Program 1:

```
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]+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):

                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 exsist')
            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 exsist')
    return None

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

def heuristic(n):
    h_dist = {
        'A':11,
        'B':6,
        'C':99,
        'D':1,
        'E':7,
        'G':0
    }
    return h_dist[n]

graph_nodes = {
    'A' : [('B',2) , ('E',3)] ,
    'B' : [('C',1) , ('G',9)]  ,
    'C' : None  ,
    'E' : [('D' ,6)],
    'D' : [('G',1)]
}
aStarAlgo('A','G')
```

Program 2

```
def recAOStar(n):
    global final path
    print ('Expanding node:',n)
    and_nodes = []
    or_nodes = []

    if(n in allNodes):
        if 'AND' in allNodes[n]:
            and_nodes  = allNodes[n]['AND']

        if 'OR' in allNodes[n]:
            or_nodes = allNodes[n]['OR']



    if len(and_nodes) == 0 or len(or_nodes) == 0:
        return

    solvable = False
    marked = {}

    while not solvable:
        if len(marked) == len(and_nodes) + len(or_nodes):
            min_cost_least,min_cost_group_least = least_cost_group(and_nodes,or_nodes,{})

            solvable = True
            change_heuristic(n,min_cost_group_least)
            optimal_child_group[n] = min_cost_group_least
            continue
```

min_cost,min_cost_group = least_cost_group(and_nodes,or_nodes,marked)

is_expanded = 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
            minimumCost, childNodeList =
self.computeMinimumCostChildNodes(v)
            self.setHeuristicNodeValue(v, minimumCost)
            self.setStatus(v,len(childNodeList))

            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)
                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



h1 = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I':
7, 'J': 1, 'T': 3}
graph1 = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('G', 1)], [('H', 1)]],
    'C': [[('J', 1)]],
    'D': [[('E', 1), ('F', 1)]],
    'G': [[('I', 1)]]
}
G1= Graph(graph1, h1, 'A')
G1.applyAOStar()
G1.printSolution()

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

In [21]:

```
import pandas as pd
df=pd.read_csv('sports.csv',header=0,sep=',')
df
```

Out[21]:

	Sky	AirTemp	Humidity	Wind	Water	Forecast	EnjoySport
0	Sunny	Warm	Normal	Strong	Warm	Same	Yes
1	Sunny	Warm	High	Strong	Warm	Same	Yes
2	Rainy	Cold	High	Strong	Warm	Change	No
3	Sunny	Warm	High	Strong	Cool	Change	Yes

In [22]:

```
import numpy as np
concepts=np.array(df.iloc[:,0:-1])
target=np.array(df.iloc[:,-1])
print(concepts)
print(target)
```

```
[['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']
 ['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same']
 ['Rainy' 'Cold' 'High' 'Strong' 'Warm' 'Change']
 ['Sunny' 'Warm' 'High' 'Strong' 'Cool' 'Change']]
['Yes' 'Yes' 'No' 'Yes']
```

In [23]:

```
specific_h=concepts[0].copy()
general_h=[['?' for i in range(len(specific_h))] for i in range(len(specific_h))]
print(specific_h)
print(general_h)
```

```
['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?',
'?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
```

In [24]:

```
for i,h in enumerate(concepts):
    if target[i]=="Yes":
        for x in range(len(specific_h)):
            if h[x]!=specific_h[x]:
                specific_h[x]='?'
                general_h[x][x]='?'
    else:
        for x in range(len(specific_h)):
            if h[x]!=specific_h[x]:
                general_h[x][x]=specific_h[x]
            else:
                general_h[x][x]='?'
    print('General',general_h)
    print('Specific',specific_h)
    print("************************************************************")
```

```
General [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
Specific ['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']
************************************************************
General [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
Specific ['Sunny' 'Warm' '?' 'Strong' 'Warm' 'Same']
************************************************************
General [['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'Same']]
Specific ['Sunny' 'Warm' '?' 'Strong' 'Warm' 'Same']
************************************************************
General [['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
Specific ['Sunny' 'Warm' '?' 'Strong' '?' '?']
************************************************************
```

```
indices=[i for i,val in enumerate(general_h) if val==['?', '?', '?', '?', '?', '?']]
for i in indices:
    general_h.remove(['?', '?', '?', '?', '?', '?'])
```

In [25]:

```
indices
```

Out[25]:

```
[2, 3, 4, 5]
```

In [26]:

```
general_h
```

Out[26]:

```
[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?']]
```

In [12]:

```
concepts[1:]
```

Out[12]:

```
array([['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same'],
       ['Rainy', 'Cold', 'High', 'Strong', 'Warm', 'Change'],
       ['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change']],
      dtype=object)
```

In [13]:

```
concepts
```

Out[13]:

```
array([['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same'],
       ['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same'],
       ['Rainy', 'Cold', 'High', 'Strong', 'Warm', 'Change'],
       ['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change']],
```

Program 4

```
import pandas as pd
cols=["preg","gluco","bp","sk","insulin","BMI","pedigree","age","outcom
e"]
df=pd.read_csv('diabetes.csv',header=0,names=cols)
features=cols[:-1]
X=df[features]
y=df.outcome
# print(X)
y=pd.get_dummies(y).values
# print(y)

from sklearn.model_selection import train_test_split
X_train,X_test,Y_train,Y_test=train_test_split(X,y,test_size=0.3,random
_state=1)

from sklearn.tree import DecisionTreeClassifier
model=DecisionTreeClassifier(criterion="entropy",max_depth=3) #Call the
constructor
model=model.fit(X_train,Y_train) #fit the tarining data to model
y_pred=model.predict(X_test)

from sklearn import metrics
acc=metrics.accuracy_score(Y_test,y_pred)
print(acc)

from sklearn import tree
text = tree.export_text(model)
print(text)

import matplotlib.pyplot as plt
fig = plt.figure(figsize=(25,20))
tree.plot_tree(model,
               feature_names = features,
               class_names = ['0','1'],
               filled = True)
```

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))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))


#draws a random range of numbers uniformly of dim x*y
for i in range(epoch):

#Forward Propogation
    hinp1=np.dot(X,wh)
    hinp=hinp1 + bh
    hlayer_act = sigmoid(hinp)
    outinp1=np.dot(hlayer_act,wout)
    outinp= outinp1+ bout
    output = sigmoid(outinp)

#Backpropagation
    EO = y-output
    outgrad = derivatives_sigmoid(output)
```

```
    d_output = EO* outgrad
    EH = d_output.dot(wout.T)

#how much hidden layer wts contributed to error
    hiddengrad = derivatives_sigmoid(hlayer_act)
    d_hiddenlayer = EH * hiddengrad

# dotproduct of nextlayererror and currentlayerop
wout += hlayer_act.T.dot(d_output) *lr
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 pandas as pd
#Loading the PlayTennis data
PlayTennis = pd.read_csv("tennis.csv")
print("Given dataset:\n", PlayTennis,"\n")
#for GaussianNB, We can convert all the non numerical values into numerical
#values using LabelEncoder
from sklearn.preprocessing import LabelEncoder
Le = LabelEncoder()
PlayTennis['Outlook'] = Le.fit_transform(PlayTennis['Outlook'])
PlayTennis['Temperature'] = Le.fit_transform(PlayTennis['Temperature'])
PlayTennis['Humidity'] = Le.fit_transform(PlayTennis['Humidity'])
PlayTennis['Wind'] = Le.fit_transform(PlayTennis['Wind'])
PlayTennis['PlayTennis'] = Le.fit_transform(PlayTennis['PlayTennis'])
print("the encoded dataset is:\n",PlayTennis)
X = PlayTennis.drop(['PlayTennis'],axis=1) #X - holds the attribute values.
y = PlayTennis['PlayTennis'] #y - holds target values.
#print("X: \n",X,"\n")
#print("y: \n",y, "\n")
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score
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)
accuracy = accuracy_score(classifier.predict(X_test),y_test)
print("\n Accuracy is:",accuracy)
```

Program 7

```
#       Kmeans
from sklearn import datasets
from sklearn import metrics
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split

iris = datasets.load_iris()
print(iris)
X_train,X_test,y_train,y_test = train_test_split(iris.data,iris.target)
model =KMeans(n_clusters=3)
model.fit(X_train,y_train)
model.score
print('K-Mean: ',metrics.accuracy_score(y_test,model.predict(X_test)))

#-------Expectation and Maximization----------
from sklearn.mixture import GaussianMixture
model2 = GaussianMixture(n_components=3)
model2.fit(X_train,y_train)
model2.score
print('EM Algorithm:',metrics.accuracy_score(y_test,model2.predict(X_test)))
```

Program 8

```
# step 1:- read the data
import pandas as pd
data = pd.read_csv("Iris.csv")
# print some unwanted details of data
print("Dataset Length: ", len(data))
print("Dataset Shape: ", data.shape)
print("Dataset : \n", data.head())
# step 2:- store input in x, and output in y
x = data.values[:, 1:5]
y = data.values[:, 5]
# step 3:- spilt data for training and testing
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.2,
random_state=100)
# step 4:- apply algorithm
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()
knn.fit(x_train, y_train)
y_pred = knn.predict(x_test)
# step 5:- print actual and expected output
print("Actual Output: ")
print(y_test)
print("Predicted Output: ")
print(y_pred)
# step 6:- print confusion matrix and accuracy score
from sklearn.metrics import confusion_matrix, accuracy_score
print("Confusion Matrix: ")
print(confusion_matrix(y_test, y_pred))
print("Accuracy Score: ", accuracy_score(y_test, y_pred) * 100)
```

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))
    beta = np.linalg.pinv(xw @ X) @ xw @ Y @ x0
    return beta

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)
domain = X
Y = np.log(np.abs(X ** 2 - 1) + .5)

draw(10)
draw(0.1)
draw(0.01)
draw(0.001)
```