#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 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
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
def heuristic(n):
        H_dist = {
            'A': 10,
            'B': 6,
            'C': 9,
            'D': 0,
            'E': 7,
            'F':6,
            'G': 7,

        }

        return H_dist[n]

Graph_nodes = {
    'A': [('B', 1), ('E', 2),('F', 5)],
    'B': [('C', 7)],
    'D': None,
    'E': [('C', 6)],
    'C': [('D', 3)],
    'F': [('C',6)],
    'G': [('D',3)],

}
aStarAlgo('A', 'D')
print()
aStarAlgo('A', 'G')
```

#program 2

```
class Graph:
    def __init__(self, graph, heuristicNodeList, startNode):
        self.graph = graph
        self.H=heuristicNodeList
        self.start=startNode
        self.parent={}
        self.status={}
        self.solutionGraph={}

    def applyAOStar(self):
        self.aoStar(self.start, False)

    def getNeighbors(self, v):
        return self.graph.get(v,'')

    def getStatus(self,v):
        return self.status.get(v,0)

    def setStatus(self,v, val):
        self.status[v]=val

    def getHeuristicNodeValue(self, n):
        return self.H.get(n,0)
```

```
def setHeuristicNodeValue(self, n, value):
    self.H[n]=value

def printSolution(self):
    print("Heuristic values finally are :",self.H)
    print("FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE:",self.start)
    print("-----------------------------------------------------------")
    print(self.solutionGraph)
    print("-----------------------------------------------------------")

def computeMinimumCostChildNodes(self, v):
    minimumCost=0
    costToChildNodeListDict={}
    costToChildNodeListDict[minimumCost]=[]
    flag=True
    for nodeInfoTupleList in self.getNeighbors(v):
        cost=0
        nodeList=[]
        for c, weight in nodeInfoTupleList:
            cost=cost+self.getHeuristicNodeValue(c)+weight
            nodeList.append(c)
        if flag==True:
            minimumCost=cost
            costToChildNodeListDict[minimumCost]=nodeList
            flag=False
        else:
            if minimumCost>cost:
                minimumCost=cost
                costToChildNodeListDict[minimumCost]=nodeList

    return minimumCost, costToChildNodeListDict[minimumCost]


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 if the MinimumCost nodes of v are solved
        for childNode in childNodeList:
            self.parent[childNode]=v
            if self.getStatus(childNode)!=-1:
                solved=solved & False
        if solved==True:
            self.setStatus(v,-1)
            self.solutionGraph[v]=childNodeList
        if v!=self.start:
            self.aoStar(self.parent[v], True)
        if backTracking==False:
            for childNode in childNodeList:
                self.setStatus(childNode,0)
                self.aoStar(childNode, False)

EXAMPLE 1
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()

EXAMPLE 2
h2 = {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5,'H': 7}
graph2 = {
 'A': [[('B', 1), ('C', 1)], [('D', 1)]],
 'B': [[('G', 1)], [('H', 1)]],
 'D': [[('E', 1), ('F', 1)]]
}

G2 = Graph(graph2, h2, 'A')
G2.applyAOStar()
G2.printSolution()
```

#program 3

```
import csv
with open("trainingexamples.csv","r") as csvFile:
    examples =[tuple(line) for line in csv.reader(csvFile)]

def more_general(hyp,exg):
    more_general_parts=[]
    for x,y in zip(hyp,exg):
        mg= x=="?" or (x!="0" and (x==y or y=="0"))
        more_general_parts.append(mg)
    return all(more_general_parts)

def consistent(example,hypothesis):
    return more_general(hypothesis,example)

def get_domains(examples):
    d=[set() for i in examples[0]]
    for x in examples:
        for i,xi in enumerate(x):
            d[i].add(xi)
    return [list(sorted(x)) for x in d]

def g_0(n):
    return ("?",)*n
def s_0(n):
    return ("0",)*n

def min_generalizations(h,x):
    h_new=list(h)
    for i in range(len(h)):
        if not consistent(x[i:i+1],h[i:i+1]):
            if h[i]=="0":
                h_new[i]=x[i]
            else:
                h_new[i]="?"
    return [tuple(h_new)]

def min_specializations(h,domains,x):
    results=[]
    for i in range(len(h)):
        if h[i]=="?":
            for val in domains[i]:
                if x[i]!=val:
                    h_new = h[:i] + (val,) + h[i+1:]
                    results.append(h_new)
        else:
            h_new=h[:i]+("0",)+h[i+1:]
            results.append(h_new)
    return results
def candidate_elimination(examples):
    domains=get_domains(examples)[:-1]
    G=set([g_0(len(domains))])
    S=set([s_0(len(domains))])
    i=0
    print("\nG[{0}]".format(i),G)
    print("\nS[{0}]".format(i),S)
    for xcx in examples:
        #print("\n",xcx)
        i+=1
        x,cx=xcx[:-1],xcx[-1]
        if cx=="Yes":
            G={g for g in G if consistent(x,g)}
            S=generalize_S(x,G,S)
        else:
            S={s for s in S if not consistent(x,s)}
            G=specialize_G(x,domains,G,S)
        print("\nG[{0}]".format(i),G)
        print("\nS[{0}]".format(i),S)
    return

def generalize_S(x,G,S):
    S_prev=list(S)
    for s in S_prev:
        if s not in S:
            continue
        if not consistent(x,s):
            S.remove(s)
            s_plus=min_generalizations(s,x)
            S.update([s for s in s_plus if any([more_general(g,s) for g in G])])
            S.difference_update([h for h in S if any([more_general(h,h1) for h1 in S if h!=h1])])
    return S

def specialize_G(x,domains,G,S):
    G_prev=list(G)

    for g in G_prev:
        if g not in G:
            continue
        if consistent(x,g):
            G.remove(g)
            g_minus=min_specializations(g,domains,x)
            G.update([g for g in g_minus if any([more_general(g,s) for s in S])])
            G.difference_update([h for h in G if any([more_general(h1,h) for h1 in G if h!=h1])])
    return G

candidate_elimination(examples)
```

#program 4

```
import math
import csv

def load_csv(filename):
    lines=csv.reader(open(filename,"r"))
    dataset = list(lines)
    headers = dataset.pop(0)
    return dataset,headers

class Node:
    def __init__ (self,attribute):
        self.attribute=attribute
        self.children=[]
        self.answer=""

def subtables(data,col,delete):
    dic={}
    coldata=[row[col] for row in data]
    attr=list(set(coldata))
    counts=[0]*len(attr)
    r=len(data)
    c=len(data[0])
    for x in range(len(attr)):
        for y in range(r):
            if data[y][col]==attr[x]:
                counts[x]+=1
    for x in range(len(attr)):
        dic[attr[x]]=[[0 for i in range(c)] for j in range(counts[x])]
        pos=0
        for y in range(r):
            if data[y][col]==attr[x]:
                if delete:
                    del data[y][col]
                dic[attr[x]][pos]=data[y]
                pos+=1
    return attr,dic

def entropy(S):
    attr=list(set(S))
    if len(attr)==1:
        return 0
    counts=[0,0]
    for i in range(2):
        counts[i]=sum([1 for x in S if attr[i]==x])/(len(S)*1.0)
    sums=0
    for cnt in counts:
        sums+=-1*cnt*math.log(cnt,2)
    return sums
def compute_gain(data,col):
    attr,dic = subtables(data,col,delete=False)
    total_size=len(data)
    entropies=[0]*len(attr)
    ratio=[0]*len(attr)
    total_entropy=entropy([row[-1] for row in data])
    for x in range(len(attr)):
        ratio[x]=len(dic[attr[x]])/(total_size*1.0)
        entropies[x]=entropy([row[-1] for row in dic[attr[x]]])

        total_entropy-=ratio[x]*entropies[x]
    return total_entropy

def build_tree(data,features):
    lastcol=[row[-1] for row in data]
    if(len(set(lastcol)))==1:
        node=Node("")
        node.answer=lastcol[0]
        return node
    n=len(data[0])-1
    gains=[0]*n
    for col in range(n):
        gains[col]=compute_gain(data,col)
    split=gains.index(max(gains))
    node=Node(features[split])
    fea = features[:split]+features[split+1:]
    attr,dic=subtables(data,split,delete=True)
    for x in range(len(attr)):
        child=build_tree(dic[attr[x]],fea)
        node.children.append((attr[x],child))
    return node
```

```
def print_tree(node,level):
    if node.answer!="":
        print(" "*level,node.answer)
        return
    print(" "*level,node.attribute)
    for value,n in node.children:
        print(" "*(level+1),value)
        print_tree(n,level+2)

def classify(node,x_test,features):
    if node.answer!="":
        print(node.answer)
        return
    pos=features.index(node.attribute)
    for value, n in node.children:
        if x_test[pos]==value:
            classify(n,x_test,features)


dataset,features=load_csv("tennis1.csv")
node1=build_tree(dataset,features)
print("The decision tree for the dataset using ID3 algorithm is")
print_tree(node1,0)
testdata,features=load_csv("tennis.csv")
for xtest in testdata:
    print("The test instance:",xtest)
    print("The label for test instance:",end=" ")
    classify(node1,xtest,features)
```

#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)
y = y/100
def sigmoid(x):
    return 1/(1+np.exp(-x))
def derivatives_sigmoid(x):
    return x*(1-x)
epoch = 10000
lr = 0.1
inputLayerNeurons = 2
hiddenLayerNeurons = 3
outputNeurons = 1
wh = np.random.uniform(size=(inputLayerNeurons, hiddenLayerNeurons))
bh = np.random.uniform(size=(1, hiddenLayerNeurons))
wout = np.random.uniform(size=(hiddenLayerNeurons, outputNeurons))
bout = np.random.uniform(size=(1, outputNeurons))
for i in range(epoch):
    hinp1 = np.dot(x, wh)
    hinp = hinp1 + bh
    hLayerAct = sigmoid(hinp)
    outinp1 = np.dot(hLayerAct, wout)
    outinp = outinp1 + bout
    output = sigmoid(outinp)

    eo = y - output
    #print("epoch = ", i, "eo = ", eo)
    outgrad = derivatives_sigmoid(output)
    d_output = eo*outgrad
    eh = d_output.dot(wout.T)

    hiddengrad = derivatives_sigmoid(hLayerAct)
    d_hiddenLayer = eh*hiddengrad
    wout += hLayerAct.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 numpy as np
import pandas as pd

person =pd.DataFrame()
person['Gender'] = ['female','female','female','female','male','male','male','male']
person['Height'] = [5,5.92,4.58,5.32,6,5.5,6.42,5.75]
person['Weight'] = [150,120,155,145,170,180,190,170]
person['Foot_Size'] = [8,7,9,10,6,5,11,10]
print("\nPerson")
print("")
print(person)

data =pd.DataFrame()
data['Gender'] =
['female','female','female','female','female','female','female','female','female','female','male','male','male','male','male','male','male','
male','male','male','female','female','female','male','male','male']
data['Height'] = [4,5.1,4.2,5.3,4.4,5.5,4.6,5.7,4.8,5.9,6,5.1,6.2,5.3,6.4,5.5,6.6,5.7,6.8,5.9,4,4.1,4.2,5.9,6,5.8]
data['Weight'] =
[130,121,132,123,134,125,136,127,138,129,160,171,162,173,164,175,166,177,168,179,110,100,130,170,185,190]
data['Foot_Size'] = [7,8,9,7,8,9,7,8,9,8,10,9,11,10,9,11,10,9,11,11,7,6,8,10,11,9]
print('\nDataset: ')
print("")
print(data)

n_male= data['Gender'][data['Gender'] =='male'].count()
n_male
n_female= data['Gender'][data['Gender'] =='female'].count()
n_female
```

```
total_ppl= data['Gender'].count()
total_ppl
p_male=n_male/total_ppl
p_male
p_female=n_female/total_ppl
p_female
data_means=data.groupby('Gender').mean()
data_means

print('\nDataset Mean')
#print("")
print(data_means)
data_variance=data.groupby('Gender').var()

print('\nData Variance')
print("")
print(data_variance)

male_height_mean=data_means['Height'][data_means.index=='male'].values[0]
male_weight_mean=data_means['Weight'][data_means.index=='male'].values[0]
male_footsize_mean=data_means['Foot_Size'][data_means.index=='male'].values[0]

print("\nmale_height_mean: ", male_height_mean)
print("male_weight_mean: ", male_weight_mean)
print("male_footsize_mean: ", male_footsize_mean)

male_height_variance=data_variance['Height'][data_variance.index=='male'].values[0]
male_weight_variance=data_variance['Weight'][data_variance.index=='male'].values[0]
male_footsize_variance=data_variance['Foot_Size'][data_variance.index=='male'].values[0]

print("\nmale_height_variance: ",male_height_variance)
print("male_weight_variance: ",male_weight_variance)
print("male_footsize_variance: ",male_footsize_variance)

female_height_mean=data_means['Height'][data_means.index=='female'].values[0]
female_weight_mean=data_means['Weight'][data_means.index=='female'].values[0]
female_footsize_mean=data_means['Foot_Size'][data_means.index=='female'].values[0]

print("\nfemale_height_mean: ", female_height_mean)
print("female_weight_mean: ", female_weight_mean)
print("female_footsize_mean: ", female_footsize_mean)

female_height_variance=data_variance['Height'][data_variance.index=='female'].values[0]
female_weight_variance=data_variance['Weight'][data_variance.index=='female'].values[0]
female_footsize_variance=data_variance['Foot_Size'][data_variance.index=='female'].values[0]

print("\nfemale_height_variance: ",female_height_variance)
print("female_weight_variance: ",female_weight_variance)
print("female_footsize_variance: ",female_footsize_variance)

def p_x_given_y(x,mean_y,variance_y):
   p =1/(np.sqrt(2*np.pi*variance_y))*np.exp((-(x-mean_y) **2)/(2*variance_y))
   return p
n = len(person['Gender'])
```

```
correct = 0
for i in range(n):
    print('\ndata: ',i)
    print('Probability male: ')

    prob_male =
p_male*p_x_given_y(person['Height'][i],male_height_mean,male_height_variance)*
p_x_given_y(person['Weight'][i],male_weight_mean,male_weight_variance)*
p_x_given_y(person['Foot_Size'][i],male_footsize_mean,male_footsize_variance)
    print(prob_male)
    print('Probability female: ')

    prob_female = p_female*p_x_given_y(person['Height'][i],female_height_mean,female_height_variance)*
p_x_given_y(person['Weight'][i],female_weight_mean,female_weight_variance)*
p_x_given_y(person['Foot_Size'][i],female_footsize_mean,female_footsize_variance)

    print(prob_female)

    if(prob_male > prob_female):
        if(person['Gender'][i] == 'male'):
            correct += 1
        print("Probably: Male")
    else:
        if(person['Gender'][i] == 'female'):
            correct += 1
        print("Probably: Female")

accuracy = correct/total_length*100
print("\nAccuracy:",accuracy)
```

#program 7

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
import sklearn.metrics as sm
import pandas as pd
import numpy as np
l1 = [0,1,2]
```

```
def rename(s):
    l2 = []
    for i in s:
        if i not in l2:
            l2.append(i)
    for i in range(len(s)):
        pos = l2.index(s[i])
        s[i] = l1[pos]

    return s

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']

plt.figure(figsize=(14,7))
model = KMeans(n_clusters=3)
model.fit(X)
plt.figure(figsize=(14,7))
colormap = np.array(['red', 'green', 'black'])

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

plt.subplot(1,2,2)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[model.labels_], s=40)
plt.title('K Mean Classification')
plt.show()

km = rename(model.labels_)
print("Accuracy of KMeans is ",sm.accuracy_score(y, km))
print("Confusion Matrix for KMeans is \n",sm.confusion_matrix(y, km))

from sklearn import preprocessing
scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns = X.columns)

from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=3)
gmm.fit(xs)

y_cluster_gmm = gmm.predict(xs)
plt.subplot(1, 2, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y_cluster_gmm], s=40)
plt.title('GMM Classification')
plt.show()
em = rename(y_cluster_gmm)
print("Accuracy of EM is ",sm.accuracy_score(y, em))
print("Confusion Matrix for EM is \n", sm.confusion_matrix(y, em))
```

#program 8

```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
import numpy as np
from sklearn.model_selection import train_test_split

iris_dataset = load_iris()
print("iris features \ target names \n", iris_dataset.target_names)

for i in range(len(iris_dataset.target_names)):
    print('[{0}]:[{1}]'.format(i, iris_dataset.target_names[i]))

x_train, x_test, y_train, y_test = train_test_split(iris_dataset["data"], iris_dataset["target"], random_state = 0)

kn = KNeighborsClassifier(n_neighbors = 5)
kn.fit(x_train, y_train)
x_new = np.array([[5, 2.9, 1, 0.2]])
print("\nX_new \n", x_new)
prediction = kn.predict(x_new)

print("\nPredicted Feature : {}\n".format(prediction))
print("\nPredicted Feature Name: {}\n".format(iris_dataset['target_names'][prediction]))

i = 1
x = x_test[i]
x_new = np.array([x])
print("\nX_new \n", x_new)

for i in range(len(x_test)):
    x = x_test[i]
    x_new = np.array([x])
    prediction = kn.predict(x_new)
    print("actual : {0} {1}, pred : {2}{3}".format(y_test[i], iris_dataset["target_names"][y_test[i]], prediction,
iris_dataset["target_names"][prediction]))
print("test score[accuracy]: {:.2F}\n".format(kn.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))
    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 = 'lime')
    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) + 0.5)
draw(10)
draw(.1)
draw(.01)
draw(.001)
```