Experiment 1

Aim: Provide the PEAS description and TASK Environment for a given real world AI Problem.

Objective: To analyze the Performance Measure, Environment, Actuators, Sensors (PEAS) and different categories of TASK environment for given problem before building an intelligentagent.

Theory:

The goal of AI is to build intelligent system which can think and act rationally. For each possible percept sequence, a rational agent should select an action that is expected to maximizeits performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has. Rationality is relative to a performance measure.

Designer of rational agent can judge rationality based on:

- The performance measure that defines the criterion of success.
- The agent prior knowledge of the environment.
- The possible actions that the agent can perform.
- The agent's percept sequence to date.

When we define a rational agent, we group these properties under PEAS, the problem specification for the task environment.

Performance Measure:

If the objective function to judge the performance of the agent, things we can evaluate an agentagainst to know how well it performs.

Environment:

It the real environment where the agent needs to deliberate actions. What the agent can perceive.

Actuators:

These are the tools, equipment or organs using which agent performs actions in the environment. This works as output of the agent. What an agent can use to act in its environment.

Sensors:

These are tools, organs using which agent captures the state of the environment. This work as input to the agent. What an agent can use to perceive its environment.

TASK Environment:

The range of task environments that might arise in AI is obviously vast. We can, however,identify a fairly small number of dimensions along which task environments can be categorized. These dimensions determine, to a large extent, the appropriate agent design and the applicability of each of the principal families of techniques for agent implementation.

1. **Observable (Fully/Partially):** It is a partially observable environment. When an agent can't determine the complete state of the environment at all points of time, then it is called a partially observable environment. Here, the auctioneering agent is not capable of knowing the state of the environment fully at all points in time. Simply, we can say that wherever the agent has to deal with humans in the task environment, it can't observe the state fully.

2. **Agents (Single/Multi):** It is single-agent activity. Because only one agent is involved in this environment and is operating by itself. There are other human agents involved in the activity but they all are passing their percept sequence to the central agent – our auction agent. So, it is still a single-agent environment.

3. **Deterministic (Deterministic/Stochastic):** It is stochastic activity. Because in bidding the outcome can't be determined based on a specific state of the agent. It is the process where the outcome involves some randomness and has some uncertainty

4. **Episodic (Episodic/Sequential):** It is a sequential task environment. In the episodic environment, the episodes are independent of each other. The action performed in one episode doesn't affect subsequent episodes. Here in auction activity, if one bidder set the value X then the next bidder can't set the lesser value than X. So, the episodes are not independent here. Therefore, it is a sequential activity. There is high uncertainty in the environment.

5. **Static (Static/Semi/Dynamic):** It is a dynamic activity. The static activity is the one in which one particular state of the environment doesn't change over time. But here in the auction activity,the states are highly subjective to the change. A static environment is the crossword solving problem where numbers don't change.

6. **Discrete (Discrete/Continuous):** It is a continuous activity. The discrete environment is one that has a finite number of states. But here in auction activity, bidders can set the value forever. The number of states can be 1 or 1000. There is randomness in the environment. Thus, it is a continuous environment

PEAS Descriptors Examples/Problems

1. **PEAS descriptor for Automated Car Driver:**

Performance Measure:

- **Safety**: Automated system should be able to drive the car safely without dashinganywhere.
- **Optimum speed:** Automated system should be able to maintain the optimal speeddepending upon the surroundings.
- **Comfortable journey:** Automated system should be able to give a comfortablejourney to the end user.

Environment:

- **Roads:** Automated car driver should be able to drive on any kind of a road rangingfrom city roads to highway.
- **Traffic conditions:** You will find different sort of traffic conditions for different typeof roads.

Actuators:

- **Steering wheel:** used to direct car in desired directions.
- **Accelerator, gear:** To increase or decrease speed of the car.

Sensors:

- To take i/p from environment in car driving example cameras, sonar system etc.

2. **TASK ENVIORNMENT for automated car driver:**

Fully observable vs. partially observable:

If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is fully observable. A task environment is effectively fully observableif the sensors detect all aspects that are relevant to the choice of action; relevance, in turm, depends on the performance measure. Fully observable environments are convenient because the agent need not maintain any internal state to keep track of the world.An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missingfrom the sensordata.

For example: an automated taxi cannot see what other drivers are thinking.

Output:

Task Environment Table:

	Task Environment	Fully observable	Single agent	Stochastic	Episodic	Static	Discrete	Known
Sr No.	Examples	Partially observable	Multi agent	Deterministic	Sequential	Dynamic	Continuous	Unknown
1	Shopping for used AI books	Fully observable	Single agent	Deterministic	Episodic	Static	Discrete	Known
2	Playing a tennis match	Partially observable	Multi agent	Stochastic	Sequential	Dynamic	Continuous	Unknown
3	Performing a high jump	Fully observable	Single agent	Deterministic	Sequential	Static	Discrete	Known
4	Knitting a Sweater	Fully observable	Single agent	Deterministic	Sequential	Static	Discrete	Known
5	Bidding on item at an auction	Partially observable	Multi agent	Stochastic	Sequential	Static	Continuous	Unknown

PEAS Description table:

Sr No.	PEAS Description Examples	Performance Measure	Environment	Actuator	Sensor
1	Shopping for used AI books	quality, condition, relevance, price	bookstore, availability, customers, website	display, section, voice, bank account	vision, hearing, touch, reviews
2	Playing a tennis match	winning,points scored, errors, strategy, efficiency	tennis court, net, boundaries, audience, opponent	tennis racket, hands, legs,	vision, auditory, tactile sensors
3	Performing a high jump	altitude, speed, landing, accuracy, technique	thigh jump bar, jumping mat, judges, audience, participants	hands and arms, legs, body	vision, gravity, wind,
4	Knitting a Sweater	quality, appearance uniformity, accuracy	Yarn knitting needles, design, skills	hands and fingers, knitting needles	Vision tactile feedback,
5	Bidding on item at an auction	winning rate, profitability, efficiency	auction platform,other bidders, auctioneer	bid placement mechanism,	bid status, current price, auction timer

Conclusion:

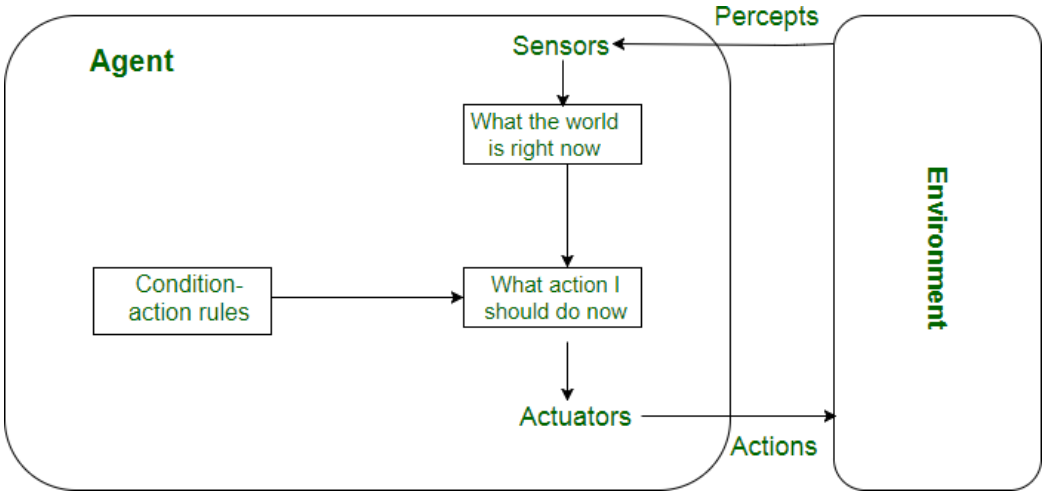
Thus, we have studied to analyze the Performance Measure, Environment, Actuators, Sensors (PEAS) and different categories of TASK environment for given problem before building an intelligent agent.

Experiment 2

Aim: Identify suitable Agent Architecture and type for the problem.

Objective: To study the structure, characteristics of intelligent agent and identify the type of any rational agent.

Theory:



Simple Reflex agent:

- The Simple reflex agents are the simplest agents. These agents take decisions on the basis of the current percepts and ignore the rest of the percept history.
- These agents only succeed in the fully observable environment.
- The Simple reflex agent does not consider any part of percepts history during their decision andaction process.
- The Simple reflex agent works on Condition-action rule, which means it maps the current state toaction. Such as a Room Cleaner agent, it works only if there is dirt in the room.

Model-based reflex agent:

- The Model-based agent can work in a partially observable environment, and track the situation.
- A model-based agent has two important factors:
- **Model:** It is knowledge about "how things happen in the world," so it is called a Model-basedagent.
- **Internal State:** It is a representation of the current state based on percept history.
- These agents have the model, "which is knowledge of the world" and based on the model they perform actions.
- Updating the agent state requires information about:
  - How the world evolves
  - How the agent's action affects the world.

Goal-based agents

- The knowledge of the current state environment is not always sufficient to decide for an agent to what to do.
- The agent needs to know its goal which describes desirable situations.
- Goal-based agents expand the capabilities of the model-based agent by having the "goal" information.
- They choose an action, so that they can achieve the goal.
- These agents may have to consider a long sequence of possible actions before deciding whether the

goal is achieved or not. Such considerations of different scenario are called searching and planning, which makes an agent proactive.

Utility-based agents

- These agents are similar to the goal-based agent but provide an extra component of utility measurement which makes them different by providing a measure of success at a given state.
- Utility-based agent act based not only goals but also the best way to achieve the goal.
- The Utility-based agent is useful when there are multiple possible alternatives, and an agent has to choose in order to perform the best action.
- The utility function maps each state to a real number to check how efficiently each action achieves the goals.

Learning Agents

- A learning agent in AI is the type of agent which can learn from its past experiences, or it has learning capabilities.
- It starts to act with basic knowledge and then able to act and adapt automatically through learning.
- A learning agent has mainly four conceptual components, which are:
  a. **Learning element:** It is responsible for making improvements by learning from environment
  b. **Critic:** Learning element takes feedback from critic which describes that how well the agent is doing with respect to a fixed performance standard.
  c. **Performance element:** It is responsible for selecting external action
  d. **Problem generator:** This component is responsible for suggesting actions that will lead to new and informative experiences.
- Hence, learning agents are able to learn, analyze performance, and look for new ways to improve theperformance.

Output:

Identify the agent architecture for a given problem statement:

Vacuum Cleaner Agent

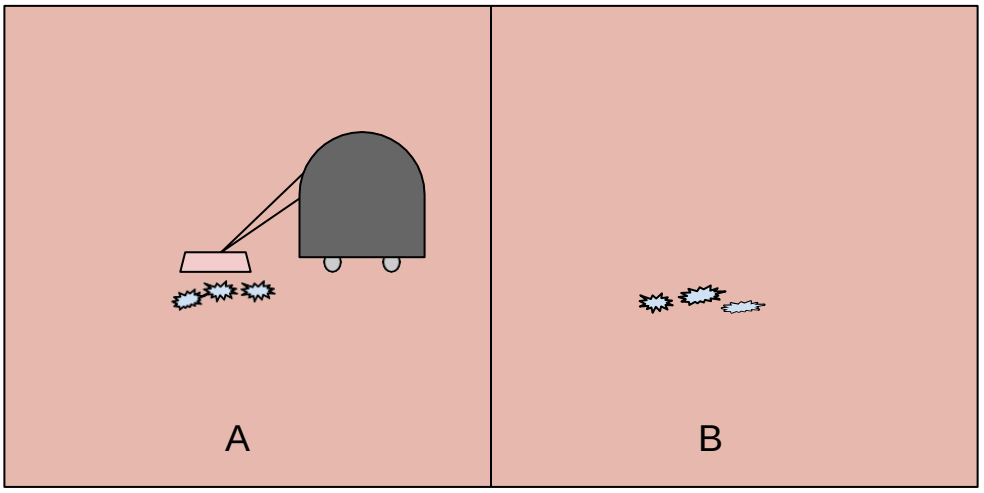
Type of Agent:

Goal-based agents: These kinds of agents take decisions based on how far they are currently from their **goal**(description of desirable situations). Their every action is intended to reduce its distance from the goal. This allows the agent a way to choose among multiple possibilities, selecting the one which reaches a goal state.

Initial State:

The initial state of the vacuum cleaner agent refers to its starting condition before any actions are taken.

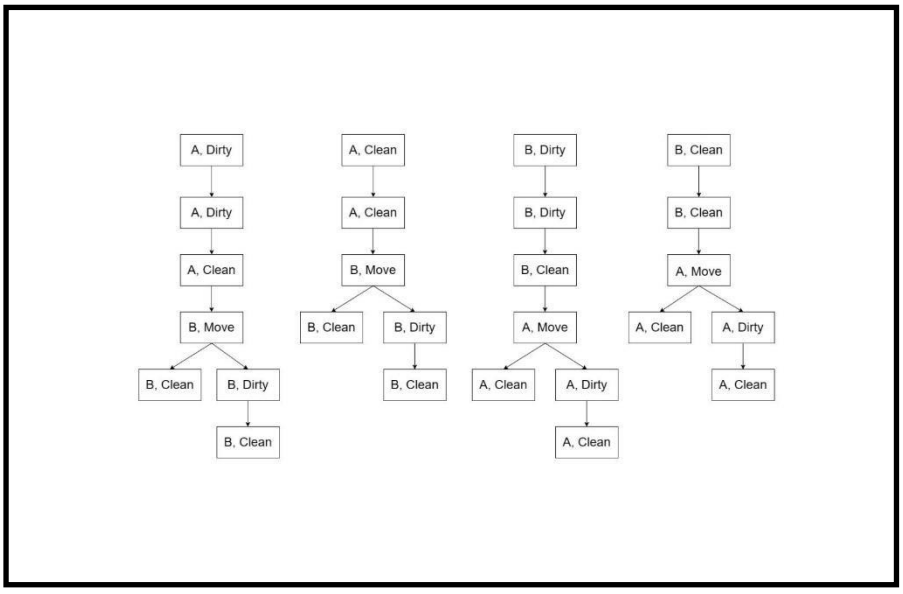
Here the initial state is [A,dirty]



Description: (Architecture)

The agent's environment has two adjacent boxes A and B which can either be clean or dirty. A vacuum cleaner agent can perform the following actions: clean, idle and move to the adjacent box.

Successor:



Goal State:

The goal of the vacuum cleaner agent is to clean both the boxes A and B i.e. [A, clean],[B, clean] by taking all possible actions.

Path Cost:

There are a total of 4 possible paths where each step costs 1 unit.

Path 1 - 5 units.

Path 2- 4 units.

Path 3 - 5 units.

Path 4- 4 units.

For the given example where [A,dirty], the path is:

[A dirty]→[A clean]→[B move]→[B dirty]→[B clean]

Since there are a total of 5 stages; therefore, the path cost is 5 units.

Conclusion:

Thus, we have learned to study the structure, characteristics of intelligent agent and identify the type of any rational agent.

Experiment 3

Aim: Implementation of Breadth first search for problem solving

Objective: To study the uninformed searching techniques and its implementation for problem solving.

Theory:
Artificial Intelligence is the study of building agents that act rationally. Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.

A search problem consists of:

- **A State Space**. Set of all possible states where you can be.
- **A Start State**. The state from where the search begins.
- **A Goal Test**. A function that looks at the current state returns whether or not it is the goal state.

The Solution to a search problem is a sequence of actions, called the plan that transforms the start state to the goal state.
This plan is achieved through search algorithms.

Breadth First Search: BFS is uninformed search method. It is also called blind search.
Uninformed search strategies use only the information available in the problem definition. A search strategy is defined by picking the order of node expansion. It expands nodes from the root of the tree and then generates one level of the tree at a time until a solution is found. It is very easily implemented by maintaining a queue of nodes. Initially the queue contains just the root. In each iteration, node at the head of the queue is removed and then expanded. The generated child nodes are then added to the tail of the queue.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the neighbor nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbor nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:
1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

BFS Algorithm:

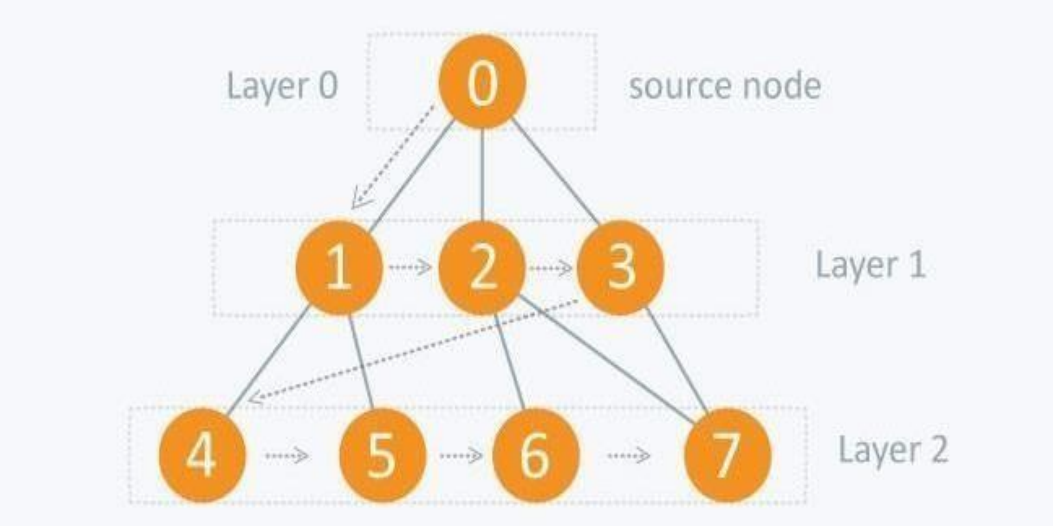
Pseudocode:

BFS (G, s) //Where G is the graph and s is the source node
let Q be queue.
Q.enqueue(s) //Inserting s in queue until all its neighbour vertices are marked.
mark s as visited.

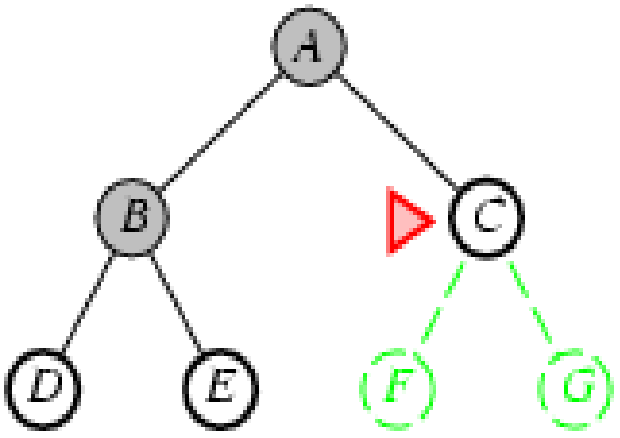
while (Q is not empty)
//Removing that vertex from queue,whose neighbour will be visited now v
= Q.dequeue( )

//processing all the neighbours of v
for all neighbours w of v in Graph G
if w is not visited
Q.enqueue(w) //Stores w in Q to further visit its neighbour
mark w as visited

Working of BFS:



Example: Initial Node: A Goal Node: C



Searching Strategies are evaluated along the following dimensions:

1. Completeness: does it always find a solution if one exists?
2. Time complexity: number of nodes generated
3. Space complexity: maximum number of nodes in memory
4. Optimality: does it always find a least-cost solution?

Properties of Breadth-first search:

1. **Complete**: - Yes: if b is finite.
2. **Time Complexity**: O(b^d+1)
3. **Space Complexity**: O(b^d+1)
4. **Optimal**: Yes.

Advantages of Breadth-First Search:

1. Breadth first search will never get trapped exploring the useless path forever.
2. If there is a solution, BFS will definitely find it out.
3. If there is more than one solution then BFS can find the minimal one that requires less number of steps.

Disadvantages of Breadth-First Search:

1. The main drawback of Breadth first search is its memory requirement. Since each level of the tree must be saved in order to generate the next level, and the amount of memory is proportional to the number of nodes stored.
2. If the solution is farther away from the root, breath first search will consume lotof time.

Applications:

How to determine the level of each node in the given tree?

As you know in BFS, you traverse level wise. You can also use BFS to determine thelevel of each node.

Code:

```
from collections import deque
def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)
    while queue:
        current = queue.popleft()
        for neighbor in graph[current]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
    return visited
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
print(bfs(graph, 'A'))
```

Output:

```
PS D:\Vartak college\SEM 5\AI EXP\New folder> py .\bfs.py
{'D', 'E', 'C', 'A', 'F', 'B'}
```

Conclusion:

Thus, we have studied to implement uninformed searching techniques. Breadth-first search is an uninformed search algorithm that explores nodes in layers outward from a start node using a queue to traverse nodes in breadth-first order while marking visited nodes to avoid repetition.

Experiment 4

Aim: Implementation of Depth first search and its depth limited version (IDDFS).

Objective: To study the uninformed searching techniques and its implementation for problem solving.

Theory:

Artificial Intelligence is the study of building agents that act rationally. Most of the time, theseagents perform some kind of search algorithm in the background in order to achieve their tasks. A search problem consists of:

- **A State Space.** Set of all possible states where you can be.
- **A Start State.** The state from where the search begins.
- **A Goal Test.** A function that looks at the current state returns whether or not itis the goal state.
- The **Solution** to a search problem is a sequence of actions, called the **plan** that transformsthe start state to the goal state.
- This plan is achieved through search algorithms.

Depth First Search: DFS is an uninformed search method. It is also called blind search. Uninformed search strategies use only the information available in the problem definition. A search strategy is defined by picking the order of node expansion. Depth First Search (DFS) searches deeper into the problem space. It is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, elseby backtracking.

The basic idea is as follows:
Pick a starting node and push all its adjacent nodes into a stack.
Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
Repeat this process until the stack is empty.
However, ensure that the nodes that are visited are marked. This will prevent you from visitingthe same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

Algorithm:
A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles. TheDFS algorithm works as follows:

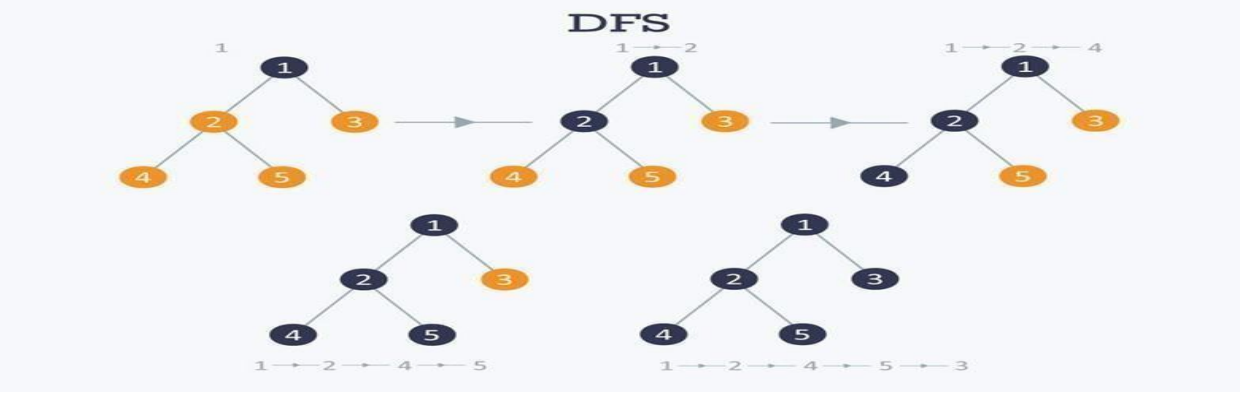
1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in thevisited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

Pseudocode:

```
DFS-iterative (G, s):   //Where G is graph and s is source vertex let S be stack
    S.push( s ) //Inserting s in stackmark s as visited.
    while ( S is not empty):
        //Pop a vertex from stack to visit next v = S.top( )
        S.pop( )
        //Push all the neighbours of v in stack that are notvisited for all neighbours w of v in Graph G:
            if wis not visited
        : S.push( w )
                mark w as visited

    DFS-recursive(G, s):
mark s as visited
        for all neighbours w of s in
```

DFS Working: Examp



Path: 1 □ 2□ 4□ 5□ 3

Searching Strategies are evaluated along the following dimensions:

1. **Completeness:** does it always find a solution if one exists?
2. **Time complexity:** number of nodes generated
3. **Space complexity:** maximum number of nodes in memory
4. **Optimality:** does it always find a least-cost solution?

Properties of depth-first search:

1. Complete:- No: fails in infinite-depth spaces, spaces with loops.
2. Time Complexity: O(b^m)

3. Space Complexity: O(bm), i.e., linear space!
4. Optimal: No

Advantages of Depth-First Search:

1. Memory requirement is only linear with respect to the search graph.
2. The time complexity of a depth-first Search to depth d is O(b^d)
3. If depth-first search finds solution without exploring much in a path then the time andspace it takes will be very less.

Disadvantages of Depth-First Search:

1. There is a possibility that it may go down the left-most path forever. Even a finitegraph can generate an infinite tree.
2. Depth-First Search is not guaranteed to find the solution.
3. No guarantee to find a optimum solution, if more than one solution exists.

Applications:

How to find connected components using DFS?

A graph is said to be disconnected if it is not connected, i.e. if two nodes exist in the graphsuch that there is no edge in between those nodes. In an undirected graph, a connected component is a set of vertices in a graph that are linked to each other by paths.

Consider the example given in the diagram. Graph G is a disconnected graph and has thefollowing 3 connected components.

Code:

```
def iterative_dfs(graph, start):
    stack = [start]
    visited = set()

    while stack:
        current = stack.pop()
        if current not in visited:
            visited.add(current)
            for neighbor in graph[current]:
                stack.append(neighbor)

    return visited

def recursive_dfs(graph, current, visited=set()):
    if current not in visited:
        visited.add(current)
        for neighbor in graph[current]:
```

```
            recursive_dfs(graph, neighbor, visited)
    return visited

graph = {'A': ['B', 'C'],
         'B': ['A', 'D', 'E'],
         'C': ['A', 'F'],
         'D': ['B'],
         'E': ['B', 'F'],
         'F': ['C', 'E']}

print(iterative_dfs(graph, 'A'))
print(recursive_dfs(graph, 'A'))
```

Output:

```
PS D:\Vartak college\SEM 5\AI EXP\New folder> py .\dfs.py
{'F', 'B', 'A', 'E', 'C', 'D'}
{'B', 'F', 'A', 'E', 'C', 'D'}
```

Conclusion:

Thus, we have learned to implement uninformed searching techniques. Depth-first search is an uninformed algorithm that uses recursion and backtracking to traverse nodes along entire branches using a stack before moving on, guaranteeing a solution if one exists.

Experiment 5

Aim: Implementation of A* search for problem solving.

Objective: To study the informed searching techniques and its implementation for problemsolving.

Theory:

Informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc. This knowledge help agents to explore less to the search space and find more efficiently the goal node.

The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

1. f(n)= g(n).

Were, h(n)= estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the priority queue.

A* search algorithm:

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Return to **Step 2**.

Advantages:

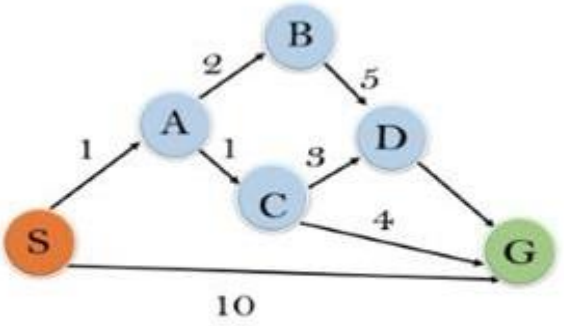
- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

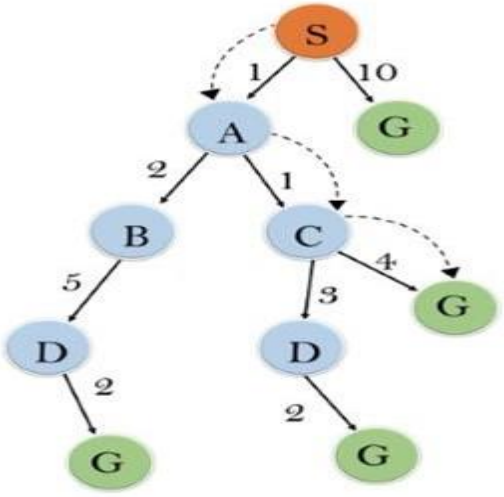
Example:

In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Solution:



Iteration1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Iteration 4 will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.

Points to remember:

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n)<="" li="">

Complete: A* algorithm is complete as long as:

- Branching factor is finite.
- Cost at every action is fixed.

Optimal: A* search algorithm is optimal if it follows below two conditions:

- **Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
- **Consistency:** Second required condition is consistency for only A* graph-search.

If the heuristic function is admissible, then A* tree search will always find the least cost path.

Time Complexity: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is **O(b^d)**

Code:

```
import heapq
dict_hn = {'Arad': 336, 'Bucharest': 0, 'Craiova': 160, 'Drobeta': 242, 'Eforie': 161,
        'Fagaras': 176, 'Giurgiu': 77, 'Hirsova': 151, 'Iasi': 226, 'Lugoj': 244,
        'Mehadia': 241, 'Neamt': 234, 'Oradea': 380, 'Pitesti': 100, 'Rimnicu': 193,
        'Sibiu': 253, 'Timisoara': 329, 'Urziceni': 80, 'Vaslui': 199, 'Zerind': 374}
dict_gn = dict(
   Arad=dict(Zerind=75, Timisoara=118, Sibiu=140),
   Bucharest=dict(Urziceni=85, Giurgiu=90, Pitesti=101, Fagaras=211),
   Craiova=dict(Drobeta=120, Pitesti=138, Rimnicu=146),
   Drobeta=dict(Mehadia=75, Craiova=120),
   Eforie=dict(Hirsova=86),
   Fagaras=dict(Sibiu=99, Bucharest=211),
   Giurgiu=dict(Bucharest=90),
   Hirsova=dict(Eforie=86, Urziceni=98),
   Iasi=dict(Neamt=87, Vaslui=92),
   Lugoj=dict(Mehadia=70, Timisoara=111),
   Mehadia=dict(Lugoj=70, Drobeta=75),
```

```
    Neamt=dict(Iasi=87),
    Oradea=dict(Zerind=71, Sibiu=151),
    Pitesti=dict(Rimnicu=97, Bucharest=101, Craiova=138),
    Rimnicu=dict(Sibiu=80, Pitesti=97, Craiova=146),
    Sibiu=dict(Rimnicu=80, Fagaras=99, Arad=140, Oradea=151),
    Timisoara=dict(Lugoj=111, Arad=118),
    Urziceni=dict(Bucharest=85, Hirsova=98, Vaslui=142),
    Vaslui=dict(Iasi=92, Urziceni=142),
    Zerind=dict(Oradea=71, Arad=75)
)
def a_star(start, goal, dict_hn, dict_gn):
    open_list = [(0, start)]
    g_values = {node: float('inf') for node in dict_hn}
    g_values[start] = 0
    parent_nodes = {}
    while open_list:
        f, current = heapq.heappop(open_list)
        if current == goal:
            return reconstruct_path(parent_nodes, current)
        for neighbor, cost in dict_gn[current].items():
            tentative_g = g_values[current] + cost
            if tentative_g < g_values[neighbor]:
                g_values[neighbor] = tentative_g
                f_value = tentative_g + dict_hn[neighbor]
                heapq.heappush(open_list, (f_value, neighbor))
                parent_nodes[neighbor] = current
    return None
def reconstruct_path(parents, current):
    path = [current]
    while current in parents:
        current = parents[current]
        path.append(current)
    path.reverse()
    return path
start_node = 'Arad'
goal_node = 'Bucharest'
path = a_star(start_node, goal_node, dict_hn, dict_gn)
if path:
    print("Path found:", path)
else:
    print("Path not found.")
```

Output:

```
PS D:\Vartak college\SEM 5\AI EXP\New folder> py .\astar.py
Path found: ['Arad', 'Sibiu', 'Rimnicu', 'Pitesti', 'Bucharest']
```

Conclusion:

Thus, we have learned to implement of A* search for problem solving. Informed search algorithms use heuristic knowledge to focus the exploration and find the goal node more efficiently. The heuristic function estimates the cost to reach the goal from a given node, guiding the search to expand the most promising nodes first.

Experiment 6

Aim: Implementation of Adversarial Search using mini-max algorithm.

Objective: To study the mini-max algorithm and its implementation for problem solving.

Theory:

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.

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.

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 maximizedvalue 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.

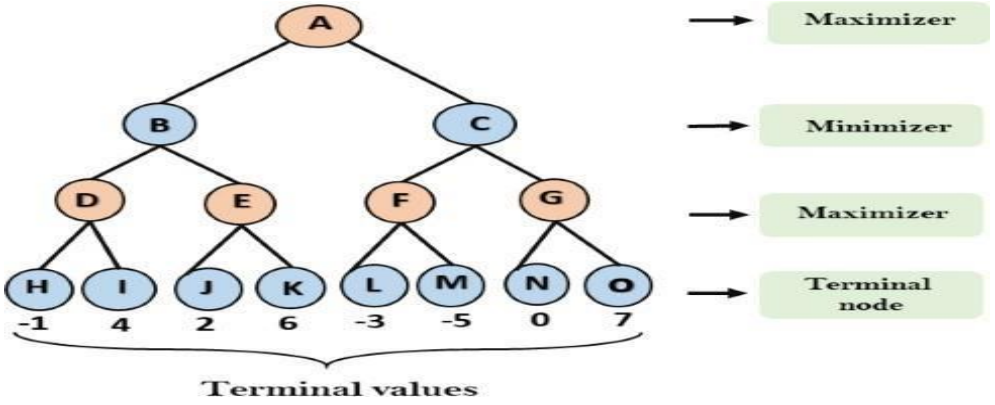
Pseudo-code for MinMax Algorithm:

```
function minimax(node, depth, maximizingPlayer) is
if depth ==0 or node is a terminal node then
    return static evaluation of node4
if MaximizingPlayer then    // for Maximizer Player
    maxEva= -infinity
    for each child of node do
    eva= minimax(child, depth-1, false)
    maxEva= max(maxEva,eva)     //gives Maximum of the values
    return maxEva11
else                // for Minimizer player
    minEva= +infinity
    for each child of node do
    eva= minimax(child, depth-1, true)
    minEva= min(minEva, eva)      //gives minimum of the values
return minEva
```

Initial call:
Minimax(node, 3, true)
Working of Min-Max Algorithm:

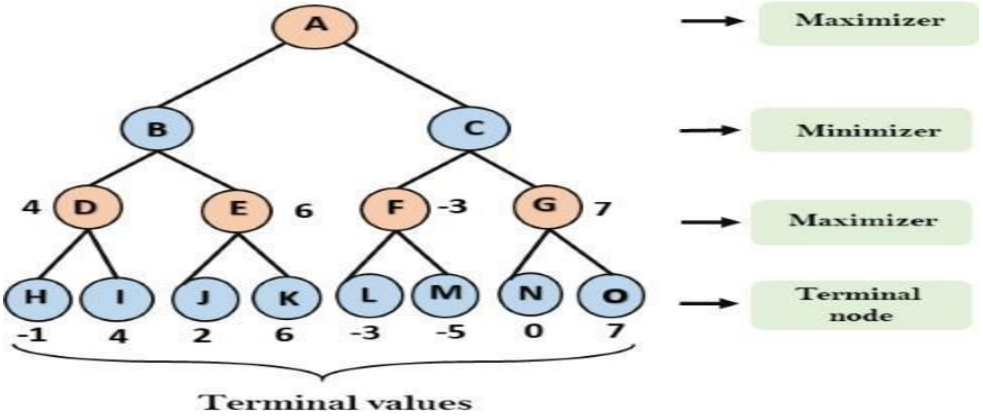
- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum 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 backtrackthe 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.



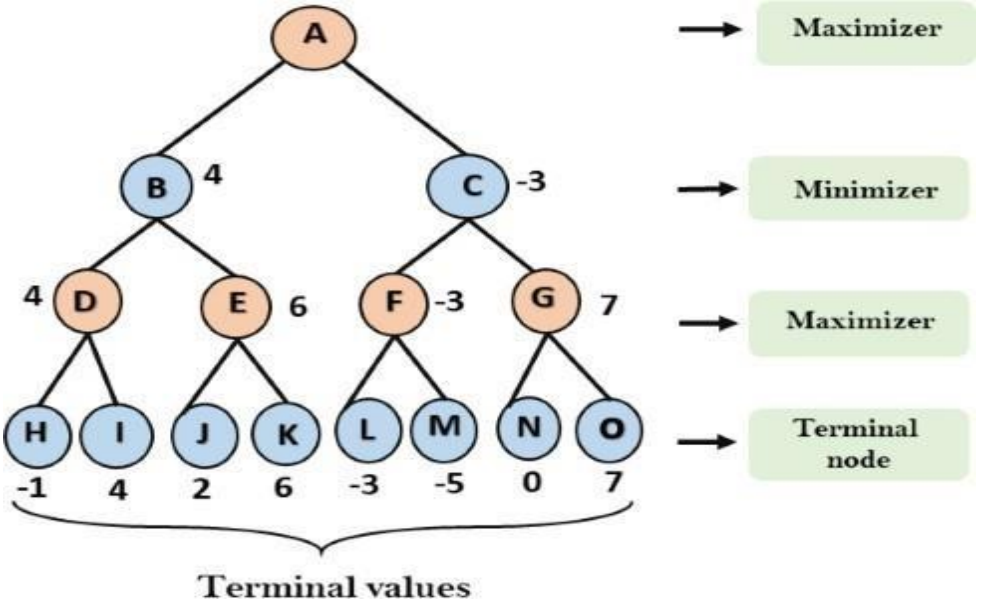
Step 2: Now, first we find the utilities value for the Maximizer, its initialvalue is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find themaximum among the all.

- 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



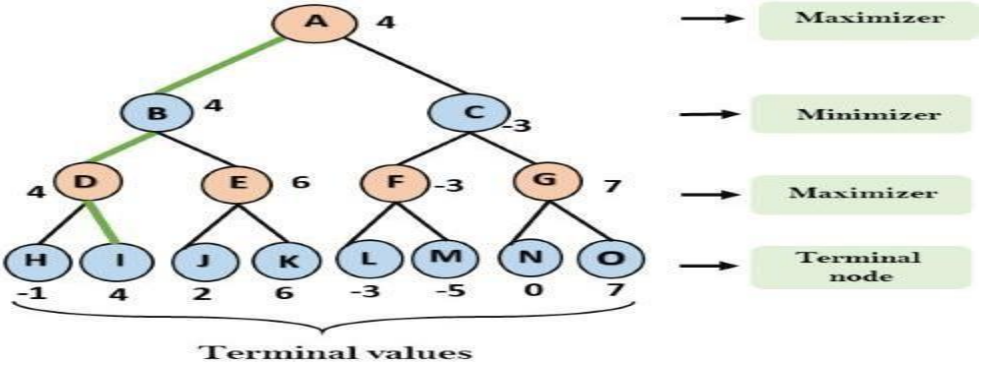
Step 3: In the next step, it's a turn for minimizer, so it will compare allnodes value with $+\infty$, and will find the 3rd layer node values.

- 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 findthe 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 whichis **O(bm)**.

Code:

```
import math
def minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth):
    if (curDepth == targetDepth):
        return scores[nodeIndex]
    if (maxTurn):
        return max(minimax(curDepth + 1, nodeIndex * 2,
                           False, scores, targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1,
                           False, scores, targetDepth))
    else:
        return min(minimax(curDepth + 1, nodeIndex * 2,
                           True, scores, targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1,
                           True, scores, targetDepth))
scores = [3, 5, 2, 9, 12, 5, 23, 23]
treeDepth = math.log(len(scores), 2)
print("The optimal value is : ", end="")
print(minimax(0, 0, True, scores, treeDepth))
```

Output:

```
PS D:\Vartak college\SEM 5\AI EXP\New folder> py .\min.py
The optimal value is : 12
```

Conclusion:

Thus, we have learned to Implement Adversarial Search using mini-max algorithm. The mini-max algorithm is a recursive strategy where two opponents take turns choosing optimal moves to minimize their maximum possible loss based on exploring future game states.

Experiment 7

Aim: Implement knowledge base in Prolog.

Objective: To study and use AI programming language to create knowledge base.

Theory:

Prolog is a logic programming language. It has important role in artificial intelligence. Unlike many other programming languages, Prolog is intended primarily as a declarative programming language. In prolog, logic is expressed as relations (called as Facts and Rules). Core heart of prologlies at the logic being applied. Formulation or Computation is carried out by running a query over these relations.

In prolog, We declare some facts. These facts constitute the Knowledge Base of the system. We can query against the Knowledge Base. We get output as affirmative if our query is already in the knowledge Base or it is implied by Knowledge Base, otherwise we get output as negative. So, Knowledge Base can be considered similar to database, against which we can query. Prolog facts are expressed in definite pattern. Facts contain entities and their relation. Entities are written withinthe parenthesis separated by comma (,). Their relation is expressed at the start and outside the parenthesis. Every fact/rule ends with a dot (.)

Take any problem and represent the knowledge (facts) in prolog. Also you can use this forreasoning purpose.

SWI-Prolog offers a comprehensive free Prolog environment. Since its start in 1987, SWI-Prolog development has been driven by the needs of real world applications. SWI-Prolog is widely used in research and education as well as commercial applications.

SWI-Prolog, **a free implementation of the programming language Prolog**. Susceptibilityweighted imaging, in magnetic resonance imaging (MRI) used in medical contexts.

Logic píogíamming languages, of which PROLOG (programming in logic) is the best known, state a program as a set of logical relations (e.g., a grandparent is the parent of a parent of someone).Such languages are similar to the SQL database language. A program is executed by an "inferenceengine" that answers a query by searching these relations systematically to make infeíences that will answer a query. PROLOG has been used extensively in natural language processing andother AI programs.

Example: The problem of murder mystery.
Five persons Alice, her husband, brother, son and daughter

Event: One murder. One of the five is victim and one is Killer.

Rules:

1) Husband and Alice was not together on the night of murder.
2) The killer and victim were on the beach.
3) On the night of murder, one male and one female was in the bar.
4) The victim was twin and the counterpart was innocent.
5) The killer was younger than the victim.
6) One child was alone at home.

Code for Prolog problem of murder mystery in Artificial Intelligence:predicates

%pair(symbol,symbol)
iskiller(symbol,symbol) male(symbol) female(symbol) isvictim(symbol) not_at_bar(symbol,symbol) not_at_beach(symbol,symbo l) not_alone(symbol) twin(symbol,symbol) younger(symbol,symbol) child(symbol)

clauses

male(husband).male(brother). male(son). female(alice).

female(daughter). twin(brother,alice).twin(son,daughter).

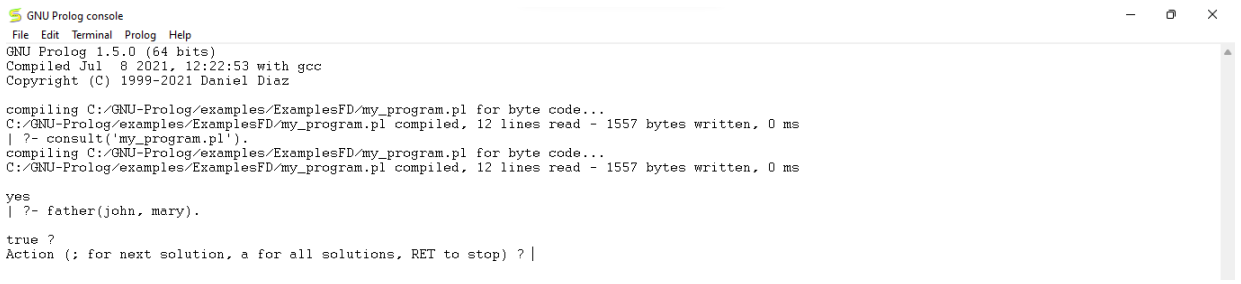
child(son). child(daughteí).

Code:

Code 1:

```
%  Facts
male(john).
male(james).
female(mary).
female(linda).
parent(john, mary).
parent(john, james).
parent(linda, mary).
parent(linda, james).

% Rules
father(X, Y):- male(X), parent(X, Y).
mother(X, Y):- female(X), parent(X, Y).
```

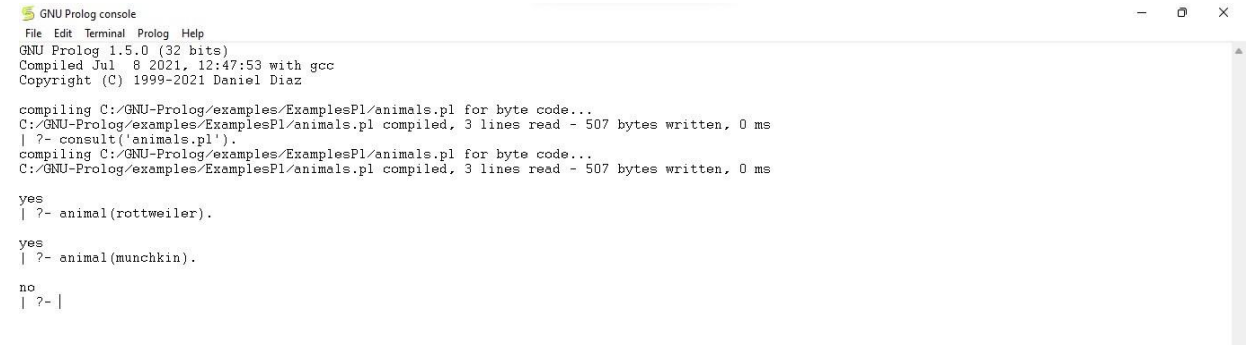


Code 2:

even(X) :- 0 is X mod 2.
odd(X) :- 1 is X mod 2.

Code 3:

dog(rottweiler).
cat(munchkin).
animal(A) :- dog(A).



Conclusion:

Thus, we have learned to implement knowledge base in Prolog. Prolog is a logic programming language. It has important role in artificial intelligence. Unlike many other programming languages, Prolog is intended primarily as a declarative programming language.

Experiment 8

Aim: Implementation of unification algorithm in Prolog.

Objective: To study about how to use AI Programming language (Prolog) for developing interfacing engine using Unification process and knowledge declared in Prolog.

Requirement: Turbo Prolog 2.0 or above / Windows Prolog.

Theory:

Unification is a process of making two different logical atomic expressions identical by findinga substitution. .....It takes two literals as input and makes them identical using substitution. Let Ψ1 and Ψ2 be two atomic sentences and be a unifier such that, Ψ1 = Ψ2 , then it can be expressed as UNIFY(Ψ1, Ψ2).

For example, if one term is f(X, Y) and the second is f(g(Y, a), h(a)) (where upper case namesare variables and lower case are constants) then the two terms can be unified by identifying X with g(h(a), a) and Y with h(a) making both terms look like f(g(h(a),a), h(a)). The unification can be represented by a pair of substitutions {X g(h(a), a)} and {Y h(a)}.

Unification Algorithm:

```
FUNCTION unify( t1, t2 ) RETURNS (unifiable : BOOLEAN, sigma :SUBSTITUTION) BEGIN
IF t1 OR t2 is a variableTHEN BEGIN
  let x be the variable and let t be the other termIF x == t THEN (unifiable, sigma) := (TRUE,
  NULL_SUBSTITUTION); ELSE IF x occurs in t THEN
  unifiable == FALSE;
  ELSE (unifiable, sigma) := (TRUE,
    {x <- t}); ENDELSE
  BEGIN
  assume t1 == f(x1,......, xn) and t2 == g(y1,
  ... ym) IF f != g OR m != n THEN
unifiable = FALSE; ELSEBEGI
  N k
  := 0;
  unifiable := TRUE;
  sigma := NULL_SUBSTITUTION;
  WHILE k < m AND
  unifiable DO BEGIN
  k := k + 1;
  (unifiable, tau) := unify( sigma( xk ), sigma( yk ) ); IF unifiable THEN sigma := compose(tau, sigma );
  END
  END
  END
  RETURN (unifiable,sigma); END
```

Implementation Notes

1. To extract the name of a functor and its arguments, you may use the special built-inrules **functor/3**, **arg/3**, and "**=..**" . (Prolog allows overloading of rule names; the notation foo/2 denotes the foo rule that takes two arguments.) They are used as follows:

   1. functor(f(x,y),F,N) ==> F=f andN=2 2. arg(1,f(x,y),A) ==> A=x 3. f(x,y) =.. L ==> L

= [f,x,y]

Incidentally, an atom is treated as a 0-argument functor.

2. As an option, you may encode functors to be unified as lists in prefix notation. Forexample, f(x) would be encoded as [f, x]. For a more complicated example, the following function:
   f(3, g(x)) would be encoded as:
   [f, 3, [g, x]]
   This notation doesn't look as nice, but it might make the implementation simpler.
3. You must choose how to distinguish variables from atoms in the expressions you are matching. For example, if **a** and **b** are constants, then unification of **a** and **b** should fail. However, if **A** and **B** are both variables, then unification should succeed, with thesingle substitution **A** -> **B**. A reasonable choice is that t, u, v, w, x, y, and z are variables, while all other letters are constants. In any case, please document your choice.

Testing Your Unifier

Here are some tests you should try before stopping work on your unifier. Harder tests are towards the bottom.

1. Two atoms should unify iff both atoms are the same. Two different atoms should failto unify.
2. A variable should unify with anything that does not contain that variable. Forexample, x should unify with f(g(y),3,(h(a,z))), but not with f(x).
3. A variable should unify with itself. For example, x should unify with x.
4. Your algorithm should handle cases where a variable appears in multiple locations
   For example, all of the following should unify:

- f(x,x) = f(a,a)
  - f(x,g(x)) = f(a, g(x))
- f(x, y) = f(y, x)
  And the following should NOTunify
- f(x,x) = f(a,b)
  - f(x,g(x)) = g(a, g(b))

5. When unifying functors, all arguments should unify. For example, g(h(1,2,3,4), 5)does not unify with g(h(1,8,3,4),5).
6. There are plenty of other things to try. These are just some examples to start.

Unification in Prolog:

The way in which Prolog matches two terms is called unification. The idea is similar to that ofunification in logic: we have two terms and we want to see if they can be made to represent thesame structure. For example, we might have in our database the single Polog clause:

parent(alan, clive). and give the query:

|?- parent(X,Y).

We would expect X to be instantiated to alan and Y to be instantiated to clive when the querysucceeds. We would say that the term parent(X,Y) unifies with the term parent(alan, clive) with X bound to alan and Y bound to clive. The unification algorithm in Prolog is roughly this:

df:un Given two terms and which are to be unified:

If and are constants (i.e. atoms or numbers) then if they are the same succeed. Otherwise fail.

If is a variable then instantiate to . Otherwise, If is avariable then instantiate to .

Otherwise, if and are complex terms with the same arity (number of arguments), find the principal functor of and principal functor of . If these are the same, then take the ordered set of arguments of of and the ordered set of arguments of . For each pair of arguments and from the same position in the term, must unify with . Otherwise fail.

For example: applying this procedure to unify foo(a,X) with foo(Y,b) we get:foo(a,X) and foo(Y,b) are complex terms with the same

arity (2). The principal functor of both terms is foo.

The arguments (in order) of foo(a,X) are a and X. The arguments (in order) of foo(Y,b)are Y and b. So a and Y must unify , and X and b must unify. Y is a variable so we instantiate Y to a.

X is a variable so we instantiate X to b.

The resulting term, after unification is foo(a,b).

The built in Prolog operator '=' can be used to unify two terms. Below are some examples ofits use. Annotations are between ** symbols.

| ?- a = a. ** Two identical atomsunify ** yes

| ?- a = b. ** Atoms don't unify if they aren'tidentical ** no

| ?- X = a. ** Unification instantiates a variable to anatom ** X=a

yes

| ?- X = Y. ** Unification binds two differently namedvariables ** X=_125451 ** to a single, unique variable name **

Y=_125451

yes

| ?- foo(a,b) = foo(a,b). ** Two identical complex termsunify ** yes

| ?- foo(a,b) = foo(X,Y).** Two complex terms unify if theyare ** X=a ** of the same arity, have the same principal**

Y=b ** functor and their argumentsunify ** yes

| ?- foo(a,Y) = foo(X,b).** Instantiation of variables may occur ** Y=b ** in either of the terms to be unified **X=

a yes

| ?- foo(a,b) = foo(X,X).** In this case there is no unification ** no ** because foo(X,X) must have the same

**

** 1st and 2nd arguments **

| ?- 2*3+4 = X+Y. ** The term 2*3+4 has principal functor + ** X=2*3 ** and therefore unifies withX+Y with X instantiated** Y=4 ** to 2*3 and Y instantiated to 4 **yes

| ?- [a,b,c] = [X,Y,Z]. ** Lists unify just like otherterms ** X=a

Y=

b Z=

cyes

| ?- [a,b,c] = [X|Y]. ** Unification using the '|' symbol can be used ** X=a ** to find the head element, X, andtail list, Y, ** Y=[b,c] ** of a list **

yes

| ?- [a,b,c] = [X,Y|Z]. ** Unification on lists doesn't haveto be ** X=a ** restricted to finding the first head element **

Y=b ** In this case we find the 1st and 2nd elements ** Z=[c] ** (X and Y) and then the tail list (Z) **

yes

| ?- [a,b,c] = [X,Y,Z|T]. ** This is a similar example buthere ** X=a** the first 3 elements are unified with

**

Y=b ** variables X, Y and Z, leaving the **Z=c ** tail, T, as an empty list

[] ** T=[]

Yes

| ?- [a,b,c] = [a|[b|[c|[]]]]. ** Prolog is quite happy to unify these ** yes** because they are just notational

**

** variants of the same Prolog term **

Code:
Code 1:

```
is_changed = True

# Add more initial facts
facts = [["vertebrate", "duck"], ["flying", "duck"], ["mammal", "cat"], ["insect", "bee"], ["vertebrate", "dog"]]

def assert_fact(fact):
  global  facts
  global is_changed
  if fact not in facts:
    facts += [fact]
    is_changed = True

while is_changed:
  is_changed = False

  for A1 in facts:
    if A1[0] == "mammal":
      assert_fact(["vertebrate", A1[1]])

    if A1[0] == "vertebrate":
      assert_fact(["animal", A1[1]])

    if A1[0] == "vertebrate" and ["flying", A1[1]] in facts:
      assert_fact(["bird", A1[1]])

print(facts)
```

Output:

```
[['vertebrate', 'duck'], ['flying', 'duck'], ['mammal', 'cat'], ['insect', 'bee'], ['vertebrate', 'dog'], ['animal', 'duck'], ['animal', 'cat'], ['animal', 'dog'], ['vertebrate', 'cat'], ['bird', 'duck']]
```

Code 2:

```
is_changed = True
facts = [["can_fly", "sparrow"], ["has_feathers", "sparrow"], ["animal", "sparrow"]]
def assert_fact(fact):
  global  facts
  global is_changed
  if fact not in facts:
    facts.append(fact)
    is_changed = True
while is_changed:
  is_changed = False
  for A1 in facts:
    if A1[0] == "animal":
      assert_fact(["living_organism", A1[1]])
    if A1[0] == "living_organism" and ["can_fly", A1[1]] in facts:
      assert_fact(["bird", A1[1]])
print(facts)

is_changed = True
facts = [["mammal", "lion"], ["has_fur", "lion"], ["animal", "lion"]]
def assert_fact(fact):
  global  facts
  global is_changed
  if fact not in facts:
    facts.append(fact)
    is_changed = True
while is_changed:
  is_changed = False
  for A1 in facts:
    if A1[0] == "animal":
      assert_fact(["living_organism", A1[1]])
    if A1[0] == "living_organism" and ["mammal", A1[1]] in facts:
      assert_fact(["warm_blooded", A1[1]])
print(facts)
```

Output:

```
[['can_fly', 'sparrow'], ['has_feathers', 'sparrow'], ['animal', 'sparrow'], ['living_organism', 'sparrow'], ['bird', 'sparrow']]
[['mammal', 'lion'], ['has_fur', 'lion'], ['animal', 'lion'], ['living_organism', 'lion'], ['warm_blooded', 'lion']]
```

Conclusion:

Thus, we have studied about how to use AI Programming language (Prolog) for developing Interfacing engine using Unification process and knowledge declared in Prolog.

Experiment 9

Aim: Implementation of Bayes Belief Network

Objective: To study about how to use Bayes Belief Network in reasoning process.

Theory:

Bayesian Belief Network or Bayesian Network or Belief Network is a Probabilistic GraphicalModel (PGM) that represents conditional dependencies between random variables through a Directed Acyclic Graph (DAG). Bayesian Networks are applied in many fields. The main objective of these networks is trying to understand the structure of causality relations.

For example, disease diagnosis, optimized web search, spam filtering, gene regulatory networks, etc.

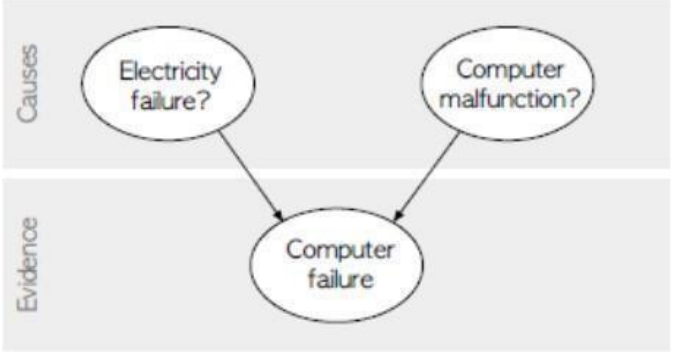
Bayesian Belief Network is a graphical representation of different probabilistic relationships among random variables in a particular set. It is a classifier with no dependency on attributes
i.e it is condition independent. Due to its feature of joint probability, the probability in BayesianBelief Network is derived, based on a condition — P(attribute/parent) i.e probability of an attribute, true over parent attribute.

A Bayesian network represents the causal probabilistic relationship among a set of random variables, their conditional dependences, and it provides a compact representationof a joint probability distribution. It consists of two major parts: a directed acyclic graph and a set of conditional probability distributions. The directed acyclic graph is a set of random variablesrepresented by nodes. For health measurement, a node may be a health domain, and the statesof the node would be the possible responses to that domain. If there exists a causal probabilisticdependence between two random variables in the graph, the corresponding two nodes are connected by a directed edge, while the directed edge from a node A to a node B indicatesthat the random variable A causes the random variable B. Since the directed edges represent a static causal probabilistic dependence, cycles are not allowed in the graph. A conditional probability distribution is defined for each node in the graph. In other words, the conditional probability distribution of a node (random variable) is defined for every possible outcome of the preceding causal node(s).

Example 1:

Suppose we attempt to turn on our computer, but the computer does not start(observation/evidence). We would like to know which of the possible causes of

computer failure is more likely. In this simplified illustration, we assume only two possible causes of this misfortune: electricity failure and computer malfunction. The corresponding directed acyclic graph is depicted in figure.



The two causes in this banal example are assumed to be independent (there is no edge betweenthe two causal nodes), but this assumption is not necessary in general. Unless there is a cycle in the graph, Bayesian networks are able to capture as many causal relations as it is necessary to credibly describe the real-life situation. Since a directed acyclic graph represents a hierarchical arrangement, it is unequivocal to use terms such as parent, child, ancestor, or descendant for certain node.

In figure, both electricity failure and computer malfunction are ancestors and parents of computer failure; analogically computer failure is a descendant and a child of both electricity failure and computer malfunction.

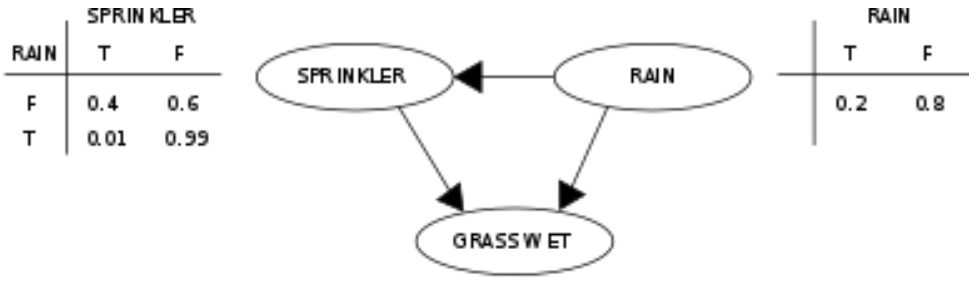
The goal is to calculate the posterior conditional probability distribution of each of the possibleunobserved causes given the observed evidence, i.e. P [Cause | Evidence]. However, inpractice we are often able to obtain only the converse conditional probability distribution of observing evidence given the cause, P [Evidence j Cause]. The whole concept of Bayesian networks is built on Bayes theorem, which helps us to express the conditional probability distribution of cause given the observed evidence using the converse conditional probability ofobserving evidence given the cause:

$$P[\text{Cause} \mid \text{Evidence}] = P[\text{Evidence} \mid \text{Cause}] \cdot \frac{P[\text{Cause}]}{P[\text{Evidence}]}$$

Any node in a Bayesian network is always conditionally independent of its all non-

descendants given that node's parents. Hence, the joint probability distribution of all random variables in the graph factorizes into a series of conditional probability distributions of randomvariables given their parents. Therefore, we can build a full probability model by onlyspecifying the conditional probability distribution in every node

Example 2: A Bayesian network with conditional probability tables



SPRINKLER	T	F
F	0.4	0.6
T	0.01	0.99

RAIN	T	F
	0.2	0.8

SPRINKLER	RAIN	GRASS WET T	GRASS WET F
F	F	0.0	1.0
F	T	0.8	0.2
T	F	0.9	0.1
T	T	0.99	0.01

Code:

```
knowledge_base = {
  'rule1': {
    'condition': lambda person: person['age'] >= 18,
    'result': 'You are eligible for a loan.'
  },
  'rule2': {
    'condition': lambda person: person['income'] > 30000,
    'result': 'You meet the income requirement for a loan.'
  },
  'rule3': {
    'condition': lambda person: person['credit_score'] >= 650,
    'result': 'You have a good credit score.'
  },
  'rule4': {
    'condition': lambda person: person['employment_status'] == 'employed',
    'result': 'You have a stable job.'
  },
  'rule5': {
    'condition': lambda person: person['age'] <= 60,
    'result': 'Your age is within the acceptable range for a loan.'
  },
  'rule6': {
    'condition': lambda person: person['income'] > 50000,
    'result': 'Your income is above the threshold for a higher loan amount.'
  },
  'rule7': {
    'condition': lambda person: person['credit_score'] >= 750,
    'result': 'Your excellent credit score qualifies you for lower interest rates.'
  },
  'rule8': {
    'condition': lambda person: person['employment_status'] == 'self-employed',
    'result': 'Your self-employment status is considered for a loan application.'
  }
}

def evaluate_rules(person):
  results = []

  for rule_name in ['rule1', 'rule2', 'rule3', 'rule4']:
    if not knowledge_base[rule_name]['condition'](person):
      return results

  results.append(knowledge_base[rule_name]['result'])

  return results

age = int(input("Enter your age: "))
income = float(input("Enter your annual income: "))
credit_score = int(input("Enter your credit score: "))
employment_status = input("Enter your employment status (employed/unemployed/self-employed): ")
```

```
user_person = {
  'age': age,
  'income': income,
  'credit_score': credit_score,
  'employment_status': employment_status
}

eligibility_results = evaluate_rules(user_person)

if eligibility_results:
  print("Loan Eligibility Results:")
  for result in eligibility_results:
    print(result)
else:
  print("You are not eligible for a loan.")
```

Output:

```
Enter your age: 25
Enter your annual income: 40000
Enter your credit score: 720
Enter your employment status (employed/unemployed/self-employed): employed

Loan Eligibility Results:
You are eligible for a loan.
You meet the income requirement for a loan.
You have a good credit score.
You have a stable job.
```

Conclusion:

Thus, we have studied about how to use Bayes Belief Network in reasoning process. Bayesian Belief Networks are Probabilistic Graphical Models represented as Directed Acyclic Graphs that model conditional dependencies between random variables to understand causal structure.

Experiment 10

Aim: Case Study on Expert System of real world.

Objective:

1. To develop an analysis and design ability in students to develop the AI applications inexisting domain.
2. Also, to develop technical writing skill in students.

Theory:

Report on the design of Expert system application for healthcare domain:Alexa

Capabilities of Alexa:

- Natural Language Understanding:
  Alexa's core strength lies in its ability to understand and respond to natural language queries. It uses sophisticated Natural Language Processing (NLP) algorithms to comprehend spoken language, enabling users to interact with it in a conversational manner.

- Information Retrieval:
  Alexa can provide real-time information, such as weather updates, news, sports scores, and more. Users can simply ask Alexa for the information they need, and it retrieves data from various sources to deliver timely responses.

- Smart Home Integration:
  One of Alexa's standout features is its integration with smart home devices. Users can control lights, thermostats, security systems, and a wide array of other IoT devices using voice commands. Alexa serves as an expert in home automation.

- Personal Assistant Functions:
  Alexa can set reminders, manage calendars, create to-do lists, and even make online purchases. It acts as a virtual personal assistant, helping users stay organized and efficient.

- Music and Entertainment:
  Alexa can stream music, podcasts, and audiobooks from popular services. Its music recommendation system and playlist creation capabilities demonstrate its expertise in entertainment.

- Third-Party Skills:
  The Alexa Skills Kit allows developers to create third-party "skills" that extend Alexa's capabilities. This open ecosystem allows Alexa to serve as an expert in a wide variety of fields, from cooking to fitness to education.

Development of Alexa:

- Alexa's development involved a combination of cutting-edge technologies, including machine learning, deep learning, and cloud computing. Amazon leveraged its vast cloud infrastructure through Amazon Web Services (AWS) to support Alexa's processing requirements. Key components include:

- Wake Word Detection: Alexa uses keyword spotting to wake up and listen for commands. When it detects the "wake word" (usually "Alexa"), it begins processing the user's request.

- Automatic Speech Recognition (ASR): ASR converts spoken language into text, allowing Alexa to understand user queries accurately.

- Natural Language Understanding (NLU): NLU processes the text input to derive meaning and intent from user requests. It identifies entities and actions, enabling Alexa to formulate appropriate responses.

- Cloud-Based Computing: Alexa's complex tasks, like language understanding and response generation, rely on cloud-based AI models and databases. This allows for frequent updates and improvements.

Impact of Alexa:

- Alexa has had a significant impact on both consumers and the tech industry:

- Smart Home Revolution: Alexa has been a driving force in the proliferation of smart home technology, making it more accessible and user-friendly for millions of households.

- Voice Commerce: Alexa's ability to make purchases has spurred voice commerce. Users can shop online, reorder products, and check the status of their Amazon deliveries using their voice.

- Voice Assistants as a Standard: The success of Alexa has prompted other tech giants like Google and Apple to develop their voice assistants, further advancing the field of natural language understanding and AI.

- Ubiquitous Integration: Alexa can be found in a wide range of devices, from smart speakers to cars, creating an extensive ecosystem that brings expert-level functionality to various aspects of life.

Report on the design of Expert system application for healthcare domain.

Designing an expert system for healthcare is a multifaceted process that leverages advanced technologies to enhance patient care and medical decision-making. It encompasses several key components. The knowledge base, a foundational element, contains a vast repository of medical data, including clinical guidelines, research papers, and patient records, continuously updated to ensure accuracy and relevancy. The inference engine, employing rule-based reasoning, fuzzy logic, and machine learning algorithms, evaluates patient data to generate precise diagnostic recommendations.

The user interface is a critical component, requiring user-friendliness, voice and text inputs, and accessibility. Data integration ensures seamless connectivity with existing healthcare systems and electronic health records, facilitating in-depth analysis for accurate diagnoses. Personalization is paramount, tailoring recommendations to individual patient profiles. Furthermore, the system provides decision support to healthcare professionals, suggesting treatment plans and medication dosages, ultimately reducing medical errors.

Continuous monitoring and alert systems track patient conditions, enabling early intervention. Data security, including encryption and access controls, is crucial for preserving patient confidentiality and integrity. Validation, rigorous testing, and adherence to regulatory requirements are prerequisites for deployment. Ethical considerations are paramount, with expert systems complementing, not replacing, medical professionals.

Conclusion:

The design of an expert system for the healthcare domain is a pivotal step toward improving patient care and supporting healthcare professionals. Its core components, knowledge representation, and inference mechanisms serve as the foundation for informed decision-making. However, while promising, the system must address challenges related to accuracy and ethical considerations surrounding patient data privacy. The healthcare expert system represents a transformative technology with the potential to enhance medical practices and patient outcomes.