**Graphs:**

# Lab 4A

# **SAMEER ALI BUTT**

# **01-134202-095**

Consider a simple (directed) graph (digraph) having six nodes (A-F) and the following arcs (directed edges):

A -> B A -> C B -> C B -> D C -> D C -> F D -> C D -> E E -> F F -> C F -> E

It can be represented by the following Python data structure:

graph = {'A': ['B', 'C'],

'B': ['C', 'D'],

'C': ['D',‟F‟],

'D': ['C',‟E‟],

'E': ['F'],

'F': ['C',‟E‟]}

This is a dictionary whose keys are the nodes of the graph. For each key, the corresponding value is a list containing the nodes that are connected by a direct arc from this node. This is about as simple as it gets (even simpler, the nodes could be represented by numbers instead of names, but names are more convenient and can easily be made to carry more information, such as city names).

Let's write a simple function to determine a path between two nodes. It takes a graph and the start and end nodes as arguments. It will return a list of nodes (including the start and end nodes) comprising the path. When no path can be found, it returns None. The same node will not occur more than once on the path returned (i.e. it won't contain cycles). The algorithm uses an important technique called *backtracking*: it tries each possibility in turn until it finds a solution.

def find\_path(graph, start, end, path=[]): path = path + [start]

if start == end: return path

if start not in graph: return None

newpath = None # Define newpath before the loop for node in graph[start]:

if node not in path:

newpath = find\_path(graph, node, end, path) if newpath:

break # Exit the loop if newpath is assigned a value return newpath # Return newpath outside the loop

A sample run of the function find\_path() (using the graph above):

>>> find\_path(graph, 'A', 'D') ['A', 'B', 'C', 'D']

### Example 2

class DWGraph:

def init (self): self.graph = {}

def add\_node(self, node): if node not in

self.graph:

self.graph[node] =

{}

def add\_edge(self, start, end, weight):

if start not in self.graph:

self.add\_node(start)

if end not in self.graph:

self.add\_node(end) self.graph[start][end]

= weight

g = DWGraph() g.add\_edge('A', 'B', 2)

g.add\_edge('A', 'C', 1)

**Lab Journal 4A:**

1. Change the function find path to return shortest path.

**Code:**

ef shortest\_path(graph, start, end, path=[]):

path = path + [start]

if start == end:

return path

shortest = None

for node in graph[start]:

if node not in path:

newpath = shortest\_path(graph, node, end, path.copy()) # Use a copy of the path

if newpath:

if not shortest or len(newpath) < len(shortest):

shortest = newpath

return shortest

graph = {

'A': ['B', 'C'],

'B': ['C', 'D'],

'C': ['D', 'F'],

'D': ['C', 'E'],

'E': ['F'],

'F': ['C', 'E']

}

print("Graph:")

print(graph)

print("Shortest path is:")

print(shortest\_path(graph, 'A', 'F'))

**Output:**

****

1. Consider a simple (directed) graph (digraph) having six nodes (A-F) and the following arcs (directed edges) with respective cost of edge given in parentheses:

A -> B (2)

A -> C (1)

B -> C (2)

B -> D (5)

C -> D (1)

C -> F (3)

D -> C (1)

D -> E (4)

E -> F (3)

F -> C (1)

F -> E (2)

graph = {

'A': {'B': 2, 'C': 1},

'B': {'C': 2, 'D': 5},

'C': {'D': 1, 'F': 3},

'D': {'C': 1, 'E': 4},

'E': {'F': 3},

'F': {'C': 1, 'E': 2}

}

Using the code for a directed weighted graph, instantiate an object of DWGraph in

main , add the nodes and edges of the graph using the relevant functions, and implement a function find\_path() that takes starting and ending nodes as arguments and returns at least one

path (if one exists) between those two nodes. The function should also keep track of the cost of the path and return the total cost as well as the path. Print the path and its cost in main.

**Code:**

class Graph:

def \_\_init\_\_(self, nodes=None, edges=None):

self.nodes = []

self.adj = {}

if nodes is not None:

self.add\_nodes\_from(nodes)

if edges is not None:

self.add\_edges\_from(edges)

def length(self):

return len(self.nodes)

def traverse(self):

return 'V: %s\nE: %s' % (self.nodes, self.adj)

def add\_node(self, n):

if n not in self.nodes:

self.nodes.append(n)

self.adj[n] = []

def add\_nodes\_from(self, i):

for n in i:

self.add\_node(n)

def add\_edge(self, u, v): # undirected unweighted graph

self.adj[u] = self.adj.get(u, []) + [v]

self.adj[v] = self.adj.get(v, []) + [u]

def add\_edges\_from(self, i):

for n in i:

self.add\_edge(\*n)

def number\_of\_nodes(self):

return len(self.nodes)

def number\_of\_edges(self):

return sum(len(l) for \_, l in self.adj.items()) // 2

class DGraph(Graph):

def add\_edge(self, u, v):

self.adj[u] = self.adj.get(u, []) + [v]

class WGraph(Graph):

def \_\_init\_\_(self, nodes=None, edges=None):

self.nodes = []

self.adj = {}

self.weight = {}

if nodes is not None:

self.add\_nodes\_from(nodes)

if edges is not None:

self.add\_edges\_from(edges)

def add\_edge(self, u, v, w):

self.adj[u] = self.adj.get(u, []) + [v]

self.adj[v] = self.adj.get(v, []) + [u]

self.weight[(u, v)] = w

self.weight[(v, u)] = w

def get\_weight(self, u, v):

return self.weight[(u, v)]

class DWGraph(WGraph):

def add\_edge(self, u, v, w):

self.adj[u] = self.adj.get(u, []) + [v]

self.adj[v] = self.adj.get(v, []) + [u]

self.weight[(u, v)] = w

self.weight[(v, u)] = w

def get\_weight(self, u, v):

return self.weight[(u, v)]

def find\_path(self, start, end, path=None, cost=0):

if path is None:

path = []

newpath = None

path = path + [start]

if start == end:

return path, cost

if start not in self.adj:

return None, cost

paths = []

for node in self.adj[start]:

if node not in path:

newpath, cost = self.find\_path(node, end, path, cost + self.get\_weight(start, node))

if newpath is not None:

return newpath, cost

return None, cost

D = DWGraph()

D.add\_node('A')

D.add\_edge('A', 'B', 2)

D.add\_edge('A', 'C', 1)

D.add\_edge('B', 'C', 2)

D.add\_edge('B', 'D', 5)

D.add\_edge('C', 'D', 1)

D.add\_edge('C', 'F', 3)

D.add\_edge('D', 'C', 1)

D.add\_edge('D', 'E', 4)

D.add\_edge('E', 'F', 3)

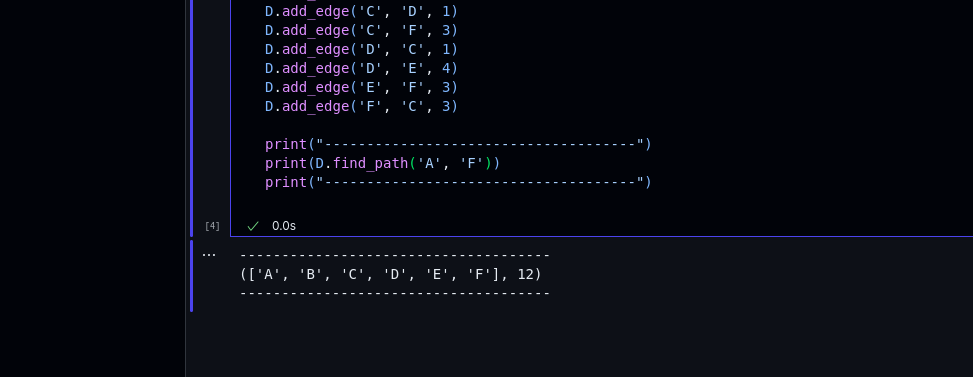
D.add\_edge('F', 'C', 3)

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

print(D.find\_path('A', 'F'))

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

**Output:**

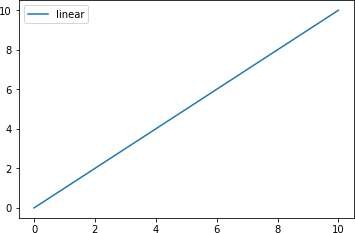
****

# Lab 4B

## Matplotlib:

Data Visualization is very much important task in machine learning. For data visualization in python we use matplotlib and sklearn modules. With the usage of these modules we can have clear view of how data points are rendered in 2D space. Below are some exercises which would help you to plot graphs in an easier way.

### Example-1: Line Plot in Python.

import matplotlib.pyplot as plt import numpy as np

# Prepare the data

# Generates 100 samples between 0 to 10

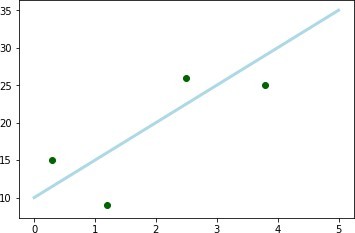
x = np.linspace(0, 10, 100) # Plot the data

plt.plot(x, x, label='linear') # Add a legend

plt.legend()

# Show the plot plt.show()

### Example-2: Scatter Plot in Python.

import matplotlib.pyplot as plt plt.plot([0, 1, 2, 3, 4, 5], [10, 15, 20,

25, 30, 35], color='lightblue',

linewidth=3)

plt.scatter([0.3, 3.8, 1.2, 2.5], [15,

25, 9, 26], color='darkgreen', marker='o')

plt.show()

### Example-3: Sub Plots in Python.

import matplotlib.pyplot as plt fig = plt.figure()

# Set up Axes

sub1 = fig.add\_subplot(221) sub2 = fig.add\_subplot(222) sub3 = fig.add\_subplot(223) sub4 = fig.add\_subplot(224)

# Scatter the data

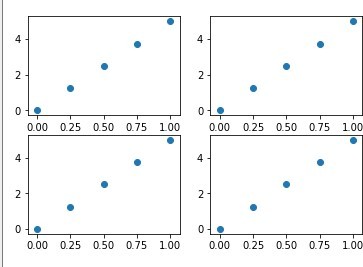
sub1.scatter(np.linspace(0, 1, 5), np.linspace(0, 5, 5))

sub2.scatter(np.linspace(0, 1, 5), np.linspace(0, 5, 5))

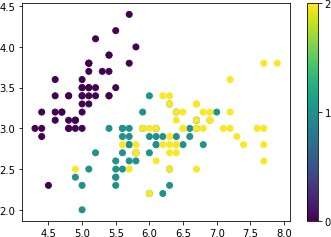
sub3.scatter(np.linspace(0, 1, 5), np.linspace(0, 5, 5))

sub4.scatter(np.linspace(0, 1, 5), np.linspace(0, 5, 5)) # Show the plot

plt.show()



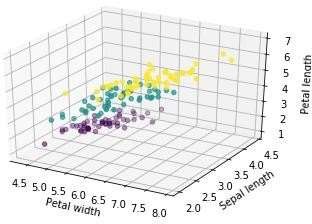
### Example-4: Iris Dataset with 2 Features.

from sklearn import datasets import matplotlib.pyplot as plt

iris = datasets.load\_iris() plt.scatter(iris.data[:,0],iris.data[:,1], c=iris.target)

plt.colorbar(ticks=[0, 1, 2]) plt.show()

### Example-4: 3D Plots in Python.

from mpl\_toolkits import mplot3d from sklearn import datasets import numpy as np

import matplotlib.pyplot as plt fig = plt.figure()

ax = plt.axes(projection='3d') iris = datasets.load\_iris() ax.scatter(iris.data[:,0], iris.data[:,1],iris.data[:,2], c=iris.target) ax.set\_xlabel('Petal width') ax.set\_ylabel('Sepal length') ax.set\_zlabel('Petal length')

## Lab Journal 4B:

Visualize the following data in python. Please provide the reason for the choice of graph.

|  |  |
| --- | --- |
| Feature 1 | Feature 2 |
| 12 | 4 |
| 11 | 5 |
| 8 | 1 |
| 6 | 4 |
| 9 | 3 |
| 6 | 6 |
| 10 | 2 |

**Code:**

import matplotlib.pyplot as plt

import numpy as np

from mpl\_toolkits.mplot3d import Axes3D # Import Axes3D from mpl\_toolkits.mplot3d

from sklearn import datasets

x = np.array([12, 11, 8, 6, 9, 6, 10])

y = np.array([4, 5, 1, 4, 3, 6, 2])

# Plotting the first set of data

plt.figure()

plt.plot(x, y, label='linear', color='lightblue', linewidth=3)

plt.scatter([0.3, 3.8, 1.2, 2.5], [15, 25, 9, 26], color='darkgreen', marker='P')

plt.legend()

plt.show()

# Creating a 2x2 subplot

fig = plt.figure()

sub1 = fig.add\_subplot(221)

sub2 = fig.add\_subplot(222)

sub3 = fig.add\_subplot(223)

sub4 = fig.add\_subplot(224)

# Scatter plots in the subplots

sub1.scatter(np.linspace(0, 1, 5), np.linspace(0, 5, 5))

sub2.scatter(np.linspace(0, 1, 5), np.linspace(0, 5, 5))

sub3.scatter(np.linspace(0, 1, 5), np.linspace(0, 5, 5))

sub4.scatter(np.linspace(0, 1, 5), np.linspace(0, 5, 5))

plt.show()

# Creating a 3D scatter plot

fig = plt.figure()

ax = fig.add\_subplot(111, projection='3d') # Use add\_subplot to create a 3D subplot

iris = datasets.load\_iris()

ax.scatter(iris.data[:, 0], iris.data[:, 1], iris.data[:, 2], c=iris.target)

ax.set\_xlabel('Petal width')

ax.set\_ylabel('Sepal length')

ax.set\_zlabel('Petal length')

plt.show()

# 2D scatter plot with a color bar

iris = datasets.load\_iris()

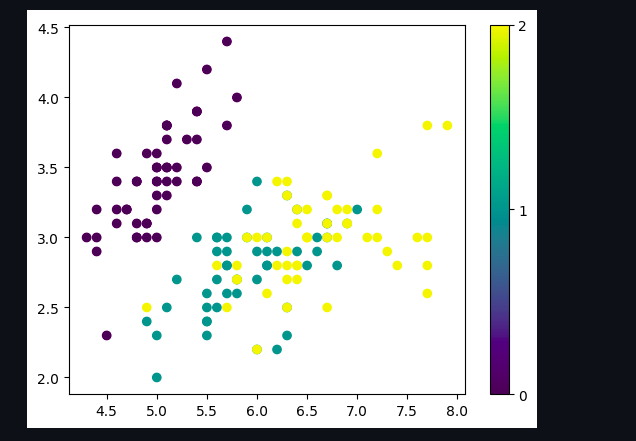
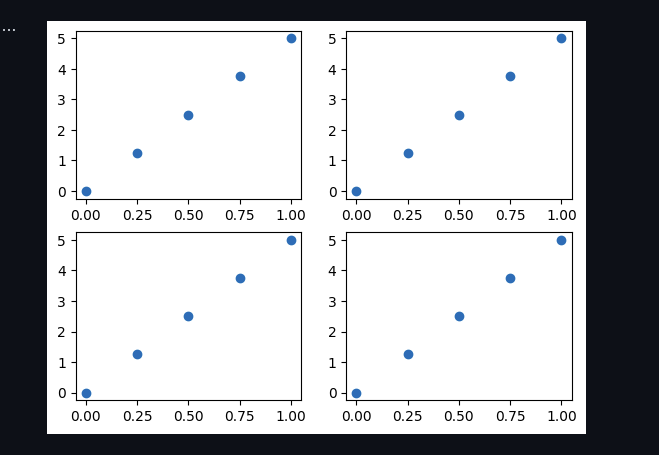
plt.scatter(iris.data[:, 0], iris.data[:, 1], c=iris.target)

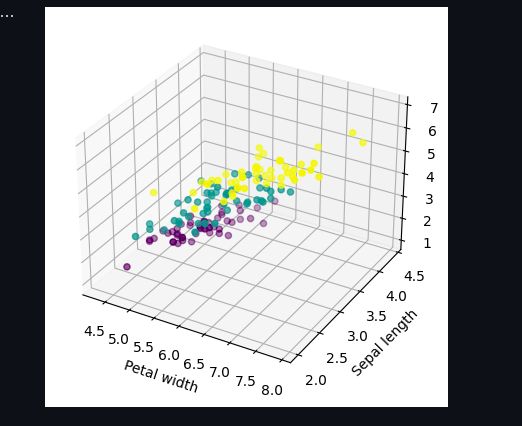
plt.colorbar(ticks=[0, 1, 2])

plt.show()

**Output:**

****

****

****