# LAB 1

**Title: WAP to traverse a graph using BFS. Breadth First Search:**

A BFS, or Breadth-First Search, is a graph traversal algorithm that explores all the vertices of a graph in breadth-first order. It starts at a given source vertex, visits all its neighboring vertices, then visits the neighbors of those vertices, and so on, until all vertices have been visited. This algorithm uses a queue data structure to maintain the order of exploration.

# Algorithm:

1. Create a queue data structure and initialize it with the source vertex.
2. Create a visited array or set to keep track of visited vertices.
3. Mark the source vertex as visited.
4. While the queue is not empty, repeat steps 5-8.
5. Dequeue a vertex from the front of the queue.
6. Process the dequeued vertex (e.g., print its value or perform required operations).
7. Enqueue all unvisited neighboring vertices of the dequeued vertex.
8. Mark each newly enqueued vertex as visited and enqueue it into the queue.
9. If there are no more vertices in the queue, the BFS traversal is complete.

# Tools: Colab

# Language: Python

**Source Code:**

from queue import Queue adj\_list = {

"A" : ["B", "D"],

"B" : ["A", "C"],

"C" : [“B”],

"D" : [“A”,"E", "F"],

"E" : [“D”,"F", "G"],

"F" : [“D”,"E", "H"],

"G" : ["E", "H"],

"H" : ["G", "F"]

}

visited = {} level = {} parent = {}

bfs\_traversal\_op = [] queue = Queue()

for node in adj\_list.keys(): visited[node] = False level[node] = -1 parent[node] = None

s = "A"

visited[s] = True

level[s] = 0 queue.put(s)

while not queue.empty(): u = queue.get()

bfs\_traversal\_op.append(u) for v in adj\_list[u]:

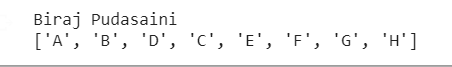
if not visited[v]: visited[v] = True parent[v] = u

level[v] = level[u] + 1 queue.put(v)

print(“Biraj Pudasaini\n”)

print(bfs\_traversal\_op)

# Output:

****

**Conclusion:**

The algorithm guarantees that each vertex is visited exactly once and explores the graph in a level- by-level manner, starting from the source vertex and moving outward.

# LAB 2

**Title: WAP to search goal node using BFS. Breadth First Search:**

A BFS, or Breadth-First Search, is a graph traversal algorithm that explores all the vertices of a graph in breadth-first order. It starts at a given source vertex, visits all its neighboring vertices, then visits the neighbors of those vertices, and so on, until all vertices have been visited. This algorithm uses a queue data structure to maintain the order of exploration.

# Algorithm:

1. Create an empty queue and a set to track visited nodes.
2. Enqueue the initial node (start) into the queue.
3. While the queue is not empty:
   * Dequeue the first node (current) from the queue.
   * If the current node is the goal node, return it as the result.
   * Mark the current node as visited by adding it to the visited set.
   * Explore all the neighbors of the current node:
     + For each neighbor, check if it has not been visited and is not already in the queue.
     + If both conditions are met, enqueue the neighbor into the queue.
4. If the goal node is not found after exploring all nodes, return "goal node not found."

# Tools: Visual Studio Code Language: Python

**Source Code:**

from queue import Queue

# Adjacency list representation of the graph

adj\_list = {

    "A": ["B", "D"],

    "B": ["A", "C"],

    "C": ["B"],

    "D": ["A", "E", "F"],

    "E": ["D", "F", "G"],

    "F": ["D", "E", "H"],

    "G": ["H", "E"],

    "H": ["F", "G"],

}

visited = {}

level = {}

parent = {}

bfs\_traversal\_op = []

queue = Queue()

# Initialization

for node in adj\_list.keys():

    visited[node] = False

    parent[node] = None

    level[node] = -1

S = "A"  # Starting node

target\_node = "D"  # Target node

visited[S] = True

level[S] = 0

queue.put(S)

while not queue.empty():

    u = queue.get()

    bfs\_traversal\_op.append(u)

    if u == target\_node:

        break

    for V in adj\_list[u]:

        if not visited[V]:

            visited[V] = True

            parent[V] = u

            level[V] = level[u] + 1

            queue.put(V)

print("Biraj Pudasaini")

print(bfs\_traversal\_op)

if target\_node in visited:

    path = [target\_node]

    while parent[target\_node] is not None:

        path.insert(0, parent[target\_node])

        target\_node = parent[target\_node]

    print("BFS Traversal Path:")

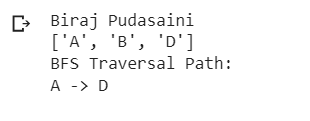
    print(" -> ".join(path))

else:

    print("Target node not found in the graph.")

# Output:

Goal node is D



# Conclusion:

The algorithm guarantees that each vertex is visited exactly once and explores the graph in a level- by-level manner, starting from the source vertex and moving outward, stop when goal was found.

# LAB 3

**Title: WAP to traverse through a graph using DFS. Depth First Search:**

DFS, or Depth-First Search, is a graph traversal algorithm that explores all the vertices of a graph in depth-first order. It starts at a given source vertex, explores as far as possible along each branch before backtracking, and uses a stack to keep track of the vertices being explored. DFS is commonly used for tasks such as finding connected components, detecting cycles, or exploring all reachable vertices from a given source.

# Algorithm:

1. Create a stack data structure and initialize it with the source vertex.
2. Create a visited array or set to keep track of visited vertices.
3. Mark the source vertex as visited.
4. While the stack is not empty, repeat steps 5-8.
5. Pop a vertex from the top of the stack.
6. Process the popped vertex (e.g., print its value or perform required operations).
7. Push all unvisited neighboring vertices of the popped vertex onto the stack.
8. Mark each newly pushed vertex as visited.
9. If there are no more vertices in the stack, the DFS traversal is complete.

# Tools: Visual Studio Code Language: Python

**Source Code:**

from queue import Queue

adj\_list = {

    "A": ["B", "D"],

    "B": ["A", "C"],

    "C": ["B"],

    "D": ["A", "E", "F"],

    "E": ["D", "F", "G"],

    "F": ["D", "E", "H"],

    "G": ["E", "H"],

    "H": ["G", "F"]

}

state = {}

parent = {}

trav\_step = {}

dfs\_traversal\_op = []

for node in adj\_list.keys():

    state[node] = "IDLE"

    parent[node] = None

    trav\_step[node] = [-1, -1]

step = 0

def dfs\_func(u):

    global step

    state[u] = "Start"

    trav\_step[u][0] = step

    dfs\_traversal\_op.append(u)

    step += 1

    for v in adj\_list[u]:

        if state[v] == "IDLE":

            parent[v] = u

            dfs\_func(v)

    state[u] = "End"

    trav\_step[u][1] = step

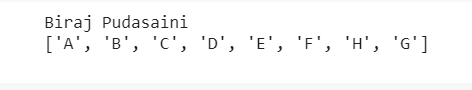
    step += 1

dfs\_func("A")

print("Biraj Pudasaini")

print(dfs\_traversal\_op)

# Output:

****

**Conclusion:**

The algorithm explores the graph in a depth-first manner by visiting the deepest unvisited vertex first, backtracking when necessary to explore other branches. It ensures that each vertex is visited exactly once. DFS is commonly used for tasks such as finding connected components, detecting cycles, or exploring all reachable vertices from a given source.

# LAB 4

**Title: WAP to search goal node using DFS. Depth First Search:**

DFS, or Depth-First Search, is a graph traversal algorithm that explores all the vertices of a graph in depth-first order. It starts at a given source vertex, explores as far as possible along each branch before backtracking, and uses a stack to keep track of the vertices being explored. DFS is commonly used for tasks such as finding connected components, detecting cycles, or exploring all reachable vertices from a given source.

# Algorithm:

* 1. Create an empty stack to hold nodes to be explored. Each element in the stack will be a tuple containing the current node and the path taken to reach that node.
  2. Push the start node along with the path [start\_node] onto the stack.
  3. While the stack is not empty, do the following:
     1. Pop a node and its path from the stack.
     2. If the current node is the goal node, return the path taken to reach it.
     3. Otherwise, explore all unvisited neighbors of the current node:
        1. For each unvisited neighbor, create a new path by appending the neighbor to the path taken so far.
        2. Push the neighbor and the new path onto the stack.
  4. If the loop finishes without finding the goal node, return None to indicate that the goal node is not present in the graph.

# Tools: Visual Studio Code Language: Python

**Source Code:**

from queue import Queue

adj\_list = {

    "A": ["B", "D"],

    "B": ["A", "C"],

    "C": ["B"],

    "D": ["A", "E", "F"],

    "E": ["D", "F", "G"],

    "F": ["D", "E", "H"],

    "G": ["E", "H"],

    "H": ["G", "F"]

}

state = {}

trav\_step = {}

parent = {}

dfs\_traversal\_op = []

target\_node = "D"

print("Biraj Pudasaini ")

for node in adj\_list.keys():

    state[node] = "Idle"

    parent[node] = None

    trav\_step[node] = [-1, -1]

step = 0

def dfs\_func(u):

    global step

    state[u] = "Start"

    trav\_step[u][0] = step

    dfs\_traversal\_op.append(u)

    step += 1

    if u == target\_node:

        print(dfs\_traversal\_op)

    for v in adj\_list[u]:

        if state[v] == "Idle":

            parent[v] = u

            if dfs\_func(v):

                return True

    state[u] = "End"

    trav\_step[u][1] = step

    step += 1

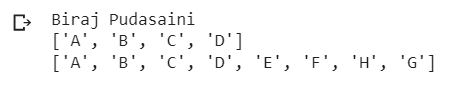
    return False

dfs\_func("A")

print(dfs\_traversal\_op)

# Output:

Goal node is D



# Conclusion:

The algorithm explores the graph in a depth-first manner by visiting the deepest unvisited vertex first, backtracking when necessary to explore other branches. It ensures that each vertex is visited exactly once. DFS is commonly used for tasks such as finding connected components, detecting cycles, or exploring all reachable vertices from a given source and stops when goal node was found.

# LAB 5

**Title: Simulate a knowledge base for listed dataset and perform operations in Prolog. Background:**

A knowledge base typically contains a wide range of information, including facts, rules, procedures, guidelines, instructions, and other structured or unstructured data. It can be designed to cover various domains, such as scientific knowledge, historical data, technical documentation, customer support information, and much more.

Knowledge bases are commonly used in various fields and applications, including customer service and support systems, expert systems, chatbots, natural language processing (NLP) models, search engines, and decision support systems.

**Tools:** SWI-Prolog

# Dataset:

Tiger eats meat.

Cow eats vegetables.

Human eats meat.

Human eats vegetables. Carnivorous eats meat.

Omnivorous eats vegetables and meat.

# Logical Operations:

1. List out the animals that eats meat only.
2. List out the animals that easts vegetables only.
3. List out the animals that eats meat and vegetables.
4. List out the animals that eats meat.
5. List out the animals that eats vegetables.

# Code:

eats(tiger,meat). eats(cow, vegetables). eats(human, meat). eats(human, vegetables).

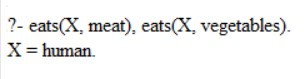
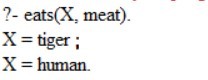
carnivorous(X) :- eats(X, meat).

omnivorous(X):- eats(X, meat), eats(X,vegetables). herbivorous(X):- eats(X, vegetables).

# Output:

1. List out the animals that eats meat only.
2. List out the animals that easts vegetables only.



1. List out the animals that eats meat and vegetables.
2. List out the animals that eats meat.

# Conclusion:

Hence we use the knowledge base for inference of knowledge. Prolog helps for knowledge base simulation.

# LAB 6

**Title: Simulate a knowledge base for listed dataset and perform operations in Prolog. Background:**

A knowledge base typically contains a wide range of information, including facts, rules, procedures, guidelines, instructions, and other structured or unstructured data. It can be designed to cover various domains, such as scientific knowledge, historical data, technical documentation, customer support information, and much more.

Knowledge bases are commonly used in various fields and applications, including customer service and support systems, expert systems, chatbots, natural language processing (NLP) models, search engines, and decision support systems.

**Tools:** SWI-Prolog

# Dataset:

Kheer contains sugar. Haluwa contains sugar.

Pickle contains salt. Daal contains salt.

Sweet\_Dish is a dish which contains sugar. Salt\_Dish is a dish which contains salt.

# Logical Operations:

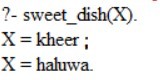
* 1. List of sweet dishes.
  2. List of salt dishes.

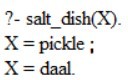
# Code:

contains(kheer, sugar). contains(haluwa, sugar). contains(pickle, salt). contains(daal, salt).

sweet\_dish(A) :- contains(A, sugar). salt\_dish(A) :- contains(A, salt).

# Output:

1. List of sweet dishes.
2. List of salt dishes.



# Conclusion:

Hence, we use the knowledge base for inference of knowledge. Prolog helps for knowledge base simulation.

# LAB 7

**Title: Program to implement neural network for the given dataset. Background:**

A neural network is a computational model inspired by the structure and functioning of the human brain. It is a type of machine learning algorithm that is particularly well-suited for tasks involving pattern recognition, classification, regression, and other complex data processing tasks. Neural networks are capable of learning from data and making predictions based on the patterns they discover.

The basic building block of a neural network is a neuron, which is also referred to as a node or unit. Neurons are organized into layers, forming a network structure. A typical neural network consists of three main types of layers:

* Input Layer: The first layer of the neural network, where the input data is fed. Each neuron in this layer corresponds to a feature or input variable.
* Hidden Layers: Layers that come after the input layer. These layers are responsible for learning and extracting patterns from the data. The number of hidden layers and the number of neurons in each hidden layer can vary depending on the complexity of the problem.
* Output Layer: The final layer of the neural network, responsible for producing the output predictions. The number of neurons in the output layer depends on the nature of the problem. For instance, in binary classification, there will be one neuron, while multi-class classification may have multiple neurons, each representing a class.

**Tools:** Google Collab

**Language:** Python

|  |  |  |  |
| --- | --- | --- | --- |
| **Inputs** | | | **Outputs** |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | ? |

# Source Code:

import numpy as np

from random import choice

print("\t---Neural Network Implementation \n")

print("...Using Single Layered Perceptron.  \n")

np.random.seed(2)

# Activation function (Step function)

activation\_function = lambda x: 0 if x < 0 else 1

# Training data - input patterns and corresponding output labels

training\_data = [

    (np.array([0, 1, 1]), 1),

    (np.array([1, 0, 1]), 1),

    (np.array([0, 0, 1]), 0)

]

# Model parameters

learning\_rate = 0.2

training\_steps = 100

# Initialize weights

W = np.random.rand(3)

# Training the perceptron

for i in range(training\_steps):

    x, y = choice(training\_data)

    l1 = np.dot(W, x)  # Compute dot product between input (X) and weights matrix (W)

    y\_pred = activation\_function(l1)  # Apply the activation function on l1 (output of the network)

    error = y - y\_pred  # Compute the error between the predicted output and the actual output

    # Update the weights based on the error and the learning rate

    update = learning\_rate \* error \* x

    W += update

# Output after training

print("Biraj Pudasaini\n")

print("Predictions after training on the training dataset:")

for x, \_ in training\_data:

    y\_pred = np.dot(x, W)

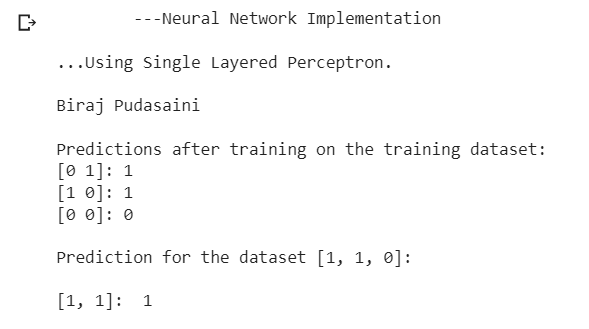
    print("{}: {}".format(x[:2], activation\_function(y\_pred)))

print("\nPrediction for the dataset [1, 1, 0]:\n")

y\_predict = np.dot([1, 1, 0], W)

print("[1, 1]: ", activation\_function(y\_predict))

# Output:

****

**Conclusion:**

Here we demonstrate the neural network by high level language to train the given dataset.

# LAB 8

**Title: Program to implement neural network for the given dataset. Background:**

The perceptron consists of a single layer of neurons (also called perceptron). Each perceptron takes multiple input values (features) and produces a single binary output (either 0 or 1).The inputs are multiplied by their corresponding weights, and the weighted inputs are summed up. A bias term is added to the sum, which helps control the output of the perceptron. The sum (weighted input + bias) is passed through an activation function, typically a step function (e.g., Heaviside step function or sign function).

The training of a perceptron involves adjusting the weights and bias based on the input data and the desired output (target) to correctly classify the inputs. The perceptron uses a learning algorithm called "Perceptron Learning Rule" or "Delta Rule." During training, the perceptron is presented with input data, and its output is compared to the target output (0 or 1).If the output is incorrect, the weights and bias are updated to bring the perceptron closer to the correct output. The learning rule is based on the error between the predicted output and the target output, and it updates the weights and bias accordingly.

**Tools:** Google Collab

**Language:** Python

# Source Code:

import numpy as np

class NeuralNetwork():

    def \_\_init\_\_(self):

        # Seed the random number generator

        np.random.seed(1)

# Set synaptic weights to a 3x1 matrix, with values from -1 to 1 and mean 0

        self.synaptic\_weights = 2 \* np.random.random((3, 1)) - 1 # converted the range of 0 to 1 to -1 to 1

    def sigmoid(self, x):

        """Takes in the weighted sum of the inputs and normalizes them through a sigmoid function."""

        return 1 / (1 + np.exp(-x))

    def sigmoid\_derivative(self, x):

        """The derivative of the sigmoid function used to calculate necessary weight adjustments."""

        return x \* (1 - x)

    def train(self, training\_inputs, training\_outputs, training\_iterations):

        """

        We train the model through trial and error, adjusting the synaptic weights each time to get a better result.

        """

        for iteration in range(training\_iterations):

            # Pass training set through the neural network

            output = self.think(training\_inputs)

            # Calculate the error rate

            error = training\_outputs - output

            # Multiply error by input and gradient of the sigmoid function

            # Less confident weights are adjusted more through the nature of the function

            adjustments = np.dot(training\_inputs.T, error \* self.sigmoid\_derivative(output))

            # Adjust synaptic weights

            self.synaptic\_weights += adjustments

    def think(self, inputs):

        """

        Pass inputs through the neural network to get output.

        """

        inputs = inputs.astype(float)

        output = self.sigmoid(np.dot(inputs, self.synaptic\_weights))

        return output

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

    # Initialize the single-neuron neural network

    neural\_network = NeuralNetwork()

    print("Biraj Pudasaini\n")

    print("Random starting synaptic weights: ")

    print(neural\_network.synaptic\_weights)

    # The training set, with 4 examples consisting of 3 input values and 1 output value

    training\_inputs = np.array([[1, 0, 1],

                                [1, 0, 0],

                                [1, 0, 1],

                                [0, 1, 1]])

    training\_outputs = np.array([[0, 1, 0, 1]]).T

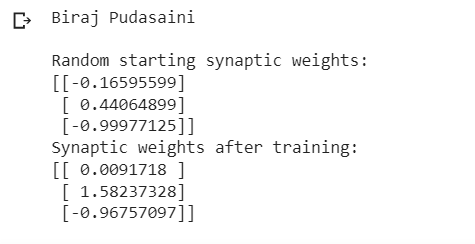
    # Train the neural network

    neural\_network.train(training\_inputs, training\_outputs, 10)

    print("Synaptic weights after training: ")

    print(neural\_network.synaptic\_weights)

# Output:

****

**Conclusion:**

Here we use single layer neural network and sigmoid activation function. The perceptron is limited to solving the linearly separable problem so to handle this multilayer neutral network with nonlinear activation function must be considered.

# LAB 9

**Title: Program to demonstrate Vacuum Cleaner problem. Background:**

The vacuum cleaner problem is a classic example in the field of artificial intelligence and robotics. It is used to illustrate how an agent (such as a robotic vacuum cleaner) can intelligently navigate an environment to achieve a specific goal (cleaning a room or area). The problem can be formulated in various ways, but the basic objective is to efficiently clean the entire environment.

Here's a simple formulation of the vacuum cleaner problem:

Problem Statement:

A robotic vacuum cleaner is placed in a grid-based environment, where each cell in the grid represents a location in the room. The vacuum cleaner is initially placed at a specific location. The objective is to navigate the environment and clean all the dirty cells in the most efficient way.

Environment:

* The environment is represented as a grid with m rows and n columns.
* Each cell in the grid can be in one of three states: dirty, clean, or an obstacle (e.g., a wall).
* The initial position of the vacuum cleaner is known. Actions:
* The vacuum cleaner can take four possible actions: move up, move down, move left, or move right.
* The cleaner cannot move through obstacles or move outside the grid boundaries. Objective:

The goal of the vacuum cleaner is to clean all the dirty cells in the environment while avoiding obstacles.

**Tools:** Visual Studio Code

**Language:** Python

# Source Code:

def vacuum\_world():

    goal\_state = {'A': '0', 'B': '0'}  # 0 indicates Clean and 1 indicates Dirty

    print("Biraj Pudasaini\n")

    location\_input = input("Enter Location of Vacuum: ")  # User input of location vacuum

    status\_input = input("Enter status of " + location\_input + ": ")  # User input of location status

    status\_input\_complement = input("Enter status of other room: ")  # User input of the other room status

    print("Initial Location Condition" + str(goal\_state))

    if location\_input == 'A':

        print("Vacuum is placed in Location A")

        if status\_input == '1':

            print("Location A is Dirty.")

            goal\_state['A'] = '0'

            print("Location A has been Cleaned.")

        if status\_input\_complement == '1':

            print("Location B is Dirty.")

            print("Moving right to the Location B. ")

            goal\_state['B'] = '0'

            print("Location B has been Cleaned. ")

        else:

            print("Location B is already clean.")

    if status\_input == '0':

        print("Location A is already clean")

        if status\_input\_complement == '1':  # if B is Dirty

            print("Location B is Dirty.")

            print("Moving RIGHT to the Location B. ")

            goal\_state['B'] = '0'

            print("Location B has been Cleaned. ")

        else:

            print("Location B is already clean.")

    else:

        print("Vacuum is placed in location B")

        if status\_input == '1':

            print("Location B is Dirty.")

            goal\_state['B'] = '0'

            print("Location B has been Cleaned.")

        if status\_input\_complement == '1':

            print("Location A is Dirty.")

            print("Moving LEFT to the Location A. ")

            goal\_state['A'] = '0'

            print("Location A has been Cleaned.")

        else:

            print("Location B is already clean.")

    if status\_input\_complement == '1':  # if A is Dirty

        print("Location A is Dirty.")

        print("Moving LEFT to the Location A. ")

        goal\_state['A'] = '0'

        print("Location A has been Cleaned. ")

    else:

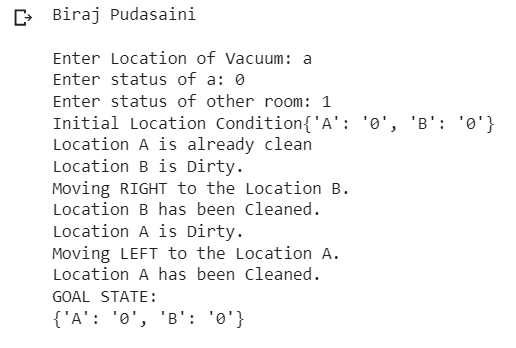
        print("Location A is already clean.")

    print("GOAL STATE: ")

    print(goal\_state)

vacuum\_world()

# Output:

****

**Conclusion:**

In this code we create vacuum cleaner at A and status of A was clean rest of other room are dirty so vacuum move right and clean the B room. This is simple rule-based agent that makes random moves.

**LAB 10**

**Title: Program to implement a Water Jug Problem. Background:**

The Water Jug problem, also known as the Die-Hard problem, is a classic puzzle that involves two jugs of different capacities and a target volume of water that needs to be measured using these jugs. The objective is to find a sequence of pouring operations that will result in obtaining the desired target volume in one of the jugs.

Problem Statement:

Given two jugs of capacities jug1\_capacity and jug2\_capacity, and a target volume target\_volume, find a sequence of pouring operations to achieve exactly target\_volume liters of water in one of the jugs.

The pouring operations are defined as follows:

* Fill either jug completely.
* Empty either jug.
* Pour water from one jug into the other until the first jug is empty or the second jug is full.

**Tools:** Visual Studio Code

**Language:** Python

# Source Code:

from collections import defaultdict

jug1 = 6

jug2 = 4

aim = 2

processed = defaultdict(lambda: False)

def waterJug(amt1, amt2):

    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):

        print(amt1, amt2)

        return True

    if processed[(amt1, amt2)] == False:

        print(amt1, amt2)

        processed[(amt1, amt2)] = True

        return (

            waterJug(0, amt2) or

            waterJug(amt1, 0) or

            waterJug(jug1, amt2) or

            waterJug(amt1, jug2) or

            waterJug(amt1 + min(amt2, (jug1 - amt1)), amt2 - min(amt2, (jug1 - amt1))) or

            waterJug(amt1 - min(amt1, (jug2 - amt2)), amt2 + min(amt1, (jug2 - amt2)))

        )

    else:

        return False

print("Biraj Pudasaini\n")

print("--Water Jug Problem--\n")

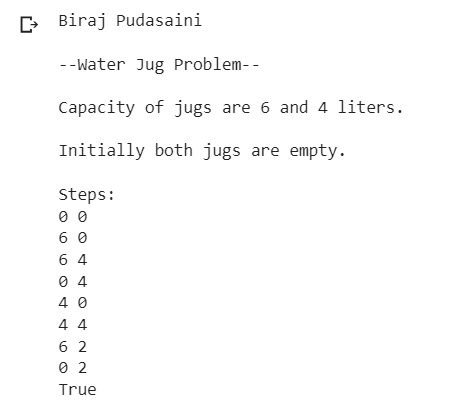
print(f"Capacity of jugs are {jug1} and {jug2} liters.\n")

print("Initially both jugs are empty.\n")

print("Steps: ")

waterJug(0, 0)

**Output:**



**Conclusion:**

In this code, we use a depth-first search (DFS) approach to explore all possible pouring operations until the target volume is achieved. The function water\_jug\_dfs recursively generates all possible paths and returns the solution path once the target volume is obtained in one of the jugs.

**COLLEGE OF APPLIED BUSINESS AND TECHNOLOGY**

Gangahity, Chabahil, Kathmandu-7, Nepal

(Affiliated to Tribhuvan University)



**Laboratory Assignment Report of Artificial Intelligent (CSC-261)**

Submitted by: Submitted to:

**Name:** Biraj Pudasaini **Instructor:** Sudarshan Sharma

**Roll No:** 108

**Semester:** Fourth

**Faculty:** Science and Technology

**Level:** Bachelor

**Program:** CSIT

# Index

|  |  |  |
| --- | --- | --- |
| **Task No.** | **Title** | **Signature** |
| 1. | WAP to traverse a graph using BFS |  |
| 2. | WAP to search goal node using BFS |  |
| 3. | WAP to traverse through a graph using DFS. |  |
| 4. | WAP to search goal node using DFS. |  |
| 5. | Simulate a knowledge base for listed dataset and perform operations in Prolog. |  |
| 6. | Simulate a knowledge base for listed dataset and perform operations in Prolog**.** |  |
| 7. | Program to implement neural network for the given dataset. |  |
| 8. | Program to implement neural network for the given dataset. |  |
| 9. | Program to demonstrate Vacuum Cleaner problem. |  |
| 10. | Program to implement a Water Jug Problem. |  |