Lab 1: Python Program for Breadth-First Search

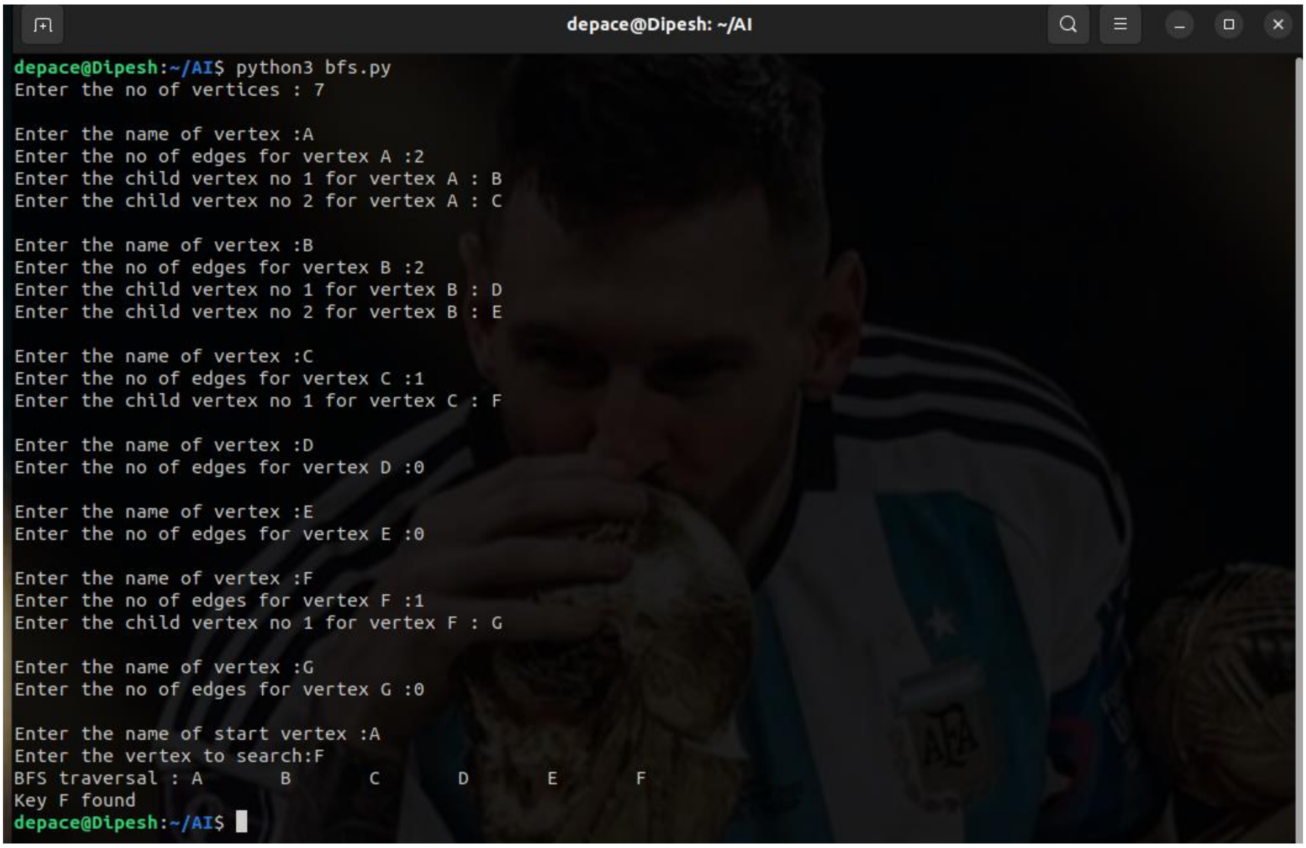
Theory:
BFS stands for Breadth-First Search. It is an algorithm used for traversing or searching tree or graph data structures. BFS explores all the vertices of a graph or all the elements of a tree level by level, starting from a specified source vertex or root node. It visits all the neighbors of a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

Implement BFS using any high level language.

Source Code:
```
from collections import deque
def bfs(graph, start, key):
    found=False
    visited = set()
    queue = deque([start])
    while queue:
        vertex = queue.popleft()
        if vertex not in visited:
            visited.add(vertex)
            print(vertex,end="\t")
            queue.extend(graph[vertex] - visited)
            if key==vertex:
                found=True
                return found
    return found
graph = {}
def add_edge(u, v):
    if u not in graph:
        graph[u] = set()
    graph[u].add(v)

def add_vertex(graph, vertex):
    if vertex not in graph:
```

```
        graph[vertex] = set()
vertices=int(input("Enter the no of vertices : "))

for i in range(vertices):
    print()
    u=input(f"Enter the name of vertex :")
    edges=int(input(f"Enter the no of edges for vertex {u} :"))
    if edges==0:
        add_vertex(graph,u)
    for j in range(edges):
        v=input(f"Enter the child vertex no {j+1} for vertex {u} : ")
        add_edge(u,v)
print()
start_vertex =input(f"Enter the name of start vertex :")
key=input("Enter the vertex to search:")
print("BFS traversal : ",end="")
found= bfs(graph, start_vertex, key)
print()
if(found==True):
    print(f"Key {key} found")
else:
    print(f"Key {key} not found
```

Output:



```
depace@Dipesh:~/AI$ python3 bfs.py
Enter the no of vertices : 7

Enter the name of vertex :A
Enter the no of edges for vertex A :2
Enter the child vertex no 1 for vertex A : B
Enter the child vertex no 2 for vertex A : C

Enter the name of vertex :B
Enter the no of edges for vertex B :2
Enter the child vertex no 1 for vertex B : D
Enter the child vertex no 2 for vertex B : E

Enter the name of vertex :C
Enter the no of edges for vertex C :1
Enter the child vertex no 1 for vertex C : F

Enter the name of vertex :D
Enter the no of edges for vertex D :0

Enter the name of vertex :E
Enter the no of edges for vertex E :0

Enter the name of vertex :F
Enter the no of edges for vertex F :1
Enter the child vertex no 1 for vertex F : G

Enter the name of vertex :G
Enter the no of edges for vertex G :0

Enter the name of start vertex :A
Enter the vertex to search:F
BFS traversal : A       B       C       D       E       F
Key F found
depace@Dipesh:~/AI$
```

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:
```
def dfs(graph, vertex, visited,key):
    visited.add(vertex)
    print(vertex,end="\t")
    if (key==vertex):
        return True
    for neighbor in graph.get(vertex, []):
        if neighbor not in visited:
            if dfs(graph, neighbor, visited,key):
                return True

    return False

graph = {}

def add_edge(u, v):
    if u not in graph:
        graph[u] = set()
    graph[u].add(v)

def add_vertex(graph, vertex):
    if vertex not in graph:
        graph[vertex] = set()

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

for i in range(vertices):
```

```
    print()
    u=input(f"Enter the name of vertex :")
    edges = int(input(f"Enter the number of edges for vertex {u}: "))
    if edges == 0:
        add_vertex(graph, u)
    for j in range(edges):
        v=input(f"Enter the child vertex no {j+1} for vertex {u} : ")
        add_edge(u,v)

print()
start_vertex = input("Enter the starting vertex: ")
key=input("Enter the vertex to search:")
print()
print("DFS traversal : ",end="")
visited = set()
found=dfs(graph, start_vertex, visited,key)
print()
if found:
        print(f"Key {key} found")
else:
        print(f"Key {key} not found")
```

Output:

```
depace@Dipesh: ~/AI
depace@Dipesh:~/AI$ python3 dfs.py
Enter the number of vertices:7

Enter the name of vertex :a
Enter the number of edges for vertex a: 2
Enter the child vertex no 1 for vertex a : b
Enter the child vertex no 2 for vertex a : c

Enter the name of vertex :b
Enter the number of edges for vertex b: 2
Enter the child vertex no 1 for vertex b : d
Enter the child vertex no 2 for vertex b : e

Enter the name of vertex :c
Enter the number of edges for vertex c: 1
Enter the child vertex no 1 for vertex c : f

Enter the name of vertex :d
Enter the number of edges for vertex d: 0

Enter the name of vertex :e
Enter the number of edges for vertex e: 0

Enter the name of vertex :f
Enter the number of edges for vertex f: 1
Enter the child vertex no 1 for vertex f : g

Enter the name of vertex :g
Enter the number of edges for vertex g: 0

Enter the starting vertex: a
Enter the vertex to search:f

DFS traversal : a       b       d       e       c       f
Key f found
depace@Dipesh:~/AI$
```

Lab 3: Python Program for 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.

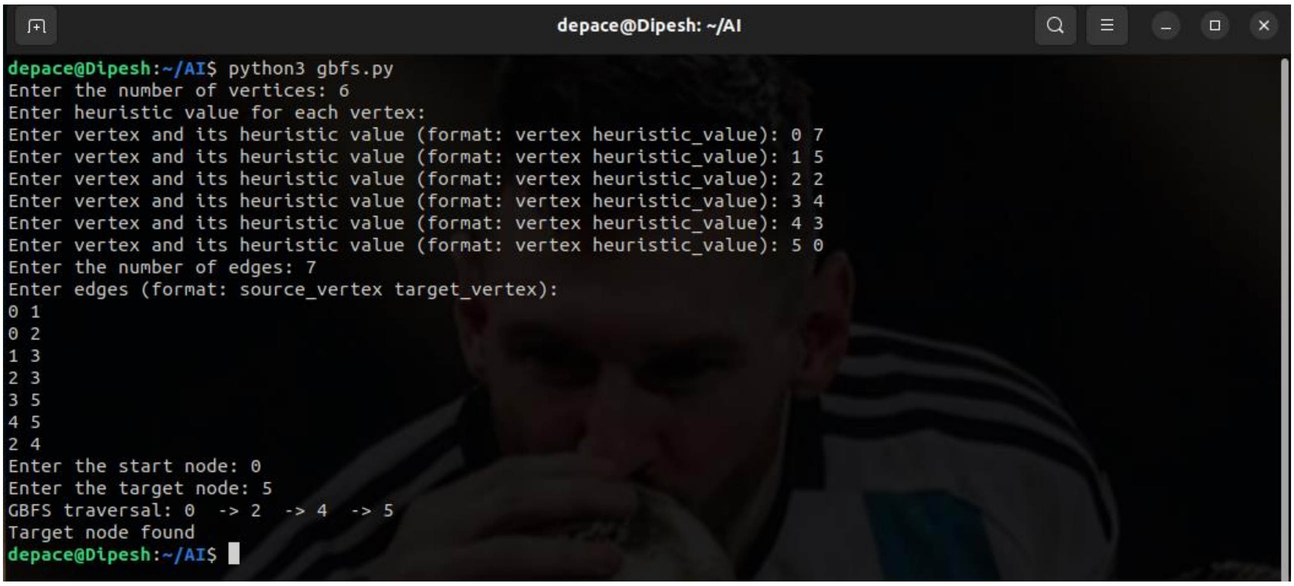
Source Code:

```
from collections import deque

def gbfs(graph, start, target):
    visited = set()
    queue = deque([(0, start)])  # Priority queue with priority as heuristic
value
    while queue:
        _, current = queue.popleft()  # Pop the node with the lowest heuristic
value
        visited.add(current)
        print(current, end=" ")  # Print the current node
        if current == target:
            return True
        print(" -> ",end="")
        neighbors = graph.get(current, {}).get('neighbors', {})
        for neighbor, heuristic_value in neighbors.items():
            if neighbor not in visited:
                if 'heuristic' not in graph.get(neighbor, {}):
                    print(f"Heuristic value not provided for vertex {neighbor}.
Skipping.")
                    continue
```

```
            queue.append((graph[neighbor]['heuristic'], neighbor))  # Add
neighbors to the priority queue
            queue = deque(sorted(queue, key=lambda x: x[0]))  # Sort the
queue based on heuristic value
    return False

# Take input for the graph
graph = {}
vertices = int(input("Enter the number of vertices: "))

# Input heuristic values for each vertex
print("Enter heuristic value for each vertex:")
for _ in range(vertices):
    vertex, heuristic_value = input("Enter vertex and its heuristic value
(format: vertex heuristic_value): ").split()
    graph[vertex] = {'heuristic': int(heuristic_value)}

# Input edges
edges = int(input("Enter the number of edges: "))
print("Enter edges (format: source_vertex target_vertex): ")
for _ in range(edges):
    source, target = input().split()
    if source not in graph:
        print(f"Vertex {source} not found in the graph. Skipping edge input.")
        continue
    if target not in graph:
        print(f"Vertex {target} not found in the graph. Skipping edge input.")
        continue
    graph[source].setdefault('neighbors', {}).update({target:
graph[target]['heuristic']})

start_node = input("Enter the start node: ")
target_node = input("Enter the target node: ")

print("GBFS traversal:", end=" ")
if gbfs(graph, start_node, target_node):
    print("\nTarget node found")
```

```
else:
    print("\nTarget node not found")
```

Output:



```
depace@Dipesh:~/AI$ python3 gbfs.py
Enter the number of vertices: 6
Enter heuristic value for each vertex:
Enter vertex and its heuristic value (format: vertex heuristic_value): 0 7
Enter vertex and its heuristic value (format: vertex heuristic_value): 1 5
Enter vertex and its heuristic value (format: vertex heuristic_value): 2 2
Enter vertex and its heuristic value (format: vertex heuristic_value): 3 4
Enter vertex and its heuristic value (format: vertex heuristic_value): 4 3
Enter vertex and its heuristic value (format: vertex heuristic_value): 5 0
Enter the number of edges: 7
Enter edges (format: source_vertex target_vertex):
0 1
0 2
1 3
2 3
3 5
4 5
2 4
Enter the start node: 0
Enter the target node: 5
GBFS traversal: 0  -> 2  -> 4  -> 5
Target node found
depace@Dipesh:~/AI$
```

Lab 4: Python Program for 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.

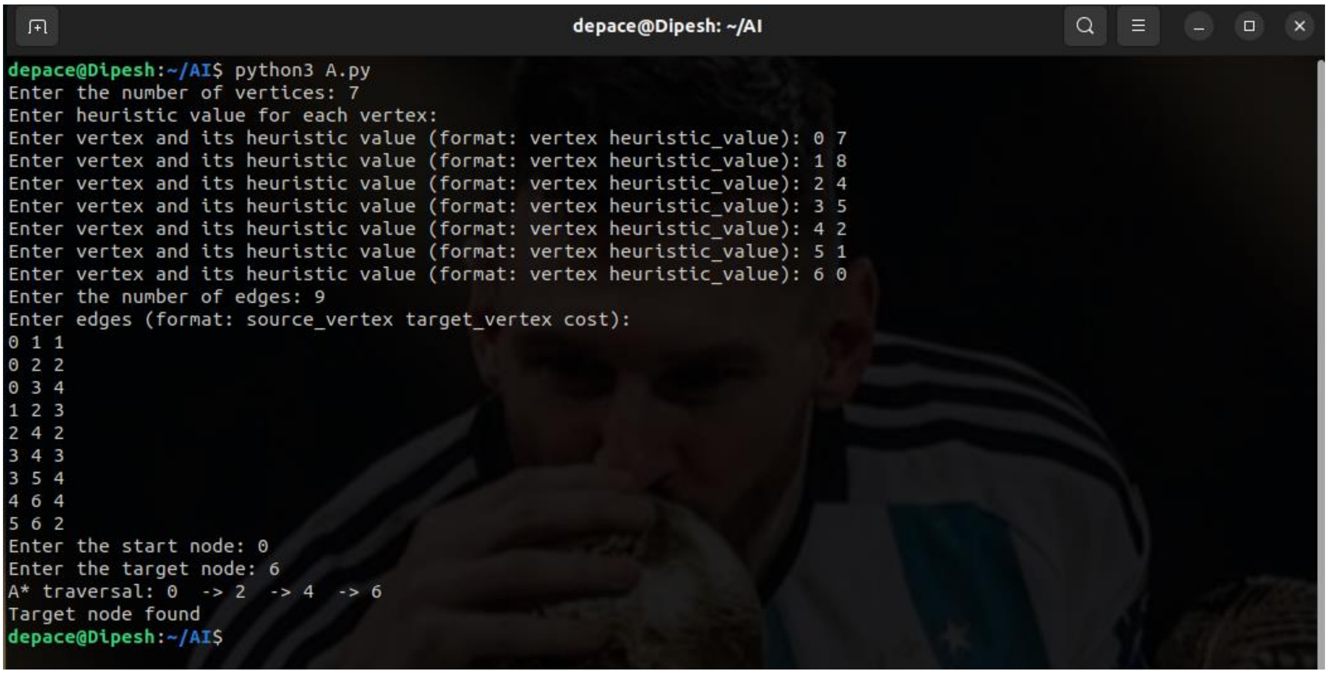
Source Code:

```
from queue import PriorityQueue

def astar(graph, start, target):
    visited = set()
    queue = PriorityQueue()
    queue.put((0 + graph[start]['heuristic'], 0, start))  # (f, g, node)
    while not queue.empty():
        _, cost, current = queue.get()
        visited.add(current)
        print(current, end=" ")  # Print the current node
        if current == target:
            return True
        print(" -> ",end="")
        neighbors = graph.get(current, {}).get('neighbors', {})
        for neighbor, heuristic_value in neighbors.items():
            if neighbor not in visited:
                if 'heuristic' not in graph.get(neighbor, {}):
                    print(f"Heuristic value not provided for vertex {neighbor}.
Skipping.")
                    continue
```

```
            g = cost + neighbors[neighbor]  # Actual cost from start to
neighbor
            queue.put((g + graph[neighbor]['heuristic'], g, neighbor))  # Add
neighbor to the priority queue
    return False

# Take input for the graph
graph = {}
vertices = int(input("Enter the number of vertices: "))

# Input heuristic values for each vertex
print("Enter heuristic value for each vertex:")
for _ in range(vertices):
    vertex, heuristic_value = input("Enter vertex and its heuristic value
(format: vertex heuristic_value): ").split()
    graph[vertex] = {'heuristic': int(heuristic_value)}

# Input edges
edges = int(input("Enter the number of edges: "))
print("Enter edges (format: source_vertex target_vertex cost): ")
for _ in range(edges):
    source, target, cost = input().split()
    cost = int(cost)
    if source not in graph:
        print(f"Vertex {source} not found in the graph. Skipping edge input.")
        continue
    if target not in graph:
        print(f"Vertex {target} not found in the graph. Skipping edge input.")
        continue
    graph[source].setdefault('neighbors', {}).update({target: cost})

start_node = input("Enter the start node: ")
target_node = input("Enter the target node: ")

print("A* traversal:", end=" ")
if astar(graph, start_node, target_node):
    print("\nTarget node found")
```

```
else:
    print("\nTarget node not found")
```

Output:



```
depace@Dipesh: ~/AI
depace@Dipesh:~/AI$ python3 A.py
Enter the number of vertices: 7
Enter heuristic value for each vertex:
Enter vertex and its heuristic value (format: vertex heuristic_value): 0 7
Enter vertex and its heuristic value (format: vertex heuristic_value): 1 8
Enter vertex and its heuristic value (format: vertex heuristic_value): 2 4
Enter vertex and its heuristic value (format: vertex heuristic_value): 3 5
Enter vertex and its heuristic value (format: vertex heuristic_value): 4 2
Enter vertex and its heuristic value (format: vertex heuristic_value): 5 1
Enter vertex and its heuristic value (format: vertex heuristic_value): 6 0
Enter the number of edges: 9
Enter edges (format: source_vertex target_vertex cost):
0 1 1
0 2 2
0 3 4
1 2 3
2 4 2
3 4 3
3 5 4
4 6 4
5 6 2
Enter the start node: 0
Enter the target node: 6
A* traversal: 0  -> 2  -> 4  -> 6
Target node found
depace@Dipesh:~/AI$
```

Lab 5: Python Program for Crypto Arithmetic

Theory:

A cryptoarithmetic problem, also known as a cryptoarithmetic or an alphametic, is a type of puzzle where arithmetic equations are encoded with letters

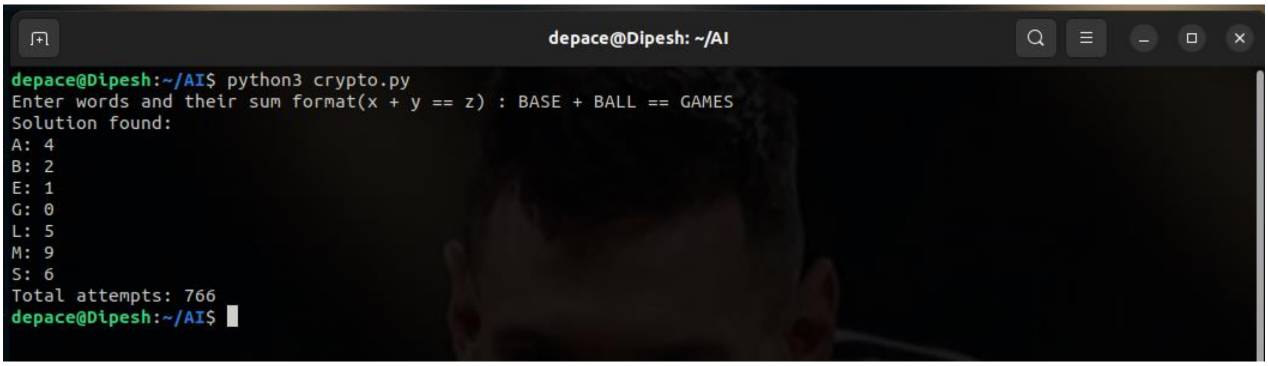
Implement Crypto Arihmetic Problem using any high level language.

Source Code:

```
from itertools import permutations

def solve_cryptarithmetic(puzzle):
    # Extracting unique letters from the puzzle
    unique_letters = set([char for char in puzzle if char.isalpha()])
    letters_count = len(unique_letters)
    if letters_count > 10:
        print("Invalid puzzle: More than 10 unique letters")
        return

    # Generate all permutations of digits from 0 to 9
    digit_permutations = permutations(range(10), letters_count)
    attempts = 0

    for digit_assignment in digit_permutations:
        attempts += 1
        assignment = dict(zip(unique_letters, digit_assignment))
        # Check if the assignment satisfies the puzzle
        if satisfies_puzzle(puzzle, assignment):
            return assignment, attempts

    return None, attempts
def satisfies_puzzle(puzzle, assignment):
    # Replace letters in the puzzle with digits
```

```
    for letter, digit in assignment.items():
        puzzle = puzzle.replace(letter, str(digit))

    # Format the puzzle expression to remove leading zeros
    puzzle = puzzle.replace(" 0", " ")

    # Evaluate the arithmetic expression
    try:
        return eval(puzzle)
    except ZeroDivisionError:
        return False
puzzle = input("Enter words and their sum format(x + y == z) : ")
solution, attempts = solve_cryptarithmetic(puzzle)
if solution:
    print("Solution found:")
    for letter, digit in sorted(solution.items()):
        print(f"{letter}: {digit}")
    print(f"Total attempts: {attempts}")
else:
    print("No solution found.")
    print(f"Total attempts: {attempts}")
```

Output



Lab 6: Python Program for Vacuum Cleaner problem

Theory:

In the realm of artificial intelligence, solving problems related to vacuum cleaners involves designing algorithms for efficient navigation, obstacle avoidance, and cleaning pattern optimization. AI enables adaptability to different environments and levels of dirtiness, allowing the vacuum cleaner to learn and adjust its strategies accordingly. Additionally, smart scheduling algorithms optimize cleaning times based on occupancy and user preferences, while energy-efficient management enhances overall performance. Through AI advancements, vacuum cleaners can autonomously provide thorough and efficient cleaning while minimizing user intervention.

Implement Vacuum Cleaner Problem using any high level language.

Source Code:

```
import random
class VacuumCleaner:
    def __init__(self, rows, cols):
        self.rows = rows
        self.cols = cols
        self.position = (random.randint(0, rows-1), random.randint(0, cols-1))
        self.grid = [[random.choice([True, False]) for _ in range(cols)] for _
in range(rows)]

    def print_grid(self):
        for i in range(self.rows):
            for j in range(self.cols):
                if self.position == (i, j):
                    print("V" if self.grid[i][j] else "V", end=" ")  # Vacuum
cleaner symbol
                else:
                    print("D" if self.grid[i][j] else "-", end=" ")  # Dirt symbol
            print()
```

```
    def clean(self):
        cleaned = 0
        total_dirt = sum(row.count(True) for row in self.grid)
        while cleaned < total_dirt:
            if self.grid[self.position[0]][self.position[1]]:
                self.grid[self.position[0]][self.position[1]] = False  # Clean the
dirt
                cleaned += 1
            self.print_grid()
            move_direction = input("Enter direction to move
(up/down/left/right): ")
            self.move(move_direction)  # Corrected line to call move method
with input direction
        print("All dirt cleaned!")

    def move(self, direction):
        if direction == "up" and self.position[0] > 0:
            self.position = (self.position[0] - 1, self.position[1])
        elif direction == "down" and self.position[0] < self.rows - 1:
            self.position = (self.position[0] + 1, self.position[1])
        elif direction == "left" and self.position[1] > 0:
            self.position = (self.position[0], self.position[1] - 1)
        elif direction == "right" and self.position[1] < self.cols - 1:
            self.position = (self.position[0], self.position[1] + 1)

rows=int(input("Enter no of rows:"))
columns=int(input("Enter no of columns:"))
vacuum = VacuumCleaner(rows,columns)
print("Initial grid:")
vacuum.clean()
```

Output

```
depace@Dipesh: ~/AI
depace@Dipesh:~/AI$ python3 "vacuum cleaner.py"
Enter no of rows:3
Enter no of columns:3
Initial grid:
D D D
V - -
D - D
Enter direction to move (up/down/left/right): up
V D D
- - -
D - D
Enter direction to move (up/down/left/right): right
- V D
- - -
D - D
Enter direction to move (up/down/left/right): right
- - V
- - -
D - D
Enter direction to move (up/down/left/right): down
- - -
- - V
D - D
Enter direction to move (up/down/left/right): down
- - -
- - -
D - V
Enter direction to move (up/down/left/right): left
- - -
- - -
D V -
Enter direction to move (up/down/left/right): left
- - -
- - -
V - -
Enter direction to move (up/down/left/right): up
All dirt cleaned!
depace@Dipesh:~/AI$
```

Lab 7: Python Program for Water Jug problem

Theory:

The water jug problem is a classic puzzle involving two jugs with different capacities and the goal of measuring a specific quantity of water. Players can only fill, empty, or pour water between jugs. By strategizing these operations, the desired amount of water can be attained. Creative thinking and experimentation are key to mastering this problem.

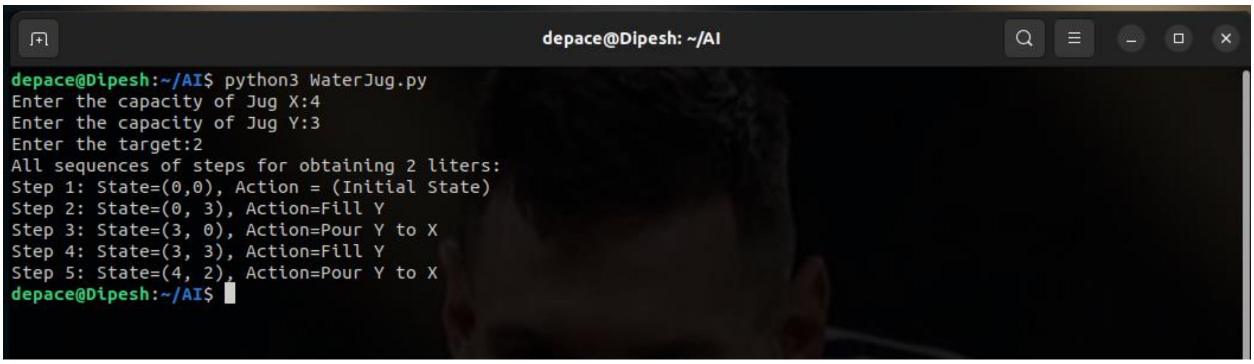
Implement Water Jug Problem using any high level language.

Source Code:

```
from collections import deque

def water_jug(jug_x, jug_y, target):
    visited = set()
    queue = deque([((0, 0), [])])  # Initial state is (0, 0) with empty sequence

    while queue:
        current_state, sequence = queue.popleft()

        if target in current_state:
            return sequence

        if current_state in visited:
            continue

        visited.add(current_state)

        actions = [
            ((jug_x, current_state[1]), "Fill X"),
            ((current_state[0], jug_y), "Fill Y"),
            ((0, current_state[1]), "Empty X"),
            ((current_state[0], 0), "Empty Y"),
```

```
            ((max(0, current_state[0] - (jug_y - current_state[1])), min(jug_y, current_state[0] + current_state[1])), "Pour X to Y"),
            ((min(jug_x, current_state[0] + current_state[1]), max(0, current_state[1] - (jug_x - current_state[0]))), "Pour Y to X")
        ]

        for next_state, action in actions:
            if next_state not in visited:
                queue.append((next_state, sequence + [(next_state, action)]))

    return None

# Example usage
jug_x = int(input("Enter the capacity of Jug X:"))  # Capacity of jug X
jug_y = int(input("Enter the capacity of Jug Y:"))  # Capacity of jug Y
target = int(input("Enter the target:"))  # Target amount of water

result = water_jug(jug_x, jug_y, target)
if result:
    print(f"All sequences of steps for obtaining {target} liters:")
    print(f"Step 1: State=(0,0), Action = (Initial State)")
    for i, (state, action) in enumerate(result):
        print(f"Step {i + 2}: State={state}, Action={action}")
else:
    print(f"Target amount of {target} liters cannot be obtained with the given jugs.")
```

Output



LAB 8: PROLOG BASIC PREDICTIONS

Given Knowledge:

Sparrow is a bird.
Eagle is a bird.
Oak is a tree.
Pine is a tree.
Every tree provides shade.

Goal:

Birds do not provide shade.

Prolog Program:

Bird(sparrow).
Bird(eagle).
Tree(oak).
Tree(pine).
Provides_shade(X):-tree(X).

Output:

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

?- provides_shade(sparrow).
false.

?-
```

LAB 9: Ancestor Problem (of your own)

Prolog program for ancestor problem of your own.

Prolog Program:

```
male(daman).
male(dipesh).
male(shiva).
male(saimon).
male(ramu).
male(arson).
male(chabilal).
female(sita).
female(dipika).
female(padma).
female(sushila).
female(devi).
parent(daman,dipesh).
parent(daman,dipika).
parent(sita,dipesh).
parent(sita,dipika).
parent(shiva,saimon).
parent(sushila,saimon).
parent(padma,arson).
parent(ramu,arson).
parent(chabilal,sita).
parent(devi,sita).
parent(chabilal,shiva).
parent(devi,shiva).
parent(chabilal,padma).
parent(devi,padma).
```

```
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(Z,Y),parent(X,Z).
```

Output:

```
?-
% e:/program/prolog/facts compiled 0.00 sec, 0 clauses
?- father(X,Y).
X = daman,
Y = dipesh ;
X = daman,
Y = dipika ;
X = shiva,
Y = saimon ;
X = ramu,
Y = arson ;
X = chabilal,
Y = sita ;
X = chabilal,
Y = shiva .

?- mother(X,Y).
X = sita,
Y = dipesh ;
X = sita,
Y = dipika ;
X = sushila,
Y = saimon ;
X = padma,
Y = arson ;
X = devi,
Y = sita ;
X = devi,
Y = shiva ;
X = devi,
Y = padma.

?- sister(X,Y).
X = dipika,
Y = dipesh ;
X = dipika,
Y = dipesh ;
X = sita,
Y = shiva ;
X = sita,
Y = padma ;
X = sita,
Y = shiva ;
X = sita,
Y = padma ;
X = padma,
Y = sita ;
X = padma,
Y = shiva ;
X = padma,
Y = sita ;
X = padma,
Y = shiva ;
```

```
?- brother(X,Y).
X = dipesh,
Y = dipika ;
X = dipesh,
Y = dipika ;
X = shiva,
Y = sita ;
X = shiva,
Y = padma ;
X = shiva,
Y = sita ;
X = shiva,
Y = padma ;

?- grandparent(X,Y).
X = chabilal,
Y = dipesh ;
X = devi,
Y = dipesh ;
X = chabilal,
Y = dipika ;
X = devi,
Y = dipika ;
X = chabilal,
Y = saimon ;
X = devi,
Y = saimon ;
X = chabilal,
Y = arson .

?-
```

Lab 10: 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:

```
animal(dog) :- is_true("has fur"), is_true("says woof").
animal(cat) :- is_true("has fur"), is_true("says meow").
animal(duck) :- is_true("has feathers"), is_true("says quack").

is_true(Q) :-
format("~s?\n", [Q]),
read(yes)
```

Output

```
?- animal(A).
has fur?
|: no.
has fur?
|: no.
has feathers?
|: no.

false.

?- animal(A).
has fur?
|: no.
has fur?
|: no.
has feathers?
|: yes.
says quack?
|: yes.

A = duck.

?- animal(A).
has fur?
|: yes.
says woof?
|: yes.

A = dog .

?-
```

Lab 11: 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.

Source Code:

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 = """I just dont want to talk to you
more after what happened."""
tokens = nltk.word_tokenize(sentence)
print(tokens)
tagged = nltk.pos_tag(tokens)
print(tagged)
```

Output

```
[nltk_data]  Done downloading collection book
['I', 'just', 'dont', 'want', 'to', 'talk', 'to', 'you', 'more', 'after', 'what', 'happened', '.']
[('I', 'PRP'), ('just', 'RB'), ('dont', 'VB'), ('want', 'VBP'), ('to', 'TO'), ('talk', 'VB'), ('to', 'TO'
), ('you', 'PRP'), ('more', 'JJR'), ('after', 'IN'), ('what', 'WP'), ('happened', 'VBD'), ('.', '.')]
PS E:\Program\AI>
```

Lab 12: 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.

Source Code:

```
import nltk
from nltk.tree import Tree

# Example sentence and parse tree string
sentence = "The dog is chasing the cat."
parse_tree_string = "(S (NP (Det The) (N dog)) (VP (V is) (NP (V chasing) (Det the) (N cat))))"
# Convert the parse tree string into an NLTK Tree object
parse_tree = Tree.fromstring(parse_tree_string)
# Draw the parse tree
parse_tree.pretty_print()
```

Output

```
                  python -u "e:\Program\AI\parse.py"
             S
       _______|___________
      |                   VP
      |         ___________|___
      NP        |               NP
   ___|___      |       _________|___
Det       N     V     V         Det   N
 |        |     |     |          |    |
The      dog    is chasing      the  cat

PS E:\Program\AI>
```