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:

```
Enter the number of vertices: 6
Enter the number of edges: 5
Enter the source and destination of edge: 0 1
Enter the source and destination of edge: 0 2
Enter the source and destination of edge: 1 3
Enter the source and destination of edge: 1 4
Enter the source and destination of edge: 2 5
Enter the starting vertex for BFS: 0
Enter the vertex to search for: 3
BFS traversal:
0 1 2 3 Key 3 found!
```

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:

```
Enter the number of vertices: 6
Enter the number of edges: 5
Enter the source and destination of edge: 0 2
Enter the source and destination of edge: 0 1
Enter the source and destination of edge: 2 3
Enter the source and destination of edge: 2 4
Enter the source and destination of edge: 1 5
Enter the starting vertex for DFS: 0
Enter the vertex to search for: 5
DFS traversal:
0 2 3 4 1 5
Key 5 found!
```

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:

```
Enter the number of nodes: 7
Enter the number of edges: 8
Enter the edges (source destination):
0 1
0 2
1 3
2 3
3 4
3 5
4 5
5 6
Enter the heuristic values for each node separated by spaces:
7 4 5 2 3 6 0
Enter the starting node: 0
Enter the goal node: 6
GBFS Path: 0 -> 1 -> 3 -> 4 -> 6
```

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:

```
Enter the number of nodes: 5
Enter the number of edges: 6
Enter the edges in the format: source destination cost
0 1 1
0 2 3
1 3 5
2 3 2
2 4 6
3 4 4
Enter the heuristic values for each node:
Node 0: 7
Node 1: 5
Node 2: 3
Node 3: 2
Node 4: 0
Enter the start node: 0
Enter the goal node: 4

A* Search:
Start Node: 0
Goal Node: 4
Path: 0 -> 2 -> 4
```

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:

```
Enter the first word: send
Enter the second word: more
Enter the sum word: money


Solution found:
n = 4
e = 3
  = 5
s = 6
m = 7
y = 8
o = 0
r = 1
d = 2
```

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:

```
?- provides_shade(oak).
true.

?- provides_shade(sparrow).
false.

?-
```

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:

```
?- parent(X, Y).
X = pam,
Y = bob ;
X = tom,
Y = bob ;
X = tom,
Y = liz .

?- grandparent(X, Y).
X = pam,
Y = ann ;
X = pam,
Y = pat ;
X = pam,
Y = peter .

?- grandfather(X, Y).
X = tom,
Y = ann ;
X = tom,
Y = pat ;
X = tom,
Y = peter .

?- mother(X, Y).
X = pam,
Y = bob ;
X = pat,
Y = jim .

?- uncle(X, Z).
X = peter,
Z = jim ;
false.
```

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:

```
?- fruit(X).
is red?
|: no.
is yellow?
|: yes.
is sweet?
|: no.
is yellow?
|: yes.
is sour?
|: yes.

X = lemon.

?- fruit(X).
is red?
|: yes.
is sweet?
|: yes.

X = apple .
```

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:

```
Sequence of actions:
fill_a
pour_a_to_b
empty_b
pour_a_to_b
fill_a
pour_a_to_b
empty_b
pour_a_to_b
```

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:

```
['Tokenization', 'is', 'the', 'process', 'of', 'breaking', 'text', 'into', 'smaller', 'units', 'called'
, 'tokens', '.']
[('Tokenization', 'NN'), ('is', 'VBZ'), ('the', 'DT'), ('process', 'NN'), ('of', 'IN'), ('breaking', 'V
BG'), ('text', 'NN'), ('into', 'IN'), ('smaller', 'JJR'), ('units', 'NNS'), ('called', 'VBD'), ('tokens
', 'NNS'), ('.', '.')]
PS C:\Users\deepu\OneDrive\Desktop\AI>
```

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:

```
PS C:\Users\deepu\OneDrive\Desktop\AI> & C:/Python
se_tree.py
                   S
      _____________|_____
     |                   VP
     |          _________|___
     |         |             VP
     |         |       ______|___
     |         |      |          PP
     |         |      |     _____|___
     NP        |      |    |         NP
  ___|___      |      |    |      ___|___
Det      N     V      V    P    Det      N
 |       |     |      |    |     |       |
The     cat    is  sitting on   the     mat
```

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:

```
Step 1, Location: A, Status: {'A': 'dirty', 'B': 'dirty'}
Cleaning location A
Step 2, Location: A, Status: {'A': 'clean', 'B': 'dirty'}
Step 3, Location: B, Status: {'A': 'clean', 'B': 'dirty'}
Cleaning location B
All locations are clean. Stopping...
```