Progarm 1:Astar

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': 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,

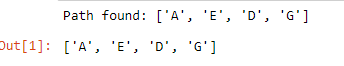
'E': [('D', 6)],

'D': [('G', 1)],

}

aStarAlgo('A', 'G')

output:



Program 2:Ao star

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

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 = {}

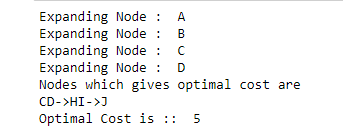
optimal\_cost = recAOStar('A')

print('Nodes which gives optimal cost are')

print\_path('A')

print('\nOptimal Cost is :: ', optimal\_cost)

OUTPUT:



PROGRAM 3:CANDIATE ELIMINATION

import numpy as np

import pandas as pd

data = pd.DataFrame(data=pd.read\_csv('p3 .csv'))

concepts = np.array(data.iloc[:,0:-1])

target = np.array(data.iloc[:,-1])

def learn(concepts, target):

specific\_h = concepts[0].copy()

print("\ninitialization of specific\_h and general\_h")

print(specific\_h)

general\_h = [["?" for i in range(len(specific\_h))] for i in range(len(specific\_h))]

print(general\_h)

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] = '?'

if target[i] == "No":

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("\nsteps of Candidate Elimination Algorithm",i+1)

print(specific\_h)

print(general\_h)

indices = [i for i, val in enumerate(general\_h) if val == ['?', '?', '?', '?', '?', '?']]

for i in indices:

general\_h.remove(['?', '?', '?', '?', '?', '?'])

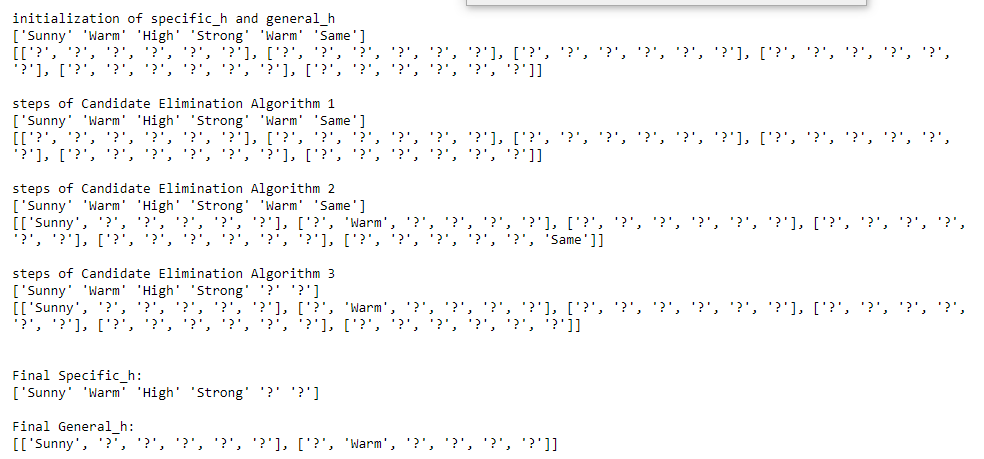
return specific\_h, general\_h

s\_final, g\_final = learn(concepts, target)

print("\n\nFinal Specific\_h:", s\_final, sep="\n")

print("\nFinal General\_h:", g\_final, sep="\n")

OUTPUT:



PROGRAM 4:ID3

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.playtennis=""

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.playtennis=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.playtennis!="":

print(" "\*level,node.playtennis)

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.playtennis!="":

print(node.playtennis)

return

pos=features.index(node.attribute)

for value, n in node.children:

if x\_test[pos]==value:

classify(n,x\_test,features)

'''Main program'''

dataset,features=load\_csv("p4.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("id3.csv")

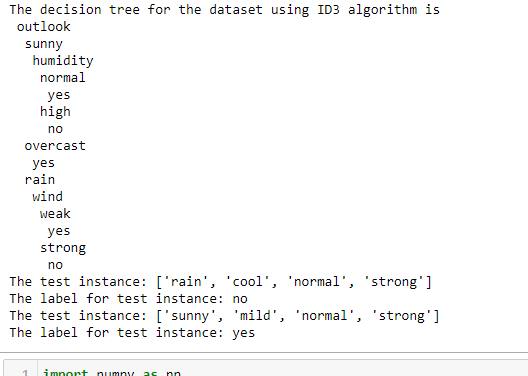
for xtest in testdata:

print("The test instance:",xtest)

print("The label for test instance:",end=" ")

classify(node1,xtest,features)

OUTPUT:



PROGRAM 5:BACKPROPAGATION

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=6000

lr=0.1

inputlayer\_neurons = 2

hiddenlayer\_neurons = 3

output\_neurons = 1

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))

for i in range(epoch):

hinp1=np.dot(X,wh)

hinp=hinp1 + bh

hlayer\_act = sigmoid(hinp)

outinp1=np.dot(hlayer\_act,wout)

outinp= outinp1+ bout

output = sigmoid(outinp)

EO = y-output

outgrad = derivatives\_sigmoid(output)

d\_output = EO\* outgrad

EH = d\_output.dot(wout.T)

hiddengrad = derivatives\_sigmoid(hlayer\_act)

d\_hiddenlayer = EH \* hiddengrad

wout += hlayer\_act.T.dot(d\_output) \*lr

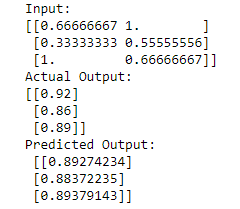
wh += X.T.dot(d\_hiddenlayer) \*lr

print("Input: \n" + str(X))

print("Actual Output: \n" + str(y))

print("Predicted Output: \n" ,output)

OUTPUT:



PROGRAM 6:

import csv

import random

import math

def loadcsv(filename):

lines = csv.reader(open(filename, "r"));

dataset = list(lines)

for i in range(len(dataset)):

dataset[i] = [float(x) for x in dataset[i]]

return dataset

def splitdataset(dataset, splitratio):

trainsize = int(len(dataset) \* splitratio)

trainset = []

copy = list(dataset)

while len(trainset) < trainsize:

index = random.randrange(len(copy));

trainset.append(copy.pop(index))

return [trainset, copy]

def separatebyclass(dataset):

separated = {}

for i in range(len(dataset)):

vector = dataset[i]

if (vector[-1] not in separated):

separated[vector[-1]] = []

separated[vector[-1]].append(vector)

return separated

def mean(numbers):

return sum(numbers)/float(len(numbers))

def stdev(numbers):

avg = mean(numbers)

variance = sum([pow(x-avg,2) for x in numbers])/float(len(numbers)-1)

return math.sqrt(variance)

def summarize(dataset):

summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(\*dataset)];

del summaries[-1]

return summaries

def summarizebyclass(dataset):

separated = separatebyclass(dataset)

summaries = {}

for classvalue, instances in separated.items():

summaries[classvalue] = summarize(instances)

return summaries

def calculateprobability(x, mean, stdev):

exponent = math.exp(-(math.pow(x-mean,2)/(2\*math.pow(stdev,2))))

return (1 / (math.sqrt(2\*math.pi) \* stdev)) \* exponent

def calculateclassprobabilities(summaries, inputvector):

probabilities = {}

for classvalue, classsummaries in summaries.items():

probabilities[classvalue] = 1

for i in range(len(classsummaries)):

mean, stdev = classsummaries[i]

x = inputvector[i]

probabilities[classvalue] \*=calculateprobability(x, mean, stdev);

return probabilities

def predict(summaries, inputvector):

probabilities = calculateclassprobabilities(summaries, inputvector)

bestLabel, bestProb = None, -1

for classvalue, probability in probabilities.items():

if bestLabel is None or probability > bestProb:

bestProb = probability

bestLabel = classvalue

return bestLabel

def getpredictions(summaries, testset):

predictions = []

for i in range(len(testset)):

result = predict(summaries, testset[i])

predictions.append(result)

return predictions

def getaccuracy(testset, predictions):

correct = 0

for i in range(len(testset)):

if testset[i][-1] == predictions[i]:

correct += 1

return (correct/float(len(testset))) \* 100.0

def main():

filename = 'naivedata.csv'

splitratio = 0.67

dataset = loadcsv(filename);

trainingset, testset = splitdataset(dataset, splitratio)

print('Split {0} rows into train={1} and test={2}rows'.format(len(dataset), len(trainingset),len(testset)))

summaries = summarizebyclass(trainingset);

predictions = getpredictions(summaries, testset)

accuracy = getaccuracy(testset, predictions)

print('Accuracy of the classifier is :{0}%'.format(accuracy))

main()

OUTPUT:



PROGRAM 7:EM 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 = ['Targets']

model = KMeans(n\_clusters=3)

model.fit(X)

plt.figure(figsize=(14,7))

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

plt.subplot(2, 2, 1)

plt.scatter(X.Petal\_Length, X.Petal\_Width, c=colormap[y.Targets], s=40)

plt.title('Real Classification')

plt.xlabel('Petal Length')

plt.ylabel('Petal Width')

plt.subplot(2, 2, 2)

plt.scatter(X.Petal\_Length, X.Petal\_Width, c=colormap[model.labels\_], s=40)

plt.title('K Mean Classification')

plt.xlabel('Petal Length')

plt.ylabel('Petal Width')

print('The accuracy score of K-Mean:\n ',sm.accuracy\_score(y, model.labels\_))

print('The Confusion matrixof K-Mean:\n ',sm.confusion\_matrix(y, model.labels\_))

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\_gmm = gmm.predict(xs)

plt.subplot(2, 2, 3)

plt.scatter(X.Petal\_Length, X.Petal\_Width, c=colormap[y\_gmm], s=40)

plt.title('GMM Classification')

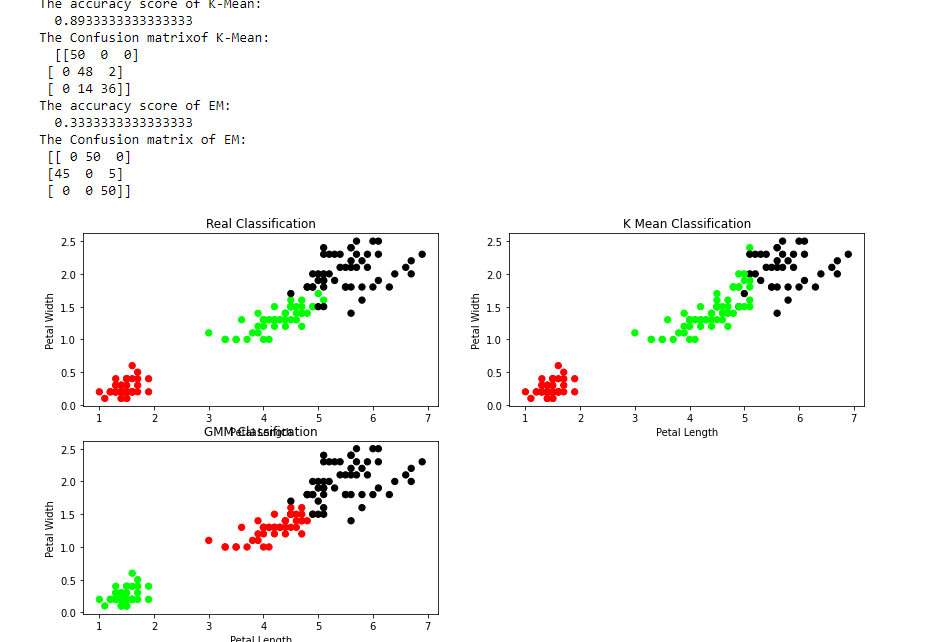
plt.xlabel('Petal Length')

plt.ylabel('Petal Width')

print('The accuracy score of EM:\n ',sm.accuracy\_score(y, y\_gmm))

print('The Confusion matrix of EM: \n',sm.confusion\_matrix(y, y\_gmm))

OUTPUT:



PROGRAM 8:KN NEIGHOUR

from sklearn.model\_selection import train\_test\_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import classification\_report, confusion\_matrix

from sklearn import datasets

iris=datasets.load\_iris()

x = iris.data

y = iris.target

print ('sepal-length', 'sepal-width', 'petal-length', 'petal-width')

print(x)

print('class: 0-Iris-Setosa, 1- Iris-Versicolour, 2- Iris-Virginica')

print(y)

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x,y,test\_size=0.3)

#To Training the model and Nearest nighbors K=5

classifier = KNeighborsClassifier(n\_neighbors=5)

classifier.fit(x\_train, y\_train)

#To make predictions on our test data

y\_pred=classifier.predict(x\_test)

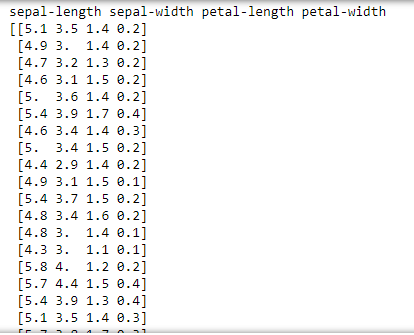
print('Confusion Matrix')

print(confusion\_matrix(y\_test,y\_pred))

print('Accuracy Metrics')

print(classification\_report(y\_test,y\_pred))

OUTPUT:



PROGRAM 9:LOCALLY WEIGHTED

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

#the Gaussian Kernel

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

#convert to matrix form

mbill = np.mat(bill)

mtip = np.mat(tip)

m= np.shape(mbill)[1]

one = np.ones((1,m),dtype=int)

#horizontally stack

X= np.hstack((one.T,mbill.T))

print("X.shape:",X.shape)

#set k here (0.5)

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')

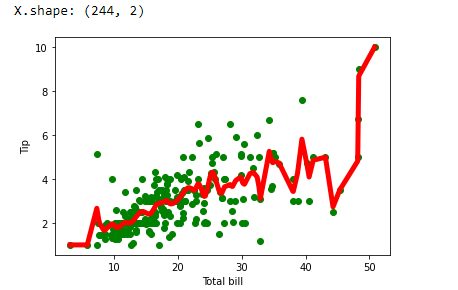
ax.plot(xsort[:,1],ypred[SortIndex], color = 'red', linewidth=5)

plt.xlabel('Total bill')

plt.ylabel('Tip')

plt.show()

OUTPUT:



Travel sale man program

from sys import maxsize

from itertools import permutations

V = 4

# implementation of traveling Salesman Problem

def travellingSalesmanProblem(graph, s):

    # store all vertex apart from source vertex

    vertex = []

    for i in range(V):

        if i != s:

            vertex.append(i)

    # store minimum weight Hamiltonian Cycle

    min\_path = maxsize

    next\_permutation=permutations(vertex)

    for i in next\_permutation:

        # store current Path weight(cost)

        current\_pathweight = 0

        # compute current path weight

        k = s

        for j in i:

            current\_pathweight += graph[k][j]

            k = j

        current\_pathweight += graph[k][s]

        # update minimum

        min\_path = min(min\_path, current\_pathweight)

    return min\_path

# Driver Code

if \_\_name\_\_ == "\_\_main\_\_":

    # matrix representation of graph

    graph = [[0, 10, 15, 20], [10, 0, 35, 25],

            [15, 35, 0, 30], [20, 25, 30, 0]]

    s = 0

    print(travellingSalesmanProblem(graph, s))

output:80

WATER JUG PROBLEM

# This function is used to initialize the

# dictionary elements with a default value.

from collections import defaultdict

# jug1 and jug2 contain the value

# for max capacity in respective jugs

# and aim is the amount of water to be measured.

jug1, jug2, aim = 4, 3, 2

# Initialize dictionary with

# default value as false.

visited = defaultdict(lambda: False)

# Recursive function which prints the

# intermediate steps to reach the final

# solution and return boolean value

# (True if solution is possible, otherwise False).

# amt1 and amt2 are the amount of water present

# in both jugs at a certain point of time.

def waterJugSolver(amt1, amt2):

    # Checks for our goal and

    # returns true if achieved.

    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):

        print(amt1, amt2)

        return True

    # Checks if we have already visited the

    # combination or not. If not, then it proceeds further.

    if visited[(amt1, amt2)] == False:

        print(amt1, amt2)

        # Changes the boolean value of

        # the combination as it is visited.

        visited[(amt1, amt2)] = True

        # Check for all the 6 possibilities and

        # see if a solution is found in any one of them.

        return (waterJugSolver(0, amt2) or

                waterJugSolver(amt1, 0) or

                waterJugSolver(jug1, amt2) or

                waterJugSolver(amt1, jug2) or

                waterJugSolver(amt1 + min(amt2, (jug1-amt1)),

                amt2 - min(amt2, (jug1-amt1))) or

                waterJugSolver(amt1 - min(amt1, (jug2-amt2)),

                amt2 + min(amt1, (jug2-amt2))))

    # Return False if the combination is

    # already visited to avoid repetition otherwise

    # recursion will enter an infinite loop.

    else:

        return False

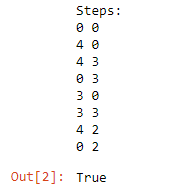
print("Steps: ")

# Call the function and pass the

# initial amount of water present in both jugs.

waterJugSolver(0, 0)

OUTPUT:



DFS:

# Using a Python dictionary to act as an adjacency list

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = set() # Set to keep track of visited nodes of graph.

**def** **dfs**(visited, graph, node): #function for dfs

**if** node **not** **in** visited:

**print** (node)

visited.add(node)

**for** neighbour **in** graph[node]:

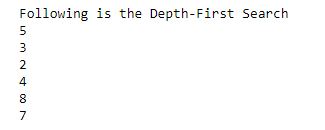
dfs(visited, graph, neighbour)

# Driver Code

**print**("Following is the Depth-First Search")

dfs(visited, graph, '5')

OUTPUT:



TIC TOC TOE

# Tic-Tac-Toe Program using

# random number in Python

# importing all necessary libraries

import numpy as np

import random

from time import sleep

# Creates an empty board

def create\_board():

return(np.array([[0, 0, 0],

[0, 0, 0],

[0, 0, 0]]))

# Check for empty places on board

def possibilities(board):

l = []

for i in range(len(board)):

for j in range(len(board)):

if board[i][j] == 0:

l.append((i, j))

return(l)

# Select a random place for the player

def random\_place(board, player):

selection = possibilities(board)

current\_loc = random.choice(selection)

board[current\_loc] = player

return(board)

# Checks whether the player has three

# of their marks in a horizontal row

def row\_win(board, player):

for x in range(len(board)):

win = True

for y in range(len(board)):

if board[x, y] != player:

win = False

continue

if win == True:

return(win)

return(win)

# Checks whether the player has three

# of their marks in a vertical row

def col\_win(board, player):

for x in range(len(board)):

win = True

for y in range(len(board)):

if board[y][x] != player:

win = False

continue

if win == True:

return(win)

return(win)

# Checks whether the player has three

# of their marks in a diagonal row

def diag\_win(board, player):

win = True

y = 0

for x in range(len(board)):

if board[x, x] != player:

win = False

if win:

return win

win = True

if win:

for x in range(len(board)):

y = len(board) - 1 - x

if board[x, y] != player:

win = False

return win

# Evaluates whether there is

# a winner or a tie

def evaluate(board):

winner = 0

for player in [1, 2]:

if (row\_win(board, player) or

col\_win(board,player) or

diag\_win(board,player)):

winner = player

if np.all(board != 0) and winner == 0:

winner = -1

return winner

# Main function to start the game

def play\_game():

board, winner, counter = create\_board(), 0, 1

print(board)

sleep(2)

while winner == 0:

for player in [1, 2]:

board = random\_place(board, player)

print("Board after " + str(counter) + " move")

print(board)

sleep(2)

counter += 1

winner = evaluate(board)

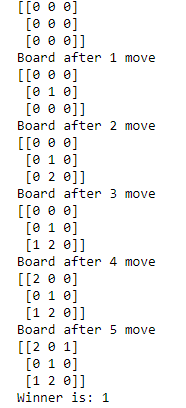
if winner != 0:

break

return(winner)

# Driver Code

print("Winner is: " + str(play\_game()))



8 PUZZLE

from collections import defaultdict

# This class represents a directed graph

# using adjacency list representation

class Graph:

# Constructor

def \_\_init\_\_(self):

# default dictionary to store graph

self.graph = defaultdict(list)

# function to add an edge to graph

def addEdge(self,u,v):

self.graph[u].append(v)

# Function to print a BFS of graph

def BFS(self, s):

# Mark all the vertices as not visited

visited = [False] \* (max(self.graph) + 1)

# Create a queue for BFS

queue = []

# Mark the source node as

# visited and enqueue it

queue.append(s)

visited[s] = True

while queue:

# Dequeue a vertex from

# queue and print it

s = queue.pop(0)

print (s, end = " ")

# Get all adjacent vertices of the

# dequeued vertex s. If a adjacent

# has not been visited, then mark it

# visited and enqueue it

for i in self.graph[s]:

if visited[i] == False:

queue.append(i)

visited[i] = True

# Driver code

# Create a graph given in

# the above diagram

g = Graph()

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)

g.addEdge(3, 3)

print ("Following is Breadth First Traversal"

" (starting from vertex 2)")

g.BFS(2)



FINDS ALGORITHM

import pandas as pd

df = pd.read\_csv("p3 .csv")

spe\_df = df.loc[df["enjoy sport"].str.upper()=="YES"]

gene\_df = df.loc[df["enjoy sport"].str.upper()=="NO"]

spe\_df = spe\_df.iloc[:,:-1]

gene\_df = gene\_df.iloc[:,:-1]

base = spe\_df.iloc[0]

for x in range(1,len(spe\_df)):

base = base.where(spe\_df.iloc[x]==base,other="???")

print("Specific :- \n",base.values)

BEST FIRST SEARCH

from queue import PriorityQueue

v = 14

graph = [[] for i in range(v)]

# Function For Implementing Best First Search

# Gives output path having lowest cost

def best\_first\_search(source, target, n):

visited = [0] \* n

visited = True

pq = PriorityQueue()

pq.put((0, source))

while pq.empty() == False:

u = pq.get()[1]

# Displaying the path having lowest cost

print(u, end=" ")

if u == target:

break

for v, c in graph[u]:

if visited[v] == False:

visited[v] = True

pq.put((c, v))

print()

# Function for adding edges to graph

def addedge(x, y, cost):

graph[x].append((y, cost))

graph[y].append((x, cost))

# The nodes shown in above example(by alphabets) are

# implemented using integers addedge(x,y,cost);

addedge(0, 1, 3)

addedge(0, 2, 6)

addedge(0, 3, 5)

addedge(1, 4, 9)

addedge(1, 5, 8)

addedge(2, 6, 12)

addedge(2, 7, 14)

addedge(3, 8, 7)

addedge(8, 9, 5)

addedge(8, 10, 6)

addedge(9, 11, 1)

addedge(9, 12, 10)

addedge(9, 13, 2)

source = 0

target = 9

best\_first\_search(source, target, v)

BREADTH FIRST SEARCH

from collections import defaultdict

# This class represents a directed graph

# using adjacency list representation

class Graph:

# Constructor

def \_\_init\_\_(self):

# default dictionary to store graph

self.graph = defaultdict(list)

# function to add an edge to graph

def addEdge(self,u,v):

self.graph[u].append(v)

# Function to print a BFS of graph

def BFS(self, s):

# Mark all the vertices as not visited

visited = [False] \* (max(self.graph) + 1)

# Create a queue for BFS

queue = []

# Mark the source node as

# visited and enqueue it

queue.append(s)

visited[s] = True

while queue:

# Dequeue a vertex from

# queue and print it

s = queue.pop(0)

print (s, end = " ")

# Get all adjacent vertices of the

# dequeued vertex s. If a adjacent

# has not been visited, then mark it

# visited and enqueue it

for i in self.graph[s]:

if visited[i] == False:

queue.append(i)

visited[i] = True

# Driver code

# Create a graph given in

# the above diagram

g = Graph()

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)

g.addEdge(3, 3)

print ("Following is Breadth First Traversal"

" (starting from vertex 2)")

g.BFS(2)