Part – A

1. **What is Game play in Artificial Intelligence?**

General game playing (GGP) is the design of artificial intelligence programs to be able to play more than one game successfully. For instance, a chess-playing computer program cannot play checkers. General game playing is considered as a necessary milestone on the way to artificial general intelligence. Gameplay is the specific way in which players interact with a game, and in particular with video games. Gameplay is the pattern defined through the game rules, connection between player and the game, challenges and overcoming them, plot and player's connection with it

1. **What are the characteristics of Game theory?**

Game theory, branch of applied mathematics that provides tools for analyzing situations in which parties, called p layers, make decisions that are interdependent. This interdependence causes each player to consider the other player's possible decisions, or strategies, in formulating strategy

1. **How are optimal decision made?**

To make an optimal decision, economists ask: ―What are the extra (marginal) costs and what are the extra (marginal) benefits associated with the decision? If the extra benefits are bigger than the extra costs, you shall go ahead with the decision, namely the decision is good.

1. **What is an Alpha-Beta Pruning?**

Alpha Beta Pruning is a method that optimizes the minimax algorithm. The number of states to be visited by the minimax algorithm are exponential, which shoots up the time complexity. Some of the branches of the decision tree are useless, and the same result ca n be achieved if they were never visited.

1. **What are the four steps of MCTS?**

1 — MCTS is a simple algorithm to implement.

2 — Monte Carlo Tree Search is a heuristic algorithm. MCTS can operate effectively without any knowledge i n the particular domain, apart from the rules and end conditions, and can find its own moves and learn from them by playing random playouts.

1. **How are constraints propagated in forward checking?**

Forward checking detects the inconsistency earlier than simple backtracking and thus it allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking. This reduces the search tree and (hopefully) the overall amount of work done.

1. **What is partial observability in AI?**

Partial observability means that an agent does not know the state of the world or that the agents act simultaneously. In a partially observable system the observer may utilize a memory system in order to add information to the observer's understanding of the system. An example of a partially observable system would be a card game in which some of the cards are discarded into a pile face down

1. **What is CSP algorithm**?

In general, a CSP is a problem composed of a finite set of variables, each of which has a finite domain of values, and a set of constraints. The task is to find an assignment of a value for each variable such that the assignments satisfy all the constraints. In some problems, the goal is to find all such assignments.

1. **What is local search for CSP?**

A local search problem consists of a: CSP: a set of variables, domains for these variables, and constraints on their joint values. A node in the search space will be a complete assignment to all of the variables. Local search is an incomplete method for finding a solution to a problem. It is based on iteratively improving an assignment of the variables until all constraints are satisfied. In particular, local search algorithms typically modify the value of a variable in an assignment at each step.

1. **How Knowledge is represented?**

Knowledge Representation in AI describes the representation of knowledge. Basically, it is a study of how the beliefs, intentions, and judgments of an intelligent agent can be expressed suitably for automated reasoning.

1. **What are the elements of propositional logic?**

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.

1. **What is Skolemization?**

Skolemization in Artificial Intelligence is a procedure used when there is a requirement of the reduction of any first-order formula to its Skolem normal form. This is usually done when there is a need for proving a theorem by using programming.

1. **What are classes of agents?**

Simple Reflex Agent

Model-based reflex agent

Goal-based agents

Utility-based agent

Learning agent

1. **Define terms.**

Term = logical expression that refers to an object. There are 2 kinds of terms:

constant symbols: Table, Computer

function symbols: LeftLeg(Pete), Sqrt(3), Plus(2,3) etc

1. **What is Data driven search?**

Forward chaining is the logical process of inferring unknown truths from known data and moving forward using determined conditions and rules to find a solution. The opposite of forward chaining is backward chaining.

1. **Define Belief - desire - intention architecture?**

The belief–desire–intention software model (BDI) is a software model developed for programming intelligent agents. Superficially characterized by the implementation of an agent's beliefs, desires and intentions, it actually uses these concepts to solve a particular problem in agent programming.

1. **Define Coherence.**

Coherence arguments say that if an entity's preferences do not adhere to the axioms of expected utility theory, then that entity is susceptible to losing things that it values. This does not imply that advanced AI systems must adhere to these axioms ('be coherent'), or that they must be goal-directed.

1. **State the advantage of horizontal layered architectures**.

The advantage of horizontal layer architecture is that only n layers are required for mapping to n different types of behaviors. However, a mediator function is used to control the inconsistent actions between layer interactions.

1. **Define description logics**.

A description logic is used to describe classes, properties, and individuals. One of the main ideas behind a description logic is to separate a terminological knowledge base that describes the terminology, which should remain constant as the domain being modeled changes, and an assertional knowledge base that describes what is true in some domain at some point in time.

**UNIT - III**

1. **What is the basic strategy in game playing in artificial intelligence?**

A strategy defines a complete plan of action for a given player. Given enough processing time an optimal strategy can be found for games of perfect information by enumerating paths of a game tree. However, in practice this can only be done for small games.

# Mini-Max Algorithm in Artificial Intelligence

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

## Pseudo-code for MinMax Algorithm:

1. function minimax(node, depth, maximizingPlayer) is
2. **if** depth ==0 or node is a terminal node then
3. **return** **static** evaluation of node
5. **if** MaximizingPlayer then      // for Maximizer Player
6. maxEva= -infinity
7. **for** each child of node **do**
8. eva= minimax(child, depth-1, **false**)
9. maxEva= max(maxEva,eva)        //gives Maximum of the values
10. **return** maxEva
12. **else**                         // for Minimizer player
13. minEva= +infinity
14. **for** each child of node **do**
15. eva= minimax(child, depth-1, **true**)
16. minEva= min(minEva, eva)         //gives minimum of the values
17. **return** minEva

**Initial call:**

**Minimax(node, 3, true)**

## Working of Min-Max Algorithm:

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

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



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

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



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

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



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

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



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

## Properties of Mini-Max algorithm:

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

## Limitation of the minimax Algorithm:

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

* 1. **Why is game theory important in AI ?**

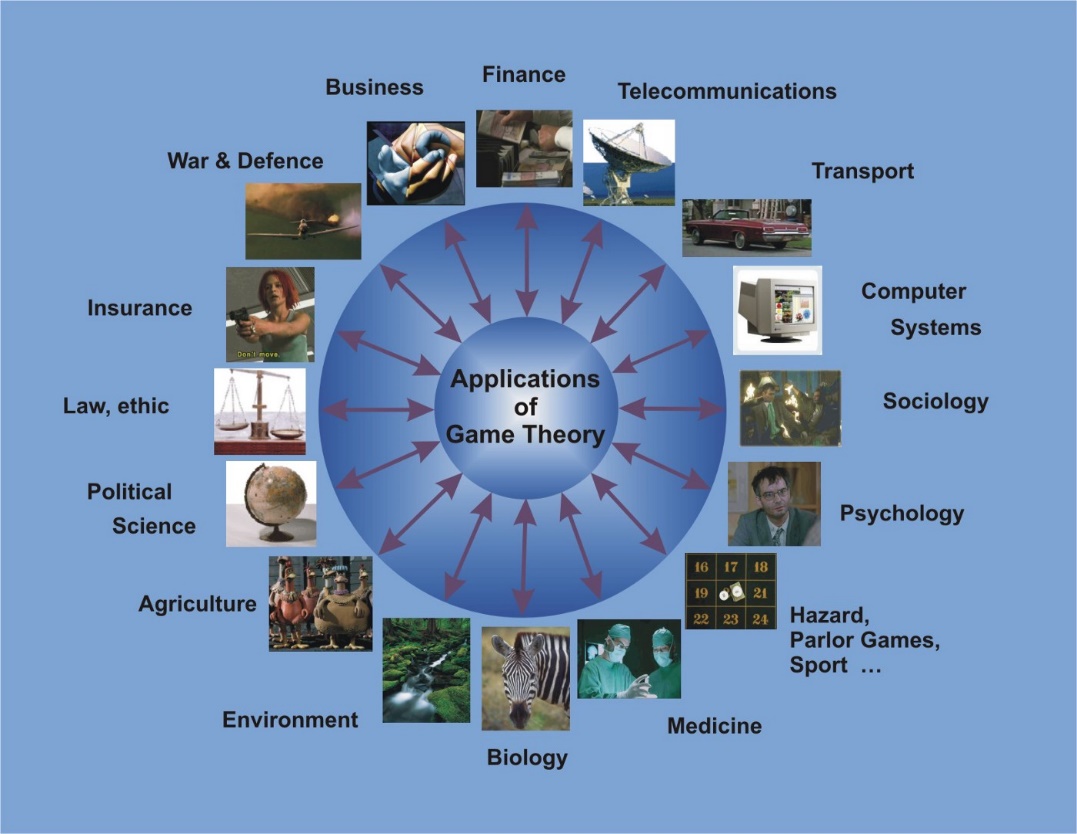
Game Theory is a branch of mathematics used to model the strategic interaction between different players in a context with predefined rules and outcomes. Game Theory can be applied in different ambit of Artificial Intelligence:

* Multi-agent AI systems.
* Imitation and Reinforcement Learning.
* Adversary training in Generative Adversarial Networks (GANs).

Game Theory can also be used to describe many situations in our daily life and Machine Learning models (Figure 1).

For example, a Classification algorithm such as [SVM (Support Vector Machines)](https://towardsdatascience.com/svm-feature-selection-and-kernels-840781cc1a6c) can be explained in terms of a two-player game in which one player is challenging the other to find the best hyper-plane giving him the most difficult points to classify. The game will then converge to a solution which will be a trade-off between the strategic abilities of the two players (eg. how well the fist player was challenging the second one to classify difficult data points and how good was the second player to identify the best decision boundary).

Figure 1: Game Theory Applications [1]



**Game Theory**

Game Theory can be divided into 5 main types of games:

* **Cooperative vs Non-Cooperative Games:**In cooperative games, participants can establish alliances in order to maximise their chances to win the game (eg. negotiations). In non-cooperative games, participants can’t instead form alliances (eg. wars).
* **Symmetric vs Asymmetric Games:**In a symmetric game all the participants have the same goals and just their strategies implemented in order to achieve them will determine who wins the game (eg. chess). In asymmetric games instead, the participants have different or conflicting goals.
* **Perfect vs Imperfect Information Games:**In Perfect Information games all the players can see the other players moves (eg. chess). Instead, in Imperfect Information games, the other players' moves are hidden (eg. card games).
* **Simultaneous vs Sequential Games:**In Simultaneous games, the different players can take actions concurrently. Instead in Sequential games, each player is aware of the other players' previous actions (eg. board games).
* **Zero-Sum vs Non-Zero Sum Games:**In Zero Sum games, if a player gains something that causes a loss to the other players. In Non-Zero Sum games, instead, multiple players can take benefit of the gains of another player.

Different aspects of Game Theory are commonly used in Artificial Intelligence, I will now introduce you to the Nash Equilibrium, Inverse Game Theory and give you some practical examples.

If you are interested in implementing Game Theory Algorithms in Python, the [Nashpy library](https://nashpy.readthedocs.io/en/stable/) is a good place where to start.

**Nash Equilibrium**

The Nash Equilibrium is a condition in which all the players involved in the game agree that there is no best solution to the game than the actual situation they are in at this point. None of the players would have an advantage in changing their current strategy (based on the decisions made by the other players).

Following our example of before, an example of Nash Equilibrium can be when the SVM classifier agrees on which hyper-plane to use classify our data.

One of the most common examples used to explain Nash Equilibrium is the Prisoner’s Dilemma. Let’s imagine two criminals get arrested and they are held in confinement without having any possibility to communicate with each other (Figure 2).

* If any of the two prisoners will confess the other committed a crime, the first one will be set free while the other will spend 10 years in prison.
* If neither of them confesses they spend just one year in prison for each.
* If they both confess, they instead both spend 5 years in prison.

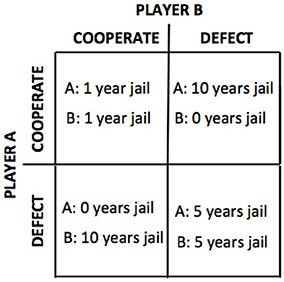


Figure 2: Payoff Matrix [2]

In this case, the Nash Equilibrium is reached when both criminals betray each other.

An easy way to find out if a game has reached a Nash Equilibrium can be to reveal your strategy to your opponents. If after your revelation, none of them changes their strategy, the Nash Equilibrium is demonstrated.

Unfortunately, a Nash Equilibrium is easier to be achieved in Symmetric than Asymmetric games. Asymmetric games are in fact the most common in real-world applications and Artificial Intelligence.

**Inverse Game Theory**

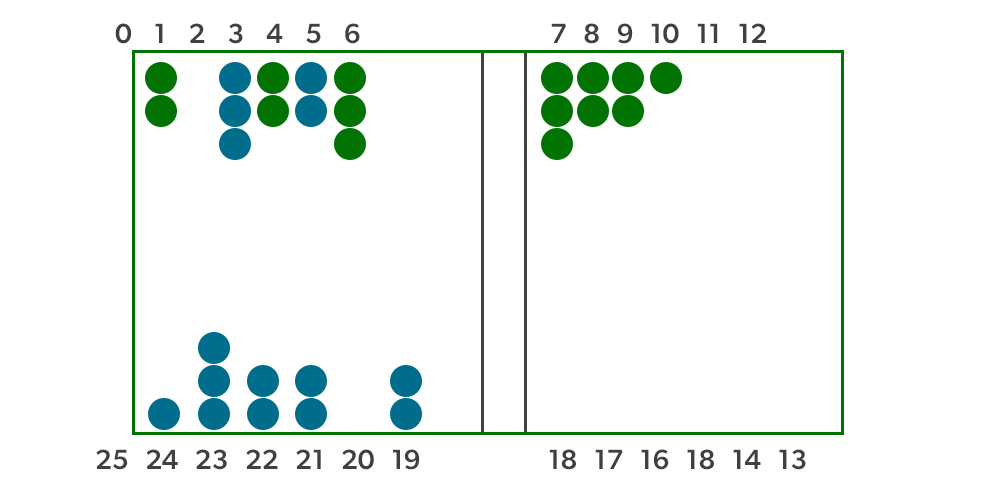
Game Theory aims to understand the dynamics of a game to optimise the possible outcome of its players. Inverse Game Theory instead aims to design a game based on the players' strategies and aims. Inverse Game Theory plays an important role in designing AI Agents environments.

* 1. **Draw the schematic game tree for a backgammon position with help of stochastic games.**

Many unforeseeable external occurrences can place us in unforeseen circumstances in real life. Many games, such as dice tossing, have a random element to reflect this unpredictability. These are known as stochastic games. Backgammon is a classic game that mixes skill and luck.

**EXAMPLE :**

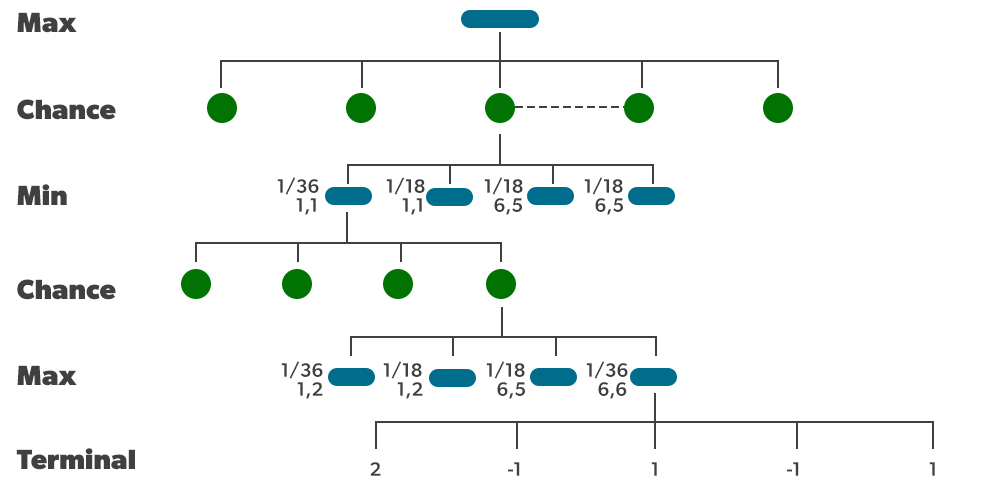
Has rolled a 6–5 and has four alternative moves in the backgammon scenario.



This is a standard backgammon position. The object of the game is to get all of one’s pieces off the board as quickly as possible. White moves in a clockwise direction toward 25, while Black moves in a counterclockwise direction toward 0. Unless there are many opponent pieces, a piece can advance to any position; if there is only one opponent, it is caught and must start over. White has rolled a 6–5 and must pick between four valid moves: (5–10,5–11), (5–11,19–24), (5–10,10–16), and (5–11,11–16), where the notation (5–11,11–16) denotes moving one piece from position 5 to 11 and then another from 11 to 16.

**Stochastic game tree for a backgammon position**

White knows his or her own legal moves, but he or she has no idea how Black will roll, and thus has no idea what Black’s legal moves will be. That means White won’t be able to build a normal game tree-like in chess or tic-tac-toe. In backgammon, in addition to M A X and M I N nodes, a game tree must include chance nodes. The figure below depicts chance nodes as circles. The possible dice rolls are indicated by the branches leading from each chance node; each branch is labelled with the roll and its probability. There are 36 different ways to roll two dice, each equally likely, yet there are only 21 distinct rolls because a 6–5 is the same as a 5–6. P (1–1) = 1/36 because each of the six doubles (1–1 through 6–6) has a probability of 1/36. Each of the other 15 rolls has a 1/18 chance of happening.



The following phase is to learn how to make good decisions. Obviously, we want to choose the move that will put us in the best position. Positions, on the other hand, do not have specific minimum and maximum values. Instead, we can only compute a position’s anticipated value, which is the average of all potential outcomes of the chance nodes.

As a result, we can generalize the deterministic minimax value to an expected-minimax value for games with chance nodes. Terminal nodes, MAX and MIN nodes (for which the dice roll is known), and MAX and MIN nodes (for which the dice roll is unknown) all function as before. We compute the expected value for chance nodes, which is the sum of all outcomes, weighted by the probability of each chance action.

**4. Describe how the minimax and alpha–beta algorithms change for two-player, nonzero-sum games in which each player has a distinct utility function and both utility functions are known to both players. If there are no constraints on the two terminal utilities, is it possible for any node to be pruned by alpha–beta? What if the player’s utility functions on any state differ by at most a constant k, making the game almost cooperative?**

Ever since the advent of Artificial Intelligence (AI), game playing has been one of the most interesting applications of AI. The first chess programs were written by Claude Shannon and by Alan Turing in 1950, almost as soon as the computers became programmable. Games such as chess, tic-tac-toe, and Go are interesting because they offer a pure abstraction of the competition between the two armies. It is this abstraction which makes game playing an attractive area for AI research. we will go through the basics of the Minimax algorithm along with the functioning of the algorithm. We will also take a look at the optimization of the minimax algorithm, alpha-beta pruning.

**What is the Minimax algorithm?**

Minimax is a recursive algorithm which is used to choose an optimal move for a player assuming that the other player is also playing optimally. It is used in games such as tic-tac-toe, go, chess, Isola, checkers, and many other two-player games. Such games are called games of perfect information because it is possible to see all the possible moves of a particular game. There can be two-player games which are not of perfect information such as Scrabble because the opponent’s move cannot be predicted.It is similar to how we think when we play a game: “if I make this move, then my opponent can only make only these moves,” and so on. Minimax is called so because it helps in minimizing the loss when the other player chooses the strategy having the maximum loss.

## Terminology

* **Game Tree**: It is a structure in the form of a tree consisting of all the possible moves which allow you to move from a state of the game to the next state.

A game can be defined as a search problem with the following components:

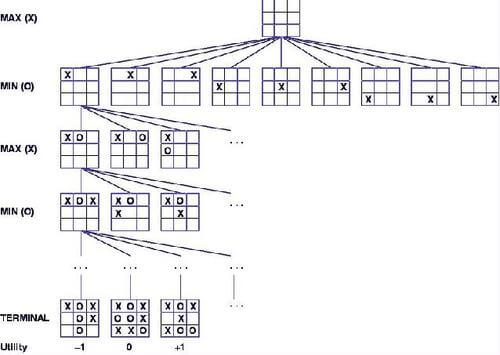
* **Initial state**: It comprises the position of the board and showing whose move it is.
* **Successor function**: It defines what the legal moves a player can make are.
* **Terminal state**: It is the position of the board when the game gets over.
* **Utility function**: It is a function which assigns a numeric value for the outcome of a game. For instance, in chess or tic-tac-toe, the outcome is either a win, a loss, or a draw, and these can be represented by the values +1, -1, or 0, respectively. There are games that have a much larger range of possible outcomes; for instance, the utilities in backgammon varies from +192 to -192. A utility function can also be called a payoff function.

## How does the algorithm work?

There are two players involved in a game, called MIN and MAX. The player MAX tries to get the highest possible score and MIN tries to get the lowest possible score, i.e., MIN and MAX try to act opposite of each other.

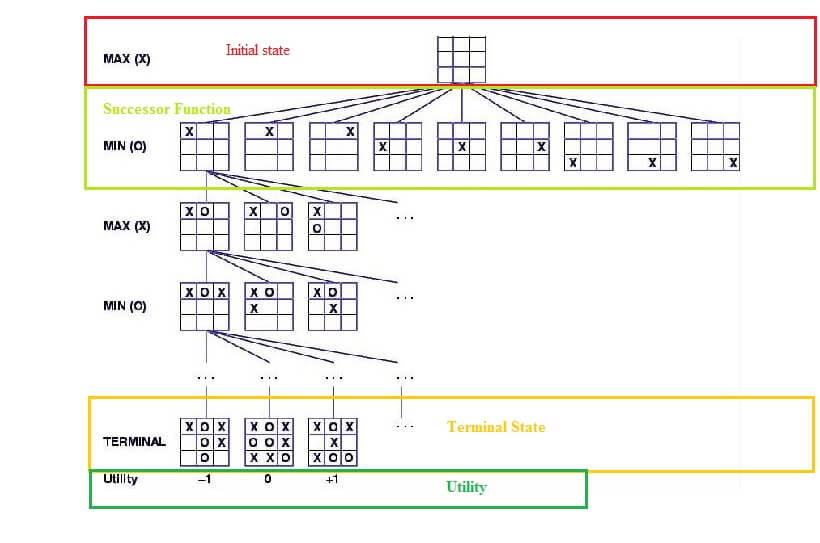
The general process of the Minimax algorithm is as follows:

**Step 1**: First, generate the entire game tree starting with the current position of the game all the way upto the terminal states. This is how the game tree looks like for the game tic-tac-toe.

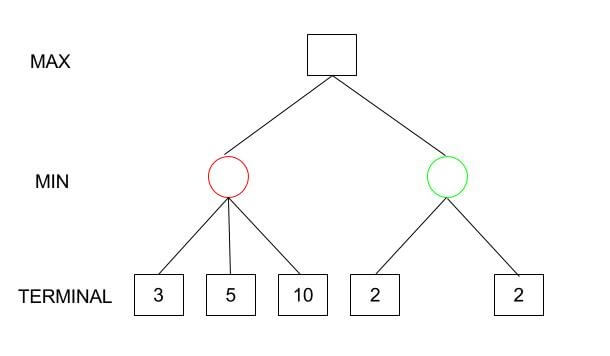
[](https://blog-c7ff.kxcdn.com/blog/wp-content/uploads/2017/03/tic-tac-toe.jpg)

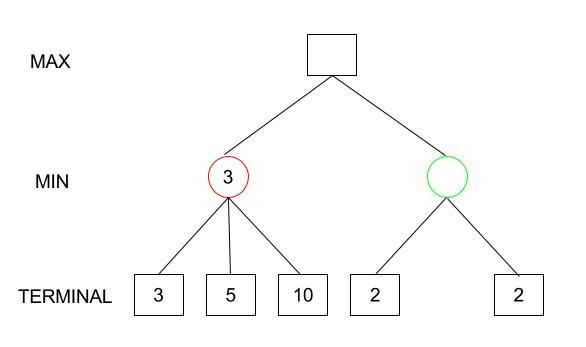
Let us understand the defined terminology in terms of the diagram above.

1. The initial state is the first layer that defines that the board is blank it’s MAX’s turn to play.
2. Successor function lists all the possible successor moves. It is defined for all the layers in the tree.
3. Terminal State is the last layer of the tree that shows the final state, i.e whether the player MAX wins, loses, or ties with the opponent.
4. Utilities in this case for the terminal states are 1, 0, and -1 as discussed earlier, and they can be used to determine the utilities of the other nodes as well.

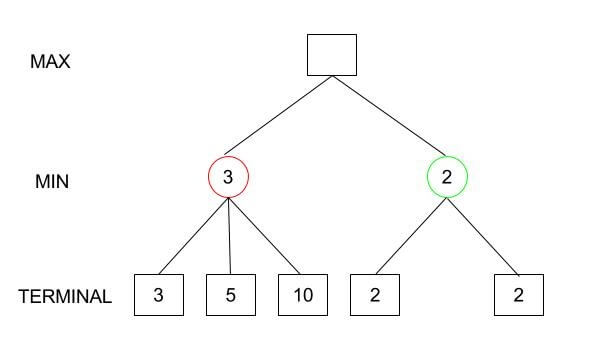
[](https://blog-c7ff.kxcdn.com/blog/wp-content/uploads/2017/03/ttt-minimax.jpg)

**Step 2**: Apply the utility function to get the utility values for all the terminal states.  
**Step 3**: Determine the utilities of the higher nodes with the help of the utilities of the terminal nodes. For instance, in the diagram below, we have the utilities for the terminal states written in the squares.

[](https://blog-c7ff.kxcdn.com/blog/wp-content/uploads/2017/03/Minimax-1.jpg)

Let us calculate the utility for the left node(red) of the layer above the terminal. Since it is the move of the player MIN, we will choose the minimum of all the utilities. For this case, we have to evaluate MIN{3, 5, 10}, which we know is certainly 3. So the utility for the red node is 3.  
[](https://blog-c7ff.kxcdn.com/blog/wp-content/uploads/2017/03/Minimax-2.jpg)

Similarly, for the green node in the same layer, we will have to evaluate MIN{2,2} which is 2.

[](https://blog-c7ff.kxcdn.com/blog/wp-content/uploads/2017/03/Minimax-3.jpg)

**Step 4**: Calculate the utility values with the help of leaves considering one layer at a time until the root of the tree.  
**Step 5**: Eventually, all the backed-up values reach to the root of the tree, i.e., the topmost point. At that point, MAX has to choose the highest value.

In our example, we only have 3 layers so we immediately reached to the root but in actual games, there will be many more layers and nodes. So we have to evaluate MAX{3,2} which is 3.

Therefore, the best opening move for MAX is the left node(or the red one). This move is called the minimax decision as it maximizes the utility following the assumption that the opponent is also playing optimally to minimize it.

To summarize,

Minimax Decision = MAX{MIN{3,5,10},MIN{2,2}}  
= MAX{3,2}  
= 3

## Optimization

Game trees are, in general, very time consuming to build, and it’s only for simple games that it can be generated in a short time.

If there are \(b\) legal moves, i.e., \(b\) nodes at each point and the maximum depth of the tree is \(m\), the time complexity of the minimax algorithm is of the order \(b^m (O(b^m))\).

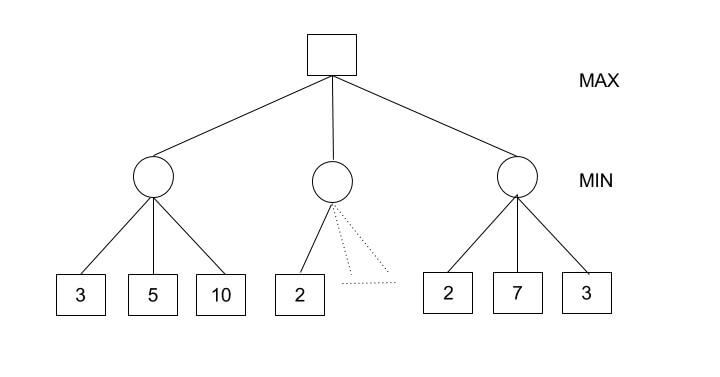
Tocurb this situation, there are a few optimizations that can be added to the algorithm.

Fortunately, it is viable to find the actual minimax decision without even looking at every node of the game tree. Hence, we eliminate nodes from the tree without analyzing, and this process is called pruning.

## Alpha-beta pruning

The method that we are going to look in this article is called alpha-beta pruning.

If we apply alpha-beta pruning to a standard minimax algorithm, it returns the same move as the standard one, but it removes (prunes) all the nodes that are possibly not affecting the final decision.

Let us understand the intuition behind this first and then we will formalize the algorithm. Suppose, we have the following game tree:  
[](https://blog-c7ff.kxcdn.com/blog/wp-content/uploads/2017/03/Minimax-algorithm-with-alpha-beta-pruning-for-AI-1.jpg)

In this case,  
Minimax Decision = MAX{MIN{3,5,10}, MIN{2,a,b}, MIN{2,7,3}}  
= MAX{3,c,2}  
= 3

You would be surprised!

How could we calculate the maximum with a missing value? Here is the trick. MIN{2,a,b} would certainly be less than or equal to 2, i.e., c<=2 and hence MAX{3,c,2} has to be 3.

The question now is do we really need to calculate c? Of course not.

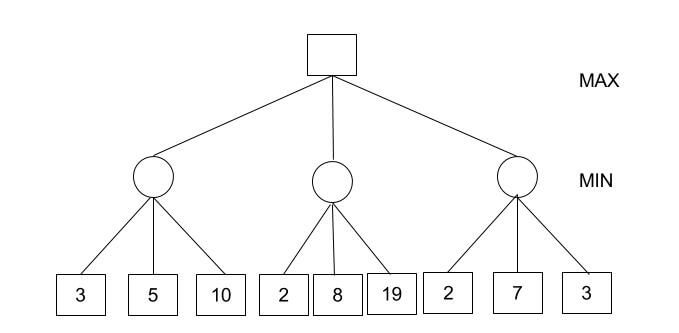
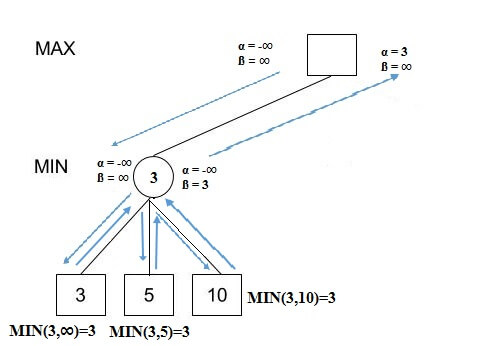
We could have reached a conclusion without looking at those nodes. And this is where alpha-beta pruning comes into the picture.

### A few definitions:

**Alpha:** It is the best choice so far for the player MAX. We want to get the highest possible value here.  
**Beta:** It is the best choice so far for MIN, and it has to be the lowest possible value.

**Note:** Each node has to keep track of its alpha and beta values. Alpha can be updated only when it’s MAX’s turn and, similarly, beta can be updated only when it’s MIN’s chance.

**How does alpha-beta pruning work?**

1. Initialize alpha = -infinity and beta = infinity as the worst possible cases. The condition to prune a node is when alpha becomes greater than or equal to beta.[](https://blog-c7ff.kxcdn.com/blog/wp-content/uploads/2017/03/alpha-beta-pruning-1.jpg)
2. Start with assigning the initial values of alpha and beta to root and since alpha is less than beta we don’t prune it.
3. Carry these values of alpha and beta to the child node on the left. And now from the utility value of the terminal state, we will update the values of alpha and be, so we don’t have to update the value of beta. Again, we don’t prune because the condition remains the same. Similarly, the third child node also. And then backtracking to the root we set alpha=3 because that is the minimum value that alpha can have.
4. [](https://blog-c7ff.kxcdn.com/blog/wp-content/uploads/2017/03/Alpha-beta-1.jpg)
5. Now, alpha=3 and beta=infinity at the root. So, we don’t prune. Carrying this to the center node, and calculating MIN{2, infinity}, we get alpha=3 and beta=2.
6. Prune the second and third child nodes because alpha is now greater than beta.
7. Alpha at the root remains 3 because it is greater than 2. Carrying this to the rightmost child node, evaluate MIN{infinity,2}=2. Update beta to 2 and alpha remains 3.
8. Prune the second and third child nodes because alpha is now greater than beta.
9. Hence, we get 3, 2, 2 at the left, center, and right MIN nodes, respectively. And calculating MAX{3,2,2}, we get 3. Therefore, without even looking at four leaves we could correctly find the minimax decision.

## Conclusion

Games are very appealing and writing game-playing programs is perhaps even more exciting. What Grand Prix racing is to the car industry, game playing is to AI.

Just as we would not expect a racing car to run perfectly on a bumpy road, we should not expect game playing algorithms to be perfect for every situation.

So is the minimax algorithm. It may not be the best solution to all kinds of computer games that need to have AI.

But given a good implementation, it can create a tough competitor.

# How does Monte Carlo tree search work with suitable example?

**Monte Carlo Tree Search** (MCTS) is a search technique in the field of Artificial Intelligence (AI). It is a probabilistic and heuristic driven search algorithm that combines the classic tree search implementations alongside machine learning principles of reinforcement learning.In tree search, there’s always the possibility that the current best action is actually not the most optimal action. In such cases, MCTS algorithm becomes useful as it continues to evaluate other alternatives periodically during the learning phase by executing them, instead of the current perceived optimal strategy. This is known as the ”

***exploration-exploitation trade-off*** “.

# Monte Carlo Tree Search (MCTS) algorithm:

In MCTS, nodes are the building blocks of the search tree. These nodes are formed based on the outcome of a number of simulations. The process of Monte Carlo Tree Search can be broken down into four distinct steps, viz., selection, expansion, simulation and backpropagation. Each of these steps is explained in details below:

* **Selection:** In this process, the MCTS algorithm traverses the current tree from the root node using a specific strategy. The strategy uses an evaluation function to optimally select nodes with the highest estimated value. MCTS uses the Upper Confidence Bound (UCB) formula applied to trees as the strategy in the selection process to traverse the tree. It balances the exploration-exploitation trade-off. During tree traversal, a node is selected based on some parameters that return the maximum value. The parameters are characterized by the formula that is typically used for this purpose is given below.



where; Si = value of a node i xi = empirical mean of a node i C = a constant

t = total number of simulations

When traversing a tree during the selection process, the child node that returns the greatest value from the above equation will be one that will get selected.

During traversal, once a child node is found which is also a leaf node, the MCTS jumps into the expansion step.

* **Expansion:**

In this process, a new child node is added to the tree to that node which was optimally reached during the selection process.

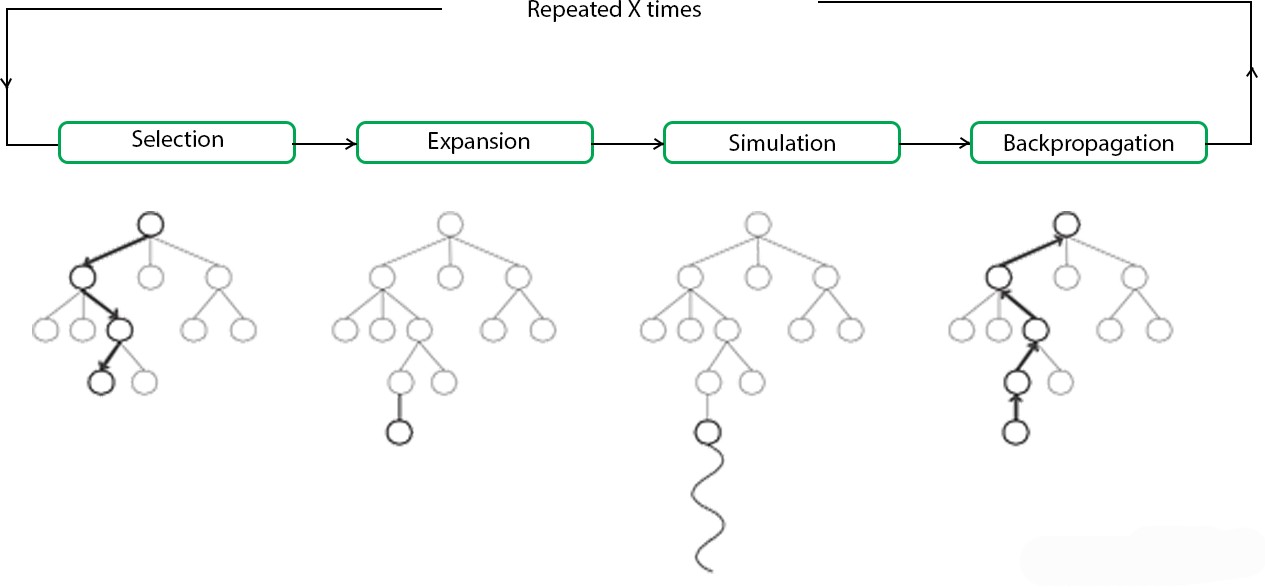
* **Simulation:**

In this process, a simulation is performed by choosing moves or strategies until a result or predefined state is achieved.

* **Backpropagation:**

After determining the value of the newly added node, the remaining tree must be updated. So, the backpropagation process is performed, where it back propagates from the new node to the root node. During the process, the number of simulations stored in each node is incremented. Also, if the new node's simulation results in a win, then the number of wins is also incremented.

The above steps can be visually understood by the diagram given below:



These types of algorithms are particularly useful in turn based games where there is no element of chance in the game mechanics, such as Tic Tac Toe, Connect 4, Checkers, Chess, Go, etc. This has recently been used by Artificial Intelligence Programs like AlphaGo, to play against the world’s top Go players. But, its application is not limited to games only. It can be used in any situation which is described by state-action pairs and simulations used to forecast outcomes.

# Advantages of Monte Carlo Tree Search:

1. MCTS is a simple algorithm to implement.
2. Monte Carlo Tree Search is a heuristic algorithm. MCTS can operate effectively without any knowledge in the particular domain, apart from the rules and end conditions, and can find its own moves and learn from them by playing random playouts.
3. The MCTS can be saved in any intermediate state and that state can be used in future use cases whenever required.
4. MCTS supports asymmetric expansion of the search tree based on the circumstances in which it is operating.

# Disadvantages of Monte Carlo Tree Search:

1. As the tree growth becomes rapid after a few iterations, it requires a huge amount of memory.
2. There is a bit of a reliability issue with Monte Carlo Tree Search. In certain scenarios, there might be a single branch or path that might lead to loss against the opposition when implemented for those turn-based games. This is mainly due to the vast amount of combinations and each of the nodes might not be visited enough number of times to understand its result or outcome in the long run.
3. MCTS algorithm needs a huge number of iterations to be able to effectively decide the most efficient path. So, there is a bit of a speed issue there.
4. **How is CSP formulated as a search problem? What is backtracking search in CSP?**

CSPs represent a state with a set of variable/value pairs and represent the conditions for a solution by a set of constraints on the variables. Many important real-world problems can be described as CSPs.

CSP (constraint satisfaction problem): Use a factored representation (a set of variables, each of which has a value) for each state, a problem that is solved when each variable has a value that satisfies all the constraints on the variable is called a CSP.

A CSP consists of 3 components:

·X is a set of variables, {X1, …, Xn}.

·D is a set of domains, {D1, …, Dn}, one for each variable.

Each domain Di consists of a set of allowable values, {v1, …, vk} for variable Xi.

C is a set of constraints that specify allowable combination of values.

Each constraint Ci consists of pair <scope, rel>, where scope is a tuple of variables that participate in the constraint, and rel is a relation that defines the values that those variables can take on.

**To formulate a CSP:**

define the variables to be the regions X = {WA, NT, Q, NSW, V, SA, T}.

The domain of each variable is the set Di = {red, green, blue}.

The constraints is C = {SA≠WA, SAW≠NT, SA≠Q, SA≠NSW, SA≠V, WA≠NT, NT≠Q, Q≠NSW, NSW≠V}. ( SA≠WA is a shortcut for <(SA,WA),SA≠WA>. )

Constraint graph: The nodes of the graph correspond to variables of the problem, and a link connects to any two variables that participate in a constraint.

**Advantage of formulating a problem as a CSP:**

1) CSPs yield a natural representation for a wide variety of problems;

2) CSP solvers can be faster than state-space searchers because the CSP solver can quickly eliminate large swatches of the search space;

3) With CSP, once we find out that a partial assignment is not a solution, we can immediately discard further refinements of the partial assignment.

4) We can see why a assignment is not a solution—which variables violate a constraint.

**Backtracking search for CSPs**

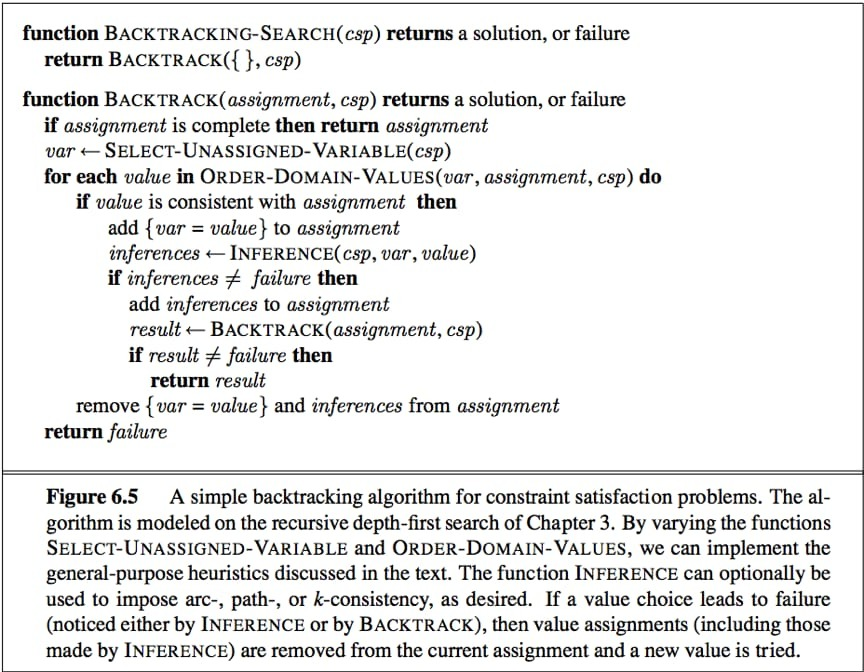
Backtracking search, a form of depth-first search, is commonly used for solving CSPs. Inference can be interwoven with search.

Commutativity: CSPs are all commutative. A problem is commutative if the order of application of any given set of actions has no effect on the outcome.

Backtracking search: A depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.

Backtracking algorithm repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value.

There is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, action function, transition model, or goal test.

BACKTRACKING-SARCH keeps only a single representation of a state and alters that representation rather than creating a new ones.

To solve CSPs efficiently without domain-specific knowledge, address following questions:

1)function SELECT-UNASSIGNED-VARIABLE: which variable should be assigned next? function ORDER-DOMAIN-VALUES: in what order should its values be tried?

2)function INFERENCE: what inferences should be performed at each step in the search?

3)When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

**7. How is it examine the game of Kriegspiel, a partially observable variant of chess in which pieces can move but are completely invisible to the opponent (7)**

**Kriegspiel:** A partially observable variant of chess in which pieces can move but are completely invisible to the opponent.

The rules of Kriegspiel are as follows:

* White and Black each see a board containing only their own pieces.
* A referee, who can see all the pieces, adjudicates the game and periodically makes announcements that are heard by both players i.e On his turn, White proposes to the referee any move that would be legal if there were no black pieces. If the move is in fact not legal (because of the black pieces), the referee announces “illegal.” In this case, White may keep proposing moves until a legal one is found—and learns more about the location of Black’s pieces in the process.
* Once a legal move is proposed, the referee announces one or more of the following: “Capture on square *X*” if there is a capture, and “Check by *D*” if the black king is in check, where *D* is the direction of the check, and can be one of “Knight,” “Rank,” “File,” “Long diagonal,” or “Short diagonal.” (In case of discovered check, the referee may make two “Check” announcements.)
* If Black is checkmated or stalemated, the referee says so; otherwise, it is Black’s turn to move

Initially, White’s belief state is a singleton because Black’s pieces haven’t moved yet. After White makes a move and Black responds, White’s belief state contains 20 positions because Black has 20 replies to any White move. Keeping track of the belief state as the game progresses is exactly the problem of **state estimation.** We can map Kriegspiel state estimation directly onto the partially observable, nondeterministic framework, if we consider the opponent as the source of nondeterminism; that is, the RESULTS of White’s move are composed from the (predictable) outcome of White’s own move and the unpredictable outcome given by Black’s reply.

By the rules mentioned above, Kriegspiel Chess is played. It is played with three boards, where one board is given for the player who plays white, one for the player who plays the black, and one for the Referee and spectators. The board which is seen by Referee and Spectators contains both the White and Black pieces, while the boards with the players contain only whatever colour they have got.

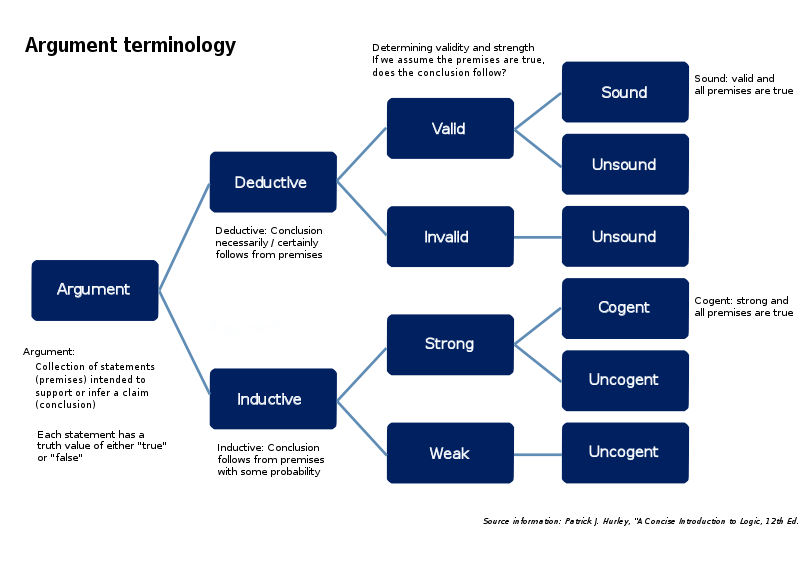
**UNIT - IV**

1. **What are the fundamentals of logical reasoning?**

* Logical or deductive reasoning involves using a given set of facts or data to deduce other facts by reasoning logically. It involves drawing specific conclusions based on premises.
* Or we can say, "Reasoning is a way to infer facts from existing data." It is a general process of thinking rationally, to find valid conclusions. In artificial intelligence, the reasoning is essential so that the machine can also think rationally as a human brain, and can perform like a human.
* Two kinds of logical reasoning are often distinguished in addition to formal deduction: induction and abduction. Given a precondition or premise, a conclusion or logical consequence and a rule or material conditional that implies the conclusion given the precondition

Each one of this can explain with following example:

* **Deductive reasoning** is the mental process of drawing deductive inferences. An inference is deductively valid if its conclusion follows logically from its premises, i.e. if it is impossible for the premises to be true and the conclusion to be false. For example, the inference from the premises "all men are mortal" and "Socrates is a man" to the conclusion "Socrates is mortal" is deductively valid.Deductive reasoning determines whether the truth of a conclusion can be determined for that rule, based solely on the truth of the premises. Example: "When it rains, things outside get wet. The grass is outside, therefore: when it rains, the grass gets wet." Mathematical logic and philosophical logic are commonly associated with this type of reasoning.
* **Inductive Reasoning** is a method of reasoning in which a body of observations is synthesized to come up with a general principle.[1] It consists of making broad generalizations based on specific observations.Inductive Reasoning attempts to support a determination of the rule. It hypothesizes a rule after numerous examples are taken to be a conclusion that follows from a precondition in terms of such a rule. Example: "The grass got wet numerous times when it rained, therefore: the grass always gets wet when it rains." This type of reasoning is commonly associated with generalization from empirical evidence. While they may be persuasive, these arguments are not deductively valid: see the problem of induction.
* **Abductive reasoning** (also called abduction,[1] abductive inference,[1] or retroduction[2]) is a form of logical inference formulated and advanced by American philosopher Charles Sanders Peirce beginning in the last third of the 19th century. It starts with an observation or set of observations and then seeks the simplest and most likely conclusion from the observations.Abductive Reasoning. sometimes called inference to the best explanation, selects a cogent set of preconditions. Given a true conclusion and a rule, it attempts to select some possible premises that, if true also, can support the conclusion, though not uniquely. Example: "When it rains, the grass gets wet. The grass is wet. Therefore, it might have rained." This kind of reasoning can be used to develop a hypothesis, which in turn can be tested by additional reasoning or data. Diagnosticians, detectives, and scientists often use this type of reasoning.



Within the context of a mathematical model, these three kinds of reasoning can be described as follows. The construction/creation of the structure of the model is abduction. Assigning values (or probability distributions) to the parameters of the model is induction. Executing/running the model is deduction.

1. **Prove that Propositional logic is a very simple logic and explain with necessary example**

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:**

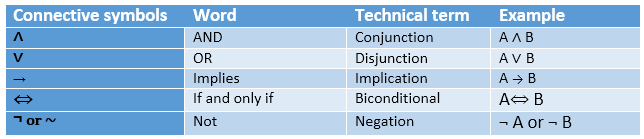
1. a) "It is raining today, and street is wet."
2. b) "Ankit is a doctor, and his clinic is in Mumbai."

## **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:

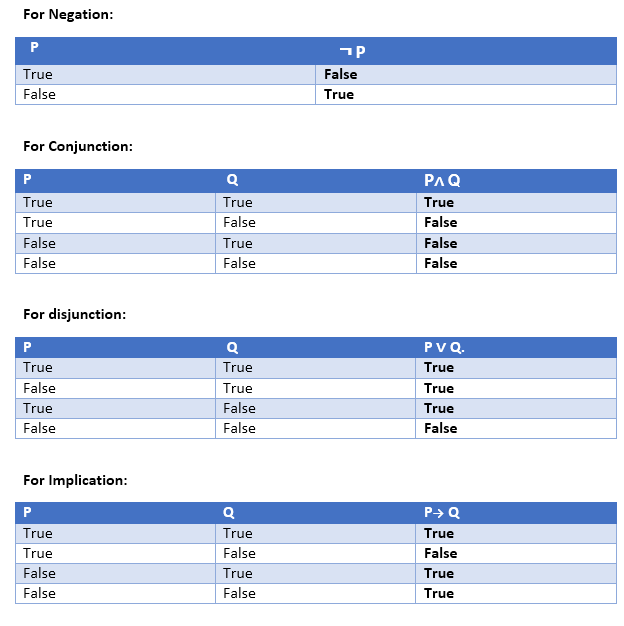
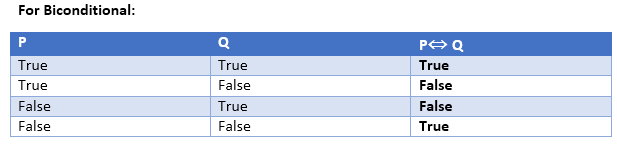
1. **Negation:** A sentence such as ¬ P is called negation of P. A literal can be either Positive literal or negative literal.
2. **Conjunction:** A sentence which has **∧**connective such as, **P ∧ Q** is called a conjunction.  
   **Example:** Rohan is intelligent and hardworking. It can be written 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:**



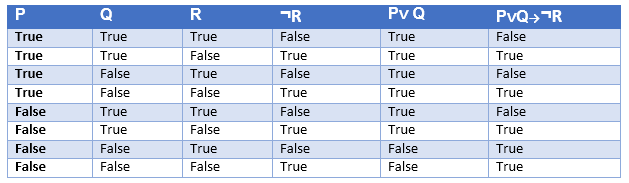
## **Truth Table:**

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:**

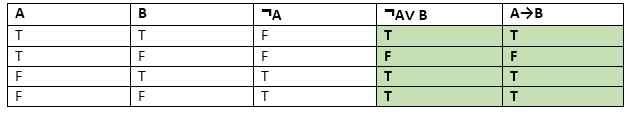
Just like arithmetic operators, there is a precedence order for propositional connectors or logical operators. This order should be followed while evaluating a propositional problem. Following is the list of the precedence order for operators:

|  |  |
| --- | --- |
| **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.

1. **Knowledge Representation in First-Order Logic**

* In first-order logic it would be easy to quantify over all objects that are inside the plane. But basic PDDL does not have a universal quantifier, so we need a different solution. The approach we use is to say that a piece of cargo ceases to be at anywhere when it is in a plane; the cargo only becomes at the new airport when it is unloaded. So At really means “available for use at a given location.”

The following plan is a solution to the problem:

[Load (C1, P1, SFO), Fly(P1, SFO, JFK),Unload(C1, P1, JFK),

Load (C2, P2, JFK), Fly(P2, JFK, SFO),Unload(C2, P2, SFO)] .

* We can take advantage of the power of first-order representations: a single description summarizes the possibility of using any of the planes by implicitly quantifying over p’.
* PDDL is a language that carefully balances the expressiveness of the language with the complexity of the algorithms that operate on it. But some problems remain difficult to express in PDDL. For example, we can’t express the goal “move all the cargo from A to B regardless of how many pieces of cargo there are” in PDDL, but we can do it in first-order logic, using a universal quantifier. Likewise, first-order logic can concisely express global constraints such as “no more than four robots can be in the same place at the same time.” PDDL can only say this with repetitious preconditions on every possible action that involves a move.
* The propositional logic representation of planning problems also has limitations, such as the fact that the notion of time is tied directly to fluents. For example, South2 means “the agent is facing south at time 2.” With that representation, there is no way to say “the agent would be facing south at time 2 if it executed a right turn at time 1; otherwise it would be facing east.” First-order logic lets us get around this limitation by replacing the notion of linear time with a notion of branching situations, using a representation called **situation calculus.**
* The organization of objects into categories is a vital part of knowledge representation. There are two choices for representing categories in first-order logic: predicates and objects. That is, we can use the predicate Basketball (b), or we can reify1 the category as an object, Basketballs. We could then say Member(b, Basketballs ), which we will abbreviate as b∈ Basketballs, to say that b is a member of the category of basketballs. We say Subset(Basketballs, Balls), abbreviated as Basketballs ⊂ Balls, to say that Basketballs is a subcategory of Balls. We will use subcategory, subclass, and subset interchangeably.

## **First-Order logic:**

* First-order logic is another way of knowledge representation in artificial intelligence. It is an extension to propositional logic.
* FOL is sufficiently expressive to represent the natural language statements in a concise way.
* First-order logic is also known as **Predicate logic or First-order predicate logic**. First-order logic is a powerful language that information about the objects in a more easy way and can also express the relationship between those objects.
* First-order logic (like natural language) does not only assume that the world contains facts like propositional logic but also assumes the following things in the world:
  + **Objects:** A, B, people, numbers, colors, wars, theories, squares, pits, wumpus, ......
  + **Relations:** **It can be unary relation such as:** red, round, is adjacent, **or n-any relation such as:** the sister of, brother of, has color, comes between
  + **Function:** Father of, best friend, third inning of, end of, ......
* As a natural language, first-order logic also has two main parts:

Syntax and Semantics

We might say that FOL is a formalized, unambiguous form of ordinary language thatavoids problems like the above. Its invention is one of the most important mathemat-ical achievements of the 19th and 20th centuries, and has had an enormous impact onmathematics, philosophy, and AI.

Basically, the way FOL avoids problems like the above is by

• using brackets and fixed operator-operand ordering to avoid syntactic ambiguity;

• allowing any one symbol (like \I" or \saw") to have just one interpretation (where\interpretation" is a precisely defined notion);

• using formalized versions of and, or, not, implies, is equivalent to, and is identical to, as a means of combining information, where these operators have fixed, well-defined meanings;

• using quantiers 9, 8 and variables like x, y, z to talk about an existing, but unspecied individual, or about all individuals.

## **Syntax of First-Order logic:**

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

### **Basic Elements of First-order logic:**

Following are the basic elements of FOL syntax:

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

### **Atomic sentences:**

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

**Example: Ravi and Ajay are brothers: => Brothers(Ravi, Ajay).  
                Chinky is a cat: => cat (Chinky)**.

### **Complex Sentences:**

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

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

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

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

1. **i). Discuss forward chaining algorithm with suitable example (6)**

**ii). Discuss backward chaining algorithm with suitable example (7)**

**Forward** **and** **backward** **chaining**

The forward-chaining algorithm PL-FC-ENTAILS?(KB , q) determines if a single proposition symbol q —the query—is entailed by a knowledge base of deﬁnite clauses. It begins from known facts (positive literals) in the knowledge base. If all the premises of an implication are known, then its conclusion is added to the set of known facts. For example, if L 1,1 and Breeze are known and (L 1,1 ^ Breeze) ⇒ B 1,1 is in the knowledge base, then B 1,1 can be added. This process continues until the query q is added or until no further inferences can be made. The detailed algorithm is shown in Figure 7. 15; the main point to remember is that it runs in linear time.

|  |  |
| --- | --- |
| **function** PL-FC-ENTAILS?(KB , q) **returns** true or false  **inputs**: KB , the knowledge base, a set of propositional deﬁnite clauses  q, the query, a proposition symbol  count ← a table, where count[c] is the number of symbols in c’s premise  inferred ← a table, where inferred[s] is initially false for all symbols  agenda ← a queue of symbols, initially symbols known to be true in KB  **while** agenda is not empty **do**  p ← POP(agenda)  **if** p = q **then** **return** true  **if** inferred[p] = false **then**  inferred[p] ← true  **for** **each** clause c in KB where p is in c .PREMISE **do**  decrement count[c]  **if** count[c] = 0 **then** add c .CONCLUSION to agenda  **return** false | |
| **Figure** **7.15** The forward-chaining algorithm for propositional logic. The agenda keeps track of symbols known to be true but not yet “processed.” The count table keeps track of how many premises of each implication are as yet unknown. Whenever a new symbol p from the agenda is processed, the count is reduced by one for each implication in whose premise p appears (easily identiﬁed in constant time with appropriate indexing.) If a count reaches zero, all the premises of the implication are known, so its conclusion can be added to the agenda. Finally, we need to keep track of which symbols have been processed; a symbol that is already in the set of inferred symbols need not be added to the agenda again. This avoids redundant work and prevents loops caused by implications such as P ⇒ Q and Q ⇒ P . |

The best way to understand the algorithm is through an example and a picture. Figure 7. 16(a) shows a simple knowledge base of Horn clauses with A and B as known facts. Figure 7. 16(b) shows the same knowledge base drawn as an **AND–OR** **graph**. In AND–OR graphs, multiple links joined by an arc indicate a conjunction—every link must be proved—while multiple links without an arc indicate a disjunction—any link can be proved. It is easy to see how forward chaining works in the graph. The known leaves

(here, A and B ) are set, and inference propagates up the graph as far as possible. Wherever a conjunction appears, the propagation waits until all the conjuncts are known before proceeding. The reader is encouraged to work through the example in detail.

It is easy to see that forward chaining is **sound**: every inference is essentially an application of Modus Ponens. Forward chaining is also **complete**: every entailed atomic sentence will be derived. The easiest way to see this is to consider the ﬁnal state of the inferred table (after the algorithm reaches a **ﬁxed** **point** where no new inferences are possible). The table contains true for each symbol inferred during the process, and false for all other symbols. We can view the table as a logical model; moreover, *every* *deﬁnite* *clause* *in* *the* *original* *KB* *is* *true* *in* *this* *model.* To see this, assume the opposite, namely that some clause a1^ ... ^ak ⇒ b is false in the model. Then a1 ^ ... ^ ak must be true in the model and b must be false in the model. But this contradicts our assumption that the algorithm has reached a ﬁxed point! We can conclude, therefore, that the set of atomic sentences inferred at the ﬁxed point deﬁnes a model of the original KB. Furthermore, any atomic sentence q that is entailed by the KB must be true in all its models and in this model in particular. Hence, every entailed atomic sentence q must be inferred by the algorithm.

**Forward chaining** is an example of the general concept of **data-driven** reasoning—that is, reasoning in which the focus of attention starts with the known data. It can be used within an agent to derive conclusions from incoming percepts, often without a speciﬁc query in mind. For example, the wumpus agent might TELL its percepts to the knowledge base using

|  |
| --- |
| *Q*  P ⇒ Q  *P*  *M*  *L*  L ∧ M ⇒ P  B ∧ L ⇒ M  A ∧ P ⇒ L  A ∧ B ⇒ L  A  B  *A*  *B*  (b)  (a) |
| **Figure** **7.16** (a) A set of Horn clauses. (b) The corresponding AND– OR graph. |

an incremental forward-chaining algorithm in which new facts can be added to the agenda to initiate new inferences. In humans, a certain amount of data-driven reasoning occurs as new information arrives. For example, if I am indoors and hear rain starting to fall, it might occur to me that the picnic will be canceled. Yet it will probably not occur to me that the seventeenth petal on the largest rose in my neighbor’s garden will get wet; humans keep forward chaining under careful control, lest they be swamped with irrelevant consequences.

**The backward-chaining algorithm**, as its name suggests, works backward from the query. If the query q is known to be true, then no work is needed. Otherwise, the algorithm ﬁnds those implications in the knowledge base whose conclusion is q. If all the premises of one of those implications can be proved true (by backward chaining), then q is true. When applied to the query Q in Figure 7. 16, it works back down the graph until it reaches a set of known facts, A and B , that forms the basis for a proof. The algorithm is essentially identical to the AND -OR -GRAPH -SEARCH algorithm. As with forward chaining, an efﬁcient implementation runs in linear time.

**Backward chaining** is a form of **goal-directed** **reasoning**. It is useful for answering speciﬁc questions such as “What shall I do now?” and “Where are my keys?” Often, the cost of backward chaining is *much* *less* than linear in the size of the knowledge base, because the process touches only relevant facts.

1. **Explain with suitable for each i). Conjunctive normal form for first-order logic**

**ii). The resolution inference rule**

**iii). Completeness of resolution**

**Resolution in First Order Logic (FOL):-**

* Resolution is theorem proving technique that proofs by contradictions.
* It is used, if there are various statements are given & 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 conjunctive normal form or clausal form.

Clause: Disjunction of literals is called clause.

Conjunctive NF: A sentence represented as a conjunction of clauses said to be CNF

**Steps for Resolution:**

1. Conversion of facts into FOL.
2. Convert FOL statements into CNF.
3. Negate the statement which needs to prove (by contradiction).
4. Draw resolution graph (Unification).

**Examples:-**

1. John likes all kind of food.
2. Apple & 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:**

1. John likes peanuts.

**Solution:**

**Step 1:** Conversion of facts into FOL.

1. ∀x: food (x) ⟶ likes (John, x)
2. food (Apple) ^ food (vegetables)
3. ∀x ∀y: eats(x, y) ^ ¬killed(x) ⟶ food(y)
4. Eats(Anil, peanuts) ^ alive(Anil)
5. ∀x: eats(Anil, x) ⟶ eats(Harry, x)
6. ∀x: ¬killed(x) ⟶ alive(x)
7. ∀x: alive(x) ⟶ ¬killed(x)
8. likes(John, Peanuts)

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

(Why CNF? : CNF makes easier for resolution proofs)

1. Eliminate all implications (⟶) & rewrite.
2. ∀x ¬food (x) V likes (John, x)
3. food (Apple) ^ food (vegetables)
4. ∀x ∀y ¬[eats (x, y) ^ ¬killed (x)] V food(y)
5. eats (Anil, peanuts) ^ alive (Anil)
6. ∀x ¬ eats (Anil, x) V eats (Harry, x)
7. ∀x ¬[¬ killed (x)] V alive (x)
8. ∀x ¬ alive(x) V ¬ killed (x)
9. likes (John, Peanuts)
10. Move negation (¬) inwards and rewrite.
11. ∀x ¬food (x) V likes (John, x)
12. food (Apple) ^ food (vegetables)
13. ∀x ∀y ¬eats (x, y) V killed (x) V food(y)
14. eats (Anil, peanuts) ^ alive (Anil)
15. ∀x ¬ eats (Anil, x) V eats (Harry, x)
16. ∀x ¬ killed (x) V alive (x)
17. ∀x ¬ alive(x) V ¬ killed (x)
18. likes (John, Peanuts)
19. Rename variables or Standardize variables.
20. ∀x ¬food (x) V likes (John, x)
21. food (Apple) ^ food (vegetables)
22. ∀x ∀y ¬eats (y, z) V killed (y) ^ food (z)
23. eats (Anil, peanuts) ^ alive (Anil)
24. ∀x ¬ eats (Anil, w) V eats (Harry, w)
25. ∀x ¬killed (g) V alive (g)
26. ∀x ¬ alive(k) V ¬ killed (k)
27. likes (John, Peanuts)
28. Eliminate existential instantiation quantifies by elimination.

{But in this problem there are no Ǝ so all statements remain so}

1. Drop universal quantifiers:
2. ¬food (x) V likes (John, x)
3. food (Apple)
4. food (vegetables)
5. ¬eats (y, z) V killed (y) V food (z)
6. eats (Anil, peanuts)
7. alive (Anil)
8. ¬ eats (Anil, w) V eats (Harry, w)
9. killed (g) V alive (g)
10. ¬ alive(k) V ¬ killed (k)
11. likes (John, Peanuts)

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

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

**Step 4:-** Draw resolution graph.

Now, in the step, we will solve the problem by resolution tree using substitution, for the above problem it will be given as

likes (John, Peanuts) ¬food (x) V likes (John, x)

{peanuts/x}

¬food (peanuts) ¬eats (y, z) V killed (y) V food (z)

{peanuts/z}

¬eats (y, peanuts) V killed (y) eats (Anil, peanuts)

{Anil/y}

killed(Anil) ¬alive (k) V ¬ killed (k)

{Anil/k}

¬alive (Anil) alive (Anil)

{ \_ }

Hence Proved