A local search algorithm in artificial intelligence is a type of optimization algorithm used to find the best solution to a problem by repeatedly making minor adjustments to an initial solution.

When trying to find an exact solution to a problem or when doing so would be too computationally expensive, local search algorithms come in particularly handy.

A **local search algorithm** in artificial intelligence works by starting with an initial solution and then making minor adjustments to it in the hopes of discovering a better one. Every time the algorithm iterates, the current solution is assessed, and a small modification to the current solution creates a new solution. The current solution is then compared to the new one, and if the new one is superior, it replaces the old one. This process keeps going until a satisfactory answer is discovered or a predetermined stopping criterion is satisfied.

Introduction

The local search algorithm in artificial intelligence is a family of optimization algorithms used to find the best possible solution to a problem by iteratively making small changes to an initial solution.

These algorithms are used to solve optimization problems in which an exact solution is either impossible to find or computationally expensive.

Hill climbing, simulated annealing, tabu search, and genetic algorithms are a few examples of different kinds of local search algorithms. Each of these algorithms operates a little bit differently, but they all follow the same fundamental procedure of iteratively creating new solutions and comparing them to the existing solution to determine whether they are superior.

The local search algorithm in artificial intelligence is a crucial tool in the field of artificial intelligence and is frequently employed to address a range of optimization issues.

Applications for local search algorithms include scheduling, routing, and resource allocation. They are particularly helpful for issues where the search space is very large and can be used to solve both discrete and continuous optimization problems.

Local Search

The nodes are expanded systematically by informed and uninformed searches in different ways:

- storing various routes in memory and
- choosing the most appropriate route,

Hence, a solution state is needed to get to the goal node. Beyond these **"classical search algorithms,"** however, there are some **"local search algorithms"** that only consider the solution state required to reach the target node and disregard path cost.

In contrast to multiple paths, a local search algorithm completes its task by traversing a single current node and generally following that node's neighbors.

To solve challenging optimization issues, local search algorithms are frequently combined with other optimization methods like constraint satisfaction or linear programming. Since they can quickly converge on a solution that is close to the optimal solution, even if they do not find the exact optimal solution, they are especially helpful for problems where the search space is very large.

Q. Does a problem that is only optimized work with the local search algorithm?

The local search algorithm does indeed function for perfectly optimized problems. Any node that can provide a solution can be said to have a pure optimization problem. But the objective function's objective is to identify the best state among all of them. Unfortunately, the pure optimization problem is unable to generate excellent solutions to move from the current state to the goal state.

One of the **key benefits of local search algorithms** is that they can be very efficient, particularly when compared to other optimization techniques such as exhaustive search or dynamic programming. This is because local search algorithms only need to explore a relatively small portion of the entire search space, which can save a significant amount of time and computational resources.

However, one of the **main limitations of local search algorithms** is that they can become trapped in local optima, which are solutions that are better than all of their neighbors but are not the best possible solution overall. To overcome this limitation, many local search algorithms use various techniques such as randomization, memory, or multiple starting points to help them escape from local optima and find better solutions.

Working on a Local Search Algorithm

Local search algorithms are a type of optimization algorithm that iteratively improves the solution to a problem by making small, local changes to it. Here are the general steps of a local search algorithm:

- **Initialization**:
  The algorithm starts with an initial solution to the problem. This solution can be generated randomly or using a heuristic.
- **Evaluation**:
  The quality of the initial solution is evaluated using an objective function. The objective function measures how good the solution is, based on the problem constraints and requirements.
- **Neighborhood search**:
  The algorithm generates neighboring solutions by making small modifications to the current solution. These modifications can be random or guided by heuristics.
- **Selection**:
  The neighboring solutions are evaluated using the objective function, and the best solution is selected as the new current solution.
- **Termination**:
  The algorithm terminates when a stopping criterion is met. This criterion can be a maximum number of iterations, a threshold value for the objective function, or a time limit.
- **Solution**:
  The final solution is the best solution found during the search process.

Local search algorithms are frequently employed in situations where it is computationally impractical or impossible to find an exact solution. The traveling salesman problem, the knapsack issue, and the graph coloring issue are a few examples of these issues.

Hill Climbing Algorithm in Artificial Intelligence

- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.
- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.
- It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.
- A node of hill climbing algorithm has two components which are state and value.
- Hill Climbing is mostly used when a good heuristic is available.
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

Features of Hill Climbing:

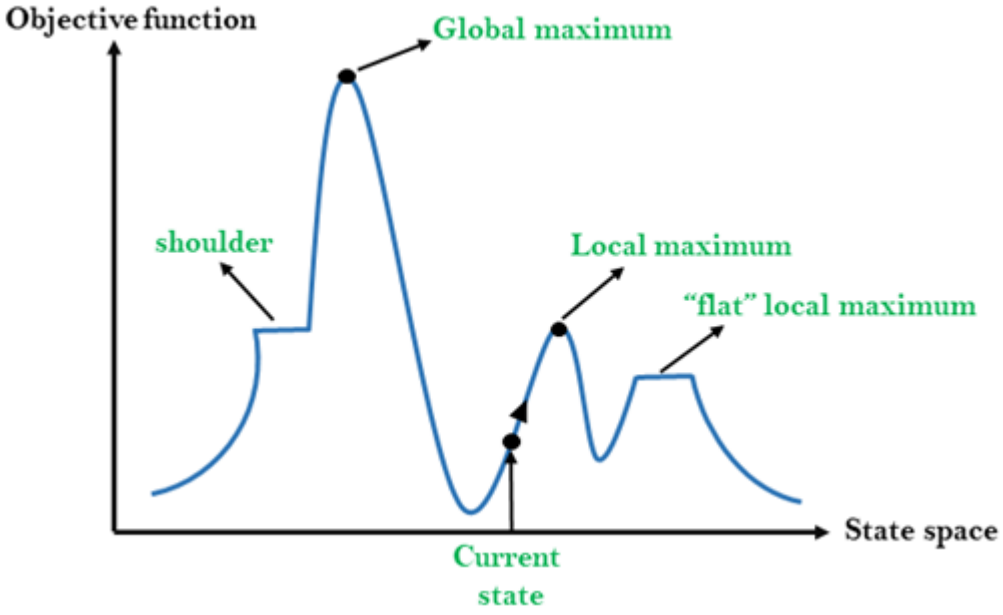
Following are some main features of Hill Climbing Algorithm:

- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

State-space Diagram for Hill Climbing:

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

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



Different regions in the state space landscape:

Local Maximum: Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.

Global Maximum: Global maximum is the best possible state of state space landscape. It has the highest value of objective function.

Current state: It is a state in a landscape diagram where an agent is currently present.

Flat local maximum: It is a flat space in the landscape where all the neighbor states of current states have the same value.

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

Types of Hill Climbing Algorithm:

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

1. Simple Hill Climbing:

Simple hill climbing is the simplest way to implement a hill climbing algorithm. **It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state**. It only checks it's one successor state, and if it finds better than the current state, then move else be in the same state. This algorithm has the following features:

- Less time consuming
- Less optimal solution and the solution is not guaranteed

Algorithm for Simple Hill Climbing:

- **Step 1:** Evaluate the initial state, if it is goal state then return success and Stop.
- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.
- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:
    1. If it is goal state, then return success and quit.
    2. Else if it is better than the current state then assign new state as a current state.
    3. Else if not better than the current state, then return to step2.
- **Step 5:** Exit.

2. Steepest-Ascent hill climbing:

The steepest-Ascent algorithm is a variation of simple hill climbing algorithm. This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbors

Algorithm for Steepest-Ascent hill climbing:

- **Step 1:** Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.
- **Step 2:** Loop until a solution is found or the current state does not change.
    1. Let SUCC be a state such that any successor of the current state will be better than it.
    2. For each operator that applies to the current state:
        I. Apply the new operator and generate a new state.
        II. Evaluate the new state.
        III. If it is goal state, then return it and quit, else compare it to the SUCC.
        IV. If it is better than SUCC, then set new state as SUCC.

V. If the SUCC is better than the current state, then set current state to SUCC.

Step 5: Exit.

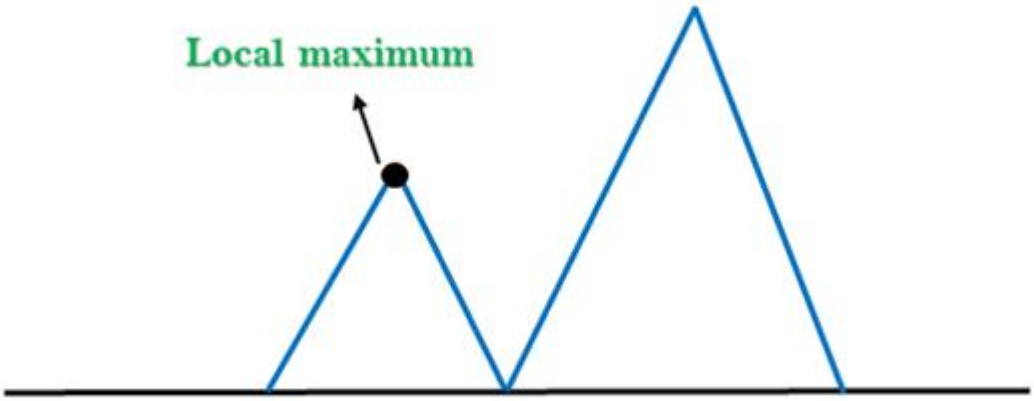
3. Stochastic hill climbing:

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

Problems in Hill Climbing Algorithm:

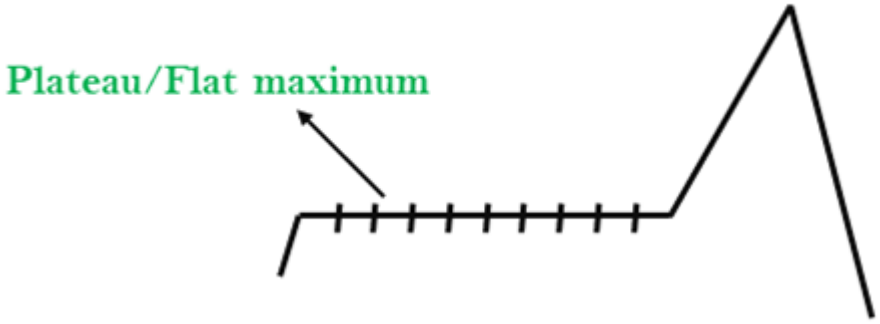
1. Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

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



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

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



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

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

Simulated Annealing:

A hill-climbing algorithm which never makes a move towards a lower value guaranteed to be incomplete because it can get stuck on a local maximum. And if algorithm applies a random walk, by moving a successor, then it may complete but not efficient. **Simulated Annealing** is an algorithm which yields both efficiency and completeness.

In mechanical term **Annealing** is a process of hardening a metal or glass to a high temperature then cooling gradually, so this allows the metal to reach a low-energy crystalline state. The same process is used in simulated annealing in which the algorithm picks a random move, instead of picking the best move. If the random move improves the state, then it follows the same path. Otherwise, the algorithm follows the path which has a probability of less than 1 or it moves downhill and chooses another path.

Adversarial Search

Adversarial search is a search, where we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us.

- In previous topics, we have studied the search strategies which are only associated with a single agent that aims to find the solution which often expressed in the form of a sequence of actions.
- But, there might be some situations where more than one agent is searching for the solution in the same search space, and this situation usually occurs in game playing.
- The environment with more than one agent is termed as **multi-agent environment**, in which each agent is an opponent of other agent and playing against each other. Each agent needs to consider the action of other agent and effect of that action on their performance.
- So, **Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called adversarial searches, often known as Games**.

- Games are modeled as a Search problem and heuristic evaluation function, and these are the two main factors which help to model and solve games in AI.

Types of Games in AI:

	Deterministic	Chance Moves
Perfect information	Chess, Checkers, go, Othello	Backgammon, monopoly
Imperfect information	Battleships, blind, tic-tac-toe	Bridge, poker, scrabble, nuclear war

- **Perfect information:** A game with the perfect information is that in which agents can look into the complete board. Agents have all the information about the game, and they can see each other moves also. Examples are Chess, Checkers, Go, etc.
- **Imperfect information:** If in a game agents do not have all information about the game and not aware with what's going on, such type of games are called the game with imperfect information, such as tic-tac-toe, Battleship, blind, Bridge, etc.
- **Deterministic games:** Deterministic games are those games which follow a strict pattern and set of rules for the games, and there is no randomness associated with them. Examples are chess, Checkers, Go, tic-tac-toe, etc.
- **Non-deterministic games:** Non-deterministic are those games which have various unpredictable events and has a factor of chance or luck. This factor of chance or luck is introduced by either dice or cards. These are random, and each action response is not fixed. Such games are also called as stochastic games.
  Example: Backgammon, Monopoly, Poker, etc.

Note: In this topic, we will discuss deterministic games, fully observable environment, zero-sum, and where each agent acts alternatively.

Zero-Sum Game

- Zero-sum games are adversarial search which involves pure competition.
- In Zero-sum game each agent's gain or loss of utility is exactly balanced by the losses or gains of utility of another agent.
- One player of the game try to maximize one single value, while other player tries to minimize it.
- Each move by one player in the game is called as ply.
- Chess and tic-tac-toe are examples of a Zero-sum game.

Zero-sum game: Embedded thinking

The Zero-sum game involved embedded thinking in which one agent or player is trying to figure out:

- What to do.
- How to decide the move

- Needs to think about his opponent as well
- The opponent also thinks what to do

Each of the players is trying to find out the response of his opponent to their actions. This requires embedded thinking or backward reasoning to solve the game problems in AI.

Formalization of the problem:

A game can be defined as a type of search in AI which can be formalized of the following elements:

- **Initial state:** It specifies how the game is set up at the start.
- **Player(s):** It specifies which player has moved in the state space.
- **Action(s):** It returns the set of legal moves in state space.
- **Result(s, a):** It is the transition model, which specifies the result of moves in the state space.
- **Terminal-Test(s):** Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called terminal states.
- **Utility(s, p):** A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called payoff function. For Chess, the outcomes are a win, loss, or draw and its payoff values are +1, 0, ½. And for tic-tac-toe, utility values are +1, -1, and 0.

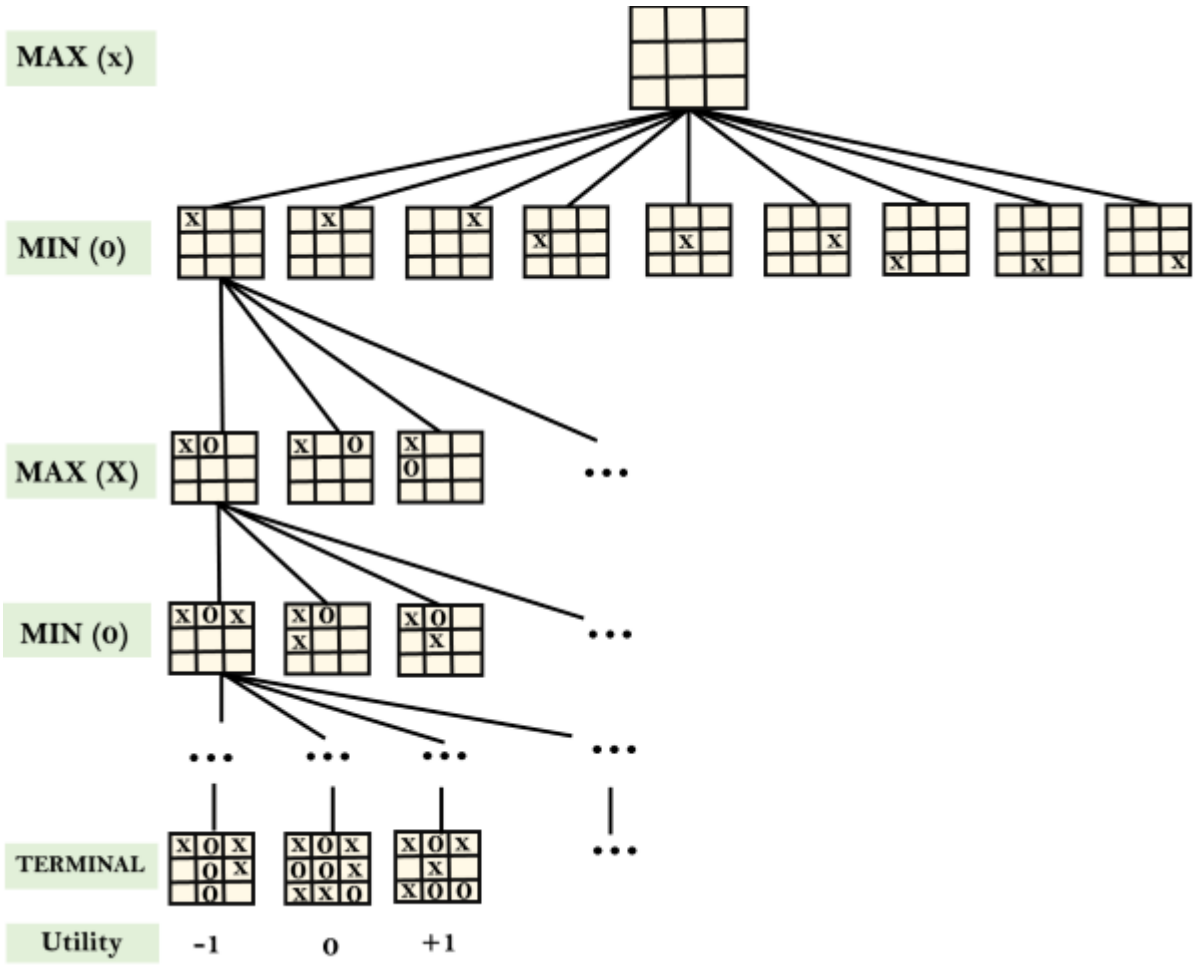
Game tree:

A game tree is a tree where nodes of the tree are the game states and Edges of the tree are the moves by players. Game tree involves initial state, actions function, and result Function.

Example: Tic-Tac-Toe game tree:

The following figure is showing part of the game-tree for tic-tac-toe game. Following are some key points of the game:

- There are two players MAX and MIN.
- Players have an alternate turn and start with MAX.
- MAX maximizes the result of the game tree
- MIN minimizes the result.



Example Explanation:

- From the initial state, MAX has 9 possible moves as he starts first. MAX place x and MIN place o, and both player plays alternatively until we reach a leaf node where one player has three in a row or all squares are filled.
- Both players will compute each node, minimax, the minimax value which is the best achievable utility against an optimal adversary.
- Suppose both the players are well aware of the tic-tac-toe and playing the best play. Each player is doing his best to prevent another one from winning. MIN is acting against Max in the game.
- So in the game tree, we have a layer of Max, a layer of MIN, and each layer is called as **Ply**. Max place x, then MIN puts o to prevent Max from winning, and this game continues until the terminal node.
- In this either MIN wins, MAX wins, or it's a draw. This game-tree is the whole search space of possibilities that MIN and MAX are playing tic-tac-toe and taking turns alternately.

Hence adversarial Search for the minimax procedure works as follows:

- It aims to find the optimal strategy for MAX to win the game.
- It follows the approach of Depth-first search.
- In the game tree, optimal leaf node could appear at any depth of the tree.
- Propagate the minimax values up to the tree until the terminal node discovered.

In a given game tree, the optimal strategy can be determined from the minimax value of each node, which can be written as MINIMAX(n). MAX prefer to move to a state of maximum value and MIN prefer to move to a state of minimum value then:

For a state S MINIMAX(s) =

$$\begin{cases} \text{UTILITY}(s) & \text{If TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MIN}. \end{cases}$$

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
4.
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
11.
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
```

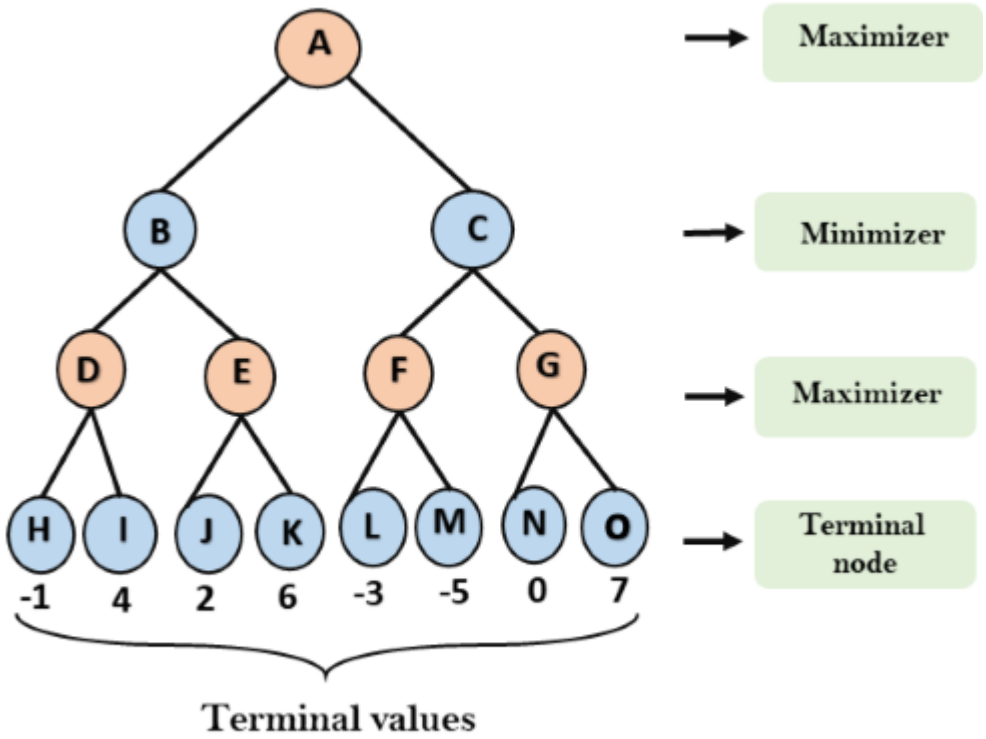
Initial call:

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:

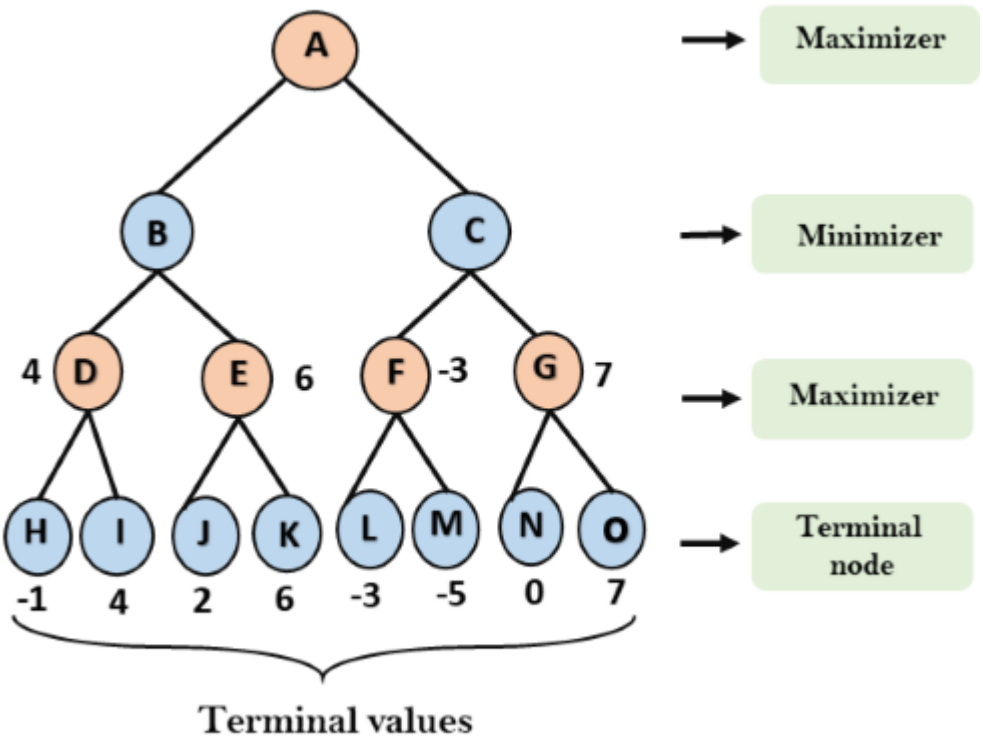
Step-1: In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =-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

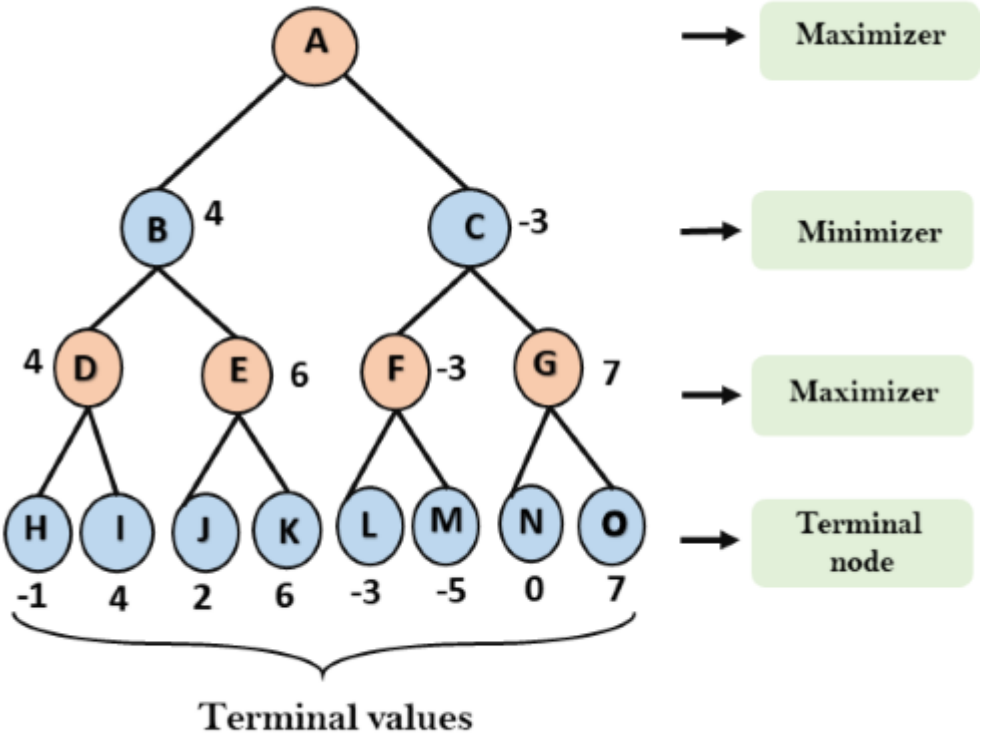
- For node D max(-1,- -∞) => max(-1,4)= 4

- 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



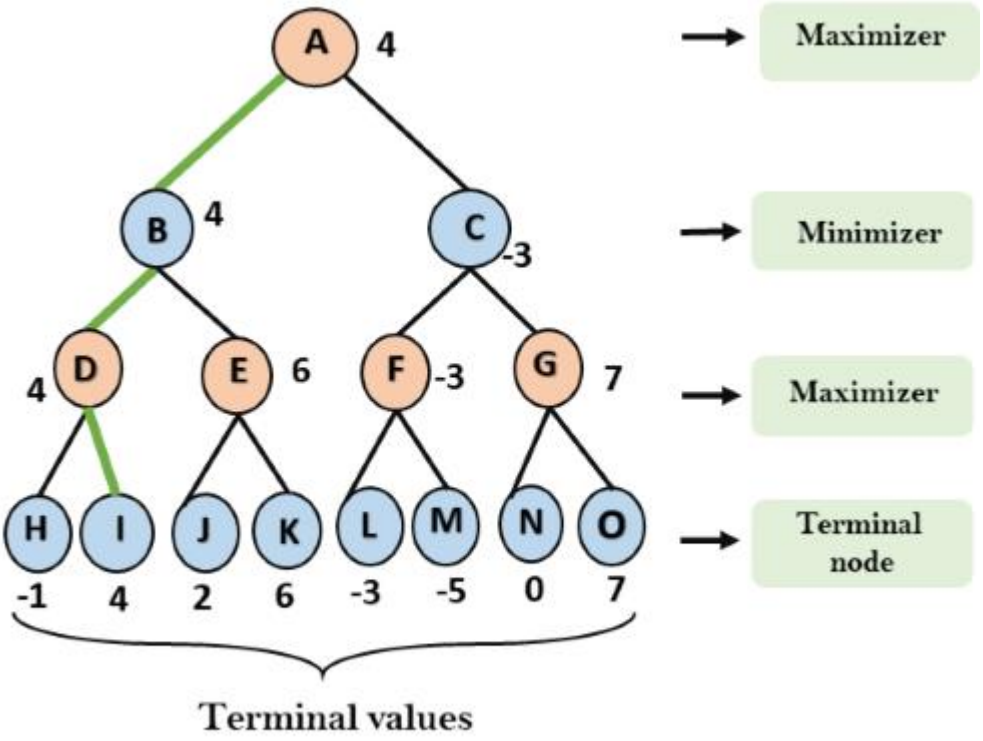
Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

- 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 **$O(b^m)$**, 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. This limitation of the minimax algorithm can be improved from **alpha-beta pruning** which we have discussed in the next topic.

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.

Note: To better understand this topic, kindly study the minimax algorithm.

Condition for Alpha-beta pruning:

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

1. 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.

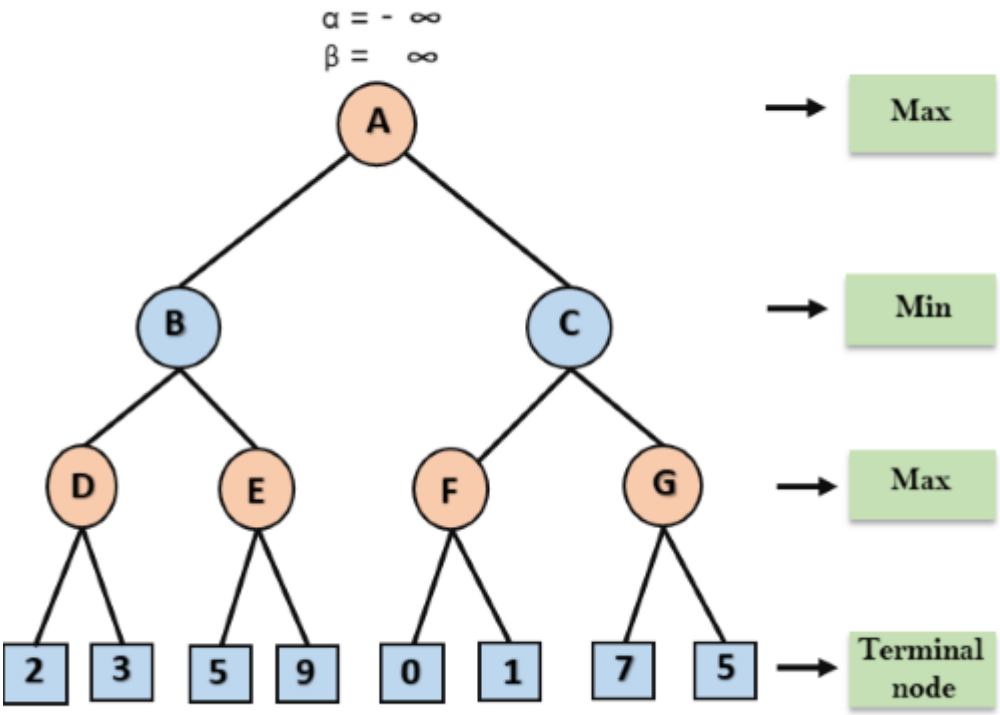
Pseudo-code for Alpha-beta Pruning:

```
1. function minimax(node, depth, alpha, beta, maximizingPlayer) is
2. if depth ==0 or node is a terminal node then
3. return static evaluation of node
4.
5. if MaximizingPlayer then      // for Maximizer Player
6.    maxEva= -infinity
7.    for each child of node do
8.    eva= minimax(child, depth-1, alpha, beta, False)
9.   maxEva= max(maxEva, eva)
10.  alpha= max(alpha, maxEva)
11.   if beta<=alpha
12.  break
13.  return maxEva
14.
15. else                    // for Minimizer player
16.   minEva= +infinity
17.   for each child of node do
18.   eva= minimax(child, depth-1, alpha, beta, true)
19.   minEva= min(minEva, eva)
20.   beta= min(beta, eva)
21.    if beta<=alpha
22.   break
23.  return minEva
```

Working of Alpha-Beta Pruning:

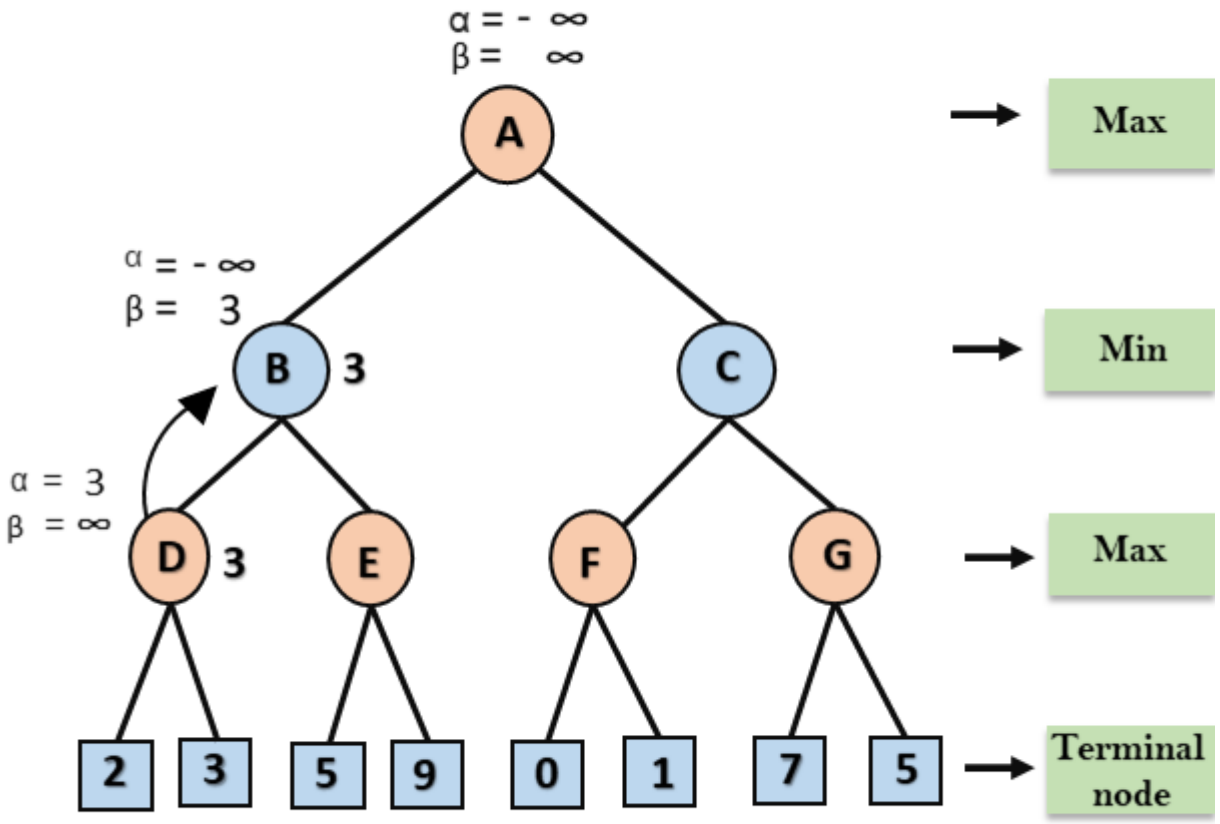
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

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.



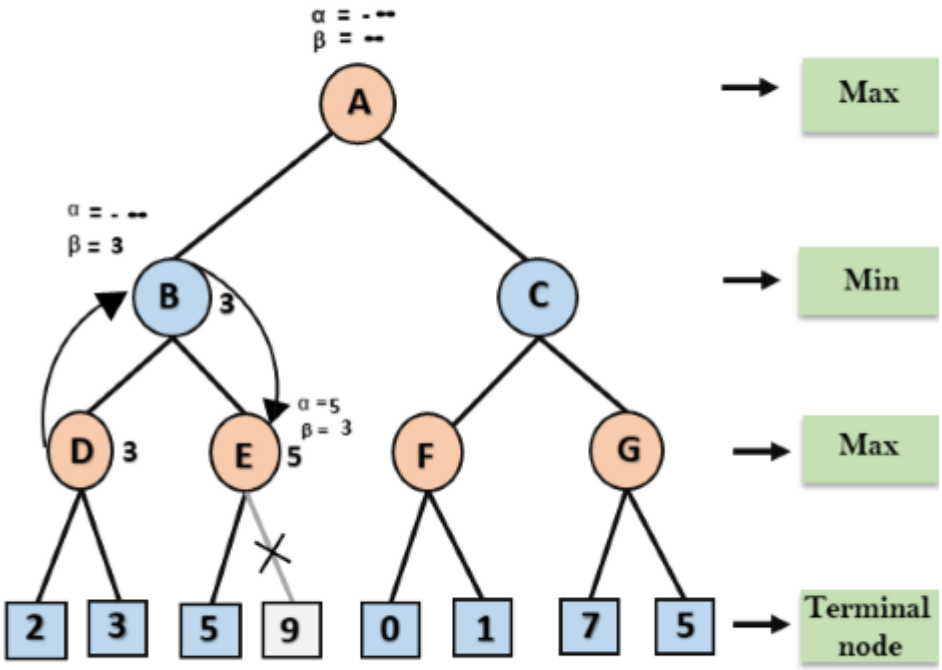
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

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.

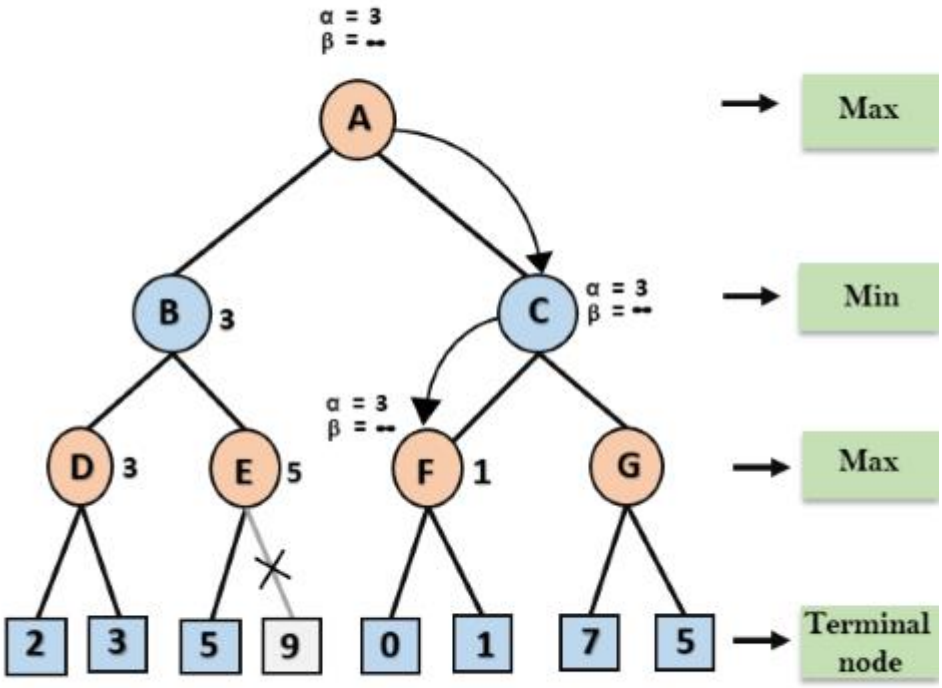
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will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



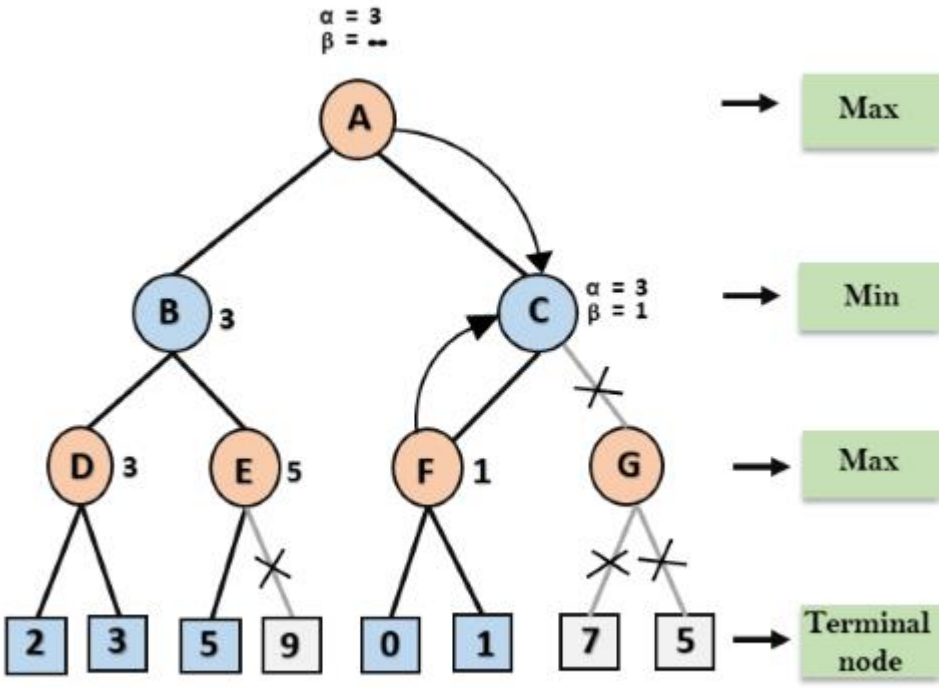
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

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

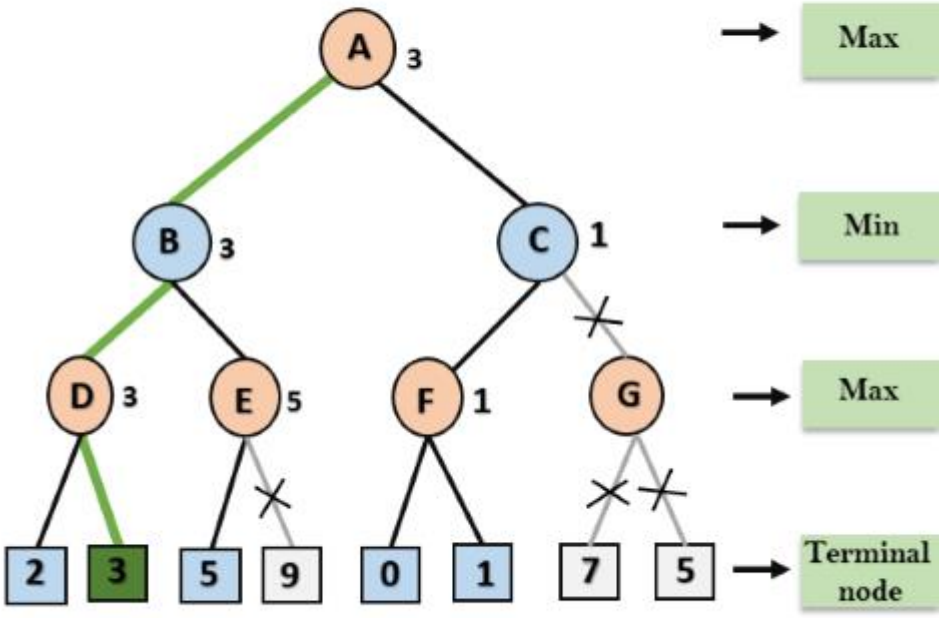
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



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 $O(b^m)$.
- **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 $O(b^{m/2})$.

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.

Map Colouring Algorithm

Map colouring problem states that given a graph G {V, E} where V and E are the set of vertices and edges of the graph, all vertices in in V need to be coloured in such a way that no two adjacent vertices must have the same colour.

The real-world applications of this algorithm are – assigning mobile radio frequencies, making schedules, designing Sudoku, allocating registers etc.

Map Colouring Algorithm

With the map colouring algorithm, a graph G and the colours to be added to the graph are taken as an input and a coloured graph with no two adjacent vertices having the same colour is achieved.

Algorithm

- Initiate all the vertices in the graph.
- Select the node with the highest degree to colour it with any colour.
- Choose the colour to be used on the graph with the help of the selection colour function so that no adjacent vertex is having the same colour.
- Check if the colour can be added and if it does, add it to the solution set.
- Repeat the process from step 2 until the output set is ready.

Examples

Step 1

Find degrees of all the vertices –

A – 4

B – 2

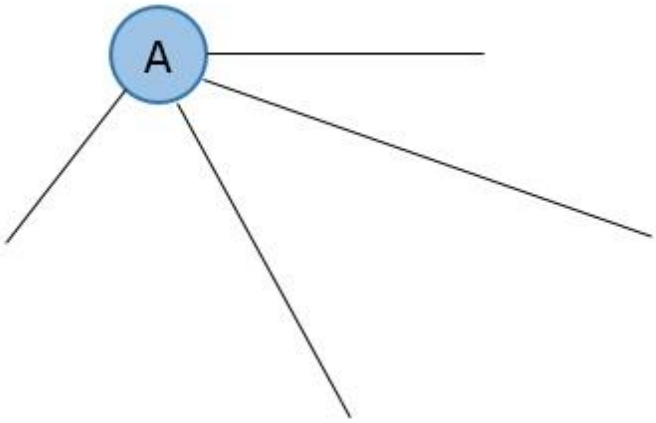
C – 2

D – 3

E – 3

Step 2

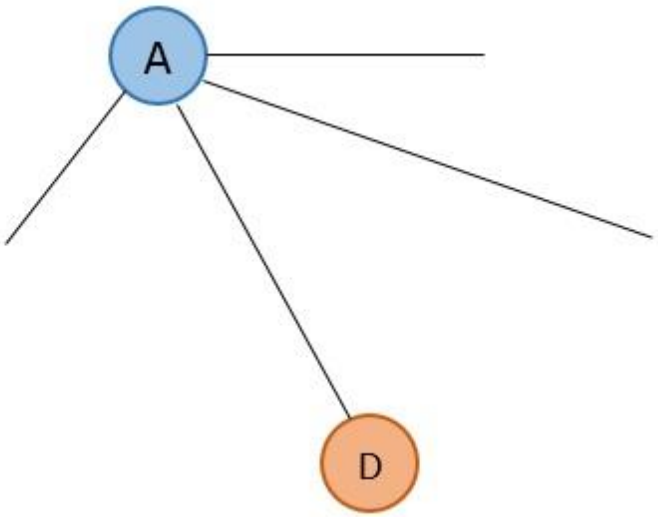
Choose the vertex with the highest degree to colour first, i.e., A and choose a colour using selection colour function. Check if the colour can be added to the vertex and if yes, add it to the solution set.



Step 3

Select any vertex with the next highest degree from the remaining vertices and colour it using selection colour function.

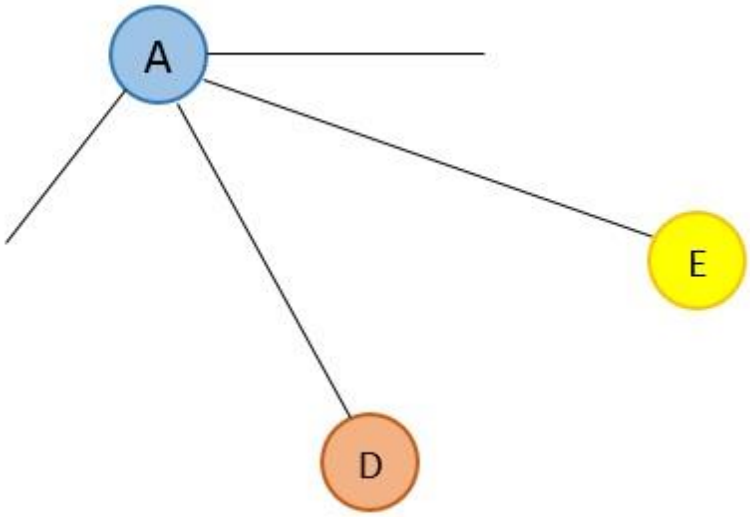
D and E both have the next highest degree 3, so choose any one between them, say D.



D is adjacent to A, therefore it cannot be coloured in the same colour as A. Hence, choose a different colour using selection colour function.

Step 4

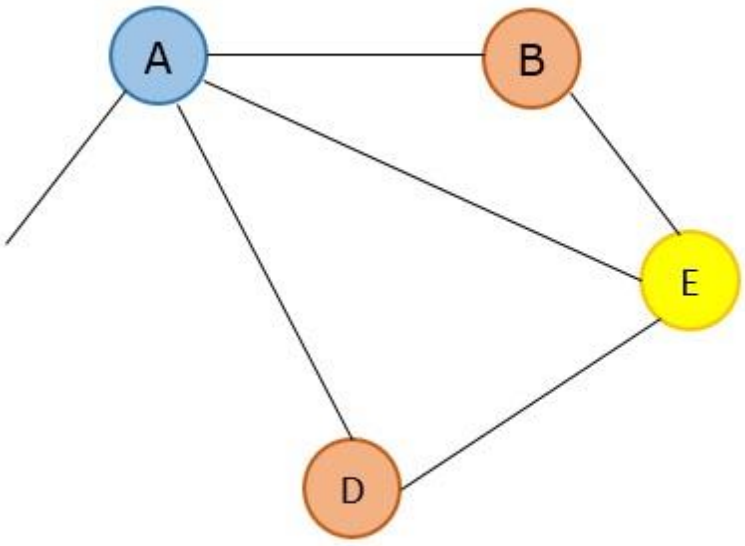
The next highest degree vertex is E, hence choose E.



E is adjacent to both A and D, therefore it cannot be coloured in the same colours as A and D. Choose a different colour using selection colour function.

Step 5

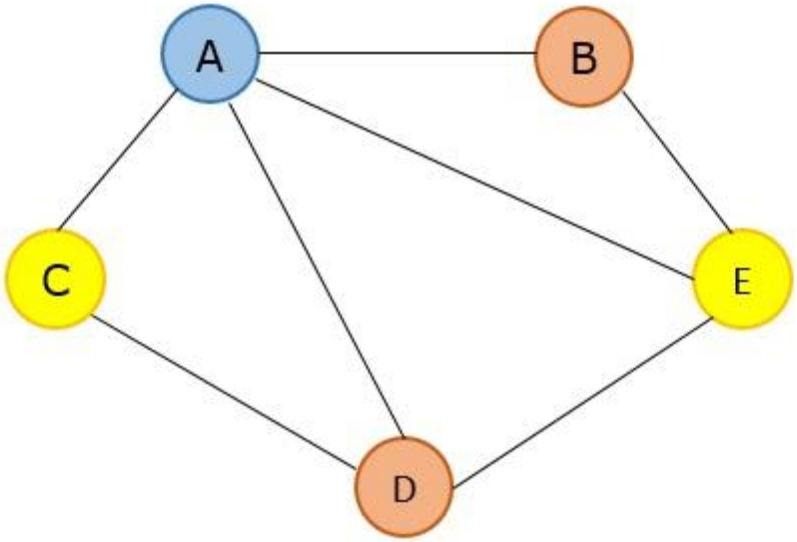
The next highest degree vertices are B and C. Thus, choose any one randomly.



B is adjacent to both A and E, thus not allowing to be coloured in the colours of A and E but it is not adjacent to D, so it can be coloured with D's colour.

Step 6

The next and the last vertex remaining is C, which is adjacent to both A and D, not allowing it to be coloured using the colours of A and D. But it is not adjacent to E, so it can be coloured in E's colour.



Constraint Satisfaction Problems (CSP) in Artificial Intelligence

Finding a solution that meets a set of constraints is the goal of constraint satisfaction problems (CSPs), a type of AI issue. Finding values for a group of variables that fulfill a set of restrictions or rules is the aim of constraint satisfaction problems. For tasks including resource allocation, planning, scheduling, and decision-making, CSPs are frequently employed in AI.

There are mainly three basic components in the constraint satisfaction problem:

Variables: The things that need to be determined are variables. Variables in a CSP are the objects that must have values assigned to them in order to satisfy a particular set of constraints. Boolean, integer, and categorical variables are just a few examples of the various types of variables Variables, for instance, could stand in for the many puzzle cells that need to be filled with numbers in a sudoku puzzle.

Domains: The range of potential values that a variable can have is represented by domains. Depending on the issue, a domain may be finite or limitless. For instance, in Sudoku, the set of numbers from 1 to 9 can serve as the domain of a variable representing a problem cell.

Constraints: The guidelines that control how variables relate to one another are known as constraints. Constraints in a CSP define the ranges of possible values for variables. Unary constraints, binary constraints, and higher-order constraints are only a few examples of the various sorts of constraints. For instance, in a sudoku problem, the restrictions might be that each row, column, and 3×3 box can only have one instance of each number from 1 to 9.

Constraint Satisfaction Problems (CSP) representation:

- The finite set of variables V_1, V_2, V_3 ……..V_n.
- Non-empty domain for every single variable D_1, D_2, D_3 ……..D_n.
- The finite set of constraints C_1, C_2 ……., Cm.
    - where each constraint C_i restricts the possible values for variables,
        - e.g., $V_1 \neq V_2$

- Each constraint C_i is a pair <scope, relation>
    - Example: <(V_1, V_2), V_1 not equal to V_2>
- Scope = set of variables that participate in constraint.
- Relation = list of valid variable value combinations.
    - There might be a clear list of permitted combinations. Perhaps a relation that is abstract and that allows for membership testing and listing.

Constraint Satisfaction Problems (CSP) algorithms:

- The **backtracking algorithm** is a depth-first search algorithm that methodically investigates the search space of potential solutions up until a solution is discovered that satisfies all the restrictions. The method begins by choosing a variable and giving it a value before repeatedly attempting to give values to the other variables. The method returns to the prior variable and tries a different value if at any time a variable cannot be given a value that fulfills the requirements. Once all assignments have been tried or a solution that satisfies all constraints has been discovered, the algorithm ends.
- The **forward-checking algorithm** is a variation of the backtracking algorithm that condenses the search space using a type of local consistency. For each unassigned variable, the method keeps a list of remaining values and applies local constraints to eliminate inconsistent values from these sets. The algorithm examines a variable's neighbors after it is given a value to see whether any of its remaining values become inconsistent and removes them from the sets if they do. The algorithm goes backward if, after forward checking, a variable has no more values.
- Algorithms for **propagating constraints** are a class that uses local consistency and inference to condense the search space. These algorithms operate by propagating restrictions between variables and removing inconsistent values from the variable domains using the information obtained.

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:

a. **Atomic Propositions**

b. **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:

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

Truth Table:

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

For Negation:

P	¬ P
True	**False**
False	**True**

For Conjunction:

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

For disjunction:

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

For Implication:

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

For Biconditional:

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

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.

P	Q	R	¬R	P∨ Q	P∨Q→¬R
True	True	True	False	True	False
True	True	False	True	True	True
True	False	True	False	True	False
True	False	False	True	True	True
False	True	True	False	True	False
False	True	False	True	True	True
False	False	True	False	False	True
False	False	False	True	False	True

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

Note: For better understanding use parenthesis to make sure of the correct interpretations. Such as ¬R∨Q, It can be interpreted as (¬R) ∨Q.

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

A	B	¬A	¬A∨ B	A→B
T	T	F	**T**	**T**
T	F	F	**F**	**F**
F	T	T	**T**	**T**
F	F	T	**T**	**T**

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.

First-Order Logic in Artificial intelligence

In the topic of Propositional logic, we have seen that how to represent statements using propositional logic. But unfortunately, in propositional logic, we can only represent the facts, which are either true or false. PL is not sufficient to represent the complex sentences or natural language statements. The propositional logic has very limited expressive power. Consider the following sentence, which we cannot represent using PL logic.

- **"Some humans are intelligent", or**
- **"Sachin likes cricket."**

To represent the above statements, PL logic is not sufficient, so we required some more powerful logic, such as first-order logic.

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 develops 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**
  - **Semantics**

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).

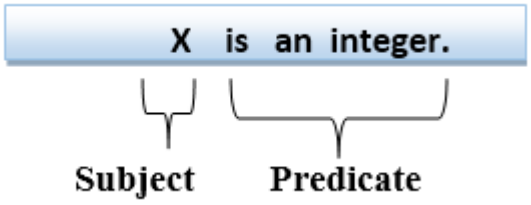
Complex Sentences:

- 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.



Quantifiers in First-order logic:

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

1. **Universal Quantifier, (for all, everyone, everything)**
2. **Existential quantifier, (for some, at least one).**

Universal Quantifier:

Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing.

The Universal quantifier is represented by a symbol ∀, which resembles an inverted A.

Note: In universal quantifier we use implication "→".

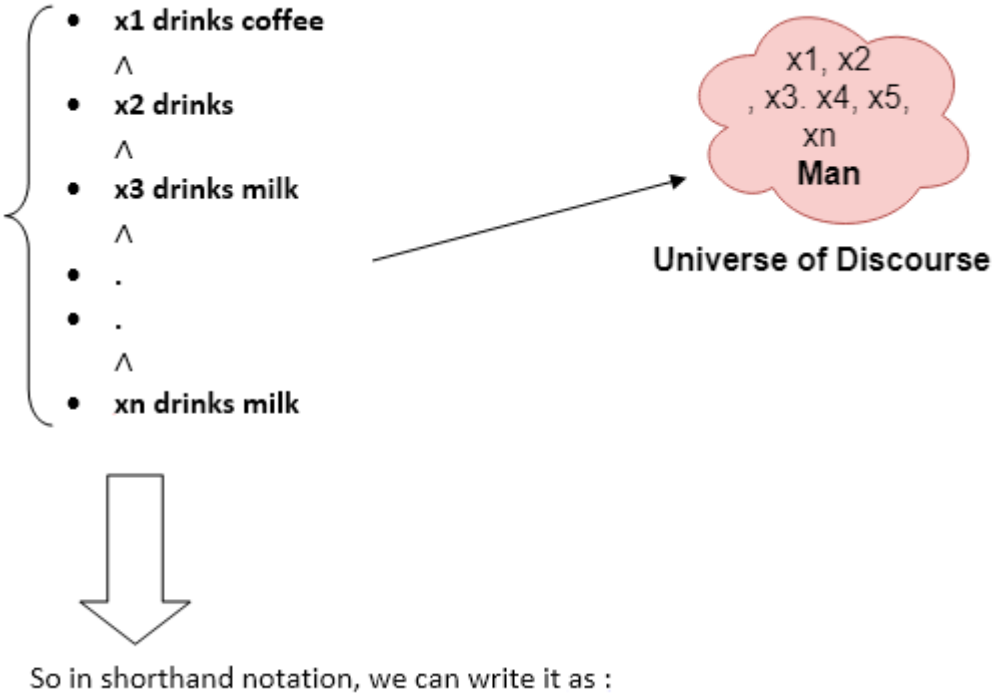
If x is a variable, then ∀x is read as:

- **For all x**
- **For each x**
- **For every x.**

Example:

All man drink coffee.

Let a variable x which refers to a cat so all x can be represented in UOD as below:



∀x man(x) → drink (x, coffee).

It will be read as: There are all x where x is a man who drink coffee.

Existential Quantifier:

Existential quantifiers are the type of quantifiers, which express that the statement within its scope is true for at least one instance of something.

It is denoted by the logical operator ∃, which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.

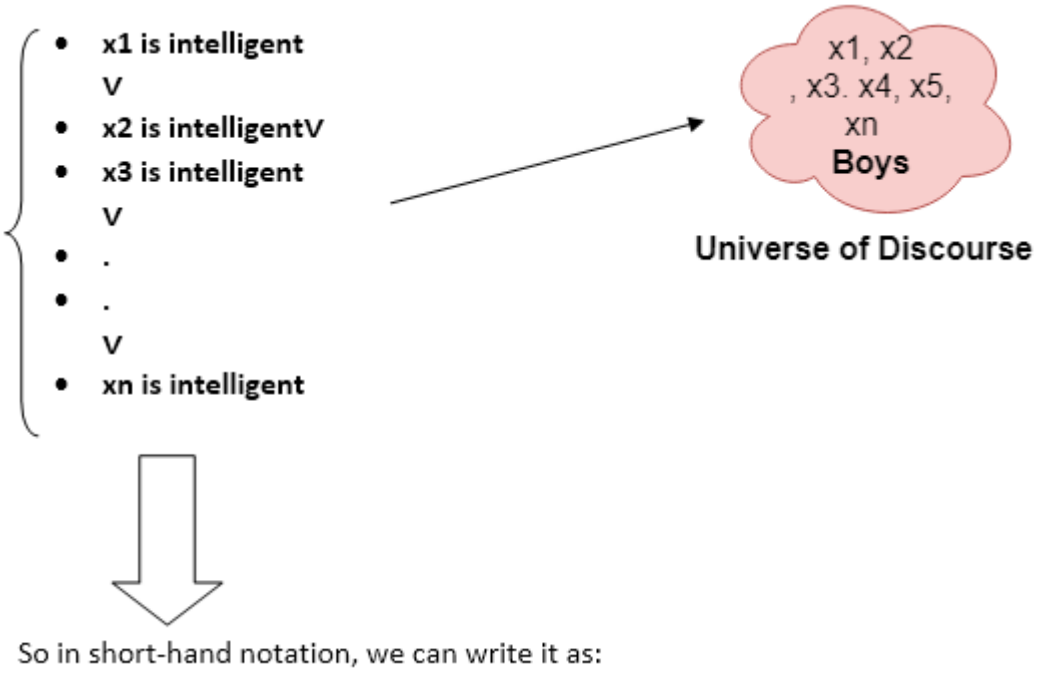
Note: In Existential quantifier we always use AND or Conjunction symbol (∧).

If x is a variable, then existential quantifier will be ∃x or ∃(x). And it will be read as:

- **There exists a 'x.'**
- **For some 'x.'**
- **For at least one 'x.'**

Example:

Some boys are intelligent.



∃x: boys(x) ∧ intelligent(x)

It will be read as: There are some x where x is a boy who is intelligent.

Points to remember:

- The main connective for universal quantifier ∀ is implication →.
- The main connective for existential quantifier ∃ is and ∧.

Properties of Quantifiers:

- In universal quantifier, ∀x∀y is similar to ∀y∀x.
- In Existential quantifier, ∃x∃y is similar to ∃y∃x.
- ∃x∀y is not similar to ∀y∃x.

Some Examples of FOL using quantifier:

1. All birds fly.
In this question the predicate is "**fly(bird)**."
And since there are all birds who fly so it will be represented as follows.
∀x bird(x) →fly(x).

2. Every man respects his parent.
In this question, the predicate is "**respect(x, y),**" **where x=man, and y= parent**.
Since there is every man so will use ∀, and it will be represented as follows:
∀x man(x) → respects (x, parent).

3. Some boys play cricket.
In this question, the predicate is "**play(x, y)**," where x= boys, and y= game. Since there are some boys so we will use **∃, and it will be represented as**:
∃x boys(x) → play(x, cricket).

4. Not all students like both Mathematics and Science.
In this question, the predicate is "**like(x, y),**" **where x= student, and y= subject**.

Since there are not all students, so we will use **∀ with negation, so** following representation for this:

¬∀ (x) [student(x) → like(x, Mathematics) ∧ like(x, Science)].

5. Only one student failed in Mathematics.
In this question, the predicate is "**failed(x, y),**" **where x= student, and y= subject**. Since there is only one student who failed in Mathematics, so we will use following representation for this:

∃(x) [student(x) → failed (x, Mathematics) ∧∀ (y) [¬(x==y) ∧ student(y) → ¬failed (x, Mathematics)].

Free and Bound Variables:

The quantifiers interact with variables which appear in a suitable way. There are two types of variables in First-order logic which are given below:

Free Variable: A variable is said to be a free variable in a formula if it occurs outside the scope of the quantifier.

Example: ∀x ∃(y)[P (x, y, z)], where z is a free variable.

Bound Variable: A variable is said to be a bound variable in a formula if it occurs within the scope of the quantifier.

Example: ∀x [A (x) B(y)], here x and y are the bound variables.

Inference in First-Order Logic

Inference in First-Order Logic is used to deduce new facts or sentences from existing sentences. Before understanding the FOL inference rule, let's understand some basic terminologies used in FOL.

Substitution:

Substitution is a fundamental operation performed on terms and formulas. It occurs in all inference systems in first-order logic. The substitution is complex in the presence of quantifiers in FOL. If we write **F[a/x]**, so it refers to substitute a constant "**a**" in place of variable "**x**".

Note: First-order logic is capable of expressing facts about some or all objects in the universe.

Equality:

Backward Skip 10sPlay VideoForward Skip 10s

First-Order logic does not only use predicate and terms for making atomic sentences but also uses another way, which is equality in FOL. For this, we can use **equality symbols** which specify that the two terms refer to the same object.

Example: Brother (John) = Smith.

As in the above example, the object referred by the **Brother (John)** is similar to the object referred by **Smith**. The equality symbol can also be used with negation to represent that two terms are not the same objects.

Example: ¬(x=y) which is equivalent to x ≠y.

FOL inference rules for quantifier:

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

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

1. Universal Generalization:

- Universal generalization is a valid inference rule which states that if premise P(c) is true for any arbitrary element c in the universe of discourse, then we can have a conclusion as ∀ x P(x).
- It can be represented as: $\frac{P(c)}{\forall x\, P(x)}$.
- This rule can be used if we want to show that every element has a similar property.
- In this rule, x must not appear as a free variable.

Example: Let's represent, P(c): "**A byte contains 8 bits**", so for **∀ x P(x)** "**All bytes contain 8 bits**.", it will also be true.

2. Universal Instantiation:

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

Example:1.

IF "Every person like ice-cream"=> ∀x P(x) so we can infer that "John likes ice-cream" => P(c)

Example: 2.

Let's take a famous example,

"All kings who are greedy are Evil." So let our knowledge base contains this detail as in the form of FOL:

∀x king(x) ∧ greedy (x) → Evil (x),

So from this information, we can infer any of the following statements using Universal Instantiation:

- **King(John) ∧ Greedy (John) → Evil (John),**
- **King(Richard) ∧ Greedy (Richard) → Evil (Richard),**
- **King(Father(John)) ∧ Greedy (Father(John)) → Evil (Father(John)),**

3. Existential Instantiation:

- Existential instantiation is also called as Existential Elimination, which is a valid inference rule in first-order logic.
- It can be applied only once to replace the existential sentence.
- The new KB is not logically equivalent to old KB, but it will be satisfiable if old KB was satisfiable.
- This rule states that one can infer P(c) from the formula given in the form of ∃x P(x) for a new constant symbol c.
- The restriction with this rule is that c used in the rule must be a new term for which P(c) is true.
- It can be represented as: $\frac{\exists x\, P(x)}{P(c)}$

Example:

From the given sentence: **∃x Crown(x) ∧ OnHead(x, John),**

So we can infer: **Crown(K) ∧ OnHead(K, John),** as long as K does not appear in the knowledge base.

- The above used K is a constant symbol, which is called **Skolem constant**.
- The Existential instantiation is a special case of **Skolemization process**.

4. Existential introduction

- An existential introduction is also known as an existential generalization, which is a valid inference rule in first-order logic.
- This rule states that if there is some element c in the universe of discourse which has a property P, then we can infer that there exists something in the universe which has the property P.
- It can be represented as: $\frac{P(c)}{\exists x P(x)}$
- **Example: Let's say that,** "Priyanka got good marks in English." "Therefore, someone got good marks in English."

Generalized Modus Ponens Rule:

For the inference process in FOL, we have a single inference rule which is called Generalized Modus Ponens. It is lifted version of Modus ponens.

Generalized Modus Ponens can be summarized as, " P implies Q and P is asserted to be true, therefore Q must be True."

According to Modus Ponens, for atomic sentences **pi, pi', q**. Where there is a substitution θ such that SUBST **(θ, pi',) = SUBST(θ, pi)**, it can be represented as:

$$\frac{p1', p2', \ldots, pn', (p1 \wedge p2 \wedge \ldots \wedge pn \Rightarrow q)}{SUBST(\theta, q)}$$

Example:

We will use this rule for Kings are evil, so we will find some x such that x is king, and x is greedy so we can infer that x is evil.

1. Here let say, p1' is king(John) p1 is king(x)
2. p2' is Greedy(y) p2 is Greedy(x)
3. θ is {x/John, y/John} q is evil(x)
4. SUBST(θ,q).

Difference between Propositional Logic and Predicate Logic

Logical reasoning forms the basis for a huge domain of computer science and mathematics. They help in establishing mathematical arguments, valid or invalid.

1. Propositional Logic :
A **proposition** is basically a declarative sentence that has a truth value. Truth value can either be true or false, but it needs to be assigned any of the two values and not be ambiguous. The purpose of using propositional logic is to analyze a statement, individually or compositely.
For example :
The following statements :

1. If x is real, then $x^2 > 0$
2. What is your name?
3. $(a+b)^2 = 100$
4. This statement is false.
5. This statement is true.

Are not propositions because they do not have a truth value. They are ambiguous.

But the following statements :

1. $(a+b)^2 = a^2 + 2ab + b^2$
2. If x is real, then $x^2 >= 0$
3. If x is real, then $x^2 < 0$
4. The sun rises in the east.
5. The sun rises in the west.

Are all propositions because they have a specific truth value, true or false.

The branch of logic that deals with proposition is **propositional logic**.

2. Predicate Logic :

Predicates are properties, additional information to better express the subject of the sentence. A quantified predicate is a proposition , that is, when you assign values to a predicate with variables it can be made a proposition.

For example :

In P(x) : x>5, x is the subject or the variable and '>5' is the predicate.

P(7) : 7>5 is a proposition where we are assigning values to the variable x, and it has a truth value, i.e. True.

The set of values that the variables of the predicate can assume is called the Universe or Domain of Discourse or Domain of Predicate.

Difference between Propositional Logic and Predicate Logic :

	Propositional Logic	**Predicate Logic**
1	Propositional logic is the logic that deals with a collection of declarative statements which have a truth value, true or false.	Predicate logic is an expression consisting of variables with a specified domain. It consists of objects, relations and functions between the objects.
2	It is the basic and most widely used logic. Also known as Boolean logic.	It is an extension of propositional logic covering predicates and quantification.
3	A proposition has a specific truth value, either true or false.	A predicate's truth value depends on the variables' value.
4	Scope analysis is not done in propositional logic.	Predicate logic helps analyze the scope of the subject over the predicate. There are three quantifiers : Universal Quantifier (∀) depicts for all, Existential Quantifier (∃) depicting there exists some and Uniqueness Quantifier (∃!) depicting exactly one.
5	Propositions are combined with Logical Operators or Logical Connectives like Negation(¬), Disjunction(∨), Conjunction(∧), Exclusive OR(⊕),	Predicate Logic adds by introducing quantifiers to the existing proposition.

	Implication(⇒), Bi-Conditional or Double Implication(⇔).	
6	It is a more generalized representation.	It is a more specialized representation.
7	It cannot deal with sets of entities.	It can deal with set of entities with the help of quantifiers.

Bayes' theorem in Artificial intelligence

Bayes' theorem:

Bayes' theorem is also known as **Bayes' rule, Bayes' law**, or **Bayesian reasoning**, which determines the probability of an event with uncertain knowledge.

In probability theory, it relates the conditional probability and marginal probabilities of two random events.

Bayes' theorem was named after the British mathematician **Thomas Bayes**. The **Bayesian inference** is an application of Bayes' theorem, which is fundamental to Bayesian statistics.

It is a way to calculate the value of P(B|A) with the knowledge of P(A|B).

Bayes' theorem allows updating the probability prediction of an event by observing new information of the real world.

Example: If cancer corresponds to one's age, then by using Bayes' theorem, we can determine the probability of cancer more accurately with the help of age.

Bayes' theorem can be derived using product rule and conditional probability of event A with known event B:

As from product rule we can write:

1. P(A ∧ B)= P(A|B) P(B) or

Similarly, the probability of event B with known event A:

1. P(A ∧ B)= P(B|A) P(A)

Equating right hand side of both the equations, we will get:

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)} \quad \text{....(a)}$$

The above equation (a) is called as **Bayes' rule** or **Bayes' theorem**. This equation is basic of most modern AI systems for **probabilistic inference**.

It shows the simple relationship between joint and conditional probabilities. Here,

P(A|B) is known as **posterior**, which we need to calculate, and it will be read as Probability of hypothesis A when we have occurred an evidence B.

P(B|A) is called the likelihood, in which we consider that hypothesis is true, then we calculate the probability of evidence.

P(A) is called the **prior probability**, probability of hypothesis before considering the evidence

P(B) is called **marginal probability**, pure probability of an evidence.

In the equation (a), in general, we can write P (B) = P(A)*P(B|Ai), hence the Bayes' rule can be written as:

$$P(A_i|B) = \frac{P(A_i)*P(B|A_i)}{\sum_{i=1}^{k} P(A_i)*P(B|A_i)}$$

Where A_1, A_2, A_3,....., A_n is a set of mutually exclusive and exhaustive events.

Applying Bayes' rule:

Bayes' rule allows us to compute the single term P(B|A) in terms of P(A|B), P(*B*), and P(A). This is very useful in cases where we have a good probability of these three terms and want to determine the fourth one. Suppose we want to perceive the effect of some unknown cause, and want to compute that cause, then the Bayes' rule becomes:

$$\mathbf{P(cause|effect) = \frac{P(effect|cause)\,P(cause)}{P(effect)}}$$

Example-1:

Question: what is the probability that a patient has diseases meningitis with a stiff neck?

Given Data:

A doctor is aware that disease meningitis causes a patient to have a stiff neck, and it occurs 80% of the time. He is also aware of some more facts, which are given as follows:

- The Known probability that a patient has meningitis disease is 1/30,000.
- The Known probability that a patient has a stiff neck is 2%.

Let a be the proposition that patient has stiff neck and b be the proposition that patient has meningitis. , so we can calculate the following as:

P(a|b) = 0.8

P(b) = 1/30000

P(a)= .02

$$\mathbf{P(b|a)} = \frac{P(a|b)P(b)}{P(a)} = \frac{0.8*(\frac{1}{30000})}{0.02} = 0.001333333.$$

Hence, we can assume that 1 patient out of 750 patients has meningitis disease with a stiff neck.

Example-2:

Question: From a standard deck of playing cards, a single card is drawn. The probability that the card is king is 4/52, then calculate posterior probability P(King|Face), which means the drawn face card is a king card.

Solution:

$$\mathbf{P(king|face)} = \frac{P(Face|king)*P(King)}{P(Face)} \quad .....(i)$$

P(king): probability that the card is King= 4/52= 1/13

P(face): probability that a card is a face card= 3/13

P(Face|King): probability of face card when we assume it is a king = 1

Putting all values in equation (i) we will get:

$$P(king|face) = \frac{1*(\frac{1}{13})}{(\frac{3}{13})} = \mathbf{1/3,}$$ **it is a probability that a face card is a king card.**

Application of Bayes' theorem in Artificial intelligence:

Following are some applications of Bayes' theorem:

- It is used to calculate the next step of the robot when the already executed step is given.
- Bayes' theorem is helpful in weather forecasting.
- It can solve the Monty Hall problem.

Bayesian networks in AI: Understanding with an example

Let’s consider a simple example to understand Bayesian networks in AI.
Suppose we want to model the relationships between these three variables: “Smoking,” “Lung cancer,” and “Cough.” And say, we are interested in understanding how smoking and lung cancer influence the likelihood of experiencing a cough. We can construct a Bayesian network to capture these relationships.
Variable definition:

- **Smoking:** This variable represents whether a person is a smoker or a non-smoker and can take values “Yes” or “No.”
- **Lung cancer:** This variable represents the presence or absence of lung cancer and can take values “Yes” or “No.”
- **Cough:** This variable represents whether a person experiences a cough and can take values “Yes” or “No.”

Graphical representation: We construct a Bayesian network by connecting the variables with directed edges, indicating the causal relationships or dependencies.

In this network, “Smoking” is the parent node for both “Lung Cancer” and “Cough.” “Lung Cancer” is the parent node for “Cough.” The graph captures the relationships between these variables.

Conditional Probability Tables (CPTs): Each node in the Bayesian network is associated with a conditional probability table that specifies the probability distribution of the node with respect to its parent nodes.

- **Smoking:** The CPT for “Smoking” might look like this:

Smoking	Probability
Yes	0.3
No	0.7

- **Lung cancer:** The CPT for “Lung Cancer” might look like this:

Smoking	Lung Cancer = yes	Lung Cancer = No
Yes	0.85	0.15
No	0.01	0.19

- **Cough:** The CPT for “Cough” might look like this:

Smoking	Cough = yes	Cough = No
Yes	0.8	0.2
No	0.1	0.9

The values in the CPTs represent the probabilities of different outcomes based on the given states of the parent nodes.

Probabilistic inference

Probabilistic inference refers to the process of reasoning and making predictions based on observed or available information using the principles of probability theory. It involves updating our beliefs or knowledge about uncertain events or variables based on new evidence or data.

Probabilistic inference in Bayesian networks refers to the process of reasoning and making predictions about the probability distributions of unobserved variables given observed evidence or data. It utilizes the graphical structure and probabilistic dependencies encoded in the Bayesian network to perform inference.
In a Bayesian network, the joint probability distribution of all variables can be factorized using the chain rule of probability and the conditional independence assumptions represented by the network's structure. This factorization allows for efficient probabilistic inference.
Probabilistic inference in Bayesian networks involves two main tasks:

1. **Querying:** When provided with observed evidence, the objective is to determine the probability distribution of one or more target variables of interest. This involves conditioning the observed evidence and propagating the probabilities throughout the network to obtain the desired probability distribution.
2. **Learning:** This task involves updating the probabilities or parameters in the Bayesian network based on observed data. It aims to refine the network's structure and probability tables to reflect the data better.

How to calculate probabilistic inference using Python?

This example of solving the Monty Hall problem, a famous probability puzzle, will help us understand how to calculate probabilistic inference.

The Monty Hall problem:

In this problem, a participant can choose one out of three doors. Behind one door is a valuable prize, while the other two doors hide goats. After the participant chooses a door, the host (Monty), who knows what's behind each door, opens one of the remaining doors to disclose a goat. The participant is then given the option to either stick with their original choice or switch to the other unopened door. The question is: What is the best strategy for the participant to maximize their chances of winning the prize?

The solution using a Bayesian network:

We can model the Monty Hall problem using a Bayesian network with three nodes representing the participant's initial choice (C), the location of the prize (P), and the door opened by the host (H).
To solve the problem, we need to calculate the probabilities of the hidden variables given the observed variables. We will calculate the probability of winning the prize when sticking with the initial choice and when switching to the other unopened door.
First, import the required dependencies:

```
import numpy as np

from pgmpy.models import BayesianModel

from pgmpy.factors.discrete import TabularCPD

import networkx as nx

import pylab as plt
```

Next, create the Bayesian network model.

```
model = BayesianModel([('C', 'H'), ('P', 'H')])
```

Now, define Conditional Probability Distributions (CPDs)

```
cpd_c = TabularCPD('C', 3, [[1/3], [1/3], [1/3]])

cpd_p = TabularCPD('P', 3, [[1/3], [1/3], [1/3]])

cpd_h = TabularCPD('H', 3, [[0, 0, 0, 0, 1/2, 1, 0, 1/2, 0],

[1/2, 0, 1, 0, 0, 0, 1/2, 0, 0],

[1/2, 1, 0, 1, 1/2, 0, 1/2, 1/2, 1]],

evidence=['C', 'P'], evidence_card=[3, 3])
```

Run the below command to add CPDs to the model.

```
model.add_cpds(cpd_c, cpd_p, cpd_h)
```

To check the model structure and associated conditional probability distributions, you can use the get_cpds() method of the BayesianModel object. If everything is fine, the method will return True. Otherwise, it will raise an error message.

```
model.check_model()
```

To infer the network and determine which door the host will open next, we need to access the posterior probability from the network by providing the evidence. The evidence, in this case, refers to the door selected by the participant and the location of the prize.

```
from pgmpy.inference import VariableElimination

infer = VariableElimination(model)

posterior_p = infer.query(['H'], evidence={'C': 2, 'P': 2})

print(posterior_p)
```

To plot our above model, we can utilize the NetworkX and Matplotlib libraries. NetworkX is a Python package used for the development, manipulation, and study of the structure, dynamics, and functions of complex networks. Matplotlib's PyLab interface provides a convenient way to create visualizations of the network as graphs with nodes and edges. Here is how you can plot the model:

```
import networkx as nx

import matplotlib.pyplot as plt

nodes = model.nodes()

edges = model.edges()

graph = nx.DiGraph()

graph.add_nodes_from(nodes)

graph.add_edges_from(edges)

# Draw the graph

nx.draw(graph, with_labels=True)

plt.savefig('model.png')
```

```
plt.close()
```

The above code snippet generates the Directed Acyclic Graph (DAG) as shown below:

The variations of Bayesian networks in AI

There are several types or variations of Bayesian networks in AI, each with its own characteristics and applications. Here are some common types:

1. **Static Bayesian networks:** These are the most basic type of Bayesian networks, where the relationships between variables are fixed and do not change over time. They are used to model dependencies among variables in a static system.
2. **Dynamic Bayesian networks:** Unlike static Bayesian networks, dynamic Bayesian networks (DBNs) allow for modeling temporal dependencies and changes over time. They can represent probabilistic relationships that evolve or transition between different states.
3. **Hidden Markov Models (HMMs):** HMMs are a type of dynamic Bayesian network widely used in modeling sequential data. They involve a set of hidden states that are not directly observable but can be deduced from observable variables. HMMs are commonly used in speech recognition, natural language processing, and bioinformatics.
4. **Continuous Bayesian networks:** Most traditional Bayesian networks assume discrete variables. However, continuous Bayesian networks deal with continuous variables and use probability distributions such as Gaussian or exponential distributions to represent the relationships between variables.
5. **Hybrid Bayesian networks:** Hybrid Bayesian networks combine discrete and continuous variables in a single model. They can handle both discrete and continuous variables simultaneously and are useful in applications where the data has mixed variable types.
6. **Influence diagrams:** Influence diagrams are a type of Bayesian network that not only represent probabilistic dependencies but also incorporate decision and utility nodes. They are used for decision analysis and optimization problems, allowing for explicitly modeling decisions, uncertainties, and utilities.
7. **Causal Bayesian networks:** While Bayesian networks typically represent associations and dependencies between variables, causal Bayesian networks aim to model causal relationships. They explicitly capture cause-and-effect relationships between variables, making them useful for understanding and predicting causal effects.

These are some of the main types of Bayesian networks. Each type has its own advantages and is suitable for different applications and problem domains.

What are Bayesian networks in AI used for?

Bayesian networks can be used for a wide range of applications in AI and ML. Here are some common uses of Bayesian networks:

1. **Probabilistic inference:** Bayesian networks allow for probabilistic inference, which means they can answer queries about the probability distribution of variables given observed evidence. They can calculate the posterior probability of unobserved variables based on the probabilistic dependencies in the network.
2. **Diagnosis and decision support:** Bayesian networks are widely used in medical diagnosis and decision support systems. By observing symptoms or evidence, the network can compute the probabilities of different diseases or conditions, aiding in the diagnostic process. They can also

assist in decision-making by considering the probabilities and utilities associated with different choices.

3. **Predictive modeling:** Bayesian networks can be used for predictive modeling tasks. Given observed variables, they can predict the values of unobserved variables or estimate their probabilities. This makes them useful in various domains, such as weather forecasting, finance, and customer behavior analysis.
4. **Risk assessment and management:** Bayesian networks are valuable for risk assessment and management. They can model the dependencies between risk factors and estimate the probabilities of different outcomes or events. This is useful in areas such as insurance underwriting, project management, and environmental risk analysis.
5. **Anomaly detection:** Bayesian networks can be used for anomaly detection tasks. By learning the normal behavior of a system or process, they can detect deviations or anomalies from the expected patterns. This is useful in cybersecurity, fraud detection, and monitoring industrial processes.
6. **Natural Language Processing:** Bayesian networks have been applied in natural language processing tasks. They can be used for tasks such as part-of-speech tagging, named entity recognition, and semantic parsing. Bayesian networks can capture the dependencies between linguistic elements and infer the most likely interpretations or structures.
7. **Environmental modeling:** Bayesian networks are employed in environmental modeling to understand complex systems and assess environmental impacts. They can model the interactions between variables such as climate, ecosystems, and human activities, enabling predictions and scenario analyses.
8. **Bioinformatics and genomics:** Bayesian networks are used in bioinformatics and genomics to model and analyze genetic and protein interactions. They can help in understanding gene regulatory networks, protein-protein interactions, and disease-gene associations.

These are just a few examples of the diverse applications of Bayesian networks. Their ability to handle uncertainty and model complex dependencies makes them a valuable tool in various domains, where reasoning under uncertainty and making probabilistic inferences are essential.

How Bayesian networks are used: An example

Now that we know the use cases and practical applications of the Bayesian network let us look into a simple usage of the network, particularly digit generation and visualization. Here, we would use Python and Sorobn, a pre-built Bayesian network architecture.

Prerequisites:

- C++ build tool: Microsoft C++ Build Tools – Visual Studio
- Graphviz: Download | Graphviz

First, let us import the necessary modules, load the dataset and preprocess it.

```
from sklearn import datasets

pixels, digits = datasets.load_digits(return_X_y=True, as_frame=True)

pixels = pixels.astype('uint8')

pixels.columns = [f"{col.split('_')[1]}-{int(col.split('_')[2])}" for col in pixels.columns]

pixels.head()
```

From the imported dataset, let us visualize the images by creating a 5×5 grid of images, with each image displayed using grayscale colormap and accompanied by a title.

```
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1 import ImageGrid

img_shape = (8, 8)

fig = plt.figure(figsize=(7, 7))

grid = ImageGrid(fig, 111, nrows_ncols=(5, 5), axes_pad=.25)

for i, ax in enumerate(grid):

img = pixels.iloc[i].values.reshape(img_shape)

ax.imshow(img, cmap='gray')

ax.set_title(digits.iloc[i])

ax.axis('off')
```

Next, define a function called ‘neighbors’ that calculates the neighboring coordinates of a given point in a grid.

```
def neighbors(r, c):

top = (r - 1, c)

left = (r, c - 1)

if r and c:

return [top, left]

if r:

return [top]

if c:

return [left]

return []

neighbors(0, 0)

neighbors(0, 1)
```

```
neighbors(1, 0)

neighbors(1, 1)
```

Create the Bayesian network.

```
import sorobn

structure = [

(f'{neighbor[0]}-{neighbor[1]}', f'{r}-{c}')

for r in range(img_shape[0])

for c in range(img_shape[1])

for neighbor in neighbors(r, c)

]

bn = sorobn.BayesNet(*structure)
```

Run the following code to visualize the network.

```
bn.graphviz()
```

Using daft library, we can also arrange the nodes in the graphical model based on their associated pixel positions.

```
import daft

pgm = daft.PGM(node_unit=.7, grid_unit=1.6, directed=True)

for rc in bn.nodes:

r, c = rc.split('-')

pgm.add_node(node=rc, x=int(c), y=img_shape[0] - int(r))

for parent, children in bn.children.items():

for child in children:

pgm.add_edge(parent, child)

pgm.render();
```

Furthermore, it is possible to define additional relationships by extending the structure of the Bayesian network accordingly.

```
def neighbors(r, c):

top_left = (r - 1, c - 1)

left = (r, c - 1)

top = (r - 1, c)

if r and c:

return [top, left, top_left]
```

```
if r:
return [top]
if c:
return [left]
return []
structure = [
(f'{neighbor[0]}-{neighbor[1]}', f'{r}-{c}')
for r in range(img_shape[0])
for c in range(img_shape[1])
for neighbor in neighbors(r, c)
]
bn = sorobn.BayesNet(*structure)
pgm = daft.PGM(node_unit=.7, grid_unit=1.6, directed=True)
for rc in bn.nodes:
r, c = rc.split('-')
pgm.add_node(node=rc, x=int(c), y=img_shape[0] - int(r))
for parent, children in bn.children.items():
for child in children:
pgm.add_edge(parent, child)
pgm.render();
```

As the above network structures do not facilitate sampling, let us define a simpler structure:

```
def neighbors(r, c):
if r == c == 0:
return []
# Even row number
if r % 2 == 0:
if c:
return [(r, c - 1)]
return [(r - 1, c)]
if c == img_shape[1] - 1:
```

```
return [(r - 1, c)]

return [(r, c + 1)]

neighbors(0, 0)

neighbors(0, 1)

neighbors(1, 7)

neighbors(2, 0)

import sorobn

structure = [

(f'{neighbor[0]}-{neighbor[1]}', f'{r}-{c}')

for r in range(img_shape[0])

for c in range(img_shape[1])

for neighbor in neighbors(r, c)

]

bn = sorobn.BayesNet(*structure)

pgm = daft.PGM(node_unit=.7, grid_unit=1.6, directed=True)

for rc in bn.nodes:

r, c = rc.split('-')

pgm.add_node(node=rc, x=int(c), y=img_shape[0] - int(r))

for parent, children in bn.children.items():

for child in children:

pgm.add_edge(parent, child)

pgm.render();
```

Now, let us fit the network to the data.

```
bn = sorobn.BayesNet(*structure)

bn = bn.fit(pixels[digits == 0])
```

Finally, we can generate samples from the Bayesian network and display them as grayscale images on a grid of subplots.

```
import pandas as pd

fig = plt.figure(figsize=(7, 7))

grid = ImageGrid(fig, 111, nrows_ncols=(5, 5), axes_pad=.1)

for ax in grid:
```

```
sample = bn.sample()

img = pd.Series(sample).values.reshape(img_shape)

ax.imshow(img, cmap='gray')

ax.axis('off')
```

You can access the whole set of codes through this GitHub link.

What is a Bayesian Neural Network (BNN)?

Bayesian Neural Networks (BNNs) are a type of neural network that incorporates Bayesian inference principles to introduce uncertainty into the weights and biases of the network. Traditional neural networks use fixed weights and biases learned through optimization algorithms like gradient descent. In contrast, BNNs treat the weights and biases as random variables with prior distributions.

Bayesian neural networks combine the flexibility and expressive power of neural networks with the ability to capture uncertainty through Bayesian inference. By treating the weights and biases of the network as random variables, BNNs provide a probabilistic framework for learning and making predictions.

In a BNN, prior distributions are assigned to the weights and biases, which reflect the initial beliefs about their values before observing any data. These priors can be chosen based on prior knowledge or assumptions about the problem domain. During the training process, the BNN updates the priors based on the observed data, resulting in posterior distributions over the weights and biases.

The key challenge in BNNs is to approximate the posterior distribution, which captures the updated beliefs about the weights and biases given the observed data. Exact inference in BNNs is generally intractable due to neural networks' complex, non-linear nature. Therefore, approximate inference methods are commonly used to estimate the posterior distribution.

Introduction

Maximum likelihood is an approach commonly used for such density estimation problems, in which a likelihood function is defined to get the probabilities of the distributed data. It is imperative to study and understand the concept of maximum likelihood as it is one of the primary and core concepts essential for learning other advanced machine learning and deep learning techniques and algorithms.

In this article, we will discuss the likelihood function, the core idea behind that, and how it works with code examples. This will help one to understand the concept better and apply the same when needed.

Let us dive into the likelihood first to understand the maximum likelihood estimation.

What is the Likelihood?

In machine learning, the likelihood is a measure of the data observations up to which it can tell us the results or the target variables value for particular data points. In simple words,

as the name suggests, the likelihood is a function that tells us how likely the specific data point suits the existing data distribution.

For example. Suppose there are two data points in the dataset. The likelihood of the first data point is greater than the second. In that case, it is assumed that the first data point provides accurate information to the final model, hence being likable for the model being informative and precise.

After this discussion, a gentle question may appear in your mind, If the working of the likelihood function is the same as the probability function, then what is the difference?

Difference Between Probability and Likelihood

Although the working and intuition of both probability and likelihood appear to be the same, there is a slight difference, here the possibility is a function that defines or tells us how accurate the particular data point is valuable and contributes to the final algorithm in data distribution and how likely is to the machine learning algorithm.

Whereas probability, in simple words is a term that describes the chance of some event or thing happening concerning other circumstances or conditions, mostly known as conditional probability.

Also, the sum of all the probabilities associated with a particular problem is one and can not exceed it, whereas the likelihood can be greater than one.

What is Maximum Likelihood Estimation?

After discussing the intuition of the likelihood function, it is clear to us that a higher likelihood is desired for every model to get an accurate model and has accurate results. So here, the term maximum likelihood represents that we are maximizing the likelihood function, called the **Maximization of the Likelihood Function**.

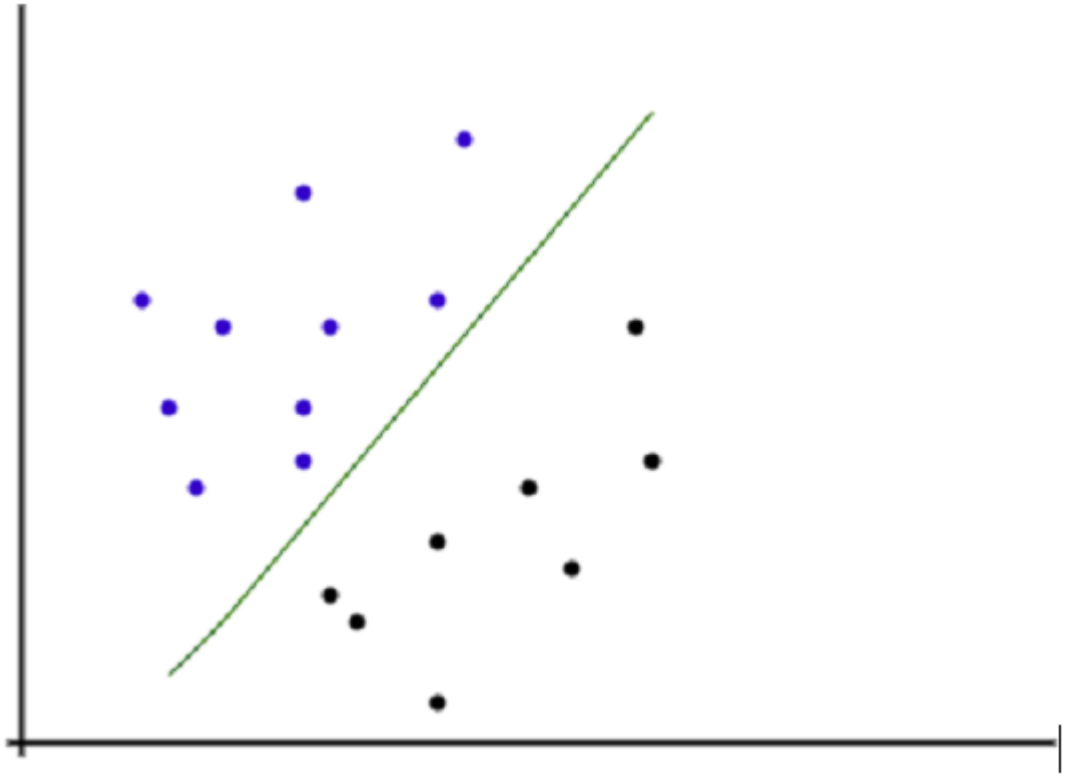
Let us try to understand the same with an example.

Let us suppose that we have a classification dataset in which the independent column is the marks of the students that they achieved in the particular exam, and the target or dependent column is categorical, which has yes and No attributes representing if students are placed on the campus placements or not.

Noe here, if we try to solve the same problem with the help of maximum likelihood estimation, the function will first calculate the probability of every data point according to every suitable condition for the target variable. In the next step, the function will plot all the data points in the two-dimensional plots and try to find the line that best fits the dataset to divide it into two parts. Here the best-fit line will be achieved after some epochs, and once achieved, the line is used to classify the data point by simply plotting it to the graph.

Maximum Likelihood: The Base

The maximum likelihood estimation is a base of some machine learning and deep learning approaches used for classification problems. One example is logistic regression, where the algorithm is used to classify the data point using the best-fit line on the graph. The same approach is known as the perceptron trick regarding deep learning algorithms.



As shown in the above image, all the data observations are plotted in a two-dimensional diagram where the X-axis represents the independent column or the training data, and the y-axis represents the target variable. The line is drawn to separate both data observations, positives and negatives. According to the algorithm, the observations that fall above the line are considered positive, and data points below the line are regarded as negative data points.

Maximum Likelihood Estimation: Code Example

We can quickly implement the maximum likelihood estimation technique using logistic regression on any classification dataset. Let us try to implement the same.

```
import pandas as pd
import numpy as np
import seaborn as sns
from sklearn.linear_model import LogisticRegression
lr=LogisticRegression()
lr.fit(X_train,y_train)
lr_pred=lr.predict(X_test)
sns.regplot(x="X",y='lr_pred',data=df_pred ,logistic=True, ci=None)
```

The above code will fit the logistic regression for the given dataset and generate the line plot for the data representing the distribution of the data and the best fit according to the algorithm.

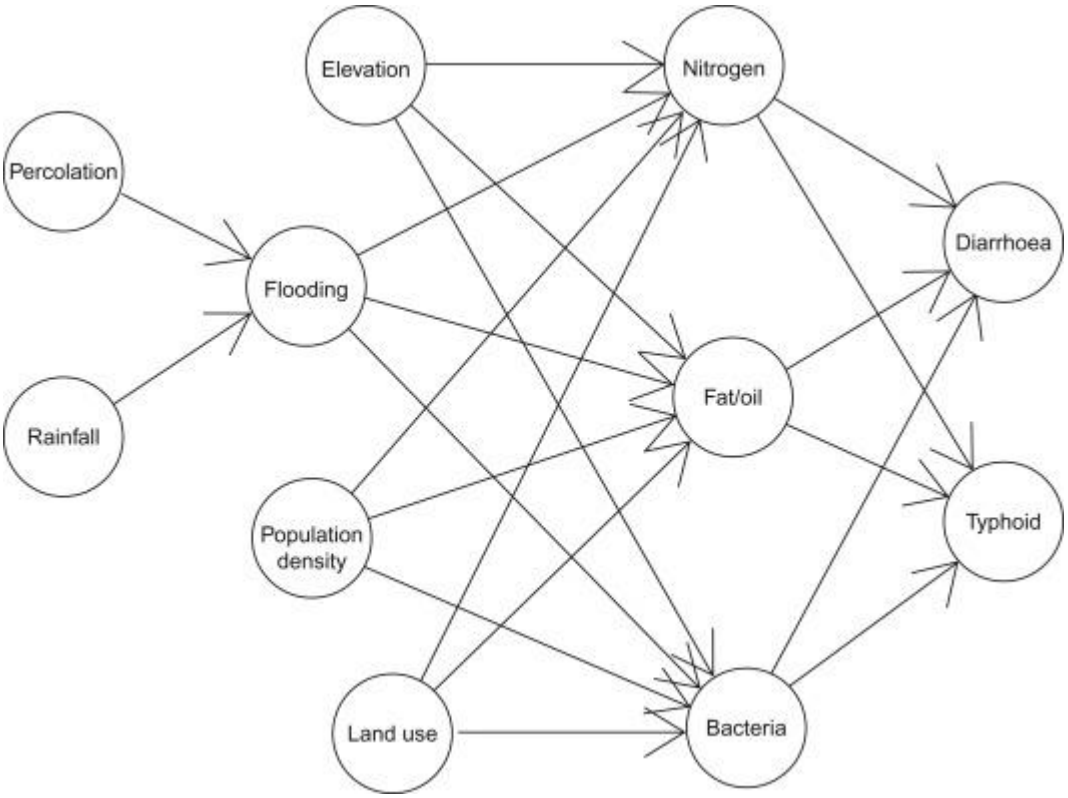
Key Takeaways

- Maximum Likelihood is a function that describes the data points and their likeliness to the model for best fitting.
- Maximum likelihood is different from the probabilistic methods, where probabilistic methods work on the principle of calculation probabilities. In contrast, the likelihood method tries o maximize the likelihood of data observations according to the data distribution.
- Maximum likelihood is an approach used for solving the problems like density distribution and is a base for some algorithms like logistic regression.
- The approach is very similar and is predominantly known as the perceptron trick in terms of deep learning methods.

2.2.3 Bayesian networks

A Bayesian network is a probabilistic graphical model that measures the conditional dependence structure of a set of random variables based on the Bayes theorem:
$P(A|B) = P(B|A)P(A)/P(B)$.

Pearl (1988) stated that Bayesian networks are graphical models that contain information about causal probability relationships between variables and are often used to aid in decision making. The causal probability relationships in a Bayesian network can be suggested by experts or updated using the Bayes theorem and new data being collected. The inter-variable dependence structure is represented by nodes (which depict the variables) and directed arcs (which depict the conditional relationships) in the form of a directed acyclic graph (DAG). As an example, the following DAG indicates that the incidence of two waterborne diseases (diarrhoea and typhoid) depends on three indicators of water samples: total nitrogen, fat/oil and bacteria count, each of which is influenced by another layer of nodes: elevation, flooding, population density and land use. Furthermore, flooding may be influenced by two factors: percolation and rainfall.



Sign in to download full-size image

There are two components involved in learning a Bayesian network: (i) structure learning, which involves discovering the DAG that best describes the causal relationships in the data, and (ii) parameter learning, which involves learning about the conditional probability distributions. The two most popular methods for determining the structure of the DAG are the DAG search algorithm (Chickering, 2002) and the K2 algorithm (Cooper & Herskovits, 1992). Both of these algorithms assign equal prior probabilities to all DAG structures and search for the structure that maximizes the probability of the data given the DAG, that is, P(data|DAG) is maximized. This probability is known as the Bayesian score. Once the DAG structure is determined, the maximum likelihood estimator is employed as the parameter learning method. Note that it is often critical to incorporate prior knowledge about causal structures in the parameter learning process. For instance, consider the causal relationship between two binary variables: rainfall (large or small) and flooding (whether there is a flood or not). Apparently, the former has impact on the latter. Denote the four corresponding events:

R: Large rainfall,
R^-: Small rainfall,
F: There is a flood,
F^-: There is no flood.

In the absence of prior knowledge, the four joint probabilities $P(R,F)$, $P(R^-,F)$, $P(R,F^-)$ and $P(R^-,F^-)$ need to be inferred using the observed data; otherwise, these probabilities can be pre-determined before fitting the Bayesian network to data. Assume that the following statement is made by an expert: if the rainfall is large, the chance of flooding is 60%; if the rainfall is small, the chance of no flooding is four times as big as that of flooding. Then we have the following causal relationship as prior information:
$P(F|R)=0.6, P(F^-|R)=0.4, P(F|R^-)=0.2, P(F^-|R^-)=0.8$.

Furthermore, assume that meteorological data show that the chance of large rainfalls is 30%, namely, $P(R)=0.3$. Then the following contingency table is determined:

	F	F^-
R	$P(R,F)=P(F\|R)P(R)=0.18$	$P(R,F^-)=P(F\|R^-)P(R^-)=0.14$
R^-	$P(R^-,F)=P(F^-\|R)P(R)=0.12$	$P(R^-,F^-)=P(F^-\|R^-)P(R^-)=0.56$

which is an example of pre-specified probabilities based on prior knowledge.
For the purpose of classification, the naïve Bayes classifier has been extensively applied in various fields, such as the classification of text documents (spam or legitimate email, sports or politics news, etc.) and automatic medical diagnosis (to be introduced in Chapter 8 of this book). Denote Y the response variable which has k possible outcomes, that is, $Y \in \{1,\ldots,K\}$, and let $X_1,\ldots,X_p$ be the p features that characterize Y. Using the Bayes theorem, the conditional probability of each outcome, given $X_1,\ldots,X_p$, is of the following form:
$P(Y=k|X_1,\ldots,X_p)=P(X_1,\ldots,X_p|Y=k)P(Y=k)/P(X_1,\ldots,X_p)$.

Note that the naïve Bayes classifier assumes that $X_1,\ldots,X_p$ are mutually independent. As a result, $P(Y=k|X_1,\ldots,X_p)$ can be re-expressed as follows:
$P(Y=k|X_1,\ldots,X_p)=P(Y=k)\prod_{j=1}^{p}P(X_j|Y=k)/P(X_1,\ldots,X_p)$,

which is proportional to $P(Y=k)\prod_{j=1}^{p}P(X_j|Y=k)$. The maximum *a posteriori* decision rule is applied and the most probable label of y is determined as follows:
$\hat{k}=\arg\max_{k\in\{1,\ldots,K\}}P(Y=k)\prod_{j=1}^{p}P(X_j|Y=k)$.

An illustration of the naïve Bayes classifier is provided here. Let's revisit the 'German credit' dataset. This time, we consider the following features: duration in months, credit amount, gender, number of people being liable, and whether a telephone is registered under the customer's name. The setting of training and test sets remains the same. The following MATLAB code is implemented:

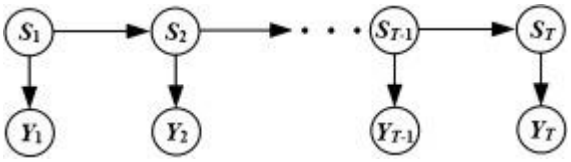
```
y = creditrisk; % Response variable
x = [duration, credit_amount, male, nppl, tele]; % Features
train = 800; % Size of training sample
xtrain = x(1:train,:); % Training sample
ytrain = y(1:train,:); % Labels of training sample
```

```
xtest = x(train+1:end,:); % Test set
ytest = y(train+1:end,:); % Labels of test set
nbayes = fitNaiveBayes(xtrain,ytrain); % Train the Naïve Bayes
ypredict = nbayes.predict(xtest); % Prediction of the test set
rate = sum(ypredict == ytest)/numel(ytest); % Compute the rate of
correct classification
```

Again, 68.5% of the customers in the test set were correctly labelled. The results are summarized in the following table:

		Predicted	
		Good	Bad
Actual	Good	121	18
	Bad	45	16

For modelling time series data, dynamic Bayesian networks can be employed to evaluate the relationship among variables at adjacent time steps (Ghahramani, 2001). A dynamic Bayesian network assumes that an event has impact on another in the future but not vice versa, implying that directed arcs should flow forward in time. A simplified form of dynamic Bayesian networks is known as hidden Markov models. Denote the observation at time t by Y_t, where t is the integer-valued time index. As stated by Ghahramani (2001), the name 'hidden Markov' is originated from two assumptions: (i) a hidden Markov model assumes that Y_t was generated by some process whose state S_t is hidden from the observer, and (ii) the states of this hidden process satisfy the first-order Markov property, where the *r*th order Markov property refers to the situation that given $S_{t-1},\ldots,S_{t-r}$, S_t is independent of S_k for $k<t-r$. The first-order Markov property also applies to Y_t with respect to the states, that is, given S_t, Y_t is independent of the states and observations at all other time indices. The following figure visualizes the hidden Markov model:



Sign in to download full-size image

Mathematically, the causality of a sequence of states and observations can be expressed as follows: $P(S_1,S_1,\ldots,S_t,Y_t)=P(S_1)P(S_1|S_1)\prod_{t=2}P(Y_t|S_t-1)P(Y_t|S_t)$.

Hidden Markov models have shown potential in a wide range of data mining applications, including digital forensics, speech recognition, robotics and bioinformatics. The interested reader is referred to Bishop (2006) and Ghahramani (2001) for a comprehensive discussion of hidden Markov models.

In summary, Bayesian networks appear to be a powerful method for combining information from different sources with varying degrees of reliability. More details of Bayesian networks and their applications can be found in Pearl (1988) and Neapolitan (1990).

View chapter

Bringing dark data to light with AI for evidence-