**Artificial Intelligence Course**

**Lab Manual Report**



**Mam Saba Aslam**

Student Name: Abdul Salam Wasti

Roll No: 18F-0326 Section: CS(6B)

Date performed: 11/07/2021

Contents

[Lab#1 (Introduction to AI, python & python IDE): 4](#_Toc76943136)

[Lab#2 (Python Basics to Advance): 5](#_Toc76943137)

[Task 01: 5](#_Toc76943138)

[Task 02: 6](#_Toc76943139)

[Python Graph (Adjacency matrix): 8](#_Toc76943140)

[Lab#3 Graph Traverse: 10](#_Toc76943141)

[Task 01 (DFS): 10](#_Toc76943142)

[Task 02 (BFS): 11](#_Toc76943143)

[Lab#4 Searches: 12](#_Toc76943144)

[Task 01 (UCS): 12](#_Toc76943145)

[Task 02 (BDS): 13](#_Toc76943146)

[Lab#5 A\* Search: 16](#_Toc76943147)

[Task 01 (UCS): 16](#_Toc76943148)

[Lab#6 Simulated annealing: 20](#_Toc76943149)

[Task 01 : 20](#_Toc76943150)

[Lab#7 MinMax with pruning: 23](#_Toc76943151)

[Task 01 : 23](#_Toc76943152)

[Lab#8 CSP: 25](#_Toc76943153)

[Task 01 : 25](#_Toc76943154)

[Task 02: 25](#_Toc76943155)

[Lab#9 Prepositional Logic: 27](#_Toc76943156)

[Task 01: 27](#_Toc76943157)

[Project 01 K-mean, K-medoid: 28](#_Toc76943158)

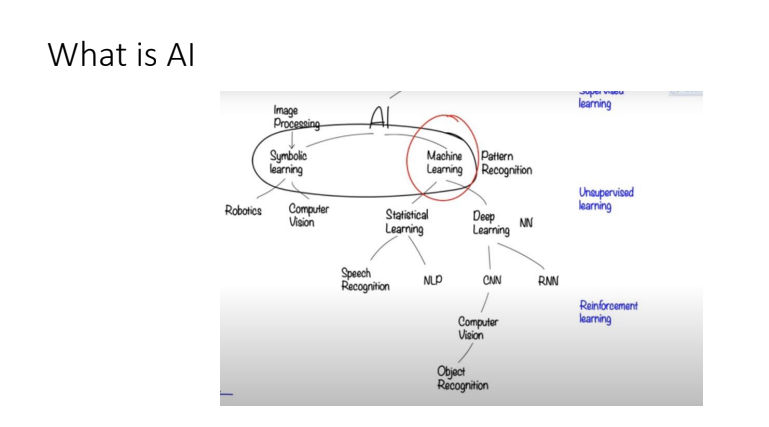
[Task 01 K-mean: 28](#_Toc76943159)

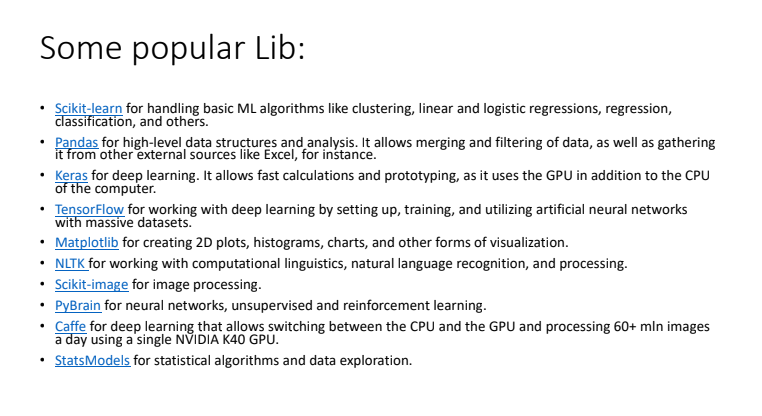
[Task 02 K-medoid: 33](#_Toc76943160)

[Project 02 Artificial Neural Network: 38](#_Toc76943161)

[Task 01 ANN: 38](#_Toc76943162)

# Lab#1 (Introduction to AI, python & python IDE):



**Python**

**Python IDE**



# Lab#2 (Python Basics to Advance):

## Task 01:

Write a program to take values from users to create a list, ask him how many number of values

he wants to add and print them.

Solution:

Code:

#declaring the list

mylist = []

#defining size of list

num = int(input(&quot;Enter size of list: &quot;))

#taking inputs using for loop

for n in range(num):

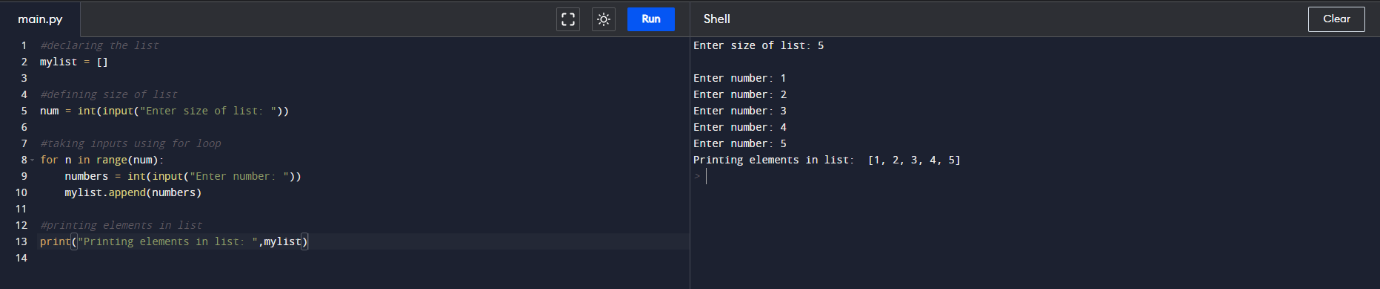
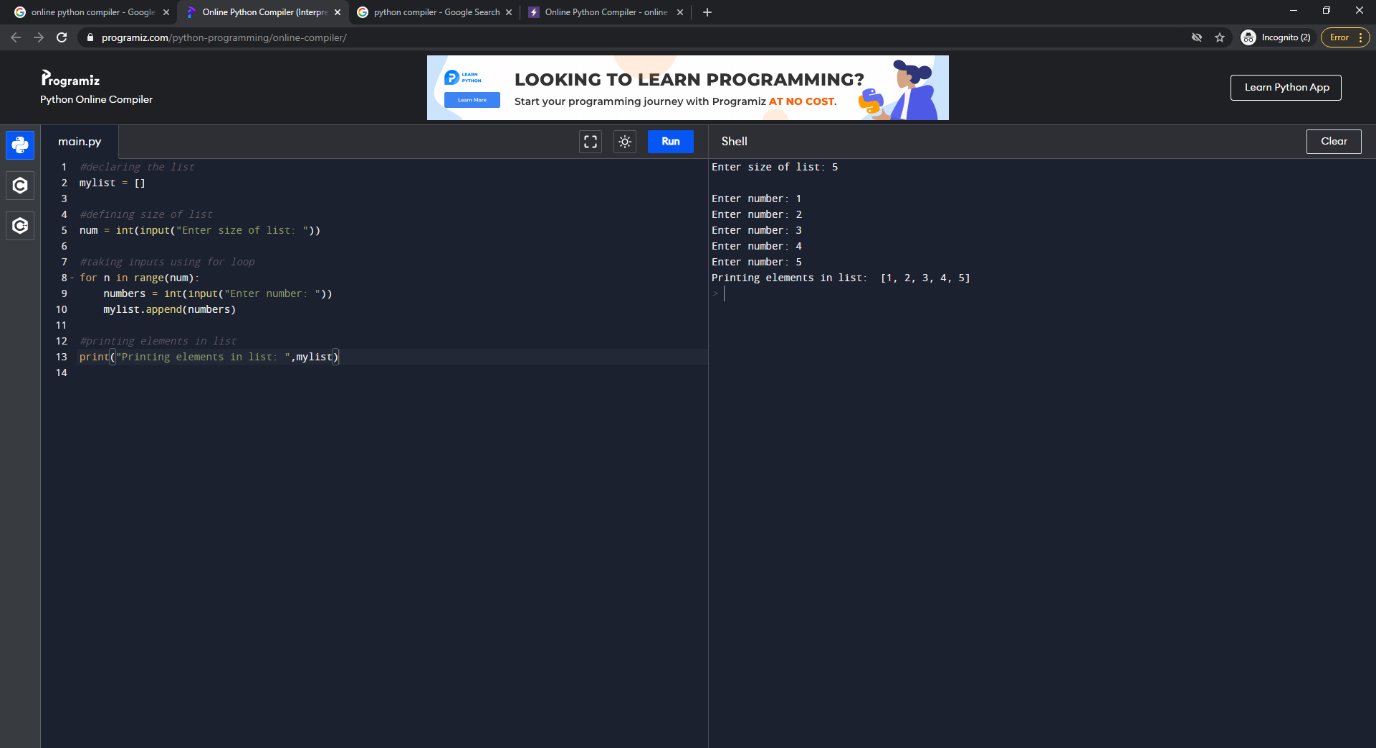
numbers = int(input(&quot;Enter number: &quot;))

mylist.append(numbers)

#printing elements in list

print(&quot;Printing elements in list: &quot;,mylist)

Screenshot:



## Task 02:

A person wants to go from Faisalsbad to Karachi, if the distance between these cities is. 1,115.3 km. Ask person if he wants to travel By air or by road. Aeroplan speed is 800 km/h. Car speed is 100 km/h. Write program in python to creating a class Travel , and two functions Fly and Road which can. calulate travel time by each means , ask him about his preferences to travel and tell him how

much time he will take to reach.

Solution:

Code:

#class defined

class Travel:

    #defining constructor

    def \_\_init\_\_(ob):

        ob.Hour=0

        ob.Min=0

        ob.timetaken=0

        ob.TotalDistance=1115.3

    #defining fly function

    def Fly(ob):

        Speed=800

        time=ob.TotalDistance/Speed

        ob.Hour=int(time)

        ob.Min = int((time - ob.Hour) \* 60)

        print("Time to reach your destination by aeroplane is:" + str(ob.Hour) + " hr " + str(ob.Min) + " min.")

    #defining road function

    def Road(ob):

        Speed=100

        time=ob.TotalDistance/Speed

        ob.Hour=int(time)

        ob.Min = int((time - ob.Hour) \* 60)

        print("Time to reach your destination by car is:" + str(ob.Hour) + " hr " + str(ob.Min) + " min.")

#main start

print("Select Travel Plan:")

print("Enter 1 to travel by Air\nEnter 2 to travel by Road")

#object of travel class defined

persons=Travel()

#taking input in variable and casting into integer

choice=int(input())

#if condition for chossing travel

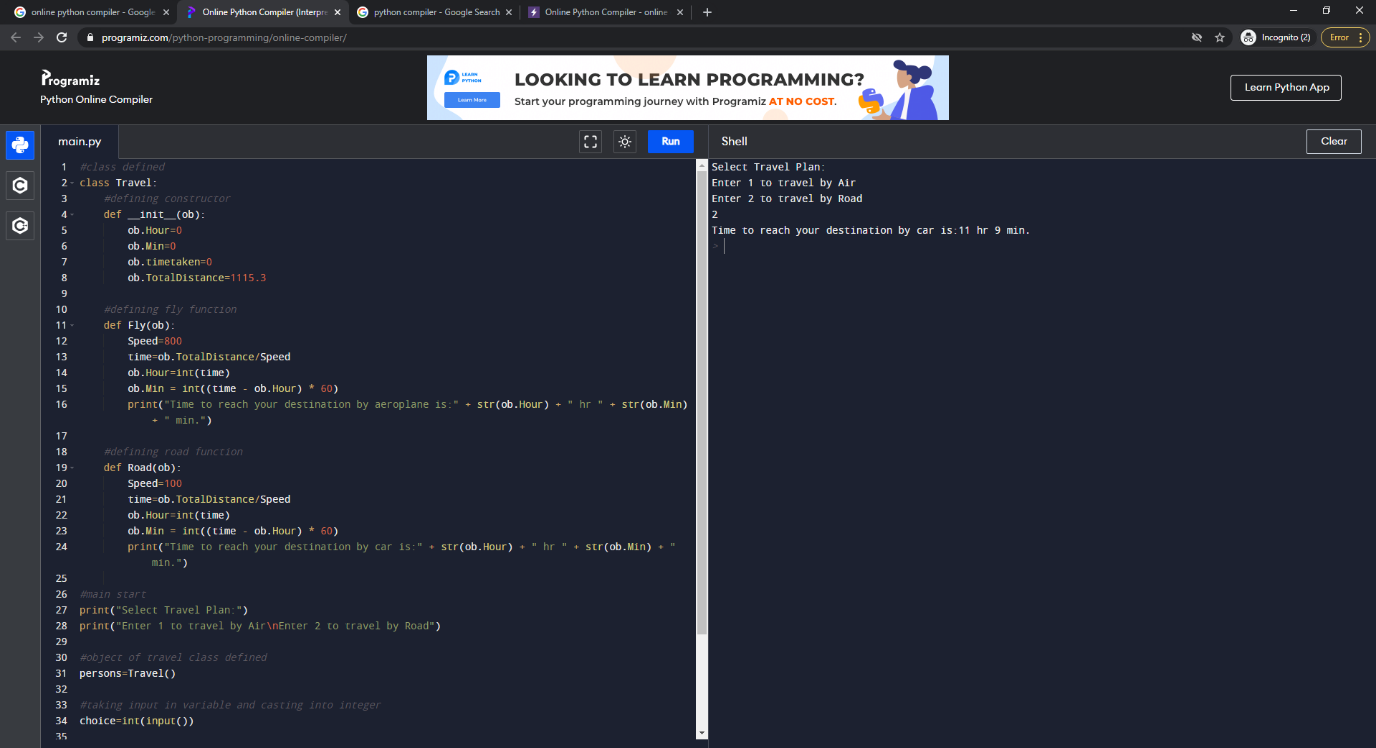
if choice==1:

    persons.Fly()

if choice==2:

    persons.Road()

Screenshot:



# Python Graph (Adjacency matrix):

Solution:

Code:

class Graph:

# Initialize the matrix

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

self.adjMatrix = []

for i in range(size):

self.adjMatrix.append([0 for i in range(size)])

self.size = size

# Add edges

def add\_edge(self, v1, v2):

if v1 == v2:

print(&quot;Same vertex %d and %d&quot; % (v1, v2))

self.adjMatrix[v1][v2] = 1

self.adjMatrix[v2][v1] = 1

# Remove edges

def remove\_edge(self, v1, v2):

if self.adjMatrix[v1][v2] == 0:

print(&quot;No edge between %d and %d&quot; % (v1, v2))

return

self.adjMatrix[v1][v2] = 0

self.adjMatrix[v2][v1] = 0

def \_\_len\_\_(self):

return self.size

# Print the matrix

def print\_matrix(self):

for row in self.adjMatrix:

for val in row:

print(&#39;{:4}&#39;.format(val)),

print

def main():

g = Graph(5)

g.add\_edge(0, 1)

g.add\_edge(0, 2)

g.add\_edge(1, 2)

g.add\_edge(2, 0)

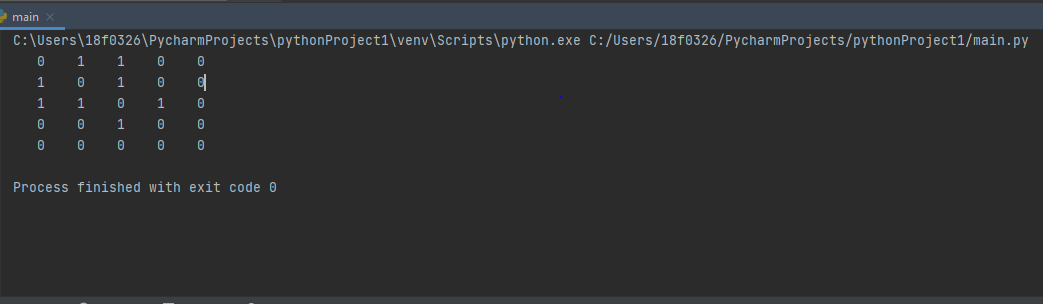
g.add\_edge(2, 3)

g.print\_matrix()

if \_\_name\_\_ == &#39;\_\_main\_\_&#39;:

main()

Screenshot:



# Lab#3 Graph Traverse:

## Task 01 (DFS):

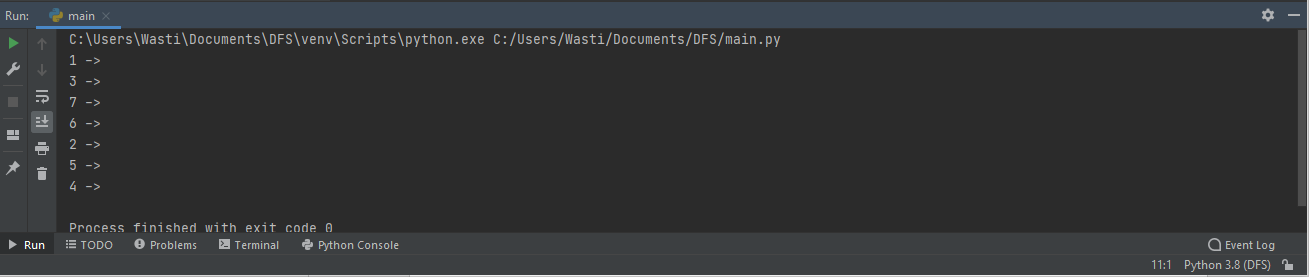
DFS Implementation.

Solution:

Code:

graphs = {  
  '1': ['2', '3'],  
  '2': ['1', '4', '5'],  
  '3': ['1', '6', '7'],  
  '4': ['2'],  
  '5': ['2'],  
  '6': ['3'],  
  '7': ['3']  
}  
  
DFSstarting = '1'  
  
stack = []  
visited = []  
  
def travers(starting,stak,visit, graph):  
    stak.append(starting)  
  
    while stak:  
        node = stak.pop()  
        visit.append(node)  
        for conectedNode in graph[node]:  
            if conectedNode not in visited and conectedNode not in stak:  
                stak.append(conectedNode)  
        print (str(node) + " -> ")  
  
  
  
travers(DFSstarting,stack,visited,graphs)

Screenshot:



## Task 02 (BFS):

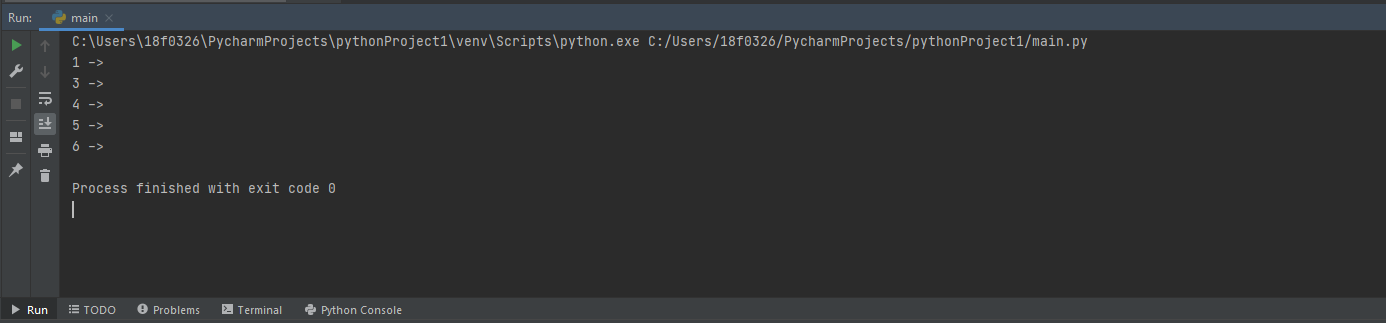
BFS Implementation.

Solution:

Code:

graphs = {  
  '1' : ['3'],  
  '3' : ['1', '4', '5'],  
  '4' : ['3', '6'],  
  '5' : ['3', '6'],  
  '6' : ['4', '5']  
}  
  
BFDstarting = '1'  
  
queue = []  
visited = []  
  
def travers(starting,queu,visit, graph):  
    queu.append(starting)  
  
    while queu:  
        node = queu.pop(0)  
        visit.append(node)  
        for conectedNode in graph[node]:  
            if conectedNode not in visited and conectedNode not in queu:  
                queu.append(conectedNode)  
        print (str(node) + " -> ")  
  
  
  
travers(BFDstarting,queue,visited,graphs)

Screenshot:



# Lab#4 Searches:

## Task 01 (UCS):

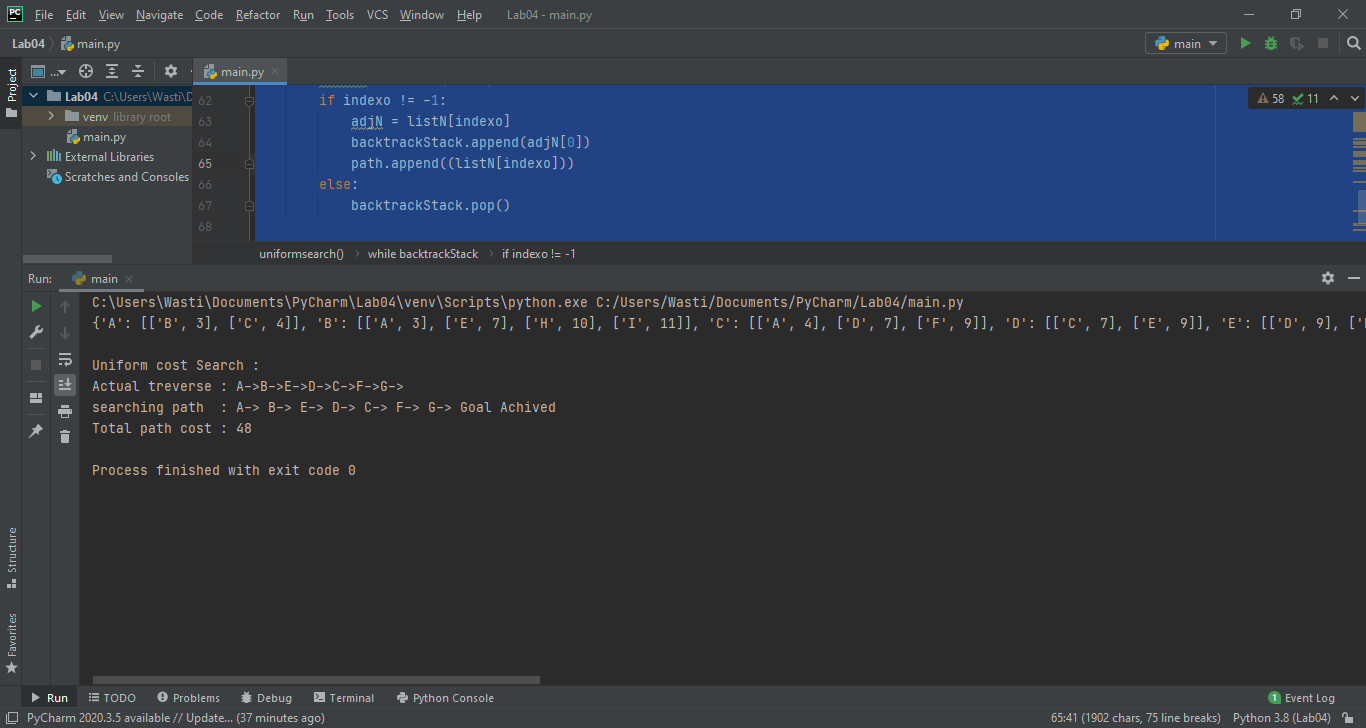
Uniform Cost Search Implementation.

Solution:

Code:

Graph = {  
    "A": [["B",3], ["C",4]],  
    "B": [["A",3], ["E",7], ["H",10], ["I",11]],  
    "C": [["A",4], ["D", 7], ["F",9]],  
    "D": [["C",7], ["E",9]],  
    "E": [["D",9], ["B",7], ["F",11], ["H",13]],  
    "F": [["C",9], ["E",11], ["G",13]],  
    "G": [["F",13], ["H",15], ["K",18]],  
    "H": [["B",10], ["E",13], ["G",15], ["K",19], ["L",20], ["J",18]],  
    "I": [["B",11], ["J",19]],  
    "J": [["H",18], ["I",19]],  
    "K": [["H",19], ["G",18]],  
    "L": [["H",20]]  
}  
  
  
  
print(Graph)  
  
def minlol(lists):  
    minValue = 999  
    index = 0  
    Mindex = -1  
    while index < len(lists):  
        lol = lists[index]  
        if minValue > lol[1] :  
            minValue = lol[1]  
            Mindex = index  
        index +=1  
    return Mindex  
  
def uniformsearch(Graph,startingN,goalN):  
    listN = []  
    backtrackStack = []  
    visited = []  
    path = []  
  
    path.append([startingN,0])  
    pathcost = 0  
    print("Actual treverse :", end=" ")  
  
    backtrackStack.append(startingN)  
    while backtrackStack:  
        listN.clear()  
        Node = backtrackStack[-1]  
        print(Node + "->", end="")  
        visited.append(Node)  
        if Node == goalN:  
  
            print("\nsearching path  :", end=" ")  
            for pathv in path:  
                pathcost = pathv[1] + pathcost  
                print(str(pathv[0]) + "->", end=" ")  
            print("Goal Achived")  
            print("Total path cost : " + str(pathcost))  
            break  
  
        for conectedNode in Graph[Node]:  
            if conectedNode[0] not in visited and conectedNode[0] not in backtrackStack:  
                listN.append(conectedNode)  
        indexo = minlol(listN)  
        if indexo != -1:  
            adjN = listN[indexo]  
            backtrackStack.append(adjN[0])  
            path.append((listN[indexo]))  
        else:  
            backtrackStack.pop()  
  
  
  
print("\nUniform cost Search :")  
uniformsearch(Graph,"A","G")

Screenshots:



## Task 02 (BDS):

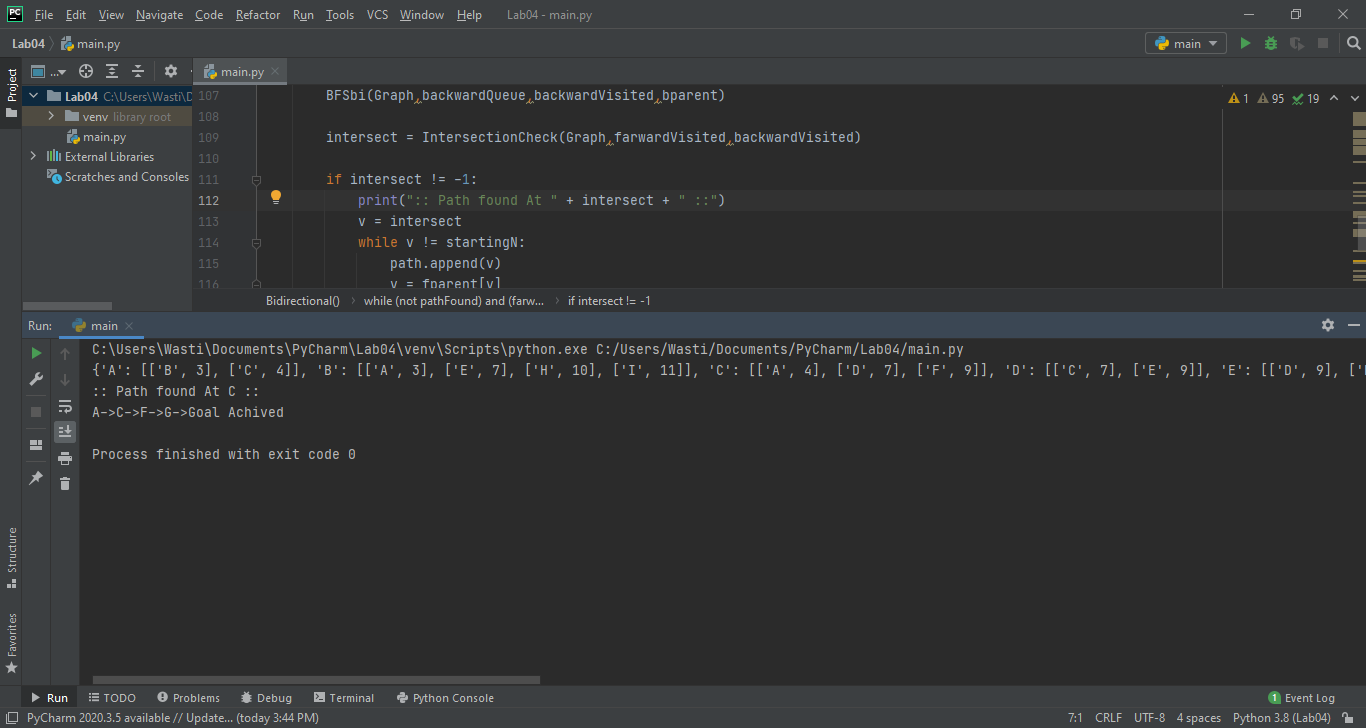
Bi-Directional Search Implementation.

Solution:

Code:

Graph = {  
    "A": [["B",3], ["C",4]],  
    "B": [["A",3], ["E",7], ["H",10], ["I",11]],  
    "C": [["A",4], ["D", 7], ["F",9]],  
    "D": [["C",7], ["E",9]],  
    "E": [["D",9], ["B",7], ["F",11], ["H",13]],  
    "F": [["C",9], ["E",11], ["G",13]],  
    "G": [["F",13], ["H",15], ["K",18]],  
    "H": [["B",10], ["E",13], ["G",15], ["K",19], ["L",20], ["J",18]],  
    "I": [["B",11], ["J",19]],  
    "J": [["H",18], ["I",19]],  
    "K": [["H",19], ["G",18]],  
    "L": [["H",20]]  
}  
  
  
  
print(Graph)  
  
  
def BFSbi(Graph, queue, visited,parent):  
    node = queue.pop(0)  
    visited.append(node)  
    for conected in Graph[node] :  
        if conected[0] not in queue and conected[0] not in visited:  
            queue.append(conected[0])  
            parent[conected[0]] = node  
    return  
  
def IntersectionCheck(Graph, fvisit,bvisit):  
    for i in fvisit:  
        for j in bvisit:  
            if i == j:  
                return i  
    return -1  
  
  
def Bidirectional(Graph, startingN, goalN):  
    farwardQueue = []  
    backwardQueue = []  
    farwardVisited = []  
    backwardVisited = []  
    fparent = {}  
    bparent = {}  
    path = []  
  
    pathFound = False  
  
    farwardQueue.append(startingN)  
    backwardQueue.append(goalN)  
  
  
    while (not pathFound) and (farwardQueue or backwardQueue):  
  
        BFSbi(Graph,farwardQueue,farwardVisited,fparent)  
  
        BFSbi(Graph,backwardQueue,backwardVisited,bparent)  
  
        intersect = IntersectionCheck(Graph,farwardVisited,backwardVisited)  
  
        if intersect != -1:  
            print(":: Path found At " + intersect + " ::")  
            v = intersect  
            while v != startingN:  
                path.append(v)  
                v = fparent[v]  
            path.append(startingN)  
  
            path.reverse()  
  
  
            v = intersect  
            while v != goalN:  
                if v not in path:  
                    path.append(v)  
                v = bparent[v]  
            path.append(goalN)  
  
            for x in range(len(path)):  
                print(path[x] + "->",end="")  
            print("Goal Achived")  
            pathFound = True  
  
  
  
  
  
  
Bidirectional(Graph,"A","G")

Screenshots:



# Lab#5 A\* Search:

## Task 01 (UCS):

Last time we Implemented BFS. Now take the same graph and

implement A\*

Solution:

Code:

#18F-0326 Abdul Salam Wasti BFS updation to A\* Search

from collections import defaultdict

class Graph:

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

self.graph = defaultdict(list)

def addEdge(self, u, v):

self.graph[u].append(v)

def printGraph(self):

print("Given Graph",self.graph)

# Function to print a BFS of graph

def BFS(self, s, e):

print("This is our starting node", s)

print("This is our ending node",e)

visited=[]

queue = []

path=[]

queue.append(s)

visited.append(s)

while queue:

nodetoselect=g.findHeuristic(queue,huristic)

s=nodetoselect

path.append(s)

print("Node to Select",s)

indexofselectedvalue = queue.index(s)

queue.pop(indexofselectedvalue)

if s==e:

break

for i in self.graph[s]:

if i not in visited:

queue.append(i)

visited.append(i)

print("this is our final path",path)

g.findactualpathvalue(path)

def AStarSearch(self, s, e):

print("This is our starting node", s)

print("This is our ending node",e)

visited=[]

queue = []

path=[]

parentChild = [] #need parent child track to get path value

c = s

parentChild.append(s)

parentChild.append(c)

queue.append(parentChild)

visited.append(s)

while queue:

nodetoselect=g.findHeuristicAndPath(queue,huristic,actualvalues)

c=nodetoselect

path.append(c)

print("Node to Select",c)

parentChild.clear()

parentChild.append(s)

parentChild.append(c)

indexofselectedvalue = queue.index(parentChild)

queue.pop(indexofselectedvalue)

if c==e:

break

s = c

for i in self.graph[s]:

if i not in visited:

parentChild = []

parentChild.append(s)

parentChild.append(i)

queue.append(parentChild)

visited.append(i)

print("this is our final path",path)

g.findactualpathvalue(path)

def findactualpathvalue(self,mypath):

actualvaluequue=[]

cost=[]

for i in range(len(mypath)-1):

a=str(mypath[i])

b=str(mypath[i+1])

valuetoappend=(a+b)

print('from ', a,'to', b, end=' ')

actualvaluequue.append(valuetoappend)

print("this is the path to go",actualvaluequue)

for i in actualvaluequue:

if i in actualvalues.keys():

cost.append(actualvalues.get(i))

print('This is the actual cost', sum(cost))

def findHeuristic(self,pqueu,dich):

keyslist=[]

valueslist=[]

for i in pqueu:

if i in dich.keys():

keyslist.append(i)

valueslist.append(dich.get(i))

indexofvalue=valueslist.index(min(valueslist))

keyofminvalue=keyslist[indexofvalue]

print('Keys list:', keyslist)

print("Value list", valueslist)

return keyofminvalue

def findHeuristicAndPath(self,pqueu,dich, actualPath):

#function to compute path value + heuristic value

keyslist=[]

valueslist=[]

for i in pqueu:

if i[1] in dich.keys():

keyslist.append(i[1])

pathCost = 0

if actualPath.get( str(i[0]+i[1]) ): #getting pathvalue by selected node and parent node

pathCost = actualPath[str(i[0]+i[1])]

valueslist.append(dich.get(i[1]) + pathCost)

indexofvalue=valueslist.index(min(valueslist))

keyofminvalue=keyslist[indexofvalue]

print('Keys list:', keyslist)

print("Value list", valueslist)

return keyofminvalue

# Driver code

g = Graph()

g.addEdge('A', 'B')

g.addEdge('A', 'C')

g.addEdge('A', 'D')

g.addEdge('B', 'E')

g.addEdge('C', 'E')

g.addEdge('C', 'F')

g.addEdge('D', 'F')

g.addEdge('E', 'H')

g.addEdge('F', 'G')

g.addEdge('H', 'G')

huristic=\

{'A': 40,

'B':32,

'C':25,

'D':35,

'E':19,

'F':17,

'H':10,

'G':0

}

actualvalues=\

{'AB':11,

'AC':14,

'AD':7,

'BE':15,

'CE':8,

'CF':10,

'DF':25,

'FG':20,

'EH':9,

'HG':10,

}

g.printGraph()

print("heuristic Values",huristic)

print("Actual Cost of the graph", actualvalues)

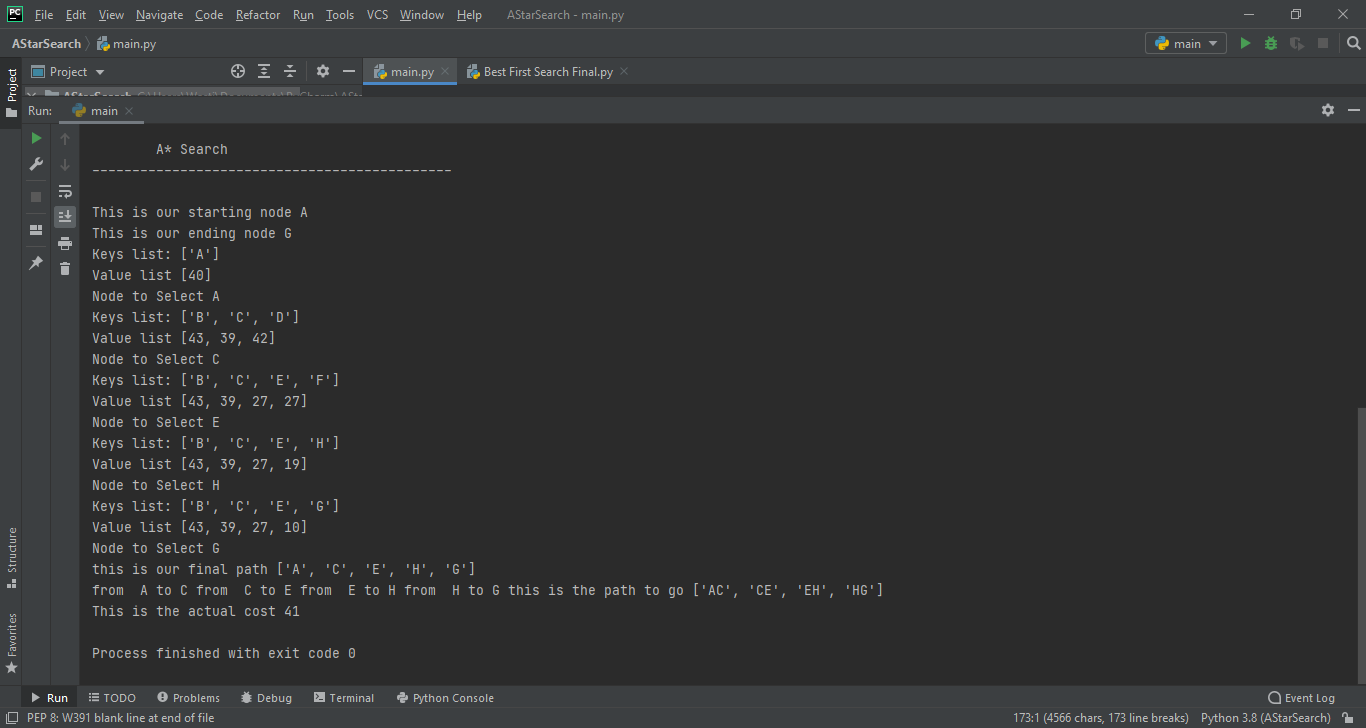
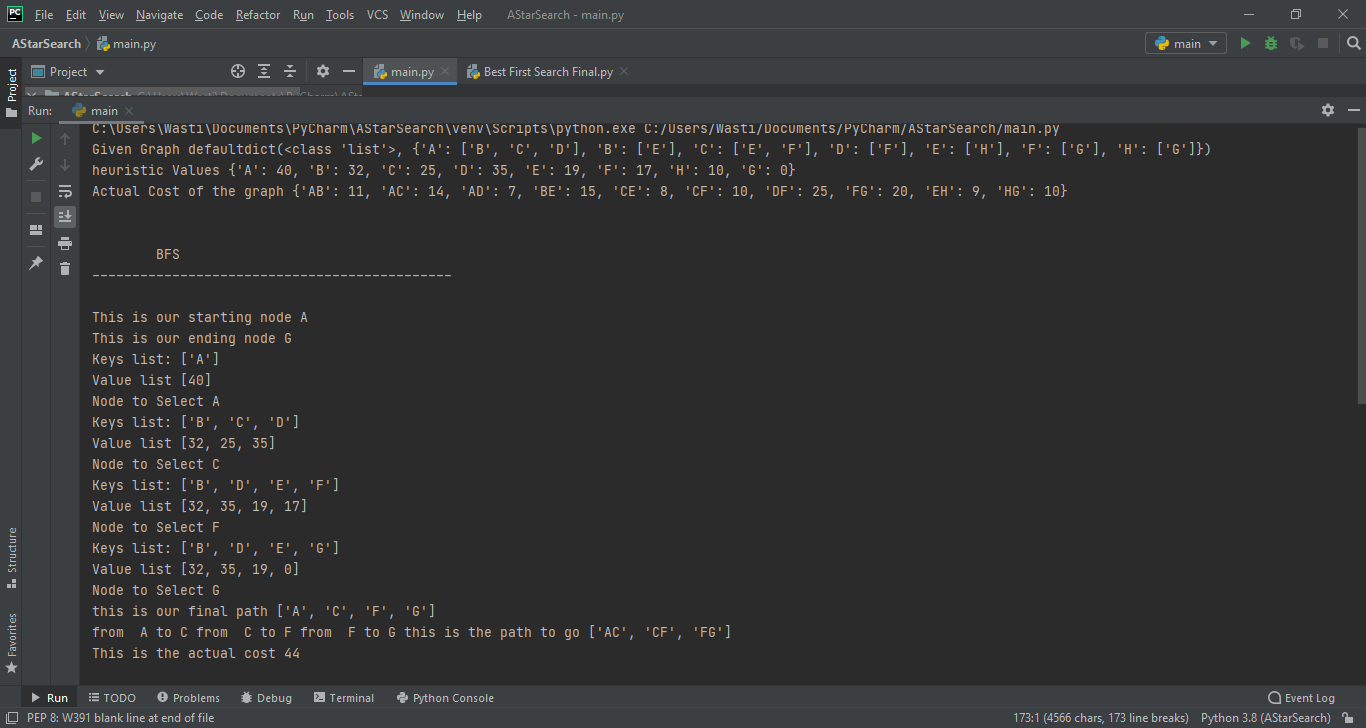
print("\n\n\t\tBFS\n---------------------------------------------\n")

g.BFS('A','G')

print("\n\n\t\tA\* Search\n---------------------------------------------\n")

g.AStarSearch('A','G')

Screenshots:

# Lab#6 Simulated annealing:

## Task 01 :

Update the code to make it simulated annealing

Solution:

Code:

from collections import defaultdict

class Graph:

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

self.graph = defaultdict(list)

def addEdge(self, u, v):

self.graph[u].append(v)

def printGraph(self):

print(self.graph)

def findactualpathvalue(self,mypath):

actualvaluequue=[]

cost=[]

for i in range(len(mypath)-1):

a=str(mypath[i])

b=str(mypath[i+1])

valuetoappend=(a+b)

actualvaluequue.append(valuetoappend)

actualvaluequue.append(str(mypath[-1])+str(mypath[0]))

for i in actualvaluequue:

if i in actualvalues.keys():

cost.append(actualvalues.get(i))

cost=sum(cost)

if mypath==['A', 'C', 'B', 'D']:

print('here is update')

cost=45

return cost

def generateallsol(self):

cities=list(g.graph.keys())

print('these are cities',cities)

allposiblesol=[]

ri = 0

for i in range(0, 3):

for j in range(0, 2):

# print (i,j)

temp = cities[3 - j]

cities[3 - j] = cities[3 - j - 1]

cities[3 - j - 1] = temp

print('route', cities)

allposiblesol.append(['A', 'B', 'C', 'D'])

for ind in range(len(cities)):

allposiblesol[ri][ind] = cities[ind]

ri += 1

return allposiblesol

def simulatedaAnealing(self):

######Update Code here

bestCostIndex = 0

bestCostList = []

allposibleroutes=g.generateallsol()

print('this is our solution space', allposibleroutes)

currentpath = allposibleroutes[0]

currentCost=g.findactualpathvalue(allposibleroutes[0])

bestCostList.append(currentCost)

print('current path ', currentpath,'current cost ', currentCost)

print('Simulated Anealing...........')

i = 1

while i < len(allposibleroutes):

nextcost = g.findactualpathvalue(allposibleroutes[i])

print('cost for the Route nbr', i, allposibleroutes[i], 'is', nextcost)

bestCostList.append(nextcost)

if nextcost < currentCost or nextcost < bestCostList[bestCostIndex]:

bestCostIndex = i

#break

#if nextcost <= currentCost:

currentCost = nextcost

currentpath = allposibleroutes[i]

i += 1

print('---------------------------------------------------')

print('current path', currentpath)

print('with cost of', currentCost)

print('\nBest path', allposibleroutes[bestCostIndex])

print('with min cost of', bestCostList[bestCostIndex])

# Driver code

g = Graph()

g.addEdge('A', 'B')

g.addEdge('A', 'C')

g.addEdge('A', 'D')

g.addEdge('B', 'A')

g.addEdge('B', 'C')

g.addEdge('B', 'D')

g.addEdge('C','A')

g.addEdge('C', 'B')

g.addEdge('C', 'D')

g.addEdge('D', 'A')

g.addEdge('D', 'B')

g.addEdge('D', 'C')

actualvalues=\

{'AB':25,

'AD':15,

'BD':45,

'BC':10,

'CD':5,

'AC':10,

'BA':25,

'DA':15,

'DB':45,

'CB':10,

'DC':5,

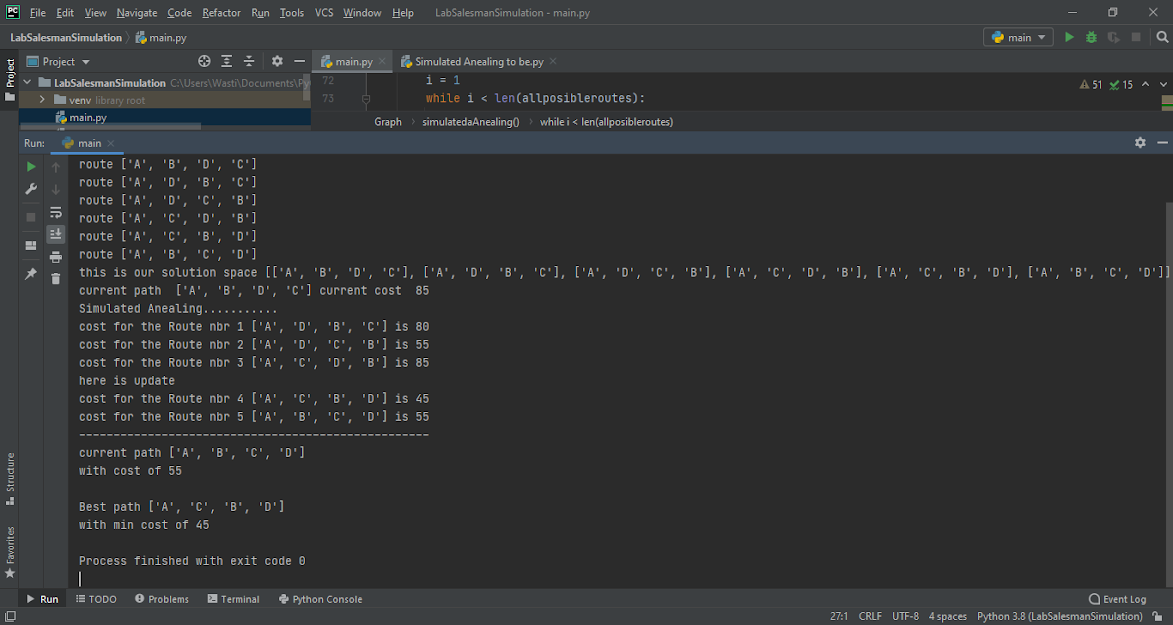
'CA':10,

}

g.printGraph()

g.simulatedaAnealing()

Screenshot:



# Lab#7 MinMax with pruning:

## Task 01 :

Implement Alpha beta pruning in python

Solution:

Code:

maxlist=[]

minlist=[]

def minimaxalgo(boardvalues,turn, alpha, beta):

if len(boardvalues)==1:

print("Optimal value",boardvalues)

elif turn==True:

#MAx turn

maxlist=boardvalues.copy()

boardvalues.clear()

print("this is max function")

i=0

while i<len(maxlist):

eval = max(maxlist[i], maxlist[i + 1])

alpha = max(alpha, eval)

#print("Alpha:",alpha)

boardvalues.append(eval)

i+=2

print('max function return',boardvalues)

minimaxalgo(boardvalues,False, alpha, beta)

elif turn==False:

#MIn turn

minlist=boardvalues.copy()

boardvalues.clear()

print('this is min Function')

i = 0

while i < len(minlist):

eval = min(minlist[i], minlist[i + 1])

beta = min(beta, eval)

#print("Beta:", beta)

boardvalues.append(eval)

i += 2

print('min function return',boardvalues)

minimaxalgo(boardvalues,True, alpha, beta)

else:

print("non veg")

def minimaxAlphaBetaalgo(boardvalues,turn, alpha, beta, index, depth):

if depth==3:

return boardvalues[index]

elif turn==True:

#MAx turn

val = -999

for x in range(2):

print(index \* 2 + x)

valueGet = minimaxAlphaBetaalgo(boardvalues,False, alpha, beta, index \* 2 + x, depth+1)

val = max(val, valueGet)

alpha = max(alpha, val)

if beta <= alpha:

break

return val

elif turn==False:

#MIn turn

val = 999

for x in range(2):

print(index \* 2 + x)

valueGet = minimaxAlphaBetaalgo(boardvalues, True, alpha, beta, index \* 2 + x, depth + 1)

val = min(val, valueGet)

beta = min(beta, val)

if beta <= alpha:

break

return val

else:

print("non veg")

#driver code

boardvalues=[4,6,2,10,14,6,22,21]

print('these are board values', boardvalues)

v = minimaxAlphaBetaalgo(boardvalues,True, -999,999, 0,0)

print("Optimal value by alpha-beta puring is: [",v,"]")

# Lab#8 CSP:

## Task 01 :

Find all (a,b) where a ∈ {1,2,3} and b ∈ {4,5,3} where ‘a’ should not be equal to b .

Solution:

Code:

#18F-0326 Abdul Salam Wasti

from constraint import \*

def q1sol():

print("\nQuestion 01")

problem = Problem()

problem.addVariable("a", [1, 2, 3])

problem.addVariable("b", [4, 5, 6])

sol = problem.getSolution()

print("without contraints", sol)

problem.addConstraint(lambda a, b: a != b, ("a", "b"))

sol = problem.getSolution()

print("with contraints", sol)

print("----------------------------------------------\n")

## Task 02:

Find all (a,b) where a ∈ {2,4,6,8,10,12} and b ∈ {3,6,12,15} where ‘a’ should be equal to ‘b’ and ‘a’ should be equal to 6.

Solution:

Code:

#18F-0326 Abdul Salam Wasti

from constraint import \*

def q2sol():

print("\nQuestion 02")

problem = Problem()

problem.addVariable("a", [2, 4, 6, 8, 10, 12])

problem.addVariable("b", [3, 6, 12, 15])

sol = problem.getSolution()

print("without contraints", sol)

problem.addConstraint(lambda a, b: a == b and a == 6, ("a", "b"))

sol = problem.getSolution()

print("with contraints", sol)

print("----------------------------------------------\n")

Main:

#18F-0326 Abdul Salam Wasti

from q1 import \*

from q2 import \*

q1sol()

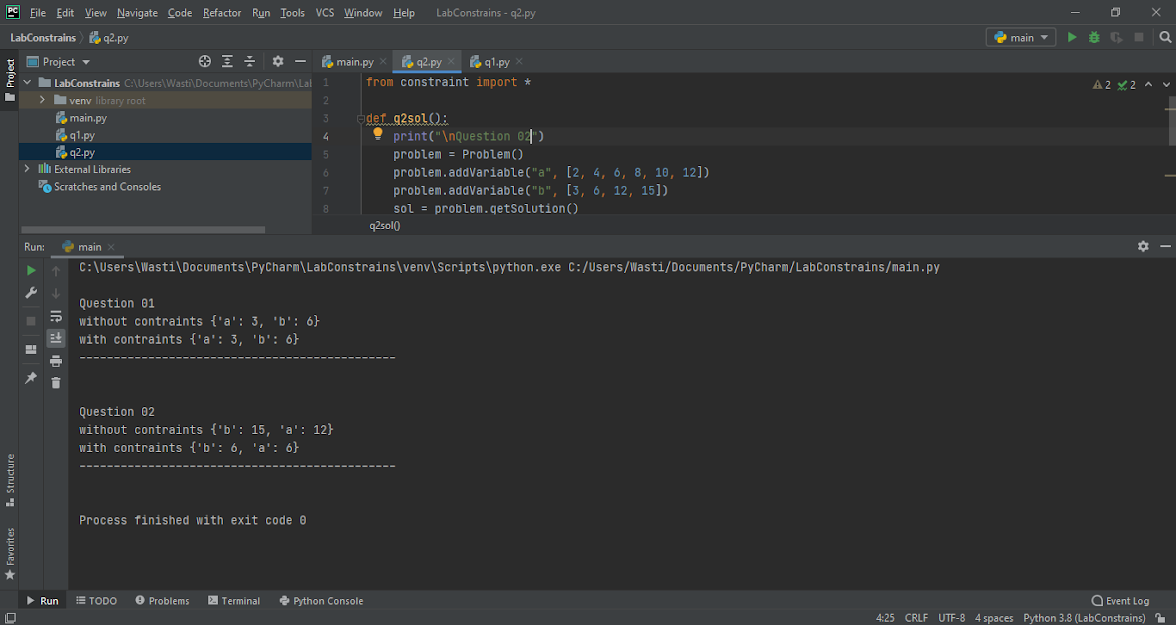
q2sol()

# Press the green button in the gutter to run the script.

#if \_\_name\_\_ == '\_\_main\_\_':

# See PyCharm help at https://www.jetbrains.com/help/pycharm/

Screenshot:

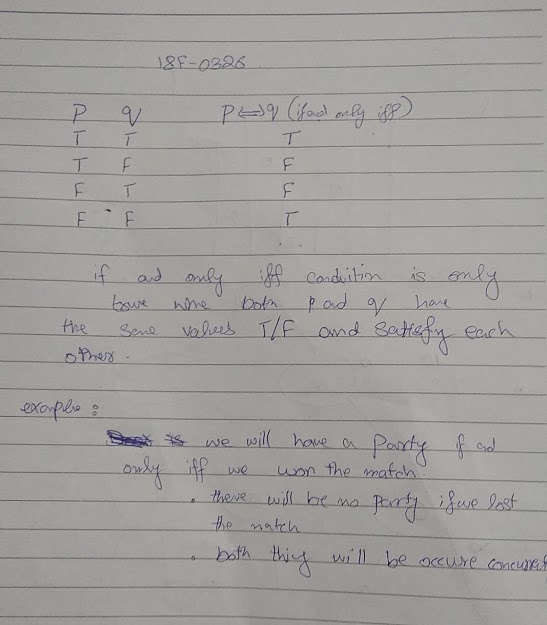


# Lab#9 Prepositional Logic:

## Task 01:

make truth table, write your own statements and explain it. try to use good words and sentences.

Solution:



# Project 01 K-mean, K-medoid:

## Task 01 K-mean:

Implement K-mean.

Solution:

Code:

# 18F-0183, 18F-0137, 18F-0326. Section 6B

import pandas as pd

import random

import math

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

# Reading the coordinates from text file

df = pd.read\_csv('cluster\_validation\_data.txt', sep=",", header=None)

df.head()

# normalize data

X = df.values

sc = StandardScaler()

sc.fit(X)

X = sc.transform(X)

# class for storing x and y coordinates

class coordinates:

def \_\_init\_\_(self, xcor, ycor):

self.x = xcor

self.y = ycor

def kmeans(X, k=3, max\_iterations=100):

"""

X: multidimensional data

k: number of clusters

max\_iterations: number of repetitions before clusters are established

"""

file = open("cluster\_validation\_data.txt", "r")

totalLines = 0

# reading file again to count total number of lines

data = file.read()

CoList = data.split("\n")

for index in CoList:

if index:

totalLines += 1

# generate k random centroids

c1 = random.randint(0,totalLines-1)

c2 = random.randint(0,totalLines-1)

c3 = random.randint(0,totalLines-1)

# compare the centroids so that each centroid must be unique

while c2 == c1 or c2 == c3:

c2 = random.randint(0, totalLines-1)

while c3 == c1 or c2 == c3:

c3 = random.randint(0, totalLines-1)

# create k number of cluster lists

cluster1 = list()

cluster2 = list()

cluster3 = list()

# store the initially selected centroids in their respective lists as class objects

cluster1.append(coordinates(X[c1][0], X[c1][1]))

cluster2.append(coordinates(X[c2][0], X[c2][1]))

cluster3.append(coordinates(X[c3][0], X[c3][1]))

"""

Establishing all clusters by calculating distance of each point from all centroids using Euclidean formula.

Then assign that point to the cluster to whose centroid it has minimum distance

"""

i = 0

while i < totalLines:

# skip the point which is either of the centroids

if i != c1 and i != c2 and i != c3:

distanceFromC1 = math.sqrt( pow((X[c1][0]-X[i][0]),2) + pow((X[c1][1]-X[i][1]),2) )

distanceFromC2 = math.sqrt( pow((X[c2][0]-X[i][0]),2) + pow((X[c2][1]-X[i][1]),2) )

distanceFromC3 = math.sqrt( pow((X[c3][0]-X[i][0]),2) + pow((X[c3][1]-X[i][1]),2) )

minDistance = min(distanceFromC1, distanceFromC2, distanceFromC3)

if minDistance == distanceFromC1:

cluster1.append(coordinates(X[i][0], X[i][1]))

if minDistance == distanceFromC2:

cluster2.append(coordinates(X[i][0], X[i][1]))

if minDistance == distanceFromC3:

cluster3.append(coordinates(X[i][0], X[i][1]))

i = i + 1

max\_iterations = max\_iterations - 1

flag1 = flag2 = flag3 = True

#continue the loop until all updated clusters become same as they were before or the maximum iterations reaches 100

while (flag1 != False and flag2 != False and flag3 != False) or max\_iterations < 0:

"""

if cluster1 is not same updatedCluster1 then we will update it by taking mean.

we will take average of x and y coordinates of all points in that cluster. The resultant will be our new

centeroid

"""

if flag1 != False:

i = 0

sumofXcord = 0

sumofYcord = 0

while i < len(cluster1): # updating centroid 1

sumofXcord = sumofXcord + cluster1[i].x

sumofYcord = sumofYcord + cluster1[i].y

i=i+1

updatedC1 = coordinates(sumofXcord/len(cluster1), sumofYcord/len(cluster1))

"""

if cluster2 is not same updatedCluster2 then we will update it by taking mean.

we will take average of x and y coordinates of all points in that cluster. The resultant will be our new

centeroid

"""

if flag2 != False:

i = 0

sumofXcord = 0

sumofYcord = 0

while i < len(cluster2): # updating centroid 2

sumofXcord = sumofXcord + cluster2[i].x

sumofYcord = sumofYcord + cluster2[i].y

i=i+1

updatedC2 = coordinates(sumofXcord/len(cluster2), sumofYcord/len(cluster2))

"""

if cluster3 is not same updatedCluster3 then we will update it by taking mean.

we will take average of x and y coordinates of all points in that cluster. The resultant will be our new

centeroid

"""

if flag3 != False:

i = 0

sumofXcord = 0

sumofYcord = 0

while i < len(cluster3): # updating centroid 3

sumofXcord = sumofXcord + cluster3[i].x

sumofYcord = sumofYcord + cluster3[i].y

i=i+1

updatedC3 = coordinates(sumofXcord/len(cluster3), sumofYcord/len(cluster3))

# creating new lists for updated clusters

updatedCluster1 = list()

updatedCluster2 = list()

updatedCluster3 = list()

"""

Establishing all clusters by calculating distance of each point from all centroids using Euclidean formula.

Then assign that point to the cluster to whose centroid it has minimum distance

"""

i = 0

while i < totalLines:

# skip the point which is either of the centroids

if (X[i][0] != updatedC1.x or X[i][1] != updatedC1.y) and (X[i][0] != updatedC2.x or X[i][1] != updatedC2.y) and (X[i][0] != updatedC3.x or X[i][1] != updatedC3.y):

distanceFromC1 = math.sqrt( pow((updatedC1.x-X[i][0]),2) + pow((updatedC1.y-X[i][1]),2) )

distanceFromC2 = math.sqrt( pow((updatedC2.x-X[i][0]),2) + pow((updatedC2.y-X[i][1]),2) )

distanceFromC3 = math.sqrt( pow((updatedC3.x-X[i][0]),2) + pow((updatedC3.y-X[i][1]),2) )

minDistance = min(distanceFromC1, distanceFromC2, distanceFromC3)

if minDistance == distanceFromC1 and flag1 != False:

updatedCluster1.append(coordinates(X[i][0], X[i][1]))

if minDistance == distanceFromC2 and flag2 != False:

updatedCluster2.append(coordinates(X[i][0], X[i][1]))

if minDistance == distanceFromC3 and flag3 != False:

updatedCluster3.append(coordinates(X[i][0], X[i][1]))

i = i + 1

"""

now we have to compare the old cluster and newly formed updatedCluster. But they can't be compared

together the order of coordinate objects might be different. So, first change them into list of integer

points and then compare them after sorting the coordinates. If they are same then that cluster will not be

updated in next iteration.

"""

list1 = []

list2 = []

for i in cluster1:

list1.append([i.x,i.y])

for i in updatedCluster1:

list2.append([i.x,i.y])

if sorted(list1) == sorted(list2) and flag2 != False:

flag1=False

else:

cluster1=updatedCluster1

list1 = []

list2 = []

for i in cluster2:

list1.append([i.x,i.y])

for i in updatedCluster2:

list2.append([i.x,i.y])

if sorted(list1) == sorted(list2) and flag2 != False:

flag2=False

else:

cluster2=updatedCluster2

list1 = []

list2 = []

for i in cluster3:

list1.append([i.x,i.y])

for i in updatedCluster3:

list2.append([i.x,i.y])

if sorted(list1) == sorted(list2) and flag3 != False:

flag3=False

else:

cluster3=updatedCluster3

i = 0

max\_iterations = max\_iterations - 1

# make P such that it will be a list of clusters. On each index there will be a different cluster in integer

# form because currently updatedCluster1, 2 and 3 are having elements which are objects of coordinates class.

P = []

a = []

b = []

c = []

for i in updatedCluster1:

a.append([i.x, i.y])

for i in updatedCluster2:

b.append([i.x, i.y])

for i in updatedCluster3:

c.append([i.x, i.y])

# a, b and c are separate clusters and P will be list of all clusters

P.append(a)

P.append(b)

P.append(c)

return P

P = kmeans(X)

# denormalize data

X = sc.inverse\_transform(X)

# using matplotlib of plot the graph

color = ['c', 'm', 'y', 'k', 'w', 'b', 'r', 'g']

iterator = 0

plt.figure(figsize=(7, 5))

for i in P: # get a cluster from list of clusters

for j in i: # iterate over coordinates of points in that cluster

plt.plot(j[0], j[1], color[iterator ] + 'o')

iterator = iterator + 1

plt.show()

Screenshot: Graphical user interface, chart, scatter chart

Description automatically generated

## Task 02 K-medoid:

Implement K-medoid.

Solution:

Code:

# 18F-0183, 18F-0137, 18F-0326. Section 6B

import pandas as pd

import random

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

# Reading the coordinates from text file

df = pd.read\_csv('cluster\_validation\_data.txt', sep=",", header=None)

df.head()

# normalize data

X = df.values

sc = StandardScaler()

sc.fit(X)

X = sc.transform(X)

# class for storing x and y coordinates

class coordinates:

def \_\_init\_\_(self, xcor, ycor):

self.x = xcor

self.y = ycor

def kmedoid(X, k=3, max\_iterations=100):

"""

X: multidimensional data

k: number of clusters

max\_iterations: number of repetitions before clusters are established

"""

file = open("cluster\_validation\_data.txt", "r")

totalLines = 0

# reading file again to count total number of lines

data = file.read()

CoList = data.split("\n")

for index in CoList:

if index:

totalLines += 1

# generate k random centroids

c1 = random.randint(0, totalLines - 1)

c2 = random.randint(0, totalLines - 1)

c3 = random.randint(0, totalLines - 1)

# compare the centroids so that each centroid must be unique

while (c2 == c1 or c2 == c3):

c2 = random.randint(0, totalLines - 1)

while (c3 == c1 or c2 == c3):

c3 = random.randint(0, totalLines - 1)

# visited list to keep check of the points already used as centroid/medoid

visited = [False] \* totalLines

visited[c1] = True

visited[c2] = True

visited[c3] = True

# create k number of cluster lists

cluster1 = list()

cluster2 = list()

cluster3 = list()

# store the initially selected centroids in their respective lists as class objects

cluster1.append(coordinates(X[c1][0], X[c1][1]))

cluster2.append(coordinates(X[c2][0], X[c2][1]))

cluster3.append(coordinates(X[c3][0], X[c3][1]))

""""

Establishing all clusters by calculating distance of each point from all centroids using distance formula.

Then assign that point to the cluster to whose centroid it has minimum distance

"""

i = 0

while i < totalLines:

if i != c1 and i != c2 and i != c3:

distanceFromC1 = abs((X[c1][0] - X[i][0])) + abs((X[c1][1] - X[i][1]))

distanceFromC2 = abs((X[c2][0] - X[i][0])) + abs((X[c2][1] - X[i][1]))

distanceFromC3 = abs((X[c3][0] - X[i][0])) + abs((X[c3][1] - X[i][1]))

minDistance = min(distanceFromC1, distanceFromC2, distanceFromC3)

if minDistance == distanceFromC1:

cluster1.append(coordinates(X[i][0], X[i][1]))

if minDistance == distanceFromC2:

cluster2.append(coordinates(X[i][0], X[i][1]))

if minDistance == distanceFromC3:

cluster3.append(coordinates(X[i][0], X[i][1]))

i = i + 1

max\_iterations = max\_iterations - 1

costDifferenceFlag = True

i = 0

oldCost = 0

# calculating cost of cluster 1 by calculating distance of its centroid to all the points in its cluster.

while i < len(cluster1):

if X[i][0] != X[c1][0] or X[i][1] != X[c1][1]:

oldCost = oldCost + abs((X[c1][0] - X[i][0]) + (X[c1][1] - X[i][1]))

i = i + 1

i = 0

# calculating cost of cluster 2 by calculating distance of its centroid to all the points in its cluster.

while i < len(cluster2):

if X[i][0] != X[c2][0] or X[i][1] != X[c2][1]:

oldCost = oldCost + abs((X[c2][0] - X[i][0]) + (X[c2][1] - X[i][1]))

i = i + 1

i = 0

# calculating cost of cluster 3 by calculating distance of its centroid to all the points in its cluster.

while i < len(cluster3):

if X[i][0] != X[c3][0] or X[i][1] != X[c3][1]:

oldCost = oldCost + abs((X[c3][0] - X[i][0]) + (X[c3][1] - X[i][1]))

i = i + 1

""""

continue this loop until the difference of total cost of old clusters and new cluster is greater than zero

or maximum iterations reaches 100.

"""

while costDifferenceFlag != False or max\_iterations < 0:

# Generate a new centroid and swap it with old centroid and check if it isn't used before

newCentroid = random.randint(0, totalLines - 1)

while visited[newCentroid] == True:

newCentroid = random.randint(0, totalLines - 1)

visited[newCentroid] = True

# Take new cluster lists and initialize them with updated centroids

updatedCluster1 = list()

updatedCluster2 = list()

updatedCluster3 = list()

updatedCluster1.append(coordinates(X[c1][0], X[c1][1]))

updatedCluster2.append(coordinates(X[c2][0], X[c2][1]))

updatedCluster3.append(coordinates(X[newCentroid][0], X[newCentroid][1]))

i = 0

""""

Establishing all clusters by calculating distance of each point from all centroids using distance formula.

Then assign that point to the cluster to whose centroid it has minimum distance

"""

while i < totalLines:

if i != c1 and i != c2 and i != newCentroid:

distanceFromC1 = abs(X[c1][0] - X[i][0]) + abs(X[c1][1] - X[i][1])

distanceFromC2 = abs(X[c2][0] - X[i][0]) + abs(X[c2][1] - X[i][1])

distanceFromNewCentroid = abs(X[newCentroid][0] - X[i][0]) + abs(X[newCentroid][1] - X[i][1])

minDistance = min(distanceFromC1, distanceFromC2, distanceFromNewCentroid)

if minDistance == distanceFromC1:

updatedCluster1.append(coordinates(X[i][0], X[i][1]))

if minDistance == distanceFromC2:

updatedCluster2.append(coordinates(X[i][0], X[i][1]))

if minDistance == distanceFromNewCentroid:

updatedCluster3.append(coordinates(X[i][0], X[i][1]))

i = i + 1

# calculating total new cost of all cluster by calculating distance of their respective centroids to all the

# points in their cluster.

i = 0

newCost = 0

for j in updatedCluster1:

if j.x != X[c1][0] or j.y != X[c1][1]:

newCost = newCost + abs(X[c1][0] - j.x) + abs(X[c1][1] - j.y)

i = i + 1

i = 0

newCost = 0

for j in updatedCluster2:

if j.x != X[c2][0] or j.y != X[c2][1]:

newCost = newCost + abs(X[c2][0] - j.x) + abs(X[c2][1] - j.y)

i = i + 1

i = 0

for j in updatedCluster3:

if j.x != X[newCentroid][0] or j.y != X[newCentroid][1]:

newCost = newCost + abs(X[newCentroid][0] - j.x) + abs(X[newCentroid][1] - j.y)

i = i + 1

# if the difference of old and new cost is greater than zero then keep that swap of centroid.

# otherwise break the loop

if newCost - oldCost > 0:

costDifferenceFlag = False

else:

oldCost = newCost

i = 0

max\_iterations = max\_iterations - 1

# make P such that it will be a list of clusters. On each index there will be a different cluster in integer

# form because currently updatedCluster1, 2 and 3 are having elements which are objects of coordinates class.

P = []

a = []

b = []

c = []

for i in updatedCluster1:

a.append([i.x, i.y])

for i in updatedCluster2:

b.append([i.x, i.y])

for i in updatedCluster3:

c.append([i.x, i.y])

# a, b and c are separate clusters and P will be list of all clusters

P.append(a)

P.append(b)

P.append(c)

return P

P = kmedoid(X)

# denormalize data

X = sc.inverse\_transform(X)

# using matplotlib of plot the graph

color = ['r', 'y', 'b', 'c', 'g', 'k', 'm']

iterator = 0

plt.figure(figsize=(7, 5))

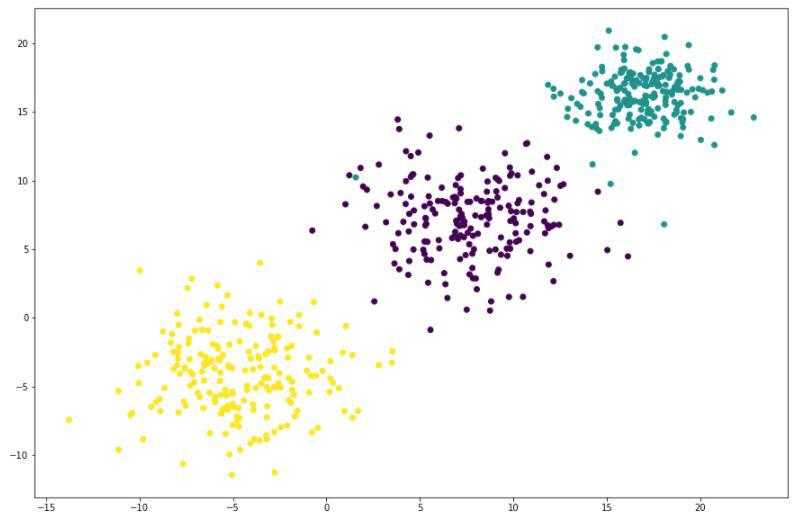
for i in P: # get a cluster from list of clusters

for j in i: # iterate over coordinates of points in that cluster

plt.plot(j[0], j[1], color[iterator] + 'o')

iterator = iterator + 1

plt.show()

Screenshot:

# Project 02 Artificial Neural Network:

## Task 01 ANN:

You have already studied different learning rules such as:

* Perceptron Training Rule
* Delta Rule
* Gradient Descent Rule

You have already established a simple neuron for AND operation. Now you may extend

the capabilities of OR, XOR functionality for your already developed perceptron.

Solution:

Code 1:

import numpy as np

operator = 'or'

attributes = np.array([[0, 0], [0, 1],[1, 0], [1, 1]])

if operator == 'and':

labels = np.array([0, 0, 0, 1])

#labels = np.array([-1, -1, -1, 1])

elif operator == 'or':

labels = np.array([0, 1, 1, 1])

elif operator == 'xor':

labels = np.array([0,1, 1, 0])

elif operator == 'nand':

labels = np.array([1,1, 1, 0])

Initializing the weights (parameters) w = [0.9, 0.9]

bias = w[0]

delcaring hyperparameters alpha = 0.005

epochs = 1000 threshold = 0.5

for i in range (0, epochs):

print("epoch", i+1)

for j in range(len(attributes)):

actual = labels [j]

sum = attributes[j][0] \* w[0] +attributes[j][1]\*w [1] if sum > threshold:

predicted = 1

else:

predicted = 0

delta = actual - predicted

for k in range (0, 2):

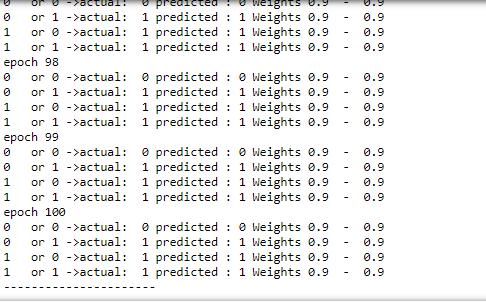
w[k] = w[k] + delta\*alpha

print(attributes[j][0]," ",operator, attributes[j][1], "-

>actual: ", actual,"predicted :",predicted, "Weights", w[0]," - ",w[1])

print ("----------------------")

Screenshot 1:



Code 2:

for x in range(0, epochs):

print ("Epochs ", x+1)

for j in range(0,len(attributes)):

actual = labels[j]

yin = (attributes[j][0]\*w[0]) + (attributes[j][1]\*w[1]) + bias

yin = (attributes[j][0]\*w[0]) + (attributes[j][1]\*w[1])

delta = actual - yin

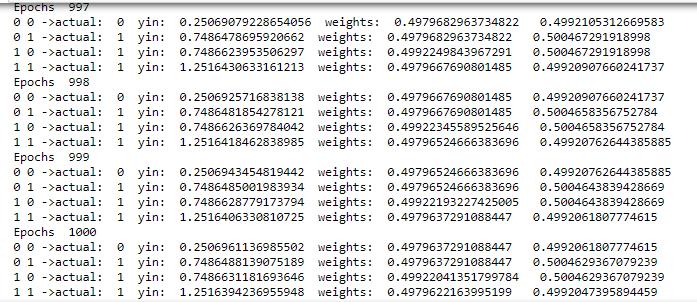
bias = bias + (alpha \* delta)

for k in range(0, 2):

w[k] = w[k] + (alpha \* delta \* attributes[j][k])

print (attributes[j][0], attributes[j][1], "->actual: ", actual, " yin: ", yin, " weights: ", w[0], " ", w[1])

Screenshot 2:



Code 3:

import numpy as np

import random

import sklearn

import pandas as pd

from sklearn.datasets import make\_regression

import pylab

import math

from scipy import stats

df = pd.read\_csv('USA\_Housing.csv', sep=",")

X = df.values

def gradient\_descent(alpha, X, ep=0.0001, max\_iter=10000):

converged = False

iter = 0

MSE = []

###### WRITE CODE HERE ########

#1. Initialize Thetas according to number of Features in your dataset theta=[0.1,2.5,2.0,4.5,4.0,3.0]

#2. Calculate total Cost for all instances i.e. (Loop Comprehension for S

um)

# Iterate Loop

while not converged:

SE=0

for row in range (0,5000):

cost = theta[0] + X[row][0]\*theta[1] + X[row][1]\*theta[2] + X[row] [2]\*theta[3] + X[row][3]\*theta[4] + X[row][4]\*theta[5]

error = X[row][5] - cost

print("cost: ", cost)

print("err: ", error)

#3. compute the gradient (d/d\_theta j(theta))

#4. Apply gradient descent algorithm to update weights

#5. Retain previous values of Thetas i.e. (By assignment to new va riables t0, t1)

gradients = [

theta[0] - (alpha\*(error)),

theta[1] - (alpha\*(error)\*X[row][0])

theta[2] - (alpha\*(error)\*X[row][1]),

theta[3] - (alpha\*(error)\*X[row][2]),

theta[4] - (alpha\*(error)\*X[row][3]),

theta[5] - (alpha\*(error)\*X[row][4])

]

for x in range(6):

theta[x]=gradients[x]

SE += (SE- error) \* (SE- error)

#6. Compute Mean Squqre error for updated thetas as done in step 2 MSE.append( (SE / 5000 ) )

#7. Calculate error differce. Keep converged flag True if done i.e. (U se Selection Structure)

if iter != 0:

error\_diff = MSE[iter] - MSE[iter-1]

if error\_diff <= ep:

converged = True

#8. Update error in J and increment iteration for reiteration

theta =gradients.copy()

iter +=1

#9. In case maximum iterations done stop the loop by making convered f lag to true

if iter > max\_iter :

converged = True

return theta # Updated thetas returned

x, y = make\_regression(n\_samples=100, n\_features=1, n\_informative=1, random\_state=0, noise=35)

print ('x.shape = %s y.shape = %s' %(x.shape, y.shape))

alpha = 0.01 # learning rate

ep = 0.01 # convergence criteria

call gredient decent, and get intercept(=theta0) and slope(=theta1) theta = gradient\_descent(0.00000000001, X, ep=0.0001, max\_iter=100) print ('theta0 =',theta[0])

print ('theta1 =',theta[1]) print ('theta2 =',theta[2]) print ('theta3 =',theta[3]) print ('theta4 =',theta[4])

check with scipy linear regression

slope, intercept, r\_value, p\_value, slope\_std\_error = stats.linregress(x[:,0], y)

print('intercept = ', intercept)

print('Slope = ', slope)

# plot

for i in range(x.shape[0]):

y\_predict = theta0 + theta1\*x

pylab.plot(x,y,'o')

pylab.plot(x,y\_predict,'k-')

pylab.show()

print ("Done!")

Screenshot:

