PROGRAM 1 : A STAR ALGORITHM

```
def aStarAlgo(start_node, stop_node):


    open_set = set(start_node) # {A}, len{open_set}=1
    closed_set = set()
    g = {} # store the distance from starting node
    parents = {}
    g[start_node] = 0
    parents[start_node] = start_node # parents['A']='A"

    while len(open_set) > 0 :
        n = None

        for v in open_set: # v='B'/'F'
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v # n='A'

        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)      # m=B weight=6 {'F','B','A'}
len{open_set}=2
                    parents[m] = n       # parents={'A':A,'B':A}
len{parent}=2
                    g[m] = g[n] + weight # g={'A':0,'B':6, 'F':3} len{g}=2


            #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)# {'F','B'} len=2
        closed_set.add(n) #{A} len=1

    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': 10,
        'B': 8,
        'C': 5,
        'D': 7,
        'E': 3,
        'F': 6,
        'G': 5,
        'H': 3,
        'I': 1,
        'J': 0
    }

    return H_dist[n]

#Describe your graph here
Graph_nodes = {
```

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

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

PROGRAM 2 : AO STAR ALGORITHM

```
def recAOStar(n):
    global finalPath
    print("Expanding Node : ", n)
    and_nodes = []
    or_nodes = []

    #Segregation of AND and 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 leaf node then return
    if len(and_nodes) == 0 and len(or_nodes) == 0:
        return

    solvable = False
    marked = {}

    while not solvable:
        # If all the child nodes are visited and expanded, take the
least cost of all the child nodes
        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_least)
            optimal_child_group[n] = min_cost_group_least
            continue

        # Least cost of the unmarked child nodes
        min_cost, min_cost_group = least_cost_group(and_nodes,
or_nodes, marked)

        is_expanded = False

        # If the child nodes have sub trees then recursively visit them
to recalculate the heuristic of the child node
        if len(min_cost_group) > 1:
            if (min_cost_group[0] in allNodes):
                is_expanded = True
                recAOStar(min_cost_group[0])
            if (min_cost_group[1] in allNodes):
                is_expanded = True
                recAOStar(min_cost_group[1])
        else:
            if (min_cost_group in allNodes):
                is_expanded = True
```

```
                recAOStar(min_cost_group)


        # If the child node had any subtree and expanded, verify if the
new heuristic value is still the least among all nodes
        if is_expanded:
            min_cost_verify, min_cost_group_verify =
least_cost_group(and_nodes, or_nodes, {})

            if min_cost_group == min_cost_group_verify:
                solvable = True
                change_heuristic(n, min_cost_verify)
                optimal_child_group[n] = min_cost_group

        # If the child node does not have any subtrees then no change
in heuristic, so update the min cost of the current node
        else:
            solvable = True
            change_heuristic(n, min_cost)
            optimal_child_group[n] = min_cost_group

        #Mark the child node which was expanded
        marked[min_cost_group] = 1
    return heuristic(n)


# Function to calculate the min cost among all the child nodes
def least_cost_group(and_nodes, or_nodes, marked):
    node_wise_cost = {}

    for node_pair in and_nodes:
        if not node_pair[0] + node_pair[1] in marked:
            cost = 0
            cost = cost + heuristic(node_pair[0]) +
heuristic(node_pair[1]) + 2
            node_wise_cost[node_pair[0] + node_pair[1]] = cost

    for node in or_nodes:
        if not node in marked:
            cost = 0
            cost = cost + heuristic(node) + 1
            node_wise_cost[node] = cost

    min_cost = 999999
    min_cost_group = None

    # Calculates the min heuristic
    for costKey in node_wise_cost:
        if node_wise_cost[costKey] < min_cost:
            min_cost = node_wise_cost[costKey]
            min_cost_group = costKey
```

```python
    return [min_cost, min_cost_group]


# Returns heuristic of a node
def heuristic(n):
    return H_dist[n]

# Updates the heuristic of a node
def change_heuristic(n, cost):
    H_dist[n] = cost
    return

# Function to print the optimal cost nodes
def print_path(node):
    print(optimal_child_group[node], end="")
    node = optimal_child_group[node]

    if len(node) > 1:
        if node[0] in optimal_child_group:
            print("->", end="")
            print_path(node[0])
        if node[1] in optimal_child_group:
            print("->", end="")
            print_path(node[1])
    else:
        if node in optimal_child_group:
            print("->", end="")
            print_path(node)


#Describe the heuristic here
H_dist = {
    'A': -1,
    'B': 4,
    'C': 2,
    'D': 3,
    'E': 6,
    'F': 8,
    'G': 2,
    'H': 0,
    'I': 0,
    'J': 0
}


#Describe your graph here
allNodes = {
    'A': {'AND': [('C', 'D')], 'OR': ['B']},
    'B': {'OR': ['E', 'F']},
    'C': {'OR': ['G'], 'AND': [('H', 'I')]},
```

```
    'D': {'OR': ['J']}
}

optimal_child_group = {}
optimal_cost = recAOStar('A')

print('Nodes which gives optimal cost are')
print_path('A')
print('\nOptimal Cost is  :: ', optimal_cost)
```

PROGRAM 3 : CANDIDATE ELIMINATION ALGORITHM

```
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))]

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

        elif i[-1] == "No":
            for j in range(len(specific)):
                if i[j] != specific[j]:
                    general[j][j] = specific[j]
                else:
                    general[j][j] = "?"

        print("\nStep " + str(data.index(i)+1) + " of Candidate Elimination
Algorithm")
        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)
```

PROGRAM 4 : ID3 ALGORITHM

```
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:
        gainz =
mutual_info_classif(df[attribute_names],df[target_attribute],discrete_featu
res=True)
        index_of_max=gainz.tolist().index(max(gainz))
        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("playtennis.csv")

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

attribute_names.remove("Class")

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

print(df)

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

PROGRAM 5 : BACK PROPOGATION ALGORITHM(ANN)

```
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=7000 #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):
    print('i = ',i)
    #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)
    hiddengrad = derivatives_sigmoid(hlayer_act)
    #how much hidden layer wts contributed to error
    d_hiddenlayer = EH * hiddengrad
    wout += hlayer_act.T.dot(d_output) *lr
    # dotproduct of nextlayererror and currentlayerop
    # bout += np.sum(d_output, axis=0,keepdims=True) *lr
    wh += X.T.dot(d_hiddenlayer) *lr
    #bh += np.sum(d_hiddenlayer, axis=0,keepdims=True) *lr
    print("Input: \n" + str(X))
    print("Actual Output: \n" + str(y))
    print('wh : ',wh)
    print('bh : ',bh)
```

```
print('wout : ',wout)
print('bout : ',bout)
print("Predicted Output: \n" ,output)
```

PROGRAM 6 : NAÏVE BAYES ALGORITHM

```
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('NBC.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_Pregnancies = LabelEncoder()
X.Pregnancies = le_Pregnancies.fit_transform(X.Pregnancies)

le_Glucose = LabelEncoder()
X.Glucose = le_Glucose.fit_transform(X.Glucose)

le_BloodPressure = LabelEncoder()
X.BloodPressure = le_BloodPressure.fit_transform(X.BloodPressure)

le_SkinThickness = LabelEncoder()
X.SkinThickness = le_SkinThickness.fit_transform(X.SkinThickness)

le_Insulin = LabelEncoder()
X.Insulin = le_Insulin.fit_transform(X.Insulin)

le_BMI = LabelEncoder()
X.BMI = le_BMI.fit_transform(X.BMI)

le_DiabeticPedigreeFunction = LabelEncoder()
X.DiabeticPedigreeFunction =
le_DiabeticPedigreeFunction.fit_transform(X.DiabeticPedigreeFunction)

le_Age = LabelEncoder()
X.Age = le_Age.fit_transform(X.Age)




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

le_Outcome = LabelEncoder()
y = le_Outcome.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)*100)
```

PROGRAM 7 : EM AND K MEANS ALGORITHM

```
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
import sklearn.metrics as metrics
import pandas as pd
names = ['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width',
'Class']
dataset = pd.read_csv("emkmeans.csv", names=names)
X = dataset.iloc[:, :-1]
label = {'Iris-setosa': 0,'Iris-versicolor': 1, 'Iris-virginica': 2}
y = [label[c] for c in dataset.iloc[:, -1]]
model=KMeans(n_clusters=3, random_state=0).fit(X)
print('The accuracy score of K-Mean: ',metrics.accuracy_score(y,
model.labels_))
print('The Confusion matrixof K-Mean:\n',metrics.confusion_matrix(y,
model.labels_))
gmm=GaussianMixture(n_components=3, random_state=0).fit(X)
y_cluster_gmm=gmm.predict(X)
print('The accuracy score of EM: ',metrics.accuracy_score(y,
y_cluster_gmm))
print('The Confusion matrix of EM:\n ',metrics.confusion_matrix(y,
y_cluster_gmm))
```

PROGRAM 8 : KNN

```
import pandas as pd

dataset = pd.read_csv('iris.csv')

feature_columns = ['sepal_length', 'sepal_width',
'petal_length','petal_width']

X = dataset[feature_columns].values

y = dataset['species'].values

from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

y = le.fit_transform(y)

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
random_state = 0)

from sklearn.neighbors import KNeighborsClassifier

classifier = KNeighborsClassifier(n_neighbors = 3)

classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)

print("y_pred y_test")

for i in range(len(y_pred)):

 print(y_pred[i], " ", y_test[i])

from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, y_pred)

print("Confusion Matrix:")

print(cm)

from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_test, y_pred)*100

print('Accuracy of our model is equal ' + str(round(accuracy, 2)) + ' %.')
```

PROGRAM 9 : LOCALLY WEEIGHTED REGRESSION

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

def kernel(point, xmat, k):
    m,n = np.shape(xmat)
    weights = np.mat(np.eye((m)))
    for j in range(m):
        diff = point - X[j]
        weights[j,j] = np.exp(diff*diff.T/(-2.0*k**2))
    return weights

def localWeight(point, xmat, ymat, k):
    wei = kernel(point,xmat,k)
    W = (X.T*(wei*X)).I*(X.T*(wei*ymat.T))
    return W

def localWeightRegression(xmat, ymat, k):
    m,n = np.shape(xmat)
    ypred = np.zeros(m)
    for i in range(m):
        ypred[i] = xmat[i]*localWeight(xmat[i],xmat,ymat,k)
    return ypred

# load data points
data = pd.read_csv('tips.csv')
bill = np.array(data.total_bill)
tip = np.array(data.tip)

#preparing and add 1 in bill
mbill = np.mat(bill)
mtip = np.mat(tip)

m= np.shape(mbill)[1]
one = np.mat(np.ones(m))
X = np.hstack((one.T,mbill.T))

#set k here
ypred = localWeightRegression(X,mtip,0.5)
SortIndex = X[:,1].argsort(0)
xsort = X[SortIndex][:,0]

fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.scatter(bill,tip, color='green')
ax.plot(xsort[:,1],ypred[SortIndex], color = 'red', linewidth=5)
plt.xlabel('Total bill')
plt.ylabel('Tip')
```

```
plt.show();
```