**UNIT-2**

AI in gaming refers to responsive and adaptive video game experiences. These AI-powered interactive experiences are usually generated via non-player characters, or NPCs, that act intelligently or creatively, as if controlled by a human game-player. AI is the engine that determines an NPC’s behavior in the game world.  
  
While AI in some form has long appeared in video games, it is considered a booming new frontier in how games are both developed and played. AI games increasingly shift the control of the game experience toward the player, whose behavior helps produce the game experience.  
  
AI procedural generation, also known as procedural storytelling, in game design refers to game data being produced algorithmically rather than every element being built specifically by a developer.

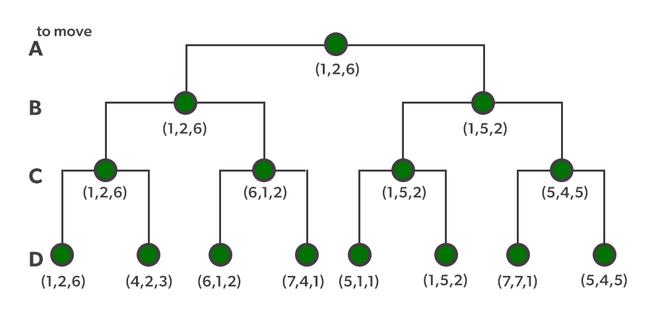
**Optimal Decision Making in Games**

Let us start with games with two players, whom we’ll refer to as MAX and MIN for obvious reasons. MAX is the first to move, and then they take turns until the game is finished. At the conclusion of the game, the victorious player receives points, while the loser receives penalties. A game can be formalized as a type of search problem that has the following elements:

* **S0:** The initial state of the game, which describes how it is set up at the start.
* **Player (s):** Defines which player in a state has the move.
* **Actions (s):** Returns a state’s set of legal moves.
* **Result (s, a):** A transition model that defines a move’s outcome.
* **Terminal-Test (s):** A terminal test that returns true if the game is over but false otherwise. Terminal states are those in which the game has come to a conclusion.
* **Utility (s, p):** A utility function (also known as a payout function or objective function ) determines the final numeric value for a game that concludes in the terminal state ***s*** for player ***p***. The result in chess is a win, a loss, or a draw, with values of +1, 0, or 1/2. Backgammon’s payoffs range from 0 to +192, but certain games have a greater range of possible outcomes. A zero-sum game is defined (confusingly) as one in which the total reward to all players is the same for each game instance. Chess is a zero-sum game because each game has a payoff of 0 + 1, 1 + 0, or 1/2 + 1/2. “Constant-sum” would have been a preferable name, 22 but zero-sum is the usual term and makes sense if each participant is charged 1.

The game tree for the game is defined by the beginning state, ACTIONS function, and RESULT function—a tree in which the nodes are game states and the edges represent movements. The figure below depicts a portion of the tic-tac-toe game tree (noughts and crosses). MAX may make nine different maneuvers from his starting position. The game alternates between MAXs setting an X and MINs placing an O until we reach leaf nodes corresponding to terminal states, such as one player having three in a row or all of the squares being filled. The utility value of the terminal state from the perspective of MAX is shown by the number on each leaf node; high values are thought to be beneficial for MAX and bad for MIN

The game tree for tic-tac-toe is relatively short, with just 9! = 362,880 terminal nodes. However, because there are over 1040 nodes in chess, the game tree is better viewed as a theoretical construct that cannot be realized in the actual world. But, no matter how big the game tree is, MAX’s goal is to find a solid move. A tree that is superimposed on the whole game tree and examines enough nodes to allow a player to identify what move to make is referred to as a search tree.

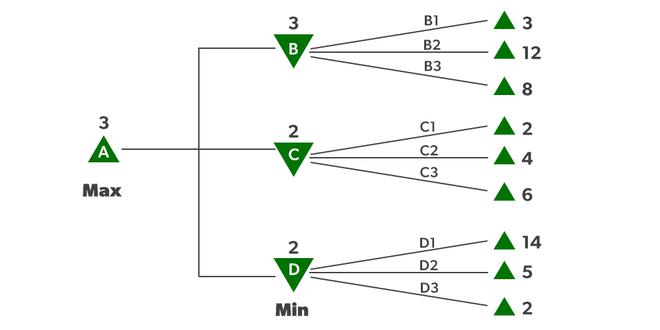


A sequence of actions leading to a goal state—a terminal state that is a win—would be the best solution in a typical search problem. MIN has something to say about it in an adversarial search. MAX must therefore devise a contingent strategy that specifies M A X’s initial state move, then MAX’s movements in the states resulting from every conceivable MIN response, then MAX’s moves in the states resulting from every possible MIN reaction to those moves, and so on. This is quite similar to the AND-OR search method, with MAX acting as OR and MIN acting as AND. When playing an infallible opponent, an optimal strategy produces results that are as least as excellent as any other plan. We’ll start by demonstrating how to find the best plan.

We’ll move to the trivial game in the ***figure below*** since even a simple game like tic-tac-toe is too complex for us to draw the full game tree on one page. MAX’s root node moves are designated by the letters a1, a2, and a3. MIN’s probable answers to a1 are b1, b2, b3, and so on. This game is over after MAX and MIN each make one move. (In game terms, this tree consists of two half-moves and is one move deep, each of which is referred to as a ply.) The terminal states in this game have utility values ranging from 2 to 14.

### Game’s Utility Function

The optimal strategy can be found from the minimax value of each node, which we express as MINIMAX, given a game tree (n). Assuming that both players play optimally from there through the finish of the game, the utility (for MAX) of being in the corresponding state is the node’s minimax value. The usefulness of a terminal state is obviously its minimax value. Furthermore, if given the option, MAX prefers to shift to a maximum value state, whereas MIN wants to move to a minimum value state. So here’s what we’ve got:



*Optimal Decision Making in Multiplayer Games*

Let’s use these definitions to analyze the game tree shown in ***the figure above***. The game’s UTILITY function provides utility values to the terminal nodes on the bottom level. Because the first MIN node, B, has three successor states with values of 3, 12, and 8, its minimax value is 3. Minimax value 2 is also used by the other two MIN nodes. The root node is a MAX node, with minimax values of 3, 2, and 2, resulting in a minimax value of 3. We can also find the root of the minimax decision: action a1 is the best option for MAX since it leads to the highest minimax value.

This concept of optimal MAX play requires that MIN plays optimally as well—it maximizes MAX’s worst-case outcome. What happens if MIN isn’t performing at its best? Then it’s a simple matter of demonstrating that MAX can perform even better. Other strategies may outperform the minimax method against suboptimal opponents, but they will always outperform optimal opponents.

# **Alpha-Beta Pruning**

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

## **Condition for Alpha-beta pruning:**

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

α>=β

## **Key points about alpha-beta pruning:**

* The Max player will only update the value of alpha.
* The Min player will only update the value of beta.
* While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
* We will only pass the alpha, beta values to the child nodes.

## **Pseudo-code for Alpha-beta Pruning:**

function minimax(node, depth, alpha, beta, maximizingPlayer) is

**if** depth ==0 or node is a terminal node then

**return** **static** evaluation of node

**if** MaximizingPlayer then      // for Maximizer Player

   maxEva= -infinity

**for** each child of node **do**

   eva= minimax(child, depth-1, alpha, beta, False)

  maxEva= max(maxEva, eva)

  alpha= max(alpha, maxEva)

**if** beta<=alpha

**break**

**return** maxEva

**else**                         // for Minimizer player

   minEva= +infinity

**for** each child of node **do**

   eva= minimax(child, depth-1, alpha, beta, **true**)

   minEva= min(minEva, eva)

   beta= min(beta, eva)

**if** beta<=alpha

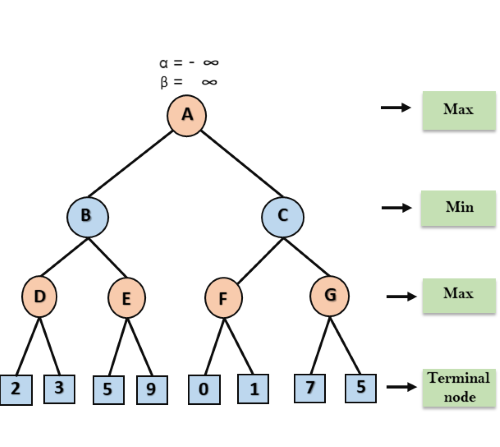
**break**

**return** minEva

## **Working of Alpha-Beta Pruning:**

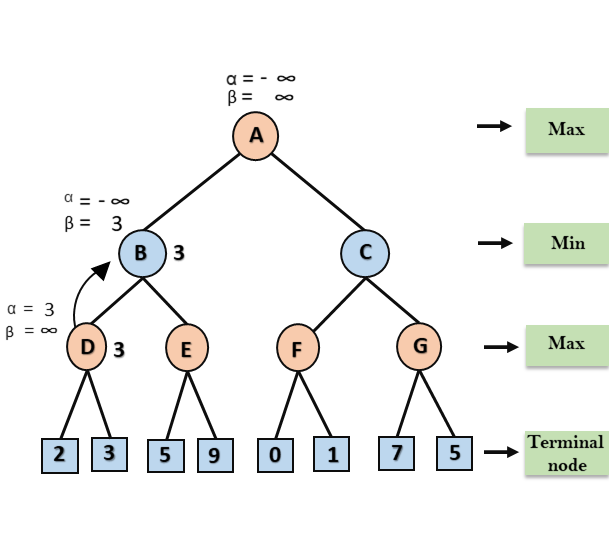
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

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



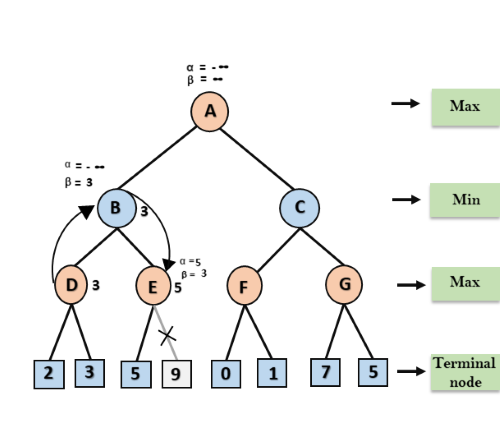
**Step 2:** At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

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



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

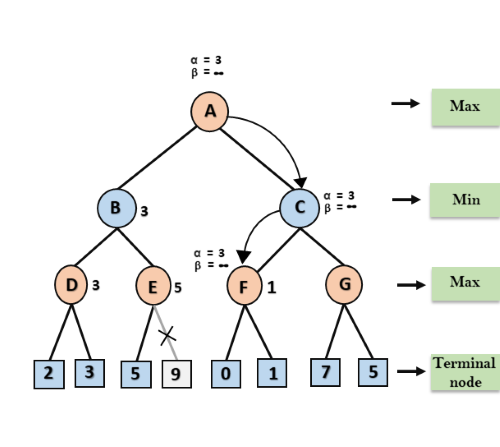
**Step 4:** At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E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



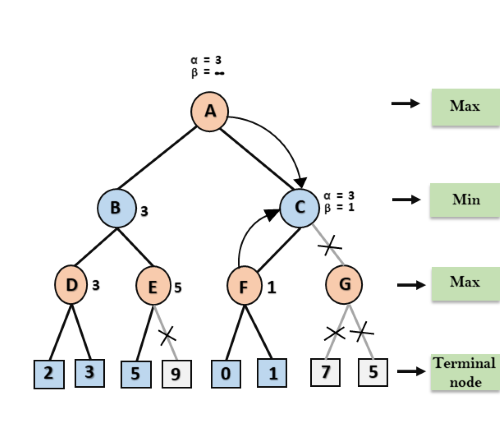
**Step 5:** At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

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

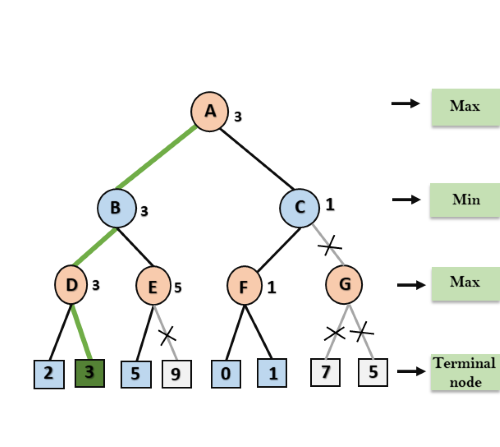
**Step 6:** At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



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



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



## **Move Ordering in Alpha-Beta pruning:**

The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning.

It can be of two types:

* **Worst ordering:** In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is O(bm).
* **Ideal ordering:** The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is O(bm/2).

## **Rules to find good ordering:**

Following are some rules to find good ordering in alpha-beta pruning:

* Occur the best move from the shallowest node.
* Order the nodes in the tree such that the best nodes are checked first.
* Use domain knowledge while finding the best move. Ex: for Chess, try order: captures first, then threats, then forward moves, backward moves.
* We can bookkeep the states, as there is a possibility that states may repeat.

# **Constraint Satisfaction Problems in Artificial Intelligence**

We have encountered a wide variety of methods, including adversarial search and instant search, to address various issues. Every method for issue has a single purpose in mind: to locate a remedy that will enable that achievement of the objective. However there were no restrictions just on bots' capability to resolve issues as well as arrive at responses in adversarial search and local search, respectively.

These section examines the constraint optimization methodology, another form or real concern method. By its name, constraints fulfilment implies that such an issue must be solved while adhering to a set of restrictions or guidelines.

Whenever a problem is actually variables comply with stringent conditions of principles, it is said to have been addressed using the solving multi - objective method. Wow what a method results in a study sought to achieve of the intricacy and organization of both the issue.

Three factors affect restriction compliance, particularly regarding:

* It refers to a group of parameters, or X.
* D: The variables are contained within a collection several domain. Every variables has a distinct scope.
* C: It is a set of restrictions that the collection of parameters must abide by.

In constraint satisfaction, domains are the areas wherein parameters were located after the restrictions that are particular to the task. Those three components make up a constraint satisfaction technique in its entirety. The pair "scope, rel" makes up the number of something like the requirement. The scope is a tuple of variables that contribute to the restriction, as well as rel is indeed a relationship that contains a list of possible solutions for the parameters should assume in order to meet the restrictions of something like the issue.

Issues with Contains A certain amount Solved

For a constraint satisfaction problem (CSP), the following conditions must be met:

* States area
* fundamental idea while behind remedy.

The definition of a state in phase space involves giving values to any or all of the parameters, like as

X1 = v1, X2 = v2, etc.

There are 3 methods to economically beneficial to something like a parameter:

1. Consistent or Legal Assignment: A task is referred to as consistent or legal if it complies with all laws and regulations.
2. Complete Assignment: An assignment in which each variable has a number associated to it and that the CSP solution is continuous. One such task is referred to as a completed task.
3. A partial assignment is one that just gives some of the variables values. Projects of this nature are referred to as incomplete assignment.

## **Domain Categories within CSP**

The parameters utilize one of the two types of domains listed below:

* Discrete Domain: This limitless area allows for the existence of a single state with numerous variables. For instance, every parameter may receive a endless number of beginning states.
* It is a finite domain with continous phases that really can describe just one area for just one particular variable. Another name for it is constant area.

## **Types of Constraints in CSP**

Basically, there are three different categories of limitations in regard towards the parameters:

* Unary restrictions are the easiest kind of restrictions because they only limit the value of one variable.
* Binary resource limits: These restrictions connect two parameters. A value between x1 and x3 can be found in a variable named x2.
* Global Resource limits: This kind of restriction includes a unrestricted amount of variables.

The main kinds of restrictions are resolved using certain kinds of resolution methodologies:

* In linear programming, when every parameter carrying an integer value only occurs in linear equation, linear constraints are frequently utilised.
* Non-linear Constraints: With non-linear programming, when each variable (an integer value) exists in a non-linear form, several types of restrictions were utilised.
* hink of a Sudoku puzzle where some of the squares have initial fills of certain integers.
* You must complete the empty squares with numbers between 1 and 9, making sure that no rows, columns, or blocks contains a recurring integer of any kind. This solving multi - objective issue is pretty elementary. A problem must be solved while taking certain limitations into consideration.
* The integer range (1-9) that really can occupy the other spaces is referred to as a domain, while the empty spaces themselves were referred as variables. The values of the variables are drawn first from realm. Constraints are the rules that determine how a variable will select the scope.

**Constraint Propagation**

Constraint propagation is the process of communicating the domain reduction of a decision variable to all of the constraints that are stated over this variable. This process can result in more domain reductions. These domain reductions, in turn, are communicated to the appropriate constraints.

### [Backtracking Search (CSPs)](http://www.cs.toronto.edu/~hojjat/384w09/Lectures/Lecture-04-Backtracking-Search.pdf)

Backtracking is a simple and widely used search-based algorithm for CSPs. It works by choosing a variable, assigning a value to it, and then recursively trying to solve the rest of the problem.

# **Backtracking**

In this topic, we will learn about the backtracking, which is a very important skill set to solve recursive solutions. Recursive functions are those that calls itself more than once. Consider an example of Palindrome:

Initially, the function isPalindrome(S, 0, 8) is called once with the parameters isPalindrome(S, 1, 7). The recursive call isPalindrome(S, 1, 7) is called once with the parameters isPalindrome(S, 2, 6).

Backtracking is one of the techniques that can be used to solve the problem. We can write the algorithm using this strategy. It uses the Brute force search to solve the problem, and the brute force search says that for the given problem, we try to make all the possible solutions and pick out the best solution from all the desired solutions. This rule is also followed in dynamic programming, but dynamic programming is used for solving optimization problems. In contrast, backtracking is not used in solving optimization problems. Backtracking is used when we have multiple solutions, and we require all those solutions.

Backtracking name itself suggests that we are going back and coming forward; if it satisfies the condition, then return success, else we go back again. It is used to solve a problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criteria.

### When to use a Backtracking algorithm?

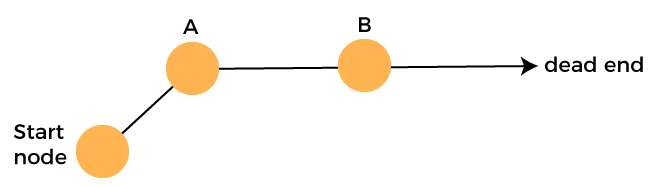
When we have multiple choices, then we make the decisions from the available choices. In the following cases, we need to use the backtracking algorithm:

* A piece of sufficient information is not available to make the best choice, so we use the backtracking strategy to try out all the possible solutions.
* Each decision leads to a new set of choices. Then again, we backtrack to make new decisions. In this case, we need to use the backtracking strategy.

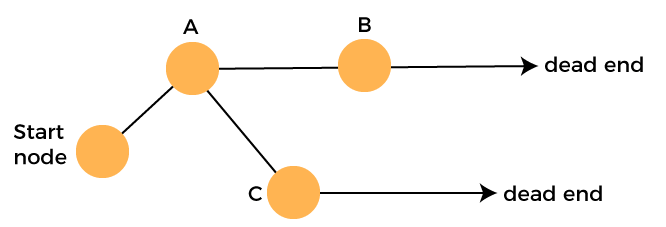
How does Backtracking work?

Backtracking is a systematic method of trying out various sequences of decisions until you find out that works. Let's understand through an example.

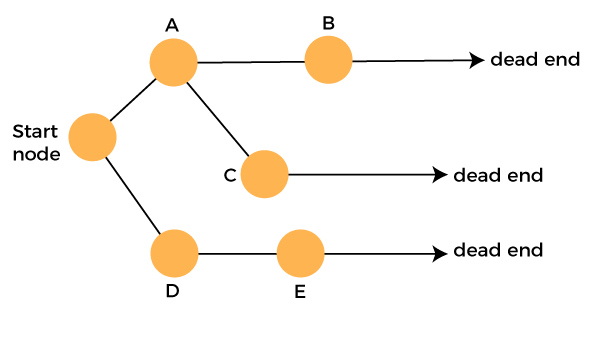
We start with a start node. First, we move to node A. Since it is not a feasible solution so we move to the next node, i.e., B. B is also not a feasible solution, and it is a dead-end so we backtrack from node B to node A.



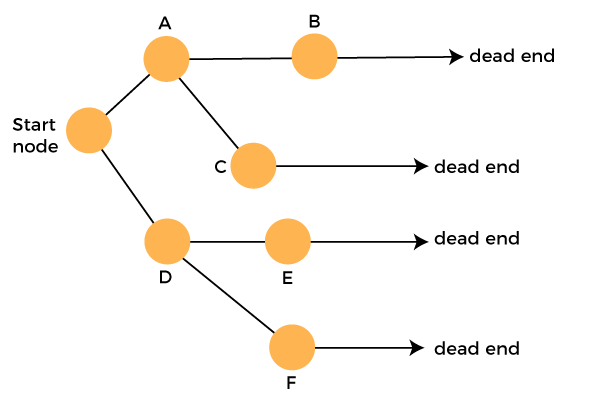
Suppose another path exists from node A to node C. So, we move from node A to node C. It is also a dead-end, so again backtrack from node C to node A. We move from node A to the starting node.



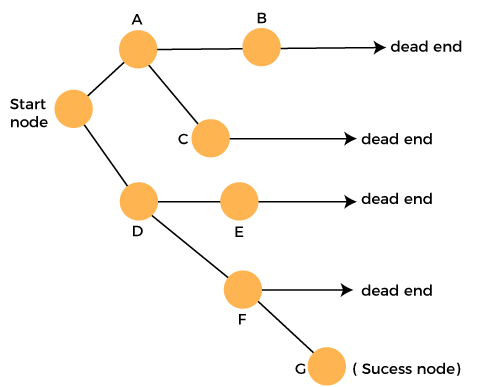
Now we will check any other path exists from the starting node. So, we move from start node to the node D. Since it is not a feasible solution so we move from node D to node E. The node E is also not a feasible solution. It is a dead end so we backtrack from node E to node D.



Suppose another path exists from node D to node F. So, we move from node D to node F. Since it is not a feasible solution and it's a dead-end, we check for another path from node F.



Suppose there is another path exists from the node F to node G so move from node F to node G. The node G is a success node.



**The terms related to the backtracking are:**

* **Live node:** The nodes that can be further generated are known as live nodes.
* **E node:** The nodes whose children are being generated and become a success node.
* **Success node:** The node is said to be a success node if it provides a feasible solution.
* **Dead node:** The node which cannot be further generated and also does not provide a feasible solution is known as a dead node.

Many problems can be solved by backtracking strategy, and that problems satisfy complex set of constraints, and these constraints are of two types:

* **Implicit constraint:** It is a rule in which how each element in a tuple is related.
* **Explicit constraint:** The rules that restrict each element to be chosen from the given set.

Applications of Backtracking

* N-queen problem
* Sum of subset problem
* Graph coloring
* Hamiliton cycle

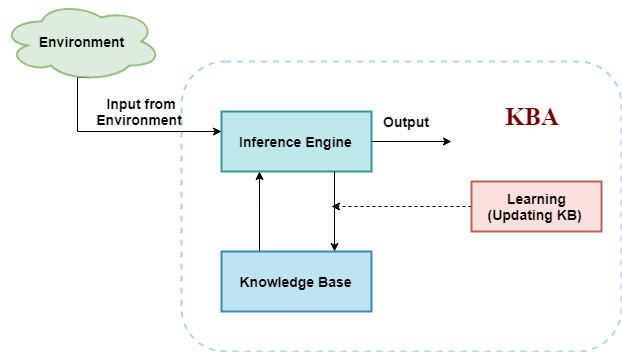
# **Knowledge-Based Agent in Artificial intelligence**

* An intelligent agent needs **knowledge** about the real world for taking decisions and **reasoning** to act efficiently.
* 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.

**Knowledge base:** Knowledge-base is a central component of a knowledge-based agent, it is also known as KB. It is a collection of sentences (here 'sentence' is a technical term and it is not identical to sentence in English). These sentences are expressed in a language which is called a knowledge representation language. The Knowledge-base of KBA stores fact about the world.

## **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.

## **Inference system**

Inference means deriving new sentences from old. Inference system allows us to add a new sentence to the knowledge base. A sentence is a proposition about the world. Inference system applies logical rules to the KB to deduce new information.

Inference system generates new facts so that an agent can update the KB. An inference system works mainly in two rules which are given as:

* **Forward chaining**
* **Backward chaining**

## **Operations Performed by KBA**

**Following are three operations which are performed by KBA in order to show the intelligent behavior:**

1. **TELL:** This operation tells the knowledge base what it perceives from the environment.
2. **ASK:** This operation asks the knowledge base what action it should perform.
3. **Perform:** It performs the selected action.

## **A generic knowledge-based agent:**

Following is the structure outline of a generic knowledge-based agents program:

1. function KB-AGENT(percept):
2. persistent: KB, a knowledge base
3. t, a counter, initially 0, indicating time
4. TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
5. Action = ASK(KB, MAKE-ACTION-QUERY(t))
6. TELL(KB, MAKE-ACTION-SENTENCE(action, t))
7. t = t + 1
8. **return** action

The knowledge-based agent takes percept as input and returns an action as output. The agent maintains the knowledge base, KB, and it initially has some background knowledge of the real world. It also has a counter to indicate the time for the whole process, and this counter is initialized with zero.

Each time when the function is called, it performs its three operations:

* Firstly it TELLs the KB what it perceives.
* Secondly, it asks KB what action it should take
* Third agent program TELLS the KB that which action was chosen.

The MAKE-PERCEPT-SENTENCE generates a sentence as setting that the agent perceived the given percept at the given time.

The MAKE-ACTION-QUERY generates a sentence to ask which action should be done at the current time.

MAKE-ACTION-SENTENCE generates a sentence which asserts that the chosen action was executed.

## **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.

However, in the real world, a successful agent can be built by combining both declarative and procedural approaches, and declarative knowledge can often be compiled into more efficient procedural code.

# **Propositional logic in Artificial intelligence**

Propositional logic (PL) is the simplest form of logic where all the statements are made by propositions. A proposition is a declarative statement which is either true or false. It is a technique of knowledge representation in logical and mathematical form.

### Example:

1. a) It is Sunday.
2. b) The Sun rises from West (False proposition)
3. c) 3+3= 7(False proposition)
4. d) 5 is a prime number.

**Following are some basic facts about propositional logic:**

* Propositional logic is also called Boolean logic as it works on 0 and 1.
* In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for a representing a proposition, such A, B, C, P, Q, R, etc.
* Propositions can be either true or false, but it cannot be both.
* Propositional logic consists of an object, relations or function, and **logical connectives**.
* These connectives are also called logical operators.
* The propositions and connectives are the basic elements of the propositional logic.
* Connectives can be said as a logical operator which connects two sentences.
* A proposition formula which is always true is called **tautology**, and it is also called a valid sentence.
* A proposition formula which is always false is called **Contradiction**.
* A proposition formula which has both true and false values is called
* Statements which are questions, commands, or opinions are not propositions such as "**Where is Rohini**", "**How are you**", "**What is your name**", are not propositions.

Syntax of propositional logic:

The syntax of propositional logic defines the allowable sentences for the knowledge representation. There are two types of Propositions:

1. **Atomic Propositions**
2. **Compound propositions**

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

**Example:**

1. a) 2+2 is 4, it is an atomic proposition as it is a **true** fact.
2. b) "The Sun is cold" is also a proposition as it is a **false** fact.

* **Compound proposition:** Compound propositions are constructed by combining simpler or atomic propositions, using parenthesis and logical connectives.

**Example:**

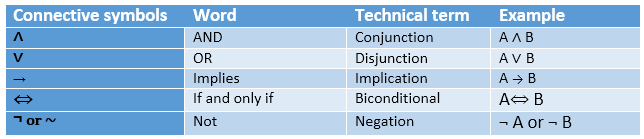
1. a) "It is raining today, and street is wet."
2. b) "Ankit is a doctor, and his clinic is in Mumbai."

## **Logical Connectives:**

Logical connectives are used to connect two simpler propositions or representing a sentence logically. We can create compound propositions with the help of logical connectives. There are mainly five connectives, which are given as follows:

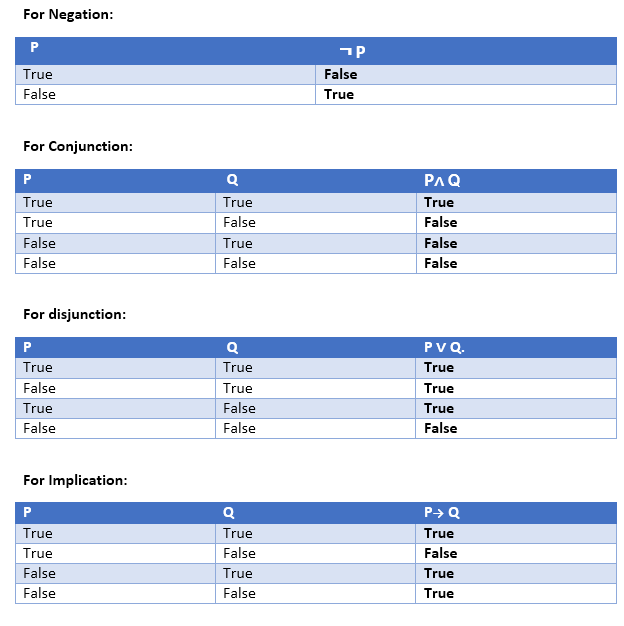
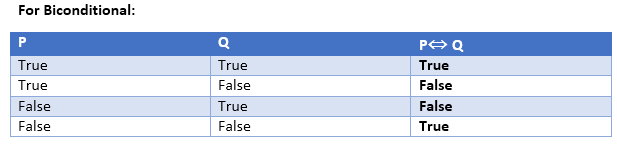
1. **Negation:** A sentence such as ¬ P is called negation of P. A literal can be either Positive literal or negative literal.
2. **Conjunction:** A sentence which has **∧**connective such as, **P ∧ Q** is called a conjunction.  
   **Example:** Rohan is intelligent and hardworking. It can be written as,  
   **P= Rohan is intelligent**,  
   **Q= Rohan is hardworking. → P∧ Q**.
3. **Disjunction:** A sentence which has ∨ connective, such as **P ∨ Q**. is called disjunction, where P and Q are the propositions.  
   **Example: "Ritika is a doctor or Engineer"**,  
   Here P= Ritika is Doctor. Q= Ritika is Doctor, so we can write it as **P ∨ Q**.
4. **Implication:** A sentence such as P → Q, is called an implication. Implications are also known as if-then rules. It can be represented as  
               **If** it is raining, then the street is wet.  
           Let P= It is raining, and Q= Street is wet, so it is represented as P → Q
5. **Biconditional:** A sentence such as **P⇔ Q is a Biconditional sentence, example If I am breathing, then I am alive**  
               P= I am breathing, Q= I am alive, it can be represented as P ⇔ Q.

### Following is the summarized table for Propositional Logic Connectives:



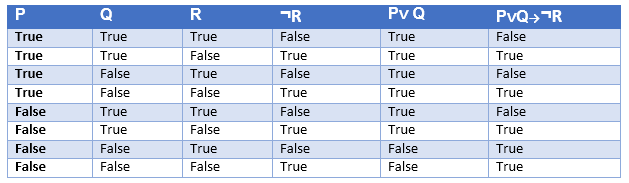
## **Truth Table:**

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:

### Truth table with three propositions:

We can build a proposition composing three propositions P, Q, and R. This truth table is made-up of 8n Tuples as we have taken three proposition symbols.



### Precedence of connectives:

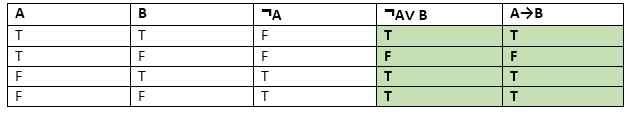
Just like arithmetic operators, there is a precedence order for propositional connectors or logical operators. This order should be followed while evaluating a propositional problem. Following is the list of the precedence order for operators:

|  |  |
| --- | --- |
| **Precedence** | **Operators** |
| First Precedence | Parenthesis |
| Second Precedence | Negation |
| Third Precedence | Conjunction(AND) |
| Fourth Precedence | Disjunction(OR) |
| Fifth Precedence | Implication |
| Six Precedence | Biconditional |

### Logical equivalence:

Logical equivalence is one of the features of propositional logic. Two propositions are said to be logically equivalent if and only if the columns in the truth table are identical to each other.

Let's take two propositions A and B, so for logical equivalence, we can write it as A⇔B. In below truth table we can see that column for ¬A∨ B and A→B, are identical hence A is Equivalent to B



**Properties of Operators:**

* **Commutativity:**
  + P∧ Q= Q ∧ P, or
  + P ∨ Q = Q ∨ P.
* **Associativity:**
  + (P ∧ Q) ∧ R= P ∧ (Q ∧ R),
  + (P ∨ Q) ∨ R= P ∨ (Q ∨ R)
* **Identity element:**
  + P ∧ True = P,
  + P ∨ True= True.
* **Distributive:**
  + P∧ (Q ∨ R) = (P ∧ Q) ∨ (P ∧ R).
  + P ∨ (Q ∧ R) = (P ∨ Q) ∧ (P ∨ R).
* **DE Morgan's Law:**
  + ¬ (P ∧ Q) = (¬P) ∨ (¬Q)
  + ¬ (P ∨ Q) = (¬ P) ∧ (¬Q).
* **Double-negation elimination:**
  + ¬ (¬P) = P.

Limitations of Propositional logic:

* We cannot represent relations like ALL, some, or none with propositional logic. Example:
  1. **All the girls are intelligent.**
  2. **Some apples are sweet.**
* Propositional logic has limited expressive power.
* In propositional logic, we cannot describe statements in terms of their properties or logical relationships.

# **Rules of Inference in Artificial intelligence**

## **Inference:**

In artificial intelligence, we need intelligent computers which can create new logic from old logic or by evidence, **so generating the conclusions from evidence and facts is termed as Inference**.

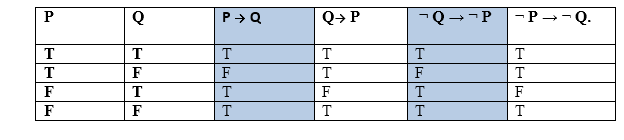
## **Inference rules:**

Inference rules are the templates for generating valid arguments. Inference rules are applied to derive proofs in artificial intelligence, and the proof is a sequence of the conclusion that leads to the desired goal.

In inference rules, the implication among all the connectives plays an important role. Following are some terminologies related to inference rules:

* **Implication:** It is one of the logical connectives which can be represented as P → Q. It is a Boolean expression.
* **Converse:** The converse of implication, which means the right-hand side proposition goes to the left-hand side and vice-versa. It can be written as Q → P.
* **Contrapositive:** The negation of converse is termed as contrapositive, and it can be represented as ¬ Q → ¬ P.
* **Inverse:** The negation of implication is called inverse. It can be represented as ¬ P → ¬ Q.

From the above term some of the compound statements are equivalent to each other, which we can prove using truth table:



Hence from the above truth table, we can prove that P → Q is equivalent to ¬ Q → ¬ P, and Q→ P is equivalent to ¬ P → ¬ Q.

## **Types of Inference rules:**

### 1. Modus Ponens:

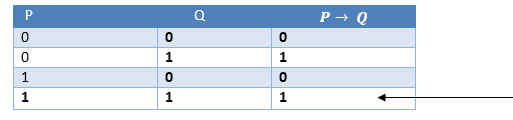
The Modus Ponens rule is one of the most important rules of inference, and it states that if P and P → Q is true, then we can infer that Q will be true. It can be represented as:

Rules of Inference in Artificial intelligence

**Example:**

Statement-1: "If I am sleepy then I go to bed" ==> P→ Q  
Statement-2: "I am sleepy" ==> P  
Conclusion: "I go to bed." ==> Q.  
Hence, we can say that, if P→ Q is true and P is true then Q will be true.

**Proof by Truth table:**



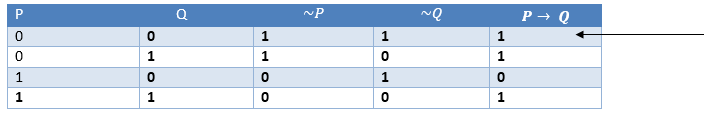
### 2. Modus Tollens:

The Modus Tollens rule state that if P→ Q is true and **¬ Q is true, then ¬ P** will also true. It can be represented as:

Rules of Inference in Artificial intelligence

**Statement-1:** "If I am sleepy then I go to bed" ==> P→ Q  
**Statement-2:** "I do not go to the bed."==> ~Q  
**Statement-3:** Which infers that "**I am not sleepy**" => ~P

**Proof by Truth table:**



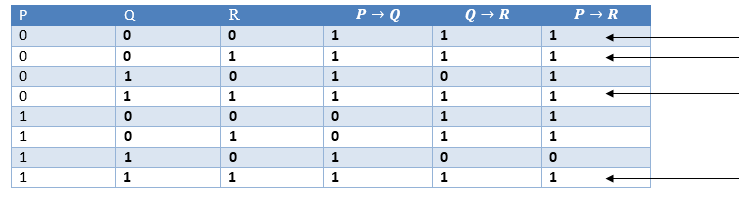
### 3. Hypothetical Syllogism:

The Hypothetical Syllogism rule state that if P→R is true whenever P→Q is true, and Q→R is true. It can be represented as the following notation:

**Example:**

**Statement-1:** If you have my home key then you can unlock my home. **P→Q**  
**Statement-2:** If you can unlock my home then you can take my money. **Q→R**  
**Conclusion:** If you have my home key then you can take my money. **P→R**

**Proof by truth table:**



### 4. Disjunctive Syllogism:

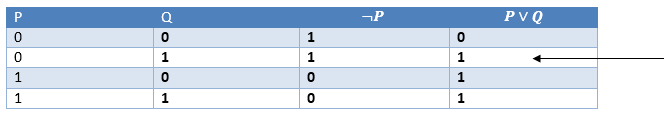
The Disjunctive syllogism rule state that if P∨Q is true, and ¬P is true, then Q will be true. It can be represented as:

Rules of Inference in Artificial intelligence

**Example:**

**Statement-1:** Today is Sunday or Monday. ==>P∨Q  
**Statement-2:** Today is not Sunday. ==> ¬P  
**Conclusion:** Today is Monday. ==> Q

**Proof by truth-table:**



### 5. Addition:

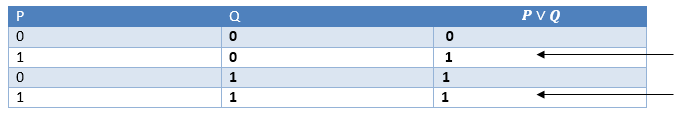
The Addition rule is one the common inference rule, and it states that If P is true, then P∨Q will be true.

Rules of Inference in Artificial intelligence

**Example:**

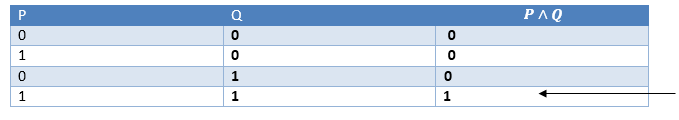
**Statement:** I have a vanilla ice-cream. ==> P  
**Statement-2:** I have Chocolate ice-cream.  
**Conclusion:** I have vanilla or chocolate ice-cream. ==> (P∨Q)

**Proof by Truth-Table:**



### 6. Simplification:

The simplification rule state that if **P∧ Q** is true, then **Q or P** will also be true. It can be represented as:

Rules of Inference in Artificial intelligence

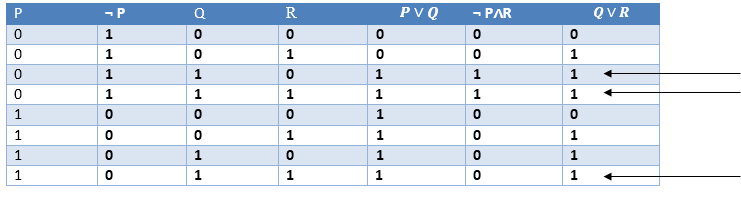
**Proof by Truth-Table:**

### 7. Resolution:

The Resolution rule state that if P∨Q and ¬ P∧R is true, then Q∨R will also be true. **It can be represented as**

Rules of Inference in Artificial intelligence

**Proof by Truth-Table:**



**Proof by resolution:**

## **Resolution**

Resolution is a theorem proving technique that proceeds by building refutation proofs, i.e., proofs by contradictions. It was invented by a Mathematician John Alan Robinson in the year 1965.

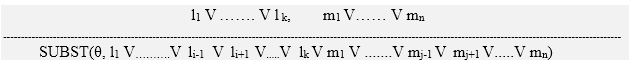
Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements. Unification is a key concept in proofs by resolutions. Resolution is a single inference rule which can efficiently operate on the **conjunctive normal form or clausal form**.

**Clause**: Disjunction of literals (an atomic sentence) is called a **clause**. It is also known as a unit clause.

**Conjunctive Normal Form**: A sentence represented as a conjunction of clauses is said to be **conjunctive normal form** or **CNF**.

## **The resolution inference rule:**

The resolution rule for first-order logic is simply a lifted version of the propositional rule. Resolution can resolve two clauses if they contain complementary literals, which are assumed to be standardized apart so that they share no variables.



Where **li** and **mj** are complementary literals.

This rule is also called the **binary resolution rule** because it only resolves exactly two literals.

### Example:

We can resolve two clauses which are given below:

**[Animal (g(x) V Loves (f(x), x)]       and       [￢ Loves(a, b) V ￢Kills(a, b)]**

Where two complimentary literals are: **Loves (f(x), x) and ￢ Loves (a, b)**

These literals can be unified with unifier **θ= [a/f(x), and b/x]**, and it will generate a resolvent clause:

**[Animal (g(x) V ￢ Kills(f(x), x)].**

## **Steps for Resolution:**

1. Conversion of facts into first-order logic.
2. Convert FOL statements into CNF
3. Negate the statement which needs to prove (proof by contradiction)
4. Draw resolution graph (unification).

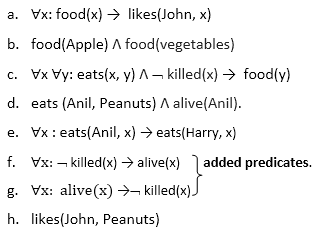
To better understand all the above steps, we will take an example in which we will apply resolution.

### Example:

1. **John likes all kind of food.**
2. **Apple and vegetable are food**
3. **Anything anyone eats and not killed is food.**
4. **Anil eats peanuts and still alive**
5. **Harry eats everything that Anil eats.**  
   **Prove by resolution that:**
6. **John likes peanuts.**

**Step-1: Conversion of Facts into FOL**

In the first step we will convert all the given statements into its first order logic.



**Step-2: Conversion of FOL into CNF**

In First order logic resolution, it is required to convert the FOL into CNF as CNF form makes easier for resolution proofs.

* **Eliminate all implication (→) and rewrite**
  1. ∀x ¬ food(x) V likes(John, x)
  2. food(Apple) Λ food(vegetables)
  3. ∀x ∀y ¬ [eats(x, y) Λ ¬ killed(x)] V food(y)
  4. eats (Anil, Peanuts) Λ alive(Anil)
  5. ∀x ¬ eats(Anil, x) V eats(Harry, x)
  6. ∀x¬ [¬ killed(x) ] V alive(x)
  7. ∀x ¬ alive(x) V ¬ killed(x)
  8. likes(John, Peanuts).
* **Move negation (¬)inwards and rewrite**
  1. ∀x ¬ food(x) V likes(John, x)
  2. food(Apple) Λ food(vegetables)
  3. ∀x ∀y ¬ eats(x, y) V killed(x) V food(y)
  4. eats (Anil, Peanuts) Λ alive(Anil)
  5. ∀x ¬ eats(Anil, x) V eats(Harry, x)
  6. ∀x ¬killed(x) ] V alive(x)
  7. ∀x ¬ alive(x) V ¬ killed(x)
  8. likes(John, Peanuts).
* **Rename variables or standardize variables**
  1. ∀x ¬ food(x) V likes(John, x)
  2. food(Apple) Λ food(vegetables)
  3. ∀y ∀z ¬ eats(y, z) V killed(y) V food(z)
  4. eats (Anil, Peanuts) Λ alive(Anil)
  5. ∀w¬ eats(Anil, w) V eats(Harry, w)
  6. ∀g ¬killed(g) ] V alive(g)
  7. ∀k ¬ alive(k) V ¬ killed(k)
  8. likes(John, Peanuts).
* **Eliminate existential instantiation quantifier by elimination.**  
  In this step, we will eliminate existential quantifier ∃, and this process is known as **Skolemization**. But in this example problem since there is no existential quantifier so all the statements will remain same in this step.
* **Drop Universal quantifiers.**  
  In this step we will drop all universal quantifier since all the statements are not implicitly quantified so we don't need it.
  1. ¬ food(x) V likes(John, x)
  2. food(Apple)
  3. food(vegetables)
  4. ¬ eats(y, z) V killed(y) V food(z)
  5. eats (Anil, Peanuts)
  6. alive(Anil)
  7. ¬ eats(Anil, w) V eats(Harry, w)
  8. killed(g) V alive(g)
  9. ¬ alive(k) V ¬ killed(k)
  10. likes(John, Peanuts).

#### **Note: Statements "food(Apple) Λ food(vegetables)" and "eats (Anil, Peanuts) Λ alive(Anil)" can be written in two separate statements.**

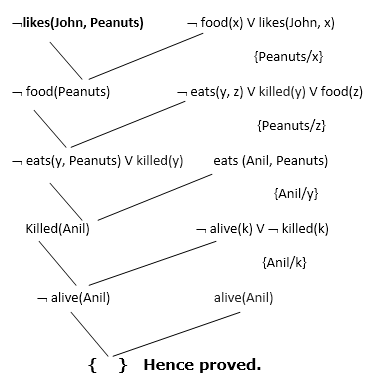
* **Distribute conjunction ∧ over disjunction ¬.**  
  This step will not make any change in this problem.

**Step-3: Negate the statement to be proved**

In this statement, we will apply negation to the conclusion statements, which will be written as ¬likes(John, Peanuts)

**Step-4: Draw Resolution graph:**

Now in this step, we will solve the problem by resolution tree using substitution. For the above problem, it will be given as follows:



Hence the negation of the conclusion has been proved as a complete contradiction with the given set of statements.

# **Forward Chaining and backward chaining in AI**

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

## **Inference engine:**

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

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

**Horn Clause and Definite clause:**

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

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

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

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

It is equivalent to p ∧ q → k.

## **A. Forward Chaining**

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

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

**Properties of Forward-Chaining:**

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

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

### Example:

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

Prove that **"Robert is criminal."**

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

### Facts Conversion into FOL:

* It is a crime for an American to sell weapons to hostile nations. (Let's say p, q, and r are variables)  
  **American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p)       ...(1)**
* Country A has some missiles. **?p Owns(A, p) ∧ Missile(p)**. It can be written in two definite clauses by using Existential Instantiation, introducing new Constant T1.  
  **Owns(A, T1)             ......(2)**  
  **Missile(T1)             .......(3)**
* All of the missiles were sold to country A by Robert.  
  **?p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A)       ......(4)**
* Missiles are weapons.  
  **Missile(p) → Weapons (p)             .......(5)**
* Enemy of America is known as hostile.  
  **Enemy(p, America) →Hostile(p)             ........(6)**
* Country A is an enemy of America.  
  **Enemy (A, America)             .........(7)**
* Robert is American  
  **American(Robert).             ..........(8)**

## **Forward chaining proof:**

**Step-1:**

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

Forward Chaining and backward chaining in AI

**Step-2:**

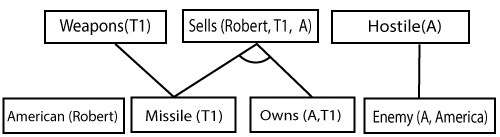
At the second step, we will see those facts which infer from available facts and with satisfied premises.

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

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

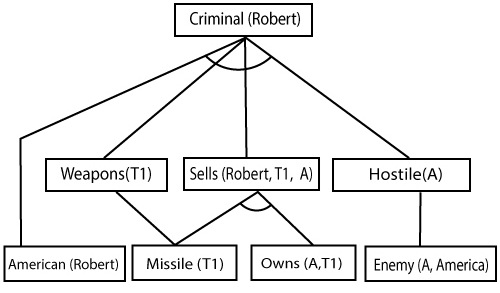
Rule-(4) satisfy with the substitution {p/T1}, **so Sells (Robert, T1, A)** is added, which infers from the conjunction of Rule (2) and (3).

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



**Step-3:**

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



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

## **B. Backward Chaining:**

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

**Properties of backward chaining:**

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

### Example:

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

* **American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) ...(1)**  
  **Owns(A, T1)                 ........(2)**
* **Missile(T1)**
* **?p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A)           ......(4)**
* **Missile(p) → Weapons (p)                 .......(5)**
* **Enemy(p, America) →Hostile(p)                 ........(6)**
* **Enemy (A, America)                 .........(7)**
* **American(Robert).                 ..........(8)**

## **Backward-Chaining proof:**

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

**Step-1:**

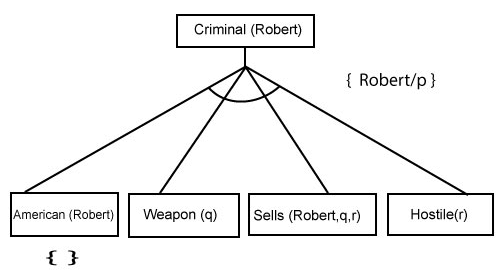
At the first step, we will take the goal fact. And from the goal fact, we will infer other facts, and at last, we will prove those facts true. So our goal fact is "Robert is Criminal," so following is the predicate of it.

Forward Chaining and backward chaining in AI

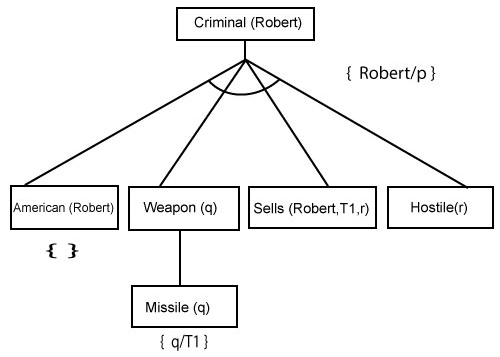
**Step-2:**

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

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



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



**Step-4:**

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



**Step-5:**

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

