**ASSIGNMENT 01:**

**WATER JUG PROBLEM:**

**Problem Description:**

The water jug problem involves finding a sequence of actions to measure out the specific amount of water using two jugs of known capacities. In this case, we have Jug-A with capacity of 3 liters and Jug-B with a capacity of 5 liters, and the goal is to measure out 4 liters of water

**Algorithm:**

STEP 1: Set jug\_a and jug\_b to 0 initially.

STEP 2: Start a while loop until either jug\_a or jug\_b reaches the goal amount.

STEP 3: If jug\_a is empty (jug\_a == 0):

Fill jug\_a to its capacity (jug\_a = cap\_a).

STEP 4: Else if jug\_b is full (jug\_b == cap\_b):

Empty jug\_b (jug\_b = 0).

STEP 5: Otherwise:

Calculate the amount that can be poured from jug\_a to jug\_b without overflowing jug\_b.

Pour the water from jug\_a to jug\_b by updating both jug\_a and jug\_b.

STEP 6: Return the final amounts in jug\_a and jug\_b once the goal is achieved or the loop exits.

**Source Code:**

def water\_jug(cap\_a, cap\_b, goal):

    jug\_a,jug\_b=0, 0    #Initializing jug\_a and jug\_b to 0

    while jug\_a != goal and jug\_b != goal:

        if jug\_a==0:

            jug\_a=cap\_a     #Filling jug\_a to its capacity

        elif jug\_b==cap\_b:

            jug\_b=0     #Making jug\_b empty

        else:

            transfer=min(jug\_a,cap\_b-jug\_b)     #Pouring the water

            jug\_a-=transfer

            jug\_b+=transfer

    return jug\_a,jug\_b

capacity\_a=3

capacity\_b=5

goal\_amount=4

final\_state=water\_jug(capacity\_a,capacity\_b,goal\_amount)

print("Final state of jug\_a and jug\_b :", final\_state)

**Output:**

Final state of jug\_a and jug\_b : (0, 4)

**ASSIGNMENT 02:**

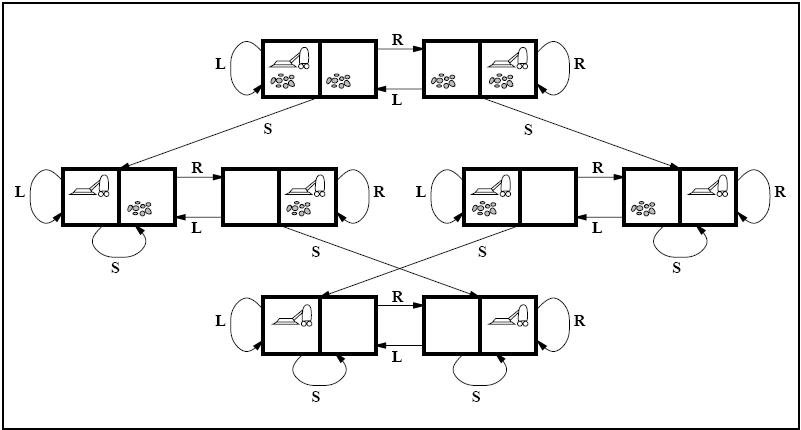
**VACUUM CLEANER PROBLEM:**

**Problem Description:**

Here, the main goal is to design a simple intelligent agent (Simple reflex agent) for a vacuum cleaner that autonomously navigates and clean a 2D grid-based environment containing dirt or dust. The agent operates in a simple world with two possible states (State 7 and State 8) and dirt being the only percept. The agent should maximize cleanliness by efficiently navigating through the environment.

**PAGE description:**

* PERCEPTS: The agent perceives the current location’s state (dirt or clean).
* ACTIONS: The agent can only perform two actions:
  1. Suck: To clean the current location
  2. Move: To move between location (left or right).
* GOAL: The goal of the agent is to reach either state7 or state 8 as shown in the below graph
* ENVIRONMENT: The environment id 2D grid where the agent move between two locations (left or right grid) and clean the dirt or dust if present.

**State space graph:**

**Source Code:**

import random

class Environment:

    def \_\_init\_\_(self):

        # Initialize locations A and B as either clean (0) or dirty (1)

        self.location\_condition = {'A': random.randint(0, 1), 'B': random.randint(0, 1)}

class SimpleReflexAgent:

    def \_\_init\_\_(self, env):

        self.environment = env

        self.clean\_location()

    def clean\_location(self):

        for location, condition in self.environment.location\_condition.items():

            print(f"Vacuum is at location {location}")

            if condition == 1:  # If location is dirty

                print(f"Location {location} is dirty.")

                self.environment.location\_condition[location] = 0  # Clean the location

                print(f"Location {location} has been cleaned.")

            else:

                print(f"Location {location} is already clean.")

# Create an instance of Environment

the\_environment = Environment()

# Create an instance of SimpleReflexAgent and pass the environment instance

the\_vacuum = SimpleReflexAgent(the\_environment)

**Output:**

Vacuum is at location A

Location A is dirty.

Location A has been cleaned.

Vacuum is at location B

Location B is dirty.

Location B has been cleaned.

Vacuum is at location A

Location A is already clean.

Vacuum is at location B

Location B is dirty.

Location B has been cleaned.

Vacuum is at location A

Location A is already clean.

Vacuum is at location B

Location B is already clean.

**ASSIGNMENT 03:**

**Implementing Breadth First Search (BFS):**

**Problem Description:**

The given transportation network represented as graph where nodes/state represent cities and edges represented the road link between the cities. Here, we have to find the path from starting city to destination city, if there exist using breadth-first-search algorithm.

Given,

Starting city: Biratnagar

Destination city: Ilam

**Algorithm:**

STEP 1: ENQUEUE THE STARTING NODE

The first step is to enqueue the starting node into a queue data structure. This node is marked as visited so that it’s not revisited later.

STEP 2: DEQUEUE A NODE AND MARK IT AS VISITED

In this step, we dequeue a node from the queue and mark it as visited. We then process this node.

STEP 3: ENQUEUE ALL ADJACENT NODES OF THE DEQUEUED NODE THAT ARE NOT YET VISITED

We then enqueue all the adjacent nodes of the dequeued node that have not been visited yet. These nodes are added to the end of the queue.

STEP 4: REPEAT STEPS 2-3 UNTIL THE QUEUE IS EMPTY

**Source Code:**

from collections import deque

def bfs(graph, start, destination):

    if start not in graph or destination not in graph:

        return "Starting or destination city not found in the graph"

    visited = set()

    queue = deque([(start, [start])])

    while queue:

        current, path = queue.popleft()

        visited.add(current)

        if current == destination:

            return path

        for neighbor in graph[current]:

            if neighbor not in visited:

                queue.append((neighbor, path + [neighbor]))

    return "Path from {} to {} does not exist".format(start, destination)

graph = {

    'Biratnagar': ['Itahari','Rangeli'],

    'Itahari': ['Belbari', 'Dhankuta'],

    'Dhankuta': ['Ilam','Bhojpur','Chainpur'],

    'Rangeli': ['Urlabari'],

    'Belbari': ['Pathari'],

    'Pathari': ['Urlabari'],

    'Urlabari': ['Gauradaha','Shivasatakshi'],

    'Shivasatakshi': ['Gauradaha','Suryodaya'],

    'Suryodaya': ['Ilam'],

    'Chainpur': ['Ilam','Khandbari'],

    'Ilam':[]

}

start\_city = 'Biratnagar'

destination\_city = 'Ilam'

result = bfs(graph, start\_city, destination\_city)

print("Path from {} to {}: {}".format(start\_city, destination\_city, result))

**Output:**

Path from Biratnagar to Ilam: ['Biratnagar', 'Itahari', 'Dhankuta', 'Ilam']

**ASSIGNMENT 04:**

**Implementing Depth First Search (DFS):**

**Problem Description:**

Suppose you have a family hierarchy as :

Great\_grand \_parents (root node)

Grand parents (Great\_grand \_parents child)

Parents (Grand parents child)

You (Parents child)

Uncle ( Grand parents child)

Parents\_Uncle (Great\_grand \_parents child)

XYZ (Parents\_Uncle child)

ABC (Parents\_Uncle child)

Now based on your family hierarchy performs a depth-first search and return the path from node Great\_grand\_parents (root node) to node You (Parents\_child).

**Algorithm:**

STEP 1: Start DFS from the Initial Node

Push the starting node onto a stack data structure.

Mark the starting node as visited.

STEP 2: Pop a Node from the Stack and Process It

Pop a node from the stack.

Process (e.g., print or store) the value of the popped node.

STEP 3: Push Unvisited Adjacent Nodes onto the Stack

For each adjacent node of the popped node that has not been visited:

Push the unvisited adjacent node onto the stack.

Mark the node as visited.

STEP 4: Repeat Steps 2-3 Until the Stack Is Empty

Repeat Steps 2 and 3 until the stack becomes empty.

**Source Code:**

class FamilyNode:

    def \_\_init\_\_(self, name):

        self.name = name

        self.children = []

def dfs\_path(node, target, path=None):

    if path is None:

        path = []

    path.append(node.name)

    if node.name == target:

        return path

    for child in node.children:

        result = dfs\_path(child, target, path.copy())

        if result:

            return result

    return None

# Creating the family hierarchy

great\_grandparents = FamilyNode("Meharman and Maya")

grandparents = FamilyNode("Krishna and Pabitra")

parents = FamilyNode("Bhim and Hemanta")

you = FamilyNode("Aakash")

uncle = FamilyNode("Lokendra")

parents\_uncle = FamilyNode("Ram and Laxmi")

xyz = FamilyNode("Resham")

abc = FamilyNode("Khagendra")

# Building the family tree

great\_grandparents.children = [grandparents, parents\_uncle]

grandparents.children = [parents, uncle]

parents.children = [you]

uncle.children = []

parents\_uncle.children = [xyz, abc]

xyz.children = []

abc.children = []

# Perform DFS and get the path

start\_node = great\_grandparents

target\_node = "Aakash"

result\_path = dfs\_path(start\_node, target\_node)

# Display the result

if result\_path:

    print("Path from {} to {}: {}".format(start\_node.name, target\_node, " -> ".join(result\_path)))

else:

    print("Node {} not found in the family hierarchy.".format(target\_node))

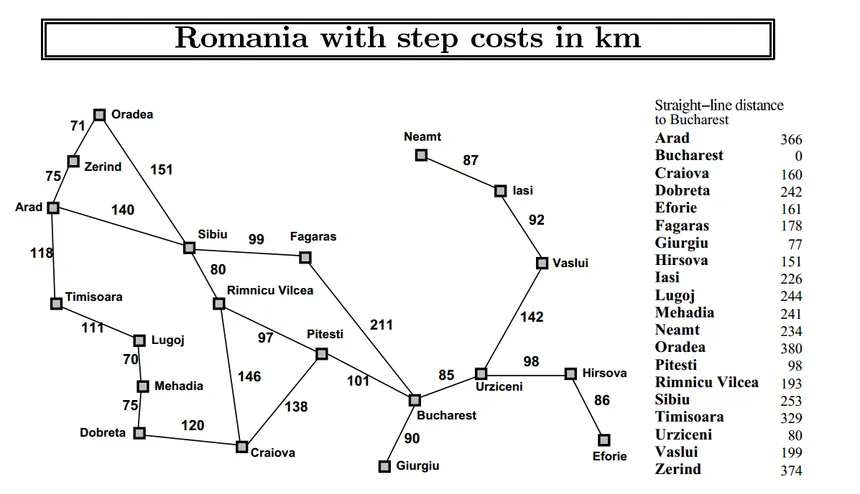
**Output:**

Path from Meharman and Maya to Aakash: Meharman and Maya -> Krishna and Pabitra -> Bhim and Hemanta -> Aakash

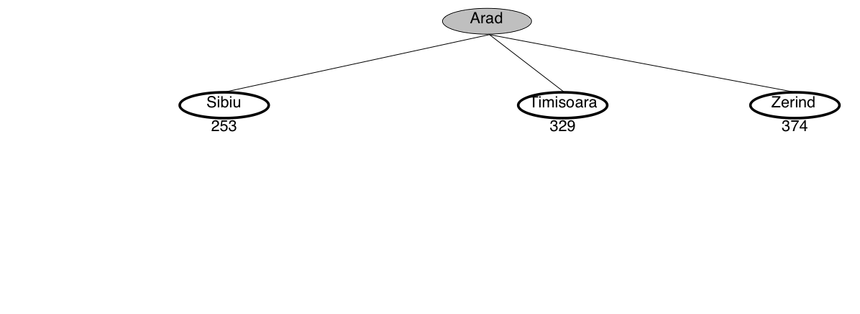
**ASSIGNMENT 05:**

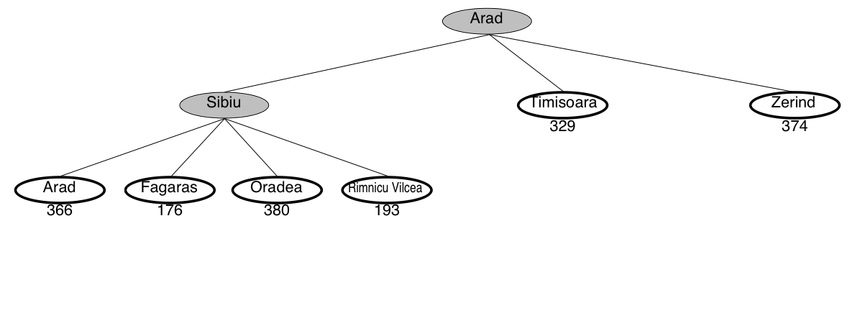
**Implementing Greedy Best-First Search:**

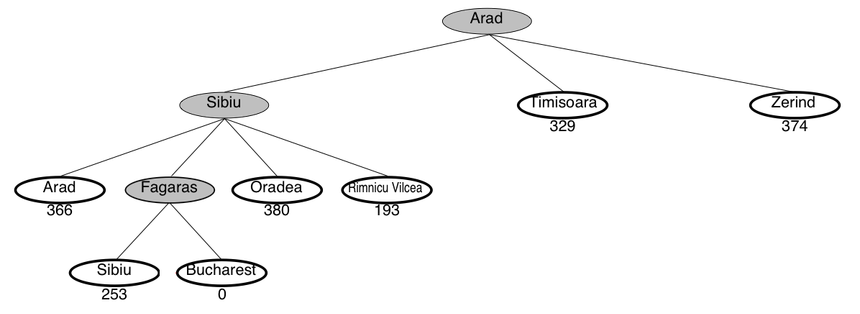
**Problem Description:**

The following search problem represented by a graph, where each node in the graph represents a state, and each edge represents a possible transition between states. Here, the main goal is to find the path from the initial state(Arad city) to reach the goal state (Bucharest) using the Greedy Best-First Search algorithm. The search algorithm utilizes a heuristic function to estimate the cost from each state to the goal. 

**Grap Representation:**







**Algorithm:**

STEP 1: Enqueue the starting node into a priority queue based on a heuristic value and mark the starting node as visited.

STEP 2: Dequeue a node from the priority queue. This node is the one with the lowest heuristic value and mark the dequeued node as visited.

STEP 3: Process the dequeued node (perform any necessary operations or checks).

STEP 4: Enqueue all adjacent nodes of the dequeued node that have not been visited yet into the priority queue. The priority is determined by the heuristic value, where lower values are prioritized.

STEP 5: Repeat Steps 2 & 3 until the priority queue is empty or the destination node is dequeued.

**Source Code:**

import heapq

graph = {

    'Arad': {'Zerind': 75, 'Sibiu': 140, 'Timisoara': 118},

    'Zerind': {'Arad': 75, 'Oradea': 71},

    'Oradea': {'Zerind': 71, 'Sibiu': 151},

    'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},

    'Timisoara': {'Arad': 118, 'Lugoj': 111},

    'Lugoj': {'Timisoara': 111, 'Mehadia': 70},

    'Mehadia': {'Lugoj': 70, 'Drobeta': 75},

    'Drobeta': {'Mehadia': 75, 'Craiova': 120},

    'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},

    'Rimnicu Vilcea': {'Sibiu': 80, 'Craiova': 146, 'Pitesti': 97},

    'Fagaras': {'Sibiu': 99, 'Bucharest': 211},

    'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},

    'Bucharest': {'Fagaras': 211, 'Pitesti': 101}

}

heuristic = {'Arad': 366, 'Zerind': 374, 'Oradea': 380, 'Sibiu': 253, 'Timisoara': 329,

             'Lugoj': 244, 'Mehadia': 241, 'Drobeta': 242, 'Craiova': 160, 'Rimnicu Vilcea': 193,

             'Fagaras': 176, 'Pitesti': 100, 'Bucharest': 0}

def greedy\_best\_first\_search(graph, start, goal):

    visited, pq = set(), [(heuristic[start], start, [])]

    while pq:

        \_, current, path = heapq.heappop(pq)

        if current in visited:

            continue

        visited.add(current)

        path = path + [current]

        if current == goal:

            return path

        for neighbor, \_ in graph[current].items():

            if neighbor not in visited:

                heapq.heappush(pq, (heuristic[neighbor], neighbor, path))

    return None

# Example: Find the path from Arad to Bucharest

start, goal = 'Arad', 'Bucharest'

result\_path = greedy\_best\_first\_search(graph, start, goal)

print(f"Path from {start} to {goal}: {result\_path}" if result\_path else f"No path found from {start} to {goal}")

**Output:**

Path from Arad to Bucharest: ['Arad', 'Sibiu', 'Fagaras', 'Bucharest']

**ASSIGNMENT 06:**

**Implementing Knowledge Representation System (Semantic Network):**

**Introduction:**

Semantic networks are alternative of predicate logic for knowledge representation. In Semantic networks, we can represent our knowledge in the form of graphical networks. This network consists of nodes representing objects and arcs which describe the relationship between those objects. Semantic networks can categorize the object in different forms and can also link those objects. Semantic networks are easy to understand and can be easily extended.

This representation consist of mainly two types of relations:

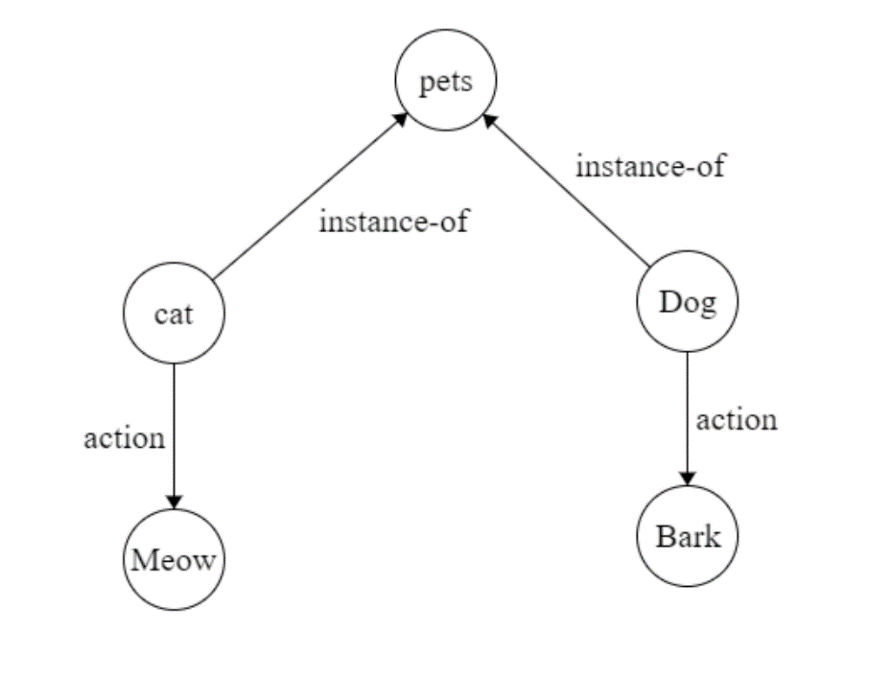
IS-A relation (Inheritance)

Kind-of-relation

**Problem Description:**

The task is to develop a simple Knowledge Representation System (KRS) that utilizes a semantic network paradigm. The system should allow users to define entities, attributes, and relationships between entities. The given facts are:

1. Cats meow.
2. Dogs bark.
3. Cats can be pets.
4. Dogs can be pets.

**Graph Representation:**

**Source Code:**

class SemanticNetwork:

    def \_\_init\_\_(self):

        self.nodes = {}

    def add\_relation(self, node1, node2):

        if node1 not in self.nodes:

            self.nodes[node1] = []

        if node2 not in self.nodes:

            self.nodes[node2] = []

        self.nodes[node1].append(node2)

        self.nodes[node2].append(node1)

    def print\_network(self):

        for node, neighbors in self.nodes.items():

            neighbors\_str = ' '.join(neighbors)

            print(f"{node}: {neighbors\_str}")

# Example usage:

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

    network = SemanticNetwork()

    # Adding relations

    network.add\_relation("cat", "meow")

    network.add\_relation("dog", "bark")

    network.add\_relation("cat", "pet")

    network.add\_relation("dog", "pet")

    # Printing the semantic network

    network.print\_network()

**Output:**

cat: meow pet

meow: cat

dog: bark pet

bark: dog

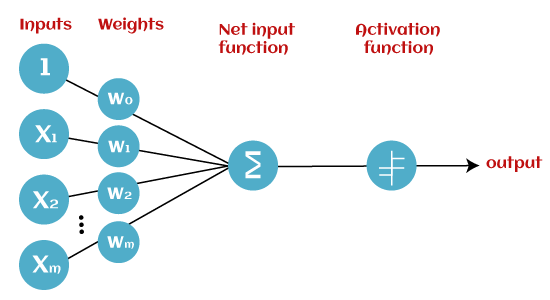
pet: cat dog

**ASSIGNMENT 08:**

**Implementing Perceptron Learning Algorithm:**

**Introduction:**

Artificial neural network is non-linear, parallel, distributed, and highly connected network having capabilities of adaptability, self-organization, fault tolerance, and Very Large Scale Integration (VLSI) implementation, which closely resembles with physical nervous system.

Perceptron Learning algorithm was originally developed by Frank Rosenblatt in late 1950s. Training pattern are presented to the network’s input; the output is computed. Then the connection weights wj are modified by an amount that is proportional to the product of difference between the actual output, y, and the desired output,d ,and the input pattern,x.

**Algorithm:**

**Step-1:**

In the first step first, multiply all input values with corresponding weight values and then add them to determine the weighted sum. Mathematically, we can calculate the weighted sum as follows:

∑wi\*xi = x1\*w1 + x2\*w2 +…wn\*xn

Add a special term called **bias 'b'** to this weighted sum to improve the model's performance.

**∑wi\*xi + b**

**Step-2:**

In the second step, an activation function is applied with the above-mentioned weighted sum, which gives us output either in binary form or a continuous value as follows:

**Y = f(∑wi\*xi + b)**

**Problem Description:**

Implement a NAND logic function using the Perceptron model. The NAND logic function returns 1 only if both inputs are 0; otherwise, it returns 0. We'll use the Perceptron model, which consists of weights (w) and a bias (b), along with the unit step function to determine the output.

**Source Code:**

import numpy as np

# define Unit Step Function

def unitStep(v):

    if v >= 0:

        return 1

    else:

        return 0

# design Perceptron Model

def perceptronModel(x, w, b):

    v = np.dot(w, x) + b

    y = unitStep(v)

    return y

# NOT Logic Function

# wNOT = -1, bNOT = 0.5

def NOT\_logicFunction(x):

    wNOT = -1

    bNOT = 0.5

    return perceptronModel(x, wNOT, bNOT)

# AND Logic Function

# w1 = 1, w2 = 1, bAND = -1.5

def AND\_logicFunction(x):

    w = np.array([1, 1])

    bAND = -1.5

    return perceptronModel(x, w, bAND)

# NAND Logic Function with AND and NOT function calls in sequence

def NAND\_logicFunction(x):

    output\_AND = AND\_logicFunction(x)

    output\_NOT = NOT\_logicFunction(output\_AND)

    return output\_NOT

if \_\_name\_\_ == '\_\_main\_\_':

    x, y = input("Enter the input x, y: ").split(' ')

    x = int(x)

    y = int(y)

    print(f"NAND({x}, {y}) = {NAND\_logicFunction(np.array([x, y]))}")

**Output:**

Enter the input x, y: 0 0

NAND(0, 0) = 1

Enter the input x, y: 0 1

NAND(0, 1) = 1

Enter the input x, y: 1 0

NAND(1, 0) = 1

Enter the input x, y: 1 1

NAND(1, 1) = 0