based agent.

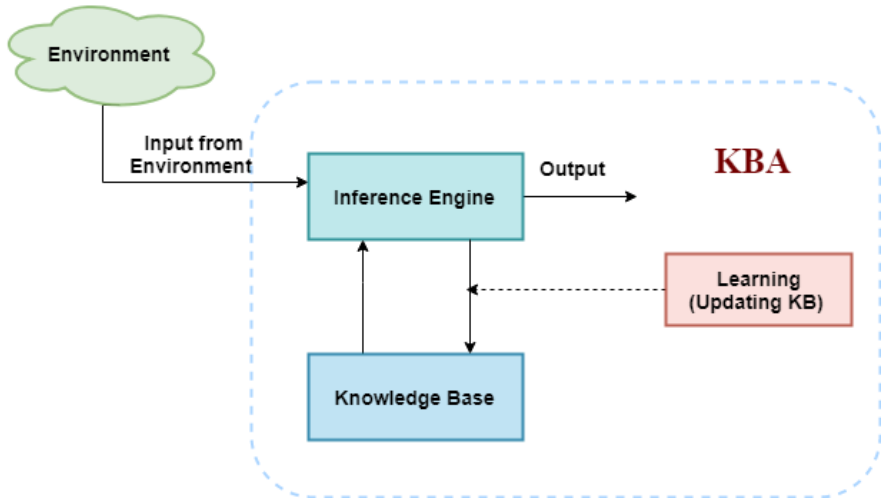
- **Answer:-** Knowledge-based agents are those agents who have the capability of **maintaining an internal state of knowledge, reason over that knowledge, update their knowledge after observations and take actions. These agents can represent the world with some formal representation and act intelligently**.

- Knowledge-based agents are composed of two main parts:
  - **Knowledge-base and**
  - **Inference system**.

A knowledge-based agent must able to do the following:

- An agent should be able to represent states, actions, etc.
- An agent Should be able to incorporate new percepts
- An agent can update the internal representation of the world
- An agent can deduce the internal representation of the world
- An agent can deduce appropriate actions.

The architecture of knowledge-based agent:



The above diagram is representing a generalized architecture for a knowledge-based agent. The knowledge-based agent (KBA) take input from the environment by perceiving the environment. The input is taken by the inference engine of the agent and which also communicate with KB to decide as per the knowledge store in KB. The learning element of KBA regularly updates the KB by learning new knowledge.

d. Explain logical connectives with truth table in propositional logic?

Answer:- Logical connectives are essential operators that combine **atomic propositions** to form **compound propositions**. These connectives allow AI

systems to build more complex rules and perform logical reasoning. Below are the most common connectives used in **propositional logic**:

Connective symbols	Word	Technical term	Example
∧	AND	Conjunction	A ∧ B
∨	OR	Disjunction	A ∨ B
→	Implies	Implication	A → B
⇔	If and only if	Biconditional	A⇔ B
¬ or ~	Not	Negation	¬ A or ¬ B

Truth Table:

In propositional logic, we need to know the truth values of propositions in all possible scenarios. We can combine all the possible combination with logical connectives, and the representation of these combinations in a tabular format is called Truth table. Following are the truth table for all logical connectives:

For Negation:

P	¬ P
True	**False**
False	**True**

For Conjunction:

P	Q	P∧ Q
True	True	**True**
True	False	**False**
False	True	**False**
False	False	**False**

For disjunction:

P	Q	P ∨ Q.
True	True	**True**
False	True	**True**
True	False	**True**
False	False	**False**

For Implication:

P	Q	P→ Q
True	True	**True**
True	False	**False**
False	True	**True**
False	False	**True**

For Biconditional:

P	Q	P⇔ Q
True	True	**True**
True	False	**False**
False	True	**False**
False	False	**True**

e. Describe the Properties of Operators in Propositional Logic.

Answer:- In propositional logic, **logical operators** follow specific properties that allow us to **manipulate and simplify logical expressions**. Understanding these properties is essential for building efficient AI systems that rely on logical reasoning.

1. De Morgan's Laws

These laws describe how **negations** distribute over **AND (∧)** and **OR (∨)** operations:

- **First Law:**
  - $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$

This means that the negation of a conjunction is equivalent to the disjunction of the negated propositions.

- **Second Law:**
  - $\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$

This means that the negation of a disjunction is equivalent to the conjunction of the negated propositions.

2. Commutative Property

This property states that the **order of the propositions** does not affect the result of **AND (∧)** and **OR (∨)** operations:

- **AND:**
  - $P \wedge Q \equiv Q \wedge P$
- **OR:**
  - $P \vee Q \equiv Q \vee P$

3. Associative Property

This property allows us to **group propositions** in any order when using **AND** or **OR** operations:

- **AND:**
  - $(P \wedge Q) \wedge R \equiv P \wedge (Q \wedge R)$
- **OR:**
  - $(P \vee Q) \vee R \equiv P \vee (Q \vee R)$

4. Distributive Property

This property states that **AND distributes over OR**, and vice versa:

- **AND over OR:**
  - P∧(Q∨R)≡(P∧Q)∨(P∧R)P
- **OR over AND:**
  - P∨(Q∧R)≡(P∨Q)∧(P∨R)

3.

a. Explain Alpha Beta Pruning Algorithm with an Example.

Answer:-

b. Explain about Propositional logic.

Answer:-

Propositional Logic is a formal system used in Artificial Intelligence (AI) to represent and reason about facts in a system. It deals with propositions, which are statements that can be either true or false. Propositional logic is used to build logical expressions using connectives like AND (∧), OR (∨), NOT (¬), IMPLIES (→), and IF AND ONLY IF (↔).

Key Components:

1. **Propositions:** Basic statements that can be true or false, e.g., "It is raining" or "John is at home."
2. **Logical Connectives:** These combine propositions to form more complex expressions:
   - AND (∧): Both propositions must be true.
   - OR (∨): At least one proposition must be true.
   - NOT (¬): Negates the truth value of a proposition.
   - IMPLIES (→): If the first proposition is true, the second must also be true.
3. **Truth Tables:** Used to evaluate the truth value of logical expressions based on all possible combinations of truth values of propositions.

Applications in AI:

- **Knowledge Representation:** Propositional logic represents facts and rules.
- **Reasoning**: It is used in inference systems to derive new facts from known facts.
- **Decision Making:** Helps agents decide on actions based on logical conclusions.

Despite its simplicity, propositional logic is limited in representing complex relationships, which is addressed by more advanced systems like First-Order Logic.

Applications of Propositional Logic in AI

1. **Knowledge Representation**:
   - Propositional logic is often used to represent facts about the world. Each proposition can represent a piece of knowledge or fact.
2. **Automated Reasoning and Inference**:
   - Propositional logic forms the basis for reasoning systems that deduce new facts or conclusions from known facts.
3. **Decision Making and Planning**:
   - Propositional logic is used in **decision-making** processes where an agent has to make decisions based on the logical relationships between propositions.
4. **Expert Systems**:
   - Propositional logic is used in expert systems to represent the knowledge base of an expert in a particular domain.
5. **Automated Theorem Proving**:
   - Propositional logic is used in automated theorem proving to prove or disprove statements using logical rules. This is essential in formal verification of systems and software.

c. Explain Propositional theorem proving.

Answer:-

Propositional Theorem Proving is a method in Artificial Intelligence (AI) used to determine whether a given logical formula (theorem) can be derived from a set of axioms or premises using formal logical rules. It plays a crucial role in automated reasoning, verification, and decision-making processes within AI systems.

Key Concepts:

1. **Propositions:**
   - These are atomic statements that are either true or false. For example, P, Q, and R can represent individual facts such as "It is raining" or "The light is on."

2. **Inference Rules:**
    - These are logical principles used to derive new facts from existing premises or axioms. Common rules include:
        - Modus Ponens: If P→Q and P are true, then Q is true.
        - Modus Tollens: If P→Q and ¬Q are true, then ¬ P is true.
3. **Theorem:**
    - A theorem is a formula that needs to be proven, typically derived from a set of axioms or facts. In AI, theorem proving is used to verify the correctness of solutions, check consistency in knowledge bases, or infer new knowledge.
4. **Axioms (Premises):**
    - These are a set of assumed truths from which theorems can be derived. They serve as the foundation for logical reasoning in propositional logic**.**

Common Methods for Propositional Theorem Proving:

1. **Truth Table Method:**
    - A truth table lists all possible truth values of the propositions and evaluates the formula based on these values. The formula is a theorem if it holds true for all possible truth assignments.
    - Limitations: For formulas with many propositions, the number of rows in the truth table grows exponentially, making it inefficient for complex formulas.
2. **Semantic Tableaux (Truth Tree):**
    - The formula is broken down into simpler components. If contradictions are found during the decomposition process, the formula is not valid. Otherwise, it is considered valid.
    - Process: The formula is negated, and the proof proceeds by systematically checking for contradictions. If no contradictions appear, the formula is a theorem.
3. **Resolution:**
    - The formula is transformed into Conjunctive Normal Form (CNF), where the formula is expressed as a conjunction of disjunctions. The resolution rule is applied to combine clauses and derive new clauses. If an empty clause (a contradiction) is derived, the formula is unsatisfiable, proving the negation is false, and the theorem is true.

- Efficient Method: Resolution is widely used in automated theorem proving systems and SAT solvers, as it allows for efficient search strategies in large knowledge bases.

4. **Davis-Putnam-Logemann-Loveland (DPLL) Algorithm:**
   - The DPLL algorithm is an efficient procedure for solving the Satisfiability (SAT) problem, which is central to propositional theorem proving. It simplifies the formula using backtracking, unit propagation, and pure literal elimination to find a solution or prove the formula unsatisfiable.

Applications in AI:

1. Automated Reasoning
2. Expert Systems
3. SAT Solvers
4. Formal Verification

d. Explain constraint satisfaction problem with graph coloring as example.

Answer:- For example, let us formulate the map coloring problem for the map of Australia, as shown below:
The CSP formulation for this problem is:

- X: {WA, NT, SA, Q, NSW, V, T}, where each variable represents a state or territory of Australia.
- D: {red, green, blue}, where each variable has the same domain of three colors.
- C: {< (WA, NT), WA!= NT >, < (WA, SA), WA!= SA >, < (NT, SA), NT!= SA >, < (NT, Q), NT!= Q >, < (SA, Q), SA!= Q >, < (SA, NSW), SA!= NSW >, < (SA, V), SA!= V >, < (Q, NSW), Q!= NSW >, < (NSW, V), NSW!= V >}, where each constraint is a binary constraint that states that two adjacent regions must have different colors.
  The algorithm starts with an empty assignment and selects the first variable to assign. According to the MRV heuristic, the variable with the

most constraints is SA, as it has six neighbors. The algorithm then tries to assign a value to SA, and according to the LCV heuristic, it chooses red, as it is the least constraining value for the neighboring variables. The algorithm then uses forward checking to prune the domains of the neighboring variables and updates the domains as follows:

- **WA: {green, blue}**
- **NT: {green, blue}**
- **Q: {green, blue}**
- **NSW: {green, blue}**
- **V: {green, blue}**
- **T: {red, green, blue}**

  The algorithm then recurses to the next level and selects the next variable to assign. According to the MRV heuristic, the variable with the most constraints is NT, as it has three neighbors. The algorithm then tries to assign a value to NT, and according to the LCV heuristic, it chooses green, as it is the least constraining value for the neighboring variables. The algorithm then uses forward checking to prune the domains of the neighboring variables and updates the domains as follows:

- **WA: {blue}**
- **Q: {blue}**
- **SA: {red}**
- **NSW: {green, blue}**
- **V: {green, blue}**
- **T: {red, green, blue}**

  The algorithm then recurses to the next level and selects the next variable to assign. According to the MRV heuristic, the variable with the most constraints is WA, as it has two neighbors. The algorithm then tries to assign a value to WA, and according to the LCV heuristic, it chooses blue, as it is the only remaining value for WA. The algorithm then uses forward checking to prune the domains of the neighboring variables and updates the domains as follows:

- **NT: {green}**
- **SA: {red}**
- **Q: {blue}**
- **NSW: {green}**
- **V: {green, blue}**
- **T: {red, green, blue}**

e. Explain the Horn Clauses and Definite Clauses in detail.

Answer:-

Horn Clauses and Definite Clauses are fundamental concepts in logic programming and artificial intelligence, particularly in knowledge representation and reasoning. These clauses are a special kind of logic formula used to simplify reasoning tasks and make logical inference more efficient. They are primarily used in Prolog and other logic programming languages to represent rules and facts.

Horn Clauses

A Horn clause is a special form of a clause (a disjunction of literals) in propositional or first-order logic. A Horn clause has at most one positive literal (i.e., a literal that is not negated) and any number of negative literals (literals that are negated).

- **Definition: A Horn clause is a disjunction of literals of the form:**

  $$\neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_n \vee Q$$

  where:
  - $P_1, P_2, \ldots, P_n$ are negative literals (i.e., negated propositions).
  - Q is a positive literal (i.e., a non-negated proposition).
- **Key Characteristics:**
  - At most one positive literal: Horn clauses have one or zero positive literals.
  - A rule: Horn clauses are used to represent rules in logic programming. They are often written as implications, where the left-hand side represents the conditions (antecedent), and the right-hand side represents the conclusion (consequent).
- **Example:**

  1. $\neg P \vee \neg Q \vee R$ (which can be interpreted as: "If P and Q are both false, then R is true").
  2. $P \rightarrow Q$ is equivalent to $\neg P \vee Q$.

Definite Clauses

A Definite Clause is a specific type of Horn Clause where there is exactly one positive literal. A definite clause is a disjunction of literals with exactly one positive literal and any number of negative literals.

- **Definition: A Definite Clause is a Horn clause where there is exactly one positive literal:**

  $$\neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_n \vee Q$$

where:

- $P1, P2, \ldots, Pn$ are negative literals (negated propositions).
- Q is the single positive literal (non-negated proposition).

- **Key Characteristics:**
  - Exactly one positive literal: Definite clauses always have a single positive literal.
  - Implication form: Definite clauses are typically used to represent implications where the conditions (on the left-hand side) lead to a conclusion (on the right-hand side).
  - Used in Logic Programming: Definite clauses are the core building blocks of programs in languages like Prolog. Each rule in Prolog can be represented as a definite clause.
- **Example:**
  - $\neg P \vee \neg Q \vee R$ is a definite clause and can be interpreted as: "If P and Q are false, then R is true."
  - $P \rightarrow Q$ is equivalent to $\neg P \vee Q$, which is also a definite clause.

4.

a. Write short note on the following Algorithm:

i. Backtracking Algorithm

ii. Forward-Checking Algorithm

iii. Constraint Propagation Algorithm

Answer:-

1. Backtracking Algorithm

The **backtracking algorithm** is a depth first search method used to systematically explore possible solutions in CSPs. It operates by assigning values to variables and backtracks if any assignment violates a constraint.

How it works:

- The algorithm selects a variable and assigns it a value.
- It recursively assigns values to subsequent variables.
- If a conflict arises (i.e., a variable cannot be assigned a valid value), the algorithm backtracks to the previous variable and tries a different value.

- The process continues until either a valid solution is found or all possibilities have been exhausted.

This method is widely used due to its simplicity but can be inefficient for large problems with many variables.

2. Forward-Checking Algorithm

The **forward-checking algorithm** is an enhancement of the backtracking algorithm that aims to reduce the search space by applying **local consistency** checks.

How it works:

- For each unassigned variable, the algorithm keeps track of remaining valid values.
- Once a variable is assigned a value, local constraints are applied to neighbouring variables, eliminating inconsistent values from their domains.
- If a neighbor has no valid values left after forward-checking, the algorithm backtracks.

This method is more efficient than pure backtracking because it prevents some conflicts before they happen, reducing unnecessary computations.

3. Constraint Propagation Algorithms

Constraint propagation algorithms further reduce the search space by enforcing **local consistency** across all variables.

How it works:

- Constraints are propagated between related variables.
- Inconsistent values are eliminated from variable domains by leveraging information gained from other variables.
- These algorithms refine the search space by making **inferences**, removing values that would lead to conflicts.

Constraint propagation is commonly used in conjunction with other CSP algorithms, such as **backtracking**, to increase efficiency by narrowing down the solution space early in the search process.

b. What is knowledge-based agent in artificial intelligence? Why we use a knowledge base? What are the various levels of knowledge-based agent? Write any two approaches to designing a knowledge-based agent?

Answer:- Knowledge-based agents are those agents who have the capability of **maintaining an internal state of knowledge, reason over that knowledge, update their knowledge after observations and take actions. These agents can represent the world with some formal representation and act intelligently**.

- Knowledge-based agents are composed of two main parts:
  - **Knowledge-base and**
  - **Inference system**.

Why use a knowledge base?

- Knowledge-base is required for updating knowledge for an agent to learn with experiences and take action as per the knowledge.

Various levels of knowledge-based agent:

A knowledge-based agent can be viewed at different levels which are given below:

1. Knowledge level

Knowledge level is the first level of knowledge-based agent, and in this level, we need to specify what the agent knows, and what the agent goals are. With these specifications, we can fix its behavior. For example, suppose an automated taxi agent needs to go from a station A to station B, and he knows the way from A to B, so this comes at the knowledge level.

2. Logical level:

At this level, we understand that how the knowledge representation of knowledge is stored. At this level, sentences are encoded into different logics. At the logical level, an encoding of knowledge into logical sentences occurs. At the logical level we can expect to the automated taxi agent to reach to the destination B.

3. Implementation level:

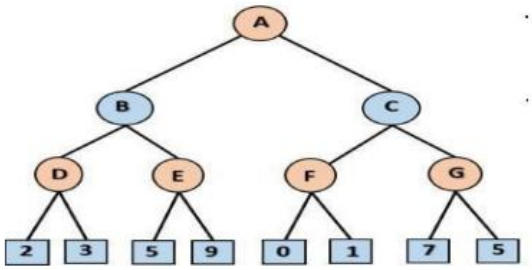
This is the physical representation of logic and knowledge. At the implementation level agent perform actions as per logical and knowledge level. At this level, an automated taxi agent actually implement his knowledge and logic so that he can reach to the destination.

Approaches to designing a knowledge-based agent:

There are mainly two approaches to build a knowledge-based agent:

1. **1. Declarative approach:** We can create a knowledge-based agent by initializing with an empty knowledge base and telling the agent all the sentences with which we want to start with. This approach is called Declarative approach.
2. **2. Procedural approach:** In the procedural approach, we directly encode desired behavior as a program code. Which means we just need to write a program that already encodes the desired behavior or agent.

c. Analyze alpha-beta pruning algorithm and the Min-max game playing algorithm for a given tree.



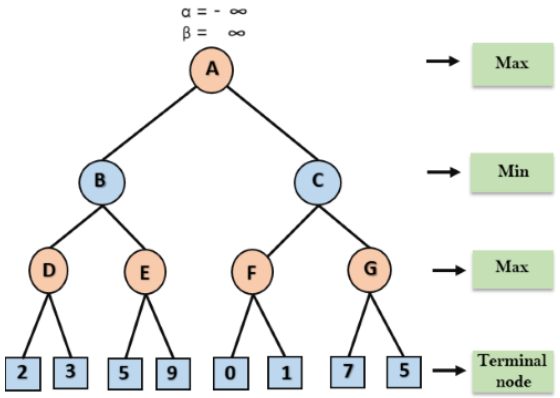
Answer:- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.

- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**. This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.

- **Alpha:** Alpha represents the best score that the maximizing player has found so far in a particular branch of the game tree. It is the highest score the maximizing player can achieve up to this point. Essentially, it tracks the maximizer's best-known option.
- **Beta:** On the other hand, Beta represents the best score that the minimizing player has found in a specific branch. It is the lowest score the minimizing player can allow up to this point. Beta tracks the minimizer's best-known option.
- The main condition which required for alpha-beta pruning is: α>=β

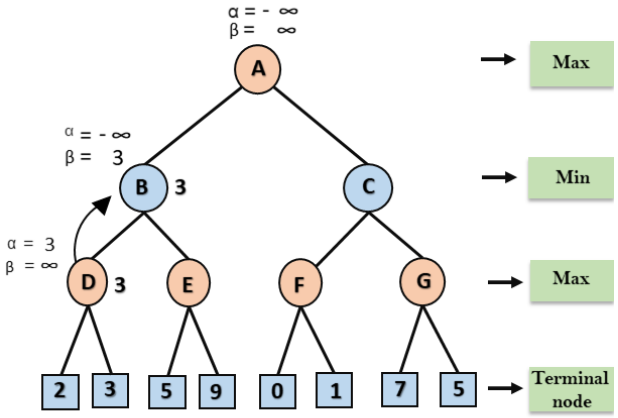
Example:

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



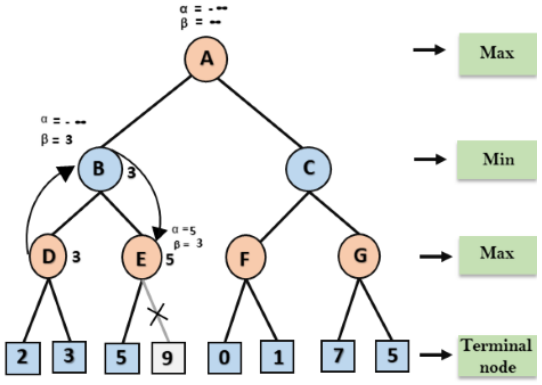
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

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



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

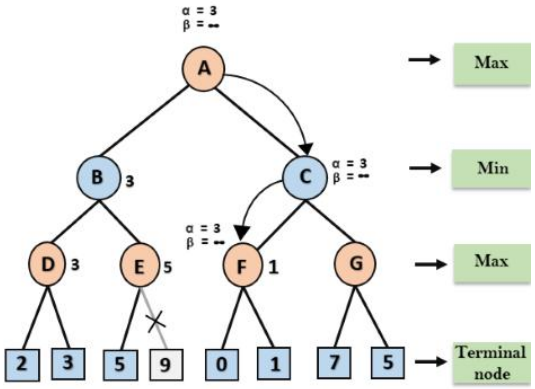
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will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



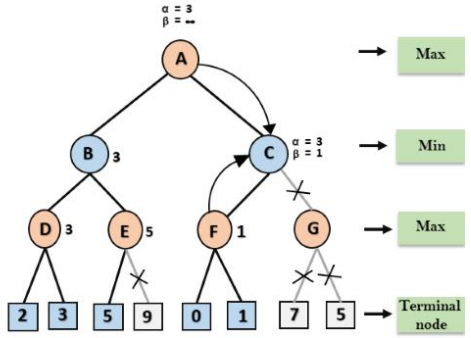
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

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

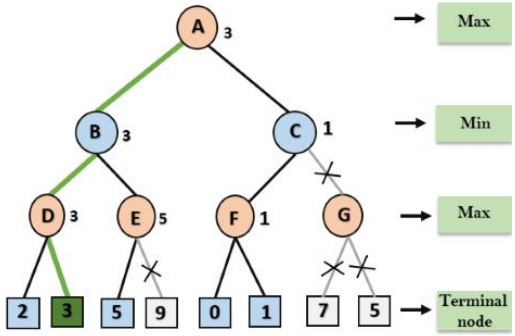
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



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



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



Unit-3

1.

a. Specify the syntax of First-order logic in BNF form.

Answer:- The syntax of First-order Logic in BNF is:

<term> ::= <variable> | <constant> | <function>(<term>, <term>, ...)
<atomic_formula> ::= <predicate>(<term>, <term>, ...)
<formula> ::= <atomic_formula>
| <formula> ∧ <formula>
| <formula> ∨ <formula>
| ¬ <formula>
| ∀ <variable> . <formula>
| ∃ <variable> . <formula>

b. What is quantifier?

Answer:- A quantifier is a language element which generates quantification, and quantification specifies the quantity of specimen in the universe of discourse.
These are the symbols that permit to determine or identify the range and scope of the variable in the logical expression.

c. Define wumpus world?

Answer:- The Wumpus world is a cave with *16* rooms *(4×4)*. Each room is connected to others through walkways (no rooms are connected diagonally). The knowledge-based agent starts from *Room[1, 1]*. The cave has – some pits, a treasure and a beast named Wumpus. The Wumpus can not move but eats the one who enters its room. If the agent enters the pit, it gets stuck there. The goal of the agent is to take the treasure and come out of the cave. The agent is rewarded, when the goal conditions are met. The agent is penalized, when it falls into a pit or being eaten by the Wumpus.

d. Evaluate the given sentence “All Pompians were Romans” write a well-formed formula in predicate logic.

Answer:- ∀x:Pompians(x) ->Romans(x)

Here, Pompians(x)**->**Romans(x) = If x is a Pompians then x is Romans and x can be anyone(all)

e. What is unification?

Answer:- Unification in AI aims to develop models and systems that can handle multiple tasks across various cognitive domains. Instead of designing specialised AI systems for narrowly defined tasks (like chatbots for conversation or algorithms for image recognition), the goal is to integrate these systems into a unified framework capable of functioning cohesively.

2.

a. Explain inference rules for quantifiers?

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

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

1. **Universal Generalization:**

- Universal generalization is a valid inference rule which states that if premise P(c) is true for any arbitrary element c in the universe of discourse, then we can have a conclusion as ∀ x P(x).
- It can be represented as: $\frac{P(c)}{\forall x\, P(x)}$.

2. Universal Instantiation:

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

3. Existential Instantiation:

- Existential instantiation is also called as Existential Elimination, which is a valid inference rule in first-order logic.

- This rule states that one can infer P(c) from the formula given in the form of ∃x P(x) for a new constant symbol c.
- The restriction with this rule is that c used in the rule must be a new term for which P(c) is true.
- It can be represented as: $\frac{\exists x\, P(x)}{P(c)}$

4. Existential introduction

- An existential introduction is also known as an existential generalization, which is a valid inference rule in first-order logic.
- This rule states that if there is some element c in the universe of discourse which has a property P, then we can infer that there exists something in the universe which has the property P.
- It can be represented as: $\frac{P(c)}{\exists x P(x)}$

b. Illustrate the syntax and semantics of first order logic.

Answer:- The syntax of FOL defines how formulas are structured using terms, predicates, connectives, and quantifiers. The semantics assigns meanings to these symbols within a domain, allowing us to interpret the formulas and evaluate their truth values.

Syntax of First-Order Logic (FOL)

The syntax of First-Order Logic specifies the rules for constructing valid expressions. These expressions include terms, predicates, logical connectives, and quantifiers.

- Terms: Represent objects in the domain.
  - Constants: Specific objects, e.g., John, 3.
  - Variables: Arbitrary objects, e.g., x, y.
  - Functions: Mappings that return objects, e.g., father(x).
- **Predicates (Atomic Formulas):** Represent relations between terms, e.g., Likes(x, y).
- **Formulas**: Formed using atomic formulas, logical connectives, and quantifiers.
  - **Connectives**: ∧ (AND), ∨ (OR), ¬ (NOT), → (IF...THEN), ↔ (IF AND ONLY IF).

- **Quantifiers**: ∀ (for all), ∃ (there exists).

The structure of a formula could be something like:
<formula> ::= <atomic_formula> | <formula> ∧ <formula> | ∀<variable>.<formula> | ∃<variable>.<formula>

Semantics of First-Order Logic (FOL)

The semantics of First-Order Logic defines the meaning of the formulas in a given domain of discourse.

- **Domain of Discourse:** The set of all objects under consideration (e.g., people, numbers).
- **Interpretation**: Assigns specific meanings to constants, functions, and predicates in the domain.
  - Constants: Refer to specific objects.
  - Variables: Can refer to any object in the domain.
  - Predicates: Represent relations, mapped to sets of tuples of domain objects.
- **Quantifiers**:
  - Universal Quantifier (∀): ∀x P(x)∀x \, P(x) means the predicate P(x)P(x) holds for all objects in the domain.
  - Existential Quantifier (∃): ∃x P(x)∃x \, P(x) means there exists at least one object in the domain such that P(x)P(x) holds.

c. Explain backward chaining process?

Answer:-

Backward Chaining is a reasoning technique used in logic-based systems, especially in automated theorem proving and expert systems. It is a form of goal-driven reasoning, where the process starts with a goal or query and works backward to find supporting facts or evidence.

In First-Order Logic (FOL), backward chaining involves trying to prove a query (the goal) by recursively breaking it down into sub-goals, until you reach known facts or premises that can be directly verified.

Steps in Backward Chaining Process

1. **Start with the Goal:**
   - The process begins with a specific goal or query, typically in the form of a formula that needs to be proven. For example, you want to prove Likes(John,Mary).
2. **Look for Matching Rules:**

- Check if there are any inference rules or knowledge base facts that match the goal. These rules typically have the form of implications or conditionals (e.g., A→B).
- The goal becomes the consequent (right-hand side) of the rule, and the process tries to satisfy the antecedent (left-hand side) by recursively solving the sub-goals.

For example, if you have a rule like:

$$\text{Likes}(x,y) \rightarrow \text{Food}(y)$$

and your goal is Likes(John,Mary), then you would need to prove Food(Mary).

3. **Recursive Sub-Goal Generation:**
   - The goal Likes(John,Mary) has now been transformed into a new sub-goal Food(Mary).
   - This new sub-goal is again processed using backward chaining. You will check the knowledge base or rules to find a fact or a rule that leads to Food(Mary).
   - Continue breaking down the problem in this way until you reach facts that are already known (e.g., Food(Mary) is true because "Mary is a food").
4. Verify Facts:
   - If you eventually reach a fact that is known (i.e., it directly matches an assertion in the knowledge base), then the goal has been successfully proven.
   - If you can't find any rules or facts to support a sub-goal, then the original goal can't be proven.
5. Backtrack if Necessary:
   - If one branch of reasoning fails (i.e., you can't prove a sub-goal), you backtrack and try another rule or hypothesis.

Example of Backward Chaining Process

Let's work through an example:

Given:

- Knowledge Base:
  - Likes(John,y)→Food(y) (If John likes something, it is food)
  - Food(Mary) (Mary is food)
  - Likes(John,Mary) (John likes Mary)
- Goal: Prove Food(Mary) (Is Mary food?)

Backward Chaining Steps:

1. Start with the goal: Food(Mary).

2. Check the knowledge base. Find that the fact Food(Mary) is explicitly stated in the knowledge base, so the goal is already satisfied.
    - The fact Food(Mary) is true, and no further reasoning is required.

d. Discuss diagnostic rules and causal rules in FOL?

e. Explain Numbers, sets, and lists in FOL.

3.

a. Consider the following sentences:

- **John likes all kinds of food**
- **Applies are food**
- **Chicken is food**
- **Anything anyone eats and isn't killed by is food**
- **Bill eats peanuts and is still alive**
- **Sue eats everything Bill eats**

i. **Translate these sentences into formulas in predicate logic**
ii. **Prove that john likes peanuts using backward chaining**
iii. **Convert the formulas of a part into clause form**
iv. **Prove that john likes peanuts using resolution**

Answer:-

1. Translating the Sentences into Predicate Logic

1. **John likes all kinds of food**

$\forall x(Food(x) \rightarrow Likes(John,x))$

2. **Apples are food**

Food(Apple)

3. **Chicken is food**

Food(Chicken)

4. **Anything anyone eats and isn't killed by is food**

$\forall x \forall y((Eats(y,x) \land \neg KilledBy(x,y)) \rightarrow Food(x))$

5. **Bill eats peanuts and is still alive**

$Eats(Bill,Peanuts) \land \neg KilledBy(Peanuts,Bill)$

6. **Sue eats everything Bill eats**

$\forall x(Eats(Bill,x) \rightarrow Eats(Sue,x))$

---

2. Proving that John Likes Peanuts using Backward Chaining

Now, we will use backward chaining to prove that "John likes peanuts". We have to derive that Likes(John,Peanuts).

1. **From the formula for John likes all kinds of food, we know:**

$\forall x(Food(x) \rightarrow Likes(John,x))$

So, to prove Likes(John,Peanuts), we need to prove that Peanuts is food.

2. **From the formula for anything anyone eats and isn't killed by is food, we know:**

$\forall x \forall y((Eats(y,x) \land \neg KilledBy(x,y)) \rightarrow Food(x))$

This tells us that if someone eats peanuts and isn't killed by peanuts, then peanuts are food.

3. **From the formula for Bill eats peanuts and is still alive, we have:**

$Eats(Bill,Peanuts) \land \neg KilledBy(Peanuts,Bill)$

Bill eats peanuts and is still alive, which satisfies the condition that Bill eats peanuts and isn't killed by them. Thus, by the previous rule, we conclude:

$Food(Peanuts)$

4. **Now, using the first formula, since we have established Food(Peanuts), we can conclude:**

$Likes(John,Peanuts)$

Thus, John likes peanuts.

---

3. Converting the Formulas into Clause Form

Clause form is a way to represent logical formulas using a set of disjunctions (ORs) of literals, which are variables or negations of variables. Let's convert the given formulas into clausal form:

1. **John likes all kinds of food**

$\forall x(Food(x) \rightarrow Likes(John,x)) \equiv \forall x(\neg Food(x) \vee Likes(John,x))$

In clausal form:

$\{\neg Food(x), Likes(John,x)\}$

2. **Apples are food**

$Food(Apple)$

In clausal form:

$\{Food(Apple)\}$

3. **Chicken is food**

$Food(Chicken)$

In clausal form:

$\{Food(Chicken)\}$

4. **Anything anyone eats and isn’t killed by is food**

$\forall x \forall y((Eats(y,x) \wedge \neg KilledBy(x,y)) \rightarrow Food(x)) \equiv \forall x \forall y(\neg Eats(y,x) \vee KilledBy(x,y) \vee Food(x))$

In clausal form:

$\{\neg Eats(y,x), KilledBy(x,y), Food(x)\}$

5. **Bill eats peanuts and is still alive**

$Eats(Bill,Peanuts) \wedge \neg KilledBy(Peanuts,Bill)$

In clausal form:

$\{Eats(Bill,Peanuts)\}$

$\{\neg KilledBy(Peanuts,Bill)\}$

6. **Sue eats everything Bill eats**

$\forall x(Eats(Bill,x) \rightarrow Eats(Sue,x)) \equiv \forall x(\neg Eats(Bill,x) \vee Eats(Sue,x))$

In clausal form:

{¬Eats(Bill,x),Eats(Sue,x)}

---

4. Proving that John Likes Peanuts using Resolution

To prove that John likes peanuts using resolution, we need to combine the clauses we have derived and attempt to derive Likes(John,Peanuts).

We start with the following clauses:

1. {¬Food(x),Likes(John,x)}
2. {Food(Apple)}
3. {Food(Chicken)}
4. {¬Eats(y,x),KilledBy(x,y),Food(x)}
5. {Eats(Bill,Peanuts)}
6. {¬KilledBy(Peanuts,Bill)}
7. {¬Eats(Bill,x),Eats(Sue,x)}

Now, let's resolve:

- From the formula Eats(Bill,Peanuts) and the formula ¬KilledBy(Peanuts,Bill), we know that Peanuts is food (from the condition "anything anyone eats and isn't killed by is food").
- Using this, we can resolve the clause {¬Food(x),Likes(John,x)} with Food(Peanuts), which gives us:

Likes(John,Peanuts)

Thus, we have proven that John likes peanuts using resolution.

b. Explain resolution in predicate logic with suitable example.

c. Explain knowledge engineering process in FOL?

d. Define Ontological Engineering? Explain with the diagram the upper ontology of the world.

e. Write short notes on universal and existential quantification in FOL?

4.

a. Explain the forward chaining process and efficient forward chaining in detail with example. What is the need of incremental forward chaining?