**Lab 1: Python Program for Breadth-First Search.**

**Theory:** BFS stands for Breadth-First Search. It is an algorithm used for traversing or searching tree or graph data structures. BFS explores all the vertices of a graph or all the elements of a tree level by level, starting from a specified source vertex or root node. It visits all the neighbors of a given vertex before moving on to the next level. BFS ensures that all vertices are visited in increasing order.

**Implement BFS using high level language.**

from collections import deque

class Graph:

  def \_\_init\_\_(self, V):

        self.V = V

        self.adj = [[] for \_ in range(V)]

    def add\_edge(self, u, v):

        self.adj[u].append(v)

        self.adj[v].append(u)

    def bfs(self, startVertex, searchVertex):

        visited = [False] \* self.V

        queue = deque()

        queue.append(startVertex)

        visited[startVertex] = True

        while queue:

            currentVertex = queue.popleft()

            print(currentVertex, end=' ')

            if currentVertex == searchVertex:

                return True

            for v in self.adj[currentVertex]:

                if not visited[v]:

                    visited[v] = True

                    queue.append(v)

        return False

def adj\_matrix():

    V = int(input("Enter the number of vertices: "))

    E = int(input("Enter the number of edges: "))

    G = Graph(V)

    for \_ in range(E):

        src, dest = map(int, input("Enter the source and destination of edge: ").split())

        G.add\_edge(src, dest)

    return G

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

    G = adj\_matrix()

    startVertex = int(input("Enter the starting vertex for BFS: "))

    searchVertex = int(input("Enter the vertex to search for: "))

    print("BFS traversal:")

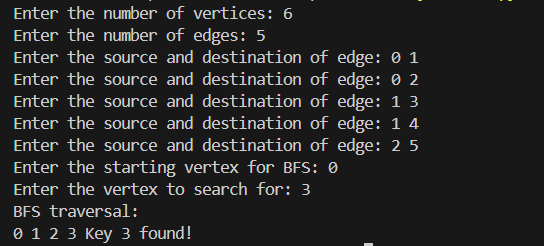
    if G.bfs(startVertex, searchVertex):

        print(f"Key {searchVertex} found!")

    else:

        print(f"Key {searchVertex} not found!")

Output:



**Lab 2: Python Program for Depth First Search**

**Theory:** DFS stands for Depth-First Search. It is another algorithm used for traversing or searching tree or graph data structures. Unlike BFS, DFS explores a path as deeply as possible before backtracking.

**Implement DFS using any high level language.**

**Source Code:**

class Graph:

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

self.V = V

self.adj = [[] for \_ in range(V)]

def add\_edge(self, u, v):

self.adj[u].append(v)

self.adj[v].append(u)

def dfs\_util(self, v, visited, searchVertex, found):

visited[v] = True

print(v, end=' ')

if v == searchVertex:

found[0] = True

return

for i in self.adj[v]:

if not visited[i]:

self.dfs\_util(i, visited, searchVertex, found)

if found[0]:

return

def dfs(self, startVertex, searchVertex):

visited = [False] \* self.V

found = [False]

self.dfs\_util(startVertex, visited, searchVertex, found)

return found[0]

def adj\_matrix():

V = int(input("Enter the number of vertices: "))

E = int(input("Enter the number of edges: "))

G = Graph(V)

for \_ in range(E):

src, dest = map(int, input("Enter the source and destination of edge: ").split())

G.add\_edge(src, dest)

return G

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

G = adj\_matrix()

startVertex = int(input("Enter the starting vertex for DFS: "))

searchVertex = int(input("Enter the vertex to search for: "))

print("DFS traversal:")

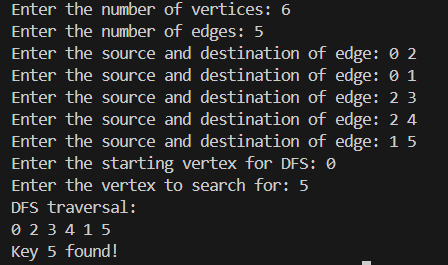
if G.dfs(startVertex, searchVertex):

print(f"\nKey {searchVertex} found!")

else:

print(f"\nKey {searchVertex} not found!")

**Output:**



**Lab 3: GREEDY- BEST FIRST SEARCH (GBFS)**

**Theory:** GBFS stands for Greedy Best-First Search. It is a search algorithm that combines the characteristics of both BFS and the greedy strategy. In GBFS, the algorithm evaluates each node based on an estimated cost to the goal, without considering the cost of reaching the current node. It always expands the node that appears to be closest to the goal according to a heuristic function.

**Implement GBFS using any high level language.**

**Source Code:**

from queue import PriorityQueue

class Node:

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

self.vertex = vertex

self.next = None

def create\_node(vertex):

return Node(vertex)

def add\_edge(graph, src, dest):

new\_node = create\_node(dest)

new\_node.next = graph[src]

graph[src] = new\_node

def gbfs(graph, heuristic, start\_node, goal\_node):

visited = [False] \* len(graph)

pq = PriorityQueue()

pq.put((heuristic[start\_node], start\_node))

visited[start\_node] = True

print("GBFS Path:", start\_node, end=" ")

while not pq.empty():

current\_node = pq.get()[1]

if current\_node == goal\_node:

print("->", goal\_node)

return

best\_child = -1

best\_heuristic = 9999

current\_neighbor = graph[current\_node]

while current\_neighbor is not None:

neighbor = current\_neighbor.vertex

if not visited[neighbor]:

visited[neighbor] = True

pq.put((heuristic[neighbor], neighbor))

if neighbor == goal\_node:

print("->", goal\_node)

return

if heuristic[neighbor] < best\_heuristic:

best\_child = neighbor

best\_heuristic = heuristic[neighbor]

current\_neighbor = current\_neighbor.next

if best\_child != -1:

print("->", best\_child, end=" ")

print("No path found.")

def main():

num\_nodes = int(input("Enter the number of nodes: "))

graph = [None] \* num\_nodes

num\_edges = int(input("Enter the number of edges: "))

print("Enter the edges (source destination):")

for \_ in range(num\_edges):

src, dest = map(int, input().split())

add\_edge(graph, src, dest)

heuristic = []

print("Enter the heuristic values for each node separated by spaces:")

heuristic\_input = input().split()

for value in heuristic\_input:

heuristic.append(int(value))

start\_node = int(input("Enter the starting node: "))

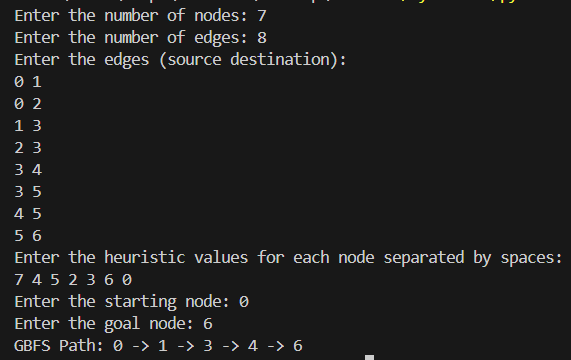
goal\_node = int(input("Enter the goal node: "))

gbfs(graph, heuristic, start\_node, goal\_node)

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

main()

**Output:**



**Lab 4: ADMISSIBLE HEURISTIC (A\*) SEARCH**

**Theory:** A\* search is a popular and widely used informed search algorithm that combines the advantages of both breadth-first search (BFS) and best-first search (greedy search). It is commonly used for path finding and optimization problems. The A\* algorithm uses a heuristic function to estimate the cost from the current node to the goal. It considers both the cost of reaching the current node from the start and the estimated cost from the current node to the goal. This combination allows A\* to make informed decisions while searching.

**Implement A\* search using any high level language.**

**Source Code:**

import heapq

MAX\_NODES = 10

INF = 9999

class Node:

def \_\_init\_\_(self, node, cost, heuristic):

self.node = node

self.cost = cost

self.heuristic = heuristic

def initialize\_graph():

global graph, heuristic

graph = [[INF] \* MAX\_NODES for \_ in range(MAX\_NODES)]

heuristic = [INF] \* MAX\_NODES

def add\_edge(source, destination, cost):

graph[source][destination] = cost

def set\_heuristic(node, value):

heuristic[node] = value

def find\_min\_cost(frontier):

min\_cost = INF

min\_index = -1

for i, (current\_cost, \_) in enumerate(frontier):

if current\_cost < min\_cost:

min\_cost = current\_cost

min\_index = i

return min\_index

def a\_star\_search(start, goal):

frontier = [(0, start)]

visited = [False] \* MAX\_NODES

parent = [-1] \* MAX\_NODES

while frontier:

current\_cost, current\_node = heapq.heappop(frontier)

if current\_node == goal:

path = []

while current\_node != -1:

path.append(current\_node)

current\_node = parent[current\_node]

print("Path:", ' -> '.join(map(str, path[::-1])))

return

for next\_node in range(MAX\_NODES):

cost = graph[current\_node][next\_node]

if cost != INF and not visited[next\_node]:

new\_cost = current\_cost + cost

priority = new\_cost + heuristic[next\_node]

heapq.heappush(frontier, (priority, next\_node))

parent[next\_node] = current\_node

visited[next\_node] = True

print("No path found.")

def main():

initialize\_graph()

num\_nodes = int(input("Enter the number of nodes: "))

num\_edges = int(input("Enter the number of edges: "))

print("Enter the edges in the format: source destination cost")

for \_ in range(num\_edges):

source, destination, cost = map(int, input().split())

add\_edge(source, destination, cost)

print("Enter the heuristic values for each node:")

for i in range(num\_nodes):

value = int(input(f"Node {i}: "))

set\_heuristic(i, value)

start\_node = int(input("Enter the start node: "))

goal\_node = int(input("Enter the goal node: "))

print("\nA\* Search:")

print("Start Node:", start\_node)

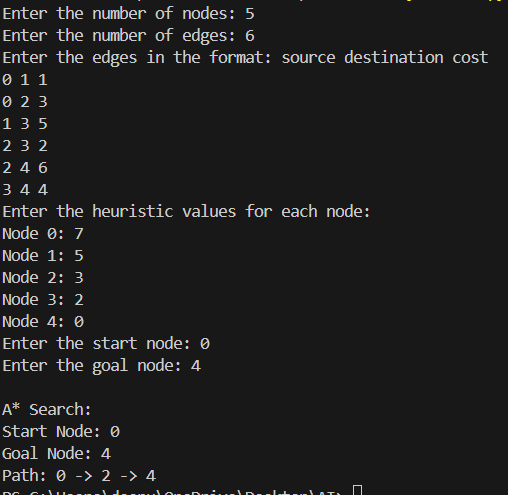
print("Goal Node:", goal\_node)

a\_star\_search(start\_node, goal\_node)

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

main()

**Output:**



**Lab 5: CRYPTOARTHEMATIC PROBLEM**

**Theory:** A cryptoarithmetic problem, also known as a cryptoarithmetic or an alphametic, is a type of puzzle where arithmetic equations are encoded with letters or symbols. The task is to decipher the encoding and find the numerical values that satisfy the equation.

(Like TWO +TWO = FOUR or SEND+MORE = MONEY ).

**Implement Cryptoarthematic problem using any high level language.**

**Source Code:**

from itertools import permutations

def solve\_cryptoarithmetic(w1, w2, w3):

    letters = set(w1 + w2 + w3)

    if len(letters) > 10:

        print("Something is wrong with the input")

        return

    letters = list(letters)

    l4 = len(letters)

    values = [0] \* l4

    def pos(x):

        return letters.index(x)

    def add(s):

        nonlocal l4

        for c in s:

            if c not in letters:

                letters.append(c)

                l4 += 1

    add(w1)

    add(w2)

    add(w3)

    tries = list(permutations(range(10), l4))

    for values in tries:

        if values[pos(w1[0])] == 0 or values[pos(w2[0])] == 0 or values[pos(w3[0])] == 0:

            continue

        n1 = int(''.join(str(values[pos(c)]) for c in w1))

        n2 = int(''.join(str(values[pos(c)]) for c in w2))

        n3 = int(''.join(str(values[pos(c)]) for c in w3))

        if n1 + n2 == n3:

            print("\n\nSolution found:")

            for i, c in enumerate(letters):

                print(f"{c} = {values[i]}")

            return

    print("\n\nNo solution found")

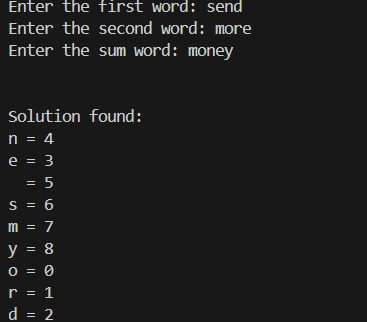
w1 = input("Enter the first word: ")

w2 = input("Enter the second word: ")

w3 = input("Enter the sum word: ")

solve\_cryptoarithmetic(w1, w2, w3)

**Output:**



**LAB 6: PROLOG BASIC PREDICTIONS**

**Given Knowledge:**

1. Sparrow is a bird.
2. Eagle is a bird.
3. Oak is a tree.
4. Pine is a tree.
5. Every tree provides shade.

**Goal:**

1. Birds do not provide shade.

**Program:**

bird(sparrow).

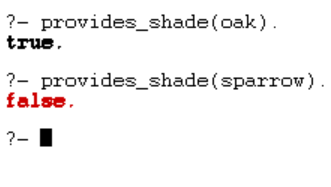
bird(eagle).

tree(oak).

tree(pine).

provides\_shade(X) :- tree(X).

**Output:**



**LAB 7: ANCESTOR PROBLEM**

**Prolog program for ancestor problem.**

female(pam).

female(liz).

female(pat).

female(ann).

male(jim).

male(bob).

male(tom).

male(peter).

parent(pam,bob).

parent(tom,bob).

parent(tom,liz).

parent(bob,ann).

parent(bob,pat).

parent(pat,jim).

parent(bob,peter).

parent(peter,jim).

mother(X,Y):- parent(X,Y), female(X).

father(X,Y):- parent(X,Y), male(X).

sister(X,Y):- parent(Z,X), parent(Z,Y), female(X), X\==Y.

brother(X,Y):- parent(Z,X), parent(Z,Y), male(X), X\==Y.

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

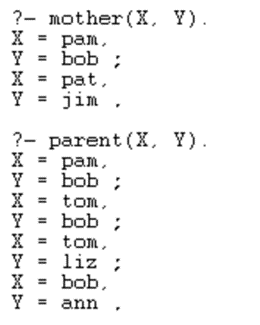
grandmother(X,Z):- mother(X,Y), parent(Y,Z).

grandfather(X,Z):- father(X,Y), parent(Y,Z).

wife(X,Y):- parent(X,Z), parent(Y,Z), female(X), male(Y).

uncle(X,Z):- brother(X,Y), parent(Y,Z).

**Output:**

****

**LAB 8: EXPERT SYSTEM**

**Theory:** An expert system is a computer-based system that emulates the decision-making ability of a human expert in a specific domain. It is designed to provide intelligent advice or solutions to users by utilizing a knowledge base, inference engine, and a user interface.

**Simple expert system in prolog using different knowledge base and rules.**

**Source code:**

fruit(apple) :- is\_true("is red"), is\_true("is sweet").

fruit(banana) :- is\_true("is yellow"), is\_true("is sweet").

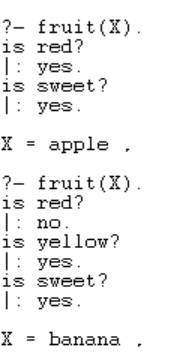
fruit(lemon) :- is\_true("is yellow"), is\_true("is sour").

is\_true(Q) :-

format("~s?\n", [Q]),

read(yes).

**Output:**



**Lab 9:WAP to solve Water Jug Problem.**

**Theory:**  
The water jug problem is a classic AI problem where you are given two jugs, a 4-gallon jug (Jug A) and a 3-gallon jug (Jug B), and your goal is to measure out exactly 2 gallons of water using these jugs. You can fill the jugs, empty them, and pour water from one jug to another.

**Source Code:**

def water\_jug\_problem():

    jug\_a = 4

    jug\_b = 3

    target = 2

    current\_a = 0

    current\_b = 0

    actions = []

    def is\_valid\_state(a, b):

        return 0 <= a <= jug\_a and 0 <= b <= jug\_b

    def perform\_action(action):

        nonlocal current\_a, current\_b

        actions.append(action)

        if action == "fill\_a":

            current\_a = jug\_a

        elif action == "fill\_b":

            current\_b = jug\_b

        elif action == "empty\_a":

            current\_a = 0

        elif action == "empty\_b":

            current\_b = 0

        elif action == "pour\_a\_to\_b":

            to\_pour = min(current\_a, jug\_b - current\_b)

            current\_a -= to\_pour

            current\_b += to\_pour

        elif action == "pour\_b\_to\_a":

            to\_pour = min(current\_b, jug\_a - current\_a)

            current\_b -= to\_pour

            current\_a += to\_pour

    while current\_a != target and current\_b != target:

        if current\_a == 0:

            perform\_action("fill\_a")

        elif current\_b == jug\_b:

            perform\_action("empty\_b")

        elif current\_a > 0 and current\_b < jug\_b:

            perform\_action("pour\_a\_to\_b")

        else:

            perform\_action("pour\_b\_to\_a")

    while current\_a != target:

        if current\_a == 0:

            perform\_action("fill\_a")

        else:

            perform\_action("pour\_a\_to\_b")

    while current\_b != target:

        if current\_b == jug\_b:

            perform\_action("empty\_b")

        else:

            perform\_action("pour\_a\_to\_b")

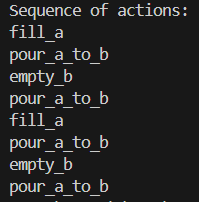
    print("Sequence of actions:")

    for action in actions:

        print(action)

water\_jug\_problem()

**Output:**



**LAB 10: NATURAL LANGUAGE PROCESSING – TOKENIZATION**

**Theory**: Natural Language Processing (NLP) refers to AI method of communicating with an intelligent systems using a natural language such as English. Processing of Natural Language is required when you want an intelligent system like robot to perform as per your instructions, when you want to hear decision from a dialogue based clinical expert system, etc.

**Tokenization** is the process of breaking a stream of textual data into words, terms, sentences, symbols, or some other meaningful elements called tokens.

**Code in Python:**

a. Import the NLTK module and download the text resources needed for the examples.

import nltk # import all the resources for Natural Language Processing with Python nltk.download("book")

b. Take a sentence and tokenize into words. Then apply a part-of-speech tagger.

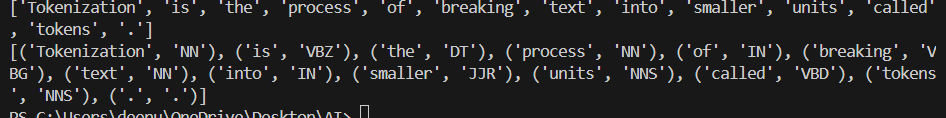
sentence = """ Tokenization is the process of breaking text into smaller units called tokens."""

tokens = nltk.word\_tokenize(sentence)

print(tokens)

tagged = nltk.pos\_tag(tokens)

print(tagged)

**Output:** 

**LAB 11: NATURAL LANGUAGE PROCESSING- PARSE TREE**

**Theory:** Natural Language Processing (NLP) refers to AI method of communicating with an intelligent systems using a natural language such as English.

Processing of Natural Language is required when you want an intelligent system like robot to perform as per your instructions, when you want to hear decision from a dialogue based clinical expert system, etc.

A Syntax tree or a parse tree is a tree representation of different syntactic categories of a sentence. It helps us to understand the syntactical structure of a sentence.

**Code in Python:**

import nltk

from nltk.tree import Tree

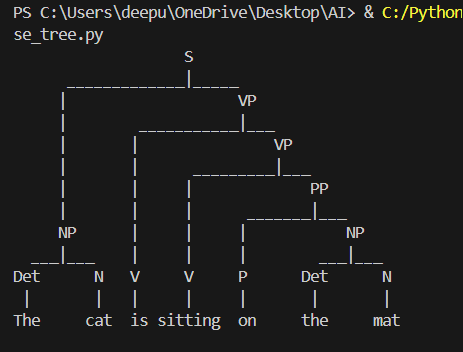
sentence = "The cat is sitting on the mat."

parse\_tree\_string = "(S (NP (Det The) (N cat)) (VP (V is) (VP (V sitting) (PP (P on) (NP (Det the) (N mat))))))"

parse\_tree = Tree.fromstring(parse\_tree\_string)

parse\_tree.pretty\_print()

**Output:**



**Lab 12:Vaccum Cleaner Problem**

**Theory:**The vacuum cleaner problem in AI is a foundational example used to illustrate the concept of intelligent agents in an environment. It involves a simple environment with two locations (A and B), where a vacuum cleaner agent moves between these locations to clean them. The problem demonstrates how the agent perceives its environment, makes decisions, and performs actions (like moving and cleaning) to achieve the goal of a clean environment. The scenario highlights key AI concepts such as agent state, actions, and goal-driven behavior in a controlled setting.

**Code in Python:**

class VacuumCleaner:

    def \_\_init\_\_(self, environment, start\_location):

        self.environment = environment

        self.location = start\_location

    def move(self, direction):

        if direction == "right" and self.location == 'A':

            self.location = 'B'

        elif direction == "left" and self.location == 'B':

            self.location = 'A'

    def clean(self):

        if self.environment[self.location] == "dirty":

            print(f"Cleaning location {self.location}")

            self.environment[self.location] = "clean"

        else:

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

    def decide\_action(self):

        if self.environment[self.location] == "dirty":

            self.clean()

        else:

            if self.location == 'A':

                self.move("right")

            else:

                self.move("left")

    def run(self, steps):

        for step in range(steps):

            print(f"Step {step + 1}, Location: {self.location}, Status: {self.environment}")

            self.decide\_action()

            if all(status == "clean" for status in self.environment.values()):

                print("All locations are clean. Stopping...")

                break

environment = {'A': 'dirty', 'B': 'dirty'}

vacuum = VacuumCleaner(environment, 'A')

vacuum.run(4)

**Output:**

