**CEE 505**

**Jean-Luc Jackson & Connor Lester**

**11/7/2016**

**Assignment #4**

**The Graph Problem**

##### **Problem Statement**

Develop a Python program that:

*Given the provided data files and code skeleton:*

1. *Design, implement, and document a program which finds paths through a given graph.*
2. *Analyze possible paths on paper and demonstrate that your code finds all existing, no duplicates, and no false ones.*
3. *Upload the documentation as a single document.*

### **Code Description**

##### **Solution Overview**

This program constructs a graph of nodes and lines by reading two text files, one containing node names and their locations, the other containing line names and the nodes they connect. Once the node-line graph is created, all possible paths can be calculated between two given nodes. The shortest path of these possible paths can be determined as well.

This program creates *Node,* *Line*, and *Path* objects containing respective methods that provide information about each object and offer certain operations. The *Node* and *Line* classes are very similar in that they both have object ID’s, contain internal lists that store connecting objects, and have methods that attach and detach nodes to lines and vice versa. The *Path* class stores information about a path between an initial and destination node, so its internal storage lists store more information such as nodes and lines traveled.

The *Graph* class drives the entire program, reading the imported files to create the graph and containing methods that facilitate calculation of all possible paths between two provided nodes. *Graph* contains various methods, most of which return node, line, or path information. The following tables list all classes and their methods.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Class** | **Functions** | **Class** | **Functions** | **Class** | **Functions** | **Class** | **Functions** |
| **Graph** | \_\_init\_\_ | **Line** | \_\_init\_\_ | **Node** | \_\_init\_\_ | **Path** | \_\_init\_\_ |
|  | \_\_str\_\_ |  | \_\_str\_\_ |  | \_\_str\_\_ |  | \_\_str\_\_ |
|  | getNode |  | \_\_len\_\_ |  | getPosition | | \_\_add\_\_ |
|  | getAllNodes | | getLength |  | getID |  | \_\_sub\_\_ |
|  | numNodes | | getID |  | getLineIDs | | getID |
|  | getNodePosition | | getNodeIDs | | attach |  | getLength |
|  | getLine |  | attach |  | detach |  | getNodeHistory |
|  | numLines |  | detach |  |  |  | getLineHistory |
|  | getLineLength | |  |  |  |  |  |
|  | getLineGivenNodes | |  |  |  |  |  |
|  | getNodeGivenLine | |  |  |  |  |  |
|  | printGraph | |  |  |  |  |  |
|  | attach |  |  |  |  |  |  |
|  | detach |  |  |  |  |  |  |
|  | findPaths |  |  |  |  |  |  |
|  | savePathstoDict | |  |  |  |  |  |
|  | getPathDict | |  |  |  |  |  |
|  | findShortestPath | |  |  |  |  |  |

##### **Implementation Details**

###### Graphing a Node-Line Graph

The *Graph* class can produce a figure plotting a visual representation of the imported node-line plot. Upon calling *printGraph*, coordinate lists are compiled by looping through the list of lines and nodes. Then, numpy and matlibplot functions are utilized to create a plot as shown in the following section. This function takes advantage of the method *annotate* to label the nodes in the plot with their string label. This is particularly useful when visually seeking a route between two nodes.

###### Finding All Possible Paths

The *Graph* class can calculate all possible paths between two requested nodes. Upon calling *findPaths*, possible paths are found using recursion taking an initial node and destination node as arguments. The recursive process occurs as follows:

1. Step on the provided node and get information about that node.
2. Evaluate all connecting lines and whether they have been traveled on before.
3. Travel down untraveled lines.
4. Create a *Path* object that documents the path traveled along this recursive iteration.
5. Call a new iteration of itself that begins at the node at the other end of each evaluated line.

*Path* objects are created and added up along the travel of *findPaths*, documenting the node and line histories. If a line has been traveled on but the node at the other end is not the destination node, the method stops traveling in that direction. If the node being evaluated is the destination node, the path up to that point is saved into a list that stores all possible paths (*allPaths*). Finally, when the list contains all possible paths between the two nodes, this list of paths is filed into an internal dictionary by calling the internal function *savePathsToDict*.

A subsequent method, *findShortestPath*, utilizes the dictionary of paths between nodes compiled by *findPaths*. Each pair of beginning-destination nodes is used as a string key that is mapped to a list value containing all possible paths between this node pair. *findShortestPath* uses this dictionary by picking out the node pair being investigated, looping through all possible paths, and saving the shortest path length by using the *Path* object’s *getLength* method.

###### Exception Handling

Throughout the code, try and except statements were used to handle issues that may arise when using this code. In order to quickly determine the origin of a failure, the structure of the code was utilized such that functions often called within other functions would perform the exception handling. For example, the *Graph* object’s *getNode* function is often called throughout the code. Therefore, when the *findPath* function is called between nodes where one or all may not exist, the *getNode* function is called and handles the *IndexError* exception.

For our purposes the code structure was viewed to act as shown below, where *Graph* acts as a ‘surface’ object built upon the structure of deeper classes.

Graph 🡪 Line 🡪 Node 🡪 Vector

Graph 🡪 Path

Some error handling, in the *Node* attachment and detachment functions for example, use if statements in order to show if the action performed worked as intended.

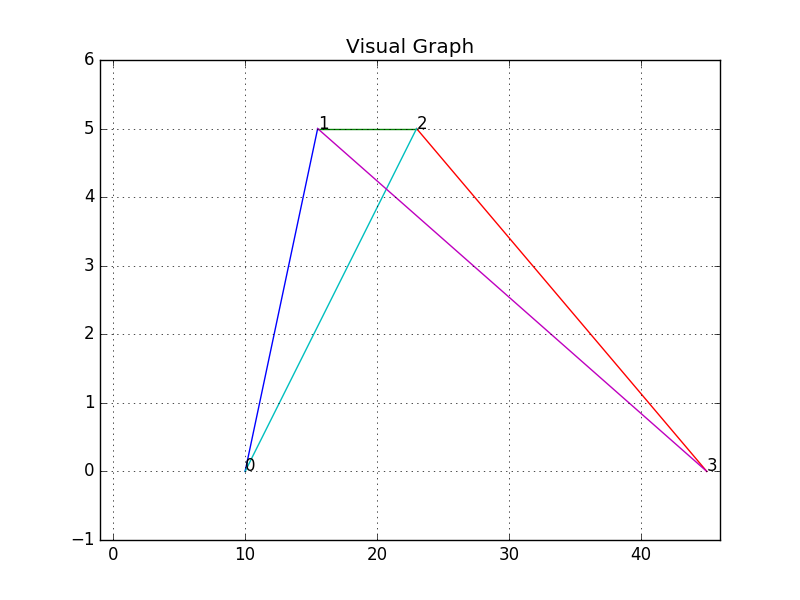
##### **Code Verification**

To verify the program, a graph for the example input *Graph1* was inspected by-hand. Appendix 1 shows this by-hand inspection, followed by a code output for comparison. Each line’s length was calculated and all possible paths between two specific nodes were found. In this case, all paths between nodes ‘0’ and ‘3’ were found. Of the four possible paths, their node histories and total path lengths were found and compared. The program produced the same paths as the by-hand inspection, including node history and total length.

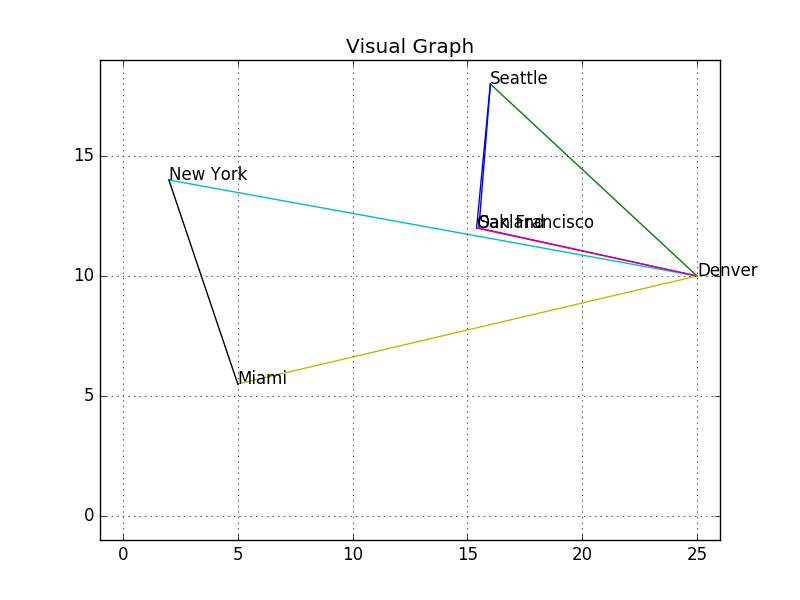
A similar inspection was performed for the example input *Graph2*. This comparison is shown in Appendix 2 and walks through a comparison of path node histories and lengths as described above for *Graph1*.

##### **Example Output**

###### Graph1 (Numbered Nodes ‘0’, ‘1’, ‘2’, ’3’)



###### Graph2 (City Nodes ‘Seattle’, ‘San Francisco’, ’Denver’, ‘Miami’, ’New York’, ’Oakland’)



### **Python Code**

#### Graph Class

*'''*

*Jean-Luc Jackson & Connor Lester*

*CEE 505 HW #4*

*11/07/16*

*'''*

#Import Directories

import os

import copy

import matplotlib.pyplot as plt

# Import Our Classes

from Line import \*

from Node import \*

from Path import \*

class **Graph**(object):

*'''*

*Class Hosts A Graph Object.*

*Reads Data From Provided .txt File*

*Creates/Stores A Series of Node-Line Relationships*

*Contained Attributes - Line Objects, Node Objects*

*Contained Algorithms - Path Finder, Shortest Path Finder*

*'''*

def **\_\_init\_\_**(*self*, folder\_name):

*'''*

*Creates a Graph Object By Reading 'nodes.txt' and 'lines.txt'*

*Files Located Provided Folder.*

*'''*

# Changes Directory To Folder Location

os.chdir( folder\_name )

*self*.folder\_name = folder\_name

#print "Directory changed to {}. Building Graph...".format(folder\_name)

# Read 'nodes.txt' and Generate List Of Nodes

#print "\nReading Nodes..."

*self*.nodes = []

f = open(*'nodes.txt'*, *'r'*)

for line in f:

splitline = line.strip().split(*'\t'*)

nodeID = splitline[0]

nodeX = float(splitline[1])

nodeY = float(splitline[2])

#Generates Node Object

thisNode = Node(nodeID,Vector([nodeX,nodeY]))

#print thisNode

*self*.nodes.append(thisNode)

f.close()

# Read 'lines.txt' and Generate List Of Lines

#print "\nReading Lines..."

*self*.lines = []

f = open(*'lines.txt'*, *'r'*)

for line in f:

splitline = line.strip().split(*'\t'*)

lineID = splitline[0]

nodeIid = splitline[1]

nodeJid = splitline[2]

nodeIobj = filter(lambda x: x.ID == nodeIid, *self*.nodes)[0]

nodeJobj = filter(lambda x: x.ID == nodeJid, *self*.nodes)[0]

# Generates Line Object

thisLine = Line(lineID,nodeIobj,nodeJobj)

#print thisLine

*self*.lines.append(thisLine)

# Attach Line IDs to Nodes

nodeIobj.attach(lineID)

nodeJobj.attach(lineID)

f.close()

*self*.pathDict = dict()

*'''STANDARD CALLS'''*

def **\_\_str\_\_**(*self*): # Print Statement

*'''*

*Returns Number of Total Lines and Nodes As String.*

*'''*

s = *"Graph is made of {} lines and {} nodes."*.format(*self*.numLines(),*self*.numNodes())

return s

*'''INFORMATION CALLS'''*

def **getNode**(*self*,nodeID):

try:

nodeID = str(nodeID)

return [i for i in *self*.nodes if i.getID() == nodeID][0]

# If Node Does Not Exist In Graph Object

except IndexError:

raise IndexError(*"Node {} does not exist."*.format(nodeID))

def **getAllNodes**(*self*):

return [n.getID() for n in *self*.nodes]

def **numNodes**(*self*):

*'''*

*Returns the Number (Integer) of Nodes That Exist Within This Graph Object.*

*'''*

return len(*self*.nodes)

def **getNodePosition**(*self*,nodeID): # Exception Handling Included In getNode()

*'''*

*Calls the Get Position Function For A Given Line.*

*'''*

node = *self*.getNode(nodeID)

return node.getPosition()

def **getLine**(*self*,lineID):

try:

lineID = str(lineID)

return [i for i in *self*.lines if i.getID() == lineID][0]

# If Line Does Not Exist In Graph Object

except IndexError:

raise IndexError(*"Line {} does not exist."*.format(lineID))

def **numLines**(*self*):

*'''*

*Returns the Number (Integer) of Lines That Exist Within This Graph Object.*

*'''*

return len(*self*.lines)

def **getLineLength**(*self*,lineID): # Exception Handling Included In getLine()

*'''*

*Calls the Length Function For A Given Line.*

*'''*

line = *self*.getLine(lineID)

return line.getLength()

def **getLineGivenNodes**(*self*,nodeIid,nodeJid): # Exception Handling Included In getNode()

*'''*

*Finds the Line Connecting Given Nodes*

*'''*

iLines = *self*.getNode(str(nodeIid)).getLineIDs()

jLines = *self*.getNode(nodeJid).getLineIDs()

try:

lineID = [i for i in iLines if i in jLines]

return [j for j in *self*.lines if j.getID() is lineID][0]

# If No Lines Attached To Provided Nodes

except IndexError:

raise IndexError(*"No line exists between nodes {} and {}."*.format(nodeIid,nodeJid))

def **getNodeGivenLine**(*self*,nodeIid,lineID): # Exception Handling Included In getNode() and getLine()

*'''*

*Finds the Second Node For A Given Node and Line*

*'''*

nodeIid = str(nodeIid)

lineID = str(lineID)

nodeIid = *self*.getNode(nodeIid).getID()

lineID = *self*.getLine(lineID).getID()

#If Line Connects To Provided Node

try:

nodeIid = [i for i in *self*.getLine(lineID).getNodeIDs()\

if i is nodeIid][0]

#If Line Connects To Another Node

try:

nodeJid = [i for i in *self*.getLine(lineID).getNodeIDs()\

if i is not nodeIid][0]

return *self*.getNode(nodeJid)

# If Line Does Not Connect To Another Node

except:

raise IndexError(*"Error: No node exists at the other end of line {} from node {}."*\

.format(lineID,nodeIid))

# If Line Does Not Connect To Provided Node

except:

raise IndexError(*"Error: Line {} does not connect to node {}."*.format(lineID, nodeIid))

def **printGraph**(*self*):

*'''*

*Provides Graph Visulization*

*'''*

try:

# Create List of X and Y Coordinates

x = []

y = []

minx, miny, maxx, maxy = 0.0, 0.0, 0.0, 0.0

for line in *self*.lines:

x = [line.nodes[0].coord[0],line.nodes[1].coord[0]]

y = [line.nodes[0].coord[1],line.nodes[1].coord[1]]

plt.plot(x,y)

#Format Plot Size

x.append(minx)

x.append(maxx)

y.append(miny)

y.append(maxy)

minx, miny, maxx, maxy = min(x), min(y), max(x), max(y)

# Format Plot

plt.title(*'Visual Graph'*)

plt.axis([minx-1,maxx+1,miny-1,maxy+1])

plt.grid(True)

# Label Nodes

for n in *self*.nodes:

plt.annotate(*'{}'*.format(n.getID()), xy = (n.getPosition()[0],n.getPosition()[1]), textcoords=*'data'*)

# Save Figure & Plot

plotfilename = *self*.folder\_name + *'.png'*

plt.savefig(plotfilename)

plt.show()

# If Any Free Lines or Nodes

except:

raise IndexError(*"Lines or nodes are not fully attached."*)

*'''ACTION CALLS'''*

def **attach**(*self*,lineID,nodeID): # Exception Handling Included In getNode() and getLine()

*'''*

*Calls Line Attachment Function.*

*'''*

line = *self*.getLine(lineID)

node = *self*.getNode(nodeID)

line.attach(node)

def **detach**(*self*,lineID,nodeID): # Exception Handling Included In getNode() and getLine()

*'''*

*Calls Line Detachment Function.*

*'''*

line = *self*.getLine(lineID)

node = *self*.getNode(nodeID)

line.detach(node)

*'''PATH FINDING CALLS'''*

def **findPaths**(*self*, startNodeID, destNodeID, pastPath=Path(), allPaths=[],first=True):

*'''*

*This Method Travels Down Lines Until destNode Has Been Reached.*

*Returns List Containing Path Objects Which Reached destNodeID.*

*Takes arguments: -startNodeID, From Which Subsequent Paths Are Found*

*-destNode, Final Node We're Seeking As Our Destination*

*-pastPath, Path We've Traveled Up To This Point*

*-allPaths, List Containing Path Objects Seeking destNoteID*

*'''*

# If it's the first recursion (just got called), clear the previous findPath data

if first:

allPaths = []

# Find Node Objects From IDs

startNode = *self*.getNode(startNodeID)

destNode = *self*.getNode(destNodeID)

# If We've Visited This Node Before, Don't Go Any Further

# Return An Empty List

for i in pastPath.getNodeHistory():

if startNodeID == i.getID():

#print "Line has already been stepped through."

lastLine = pastPath.getLineHistory()[-1]

# remove last line and length so we don't save it

pastPath -= Path([],[lastLine],lastLine.getLength())

return []

# If We've Reached Our Destination, Don't Go Any Further

# Return The Path Up to This Point

if startNode == destNode:

# Add Destination Node to Path (Step On Node)

pastPath += Path([startNode],[],0.0)

# Find The Last Line And Node

lastLine = pastPath.getLineHistory()[-1]

lastNode = pastPath.getNodeHistory()[-1]

finalPath = copy.deepcopy(pastPath)

#Save finalPath

finalPath = copy.deepcopy(pastPath)

#Remove Previous Line, Node, And Line Length

pastPath -= Path([lastNode],[lastLine],lastLine.getLength())

#print "Previous {}".format(pastPath)

*self*.savePathsToDict(allPaths)

return [finalPath]

# If This Node Has Not Been Visited

# Add Current Node to Path (Step On Node)

pastPath += Path([startNode],[],0.0)

lineIDs = startNode.getLineIDs()

# Find Path For Each Line Attached to startNode

for lineID in lineIDs:

# Find Line Object From ID

line = *self*.getLine(lineID)

# Find endNode Down Line Object

endNode = *self*.getNodeGivenLine(startNodeID, lineID)

endNodeID = endNode.getID()

#Find nextPath Usind The End Node

tempPath = pastPath + Path([],[line],line.getLength())

nextPath = *self*.findPaths(endNodeID,destNodeID,tempPath,allPaths,False)

if len(nextPath) == 0 and *self*.getLine(lineID) not in pastPath.getLineHistory() and endNode not in pastPath.getNodeHistory():

nodes = [n.getID() for n in pastPath.getNodeHistory()]

pastPath += Path([],[line],line.getLength())

if len(nextPath) == 1 and nextPath[0] not in allPaths:

allPaths.append(nextPath[0])

return allPaths

def **savePathsToDict**(*self*, allPaths):

try:

initialNode = allPaths[0].getNodeHistory()[0].getID()

lastNode = allPaths[0].getNodeHistory()[-1].getID()

except IndexError:

return

pathName = *"{}-{}"*.format(initialNode,lastNode)

*self*.pathDict[pathName] = allPaths

#print "Internal PathDict = {}".format(self.pathDict)

def **getPathDict**(*self*):

return *self*.pathDict

def **findShortestPath**(*self*, startID, endID):

*'''*

*'''*

try:

s = *"{}-{}"*.format(startID,endID)

pathsOf = *self*.pathDict[s]

except KeyError:

raise KeyError(*'Paths have not yet been found between nodes {} and {}. Use findPaths first.'*.format(startID,endID))

minL = 0

shortestPath = pathsOf[0]

for path in pathsOf[1:]:

if path.getLength() < shortestPath.getLength():

shortestPath = path

nodeHistory = shortestPath.getNodeHistory()

printHistory = [node.getID() for node in nodeHistory]

printString = *'( '*

for n in range(0,len(printHistory) - 1):

printString += printHistory[n] + *' -> '*

printString += printHistory[-1] + *' )'*

sRet = *"The shortest path from {} to {} is {} with a total length of {:.2f}."*\

.format(startID,endID,printString,shortestPath.getLength())

return sRet

#### Line Class

*'''*

*Jean-Luc Jackson & Connor Lester*

*CEE 505 HW #4*

*11/07/16*

*'''*

class **Line**(object):

*'''*

*Class Hosts A Line Object.*

*Contained Attributes - Line ID, Node Objects And Node IDs.*

*'''*

def **\_\_init\_\_**(*self*, ID, nodeIobj, nodeJobj):

*'''*

*Constructor.*

*'''*

# Store ID

*self*.ID = ID

# Store/Sort Node Object As List

*self*.nodes = [nodeIobj, nodeJobj]

*self*.nodes.sort()

# Store/Sort Node Coordinates As List

*self*.nodeIDs = [nodeIobj.ID, nodeJobj.ID]

*self*.nodeIDs.sort()

*'''STANDARD CALLS'''*

def **\_\_str\_\_**(*self*): # Print Statement

s = *"Line {} connects node {} to node {}."*.format(*self*.ID,*self*.nodeIDs[0],*self*.nodeIDs[1])

return s

def **\_\_len\_\_**(*self*): # Raises An Error

raise TypeError(*"Line object has no integer length. Use getLength() for line length."*)

*'''INFORMATION CALLS'''*

def **getLength**(*self*):

*'''*

*Uses Vector Math to Calculate the Length of the Line (See Vector Class).*

*Returns the Length of the Line as a Float.*

*'''*

try:

a = (*self*.nodes[0].coord)

b = (*self*.nodes[1].coord)

ab = (b-a)

c = (ab[0]\*\*2+ab[1]\*\*2)\*\*(0.5)

return c

# If Line Is Not Connected To Two Nodes (Line Has Zero Length)

except:

raise IndexError(*"Line {} is not fully connected and has no length."*\

.format(*self*.ID))

def **getID**(*self*):

*'''*

*Returns Line ID.*

*'''*

return *self*.ID

def **getNodeIDs**(*self*):

*'''*

*Returns Node ID List.*

*'''*

return *self*.nodeIDs

*'''ACTION CALLS'''*

def **attach**(*self*, nodeobj):

*'''*

*Attaches Line to the Provided Node.*

*Adds Node to Internal Node List.*

*Adds Node ID to Internal Node ID List.*

*'''*

# If the Line is Not Full And If the Provided Node is Not Already Attached

if len(*self*.nodeIDs) < 2 and (nodeobj not in *self*.nodes):

# Appends Node Object

*self*.nodes.append(nodeobj)

*self*.nodes.sort()

# Appends Node ID

*self*.nodeIDs.append(nodeobj.ID)

*self*.nodeIDs.sort()

print *"Line {} attached to node {}. Line now connects nodes {} and {}."*\

.format(*self*.ID,nodeobj.ID,*self*.nodeIDs[0],*self*.nodeIDs[1])

# Calls Node Attachment And Passes Line ID

nodeobj.attach(*self*.ID)

# If the Line is Full Xor the Provided Node is Already Attached

else:

print *"Error: Line {} already connected to nodes {} and {}."*\

.format(*self*.ID,*self*.nodeIDs[0],*self*.nodeIDs[1]),\

*"Use detach() before attaching to another node."*

def **detach**(*self*, nodeobj):

*'''*

*Detaches Line From the Provided Node.*

*Removes Node From Internal Node List.*

*Removes Node ID From Internal Node ID List.*

*'''*

# If the Line is Attached to At Least One Node

if len(*self*.nodes) > 0:

# If the Line is Attached to the Provide Node

if nodeobj in *self*.nodes:

*self*.nodes.remove(nodeobj)

*self*.nodeIDs.remove(nodeobj.ID)

print *"Line {} detached from node {}."*.format(*self*.ID,nodeobj.ID)

# Calls Node Detachment And Passes Line ID

nodeobj.detach(*self*.ID)

# If the Line is Not Attached to the Provided Node

else:

print *"Error: Line {} is not connected to node {}."*\

.format(*self*.ID,nodeobj.ID),\

*"Use attach() to do so."*

# If the Line is Not Attached to Any Nodes

else:

print *"Error: Line {} is not connected to any nodes. Use attach() to do so."*.format(*self*.ID)

#### Node Class

*'''*

*Jean-Luc Jackson & Connor Lester*

*CEE 505 HW #4*

*11/07/16*

*'''*

from Vector import \*

class **Node**(object):

*'''*

*Class Hosts a Node Object.*

*Contained Attributes - Node ID and Vector Object.*

*Vector Object Functions as Coordinates For Node's Position.*

*'''*

def **\_\_init\_\_**(*self*, ID, v):

*'''*

*Constructor.*

*'''*

# Store ID

*self*.ID = ID

# Store Node Coordinates As Vector

*self*.coord = Vector(v)

# Keep A List of Attached Lines (their IDs)

*self*.lineIDs = []

def **getPosition**(*self*):

*'''*

*Returns Coordinates of the Node (Vector Object).*

*'''*

return *self*.coord

*'''STANDARD CALLS'''*

def **\_\_str\_\_**(*self*): # Print Statement

s = *"Node {} located at coordinates ({}, {})."*.format(*self*.ID,*self*.coord[0],*self*.coord[1])

return s

*'''INFORMATION CALLS'''*

def **getID**(*self*):

*'''*

*Returns Node ID.*

*'''*

return *self*.ID

def **getLineIDs**(*self*):

*'''*

*Returns Line ID List.*

*'''*

return *self*.lineIDs

*'''ACTION CALLS'''*

def **attach**(*self*, lineID):

*'''*

*Attaches Node to the Provided Line.*

*Adds Line ID to Internal Line ID List.*

*'''*

# If the Line Provided Is Not Attached

if lineID not in *self*.lineIDs:

*self*.lineIDs.append(lineID)

*self*.lineIDs.sort()

print *"Node {} attached to line {}."*.format(*self*.ID,lineID)

# If the Line Provided Is Attached

else:

print *"Error: Node {} already connected to line {}."*\

.format(*self*.ID,lineID)

def **detach**(*self*, lineID):

*'''*

*Detaches Node From the Provided Line.*

*Adds Line ID to Internal Line ID List.*

*'''*

# If the Node is Attached to At Least One Line

if len(*self*.lineIDs) > 0:

# If the Node is Attached to the Provided Line

if lineID in *self*.lineIDs:

*self*.lineIDs.remove(lineID)

print *"Node {} detached from line {}."*\

.format(*self*.ID,lineID)

# If the Node is Not Attached to the Provided Line

else:

print *"Error: Node {} is not connected to line {}."*.format(*self*.ID,lineID)

# If the Node is Not Attached to Any Lines

else:

print *"Error: Node {} is not connected to any lines."*.format(*self*.ID)

#### Path Class

*'''*

*Jean-Luc Jackson & Connor Lester*

*CEE 505 HW #4*

*11/07/16*

*'''*

class **Path**(object):

*'''*

*Class Hosts a Path Object.*

*Contained Attributes - Path ID, Length, and Node/Line Histories.*

*'''*

def **\_\_init\_\_**(*self*, nodeHistory=[], lineHistory=[], length=0.0):

*'''*

*CHANGE SO THAT passes node objects and line objects and can calculate length*

*by looping through lineHistory and summing up line.getLength() values.*

*affects Path() implementation in Graph class*

*'''*

# Initialize List of Nodes Visited - Contains Node Objects

*self*.nodeHistory = nodeHistory

# Initialize List of Lines Traveled - Contains Line Objects

*self*.lineHistory = lineHistory

# Initialize Length to Zero

*self*.length = float(length)

# Create pathID

s = *''*

for i in range(0,len(nodeHistory)):

s += str(nodeHistory[i].getID()) + *' -> '*

*self*.ID = *'( '* + s[:len(s) - 4] + *' )'*

*'''STANDARD CALLS'''*

def **\_\_str\_\_**(*self*): # Print Statement

s = *"Path {} has a length {} and has seen {} nodes and {} lines."*\

.format(*self*.ID,*self*.length, len(*self*.nodeHistory), len(*self*.lineHistory))

return s

def **\_\_add\_\_**(*self*,otherPath): # Add Error If Trying To Add Integer/Float

# Step Through Line - Add Line to lineHistory

newLineHistory = *self*.lineHistory + otherPath.getLineHistory()

# Step On Node - Add Node to nodeHistory

newNodeHistory = *self*.nodeHistory + otherPath.getNodeHistory()

# Increase Path's Length

newLength = *self*.length + otherPath.getLength()

return Path(newNodeHistory, newLineHistory, newLength)

def **\_\_sub\_\_**(*self*,otherPath): # Add Error If Trying To Subtract Integer/Float

# Step Back Through Line - Remove Line From lineHistory

newLineHistory = *self*.lineHistory

newNodeHistory = *self*.nodeHistory

for line in otherPath.getLineHistory():

newLineHistory = *self*.lineHistory

*self*.lineHistory.remove(line)

# Step On Node - Removed Node From nodeHistory

for node in otherPath.getNodeHistory():

newNodeHistory = *self*.nodeHistory

*self*.nodeHistory.remove(node)

# Decrease Path's Length

newLength = *self*.length - otherPath.getLength()

return Path(newNodeHistory, newLineHistory, newLength)

*'''INFORMATION CALLS'''*

def **getID**(*self*):

return *self*.ID

def **getLength**(*self*):

# Return the Current Length

return *self*.length

def **getNodeHistory**(*self*):

# Return Current nodeHistory

return *self*.nodeHistory

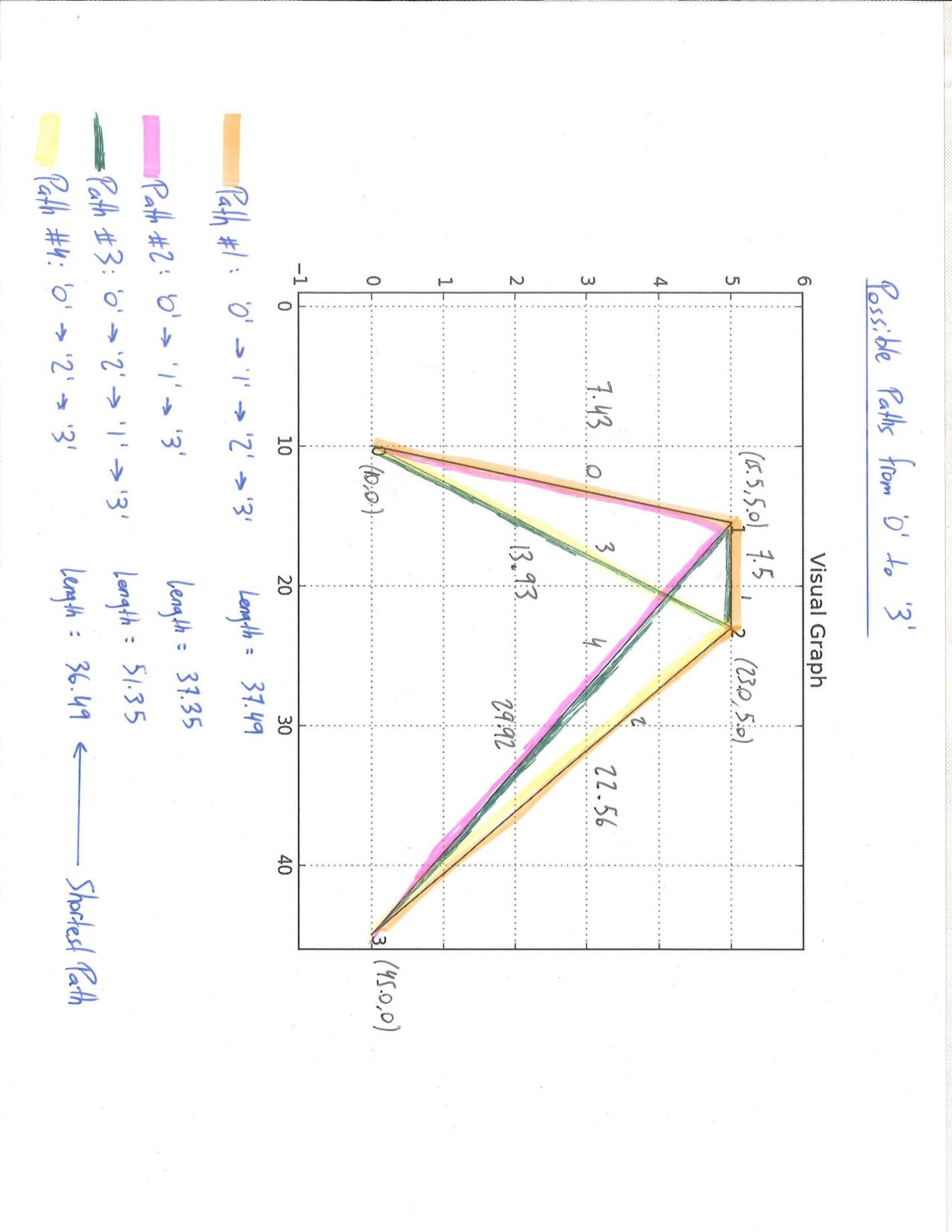
def **getLineHistory**(*self*):

# Return Current lineHistory

return *self*.lineHistory

### **Appendix 1**

#### By-Hand Inspection



#### Code Comparison

##### Code Input:

from Graph import \*

g = Graph(*'Graph1'*)

print *"\n"*

print *"Finding all paths from node '0' to '3'...\n"*

paths0to3 = g.findPaths(*'0'*, *'3'*)

paths = []

nodeHistories = []

lengths = []

for path in paths0to3:

nodeHistories = [node.getID() for node in path.getNodeHistory()]

print *"Path {}:"*.format(path.getID())

print *" NodeHistory = {}"*.format(nodeHistories)

print *" Length = {}"*.format(path.getLength())

shortest0to3 = g.findShortestPath(*'0'*, *'3'*)

print *"\n"*,shortest0to3

##### Code Output:

Node 0 attached to line 0.

Node 1 attached to line 0.

Node 1 attached to line 1.

Node 2 attached to line 1.

Node 2 attached to line 2.

Node 3 attached to line 2.

Node 0 attached to line 3.

Node 2 attached to line 3.

Node 1 attached to line 4.

Node 3 attached to line 4.

Finding all paths from node '0' to '3'...

Path ( 0 -> 1 -> 2 -> 3 ):

NodeHistory = ['0', '1', '2', '3']

Length = 37.494062719

Path ( 0 -> 1 -> 3 ):

NodeHistory = ['0', '1', '3']

Length = 37.3537629744

Path ( 0 -> 2 -> 1 -> 3 ):

NodeHistory = ['0', '2', '1', '3']

Length = 51.3491168779

Path ( 0 -> 2 -> 3 ):

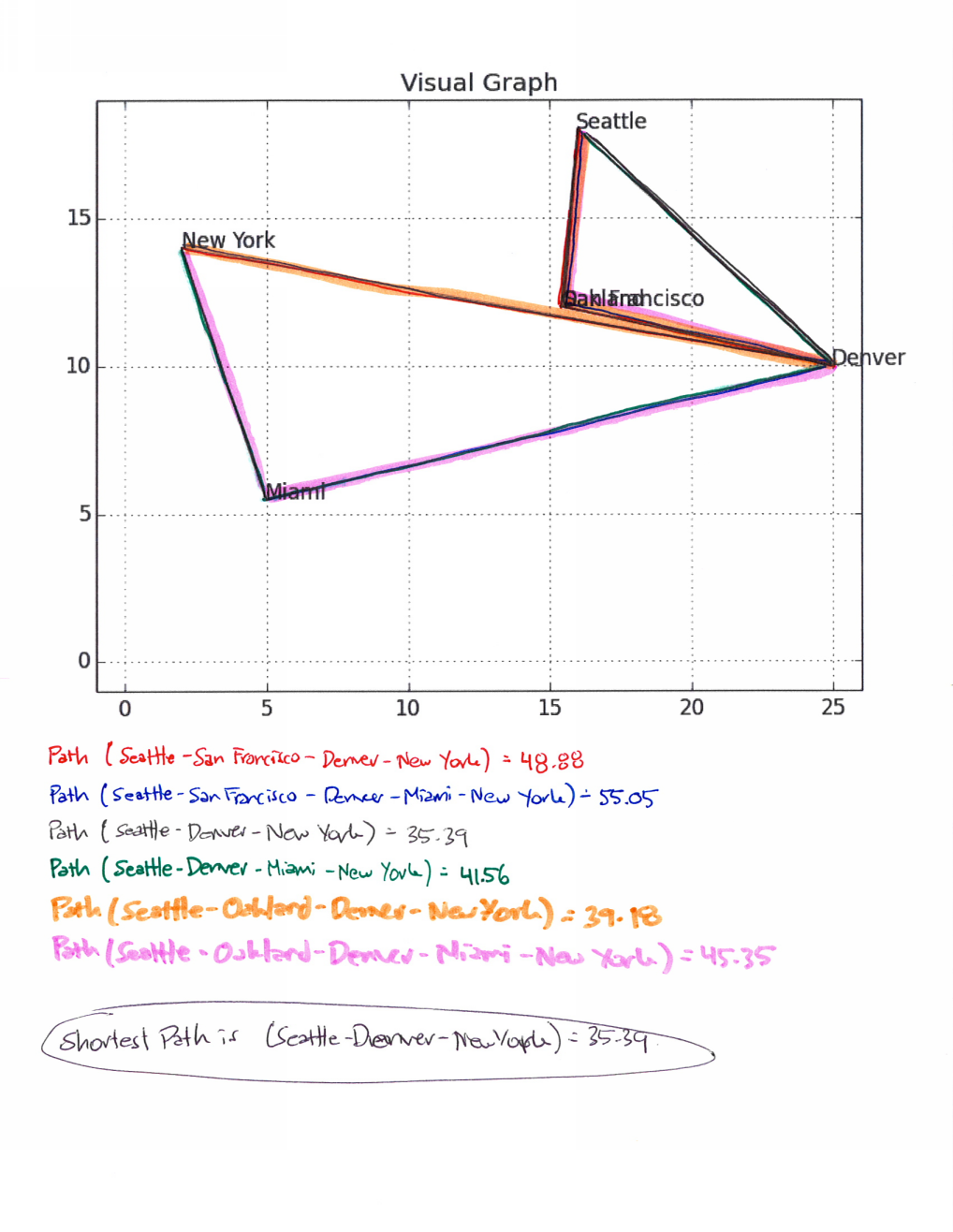
NodeHistory = ['0', '2', '3']

Length = 36.4894166225

The shortest path from 0 to 3 is ( 0 -> 2 -> 3 ) with a total length of 36.49.

### **Appendix 2**

#### By-Hand Inspection



#### Code Comparison

##### Code Input:

from Graph import \*

g = Graph(*'Graph2'*)

print *"\n"*

print *"Finding all paths from node 'Seattle' to 'New York'...\n"*

pathsSEAtoNY = g.findPaths(*'Seattle'*, *'New York'*)

paths = []

nodeHistories = []

lengths = []

for path in pathsSEAtoNY:

nodeHistories = [node.getID() for node in path.getNodeHistory()]

print *"Path {}:"*.format(path.getID())

print *" NodeHistory = {}"*.format(nodeHistories)

print *" Length = {}"*.format(path.getLength())

shortestSEAtoNY = g.findShortestPath(*'Seattle'*, *'New York'*)

print *"\n"*,shortestSEAtoNY

##### Code Output:

Node Seattle attached to line AK1.

Node San Francisco attached to line AK1.

Node Seattle attached to line AK2.

Node Denver attached to line AK2.

Node Oakland attached to line SWA1.

Node Denver attached to line SWA1.

Node Denver attached to line UA1.

Node New York attached to line UA1.

Node Denver attached to line UA2.

Node San Francisco attached to line UA2.

Node Denver attached to line UA3.

Node Miami attached to line UA3.

Node New York attached to line JB1.

Node Miami attached to line JB1.

Node Oakland attached to line SWA3.

Node Seattle attached to line SWA3.

Finding all paths from node 'Seattle' to 'New York'...

Path ( Seattle -> San Francisco -> Denver -> New York ):

NodeHistory = ['Seattle', 'San Francisco', 'Denver', 'New York']

Length = 48.8803968064

Path ( Seattle -> San Francisco -> Denver -> Miami -> New York ):

NodeHistory = ['Seattle', 'San Francisco', 'Denver', 'Miami', 'New York']

Length = 55.0490399352

Path ( Seattle -> Denver -> New York ):

NodeHistory = ['Seattle', 'Denver', 'New York']

Length = 35.3868296386

Path ( Seattle -> Denver -> Miami -> New York ):

NodeHistory = ['Seattle', 'Denver', 'Miami', 'New York']

Length = 41.5554727675

Path ( Seattle -> Oakland -> Denver -> New York ):

NodeHistory = ['Seattle', 'Oakland', 'Denver', 'New York']

Length = 39.1812809702

Path ( Seattle -> Oakland -> Denver -> Miami -> New York ):

NodeHistory = ['Seattle', 'Oakland', 'Denver', 'Miami', 'New York']

Length = 45.349924099

The shortest path from Seattle to New York is ( Seattle -> Denver -> New York ) with a total length of 35.39.