

18CSC305J

Artificial Intelligence

Record

Register no:	RA1911003010090
Name of the student:	Naman Jain
Semester:	6th
Department:	CSE



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)



SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
S.R.M. NAGAR, KATTANKULATHUR - 603 203 KANCHEEPURAM DISTRICT

BONAFIDE CERTIFICATE

Register No: RA1911003010090

*Certified to be the bonafide record of work done by **NAMAN JAIN** of
CSE B.Tech Degree course in the Practical **18CSC305J – Artificial Intelligence**
in **SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, KATTANKULATHUR**
during the academic year **2021-2022**.*

Lab Incharge

Date:

Year Co-ordinator

*Submitted for University Examination held in **Artificial Intelligence Lab**, SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, Kattankulathur.*

Date:

Examiner-1

Examiner-2

INDEX SHEET

Exp. No	Date of Experiment	Name of the Experiment	Marks	Staff Signature
1	12/01/2022	Implementation of Toy Problem		
2	21/01/2022	Implementation of Real World Problem		
3a	31/01/2022	Implementation of Constraint Satisfaction Problem.		
3b	07/02/2021	Implementation of Graph Coloring Problem		
4	21/02/2021	Implementation of BFS and DFS		
5	28/02/2021	Implementation of Best first search and A* Algorithm		
6a	07/03/2021	Implementation of Unification		
6b	14/03/2021	Implementation of Resolution		
7	28/03/2021	Implementation of uncertain methods for Dempster Shafer's Theory.		
8	04/04/2021	Implementation of learning algorithms for an application		
9	11/04/2021	Implementation of sentiment analysis using NLP		
10	18/04/2021	Applying deep learning methods to solve an application		

Total Mark:

Average:

Artificial Intelligence (18CSC305J)

Experiment 1

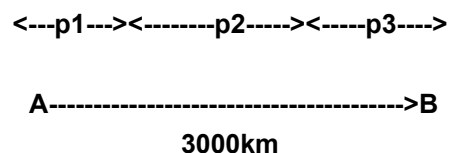
Naman Jain
RA1911003010090
B1 CSE

Aim: Implementation of Toy Problem using python

Problem Statement:

A person has 3000 bananas and a camel. The person wants to transport the maximum number of bananas to a destination which is 1000 KMs away, using only the camel as a mode of transportation. The camel cannot carry more than 1000 bananas at a time and eats a banana every km it travels. What is the maximum number of bananas that can be transferred to the destination using only camel (no other mode of transportation is allowed).

Algorithm:



Since there are 3000 bananas and the Camel can only carry 1000 bananas, he will have to make 3 trips to carry them all to any point in between.

When bananas are reduced to 2000 then the Camel can shift them to another point in 2 trips and when the number of bananas left is ≤ 1000 , then he should not return and only move forward.

In the first part, P1, to shift the bananas by 1Km, the Camel will have to

1. Move forward with 1000 bananas – Will eat up 1 banana on the way forward
2. Leave 998 bananas after 1 km and return with 1 banana – will eat up 1 banana on the way back
3. Pick up the next 1000 bananas and move forward – Will eat up 1 banana on the way forward
4. Leave 998 bananas after 1 km and return with 1 banana - will eat up 1 banana on the way back

5. Will carry the last 1000 bananas from point a and move forward – will eat up 1 banana

Note: After point 5 the Camel does not need to return to point A again.

So to shift 3000 bananas by 1km, the Camel will eat up 5 bananas.

After moving to 200 km the Camel would have eaten up 1000 bananas and is now left with 2000 bananas.

Hence the length of part P1 is 200 Km.

Now in Part P2, the Camel needs to do the following to shift the Bananas by 1km.

1. Move forward with 1000 bananas - Will eat up 1 banana on the way forward
2. Leave 998 bananas after 1 km and return with 1 banana - will eat up this 1 banana on the way back
3. Pick up the next 1000 bananas and move forward - Will eat up 1 banana on the way forward

Note: After point 3 the Camel does not need to return to the starting point of P2.

So to shift 2000 bananas by 1km, the Camel will eat up 3 bananas.

After moving to 333 km the camel would have eaten up 1000 bananas and is now left with the last 1000 bananas.

Because it is a multiple of 3, after 333 times there are 1001 bananas left. Therefore, the camel must take 1000 bananas and then take 1 back with him to pick up the last banana, leaving him with 999 bananas. This would leave the camel at 333 km + 200 km, so the merchant only needs to travel 466 km, eating 1 banana for every km.

1000 bananas - 466 bananas for each remaining km equals 534 bananas left at point B.

Therefore, the merchant has 533 bananas to sell at point B.

Program:

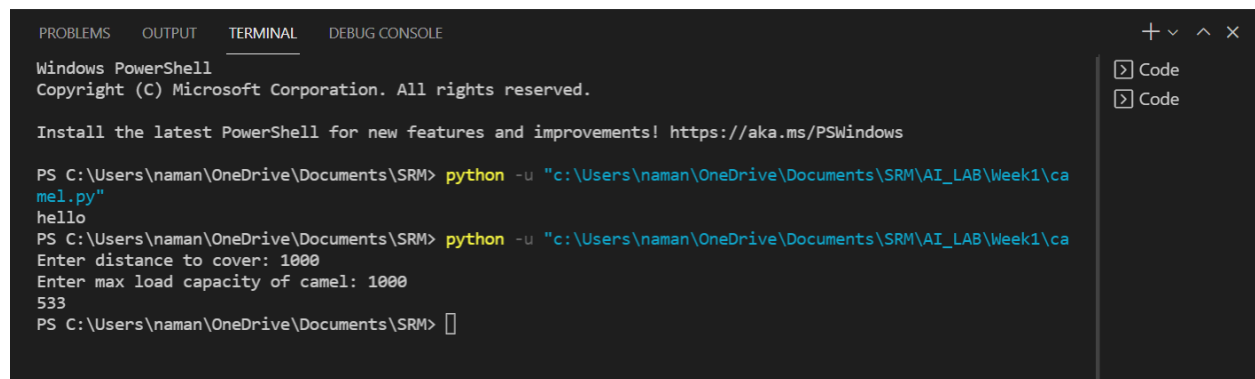
```
total=int(input('Enter no. of bananas at starting: '))
distance=int(input('Enter distance to cover: '))
load_capacity=int(input('Enter max load capacity of camel: '))
lose=0
start=total
for i in range(distance):
    while start>0:
        start=start-load_capacity

        if start==1:
            lose=lose-1
            lose=lose+2

    lose=lose-1
    start=total-lose
```

```
    if start==0:
        break
print(start)
```

Output:



```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\naman\OneDrive\Documents\SRM> python -u "c:\Users\naman\OneDrive\Documents\SRM\AI_LAB\Week1\camel.py"
hello
PS C:\Users\naman\OneDrive\Documents\SRM> python -u "c:\Users\naman\OneDrive\Documents\SRM\AI_LAB\Week1\camel.py"
Enter distance to cover: 1000
Enter max load capacity of camel: 1000
533
PS C:\Users\naman\OneDrive\Documents\SRM> 
```

Result:

Theoretically, the maximum number of bananas that can be transported using one camel, keeping all boundary conditions in mind is 534.

Experimentally, the maximum number of bananas that can be transported using one camel is 533.

Theoretical value \approx Experimental value

Therefore, we can say that the camel banana problem has been successfully implemented and the final number of bananas that can be transported is verified.

Artificial Intelligence (18CSC305J)

Experiment 2

Naman Jain
RA1911003010090
B1 CSE

Aim: Implementation of Real World Problems.

Problem Statement:

(1) TSP (Travelling salesman problem)

Given a set of cities and distance between every pair of cities as an adjacency matrix, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

(2) Vacuum cleaner to remove dirt in two rooms

Developed a simple reflex agent program in Python for the vacuum-cleaner world problem. This program defines the States, Goal State, Goal Test, Actions, Transition Model, and Path Cost. For each possible initial state, the program returns a sequence of actions that leads to the goal state, along with the path cost. Where A and B are the two adjacent rooms, 0 means CLEAN, and 1 means DIRTY.

TSP (Travelling salesman problem)

Algorithm:

- Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point.
- Generate all $(n-1)!$ permutations of cities.
- Calculate the cost of every permutation and keep track of the minimum cost permutation.
- Return the permutation with minimum cost.

Program:

```
# Python3 program to implement traveling salesman
# problem using naive approach.
from sys import maxsize
```

```

from itertools import permutations

# implementation of traveling Salesman Problem
def travellingSalesmanProblem(graph, s):

    # store all vertex apart from source vertex
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)

    # store minimum weight Hamiltonian Cycle
    min_path = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:

        # store current Path weight(cost)
        current_pathweight = 0

        # compute current path weight
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]

        # update minimum
        min_path = min(min_path, current_pathweight)

    return min_path

# Driver Code
if __name__ == "__main__":
    s = 0

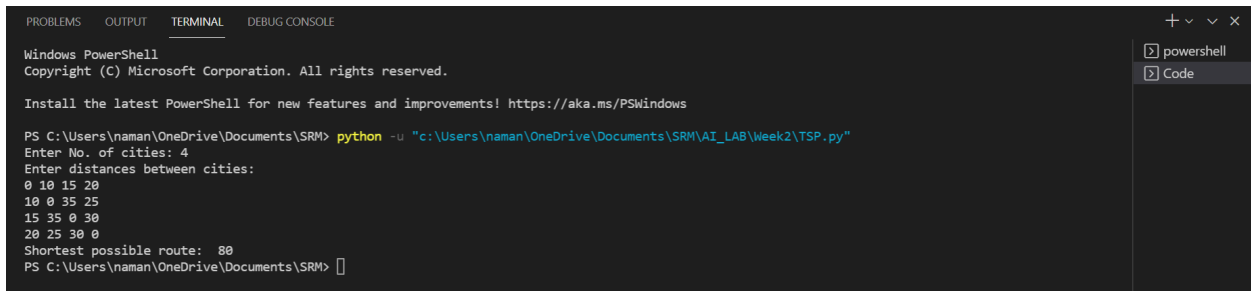
    V = int(input('Enter No. of cities: '))
    print('Enter distances between cities: ')
    a = []
    graph = []

```



```
for i in range(V):  
    a = [int(item) for item in input().split()]  
    graph.append(a)  
    a = []  
  
print("Shortest possible route: ", travellingSalesmanProblem(graph,  
s))
```

Output:



```
Windows PowerShell  
Copyright (C) Microsoft Corporation. All rights reserved.  
  
Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows  
  
PS C:\Users\naman\OneDrive\Documents\SRM> python -u "c:\Users\naman\OneDrive\Documents\SRM\AI_LAB\Week2\TSP.py"  
Enter No. of cities: 4  
Enter distances between cities:  
0 10 15 20  
10 0 35 25  
15 35 0 30  
20 25 30 0  
Shortest possible route: 80  
PS C:\Users\naman\OneDrive\Documents\SRM>
```

Result:

The traveling salesman problem has been successfully implemented and the shortest possible route is found.

Vacuum cleaner to remove dirt in two rooms

Algorithm:

Initially, both the rooms are full of dirt and the vacuum cleaner can reside in any room. And to reach the final goal state, both the rooms should be clean and the vacuum cleaner again can reside in any of the two rooms. The vacuum cleaner can perform the following functions: move left, move right, move forward, move backward, and suck dust. But as there are only two rooms in our problem, the vacuum cleaner performs only the following functions here: move left, move right and suck.

Here the performance of our agent (vacuum cleaner) depends upon many factors such as time taken in cleaning, the path followed in cleaning, the number of moves the agent takes in total, etc. But we consider two main factors for estimating the performance of the agent. They are:

1. Search Cost: How long the agent takes to come up with the solution.
2. Path cost: How expensive each action in the solution is.

Program:

```
def vacuum_world():
    # initializing goal_state
    # 0 indicates Clean and 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    location_input = input("Enter Location of Vacuum: ") #user_input of
location vacuum is placed
    status_input = input("Enter status of " + location_input + ": ")
#user_input if location is dirty or clean
    status_input_complement = input("Enter status of other room: ")
    print("Initial Location Condition" + str(goal_state))

    if location_input == 'A':
        # Location A is Dirty.
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            # suck the dirt and mark it as clean
            goal_state['A'] = '0'
```

```

    cost += 1                                #cost for suck
    print("Cost for CLEANING A " + str(cost))
    print("Location A has been Cleaned.")

    if status_input_complement == '1':
        # if B is Dirty
        print("Location B is Dirty.")
        print("Moving right to the Location B.")
        cost += 1                            #cost for moving right
        print("COST for moving RIGHT" + str(cost))
        # suck the dirt and mark it as clean
        goal_state['B'] = '0'
        cost += 1                            #cost for suck
        print("COST for SUCK " + str(cost))
        print("Location B has been cleaned.")
    else:
        print("No action" + str(cost))
        # suck and mark clean
        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.")
        cost += 1                            #cost for moving right
        print("COST for moving RIGHT " + str(cost))
        # suck the dirt and mark it as clean
        goal_state['B'] = '0'
        cost += 1                            #cost for suck
        print("Cost for SUCK" + str(cost))
        print("Location B has been cleaned.")
    else:
        print("No action " + str(cost))
        print(cost)
        # suck and mark clean
        print("Location B is already clean.")

else:
    print("Vacuum is placed in location B")

```

```

# Location B is Dirty.
if status_input == '1':
    print("Location B is Dirty.")
    # suck the dirt and mark it as clean
    goal_state['B'] = '0'
    cost += 1 # cost for suck
    print("COST for CLEANING " + str(cost))
    print("Location B has been Cleaned.")

    if status_input_complement == '1':
        # if A is Dirty
        print("Location A is Dirty.")
        print("Moving LEFT to the Location A. ")
        cost += 1 # cost for moving right
        print("COST for moving LEFT" + str(cost))
        # suck the dirt and mark it as clean
        goal_state['A'] = '0'
        cost += 1 # cost for suck
        print("COST for SUCK " + str(cost))
        print("Location A has been Cleaned.")

else:
    print(cost)
    # suck and mark clean
    print("Location B is already clean.")

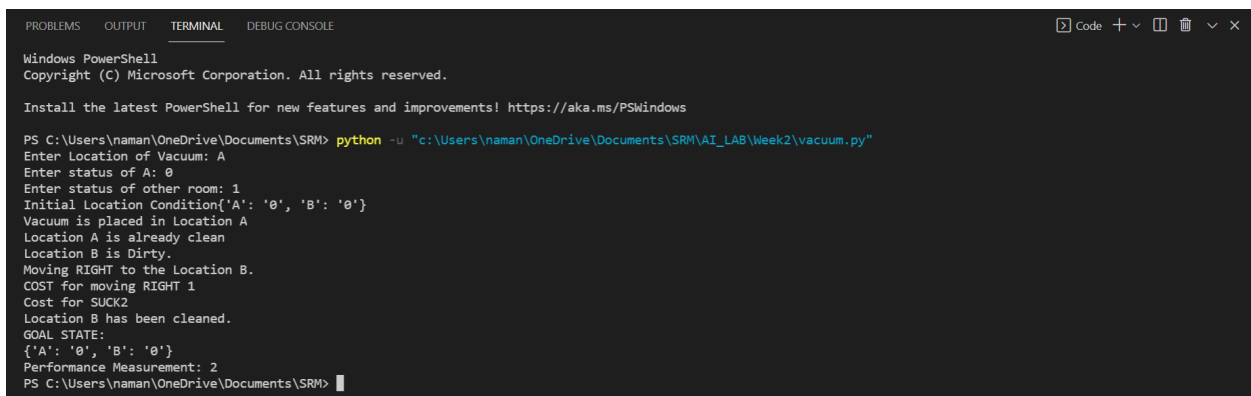
    if status_input_complement == '1': # if A is Dirty
        print("Location A is Dirty.")
        print("Moving LEFT to the Location A. ")
        cost += 1 # cost for moving right
        print("COST for moving LEFT " + str(cost))
        # suck the dirt and mark it as clean
        goal_state['A'] = '0'
        cost += 1 # cost for suck
        print("Cost for SUCK " + str(cost))
        print("Location A has been cleaned. ")
    else:
        print("No action " + str(cost))
        # suck and mark clean
        print("Location A is already clean.")

```

```
# done cleaning
print("GOAL STATE: ")
print(goal_state)
print("Performance Measurement: " + str(cost))

vacuum_world()
```

Output:



```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\naman\OneDrive\Documents\SRM> python -u "c:\Users\naman\OneDrive\Documents\SRM\AI_LAB\Week2\vacuum.py"
Enter Location of Vacuum: A
Enter status of A: 0
Enter status of other room: 1
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is already clean
Location B is Dirty.
Moving RIGHT to the Location B.
COST for moving RIGHT 1
Cost for SUCK2
Location B has been cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
PS C:\Users\naman\OneDrive\Documents\SRM>
```

Result:

Vacuum cleaner to remove dirt in two rooms problem has been successfully implemented and the performance measurement is found.

Artificial Intelligence (18CSC305J)

Experiment 3

Naman Jain
RA1911003010090
B1 CSE

Aim: Implementation of Constraint Satisfaction Problem.

Problem Statement: Given an array of strings, arr[] of size N and a string S, the task is to map integers value in the range [0,9] to every alphabet that occurs in the strings, such that the sum obtained after summing the numbers formed by encoding all strings in the array is equal to the number formed by the string S.

Program:

```
// CPP program for solving cryptographic puzzles
#include <bits/stdc++.h>
using namespace std;

// vector stores 1 corresponding to index
// number which is already assigned
// to any char, otherwise stores 0
vector<int> use(10);

// structure to store char and its corresponding integer
struct node
{
    char c;
    int v;
};

// function check for correct solution
int check(node* nodeArr, const int count, string s1,
          string s2, string s3)
{
    int val1 = 0, val2 = 0, val3 = 0, m = 1, j, i;
```

```

// calculate number corresponding to first string
for (i = s1.length() - 1; i >= 0; i--)
{
    char ch = s1[i];
    for (j = 0; j < count; j++)
        if (nodeArr[j].c == ch)
            break;

    val1 += m * nodeArr[j].v;
    m *= 10;
}
m = 1;

// calculate number corresponding to second string
for (i = s2.length() - 1; i >= 0; i--)
{
    char ch = s2[i];
    for (j = 0; j < count; j++)
        if (nodeArr[j].c == ch)
            break;

    val2 += m * nodeArr[j].v;
    m *= 10;
}
m = 1;

// calculate number corresponding to third string
for (i = s3.length() - 1; i >= 0; i--)
{
    char ch = s3[i];
    for (j = 0; j < count; j++)
        if (nodeArr[j].c == ch)
            break;

    val3 += m * nodeArr[j].v;
    m *= 10;
}

// sum of first two number equal to third return true
if (val3 == (val1 + val2))

```

```

        return 1;

    // else return false
    return 0;
}

// Recursive function to check solution for all permutations
bool permutation(const int count, node* nodeArr, int n,
                string s1, string s2, string s3)
{
    // Base case
    if (n == count - 1)
    {

        // check for all numbers not used yet
        for (int i = 0; i < 10; i++)
        {

            // if not used
            if (use[i] == 0)
            {

                // assign char at index n integer i
                nodeArr[n].v = i;

                // if solution found
                if (check(nodeArr, count, s1, s2, s3) == 1)
                {
                    cout << "\nSolution found: ";
                    for (int j = 0; j < count; j++)
                        cout << " " << nodeArr[j].c << " = "
                            << nodeArr[j].v;
                    return true;
                }
            }
        }
        return false;
    }

    for (int i = 0; i < 10; i++)

```



```

{

    // if ith integer not used yet
    if (use[i] == 0)
    {

        // assign char at index n integer i
        nodeArr[n].v = i;

        // mark it as not available for other char
        use[i] = 1;

        // call recursive function
        if (permutation(count, nodeArr, n + 1, s1, s2, s3))
            return true;

        // backtrack for all other possible solutions
        use[i] = 0;
    }
}

return false;
}

bool solveCryptographic(string s1, string s2,
                        string s3)
{

    // count to store number of unique char
    int count = 0;

    // Length of all three strings
    int l1 = s1.length();
    int l2 = s2.length();
    int l3 = s3.length();

    // vector to store frequency of each char
    vector<int> freq(26);

    for (int i = 0; i < l1; i++)
        ++freq[s1[i] - 'A'];

```

```

    for (int i = 0; i < 12; i++)
        ++freq[s2[i] - 'A'];

    for (int i = 0; i < 13; i++)
        ++freq[s3[i] - 'A'];

    // count number of unique char
    for (int i = 0; i < 26; i++)
        if (freq[i] > 0)
            count++;

    // solution not possible for count greater than 10
    if (count > 10)
    {
        cout << "Invalid strings";
        return 0;
    }

    // array of nodes
    node nodeArr[count];

    // store all unique char in nodeArr
    for (int i = 0, j = 0; i < 26; i++)
    {
        if (freq[i] > 0)
        {
            nodeArr[j].c = char(i + 'A');
            j++;
        }
    }

    return permutation(count, nodeArr, 0, s1, s2, s3);
}

// Driver function
int main()
{
    string s1;
    string s2;
    string s3;

```

```

cout<<"First String: "<<endl;
cin>>s1;
cout<<"Second String: "<<endl;
cin>>s2;
cout<<"Output String: "<<endl;
cin>>s3;

if (solveCryptographic(s1, s2, s3) == false)
    cout << "No solution";
return 0;
}

```

Output:

```

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\naman\OneDrive\Documents\SRM\AI_LAB> cd "c:\Users\naman\OneDrive\Documents\SRM\AI_LAB\Week3\" ; if ($?) { g++ cryptoarithmetic.cpp -o c
ryptoarithmetic } ; if ($?) { .\cryptoarithmetic }
First String:
CROSS
Second String:
ROADS
Output String:
DANGER

Solution found: A = 5 C = 9 D = 1 E = 4 G = 7 N = 8 O = 2 R = 6 S = 3
PS C:\Users\naman\OneDrive\Documents\SRM\AI_LAB\Week3> 

```

C	R	O	S	S		9	6	2	3	3	
R	O	A	D	S	=>	6	2	5	1	3	
D	A	N	G	E	R	1	5	8	7	4	6

Result: The constraint satisfaction problem has been successfully implemented.

Artificial Intelligence (18CSC305J)

Experiment 3

Naman Jain
RA1911003010090
B1 CSE

Aim: Implementation of Graph Coloring Problem.

Problem Statement: The problem is, given m colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using the same color.

Program:

```
# class to represent a graph object
class Graph:

    # Constructor
    def __init__(self, edges, N):

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

        # add edges to the undirected graph
        for (src, dest) in edges:
            self.adj[src].append(dest)
            self.adj[dest].append(src)

# Function to assign colors to vertices of graph
def colorGraph(graph):

    # stores color assigned to each vertex
    result = {}

    # assign color to vertex one by one
    for u in range(N):
```

```

        # set to store color of adjacent vertices of u
        # check colors of adjacent vertices of u and store in set
        assigned = set([result.get(i) for i in graph.adj[u] if i in
result])

        # check for first free color
        color = 1
        for c in assigned:
            if color != c:
                break
            color = color + 1

        # assigns vertex u the first available color
        result[u] = color

    for v in range(N):
        print("Color assigned to vertex", v, "is", colors[result[v]])

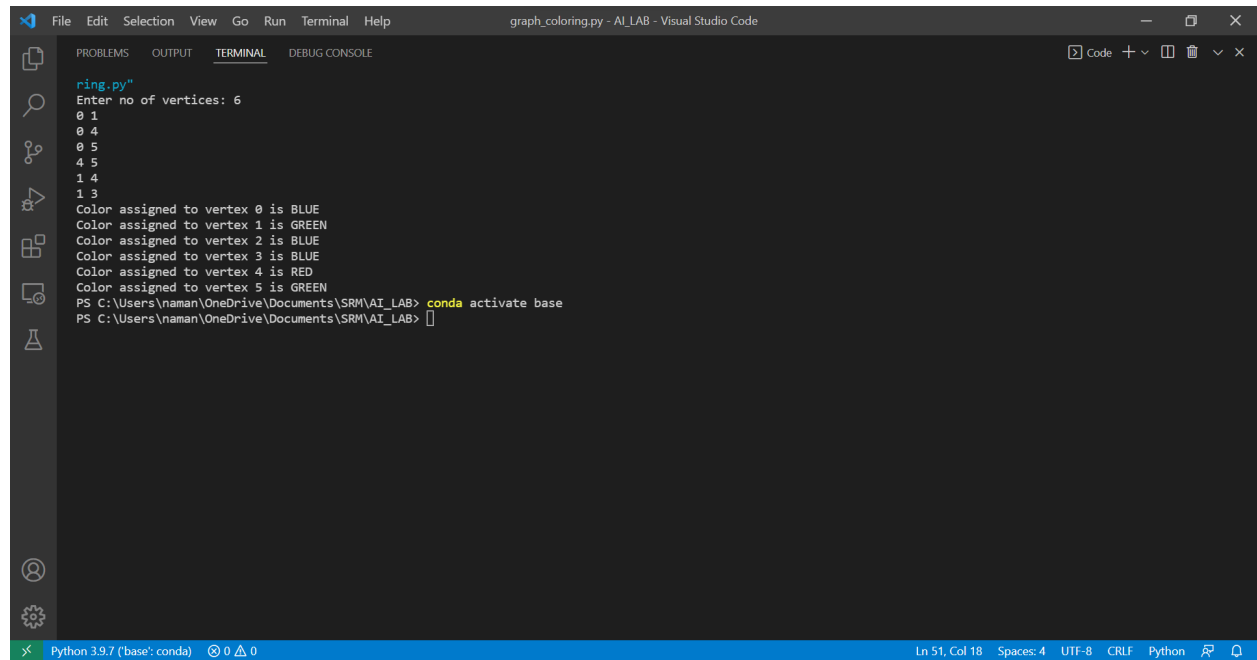
    # Add more colors for graphs with many more vertices
    colors = [
        "MAUVE", "BLUE", "GREEN", "RED", "YELLOW", "ORANGE", "PINK", "BLACK", "BROWN", "WHI
TE",
        "PURPLE", "VIOLET"]

N = int(input("Enter no of vertices: "))
edges = list(tuple(map(int, input().split()))) for r in range(N))

graph = Graph(edges, N)
    # color graph using greedy algorithm
colorGraph(graph)

```

Output:



The screenshot shows a Visual Studio Code window with a terminal open. The terminal output is as follows:

```
ring.py"
Enter no of vertices: 6
0 1
0 4
0 5
4 5
1 4
1 3
Color assigned to vertex 0 is BLUE
Color assigned to vertex 1 is GREEN
Color assigned to vertex 2 is BLUE
Color assigned to vertex 3 is BLUE
Color assigned to vertex 4 is RED
Color assigned to vertex 5 is GREEN
PS C:\Users\naman\OneDrive\Documents\SRM\AI_LAB> conda activate base
PS C:\Users\naman\OneDrive\Documents\SRM\AI_LAB> 
```

The status bar at the bottom indicates the environment is Python 3.9.7 (base: conda) and the file is graph_coloring.py - AI_LAB - Visual Studio Code.

Result: The graph coloring problem has been successfully implemented.

Artificial Intelligence (18CSC305J)

Experiment 4

Naman Jain
RA1911003010090
B1 CSE

Breadth-First Search

Aim: To implement BFS(Breadth-first search) using python.

Problem Statement: Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

Algorithm:

- SET STATUS = 1 (ready state) for each node in G
- Enqueue the starting node A and set its STATUS = 2 (waiting state)
- Dequeue a node N. Process it and set its STATUS = 3 (processed state).
- Enqueue all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)
- Exit

Program:

```
from collections import defaultdict

# This class represents a directed graph
# using adjacency list representation
class Graph:

    # Constructor
    def __init__(self):
```

```

        self.graph = defaultdict(list)

    def addEdge(self,u,v):
        self.graph[u].append(v)

    def BFS(self, s):

        visited = [False] * (max(self.graph) + 1)

        queue = []

        queue.append(s)
        visited[s] = True

        while queue:

            s = queue.pop(0)
            print (s, end = " ")

            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True

g = Graph()
v = int(input('Enter no. of vertices '))
edges = int(input('Enter no. of edges '))
for i in range(edges):
    x,y = input().split(" ")
    g.addEdge(int(x),int(y))

start = int(input('Enter start node '))

g.BFS(start)

```


Output:

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Users\naman\OneDrive\Documents\SRM> conda activate base
PS C:\Users\naman\OneDrive\Documents\SRM> python -u "c:\Users\naman\OneDrive\Documents\SRM\AI_LAB\Week4\tempCodeRunnerFile.py"
PS C:\Users\naman\OneDrive\Documents\SRM> python -u "c:\Users\naman\OneDrive\Documents\SRM\AI_LAB\Week4\BFS.py"
Enter no. of vertices 6
Enter no. of edges 8
2 2
3 2
3 4
4 8
5 3
5 7
7 8
8 8
Enter start node 5
5 3 7 2 4 8
PS C:\Users\naman\OneDrive\Documents\SRM> 
```

Result: BFS is successfully implemented using python.

Depth First Search

Aim: To implement DFS(Depth-first search) using python.

Problem Statement: Depth-first search (DFS) algorithm starts with the initial node of graph G, and then go to deeper and deeper until we find the goal node or the node which has no children. The algorithm then backtracks from the dead-end towards the most recent node that is yet to be completely unexplored.

Algorithm:

- SET STATUS = 1 (ready state) for each node in G
- Push the starting node A on the stack and set its STATUS = 2 (waiting state)
- Pop the top node N. Process it and set its STATUS = 3 (processed state) Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

Program:

```
from collections import defaultdict

class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self, v, visited):
```

```

        visited.add(v)
        print(v, end=' ')

        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)

    def DFS(self, v):

        visited = set()

        self.DFSUtil(v, visited)

# Driver code

g = Graph()
v = int(input('Enter no. of vertices '))
edges = int(input('Enter No. of edges '))
for i in range(edges):
    x,y = input().split(" ")
    g.addEdge(int(x),int(y))

start = int(input('Enter start node '))
g.DFS(start)

```

Output:

```

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Users\naman\OneDrive\Documents\SRM> python -u "c:\Users\naman\OneDrive\Documents\SRM\AI_LAB\Week4\DFS.py"
Enter no. of vertices 6
Enter No. of edges 8
2 2
3 2
3 4
4 8
5 3
5 7
7 8
8 8
Enter start node 5
5 3 2 4 8 7
PS C:\Users\naman\OneDrive\Documents\SRM> 

```

Result: DFS is successfully implemented using python.

Artificial Intelligence (18CSC305J)
Experiment 5

Naman Jain
RA1911003010090
B1 CSE

Best First Search

Aim: To implement the Best first search using python.

Problem Statement: Best First Search uses an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search. We use a priority queue to store the costs of nodes. So the implementation is a variation of BFS, we just need to change Queue to PriorityQueue.

Algorithm:

- Place the starting node into the OPEN list.
- If the OPEN list is empty, Stop and return failure.
- Remove the node n , from the OPEN list which has the lowest value of $h(n)$, and place it in the CLOSED list.
- Expand the node n , and generate the successors of node n .
- Check each successor of node n , and find whether any node is a goal node or not. If any successor node is the goal node, then return success and terminate the search, else proceed to Step 6.
- For each successor node, the algorithm checks for evaluation function $f(n)$, and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both lists, then add it to the OPEN list.
- Return to Step 2.

Program:

```
from queue import PriorityQueue

# Function For Implementing Best First Search
# Gives output path having lowest cost

def best_first_search(source, target, n):
    visited = [0] * n
    visited[source] = True
    pq = PriorityQueue()
    pq.put((0, source))
    while pq.empty() == False:
        u = pq.get()[1]
        # Displaying the path having lowest cost

        print(u, end=" ")
        if u == target:
            break

        for v, c in graph[u]:
            if visited[v] == False:
                visited[v] = True
                pq.put((c, v))
    print()

# Function for adding edges to graph

def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))

v = int(input('Enter no. of vertices '))
graph = [[] for i in range(v)]
for i in range(v-1):
    x,y,z = input().split(" ")
    addedge(int(x),int(y),int(z))
```

```

# The nodes shown in above example(by alphabets) are
# implemented using integers addedge(x,y,cost);

source = int(input('Enter source: '))

target = int(input('Enter target: '))

best_first_search(source, target, v)

```

Output:

```

PS C:\Users\naman\OneDrive\Documents\SRM> python -u "c:\Users\naman\OneDrive\Documents\SRM\AI_LAB\Week5\BestFS.py"
Enter no. of vertices 14
0 1 3
0 2 6
0 3 5
1 4 9
1 5 8
2 6 12
2 7 14
3 8 7
8 9 5
8 10 6
9 11 1
9 12 10
9 13 2
Enter source: 0
Enter target: 9
0 1 3 2 8 9
PS C:\Users\naman\OneDrive\Documents\SRM> conda activate base
PS C:\Users\naman\OneDrive\Documents\SRM>

```

Result: Best first search is successfully implemented using python.

A* Best First Search

Aim: To implement the A* Best first search using python.

Problem Statement: A* search is the most commonly known form of best-first search. It uses the heuristic function $h(n)$, and cost to reach the node n from the start state $g(n)$. It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function.

Algorithm:

- Place the starting node in the OPEN list.
- Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
- Select the node from the OPEN list which has the smallest value of evaluation function ($g+h$), if node n is goal node then return success and stop, otherwise
- Expand node n and generate all of its successors, and put n into the closed list. For each successor n' , check whether n' is already in the OPEN or CLOSED list, if not then compute the evaluation function for n' and place it into the open list.
- Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest $g(n')$ value.
- Return to Step 2.

Program:

```
def aStarAlgo(start_node, stop_node):  
  
    open_set = set(start_node)  
    closed_set = set()  
    g = {} #store distance from starting node  
    parents = {}# parents contains an adjacency map of all nodes  
  
    #dittance of starting node from itself is zero
```

```

g[start_node] = 0
#start_node is root node i.e it has no parent nodes
#so start_node is set to its own parent node
parents[start_node] = start_node

while len(open_set) > 0:
    n = None

    #node with lowest f() is found
    for v in open_set:
        if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
            n = v

    if n == stop_node or Graph_nodes[n] == None:
        pass
    else:
        for (m, weight) in get_neighbors(n):
            #nodes 'm' not in first and last set are added to
first
            #n is set its parent
            if m not in open_set and m not in closed_set:
                open_set.add(m)
                parents[m] = n
                g[m] = g[n] + weight

            #for each node m,compare its distance from start i.e
g(m) to the
            #from start through n node
            else:
                if g[m] > g[n] + weight:
                    #update g(m)
                    g[m] = g[n] + weight
                    #change parent of m to n
                    parents[m] = n

                #if m in closed set,remove and add to open
                if m in closed_set:

```



```

        closed_set.remove(m)
        open_set.add(m)

    if n == None:
        print('Path does not exist!')
        return None

    # if the current node is the stop_node
    # then we begin reconstructin the path from it to the
start_node

    if n == stop_node:
        path = []

        while parents[n] != n:
            path.append(n)
            n = parents[n]

        path.append(start_node)

        path.reverse()

        print('Path found: {}'.format(path))
        return path

    # remove n from the open_list, and add it to closed_list
    # because all of his neighbors were inspected
    open_set.remove(n)
    closed_set.add(n)

    print('Path does not exist!')
    return None

#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None

```

```

#for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 99,
        'D': 1,
        'E': 7,
        'G': 0,

    }

    return H_dist[n]

#Describe your graph here
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1), ('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],
}

aStarAlgo('A', 'G')

```

Output:

```

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\naman\OneDrive\Documents\SRM> python -u "c:\Users\naman\OneDrive\Documents\SRM\AI_LAB\Week5\A-star.py"
Path found: ['A', 'E', 'D', 'G']
PS C:\Users\naman\OneDrive\Documents\SRM> conda activate base
PS C:\Users\naman\OneDrive\Documents\SRM> █

```

Result: A* Best first search is successfully implemented using python.

Artificial Intelligence (18CSC305J)

Experiment 6

Naman Jain
RA1911003010090
B1 CSE

Unification

Aim: To implement Unification Algorithm.

Algorithm:

- Unification is the process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process.
- It takes two literals as input and makes them identical using substitution.
- Let Ψ_1 and Ψ_2 be two atomic sentences and σ be a unifier such that, $\Psi_1\sigma = \Psi_2\sigma$, then it can be expressed as UNIFY(Ψ_1 , Ψ_2)

Program:

```
def get_index_comma(string):  
    index_list = list()  
    par_count = 0  
    for i in range(len(string)):  
        if string[i] == ',' and par_count == 0:  
            index_list.append(i)  
        elif string[i] == '(':  
            par_count += 1  
        elif string[i] == ')':  
            par_count -= 1  
    return index_list  
  
def is_variable(expr):  
    for i in expr:  
        if i == '(' or i == ')':
```

```

        return False
    return True

def process_expression(expr):
    expr = expr.replace(' ', '')
    index = None
    for i in range(len(expr)):
        if expr[i] == '(':
            index = i
            break
    predicate_symbol = expr[:index]
    expr = expr.replace(predicate_symbol, '')
    expr = expr[1:len(expr) - 1]
    arg_list = list()
    indices = get_index_comma(expr)
    if len(indices) == 0:
        arg_list.append(expr)
    else:
        arg_list.append(expr[:indices[0]])
        for i, j in zip(indices, indices[1:]):
            arg_list.append(expr[i + 1:j])
        arg_list.append(expr[indices[len(indices) - 1] + 1:])
    return predicate_symbol, arg_list

def get_arg_list(expr):
    _, arg_list = process_expression(expr)
    flag = True
    while flag:
        flag = False
        for i in arg_list:
            if not is_variable(i):
                flag = True
                _, tmp = process_expression(i)
                for j in tmp:
                    if j not in arg_list:
                        arg_list.append(j)
                arg_list.remove(i)
    return arg_list

def check_occurs(var, expr):
    arg_list = get_arg_list(expr)

```

```

    if var in arg_list:
        return True
    return False

def unify(expr1, expr2):
    if is_variable(expr1) and is_variable(expr2):
        if expr1 == expr2:
            return 'Null'
        else:
            return False
    elif is_variable(expr1) and not is_variable(expr2):
        if check_occurs(expr1, expr2):
            return False
        else:
            tmp = str(expr2) + '/' + str(expr1)
            return tmp
    elif not is_variable(expr1) and is_variable(expr2):
        if check_occurs(expr2, expr1):
            return False
        else:
            tmp = str(expr1) + '/' + str(expr2)
            return tmp
    else:
        predicate_symbol_1, arg_list_1 = process_expression(expr1)
        predicate_symbol_2, arg_list_2 = process_expression(expr2)
# Step 2
        if predicate_symbol_1 != predicate_symbol_2:
            return False
# Step 3
        elif len(arg_list_1) != len(arg_list_2):
            return False
        else:
# Step 4: Create substitution list
            sub_list = list()
# Step 5:
            for i in range(len(arg_list_1)):
                tmp = unify(arg_list_1[i], arg_list_2[i])
                if not tmp:
                    return False
                elif tmp == 'Null':

```

```

        pass
    else:
        if type(tmp) == list:
            for j in tmp:
                sub_list.append(j)
        else:
            sub_list.append(tmp)

# Step 6
return sub_list

if __name__ == '__main__':

    f1 = 'Q(a, g(x, a), f(y))'
    f2 = 'Q(a, g(f(b), a), x)'
    # f1 = input('f1 : ')
    # f2 = input('f2 : ')
    result = unify(f1, f2)
    if not result:
        print('The process of Unification failed!')
    else:
        print('The process of Unification successful!')
        print(result)

```

Output:

```

PS C:\Users\naman\OneDrive\Documents\SRM> python -u "c:\Users\naman\OneDrive\Documents\SRM\AI_LAB\Week6\Unification.py"
The process of Unification successful!
['f(b)/x', 'f(y)/x']
PS C:\Users\naman\OneDrive\Documents\SRM>

```

Result: Unification algorithm successfully implemented in python.

Artificial Intelligence (18CSC305J)

Experiment 6

Naman Jain
RA1911003010090
B1 CSE

Resolution

Aim: To implement a resolution algorithm.

Algorithm:

- Conversion of Facts into FOL - In the first step, we will convert all the given statements into their first-order logic.
- Conversion of FOL into CNF - In First-order logic resolution, it is required to convert the FOL into CNF as CNF form makes it easier for resolution proofs.
- Negate the statement to be proved - In this statement, we will apply negation to the conclusion statements, which will be written as $\neg \text{likes}(\text{John}, \text{Peanuts})$
- Draw Resolution graph - Now in this step, we will solve the problem by resolution tree using substitution.

Program:

```
import copy
import time

class Parameter:
    variable_count = 1

    def __init__(self, name=None):
        if name:
            self.type = "Constant"
            self.name = name
        else:
```

```

        self.type = "Variable"
        self.name = "v" + str(Parameter.variable_count)
        Parameter.variable_count += 1

    def isConstant(self):
        return self.type == "Constant"

    def unify(self, type_, name):
        self.type = type_
        self.name = name

    def __eq__(self, other):
        return self.name == other.name

    def __str__(self):
        return self.name

class Predicate:
    def __init__(self, name, params):
        self.name = name
        self.params = params

    def __eq__(self, other):
        return self.name == other.name and all(a == b for a, b in
zip(self.params, other.params))

    def __str__(self):
        return self.name + "(" + ",".join(str(x) for x in self.params) +
        ")"

    def getNegatedPredicate(self):
        return Predicate(negatePredicate(self.name), self.params)

class Sentence:
    sentence_count = 0

    def __init__(self, string):
        self.sentence_index = Sentence.sentence_count
        Sentence.sentence_count += 1

```



```

self.predicates = []
self.variable_map = {}
local = {}

for predicate in string.split("|"):
    name = predicate[:predicate.find("(")]
    params = []

    for param in predicate[predicate.find("(") + 1:
predicate.find(")"]].split(","):
        if param[0].islower():
            if param not in local: # Variable
                local[param] = Parameter()
                self.variable_map[local[param].name] =
local[param]

                new_param = local[param]
            else:
                new_param = Parameter(param)
                self.variable_map[param] = new_param

            params.append(new_param)

    self.predicates.append(Predicate(name, params))

def getPredicates(self):
    return [predicate.name for predicate in self.predicates]

def findPredicates(self, name):
    return [predicate for predicate in self.predicates if
predicate.name == name]

def removePredicate(self, predicate):
    self.predicates.remove(predicate)
    for key, val in self.variable_map.items():
        if not val:
            self.variable_map.pop(key)

def containsVariable(self):
    return any(not param.isConstant() for param in
self.variable_map.values())

```

```

def __eq__(self, other):
    if len(self.predicates) == 1 and self.predicates[0] == other:
        return True
    return False

def __str__(self):
    return "".join([str(predicate) for predicate in self.predicates])

class KB:
    def __init__(self, inputSentences):
        self.inputSentences = [x.replace(" ", "") for x in inputSentences]
        self.sentences = []
        self.sentence_map = {}

    def prepareKB(self):
        self.convertSentencesToCNF()
        for sentence_string in self.inputSentences:
            sentence = Sentence(sentence_string)
            for predicate in sentence.getPredicates():
                self.sentence_map[predicate] =
self.sentence_map.get(predicate, []) + [sentence]

    def convertSentencesToCNF(self):
        for sentenceIdx in range(len(self.inputSentences)):
            if "=>" in self.inputSentences[sentenceIdx]: # Do negation of
the Premise and add them as literal
                self.inputSentences[sentenceIdx] =
negateAntecedent(self.inputSentences[sentenceIdx])

    def askQueries(self, queryList):
        results = []

        for query in queryList:
            negatedQuery = Sentence(negatePredicate(query.replace(" ",
"")))

            negatedPredicate = negatedQuery.predicates[0]
            prev_sentence_map = copy.deepcopy(self.sentence_map)
            self.sentence_map[negatedPredicate.name] =
self.sentence_map.get(negatedPredicate.name, []) + [negatedQuery]

```

```

        self.timeLimit = time.time() + 40

        try:
            result = self.resolve([negatedPredicate],
[False]*(len(self.inputSentences) + 1))
        except:
            result = False

        self.sentence_map = prev_sentence_map

        if result:
            results.append("TRUE")
        else:
            results.append("FALSE")

    return results

def resolve(self, queryStack, visited, depth=0):
    if time.time() > self.timeLimit:
        raise Exception
    if queryStack:
        query = queryStack.pop(-1)
        negatedQuery = query.getNegatedPredicate()
        queryPredicateName = negatedQuery.name
        if queryPredicateName not in self.sentence_map:
            return False
        else:
            queryPredicate = negatedQuery
            for kb_sentence in self.sentence_map[queryPredicateName]:
                if not visited[kb_sentence.sentence_index]:
                    for kbPredicate in
kb_sentence.findPredicates(queryPredicateName):

                        canUnify, substitution =
performUnification(copy.deepcopy(queryPredicate),
copy.deepcopy(kbPredicate))

                        if canUnify:
                            newSentence = copy.deepcopy(kb_sentence)
                            newSentence.removePredicate(kbPredicate)

```

```

        newQueryStack = copy.deepcopy(queryStack)

        if substitution:
            for old, new in substitution.items():
                if old in
newSentence.variable_map:
                    parameter =
newSentence.variable_map[old]

newSentence.variable_map.pop(old)
                    parameter.unify("Variable" if
new[0].islower() else "Constant", new)
                    newSentence.variable_map[new]
= parameter

            for predicate in newQueryStack:
                for index, param in
enumerate(predicate.params):
                    if param.name in substitution:
                        new =
substitution[param.name]

predicate.params[index].unify("Variable" if new[0].islower() else
"Constant", new)

            for predicate in newSentence.predicates:
                newQueryStack.append(predicate)

            new_visited = copy.deepcopy(visited)
            if kb_sentence.containsVariable() and
len(kb_sentence.predicates) > 1:

new_visited[kb_sentence.sentence_index] = True

            if self.resolve(newQueryStack,
new_visited, depth + 1):
                return True

            return False
        return True

```

```

def performUnification(queryPredicate, kbPredicate):
    substitution = {}
    if queryPredicate == kbPredicate:
        return True, {}
    else:
        for query, kb in zip(queryPredicate.params, kbPredicate.params):
            if query == kb:
                continue
            if kb.isConstant():
                if not query.isConstant():
                    if query.name not in substitution:
                        substitution[query.name] = kb.name
                    elif substitution[query.name] != kb.name:
                        return False, {}
                    query.unify("Constant", kb.name)
                else:
                    return False, {}
            else:
                if not query.isConstant():
                    if kb.name not in substitution:
                        substitution[kb.name] = query.name
                    elif substitution[kb.name] != query.name:
                        return False, {}
                    kb.unify("Variable", query.name)
                else:
                    if kb.name not in substitution:
                        substitution[kb.name] = query.name
                    elif substitution[kb.name] != query.name:
                        return False, {}
        return True, substitution

def negatePredicate(predicate):
    return predicate[1:] if predicate[0] == "~" else "~" + predicate

def negateAntecedent(sentence):
    antecedent = sentence[:sentence.find("=>")]
    premise = []

```

```

        for predicate in antecedent.split("&"):
            premise.append(negatePredicate(predicate))

    premise.append(sentence[sentence.find("=>") + 2:])
    return "|".join(premise)

def getInput(filename):
    with open(filename, "r") as file:
        noOfQueries = int(file.readline().strip())
        inputQueries = [file.readline().strip() for _ in
range(noOfQueries)]
        noOfSentences = int(file.readline().strip())
        inputSentences = [file.readline().strip() for _ in
range(noOfSentences)]
        return inputQueries, inputSentences

def printOutput(filename, results):
    print(results)
    with open(filename, "w") as file:
        for line in results:
            file.write(line)
            file.write("\n")
    file.close()

if __name__ == '__main__':
    inputQueries_, inputSentences_ = getInput("AI_LAB\Week6\input.txt")
    knowledgeBase = KB(inputSentences_)
    knowledgeBase.prepareKB()
    results_ = knowledgeBase.askQueries(inputQueries_)
printOutput("output.txt", results_)

```

Input:

```
1
B (John)
2
A (x)
~A (x) | B (John)
```

Output:

```
PS C:\Users\naman\OneDrive\Documents\SRM> python -u "c:\Users\naman\OneDrive\Documents\SRM\AI_LAB\Week6\resolution.py"
['TRUE']
PS C:\Users\naman\OneDrive\Documents\SRM>
```

Result: The concept of the Resolution algorithm has been successfully implemented and executed using Python

Artificial Intelligence (18CSC305J)

Experiment 7

Naman Jain
RA1911003010090
B1 CSE

Aim: To implement uncertain methods for Dempster Shafer's Theory.

Algorithm:

- Consider all possible outcomes.
- Belief will lead to believe in some possibility by bringing out some evidence.
- Plausibility will make evidence compatible with possible outcomes.
- Set of possible conclusion (P): {p1, p2....pn} where P is a set of possible conclusions and cannot be exhaustive, i.e. at least one (p)_i must be true.
- (p)_i must be mutually exclusive.
- Power Set will contain 2ⁿ elements where n is the number of elements in the possible set.

Program:

```
import argparse
import operator
##### '' is chosen to represent the set of all elements
def getkeys(structure):
    return set([elem[0] for elem in structure])
import json

def combination(d1, d2):
    united = set(d1.keys()).union(set(d2.keys()))
    result=dict.fromkeys(united,0)#init an dictionary with union of keys
                                #from both sets
                                #and init values with 0

## Combination
```



```

for i in d1.keys():
    for j in d2.keys():
        if {' '}==i and {' '}==j:#for intersection between ' ' '
            result[i]+=d1[i]*d2[j]
        else:
            if {' '}==i:# for case ' ' 'char'
                result[j]+=d1[i]*d2[j]
            if {' '}==j:# for case 'char' ' '
                result[i]+=d1[i]*d2[j]
            if {' '}!=i and {' '}!=j:
                common = frozenset(i.intersection(j))#save
intersection
                for k in result.keys():
                    if (len(common)!=0 and(common==k)):#check if
previous intersection is in dict keys
                        result[k] += d1[i]*d2[j]#if yes, apply the
formula

                        break

##Normalisation
#Round for dict's values
for i in result.keys():
    result[i] = round(result[i],4)

#Round for sum of all values
f= sum(list(result.values()))
f=round(f,4)
#divide and round
for i in result.keys():
    result[i] =round(result[i]/f,4)

return result

def transformset(input):
    return dict((frozenset(elem[0]),elem[1]) for elem in input)

def get_mass(allLines):
    mass = {}
    previousLine = {}
    currentLine = {}
    previousLine = transformset(ast.literal_eval(allLines[0]))

```

```

#TODO: check if sum equals to 1
for line in range(1,len(allLines)):
    if allLines[line][0]!='#':
        currentLine = transformset(ast.literal_eval(allLines[line]))
        mass = combination(previousLine,currentLine)
        previousLine=mass
return mass.copy()

def get_beliefs(masses):
    belief = masses.copy()
    for i in belief.keys():
        for j in belief.keys():
            if(i!=j):
                #belief for a key = it's mass + masses of existent subsets
                if i.issuperset(j) and {' ')!=i and {' ')!=j:
                    belief[i]+=masses[j]
    for i in belief.keys():
        belief[i] = round(belief[i],4) #round with 4 digits
    return belief

def get_plausibility(masses):
    plausibility = masses.copy()

    for i in plausibility.keys():
        plausibility[i] = 0 #init elements with 0
    for i in plausibility.keys():
        for j in plausibility.keys():
            if len(i.intersection(j))!=0 and {' ')!=i:# if intersection of
i and j is not None and i is not ' '
                plausibility[i]+=masses[j];#add mass of j to
plausibilities of i
            if {' ')==j:
                plausibility[i]+=masses[j]#if j is ' ', add its mass to i

    for i in plausibility.keys():
        plausibility[i] = round(plausibility[i],4) #round 4 digits
    return plausibility

def filter_results(beliefs, plausibility):
    finalSet = {}

```

```

        #Filter elements with same values of beliefs and plausibility with
        supersets(superset.bel=this.bel, etc...)
        for elem in beliefs.keys():
            # ssetFlag = False
            # for elemb in beliefs.keys():
            #     if elem!=elemb and elemb.issuperset(elem) and
            beliefs[elem]==beliefs[elemb] and plausibility[elem]==plausibility[elemb]:
            #         ssetFlag = True
            #         break
            # if ssetFlag == False and plausibility[elem]!=0.0:
            #     finalSet[elem] = beliefs[elem]
            finalSet[elem] = beliefs[elem]
        return finalSet

def get_final_result(elements, plausibility):
    resultString = ''
    for elem in elements:
        setOfElems = elem[0]
        movieTypes = str(setOfElems)

        # for element in setOfElems:
        #     movieTypes += decodings[letter]+' '
        if elem[0] != {' '}:
            resultString += movieTypes[11:-2]+' ['+str(elem[1])+',
'+str(plausibility[elem[0]])+']\n'
    return resultString

parser = argparse.ArgumentParser()
parser.add_argument('--filename', type=str,
                    help='Path to file with reviews including file name')
args = parser.parse_args()
print(str(args.filename))

# pathToFile = args.filename
pathToFile = 'AI_LAB\Week7\review.txt'
from pathlib import Path
import ast
mass={}
my_file = Path(pathToFile)
if my_file.is_file():

```

```

with open(pathToFile) as f:
    lines = f.readlines()
if len(lines)>1:
    mass = get_mass(lines)
    # print(mass)
    # all = frozenset({' '})
    # if all in mass:
    #     del mass[all]
    beliefs = get_beliefs(mass)
    # print(beliefs)
    plausibility = get_plausibility(mass)
    # print(plausibility)
    final_set = filter_results(beliefs, plausibility)
    # # print(final_set)
    # # #order elements
    all_elements=sorted(final_set.items(),
key=operator.itemgetter(1),reverse=True)

    print("Intervals")
    print(get_final_result(all_elements, plausibility))

```

Input:

```

[[{'cold', 'flu'},0.6], [{'meningitis'},0.2], [{'indigestion'},0.1],
[{' '},0.1]]
[[{'meningitis','indigestion'},0.7], [{' '},0.3]]
[[{'cold','flu','indigestion'},0.99], [{' '},0.01]]
#cold, flu, meningitis or indigestion
#[[{'Big'},0.8], [{' '},0.2]]
#[[{'Big'},0.6], [{' '},0.4]]
#
#[[{'Big'},0.8], [{' '},0.2]]
#[[{'Small'},0.6], [{' '},0.4]]

```

```

[[{'western'},0.65], [{ 'horror'},0.25]]
[[{'western','comedy'},0.85], [{ 'horror'},0.15]]
[[{'western'},0.87], [{ 'horror'},0.13]]
#[[{'western','horror','comedy'},0.9], [{ 'drama','comedy'},0.10]]
#[[{'western','horror'},0.55], [{ 'horror'},0.45]]
#[[{'western','comedy'},0.87], [{ 'drama','horror'},0.13]]
#[[{'western','comedy','drama'},0.77], [{ 'horror'},0.23]]
#[[{'comedy','drama'},0.77], [{ ''},0.23]]
#[[{'western','horror','drama'},0.77], [{ 'comedy'},0.23]]

```

Output:

```

PS C:\Users\naman\OneDrive\Documents\SRM> python -u "c:\Users\naman\OneDrive\Documents\SRM\AI_LAB\Week7\dempster.py"
None
PS C:\Users\naman\OneDrive\Documents\SRM>

```

Result: Uncertain methods were implemented for Dempster Shafer's Theory.

Artificial Intelligence (18CSC305J)
Experiment 8

Naman Jain
RA1911003010090
B1 CSE

Aim: To implement any Machine Learning algorithm to solve a problem.

Machine Learning: Linear regression with multiple variables.

Algorithm:

- Import required libraries
- Define the dataset
- Plot the data points for better visualization
- Calculate coefficient rule:
 - initialize the parameters
 - predict the rules of a dependent variable
 - calculate error in prediction for all
 - calculate partial derivatives
 - calculate the cost of each step
 - update values
- Compute accuracy and error.

Result: Implementation of learning algorithms completed.

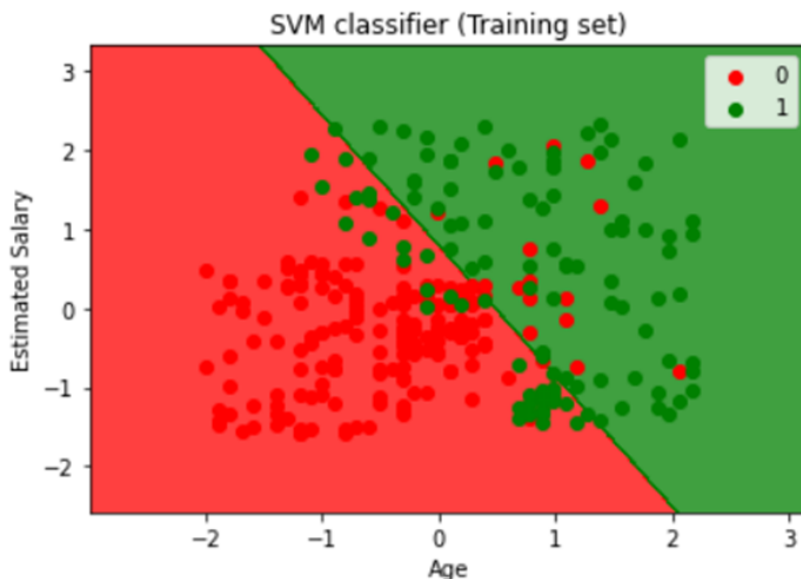
Support Vector Machine (SVM)

Aim: The objective of the SVM algorithm is to find a hyperplane in an N-dimensional space that distinctly classifies the data points.

Algorithm:

- Load the important libraries.
- Import dataset.
- Divide the dataset into train and test.
- Feature Scaling.
- Initializing the SVM classifier model.
- Fitting the SVM classifier model.
- Coming up with predictions.
- Evaluating the model's performance.

Output:



Result: The SVM classifier has divided the users into two regions. Users who purchased the SUV are in the red region with the red scatter points. And users who did not purchase the SUV are in the green region with green scatter points. The hyperplane has divided the two classes into Purchased and not purchased variables.

Apriori Algorithm

Aim: To find the association between the items whether it is strongly or weakly associated.

Algorithm:

- Computing the support for each individual item.
- Deciding on the support threshold.
- Select the frequent items.
- Finding the support of the frequent itemset.
- Repeat for larger sets.
- Generate Association Rules and compute confidence.
- Compute lift.

Output:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
30	(Toast)	(Coffee)	0.033597	0.478394	0.023666	0.704403	1.472431	0.007593	1.764582
28	(Spanish Brunch)	(Coffee)	0.018172	0.478394	0.010882	0.598837	1.251766	0.002189	1.300235
19	(Medialuna)	(Coffee)	0.061807	0.478394	0.035182	0.569231	1.189878	0.005614	1.210871
23	(Pastry)	(Coffee)	0.086107	0.478394	0.047544	0.552147	1.154168	0.006351	1.164682
0	(Alfajores)	(Coffee)	0.036344	0.478394	0.019651	0.540698	1.130235	0.002264	1.135648
17	(Juice)	(Coffee)	0.038563	0.478394	0.020602	0.534247	1.116750	0.002154	1.119919
25	(Sandwich)	(Coffee)	0.071844	0.478394	0.038246	0.532353	1.112792	0.003877	1.115384
7	(Cake)	(Coffee)	0.103856	0.478394	0.054728	0.526958	1.101515	0.005044	1.102664
27	(Scone)	(Coffee)	0.034548	0.478394	0.018067	0.522936	1.093107	0.001539	1.093366
12	(Cookies)	(Coffee)	0.054411	0.478394	0.028209	0.518447	1.083723	0.002179	1.083174
14	(Hot chocolate)	(Coffee)	0.058320	0.478394	0.029583	0.507246	1.060311	0.001683	1.058553
5	(Brownie)	(Coffee)	0.040042	0.478394	0.019651	0.490765	1.025860	0.000495	1.024293
20	(Muffin)	(Coffee)	0.038457	0.478394	0.018806	0.489011	1.022193	0.000408	1.020777
3	(Pastry)	(Bread)	0.086107	0.327205	0.029160	0.338650	1.034977	0.000985	1.017305
11	(Cake)	(Tea)	0.103856	0.142631	0.023772	0.228891	1.604781	0.008959	1.111865
38	(Tea, Coffee)	(Cake)	0.049868	0.103856	0.010037	0.201271	1.937977	0.004858	1.121962
33	(Sandwich)	(Tea)	0.071844	0.142631	0.014369	0.200000	1.402222	0.004122	1.071712
8	(Hot chocolate)	(Cake)	0.058320	0.103856	0.011410	0.195652	1.883874	0.005354	1.114125
39	(Coffee, Cake)	(Tea)	0.054728	0.142631	0.010037	0.183398	1.285822	0.002231	1.049923
10	(Tea)	(Cake)	0.142631	0.103856	0.023772	0.166667	1.604781	0.008959	1.075372
27	(Bread)	(Coffee, Bread)	0.086107	0.086107	0.011410	0.132854	1.111878	0.003118	1.016833

Result: The data has been arranged from highest to lowest with respect to confidence.

Linear Regression Algorithm

Aim: To predictive analysis and find the linear relationship between the salary and years of experience.

Algorithm:

- Load the important libraries.
- Import dataset.
- Divide the dataset into train and test.
- Fitting the Simple Linear Regression model to the training dataset.
- Prediction of Test and Training set result.
- Visualizing the Test set results

Output:



Result: The linear relationship between the salary and years of experience has successfully been found.

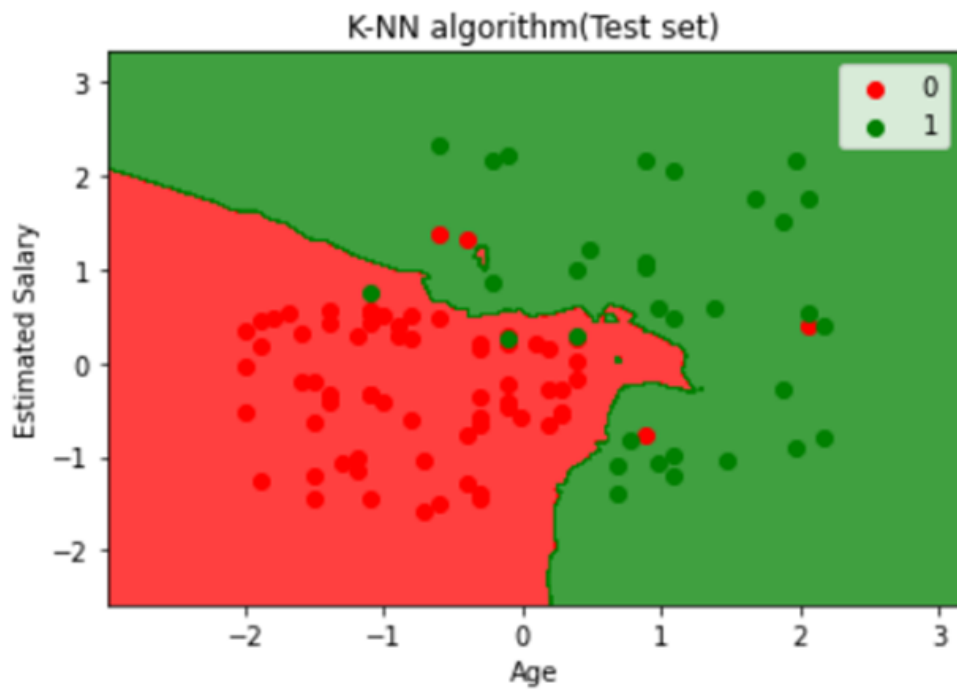
K-Nearest Neighbor Algorithm (KNN)

Aim: To assume the similarity between the new case/data and available cases and put the new case into the category that is most similar to the available categories.

Algorithm:

- Select the number K of the neighbors.
- Calculate the Euclidean distance of K number of neighbors.
- Take the K nearest neighbors as per the calculated Euclidean distance.
- Among these k neighbors, count the number of the data points in each category.
- Assign the new data points to that category for which the number of the neighbor is maximum.
- Steps to implement the K-NN algorithm:
 - Data Pre-processing step
 - Fitting the K-NN algorithm to the Training set
 - Predicting the test result
 - Test accuracy of the result (Creation of Confusion matrix)
 - Visualizing the test set result.

Output:



Result: As we can see in the graph, the predicted output is well good as most of the red points are in the red region and most of the green points are in the green region.

Artificial Intelligence (18CSC305J)

Experiment 9

Naman Jain
RA1911003010090
B1 CSE

Aim: Implementation of sentiment analysis using NLP

Algorithm:

- Load the dataset using pandas.
- Vectorize the data using TF/DF vectorizer.
- Split the dataset into training and test data.
- Use tokenizer into the text to separate texts into separate tokens.]
- Train a SVM model to classify the text.
- Try on test data and improve the model.
- Save model through pickle and run from file directly.

Output:

- Feature Vectors

```
(0, 12442) 0.02702055838392124
(0, 1128) 0.02571589890424715
(0, 4118) 0.05099142562618731
(0, 5847) 0.05037536781236324
(0, 4138) 0.06715571036579193
(0, 8958) 0.04936626445345458
(0, 5246) 0.04732970629650998
(0, 9680) 0.0426920338509739
(0, 7331) 0.04978865745480442
(0, 3603) 0.06715571036579193
(0, 2758) 0.05381945880415767
(0, 9350) 0.02724826521754145
(0, 3136) 0.03719762745122791
(0, 11835) 0.05804405422275094
(0, 2106) 0.04469877474650158
(0, 2540) 0.06276197834586864
(0, 3429) 0.08258798241515057
(0, 3094) 0.04221818354936852
(0, 9583) 0.07306337400869095
(0, 11883) 0.048179601120214625
```

```
(0, 4989) 0.07050555002602447
(0, 4800) 0.026533944026732918
(0, 11024) 0.04245283043327988
(0, 12214) 0.028324563115548446
(0, 9619) 0.02755071979934896
: :
```

- Training time

```
Training time: 8.486371s; Prediction time: 0.826755s
```

```
positive: {'precision': 0.9191919191919192, 'recall': 0.91,
'f1-score': 0.9145728643216081, 'support': 100}
```

```
negative: {'precision': 0.9108910891089109, 'recall': 0.92,
'f1-score': 0.9154228855721394, 'support': 100}
```

- Final Output

```
I am a bad student. I study in a good college SRM.
```

```
[('I', 'PRP'), ('am', 'VBP'), ('a', 'DT'), ('bad', 'JJ'),
('student', 'NN'), ('.', '.'), ('I', 'PRP'), ('study', 'VBP'),
('in', 'IN'), ('a', 'DT'), ('good', 'JJ'), ('college', 'NN'),
('SRM', 'NNP'), ('.', '.')]

```

```
Positive
```

Result: Implementation of sentiment analysis using NLP was executed successfully.

Artificial Intelligence (18CSC305J)

Experiment 10

Naman Jain
RA1911003010090
B1 CSE

Aim: To apply a deep learning method, CNN using python.

Algorithm:

- Import TensorFlow.
- Load fruit dataset.
- Divide the dataset into training and test sets.
- Reshape images.
- Create a CNN model using Keras using sequential CNN.
- Predict the output using the compiled model.

Output:

- Keras

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
0., 0.,  
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
0., 0.,  
0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.,  
0., 0.,  
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
0., 0.,  
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

- Training Image

```
1st training image as array [[[255. 255. 255.]  
[255. 255. 255.]  
[255. 255. 255.]  
...  
[255. 255. 255.]
```

```
[255. 255. 255.]
[255. 255. 255.]]
```

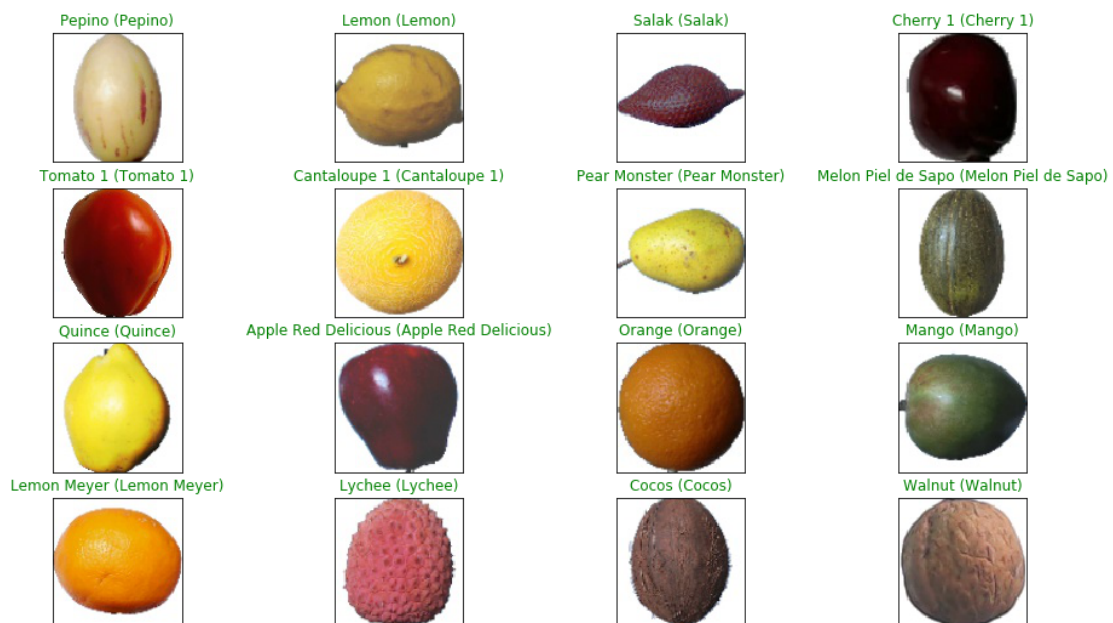
```
[[255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]
 ...
 [255. 255. 255.]
 [255. 255. 255.]
 [255. 255. 255.]]
```

- Layers

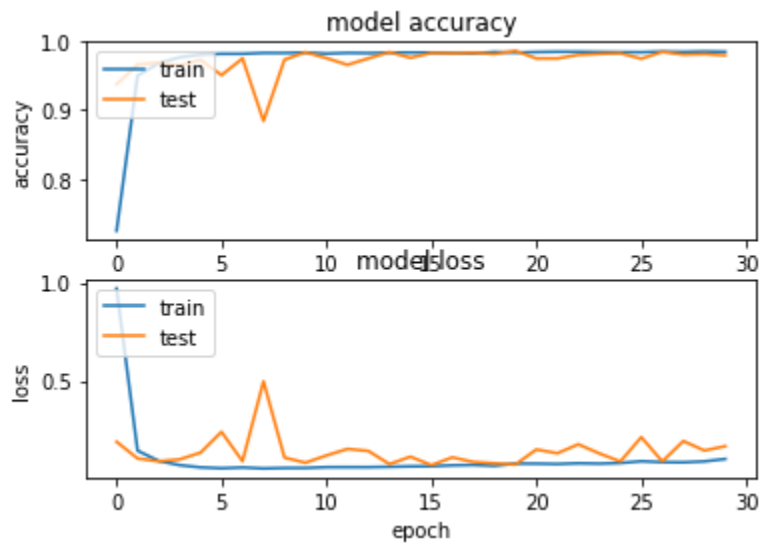
Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 100, 100, 16)	208
activation_1 (Activation)	(None, 100, 100, 16)	0
max_pooling2d_1 (MaxPooling2)	(None, 50, 50, 16)	0
conv2d_2 (Conv2D)	(None, 50, 50, 32)	2080
max_pooling2d_2 (MaxPooling2)	(None, 25, 25, 32)	0
conv2d_3 (Conv2D)	(None, 25, 25, 64)	8256
max_pooling2d_3 (MaxPooling2)	(None, 12, 12, 64)	0
conv2d_4 (Conv2D)	(None, 12, 12, 128)	32896
max_pooling2d_4 (MaxPooling2)	(None, 6, 6, 128)	0
dropout_1 (Dropout)	(None, 6, 6, 128)	0
flatten_1 (Flatten)	(None, 4608)	0
dense_1 (Dense)	(None, 150)	691350
activation_2 (Activation)	(None, 150)	0

dropout_2 (Dropout)	(None, 150)	0
dense_2 (Dense)	(None, 81)	12231
Total params: 747,021		
Trainable params: 747,021		
Non-trainable params: 0		

- Model Visualization



- Accuracy



Result: The application of deep learning methods such as CNN was implemented successfully.