Program 1:

```
Graph_nodes = {
    'A': [('B', 4), ('F', 3)],
    'B': [('C', 3), ('D', 2)],
    'C': [('D', 1), ('E', 5)],
    'D': [('C', 1), ('E', 8)],
    'E': [('I', 5), ('J', 5)],
    'F': [('G', 6),('H', 7)] ,
    'G': [('I', 3)],
    'H': [('I', 2)],
    'I': [('E', 5), ('J', 3)],


}


def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None


def h(n):
        H_dist = {
            'A': 1,
            'B': 8,
            'C': 5,
            'D': 7,
            'E': 3,
            'F': 6,
            'G': 5,
            'H': 3,
            'I': 1,
```

```
        'J': 0
    }
    return H_dist[n]

def aStarAlgo(start_node, stop_node):

    open_set = set(start_node)
    closed_set = set()
    g = {}
    parents = {}
    g[start_node] = 0
    parents[start_node] = start_node

    while len(open_set) > 0:
        n = None

        for v in open_set:
            if n == None or g[v] + h(v) < g[n] + h(n):
                n = v

        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight

                else:
                    if g[m] > g[n] + weight:
```

```
                    g[m] = g[n] + weight
                    parents[m] = n
                    if m in closed_set:
                        closed_set.remove(m)
                        open_set.add(m)

            if n == None:
                print('Path does not exist!')
                return None
            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
            open_set.remove(n)
            closed_set.add(n)

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

aStarAlgo('A', 'J')
```

program 2:

```
#AO* aglorithm by Dr. K PARAMESHA, Professor, VVCE, Mysuru, INDIA


class Graph:
    def __init__(self, graph, heuristicNodeList, startNode):  #instantiate graph object with graph topology, heuristic values, start node

        self.graph = graph
        self.H=heuristicNodeList
        self.start=startNode
        self.parent={}
        self.status={}
        self.solutionGraph={}

    def applyAOStar(self):        # starts a recursive AO* algorithm
        self.aoStar(self.start, False)

    def getNeighbors(self, v):     # gets the Neighbors of a given node
        return self.graph.get(v,'')

    def getStatus(self,v):         # return the status of a given node
        return self.status.get(v,0)  #GET IS INBUILT,RETURNS VALUE OF THE KEY. IF KEY NOT PRESENT THEN RETURN "SECOND PARAMETER"

    def setStatus(self,v, val):    # set the status of a given node
        self.status[v]=val

    def getHeuristicNodeValue(self, n):
        return self.H.get(n,0)     # always return the heuristic value of a given node

    def setHeuristicNodeValue(self, n, value):
        self.H[n]=value            # set the revised heuristic value of a given node
```

```
    def printSolution(self):
        print("FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE:",self.start)
        print("------------------------------------------------------------")
        print(self.solutionGraph)
        print("------------------------------------------------------------")

    def computeMinimumCostChildNodes(self, v):  # Computes the Minimum Cost of child nodes of a given node v
        minimumCost=0
        costToChildNodeListDict={}
        costToChildNodeListDict[minimumCost]=[]
        flag=True
        for nodeInfoTupleList in self.getNeighbors(v):  # iterate over all the set of child node/s
            cost=0
            nodeList=[]
            for c, weight in nodeInfoTupleList:
                cost=cost+self.getHeuristicNodeValue(c)+weight
                nodeList.append(c)

            if flag==True:                      # initialize Minimum Cost with the cost of first set of child node/s
                minimumCost=cost
                costToChildNodeListDict[minimumCost]=nodeList      # set the Minimum Cost child node/s
                flag=False
            else:                               # checking the Minimum Cost nodes with the current Minimum Cost
                if minimumCost>cost:
                    minimumCost=cost
                    costToChildNodeListDict[minimumCost]=nodeList  # set the Minimum Cost child node/s
```

```
        return minimumCost, costToChildNodeListDict[minimumCost]   # return Minimum Cost and
Minimum Cost child node/s



    def aoStar(self, v, backTracking):     # AO* algorithm for a start node and backTracking status flag


        print("HEURISTIC VALUES  :", self.H)

        print("SOLUTION GRAPH    :", self.solutionGraph)

        print("PROCESSING NODE   :", v)

        print("-----------------------------------------------------------------------------------------")


        if self.getStatus(v) >= 0:        # if status node v >= 0, compute Minimum Cost nodes of v(FOR
START NODE, STATUS WILL BE RETURNED AS 0)

            minimumCost, childNodeList = self.computeMinimumCostChildNodes(v)

            self.setHeuristicNodeValue(v, minimumCost)

            self.setStatus(v,len(childNodeList)) #THEN STATUS KEEPS UPDATING (HOW MANY TO
VISIT(NO OF CHILDREN))


            solved=True                   # check the Minimum Cost nodes of v are solved

            for childNode in childNodeList:

                self.parent[childNode]=v

                if self.getStatus(childNode)!=-1:

                    solved=solved & False


            if solved==True:              # if the Minimum Cost nodes of v are solved, set the current node
status as solved(-1)

                self.setStatus(v,-1)     # THIS IS WHAT SETS THE TERMINATING CONDITION

                self.solutionGraph[v]=childNodeList # update the solution graph with the solved nodes
which may be a part of solution



            if v!=self.start:           # check the current node is the start node for backtracking the current
node value
```

```
            self.aoStar(self.parent[v], True)   # backtracking the current node value with backtracking status set to true

        if backTracking==False:     # check the current call is not for backtracking
            for childNode in childNodeList:   # for each Minimum Cost child node
                self.setStatus(childNode,0)   # set the status of child node to 0(needs exploration)
                self.aoStar(childNode, False) # Minimum Cost child node is further explored with backtracking status as false


h2 = {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}  # Heuristic values of Nodes
graph2 = {                                    # Graph of Nodes and Edges
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],      # Neighbors of Node 'A', B, C & D with repective weights
    'B': [[('G', 1)], [('H', 1)]],               # Neighbors are included in a list of lists
    'D': [[('E', 1), ('F', 1)]]                  # Each sublist indicate a "OR" node or "AND" nodes
}

G2 = Graph(graph2, h2, 'A')                    # Instantiate Graph object with graph, heuristic values and start Node
G2.applyAOStar()                               # Run the AO* algorithm
G2.printSolution()                             # Print the solution graph as output of the AO* algorithm search
```

Program 3:

```
import csv

with open("trainingexamples.csv") as f:
    csv_file = csv.reader(f)
    data = list(csv_file)

    specific = data[0][:-1]
    general = [['?' for i in range(len(specific))] for j in range(len(specific))]
```

```
step=1 #for printing purpose
for i in data:
    if i[-1] == "Y":
        for j in range(len(specific)):
            if i[j] != specific[j]:
                specific[j] = "?"
                general[j][j] = "?"

    elif i[-1] == "N":
        for j in range(len(specific)):
            if i[j] != specific[j]:
                general[j][j] = specific[j]
            else:
                general[j][j] = "?"

    print("\nStep {} of candidate elimination algo".format(step))
    step+=1
    print(specific)
    print(general)

gh = [] # gh = general Hypothesis
for i in general:
    for j in i:
        if j != '?':
            gh.append(i)
            break
print("\nFinal Specific hypothesis:\n", specific)
print("\nFinal General hypothesis:\n", gh)
```

dataset:

```
sunny,warm,normal,strong,warm,same,Y

sunny,warm,high,strong,warm,same,Y

rainy,cold,high,strong,warm,change,N

sunny,warm,high,strong,cool,change,Y
```

Program 4:

```
import pandas as pd

from pprint import pprint

from sklearn.feature_selection import mutual_info_classif

from collections import Counter


def id3(df, target_attribute, attribute_names, default_class=None):

    cnt=Counter(x for x in df[target_attribute])

    if len(cnt)==1:

        return next(iter(cnt))


    elif df.empty or (not attribute_names):

         return default_class


    else:

        gains = mutual_info_classif(df[attribute_names],df[target_attribute],discrete_features=True)

        index_of_max=gains.tolist().index(max(gains))

        best_attr=attribute_names[index_of_max]

        tree={best_attr:{}}

        remaining_attribute_names=[i for i in attribute_names if i!=best_attr]


        for attr_val, data_subset in df.groupby(best_attr):

            subtree=id3(data_subset, target_attribute, remaining_attribute_names,default_class)

            tree[best_attr][attr_val]=subtree
```

```
    return tree


df=pd.read_csv("traintennis.csv")

attribute_names=df.columns.tolist()
print("List of attribute name")

attribute_names.remove("PlayTennis")

for colname in df.select_dtypes("object"):
    df[colname], _ = df[colname].factorize()

print(df)

tree= id3(df,"PlayTennis", attribute_names)
print("The tree structure")
pprint(tree)
```

Datasets:

```
Outlook,Temperature,Humidity,Wind,PlayTennis
Sunny,Hot,High,Weak,No
Sunny,Hot,High,Strong,No
Overcast,Hot,High,Weak,Yes
Rain,Mild,High,Weak,Yes
Rain,Cool,Normal,Weak,Yes
Rain,Cool,Normal,Strong,No
Overcast,Cool,Normal,Strong,Yes
Sunny,Mild,High,Weak,No
Sunny,Cool,Normal,Weak,Yes
Rain,Mild,Normal,Weak,Yes
```

Sunny,Mild,Normal,Strong,Yes

Overcast,Mild,High,Strong,Yes

Overcast,Hot,Normal,Weak,Yes

Rain,Mild,High,Strong,No

Program 5:

```
import numpy as np
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X/np.amax(X,axis=0) # maximum of X array longitudinally
y = y/100

#Sigmoid Function
def sigmoid (x):
   return 1/(1 + np.exp(-x))

#Derivative of Sigmoid Function
def derivatives_sigmoid(x):
   return x * (1 - x)

#Variable initialization
epoch=5000      #Setting training iterations
lr=0.1          #Setting learning rate
inputlayer_neurons = 2          #number of features in data set
hiddenlayer_neurons = 3         #number of hidden layers neurons
output_neurons = 1              #number of neurons at output layer

#weight and bias initialization
```

```
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))  #2,3
bh=np.random.uniform(size=(1,hiddenlayer_neurons))                  #1,3
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))   #3,1
bout=np.random.uniform(size=(1,output_neurons))                     #1,1

for i in range(epoch):
#Forward Propogation
   hinp=np.dot(X,wh)+ bh
   hlayer_act = sigmoid(hinp)     #HIDDEN LAYER ACTIVATION FUNCTION
   outinp=np.dot(hlayer_act,wout)+ bout
   output = sigmoid(outinp)

   outgrad = derivatives_sigmoid(output)
   hiddengrad = derivatives_sigmoid(hlayer_act)

   EO = y-output               #ERROR AT OUTPUT LAYER
   d_output = EO* outgrad

   EH = d_output.dot(wout.T)      #ERROR AT HIDDEN LAYER (TRANSPOSE => COZ REVERSE(BACK))
   d_hiddenlayer = EH * hiddengrad

   wout += hlayer_act.T.dot(d_output) *lr     #REMEMBER WOUT IS 3*1 MATRIX
   wh += X.T.dot(d_hiddenlayer) *lr

print("Input: \n" + str(X))
print("Actual Output: \n" + str(y))
print("Predicted Output: \n" ,output)
```

Program 6:

```
# import necessary libraries
import pandas as pd
from sklearn import tree
from sklearn.preprocessing import LabelEncoder
from sklearn.naive_bayes import GaussianNB

# Load Data from CSV
data = pd.read_csv('p-tennis.csv')
print("The first 5 Values of data is :\n", data.head())

# obtain train data and train output
X = data.iloc[:, :-1]
print("\nThe First 5 values of the train data is\n", X.head())

y = data.iloc[:, -1]
print("\nThe First 5 values of train output is\n", y.head())

# convert them in numbers
le_outlook = LabelEncoder()
X.Outlook = le_outlook.fit_transform(X.Outlook)

le_Temperature = LabelEncoder()
X.Temperature = le_Temperature.fit_transform(X.Temperature)

le_Humidity = LabelEncoder()
X.Humidity = le_Humidity.fit_transform(X.Humidity)

le_Windy = LabelEncoder()
X.Windy = le_Windy.fit_transform(X.Windy)
```

```
print("\nNow the Train output is\n", X.head())

le_PlayTennis = LabelEncoder()
y = le_PlayTennis.fit_transform(y)
print("\nNow the Train output is\n",y)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.20)

classifier = GaussianNB()
classifier.fit(X_train, y_train)

from sklearn.metrics import accuracy_score
print("Accuracy is:", accuracy_score(classifier.predict(X_test), y_test))
```

Datasets:

```
Outlook,Temperature,Humidity,Windy,PlayTennis
Sunny,Hot,High,False,No
Sunny,Hot,High,True,No
Overcast,Hot,High,False,Yes
Rainy,Mild,High,False,Yes
Rainy,Cool,Normal,False,Yes
Rainy,Cool,Normal,True,No
Overcast,Cool,Normal,True,Yes
Sunny,Mild,High,False,No
Sunny,Cool,Normal,False,Yes
Rainy,Mild,Normal,False,Yes
Sunny,Mild,Normal,True,Yes
Overcast,Mild,High,True,Yes
Overcast,Hot,Normal,False,Yes
```

Rainy,Mild,High,True,No

Program 7:

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
import sklearn.metrics as sm
import pandas as pd
import numpy as np

iris = datasets.load_iris()
X = pd.DataFrame(iris.data)
X.columns = ['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width']
Y = pd.DataFrame(iris.target)
Y.columns = ['Targets']

print(X)
print(Y)
colormap = np.array(['red', 'lime', 'black'])

plt.subplot(1,2,1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[Y.Targets], s=40)
plt.title('Real Clustering')

model1 = KMeans(n_clusters=3)
model1.fit(X)

plt.subplot(1,2,2)
```

```
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[model1.labels_], s=40)
plt.title('K Mean Clustering')
plt.show()

model2 = GaussianMixture(n_components=3)
model2.fit(X)

plt.subplot(1,2,1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[model2.predict(X)], s=40)
plt.title('EM Clustering')
plt.show()

print("Actual Target is:\n", iris.target)
print("K Means:\n",model1.labels_)
print("EM:\n",model2.predict(X))
print("Accuracy of KMeans is ",sm.accuracy_score(Y,model1.labels_))
print("Accuracy of EM is ",sm.accuracy_score(Y, model2.predict(X)))
```

Program 8:

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
iris=datasets.load_iris()
print("Iris Data set loaded...")
x_train, x_test, y_train, y_test = train_test_split(iris.data,iris.target,test_size=0.1)
#random_state=0
for i in range(len(iris.target_names)):
  print("Label", i , "-",str(iris.target_names[i]))
classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(x_train, y_train)
```

```
y_pred=classifier.predict(x_test)
print("Results of Classification using K-nn with K=5 ")
for r in range(0,len(x_test)):
   print(" Sample:", str(x_test[r]), " Actual-label:", str(y_test[r])," Predicted-label:", str(y_pred[r]))

   print("Classification Accuracy :" , classifier.score(x_test,y_test));
```

Program 9:

```
import numpy as np
import matplotlib.pyplot as plt

def local_regression(x0, X, Y, tau):
   x0 = [1, x0]
   X = [[1, i] for i in X]
   X = np.asarray(X)
   xw = (X.T) * np.exp(np.sum((X - x0) ** 2, axis=1) / (-2 * (tau**2)))
   beta = (np.linalg.pinv(xw @ X)) @ (xw @ Y)
   return (beta @ x0)

def draw(tau):
   prediction = [local_regression(x0, X, Y, tau) for x0 in domain]
   plt.plot(X, Y, 'o', color='black')
   plt.plot(domain, prediction, color='red')
   plt.show()

X = np.linspace(-3, 3, num=1000)#evenly spaced numbers over [-3,3] totally num(1000) numbers
domain = X
Y = np.log(np.abs((X ** 2) - 1) + .5)#just creating y values...chosing this to get W shaped curve

draw(10)
```

draw(0.1)

draw(0.01)

draw(0.001)