1)Write a program to implement BFS and DFS Traversal.

```
graph = {
   '5' : ['3','7'],
   '3' : ['2','4'],
   '7' : ['8'],
   '2' : [ ],
   '4' : ['8'],
   '8' : [ ],
}
visited = []
queue = []
def bfs(visited,graph,node):
  visited.append(node)
  queue.append(node)

  while queue:
      m=queue.pop(0)
      print(m,end = " ")
      for neighbour in graph[m]:
          if neighbour not in visited:
            visited.append(neighbour)
            queue.append(neighbour)

print("following is the breadth-first")
bfs(visited,graph,'5')



graph ={
    'A': ['S', 'B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'G'],
    'D': ['B'],
    'S': ['A', 'H'],
    'E': ['B'],
    'H': ['S', 'I', 'J'],
    'I': ['H', 'K'],
    'J': ['H'],
    'K': ['I'],
    'G': ['C'],
        }
visited =set()
def DFS(node,visited,graph):
    if node not in visited:
        print(node)
        visited.add(node)
        for i in graph[node]:
            DFS(i,visited,graph)
print("following is the DFS")
DFS("A",visited,graph)
```

2) Write a program to implement A* Search.

```
def aStarAlgo(start_node, stop_node):
```

```
open_set = set(start_node)
closed_set = set()
g = {}              #store distance from starting node
parents = {}        # parents contains an adjacency map of all nodes
#distance of starting node from itself is zero
g[start_node] = 0
#start_node is root node i.e it has no parent nodes
#so start_node is set to its own parent node
parents[start_node] = start_node
while len(open_set) > 0:
    n = None
    #node with lowest f() is found
    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):
            #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)
                parents[m] = n
                g[m] = g[n] + weight
            #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)
    closed_set.add(n)
    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
def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 5,
        'D': 7,
        'E': 3,
        'F': 6,
        'G': 5,
        'H': 3,
        'I': 1,
        'J': 0
    }
    return H_dist[n]
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('A', 6), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 1), ('E', 5)],
    'D': [('B', 2), ('C', 1), ('E', 8)],
    'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],
    'F': [('A', 3), ('G', 1), ('H', 7)],
    'G': [('F', 1), ('I', 3)],
    'H': [('F', 7), ('I', 2)],
    'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],
}

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

3) Write a program to implement Travelling Salesman Problem and Graph Coloring Problem

```
from sys import maxsize
from itertools import permutations
V = 4
def travellingSalesmanProblem(graph, s):
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)
    min_path = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:
        current_pathweight = 0
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]
        min_path = min(min_path, current_pathweight)
    return min_path

if __name__ == "__main__":
    graph = [[0, 10, 15, 20], [10, 0, 35, 25],
                    [15, 35, 0, 30], [20, 25, 30, 0]]
    s = 0
    print(travellingSalesmanProblem(graph, s))



colors=['red','blue','green','yellow','black']
states=['andhra','karnataka','tamilnadu','kerala']
neighbors={}
neighbors['andhra']=['karnataka','tamilnadu']
neighbors['karnataka']=['andhra','tamilnadu','kerala']
neighbors['tamilnadu']=['andhra','karnataka','kerala']
neighbors['kerala']=['karnataka','tamilnadu']
colors_of_states={}
def promising(state,color):
    for neighbor in neighbors.get(state):
        color_of_neighbor=colors_of_states.get(neighbor)
        if color_of_neighbor==color:
            return False
    return True
def get_color_for_state(state):
    for color in colors:
        if promising(state,color):
            return color
def main():
    for state in states:
      colors_of_states[state]=get_color_for_state(state)
    print(colors_of_states)
main()
```

4) Write a program to implement Knowledge Representation

```
from sympy import symbols, Or, Not, Implies, satisfiable
Rain = symbols('Rain')
Harry_Visited_Hagrid = symbols('Harry_Visited_Hagrid')
Harry_Visited_Dumbledore = symbols('Harry_Visited_Dumbledore')
sentence_1 = Implies((Rain), Harry_Visited_Hagrid)
sentence_2 = (Or(Harry_Visited_Hagrid, Harry_Visited_Dumbledore)
& Not(Harry_Visited_Hagrid & Harry_Visited_Dumbledore))
sentence_3 = Harry_Visited_Dumbledore
knowledge_base = sentence_1 & sentence_2 & sentence_3
solution = satisfiable(knowledge_base, all_models=True)
for model in solution:
    if model[Rain]:
        print("It rained today.")
    else:
        print("There is no rain today.")
```

5) Write a program to implement Bayesian Network.

```
import numpy
from pomegranate import *
guest = DiscreteDistribution({'A': 1./3, 'B': 1./3, 'C': 1./3})
prize = DiscreteDistribution({'A': 1./3, 'B': 1./3, 'C': 1./3})
monty = ConditionalProbabilityTable(
[[ 'A', 'A', 'A', 0.0 ],
[ 'A', 'A', 'B', 0.5 ],
[ 'A', 'A', 'C', 0.5 ],
[ 'A', 'B', 'A', 0.0 ],
[ 'A', 'B', 'B', 0.0 ],
[ 'A', 'B', 'C', 1.0 ],
[ 'A', 'C', 'A', 0.0 ],
[ 'A', 'C', 'B', 1.0 ],
[ 'A', 'C', 'C', 0.0 ],
[ 'B', 'A', 'A', 0.0 ],
[ 'B', 'A', 'B', 0.0 ],
[ 'B', 'A', 'C', 1.0 ],
[ 'B', 'B', 'A', 0.5 ],
[ 'B', 'B', 'B', 0.0 ],
[ 'B', 'B', 'C', 0.5 ],
[ 'B', 'C', 'A', 1.0 ],
[ 'B', 'C', 'B', 0.0 ],
[ 'B', 'C', 'C', 0.0 ],
[ 'C', 'A', 'A', 0.0 ],
[ 'C', 'A', 'B', 1.0 ],
[ 'C', 'A', 'C', 0.0 ],
[ 'C', 'B', 'A', 1.0 ],
[ 'C', 'B', 'B', 0.0 ],
[ 'C', 'B', 'C', 0.0 ],
[ 'C', 'C', 'A', 0.5 ],
[ 'C', 'C', 'B', 0.5 ],
[ 'C', 'C', 'C', 0.0 ]], [guest, prize])
s1 = State(guest, name="guest")
s2 = State(prize, name="prize")
s3 = State(monty, name="monty")
model = BayesianNetwork("Monty Hall Problem")
model.add_states(s1, s2, s3)
model.add_edge(s1, s3)
model.add_edge(s2, s3)
model.bake()
print(model.probability([['A', 'B', 'C'],['A','A','C'],['A','C','C']]))
print(model.predict([['A',None,'C'],['A','A',None],[None,'B','A']]))
```

6) Write a program to implement Hidden Markov Model.

```
import numpy as np
import itertools
import pandas as pd
# create state space and initial state probabilities
```

```
states = ['sleeping', 'eating', 'walking']

hidden_states = ['healthy', 'sick']
pi = [0.5, 0.5]
state_space = pd.Series(pi, index=hidden_states, name='states')
print(state_space)



a_df = pd.DataFrame(columns=hidden_states, index=hidden_states)
a_df.loc[hidden_states[0]] = [0.7, 0.3]
a_df.loc[hidden_states[1]] = [0.4, 0.6]

print(a_df)

observable_states = states

b_df = pd.DataFrame(columns=observable_states, index=hidden_states)
b_df.loc[hidden_states[0]] = [0.2, 0.6, 0.2]
b_df.loc[hidden_states[1]] = [0.4, 0.1, 0.5]

print(b_df)

def HMM(obsq,a_df,b_df,pi,states,hidden_states):
                    hidst=list(itertools.product(hidden_states,repeat=len(obsq)))
                    print(hidst)
                    sum=0
                    for k in hidst:
                                        prod=1
                                        for j in range(len(k)):
                                                  c=0
                                                  for i in obsq:
                                                            if c==0:

prod*=a_df[i][k[j]]*pi[hidden_states.index(k[j])]
                                                                      c=1
                                                            else:
                                                                      prod*=b_df[k[j]][k[j-
1]]*a_df[i][k[j]]
                                        sum+=prod
                                        c=0
                    return sum
def vertibi(obsq,a_df,b_df,pi,states,hidden_states):
                    sum=0
                    hidst=list(itertools.product(hidden_states,repeat=len(obsq)))
                    for k in hidst:
                                        sum1=0
                                        prod=1
                                        for j in range(len(k)):
                                                  c=0
                                                  for i in obsq:
                                                            if c==0:

          prod*=a_df[i][k[j]]*pi[hidden_states.index(k[j])]
                                                                      c=1
                                                            else:
```

```
                                                                        prod*=b_df[k[j]][k[j-
1]]*a_df[i][k[j]]
                                        c=0
                                        sum1+=prod
                                        if(sum1>sum):
                                                  sum=sum1
                                                  hs=k
                    return sum,hs




obsq=['walking','walking','walking']
print(HMM(obsq,b_df,a_df,pi,states,hidden_states))
print(vertibi(obsq,b_df,a_df,pi,states,hidden_states))
```

7) Write a program to implement Regression algorithm.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
dataset = pd.read_csv('/content/Salary_Data (2).csv')
dataset.head()
# data preprocessing
X = dataset.iloc[:, :-1].values  #independent variable array
y = dataset.iloc[:,1].values  #dependent variable vector
# splitting the dataset
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=1/3,random_state=0)
# fitting the regression model
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train,y_train) #actually produces the linear eqn for the data
# predicting the test set results
y_pred = regressor.predict(X_test)
y_pred
y_test
# visualizing the results
#plot for the TRAIN
plt.scatter(X_train, y_train, color='red') # plotting the observation line
plt.plot(X_train, regressor.predict(X_train), color='blue') # plotting the regression line
plt.title("Salary vs Experience (Training set)") # stating the title of the graph
plt.xlabel("Years of experience") # adding the name of x-axis
plt.ylabel("Salaries") # adding the name of y-axis
plt.show() # specifies end of graph
#plot for the TEST
plt.scatter(X_test, y_test, color='red')
plt.plot(X_train, regressor.predict(X_train), color='blue') # plotting the regression line
plt.title("Salary vs Experience (Testing set)")
plt.xlabel("Years of experience")
plt.ylabel("Salaries")
plt.show()
```

8) Write a program to implement decision tree based ID3 algorithm.

```
import pandas as pd
df=pd.read_csv("PlayTennis - PlayTennis.csv")
print(df)

def entropy(probs):
 import math
 return sum(-prob*math.log(prob,2) for prob in probs)

def entropy_of_list(a_list):
 from collections import Counter
 cnt = Counter (x for x in a_list)
 print(cnt)
 num_instances =len(a_list)
 probs=[x/num_instances for x in cnt.values()]
 print(num_instances)
 print(probs)
 return entropy(probs)
total_entropy= entropy_of_list(df['Play Tennis'])
print(total_entropy)

def information_gain(df,split_attribute_name, target_attribute_name, trace=0):
  df_split =df.groupby(split_attribute_name)
  print(df_split)
  for name,group in df_split:
    print("Name",name)
    print("Group",group)
    nobs=len(df.index)*1.0
    print(nobs)
    print("NOBS",nobs)
    df_agg_ent=df_split.agg({target_attribute_name: [entropy_of_list,lambda x: len(x)/nobs]
})[target_attribute_name]
    avg_info=sum(df_agg_ent['entropy_of_list'] * df_agg_ent['<lambda_0>'])
    old_entropy=entropy_of_list(df[target_attribute_name])
    return old_entropy-avg_info

def id3DT(df, target_attribute_name, attribute_names, default_class=None):
  from collections import Counter
  cnt = Counter(x for x in df[target_attribute_name])
  if len(cnt)==1:
        return next(iter(cnt))
    elif df.empty or (not attribute_names):
        return default_class
    else:
        default_class =max(cnt.keys())
#print("attributes_names:",attribute_names)
        gainz=[information_gain(df,attr, target_attribute_name) for attr in attribute_names]
        index_of_max=gainz.index(max(gainz))
        best_attr=attribute_names[index_of_max]
        tree={best_attr:{}}
        remaining_attributes_names=[i for i in attribute_names if i != best_attr]
        for attr_val, data_subset in df.groupby(best_attr):
         subtree=id3DT(data_subset,target_attribute_name,remaining_attributes_names,default_class)
         tree[best_attr][attr_val]=subtree
    return tree
attribute_names=list(df.columns)
attribute_names.remove('Play Tennis')

from pprint import pprint
tree= id3DT(df,'Play Tennis',attribute_names)
print("The Resultant Decision Tree is ")
pprint(tree)
attribute=next(iter(tree))
print("Best Attribute: \n", attribute)
print("Tree Keys\n ", tree[attribute].keys())

def classify(instance, tree, default=None):
    attribute=next(iter(tree))
    print("Key:",tree.keys())
    print("Attribute",attribute)
    if instance[attribute] in tree[attribute].keys():
      result=tree[attribute][instance[attribute]]
      print("Instance Attribute:",instance[attribute], "TreeKeys:",tree[attribute].keys())
      if isinstance(result,dict):
         return classify(instance,result)
      else:
         return result
    else:
      return default
tree1={'Outlook':['Rain','Sunny'],'Temperature':['Mild','Hot'],'Humidity':['High','High'],'Wind':['
Weak','Weak']}
df2=pd.DataFrame(tree1)
df2['Predicted']=df2.apply(classify,axis=1, args=(tree,'No'))
print(df2)
```

9) Write a program to implement K-Means Clustering algorithm.

```
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
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=['target']
plt.figure(figsize=(14,7))
colormap=np.array(['red','lime','black'])
plt.subplot(1,2,1)
plt.scatter(X.Sepal_Length,X.Sepal_Width,c=colormap[y.target],s=40)
plt.title('Sepal')
plt.subplot(1,2,2)
plt.scatter(X.Petal_length,X.Petal_Width,c=colormap[y.target],s=40)
plt.title('Petal')
model=KMeans(n_clusters=3)
model.fit(X)
print(model.labels_)
plt.subplot(1,2,1)
plt.scatter(X.Petal_length,X.Petal_Width,c=colormap[y.target],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( 'KMEANS Classfication')
```

10) Write a program to implement K-Nearest Neighbor algorithm (K-NN).

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
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']
print(X)

y = pd.DataFrame(iris.target)
y.columns = ['Targets']
print(y)

#Split the data into train and test samples
x_train,x_test,y_train,y_test = train_test_split(iris.data,iris.target,test_size=0.1)
print("Dataset is split into training and testing...")
print("Size of training data and its label",x_train.shape,y_train.shape)
print("Size of testing data and its label",x_test.shape,y_test.shape)

# prints Label no. and their names
for i in range(len(iris.target_names)):
  print("Label", i , "-",str(iris.target_names[i]))

#create object of KNN classifer
classifer = KNeighborsClassifier(n_neighbors=3)

#perform Training
classifer.fit(x_train, y_train)#perform teating
y_pred=classifer.predict(x_test)

#Display the results
print("Results of Classification using K-nn with K=3")
for r in range(0,len(x_test)):
  print(" sample:", str(x_test[r]), " Actual-label:",str(y_test[r]), " predict-label:", str(y_pred[r]))
print("Classification Accuracy :" , classifer.score(x_test,y_test))


from sklearn.metrics import classification_report, confusion_matrix
print('Confusion Matrix')
print(confusion_matrix(y_test,y_pred))
print('Accuracy Ketrics')
print(classification_report(y_test,y_pred))
```

11) Write a program to implement Back Propagation Algorithm.

```
import numpy as np
X = np.array(([2,9],[1,5],[3,6])) #Hours Studied,Hours Slept
y=np.array(([92],[86],[89])) #Test Score

y=y/100 #Max Test Score is 100
```

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

#Derivatives of Sigmoid function
def derivatives_sigmoid(x):
  return x*(1-x)
#Variable initialization
epoch=10000 #Setting training iterations
lr=0.1 #Setting learning rate
inputlayer_neurons = 2 #number of features in data set
hiddenlayers_neurons = 3 #number of hidden layers neurons
output_neurons = 1 #number of neurons of output layer

#weight and bias initialization
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayers_neurons))
bias_hidden=np.random.uniform(size=(1,hiddenlayers_neurons))  #bias matrix to the hidden
layer
weight_hidden=np.random.uniform(size=(hiddenlayers_neurons,output_neurons)) #weight
matrix to the output layer
bias_output=np.random.uniform(size=(1,output_neurons)) #matrix to output layer

for i in range(epoch):
  hinp1=np.dot(X,wh)
  hinp=hinp1+ bias_hidden
  hlayer_activation = sigmoid(hinp)

  outinp1=np.dot(hlayer_activation,weight_hidden)
  outinp = outinp1+bias_output
  output = sigmoid(outinp)

EO = y-output
outgrad=derivatives_sigmoid(output)
d_output = EO * outgrad
EH = d_output.dot(weight_hidden.T)
hiddengrad=derivatives_sigmoid(hlayer_activation)
d_hiddenlayer = EH * hiddengrad

weight_hidden += hlayer_activation.T.dot(d_output) * lr
bias_hidden += np.sum(d_hiddenlayer, axis=0,keepdims=True) * lr
wh += X.T.dot(d_hiddenlayer) * lr
bias_output += np.sum(d_output,axis=0,keepdims=True) *lr

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

12) Write a program to implement Support Vector Machine.

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt

# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data  # Features: sepal length, sepal width, petal length, petal width
y = iris.target  # Labels: three species of Iris

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize the SVM classifier
svm_model = SVC(kernel='linear')  # You can also try 'rbf', 'poly', etc.

# Train the SVM model on the training data
svm_model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = svm_model.predict(X_test)

# Evaluate the model's performance
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
print("Classification Report:\n", classification_report(y_test, y_pred))
```