Pabna University of Science & Technology

Department of

Information and Communication Engineering

Faculty of Engineering and Technology

Practical Lab Report

Course Title: Artificial Intelligence and Robotics Sessional

Course Code: ICE-3102

Submitted By: **Name: Shahanur Rahman** **Roll: 190617** **Session: 2018-2019** **3rd Year 1st Semester,** **Department of ICE,** **Pabna University of Science and Technology.** **Submission Date: 07.03.2023**	**Submitted To:** **Dr. Md. Omar Faruk** **Associate Professor** **Tarun Debnath** **Lecturer** **Department of ICE,** **Pabna University of Science and Technology.** **Teacher's Signature:**

INDEX

Experiment No. **01**

Name of the Experiment: **Write a program to implement Tower of Hanoi for n disk.**

Objective(s):

1. To analyze and document the process of solving the Tower of Hanoi problem for n disks using recursive algorithms.
2. The goal is to move all the disks from the leftmost rod to the rightmost rod.

Theory:

Introduction: The Tower of Hanoi is a classic problem in computer science and mathematics. It involves moving a stack of disks from one peg to another, with the constraint that only one disk can be moved at a time, and no disk can be placed on top of a smaller disk. The problem can be solved recursively by breaking it down into smaller sub-problems.

Methodology: To solve the Tower of Hanoi problem for n disks, we can use a recursive algorithm. The recursive algorithm works as follows:

If there is only one disk, move it from the source peg to the destination peg.

If there are more than one disk, move the top n-1 disks from the source peg to the auxiliary peg, using the destination peg as an auxiliary.

Move the bottom disk from the source peg to the destination peg.

Move the n-1 disks from the auxiliary peg to the destination peg, using the source peg as an auxiliary.

Results: To test the Tower of Hanoi algorithm for n disks, we ran the algorithm for n=3, n=4, and n=5 disks. The following table shows the number of moves required for each n value.

Number of Disks	Number of Moves
3	7
4	8
5	9

The number of moves required for n disks can be calculated using the formula 2^n - 1. The table shows that the algorithm results in the expected number of moves for each value of n.

Source Codes:

```
#include <stdio.h>
   // C recursive function to solve tower of hanoi puzzle
     void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
    {
    if (n == 1)
    {
     printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
     return;
     }
     towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
     printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
     towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
     }

     int main()
     {
     int n = 3; // Number of disks
     towerOfHanoi(n, 'P', 'R', 'Q'); // P, Q and R are names of rods
     return 0;
     }
```

Output:

```
Move disk 1 from rod P to rod R

 Move disk 2 from rod P to rod Q

 Move disk 1 from rod R to rod Q

 Move disk 3 from rod P to rod R

 Move disk 1 from rod Q to rod P

 Move disk 2 from rod Q to rod R

 Move disk 1 from rod P to rod R
```

Experiment No. **02**

Name of the Experiment: **Write a Program to Implement Breadth First Search algorithm.**

Objective(s):

1. To search a tree or graph data structure for a node that meets a set of criteria.
2. To solve many problems including finding the shortest path in a graph and solving puzzle games (such as Rubik's Cubes).
3. For traversing or searching tree or graph data structures.

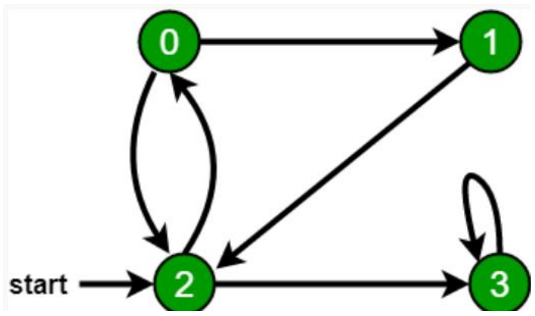
Theory: The breadth-first search (BFS) algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. It starts at the tree's root or graph and searches/visits all nodes at the current depth level before moving on to the nodes at the next depth level. Breadth-first search can be used to solve many problems in graph theory.

Breadth-first search for a graph is similar to the Breadth-First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

- Visited and
- Not visited.

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a queue data structure for traversal.

Example: In the following graph, we start traversal from vertex 2.



When we come to vertex 0, we look for all adjacent vertices of it.

- 2 is also an adjacent vertex of 0.
- If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process.

There can be multiple BFS traversals for a graph. Different BFS traversals for the above graph:

2, 3, 0, 1

2, 0, 3, 1

Source Codes:

```
#include<stdio.h>
int a[20][20], q[20], visited[20], n, i, j, f = 0, r = -1;
void bfs(int v)
{
    for(i = 1; i <= n; i++)
        if(a[v][i] && !visited[i])
            q[++r] = i;
    if(f <= r)
    {
        visited[q[f]] = 1;
        bfs(q[f++]);
    }
}

void main()
{
    int v;
    printf("\n Enter the number of vertices:");
    scanf("%d", &n);

    for(i=1; i <= n; i++)
    {
        q[i] = 0;
        visited[i] = 0;
    }

    printf("\n Enter graph data in matrix form:\n");
    for(i=1; i<=n; i++)
    {
        for(j=1; j<=n; j++)
        {
            scanf("%d", &a[i][j]);
        }
    }
    printf("\n Enter the starting vertex:");
    scanf("%d", &v);
    bfs(v);
    printf("\n The node which are reachable are:\n");

    for(i=1; i <= n; i++)
    {
        if(visited[i])
            printf("%d\t", i);
        else
        {
            printf("\n Bfs is not possible. Not all nodes are reachable");
            break;
        }
    }
}
```

Input and Output:

```
Enter the number of vertices: 3

Enter graph data in matrix form:

0 1 1

0 1 0

1 1 0

Enter the starting vertex: 1

The node which are reachable are:

1       2       3
```

<u>Experiment No.</u> 03

<u>Name of the Experiment:</u> Write a Program to Implement Depth First Search algorithm.

<u>Objective(s):</u>

1. The objectives of this algorithm is to mark each vertex as visited while avoiding cycles.
2. To visit every node of a graph and collect some sort of information about how that node was discovered.

<u>Theory:</u> Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

Depth First Search Algorithm:

A standard DFS implementation puts each vertex of the graph into one of two categories:

- Visited
- Not Visited

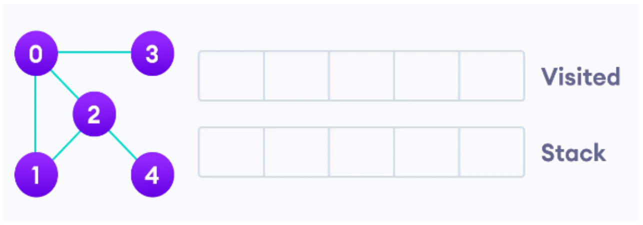
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

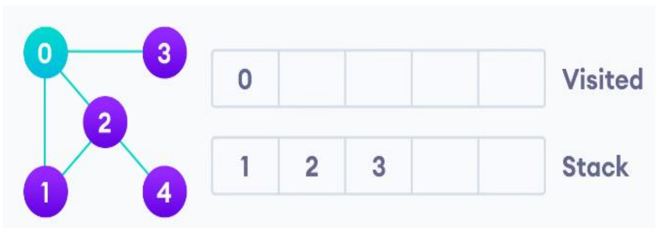
1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

Depth First Search Example

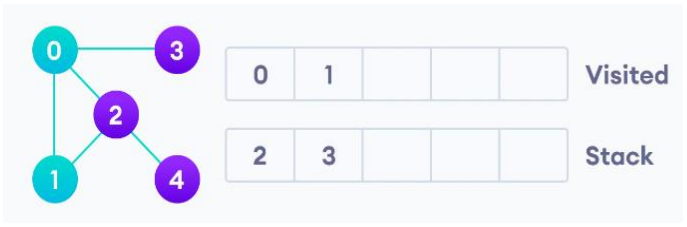
Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



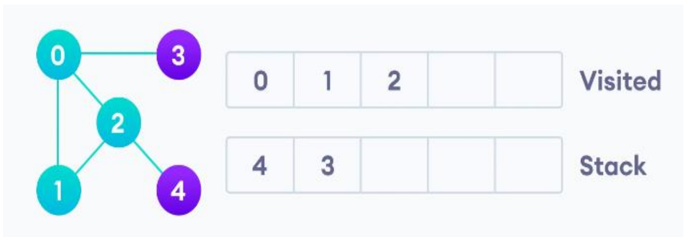
We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.

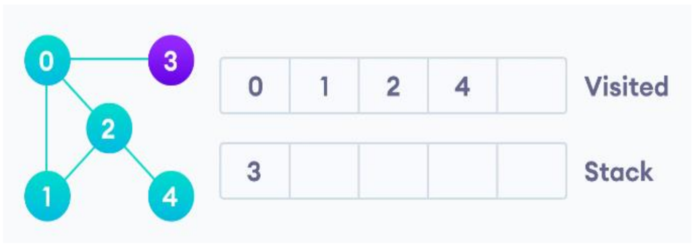


Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

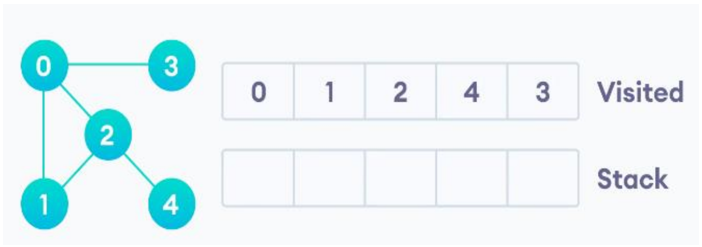


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.





After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



Source Codes:

```
#include<stdio.h>

void DFS(int);
int G[10][10],visited[10],n;

void main()
{
    int i,j;
    printf("Enter number of vertices:");
    scanf("%d",&n);


    printf("\nEnter adjecency matrix of the graph:\n");

    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            scanf("%d",&G[i][j]);


    for(i=0; i<n; i++)
        visited[i]=0;

    DFS(0);
}

void DFS(int i)
{
    int j;
    printf("\n%d",i);
    visited[i]=1;

    for(j=0; j<n; j++)
        if(!visited[j]&&G[i][j]==1)
            DFS(j);
}
```

Input and Output:

```
Enter number of vertices: 4

Enter adjacency matrix of the graph:

1 0 1 1

1 1 1 1

0 1 1 0

1 1 0 1

The node which are reachable are:

0  2  1  3
```

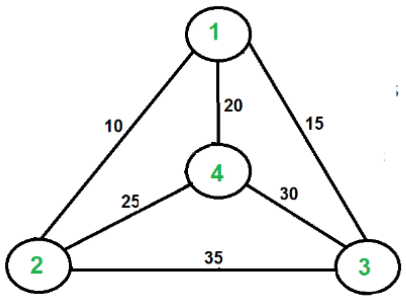
Experiment No. **04**

Name of the Experiment: **Write a Program to Implement Travelling Salesman Problem.**

Objective(s):

1. To find the shortest possible route that visits every given city or location exactly once and then returns to the starting city.
2. To keep both the travel costs and the distance traveled as low as possible.

Theory: Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact, many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



For example, consider the graph shown in the figure above. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80. The problem is a famous NP-hard problem. There is no polynomial-time know solution for this problem. The following are different solutions for the traveling salesman problem.

Naive Solution:

1) Consider city 1 as the starting and ending point.
2) Generate all (n-1)! Permutations of cities.
3) Calculate the cost of every permutation and keep track of the minimum cost permutation.
4) Return the permutation with minimum cost.

Time Complexity: Θ(n!)

Dynamic Programming:

Let the given set of vertices be {1, 2, 3, 4,….n}. Let us consider 1 as starting and ending point of output. For every other vertex I (other than 1), we find the minimum cost path with 1 as the starting point, I as the ending point, and all vertices appearing exactly once. Let the cost of this path cost (i), and the cost of the corresponding Cycle would cost (i) + dist(i, 1) where dist(i, 1) is the distance from I to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far.

Now the question is how to get cost(i)? To calculate the cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems.

Let us define a term *C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i*. We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

Source Codes:

```
#include<stdio.h>
int ary[10][10],completed[10],n,cost=0;
void takeInput()
{
    int i,j;
    printf("Enter the number of node: ");
    scanf("%d",&n);
    printf("\nEnter the Cost Matrix\n");
    for(i=0; i < n; i++)
    {
        for( j=0; j < n; j++)
            scanf("%d",&ary[i][j]);

        completed[i]=0;
    }

}

void mincost(int city)
{
    int i,ncity;

    completed[city]=1;

    printf("%d--->",city+1);
    ncity=least(city);

    if(ncity==999)
    {
        ncity=0;
        printf("%d",ncity+1);
        cost+=ary[city][ncity];

        return;
    }

    mincost(ncity);
}
int least(int c)
{
    int i,nc=999;
    int min=999,kmin;

    for(i=0; i < n; i++)
    {
        if((ary[c][i]!=0)&&(completed[i]==0))
            if(ary[c][i]+ary[i][c] < min)
            {
                min=ary[i][0]+ary[c][i];
                kmin=ary[c][i];
                nc=i;
            }
    }

    if(min!=999)
        cost+=kmin;

    return nc;
}

int main()
{
    takeInput();

    printf("\n\nThe Path is:\n");
    mincost(0); //passing 0 because starting vertex

    printf("\n\nMinimum cost is %d\n ",cost);

    return 0;
}
```

Input and Output:

```
Enter the number of node: 4

Enter the Cost Matrix:

4 0 2 1

5 1 0 2

3 0 3 1

6 8 1 0

The Path is: 1--->3--->4--->2--->1

Minimum cost is 16
```

<u>Experiment No.</u> 05

<u>Name of the Experiment:</u> (i) Create and load different datasets in python. (ii) Write a python program to compute Mean, Median, Mode, Variance and Standard Deviation using datasets.

<u>Objective(s):</u>

1. To create and load different datasets in python.

2. To compute Mean, Median, Mode, Variance and Standard Deviation using datasets.

<u>Theory:</u>

Mean: Mean is also known as average of all the numbers in the data set which is calculated by below equation.

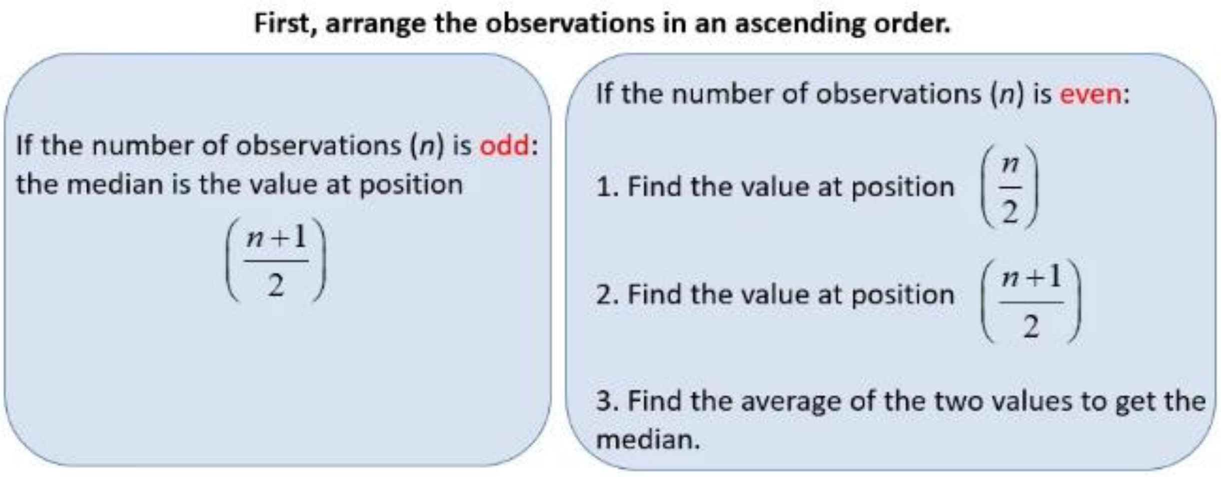
$$\text{Mean} = \frac{\text{Sum of all data values}}{\text{Number of data values}}$$

Symbolically,

$$\bar{x} = \frac{\sum x}{n}$$

where $\bar{x}$ (read as 'x bar') is the mean of the set of x values,
$\sum x$ is the sum of all the x values, and
n is the number of x values.

Median: Median is mid value in this ordered data set.



First, arrange the observations in an ascending order.

If the number of observations (n) is odd: the median is the value at position $\left(\frac{n+1}{2}\right)$

If the number of observations (n) is even:

1. Find the value at position $\left(\frac{n}{2}\right)$

2. Find the value at position $\left(\frac{n+1}{2}\right)$

3. Find the average of the two values to get the median.

Arrange the data in the increasing order and then find the mid value.

Mode: Mode is the number which occur most often in the data set.Here 150 is occurring twice so this is our mode.

Variance: Variance is the numerical values that describe the variability of the observations from its arithmetic mean and denoted by sigma-square(σ2)**.**

Variance measure how far individuals in the group are spread out, in the set of data from the mean.

$$\sigma^2 = \frac{\sum_{i=1}^{n} \left(X_i - \mu\right)^2}{n}$$

Where

Xi: Elements in the data set

mu: the population mean

Standard Deviation: It is a measure of dispersion of observation within dataset relative to their mean. It is square root of the variance and denoted by Sigma (σ).

Standard deviation is expressed in the same unit as the values in the dataset so it measure how much observations of the data set differs from its mean.

$$\sigma = \sqrt{\frac{\sum(x_i - \mu)^2}{N}}$$

σ = population standard deviation

N = the size of the population

x_i = each value from the population

μ = the population mean

Source Code:

```
#5(i):
import pandas as pd

data = {'Name': ['John', 'Mary', 'Peter', 'David'],
        'Age': [25, 30, 27, 28],
        'Country': ['USA', 'Canada', 'Australia', 'USA']}
df = pd.DataFrame(data)
print(df)
#Load dataset in Python
import pandas as pd
df = pd.read_csv('my_dataset.csv')
print(df)
```

```
#5(ii):
import statistics
# Sample dataset
data = [2, 4, 6, 2,8, 15]
# Compute mean
mean = statistics.mean(data)
print("Mean: ", mean)
# Compute median
median = statistics.median(data)
print("Median: ", median)
# Compute mode
mode = statistics.mode(data)
print("Mode: ", mode)
# Compute variance
variance = statistics.variance(data)
print("Variance: ", variance)
# Compute standard deviation
std_deviation = statistics.stdev(data)
print("Standard Deviation: ", std_deviation)
```

Output: **5(i):**

```
    Name  Age    Country
0   John   25        USA
1   Mary   30     Canada
2  Peter   27  Australia
3  David   28        USA
```

5(ii):

```
Mean:  6.167

Median:  5.0

Mode:  2

Variance:  24.167

Standard Deviation:  4.916
```

Experiment No. 06

Name of the Experiment: Write a python program to implement simple linear regression and plot the graph.

Objective(s):

1. To predict the value of an output variable (or response) based on the value of an input (or predictor) variable.
2. To use regression analysis to predict the value of a dependent variable based on an independent variable.
3. To evaluate the assumptions of regression analysis and know what to do if the assumptions are violated.

Theory: Simple linear regression is used to estimate the relationship between two quantitative variables. You can use simple linear regression when you want to know:

- How strong the relationship is between two variables (e.g., the relationship between rainfall and soil erosion).
- The value of the dependent variable at a certain value of the independent variable (e.g., the amount of soil erosion at a certain level of rainfall).

Regression models describe the relationship between variables by fitting a line to the observed data. Linear regression models use a straight line, while logistic and nonlinear regression models use a curved line. Regression allows you to estimate how a dependent variable changes as the independent variable(s) change.

If you have more than one independent variable, use multiple linear regression instead.

Assumptions of simple linear regression:

Simple linear regression is a **parametric test**, meaning that it makes certain assumptions about the data. These assumptions are:

1. **Homogeneity of variance (homoscedasticity)**: the size of the error in our prediction doesn’t change significantly across the values of the independent variable.

2. **Independence of observations**: the observations in the dataset were collected using statistically valid sampling methods, and there are no hidden relationships among observations.

3. **Normality**: The data follows a normal distribution.

Linear regression makes one additional assumption:

4. The relationship between the independent and dependent variable is **linear**: the line of best fit through the data points is a straight line (rather than a curve or some sort of grouping factor).

If your data violate the assumption of independence of observations (e.g., if observations are repeat over time), you may be able to perform a linear mixed-effects model that accounts for the additional structure in the data.

How to perform a simple linear regression

Simple linear regression formula

The formula for a simple linear regression is:

$$y = \beta_0 + \beta_1 X + \epsilon$$

- **y** is the predicted value of the dependent variable (**y**) for any given value of the independent variable (**x**).
- **B_0** is the **intercept**, the predicted value of **y** when the **x** is 0.
- **B_1** is the regression coefficient – how much we expect **y** to change as **x** increases.
- **X** is the independent variable (the variable we expect is influencing **y**).
- **epsilon** is the **error** of the estimate, or how much variation there is in our estimate of the regression coefficient.

Linear regression finds the line of best fit line through your data by searching for the regression coefficient (B_1) that minimizes the total error (e) of the model.

While you can perform a linear regression by hand, this is a tedious process, so most people use statistical programs to help them quickly analyze the data.

Source Code:

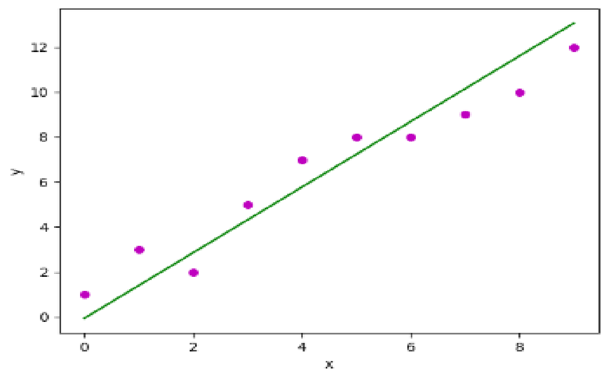
```
import numpy as np
import matplotlib.pyplot as plt
def estimate_coef(x, y):
        # number of observations/points
        n = np.size(x)
        # mean of x and y vector
        m_x = np.mean(x)
        m_y = np.mean(y)
       # calculating cross-deviation and deviation about x
        SS_xy = np.sum(y*x) - n*m_y*m_x
        SS_xx = np.sum(x*x) - n*m_x*m_x
        # calculating regression coefficients
        b_1 = SS_xy / SS_xx
        b_0 = m_y - b_1*m_x

        return (b_0, b_1)

def plot_regression_line(x, y, b):
        # plotting the actual points as scatter plot
```

```
    plt.scatter(x, y, color = "m",
                marker = "o", s = 30)

    # predicted response vector
    y_pred = b[0] + b[1]*x
    # plotting the regression line
    plt.plot(x, y_pred, color = "g")
    # putting labels
    plt.xlabel('x')
    plt.ylabel('y')
    # function to show plot
    plt.show()

def main():
    # observations / data
    x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])
    # estimating coefficients
    b = estimate_coef(x, y)
    print("Estimated coefficients:\nb_0 = {} \
          \nb_1 = {}".format(b[0], b[1]))
   # plotting regression line
    plot_regression_line(x, y, b)

if __name__ == "__main__":
    main()
```

<u>Output:</u>

```
Estimated coefficients:

b_0 = -0.0586206896552

b_1 = 1.45747126437
```

<u>Plot:</u>



Experiment No. 07

Name of the Experiment: Write a python program to implement Find S algorithm.

Objective(s):

1. To find the most specific target concept.
2. To generate a hypothesis that accurately represents the training data.
3. To minimize the number of errors in the hypothesis.

Theory: The find-S algorithm is a basic concept learning algorithm in machine learning. The find-S algorithm finds the most specific hypothesis that fits all the positive examples. We have to note here that the algorithm considers only those positive training example. The find-S algorithm starts with the most specific hypothesis and generalizes this hypothesis each time it fails to classify an observed positive training data. Hence, the Find-S algorithm moves from the most specific hypothesis to the most general hypothesis.

Important Representation:

1. ? indicates that any value is acceptable for the attribute.
2. specify a single required value (e.g., Cold) for the attribute.
3. ϕindicates that no value is acceptable.
4. The most general hypothesis is represented by: {?, ?, ?, ?, ?, ?}
5. The most specific hypothesis is represented by: {ϕ, ϕ, ϕ, ϕ, ϕ, ϕ}

Steps Involved In Find-S:

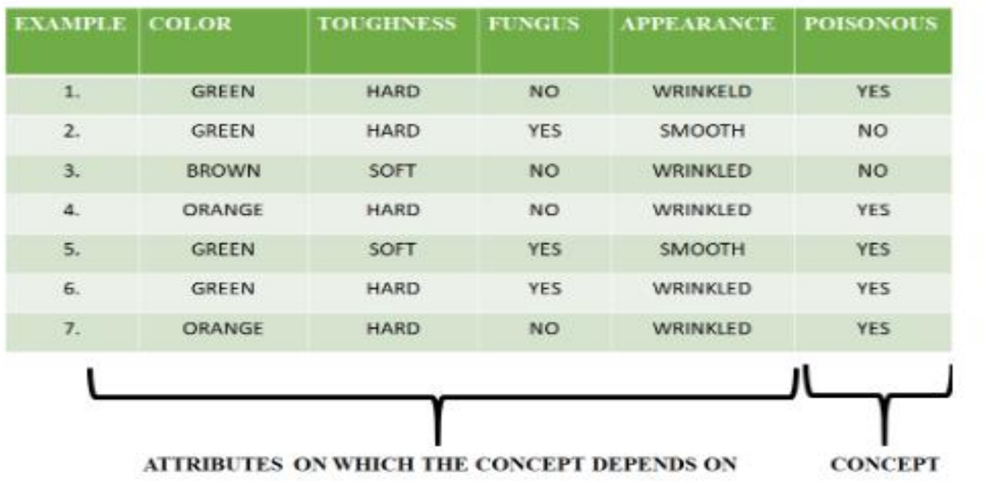
1. Start with the most specific hypothesis.
   h = {ϕ, ϕ, ϕ, ϕ, ϕ, ϕ}
2. Take the next example and if it is negative, then no changes occur to the hypothesis.
3. If the example is positive and we find that our initial hypothesis is too specific then we update our current hypothesis to a general condition.
4. Keep repeating the above steps till all the training examples are complete.
5. After we have completed all the training examples we will have the final hypothesis when can use to classify the new examples.

Example: Consider the following data set having the data about which particular seeds are poisonous-

First, we consider the hypothesis to be a more specific hypothesis. Hence, our hypothesis would be:
h = {ϕ, ϕ, ϕ, ϕ, ϕ, ϕ}

Consider example 1: The data in example 1 is {GREEN, HARD, NO, WRINKLED}. We see that our initial hypothesis is more specific and we have to generalize it for this example. Hence, the hypothesis becomes:
h = {GREEN, HARD, NO, WRINKLED }

Consider example 2:
Here we see that this example has a negative outcome. Hence we neglect this example and our hypothesis remains the same.
h = { GREEN, HARD, NO, WRINKLED }



EXAMPLE	COLOR	TOUGHNESS	FUNGUS	APPEARANCE	POISONOUS
1.	GREEN	HARD	NO	WRINKELD	YES
2.	GREEN	HARD	YES	SMOOTH	NO
3.	BROWN	SOFT	NO	WRINKLED	NO
4.	ORANGE	HARD	NO	WRINKLED	YES
5.	GREEN	SOFT	YES	SMOOTH	YES
6.	GREEN	HARD	YES	WRINKLED	YES
7.	ORANGE	HARD	NO	WRINKLED	YES

ATTRIBUTES ON WHICH THE CONCEPT DEPENDS ON
CONCEPT

Consider example 3:
Here we see that this example has a negative outcome. Hence we neglect this example and our hypothesis remains the same.
h = { GREEN, HARD, NO, WRINKLED }

Consider example 4:
The data present in example 4 is { ORANGE, HARD, NO, WRINKLED }. We compare every single attribute with the initial data and if any mismatch is found we replace that particular attribute with a general case (” ? ”). After doing the process the hypothesis becomes:
h = { ?, HARD, NO, WRINKLED }

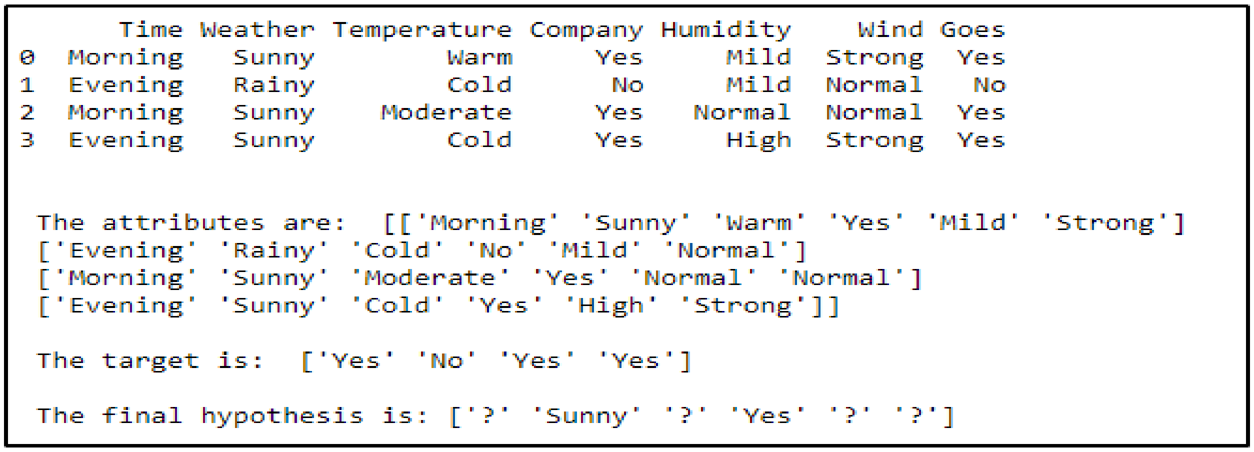
Consider example 5:
The data present in example 5 is { GREEN, SOFT, YES, SMOOTH }. We compare every single attribute with the initial data and if any mismatch is found we replace that particular attribute with a general case (” ? ”). After doing the process the hypothesis becomes :
h = { ?, ?, ?, ? }
Since we have reached a point where all the attributes in our hypothesis have the general condition, example 6 and example 7 would result in the same hypothesizes with all general attributes.
h = { ?, ?, ?, ? }

Hence, for the given data the final hypothesis would be:
Final Hypothesis: h = { ?, ?, ?, ? }

Source Code:

```
import pandas as pd
import numpy as np
 #to read the data in the csv file
data = pd.read_csv("data.csv")
print(data,"n")
 #making an array of all the attributes
d = np.array(data)[:,:-1]
print("n The attributes are: ",d)
 #segragating the target that has positive and negative examples
target = np.array(data)[:,-1]
print("n The target is: ",target)
 #training function to implement find-s algorithm
def train(c,t):
    for i, val in enumerate(t):
        if val == "Yes":
            specific_hypothesis = c[i].copy()
            break

    for i, val in enumerate(c):
        if t[i] == "Yes":
            for x in range(len(specific_hypothesis)):
                if val[x] != specific_hypothesis[x]:
                    specific_hypothesis[x] = '?'
                else:
                    pass
        return specific_hypothesis
 #obtaining the final hypothesis
print("n The final hypothesis is:",train(d,target))
```

Input and Output:



```
      Time Weather Temperature Company Humidity    Wind Goes
0  Morning   Sunny        Warm     Yes     Mild  Strong  Yes
1  Evening   Rainy        Cold      No     Mild  Normal   No
2  Morning   Sunny    Moderate     Yes   Normal  Normal  Yes
3  Evening   Sunny        Cold     Yes     High  Strong  Yes


 The attributes are:  [['Morning' 'Sunny' 'Warm' 'Yes' 'Mild' 'Strong']
 ['Evening' 'Rainy' 'Cold' 'No' 'Mild' 'Normal']
 ['Morning' 'Sunny' 'Moderate' 'Yes' 'Normal' 'Normal']
 ['Evening' 'Sunny' 'Cold' 'Yes' 'High' 'Strong']]

 The target is:  ['Yes' 'No' 'Yes' 'Yes']

 The final hypothesis is: ['?' 'Sunny' '?' 'Yes' '?' '?']
```

Experiment No. 08

Name of the Experiment: Write a python Program to implement Support Vector Machine (SVM) Algorithm.

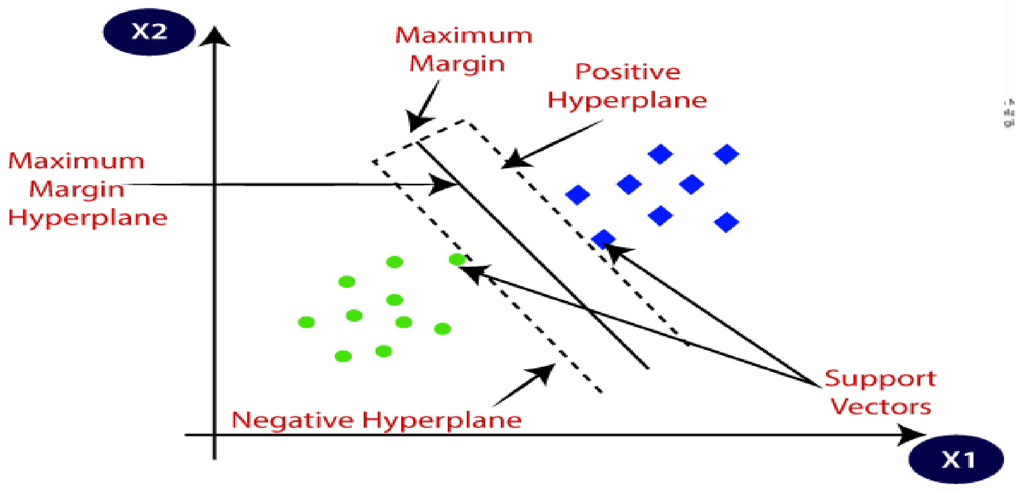
Objective(s):

1. To find the hyperplane in a high-dimensional space that maximizes the margin between the different classes.
2. To achieve good generalization performance.
3. To handle non-linearly separable data: In real-world applications.
4. To handle high-dimensional data

Theory: Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.

SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called as support vectors, and hence algorithm is termed as Support Vector Machine. Consider the below diagram in which there are two different categories that are classified using a decision boundary or hyperplane:



Example: SVM can be understood with the example that we have used in the KNN classifier. Suppose we see a strange cat that also has some features of dogs, so if we want a model that can accurately identify whether it is a cat or dog, so such a model can be created by using the SVM algorithm. We will first train our model with lots of images of cats and dogs so that it can learn about different features of cats and dogs, and then we test it with this strange creature. So as support vector creates a decision boundary between these two data (cat and dog) and choose extreme cases (support vectors), it will see the extreme case of cat and dog. SVM algorithm can be used for Face detection, image classification, text categorization, etc.

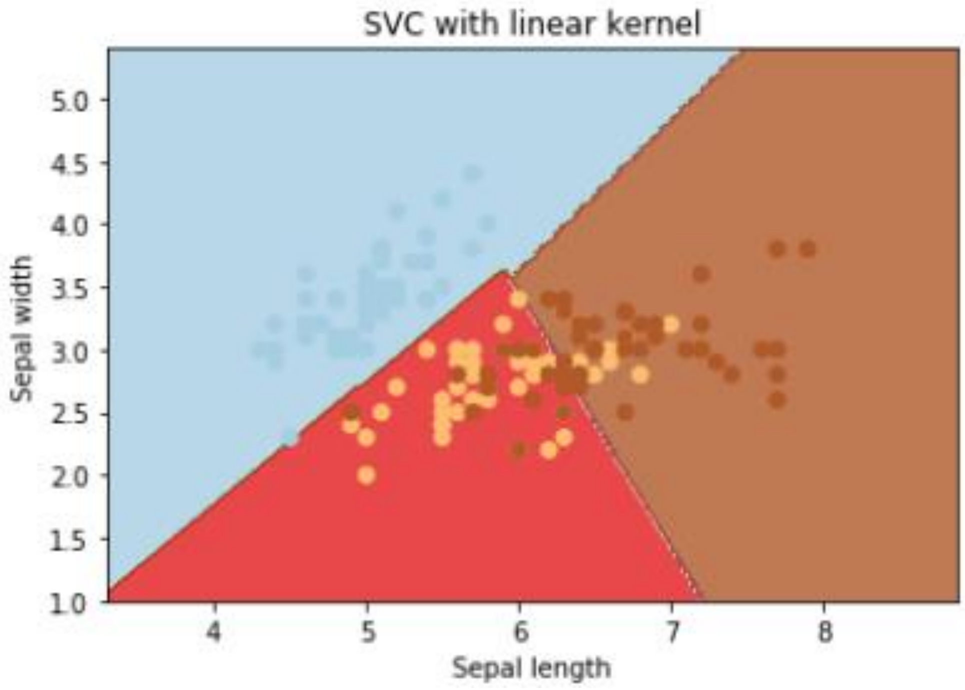
SVM can be of two types:

- **Linear SVM:** Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.
- **Non-linear SVM:** Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data and classifier used is called as Non-linear SVM classifier.

Source Code:

```
# Import the Libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets

# Import some Data from the iris Data Set
iris = datasets.load_iris()

# Take only the first two features of Data.
# To avoid the slicing, Two-Dim Dataset can be used

X = iris.data[:, :2]
y = iris.target

# C is the SVM regularization parameter
C = 1.0

# Create an Instance of SVM and Fit out the data.
# Data is not scaled so as to be able to plot the support vectors
svc = svm.SVC(kernel ='linear', C = 1).fit(X, y)

# create a mesh to plot
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
h = (x_max / x_min)/100
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

# Plot the data for Proper Visual Representation
plt.subplot(1, 1, 1)

# Predict the result by giving Data to the model
Z = svc.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap = plt.cm.Paired, alpha = 0.8)
plt.scatter(X[:, 0], X[:, 1], c = y, cmap = plt.cm.Paired)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.xlim(xx.min(), xx.max())
plt.title('SVC with linear kernel')

# Output the Plot
plt.show()
```

Output:


SVC with linear kernel
Sepal width
Sepal length