**LAB MANUAL**

**FOR**

**ARTIFICIAL INTELLIGENCE (CSL-502)**

**(SEM - V)**

**Artificial Intelligence & Data Science**

**FH-2023**

SUBJECT TITLE: ARTIFICIAL INTELLIGENCE

CLASS: T.E. AI & DS SEM: V

**Lab Objectives**

1. To design suitable Agent Architecture for a given real world AI problem.
2. To implement knowledge representation and reasoning in AI language.
3. To design a Problem-Solving Agent.
4. To incorporate reasoning under uncertainty for an AI agent.

**Lab Outcomes**

1. Identify suitable Agent Architecture for a given real world AI problem.
2. Implement simple programs using Prolog.
3. Implement various search techniques for a Problem-Solving Agent.
4. Represent natural language description as statements in Logic and apply inference rules to it.
5. Construct a Bayesian Belief Network for a given problem and draw probabilistic inferences from it.

**List of Experiments**

| **Sr. No.** | **Name of Experiment** | **Lab Outcomes** |
| --- | --- | --- |
| 1 | Provide the PEAS description and TASK environment for a given AI problem. | LO1 |
| 2 | Identify suitable agent architecture for the problem. | LO1 |
| 3 | Write simple programs using PROLOG as an AI programming Language. | LO2 |
| 4 | Implement any one of the uninformed search techniques. | LO3 |
| 5 | Implement any one of the informed search techniques. E.g. A-star algorithm for 8 puzzle problem. | LO3 |
| 6 | Implement adversarial search using min-max algorithm. | LO3 |
| 7 | Implement any one of the Local Search Techniques. E.g. Hill climbing, simulated Annealing, Genetic algorithm. | LO3 |
| 8 | Prove the goal sentence from the following set of statements in FOPL by applying forward, backward and resolution inference algorithms. | LO4 |
| 9 | Create a Bayesian Network for the given problem statement and draw inference from it. (You can use any Belief and Decision Networks Tool for modeling Bayesian Networks). | LO5 |
| 10 | Design a prototype of an Expert system. | LO5 |
| **Additional Experiments:** | | |
| 1 | Write a program to solve water jug problem. | LO3 |
| 2 | Design an application to simulate number puzzle problem. | LO3 |

Prof. Anagha S. Dhavalikar Dr. Pramod Bhavarthe

**Subject In-charge HOD**

EXPERIMENT NO. 1

**Aim:** Provide the PEAS description and TASK environment for a given AI problem.

**Prior Concepts:**

# Understanding PEAS in Artificial Intelligence

There are different types of agents in AI. PEAS System is used to categorize similar agents together. The PEAS system delivers the performance measure with respect to the environment, actuators, and sensors of the respective agent. Most of the highest performing agents are Rational Agents.

**Rational Agent:**The rational agent considers all possibilities and chooses to perform a highly efficient action. For example, it chooses the shortest path with low cost for high efficiency. **PEAS** stand for a *Performance measure, Environment, Actuator, Sensor*.

1. **Performance Measure:** Performance measure is the unit to define the success of an agent. Performance varies with agents based on their different precepts.
2. **Environment**: Environment is the surrounding of an agent at every instant. It keeps changing with time if the agent is set in motion. There are 5 major types of environments:
   * Fully Observable & Partially Observable
   * Episodic & Sequential
   * Static & Dynamic
   * Discrete & Continuous
   * Deterministic & Stochastic
3. **Actuator**: An actuator is a part of the agent that delivers the output of action to the environment.
4. **Sensor**: Sensors are the receptive parts of an agent that takes in the input for the agent.

Following table gives the examples of various agents and their PEAS.

| **Agent** | **Performance Measure** | **Environment** | **Actuator** | **Sensor** |
| --- | --- | --- | --- | --- |
| Hospital Management System | Patient’s health, Admission process, Payment | Hospital, Doctors, Patients | Prescription, Diagnosis, Scan report | Symptoms, Patient’s response |
| Automated Car Drive | The comfortable trip, Safety, Maximum Distance | Roads, Traffic, Vehicles | Steering wheel, Accelerator, Brake, Mirror | Camera, GPS, Odometer |
| Subject Tutoring | Maximize scores, Improvement is students | Classroom, Desk, Chair, Board, Staff, Students | Smart displays, Corrections | Eyes, Ears, Notebooks |
| Part-picking robot | Percentage of parts in correct bins | Conveyor belt with parts; bins | Jointed arms and hand | Camera, joint angle sensors |
| Satellite image analysis system | Correct image categorization | Downlink from orbiting satellite | Display categorization of scene | Color pixel arrays |

**New Concepts:**

Provide the PEAS description and TASK environment for the given AI problems

1. Bidding on an item at an auction.
2. Shopping for Data Warehousing books on the internet.
3. Hospital Management System.

**Solution:**

1. **Bidding on an item at an auction**

**Performance measures:**

* Cost of the item
* Quality of the item
* Value of the item
* Necessity of the item

**Environment:**

* Auctioneer
* Bidders
* Bidders Items which are to be bid

**Actuators: (means to perform the activity)**

* Speakers
* Microphones
* Display items
* Budget

**Sensors: (means to perceive the environment)**

* Camera
* Price monitor, where prices are being displayed.
* Eyes
* Ears of the attendees.

**Properties of this agent:**

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 base 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.
7. **Shopping for Data Warehousing books on the internet**

**Performance measures:**

* Price of the book
* Author of the book
* Quality of the book
* Book reviews on google.
* Obtain interested/desired books.
* Cost minimization.

**Environment:**

* Internet websites.
* Web pages of a particular website
* Vendors/Sellers
* Shippers

**Actuators:**

* Filling in the forms.
* Display to the user
* Follow URL

**Sensors:**

* Keyboard entry
* Browser used to find web pages
* HTML

**Properties of this agent:**

**1. Observable (Fully or Partial):**  This environment is partially observable. 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 shopping agent can’t see all types of books on one webpage. For example, on the current webpage, all the books have similar ratings and prices. If the user wants to see the books with high ratings then the agent has to follow a different webpage or set the filter in the search bar. Thus, the agent is interacting with a partially observable environment.

**2. Deterministic or non-deterministic:** The environment is deterministic. A task environment is said to be deterministic if the current state and actions performed in the current state completely determines the next state, otherwise, it will be a non-deterministic task environment.

Here, if the shopping agent likes a book and wants to purchase it, then the next state will be followed for the same book. The next stages will be: payment, filling in the delivery address, and order confirmation. The agent will make the payment for the selected book only. Thus, the next state is determined by the current state.

**3. Episodic/Sequential:** This is a sequential environment. An environment is said to be episodic if it consists of independent episodes and actions performed in one episode don’t affect the other episodes. In a sequential environment, the actions performed in the current state will affect the next states.

Here, if the current book is rejected by the agent then the agent will not see the same book again. The webpage will not show the same book again, once it is rejected by the agent. Therefore, the action in the current state completely changed the next possible state.

**4.Static/Dynamic:** It is a static environment. An environment is static if it does not change over time. A car driving environment is dynamic because vehicles are running continuously. The agent doesn’t know what is going to come next. But in the static environment, a particular state is completely unchangeable over time, like a web page.

1. **Hospital Management System:**

**Performance Measure:**

* Accurate and efficient patient record management
* Timely scheduling and management of appointments
* Effective allocation and utilization of hospital resources
* Reduction in waiting times for patients
* Seamless communication and coordination among healthcare staff
* Compliance with regulatory standards and patient privacy requirements

**Environment:**

* The environment consists of a hospital or healthcare facility where various activities and interactions take place
* Patients, doctors, nurses, administrative staff, and other healthcare professionals are the actors within the environment
* The environment includes physical spaces such as patient rooms, waiting areas, examination rooms, and operating theaters
* It also involves digital components like electronic health records (EHR) systems, scheduling software, and communication tools
* External factors such as regulatory guidelines, insurance systems, and technological infrastructure impact the environment

**Actuators:**

* Creating, updating, and accessing patient records in the database
* Scheduling and managing appointments for patients and healthcare providers
* Assigning and coordinating tasks among healthcare staff
* Sending notifications and reminders to patients and staff
* Generating reports and analytics for decision-making
* Controlling and monitoring the availability of hospital resources (beds, equipment, medications, etc.)

**Sensors:**

* Collecting patient information, including medical history, symptoms, vital signs, and test results
* Monitoring the availability and utilization of hospital resources
* Capturing appointment requests and preferences from patients
* Receiving communication and collaboration updates from healthcare staff
* Monitoring the overall system performance and response times
* Tracking compliance with regulatory standards and patient privacy protocols

**Task:**

* Maintaining accurate patient records and ensuring data integrity and confidentiality
* Scheduling and managing appointments efficiently, considering patient preferences and availability of healthcare providers
* Coordinating and allocating hospital resources effectively to optimize patient care
* Facilitating smooth communication and collaboration among healthcare staff
* Monitoring and improving the quality of care and patient satisfaction
* Adhering to regulatory guidelines and privacy standards
* Generating reports and analytics to support decision-making and process improvement

The above PEAS description and TASK environment are general and can vary depending on the specific requirements and functionalities of the given problem.

**Learning Objectives:**

To understand the concept of PEAS in a given AI problem.

**Conclusion/Learning outcome:**

The concept of PEAS and TASK environment is studied and described for the different types AI problems.

EXPERIMENT NO. 2

**Aim:** Identify suitable agent architecture for the problem.

(Ref: <https://rcet.org.in/uploads/academics/regulation2021/rohini_62912743812.pdf>)

**Prior Concepts:**

**Problem-solving approach in artificial intelligence problems:**

The reflex agents are known as the simplest agents because they directly map states into actions. Unfortunately, these agents fail to operate in an environment where the mapping is too large to store and learn. Goal-based agent, on the other hand, considers future actions and the desired outcomes. Here, we will discuss one type of goal-based agent known as a problem-solving agent, which uses atomic representation with no internal states visible to the problem-solving algorithms.

**Problem-solving agent**: The problem-solving agent performs precisely by defining problems and its several solutions.

* According to psychology, “a problem-solving refers to a state where we wish to reach to a definite goal from a present state or condition.”
* According to computer science, a problem-solving is a part of artificial intelligence which encompasses a number of techniques such as algorithms, heuristics to solve a problem. Therefore, a problem-solving agent is a goal-driven agent and focuses on satisfying the goal.

**PROBLEM DEFINITION:**

To build a system to solve a particular problem, we need to do four things:

1. **Define** the problem precisely. This definition must include specification of the initial situations and also final situations which constitute (i.e) acceptable solution to the problem.
2. **Analyze** the problem (i.e) important features have an immense (i.e) huge impact on the appropriateness of various techniques for solving the problems.
3. **Isolate and represent** the knowledge to solve the problem.
4. **Choose the best problem** – solving techniques and apply it to the particular problem.

**Steps performed by Problem-solving agent**

* **Goal Formulation:** It is the first and simplest step in problem-solving. It organizes the steps/sequence required to formulate one goal out of multiple goals as well as actions to achieve that goal. Goal formulation is based on the current situation and the agent’s performance measure (discussed below).
* **Problem Formulation:** It is the most important step of problem-solving which decides what actions should be taken to achieve the formulated goal. There are following five components involved in problem formulation:
* **Initial State:** It is the starting state or initial step of the agent towards its goal.
* **Actions:** It is the description of the possible actions available to the agent.
* **Transition Model:** It describes what each action does.
* **Goal Test:** It determines if the given state is a goal state.
* **Path cost:** It assigns a numeric cost to each path that follows the goal. The problemsolving agent selects a cost function, which reflects its performance measure. Remember, an optimal solution has the lowest path cost among all the solutions.
* **Note:Initial state, actions, and transition model** together define the state-space of the problem implicitly. State-space of a problem is a set of all states which can be reached from the initial state followed by any sequence of actions. The state-space forms a directed map or graph where nodes are the states, links between the nodes are actions, and the path is a sequence of states connected by the sequence of actions.

**Search:** It identifies all the best possible sequence of actions to reach the goal state from the current state. It takes a problem as an input and returns solution as its output.

**Solution:** It finds the best algorithm out of various algorithms, which may be proven as the best optimal solution.

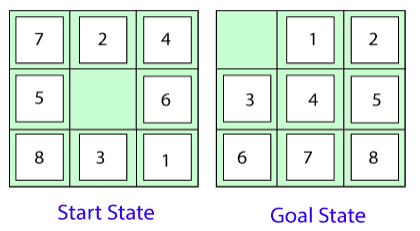
**Execution:** It executes the best optimal solution from the searching algorithms to reach the goal state from the current state.

**New Concept:**

* **Toy Problem:** It is a concise and exact description of the problem which is used by the researchers to compare the performance of algorithms.
* **Real-world Problem:** It is real-world based problems which require solutions. Unlike a toy problem, it does not depend on descriptions, but we can have a general formulation of the problem.

1. **Toy Problems**:

**8 Puzzle Problem:** Here, we have a 3×3 matrix with movable tiles numbered from 1 to 8 with a blankspace. The tile adjacent to the blank space can slide into that space. The objective is to reach a specified goal state similar to the goal state, as shown in the below figure. In the figure, our task is to convert the current state into goal state by sliding digits into the blank space.



In the above figure, our task is to convert the current(Start) state into goal state by sliding digits into the blank space. The problem formulation is as follows:

**States:** It describes the location of each numbered tiles and the blank tile. Initial State: We can start from any state as the initial state

**Actions:** Here, actions of the blank space is defined, i.e., either left, right, up or down.

**TransitionModel:** It returns the resulting state as per the given state and actions.

**Goal test:** It identifies whether we have reached the correct goal-state.

**Path cost:** The path cost is the number of steps in the path where the cost of each step is 1.

**Note:** The 8-puzzle problem is a type of sliding-block problem which is used for testing new search algorithms in artificial intelligence.

For a toy problem, which typically involves a simplified and constrained scenario, a reactive agent architecture is often suitable. Reactive agents are designed to react to the immediate environment without the need for complex planning or learning algorithms. They follow predefined rules or condition-action mappings to respond to stimuli and perform simple tasks. Here are a few examples of toy problems and their suitable agent architectures:

* 1. **Toy Problem: Obstacle Avoidance in a Maze**
* Agent Architecture: Reactive Agent
* The agent's goal is to navigate a maze while avoiding obstacles
* The agent can have predefined rules to sense the presence of obstacles and adjust its movement accordingly
* It does not require memory or complex planning but reacts to the immediate sensory inputs
  1. **Toy Problem: Tic-Tac-Toe Game**
* Agent Architecture: Deliberative Agent
* The agent's goal is to play the game of Tic-Tac-Toe against an opponent
* The agent can have a rule-based system that analyzes the current board state, plans its moves, and makes decisions accordingly
* It maintains an internal model of the game and uses it to simulate and predict the outcome of different moves
  1. **Toy Problem: Coin Collection in a Grid World**
* Agent Architecture: Model-Free Reinforcement Learning Agent
* The agent's goal is to collect coins in a grid world while avoiding obstacles
* The agent can use reinforcement learning algorithms, such as Q-learning, to learn optimal strategies through trial and error
* It interacts with the environment, receiving rewards for collecting coins and penalties for hitting obstacles, and updates its policy based on these rewards
  1. **Toy Problem: Sorting a Set of Numbers**
* Agent Architecture: Deliberative Agent
* The agent's goal is to sort a given set of numbers in ascending order
* The agent can use a deliberative approach, such as using comparison-based sorting algorithms like Bubble Sort or Quick Sort
* It analyzes the current state of the numbers and applies a set of predefined rules to rearrange them until they are sorted

1. **Some Real-world problems**:
   1. **Traveling salesperson problem(TSP):**

It is a touring problem where the salesman can visit each city only once. The objective is to find the shortest tour and sell-out the stuff in each city.

* 1. **VLSI Layout problem:** In this problem, millions of components and connections are positioned on a chip in order to minimize the area, circuit-delays, stray-capacitances, and maximizing the manufacturing yield.

The layout problem is split into two parts:

**Cell layout:** Here, the primitive components of the circuit are grouped into cells, each performing its specific function. Each cell has a fixed shape and size. The task is to place the cells on the chip without overlapping each other.

**Channel routing:** It finds a specific route for each wire through the gaps between the cells.

* 1. **Protein Design:** The objective is to find a sequence of amino acids which will fold into 3D protein having a property to cure some disease.

**Searching for solutions**

For solving different kinds of problem, an agent makes use of different strategies to reach the goal by searching the best possible algorithms. This process of searching is known as **search strategy.**

**1. Traveling Salesperson Problem (TSP)**, which involves finding the shortest route to visit a set of cities and return to the starting city, a suitable agent architecture would be a combination of a deliberative agent and a learning-based approach. Here's a suggested agent architecture for the TSP:

* 1. **Deliberative Agent:**
* The agent maintains an internal model of the problem domain, including the cities, distances between them, and the current state of the tour.
* It uses a planning algorithm, such as dynamic programming or branch and bound, to explore possible routes and find an optimal solution
* The deliberative agent considers all possible permutations of cities to determine the shortest path and updates its internal model accordingly
  1. **Learning-Based Approach (Reinforcement Learning)**
* The agent employs reinforcement learning to improve its performance over time
* It uses a learning algorithm, such as Q-learning or policy gradients, to learn from experience and make better decisions
* The learning agent receives rewards or penalties based on the quality of the solutions it generates
* By exploring different paths and updating its policy based on the received rewards, the agent gradually improves its ability to find shorter routes

The combination of a deliberative agent and a learning-based approach allows for an initial deliberative search for optimal solutions while also enabling the agent to learn from experience and refine its decision-making process. The deliberative agent provides a strong foundation for finding good solutions, and the learning component allows for adaptation and improvement over time. It's worth noting that the TSP is a well-studied problem with various algorithms and heuristics specifically designed to solve it efficiently. Different agent architectures and optimization techniques can be employed based on the size of the problem, available computational resources, and desired performance.

1. **VLSI Layout problem:**

For the VLSI (Very Large Scale Integration) Layout problem, which involves arranging electronic components on a chip to optimize performance and minimize the physical space required, a suitable agent architecture is a combination of a deliberative agent and a constraint satisfaction approach. Here's a suggested agent architecture for the VLSI Layout problem

1. **Deliberative Agent:**

* The agent maintains an internal model of the chip layout problem, including the available space, the components to be placed, and any design constraints or specifications
* It uses a planning or optimization algorithm, such as simulated annealing or genetic algorithms, to explore different component arrangements and find an optimal or near-optimal layout
* The deliberative agent evaluates the quality of the layout based on performance metrics like signal delay, power consumption, and wirelength

1. **Constraint Satisfaction Approach:**

* The agent incorporates a constraint satisfaction mechanism to ensure that the layout adheres to design rules and constraints
* It checks for constraints such as minimum spacing between components, signal routing requirements, and avoidance of routing congestion
* The agent employs constraint propagation techniques to enforce and maintain consistency in the layout solution.

The combination of a deliberative agent and a constraint satisfaction approach allows for a systematic exploration of different component arrangements while ensuring that the resulting layout meets the specified design constraints. The deliberative agent performs optimization to find the best layout in terms of performance metrics, while the constraint satisfaction approach ensures the layout's feasibility and adherence to design rules. It's important to note that the VLSI Layout problem is a complex and computationally intensive task. Various algorithms and techniques, such as partitioning, placement, and routing algorithms, are used in practice. The agent architecture can be further customized based on the specific requirements and constraints of the VLSI layout problem, as well as the available computational resources and performance goals.

**Learning Objectives:**

To understand the use of a specific Agent architecture for a given AI problem.

**Conclusion/Learning outcome:**

The use of a specific Agent architecture for a given AI problem is studied and suitable agent architecture is identified for the given AI problems.

EXPERIMENT NO. 3

**Aim:** Write simple programs using PROLOG as an AI programming Language.

**Prior Concepts:**

Prolog is a programming language that is well-suited for developing logic-based artificial intelligence applications. It is a declarative programming language, meaning that it allows the programmer to specify the rules and facts about a problem domain, and then the Prolog interpreter will use these rules and facts to automatically infer solutions to problems.

One of the key features of Prolog is its ability to handle uncertain or incomplete information. In Prolog, a programmer can specify a set of rules and facts that are known to be true, but they can also specify rules and facts that might be true or false. The Prolog interpreter will then use these rules and facts to automatically reason about the problem domain and find solutions that are most likely to be correct, given the available information.

To use Prolog, a Prolog interpreter is required to be installed. There are several different Prolog interpreters available, including SWI-Prolog, GNU Prolog, and B-Prolog. Once installed an interpreter, Prolog programs can be written using a text editor and then can be run them using the interpreter.

**New Concepts:**

**Write some simple programs in PROLOG**

* 1. **Family Relationship:**

parent(John, Jim).

parent(John, Ann).

parent(Jim, Susan).

parent(Ann, Mike).

sibling(X, Y) :- parent(Z, X), parent(Z, Y).

?- sibling(Jim, Ann).

**Output: true**

* 1. **Factorial Calculation**

factorial(0, 1).

factorial(N, Result) :-

N > 0,

N1 is N - 1,

factorial(N1, SubResult),

Result is N \* SubResult.

?- factorial(5, X).

**Output: X = 120**

* 1. **List Manipulation**

append\_list([], L, L).

append\_list([H|T], L, [H|R]) :-

append\_list(T, L, R).

?- append\_list([1, 2, 3], [4, 5], Result).

**Output: Result = [1, 2, 3, 4, 5]**

* 1. **Fibonacci sequence**

fibonacci(0, 0).

fibonacci(1, 1).

fibonacci(N, Result) :-

N > 1,

N1 is N - 1,

N2 is N - 2,

fibonacci(N1, SubResult1),

fibonacci(N2, SubResult2),

Result is SubResult1 + SubResult2.

?-fibonacci(6, X).

**Output: X = 8**

* 1. **Checking Palindromes**

palindrome([]).

palindrome([\_]).

palindrome([X | Xs]) :-

append(Middle, [X], Xs),

palindrome(Middle).

?- palindrome([a, b, b, a]).

**Output: true**

**Learning Objectives:**

To understand the concept of PROLOG as an AI programming Language.

**Conclusion/Learning outcome:**

The conceptof PROLOG as an AI programming Languageis understood and some sample programs are written in PROLOG.

EXPERIMENT NO. 4

**Aim:** Implement any one of the uninformed search techniques.

**Prior Concepts:**

Explain Uninformed Search techniques.

Uninformed search techniques are strategies for problem-solving that do not use additional information about states beyond the problem definition. Key techniques include:

1. **Breadth-First Search (BFS):** Explores all nodes at the current depth before moving deeper, using a queue.
2. **Depth-First Search (DFS):** Explores as far as possible along a branch before backtracking, using a stack.
3. **Uniform-Cost Search (UCS):** Expands the least costly node first, using a priority queue.
4. **Depth-Limited Search (DLS):** DFS with a depth limit to prevent infinite paths.
5. **Iterative Deepening Search (IDS):** Repeatedly applies DLS with increasing depth limits.
6. **Bidirectional Search:** Simultaneously searches forward from the start and backward from the goal, meeting in the middle.

**New Concept:**

* 1. **Implement the Breadth-First Search (BFS) algorithm in Prolog. BFS is an uninformed search algorithm that explores all the vertices of a graph or tree in breadth-first order.**

**Python Code:**

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = [] # List for visited nodes.

queue = [] #Initialize a queue

def bfs(visited, graph, node): #function for BFS

visited.append(node)

queue.append(node)

while queue:

m = queue.pop(0)

print (m, end = " ")

for neighbour in graph[m]:

if neighbour not in visited:

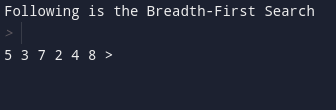
visited.append(neighbour)

queue.append(neighbour)

print("Following is the Breadth-First Search")

bfs(visited, graph, '5')

**Output:**



* 1. **Implement the Depth-First Search (DFS) algorithm in Prolog. DFS is an uninformed search algorithm that explores as far as possible along each branch before backtracking**

**Python Code:**

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = set() #

def dfs(visited, graph, node): #function for dfs

if node not in visited:

print (node)

visited.add(node)

for neighbour in graph[node]:

dfs(visited, graph, neighbour)

print("Following is the Depth-First Search")

dfs(visited, graph, '5')

**Output:**



**Learning Objectives:**

To implement any one of the uninformed search techniques.

**Conclusion/Learning outcome:**

Breadth-First Search (BFS) and Depth-First Search (DFS) techniques of uninformed search algorithm are studied and implemented in PROLOG and Python.

EXPERIMENT NO. 5

**Aim:** Implement any one of the informed search techniques. e.g. A-star algorithm for 8 puzzle problem.

**Prior Concept:**

Explain informed search techniques

**New Concept:**

Implementation of A\* algorithm for 8-puzzle problem in Python

**Python Code**

def aStarAlgo(start\_node, stop\_node):

        open\_set = set(start\_node)

        closed\_set = set()

        g = {} #store distance from starting node

        parents = {}# parents contains an adjacency map of all nodes

        #ditance of starting node from itself is zero

        g[start\_node] = 0

        #start\_node is root node i.e it has no parent nodes

        #so start\_node is set to its own parent node

        parents[start\_node] = start\_node

        while len(open\_set) > 0:

            n = None

            #node with lowest f() is found

            for v in open\_set:

                if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):

                    n = v

            if n == stop\_node or Graph\_nodes[n] == None:

                pass

            else:

                for (m, weight) in get\_neighbors(n):

                    #nodes 'm' not in first and last set are added to first

                    #n is set its parent

                    if m not in open\_set and m not in closed\_set:

                        open\_set.add(m)

                        parents[m] = n

                        g[m] = g[n] + weight

                    #for each node m,compare its distance from start i.e g(m) to the

                    #from start through n node

                    else:

                        if g[m] > g[n] + weight:

                            #update g(m)

                            g[m] = g[n] + weight

                            #change parent of m to n

                            parents[m] = n

                            #if m in closed set,remove and add to open

                            if m in closed\_set:

                                closed\_set.remove(m)

                                open\_set.add(m)

            if n == None:

                print('Path does not exist!')

                return None

            # if the current node is the stop\_node

            # then we begin reconstructin the path from it to the start\_node

            if n == stop\_node:

                path = []

                while parents[n] != n:

                    path.append(n)

                    n = parents[n]

                path.append(start\_node)

                path.reverse()

                print('Path found: {}'.format(path))

                return path

            # remove n from the open\_list, and add it to closed\_list

            # because all of his neighbors were inspected

            open\_set.remove(n)

            closed\_set.add(n)

        print('Path does not exist!')

        return None

#define fuction to return neighbor and its distance

#from the passed node

def get\_neighbors(v):

    if v in Graph\_nodes:

        return Graph\_nodes[v]

    else:

        return None

#for simplicity we ll consider heuristic distances given

#and this function returns heuristic distance for all nodes

def heuristic(n):

        H\_dist = {

            'A': 11,

            'B': 6,

            'C': 99,

            'D': 1,

            'E': 7,

            'G': 0,

        }

        return H\_dist[n]

#Describe your graph here

Graph\_nodes = {

    'A': [('B', 2), ('E', 3)],

    'B': [('C', 1),('G', 9)],

    'C': None,

    'E': [('D', 6)],

    'D': [('G', 1)],

}

aStarAlgo('A', 'G')

**Output**

Path Found: ['A','E','D','G']

**Learning Objectives:**

Implement any one of the informed search techniques. e.g. A-star algorithm for 8 puzzle problem.

**Conclusion/Learning outcome:**

A\* algorithm for 8-puzzle problem is understood and implemented in Python.

EXPERIMENT NO. 6

**Aim:** Implement adversarial search using min-max algorithm.

**Prior Concepts:**

Explain adversarial search using mini-max algorithm.

**New Concepts:**

Implementation of adversarial search using min-max algorithm.

**Python 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:**

The optimal value is: 12

**Learning Objectives:**

To understand Adversarial search using min-max algorithm.

**Conclusion/Learning outcome:**

Adversarial search using mini-max algorithm for developing a game problem is implemented in Python.

EXPERIMENT NO. 7

**Aim:** Implement any one of the Local Search Techniques. E.g. Hill climbing, simulated Annealing, Genetic algorithm.

**Prior Concepts:**

Explain Local Search Techniques like Hill climbing, simulated Annealing, Genetic algorithm.

**New Concepts:**

Implementation of Hill Climbing algorithm in PROLOG to solves the N-Queens problem.

**PYTHON Code:**

import random

def objective\_function(x):

return -x\*\*2 + 4\*x

def hill\_climbing(max\_iterations, step\_size):

current\_solution = random.uniform(-10, 10)

current\_value = objective\_function(current\_solution)

for \_ in range(max\_iterations):

neighbor = current\_solution + random.uniform(-step\_size, step\_size)

neighbor\_value = objective\_function(neighbor)

if neighbor\_value > current\_value:

current\_solution = neighbor

current\_value = neighbor\_value

return current\_solution, current\_value

max\_iterations = 1000

step\_size = 0.1

best\_solution, best\_value = hill\_climbing(max\_iterations, step\_size)

print("Best Solution:", best\_solution)

print("Best Value:", best\_value)

**OUTPUT:**

Best Solution: 1.999998232887578

Best Value: 3.999999999996877

**N-Queen problem using Hill Climbing**

import random

def calculate\_attacking\_queens(board):

n = len(board)

attacking\_queens = 0

for i in range(n):

for j in range(i+1, n):

if board[i] == board[j] or abs(board[i] - board[j]) == j - i:

attacking\_queens += 1

return attacking\_queens

def hill\_climbing\_n\_queens(board\_size, max\_iterations):

current\_board = [random.randint(0, board\_size-1) for \_ in range(board\_size)]

current\_attacks = calculate\_attacking\_queens(current\_board)

for \_ in range(max\_iterations):

new\_board = current\_board.copy()

row\_to\_move = random.randint(0, board\_size-1)

new\_position = random.randint(0, board\_size-1)

new\_board[row\_to\_move] = new\_position

new\_attacks = calculate\_attacking\_queens(new\_board)

if new\_attacks == 0:

return new\_board

if new\_attacks < current\_attacks:

current\_board = new\_board

current\_attacks = new\_attacks

return current\_board

def print\_board(board):

for row in board:

print(" ".join("Q" if i == row else "." for i in range(len(board))))

print()

# Parameters

board\_size = 8

max\_iterations = 1000

# Solve the N-Queens problem using hill climbing

solution = hill\_climbing\_n\_queens(board\_size, max\_iterations)

print("Final Solution:")

print\_board(solution)

**Final Solution:**

**. . . Q . . . .**

**. . . . . . . Q**

**Q . . . . . . .**

**. . . . Q . . .**

**. . Q . . . . .**

**. . . . . Q . .**

**. Q . . . . . .**

**. . . . . . Q .**

**Learning Objectives:**

To implement Local Search Techniques like Hill climbing, simulated Annealing, Genetic algorithm.

**Conclusion/Learning outcome:**

Local Search Techniques is understood and Hill Climbing algorithm is implemented in PROLOG to solve the N-Queens problem.

EXPERIMENT NO. 8

**Aim:** Prove the goal sentence from the following set of statements in FOPL by applying forward, backward and resolution inference algorithms.

**Prior Concepts:**

Explain First Order Prepositional Logic.

**New Concepts:**

Prove the goal sentence from the following set of statements in FOPL by applying forward, backward and resolution inference algorithms.

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

Prove by resolution that: John likes peanuts (goal sentence)

To prove the goal sentence "John likes peanuts" using First-Order Predicate Logic (FOPL) by applying the resolution inference algorithm, we'll first convert the given statements into FOPL statements and then apply the resolution algorithm.

Let's represent the statements in FOPL:

1. John likes all kinds of food:

∀x (Food(x) → Likes(John, x))

2. Apple and vegetable are food:

Food(Apple) ∧ Food(Vegetable)

3. Anything anyone eats and is not killed is food:

∀x ∀y (Eats(x, y) ∧ ¬Killed(y) → Food(y))

4. Anil eats peanuts and is still alive:

Eats(Anil, Peanuts) ∧ ¬Killed(Anil)

5. Harry eats everything that Anil eats:

∀x (Eats(Anil, x) → Eats(Harry, x))

Now, we want to prove the goal sentence: John likes peanuts (Likes(John, Peanuts)).

To prove this goal using resolution, we'll negate the goal and add it to the set of statements, then attempt to derive a contradiction. If we can derive a contradiction, it means the goal is true.

Negation of the goal: ¬Likes(John, Peanuts)

Now, let's perform resolution:

1. Apply resolution to statements 1 and 2:

From statement 1: ∀x (Food(x) → Likes(John, x))

From statement 2: Food(Apple) ∧ Food(Vegetable)

Apply universal instantiation to statement 1 with x = Apple:

Food(Apple) → Likes(John, Apple)

Apply resolution to the above two clauses:

Likes(John, Apple)

2. Apply resolution to statements 4 and 5:

From statement 4: Eats(Anil, Peanuts) ∧ ¬Killed(Anil)

From statement 5: ∀x (Eats(Anil, x) → Eats(Harry, x))

Apply universal instantiation to statement 5 with x = Peanuts:

Eats(Anil, Peanuts) → Eats(Harry, Peanuts)

Apply resolution to the above two clauses:

Eats(Harry, Peanuts)

3. Apply resolution to statements 3 and the result of step 2:

From statement 3: ∀x ∀y (Eats(x, y) ∧ ¬Killed(y) → Food(y))

Apply universal instantiation to statement 3 with x = Harry and y = Peanuts:

Eats(Harry, Peanuts) ∧ ¬Killed(Peanuts) → Food(Peanuts)

Apply resolution to the above two clauses:

Food(Peanuts)

4. Apply resolution to the result of step 1 and the result of step 3:

Likes(John, Apple) ∧ Food(Peanuts)

Apply resolution to the above two clauses:

Food(Peanuts)

5. Apply resolution to statement 2 and the result of step 4:

Food(Apple) ∧ Food(Vegetable) ∧ Food(Peanuts)

Apply resolution to the above three clauses:

Food(Peanuts)

6. Apply resolution to statements 1 and the result of step 5:

∀x (Food(x) → Likes(John, x)) ∧ Food(Peanuts)

Apply universal instantiation to the above clause with x = Peanuts:

Food(Peanuts) → Likes(John, Peanuts)

Apply resolution to the above two clauses:

Likes(John, Peanuts)

We have derived "Likes(John, Peanuts)" through resolution, which means that the goal sentence "John likes peanuts" is proven to be true based on the given FOPL statements.

**Learning Objectives:**

To prove the goal sentence from the given set of statements in FOPL by applying forward, backward and resolution inference algorithms.

**Conclusion/Learning outcome:**

First Order Prepositional Logic is understood and the goal sentence from the given set of statements in FOPL by applying forward, backward and resolution inference algorithms is proved.

EXPERIMENT NO. 9

**Aim:** Create a Bayesian Network for the given problem statement and draw inference from it. (You can use any Belief and Decision Networks Tool for modeling Bayesian Networks)

**Prior Concepts:**

[Ref: <https://www.javatpoint.com/bayesian-belief-network-in-artificial-intelligence>]

"A Bayesian network is a probabilistic graphical model which represents a set of variables and their conditional dependencies using a directed acyclic graph."

It is also called a **Bayes network, belief network, decision network**, or **Bayesian model**.

Bayesian networks are probabilistic, because these networks are built from a **probability distribution**, and also use probability theory for prediction and anomaly detection.

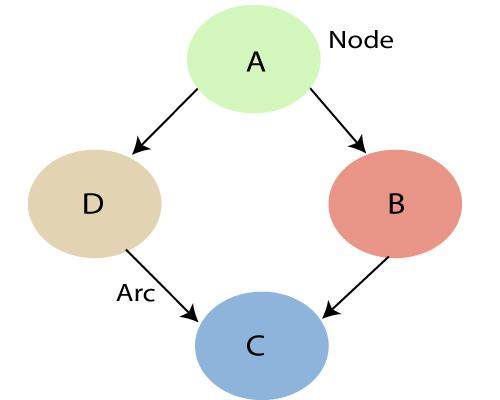
Real world applications are probabilistic in nature, and to represent the relationship between multiple events, we need a Bayesian network. It can also be used in various tasks including **prediction, anomaly detection, diagnostics, automated insight, reasoning, time series prediction**, and **decision making under uncertainty**.

Bayesian Network can be used for building models from data and experts opinions, and it consists of two parts:

* **Directed Acyclic Graph**
* **Table of conditional probabilities.**

The generalized form of Bayesian network that represents and solve decision problems under uncertain knowledge is known as an **Influence diagram**.

**A Bayesian network graph is made up of nodes and Arcs (directed links), where:**



* Each **node** corresponds to the random variables, and a variable can be **continuous** or **discrete**.
* **Arc or directed arrows** represent the causal relationship or conditional probabilities between random variables.
* These directed links or arrows connect the pair of nodes in the graph.  
  These links represent that one node directly influence the other node, and if there is no directed link that means that nodes are independent with each other.
* In the above diagram, A, B, C, and D are random variables represented by the nodes of the network graph.If we are considering node B, which is connected with node A by a directed arrow, then node A is called the parent of Node B.
* Node C is independent of node A.

The Bayesian network has mainly two components:

* **Causal Component**
* **Actual numbers**

Each node in the Bayesian network has condition probability distribution **P(Xi |Parent(Xi) )**, which determines the effect of the parent on that node.Bayesian network is based on Joint probability distribution and conditional probability. So let's first understand the joint probability distribution:

## Joint probability distribution:

## If we have variables x1, x2, x3,....., xn, then the probabilities of a different combination of x1, x2, x3.. xn, are known as Joint probability distribution.

**P[x1, x2, x3,....., xn]**, it can be written as the following way in terms of the joint probability distribution.

**= P[x1| x2, x3,....., xn]P[x2, x3,....., xn]**

**= P[x1| x2, x3,....., xn]P[x2|x3,....., xn]....P[xn-1|xn]P[xn].**

In general for each variable Xi, we can write the equation as:

P(Xi|Xi-1,........., X1) = P(Xi |Parents(Xi ))

**New Concepts:**

**Example:** Harry installed a new burglar alarm at his home to detect burglary. The alarm reliably responds at detecting a burglary but also responds for minor earthquakes. Harry has two neighbors David and Sophia, who have taken a responsibility to inform Harry at work when they hear the alarm. David always calls Harry when he hears the alarm, but sometimes he got confused with the phone ringing and calls at that time too. On the other hand, Sophia likes to listen to high music, so sometimes she misses to hear the alarm. Here we would like to compute the probability of Burglary Alarm.

**Problem:**

**Calculate the probability that alarm has sounded, but there is neither a burglary, nor an earthquake occurred, and David and Sophia both called the Harry.**

**Solution:**

* The Bayesian network for the above problem is given below. The network structure is showing that burglary and earthquake is the parent node of the alarm and directly affecting the probability of alarm's going off, but David and Sophia's calls depend on alarm probability.
* The network is representing that our assumptions do not directly perceive the burglary and also do not notice the minor earthquake, and they also not confer before calling.
* The conditional distributions for each node are given as conditional probabilities table or CPT.
* Each row in the CPT must be sum to 1 because all the entries in the table represent an exhaustive set of cases for the variable.
* In CPT, a boolean variable with k boolean parents contains 2K probabilities. Hence, if there are two parents, then CPT will contain 4 probability values

**List of all events occurring in this network:**

* **Burglary (B)**
* **Earthquake(E)**
* **Alarm(A)**
* **David Calls(D)**
* **Sophia calls(S)**

We can write the events of problem statement in the form of probability: **P[D, S, A, B, E]**, can rewrite the above probability statement using joint probability distribution:

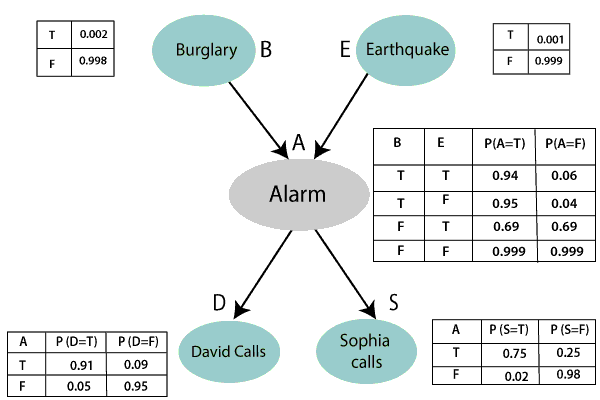
**P[D, S, A, B, E]= P[D | S, A, B, E]. P[S, A, B, E]**

**=P[D | S, A, B, E]. P[S | A, B, E]. P[A, B, E]**

**= P [D| A]. P [ S| A, B, E]. P[ A, B, E]**

**= P[D | A]. P[ S | A]. P[A| B, E]. P[B, E]**

**= P[D | A ]. P[S | A]. P[A| B, E]. P[B |E]. P[E]**



Let's take the observed probability for the Burglary and earthquake component:

P(B= True) = 0.002, which is the probability of burglary.

P(B= False)= 0.998, which is the probability of no burglary.

P(E= True)= 0.001, which is the probability of a minor earthquake

P(E= False)= 0.999, Which is the probability that an earthquake not occurred.

We can provide the conditional probabilities as per the below tables:

**Conditional probability table for Alarm A:**

The Conditional probability of Alarm A depends on Burglar and earthquake:

| **B** | **E** | **P(A= True)** | **P(A= False)** |
| --- | --- | --- | --- |
| True | True | 0.94 | 0.06 |
| True | False | 0.95 | 0.04 |
| False | True | 0.31 | 0.69 |
| False | False | 0.001 | 0.999 |

**Conditional probability table for David Calls:**

The Conditional probability of David that he will call depends on the probability of Alarm.

| **A** | **P(D= True)** | **P(D= False)** |
| --- | --- | --- |
| True | 0.91 | 0.09 |
| False | 0.05 | 0.95 |

**Conditional probability table for Sophia Calls:**

The Conditional probability of Sophia that she calls is depending on its Parent Node "Alarm."

| **A** | **P(S= True)** | **P(S= False)** |
| --- | --- | --- |
| True | 0.75 | 0.25 |
| False | 0.02 | 0.98 |

From the formula of joint distribution, we can write the problem statement in the form of probability distribution:

**P(S, D, A, ¬B, ¬E) = P (S|A) \*P (D|A)\*P (A|¬B ^ ¬E) \*P (¬B) \*P (¬E).**

= 0.75\* 0.91\* 0.001\* 0.998\*0.999

**= 0.00068045.**

**Hence, a Bayesian network can answer any query about the domain by using Joint distribution.**

**The semantics of Bayesian Network:**

There are two ways to understand the semantics of the Bayesian network, which is given below:

**1. To understand the network as the representation of the Joint probability distribution.**

It is helpful to understand how to construct the network.

**2. To understand the network as an encoding of a collection of conditional independence statements.**

It is helpful in designing inference procedure.

**Learning Objectives:**

To create a Bayesian Network for the given problem statement and draw inference from it.

**Conclusion/Learning outcome:**

A Bayesian Network is created for the given problem statement.

EXPERIMENT NO. 10

**Aim:** Design a prototype of an Expert system.

**Prior Concepts:**

Explain the concept of An Expert system

**New Concepts:**

Basic prototype of an expert system in Python

**Python Code:**

class Rule:

def \_\_init\_\_(self, condition, action):

self.condition = condition

self.action = action

class ExpertSystem:

def \_\_init\_\_(self, rules):

self.rules = rules

def infer(self, inputs):

for rule in self.rules:

if self.check\_conditions(rule.condition, inputs):

return rule.action

return "Unable to infer"

def check\_conditions(self, conditions, inputs):

for condition in conditions:

if condition not in inputs:

return False

return True

# Example usage

rules = [

Rule(['Fever', 'Cough'], 'Diagnose: Common Cold'),

Rule(['Fever', 'Sore Throat'], 'Diagnose: Influenza'),

Rule(['Rash'], 'Diagnose: Allergic Reaction'),

Rule(['Headache', 'Fatigue'], 'Diagnose: Migraine')

]

expert\_system = ExpertSystem(rules)

# Test cases

case1 = ['Fever', 'Cough']

case2 = ['Rash', 'Itching']

case3 = ['Headache', 'Fatigue']

print("Test Case 1:", expert\_system.infer(case1))

print("Test Case 2:", expert\_system.infer(case2))

print("Test Case 3:", expert\_system.infer(case3))

In this example, we have an Expert System class that takes a set of rules as input. Each rule consists of a condition (a list of symptoms or inputs) and an action (a diagnosis or output).

The **infer** method of the Expert System class takes a set of inputs and iterates through the rules to find the first rule whose conditions match the inputs. It returns the corresponding action (diagnosis) of that rule. If no rule matches the inputs, it returns "Unable to infer".

The **check\_conditions** method checks if a given set of conditions is satisfied by the provided inputs. It iterates through each condition and checks if it is present in the inputs.

In the example usage, we create an instance of the Expert System class with a set of rules. We then test the system with different input cases and print the inferred diagnosis.

You can extend this prototype by adding more rules, refining the conditions and actions, and incorporating additional functionality such as asking questions to gather inputs from the user, handling uncertainties, or providing explanations for the diagnosis.

Please note that this is a simplified prototype, and real-world expert systems can be much more complex, involving extensive knowledge bases, advanced inference mechanisms, and user interfaces.

**Learning Objectives:**

To design a prototype of an Expert system.

**Conclusion/Learning outcome:**

An expert system for a diagnosis of decease from the given symptoms is designed.

EXPERIMENT NO. 11

**Aim:** Write a program to solve water jug problem.

**New Concepts:**

**Water Jug Problem:**

The water jug problem is a classic puzzle that involves two jugs of different capacities and the goal of measuring out a specific amount of water using these jugs. The problem is often stated as follows:

You have two jugs:

Jug A, which can hold a certain number of liters (capacity A)

Jug B, which can hold a certain number of liters (capacity B)

You also have access to a water source. The goal is to measure out a specific amount of water (target liters) using these two jugs. The operations you can perform are:

1. Fill a jug from the water source.
2. Empty the contents of a jug onto the ground.
3. Pour the contents of one jug into another until either the pouring jug is empty or the receiving jug is full.

The problem is typically framed as finding a sequence of these operations that result in one of the jugs containing the desired amount of water.

To solve the water jug problem, you can use various algorithms, such as breadth-first search, depth-first search, or even mathematical techniques like the Bezout's identity. The specific approach you choose will depend on your programming environment and requirements.

Here's a general outline of how you can approach solving the water jug problem programmatically.

1. Define the capacities of the two jugs (jug A and jug B) and the target amount of water you want to measure.
2. Create a data structure to represent the state of the jugs (e.g., a tuple or a custom class). The state should include the current amount of water in each jug.
3. Implement a search algorithm (e.g., breadth-first search) to explore possible states of the jugs while keeping track of visited states to avoid infinite loops.
4. Start with an initial state where both jugs are empty, and apply the allowable operations (fill, empty, pour) to generate new states.
5. Keep searching until you find a state where one of the jugs contains the target amount of water, or until you exhaust all possible states.
6. If you find a solution, backtrack to reconstruct the sequence of operations that led to the solution.

**Python Code:**

def water\_jug\_problem(jug1\_capacity, jug2\_capacity, target):

def dfs(jug1, jug2, path):

if jug1 == target or jug2 == target:

print("Solution found:", path)

return

if (jug1, jug2) in visited\_states:

return

visited\_states.add((jug1, jug2))

# Fill jug 1

if jug1 < jug1\_capacity:

dfs(jug1\_capacity, jug2, path + "Fill jug 1\n")

# Fill jug 2

if jug2 < jug2\_capacity:

dfs(jug1, jug2\_capacity, path + "Fill jug 2\n")

# Empty jug 1

if jug1 > 0:

dfs(0, jug2, path + "Empty jug 1\n")

# Empty jug 2

if jug2 > 0:

dfs(jug1, 0, path + "Empty jug 2\n")

# Pour from jug 1 to jug 2

if jug1 > 0 and jug2 < jug2\_capacity:

pour\_amount = min(jug1, jug2\_capacity - jug2)

dfs(jug1 - pour\_amount, jug2 + pour\_amount, path + "Pour jug 1 to jug 2\n")

# Pour from jug 2 to jug 1

if jug2 > 0 and jug1 < jug1\_capacity:

pour\_amount = min(jug2, jug1\_capacity - jug1)

dfs(jug1 + pour\_amount, jug2 - pour\_amount, path + "Pour jug 2 to jug 1\n")

visited\_states = set()

dfs(0, 0, "")

# Example usage:

jug1\_capacity = 4

jug2\_capacity = 3

target\_amount = 2

water\_jug\_problem(jug1\_capacity, jug2\_capacity, target\_amount)

**Output:**

Solution found: Fill jug 1

Fill jug 2

Empty jug 1

Pour jug 2 to jug 1

Fill jug 2

Pour jug 2 to jug 1

Solution found: Fill jug 1

Pour jug 1 to jug 2

Empty jug 2

Pour jug 1 to jug 2

Fill jug 1

Pour jug 1 to jug 2

**Learning Objectives:**

To understand implementation of water jug problem using different algorithms.

**Conclusion/Learning outcome:**

Solving of a water Jug problem and its implementation is understood.

EXPERIMENT NO. 12

**Aim:** Design an application to simulate number puzzle problem (Sudoku).

**New Concepts:**

Creating a number puzzle simulation application can be a broad concept since there are many different types of number puzzles. Sudoku is a 9x9 grid where we need to fill in the numbers 1 through 9 in such a way that no row, column, or 3x3 sub grid contains the same number twice.

**Python Code:**

def print\_board(board):

for row in board:

print(" ".join(map(str, row)))

def is\_valid\_move(board, row, col, num):

# Check if 'num' is already in the same row or column

for i in range(9):

if board[row][i] == num or board[i][col] == num:

return False

# Check if 'num' is already in the 3x3 subgrid

start\_row, start\_col = 3 \* (row // 3), 3 \* (col // 3)

for i in range(3):

for j in range(3):

if board[start\_row + i][start\_col + j] == num:

return False

return True

def solve\_sudoku(board):

for row in range(9):

for col in range(9):

if board[row][col] == 0: # Empty cell

for num in range(1, 10):

if is\_valid\_move(board, row, col, num):

board[row][col] = num # Try placing 'num' in the cell

if solve\_sudoku(board):

return True # If it leads to a solution, return True

board[row][col] = 0 # Otherwise, backtrack by setting the cell to 0

return False # If no valid number can be placed, backtrack

return True # All cells are filled (puzzle is solved)

if \_\_name\_\_ == "\_\_main\_\_":

sudoku\_board = [

[5, 3, 0, 0, 7, 0, 0, 0, 0],

[6, 0, 0, 1, 9, 5, 0, 0, 0],

[0, 9, 8, 0, 0, 0, 0, 6, 0],

[8, 0, 0, 0, 6, 0, 0, 0, 3],

[4, 0, 0, 8, 0, 3, 0, 0, 1],

[7, 0, 0, 0, 2, 0, 0, 0, 6],

[0, 6, 0, 0, 0, 0, 2, 8, 0],

[0, 0, 0, 4, 1, 9, 0, 0, 5],

[0, 0, 0, 0, 8, 0, 0, 7, 9]

]

print("Sudoku Puzzle:")

print\_board(sudoku\_board)

if solve\_sudoku(sudoku\_board):

print("\nSudoku Solved:")

print\_board(sudoku\_board)

else:

print("\nNo solution found.")

**Output:**

Sudoku Puzzle:

5 3 0 0 7 0 0 0 0

6 0 0 1 9 5 0 0 0

0 9 8 0 0 0 0 6 0

8 0 0 0 6 0 0 0 3

4 0 0 8 0 3 0 0 1

7 0 0 0 2 0 0 0 6

0 6 0 0 0 0 2 8 0

0 0 0 4 1 9 0 0 5

0 0 0 0 8 0 0 7 9

Sudoku Solved:

5 3 4 6 7 8 9 1 2

6 7 2 1 9 5 3 4 8

1 9 8 3 4 2 5 6 7

8 5 9 7 6 1 4 2 3

4 2 6 8 5 3 7 9 1

7 1 3 9 2 4 8 5 6

9 6 1 5 3 7 2 8 4

2 8 7 4 1 9 6 3 5

3 4 5 2 8 6 1 7 9

**Learning Objectives:**

To design an application to simulate number puzzle problem like Sudoku

**Conclusion/Learning outcome:**

Simulation of Sudoku problem has been done using backtracking algorithm and the given solution is verified.