**DSA30AT PROJECT**

GROUP NAME :ANONYMOUS

**Stud no & Name**

218054528 Ngobeni L

219471122 Makalapetlo MC

216471504 Shiviti FV

QUESTION ON FIGURE 1

1.If no other routing information is given,how would you model or describe the problem of Figure 1?

Ans1.1: The connection of the network is shown by the graph where the set of vertices/nodes are in the position that can be differentiated to one another and are given different values. Capacities of the links are all 100 Mbps they are shown as edges connecting the network nodes that is corresponded to it. Those edges have at least one wiight that is representing cost,bandwidth and the delays.

1.2 Undirected weight graph G=(V,E)models a WAN network where

V=set of nodes/vertices on the network

E=set of link/edge between nodes/vertices

Wuv=weight of the link uv

The number of vertices edges is represented as |V| or |E| in G.The path that is in G is the sequence of network nodes ,where (v1,v2) is the link in G for1<= I <=K-1.The weight of path p,denotes as w(p),is the total of all edges in p.If no replacates verteces in p the path is not complicated .The path with the lowest w(p) has shortest pathe between vi and vj .wi,j denotes the weight ofshortest path between vi and vj in G.

2. What is the dandwidth demands z(m) in the figure 1 scenario?(2marks)

Ans: 5mbs

3.Deduce or infer the general formular that the traffic of every link at the time t in the figure 1 scenario,should satisfy.

P=A(P+Bp)

Questions on Figure 2:

1.Write Dijkstra’s algorithm in Python ver 3.8 or latest version, to find the shortest path from node H to node L. (15 marks)

2. Show comments in your Python code

Ans1&2:

import os

import sys

from PyQt5.QtCore import \*

from PyQt5.QtGui import \*

from PyQt5.uic import \*

from PyQt5.QtWidgets import \*

class Run(QMainWindow):

def \_\_init\_\_(self): #default constructor

QMainWindow.\_\_init\_\_(self) #inintial constructor

self.setFixedHeight(600) #width of the app

self.setFixedWidth(900) #height of the app

loadUi("net.ui", self) #load the design

self.isS, self.isD = True, False # allow the sorce node and the dest to be assign exchageable

self.run.clicked.connect(self.runApp) # add functionality to the button

self.btnA.clicked.connect(lambda: self.setNodes("A")) # add functionality to the button

self.btnB.clicked.connect(lambda: self.setNodes("B"))# add functionality to the button

self.btnC.clicked.connect(lambda: self.setNodes("C"))# add functionality to the button

self.btnD.clicked.connect(lambda: self.setNodes("D"))# add functionality to the button

self.btnF.clicked.connect(lambda: self.setNodes("F"))# add functionality to the button

self.btnG.clicked.connect(lambda: self.setNodes("G"))# add functionality to the button

self.btnH.clicked.connect(lambda: self.setNodes("H"))# add functionality to the button

self.btnI.clicked.connect(lambda: self.setNodes("I"))# add functionality to the button

self.btnJ.clicked.connect(lambda: self.setNodes("J"))# add functionality to the button

self.btnK.clicked.connect(lambda: self.setNodes("K"))# add functionality to the button

self.btnL.clicked.connect(lambda: self.setNodes("L"))# add functionality to the button

self.btnS.clicked.connect(lambda: self.setNodes("S"))# add functionality to the button

self.btnT.clicked.connect(lambda: self.setNodes("T"))# add functionality to the button

self.buttons = [self.btnA, self.btnB, self.btnC, self.btnD, self.btnF, self.btnG, self.btnH, self.btnI, self.btnJ, self.btnK, self.btnL, self.btnS, self.btnT]

self.validNodes = [] #create a list of valid node

for n in self.buttons: # assign the list using a loop

self.validNodes.append(n.text()) #append the node to the list

self.setUp() #call the method

self.show() #show the to the screeen

def runApp(self):

s = self.srcI.text() # the text that indicates the source node

d = self.destI.text()# the text that indicates the dest node

if(d == "NONE" or s == "NONE"): #validate that the nodes are assigned

self.pathI.setText("source node or destinatin node cannot be empty") #popup the error

else:

self.calculate(s, d) #call the calc method

def setNodes(self, n):

if(self.isS): #sorce is allow

for btn in self.buttons: #loop throught all the nodes

btn.setStyleSheet("color:black;background-color:transparent;")#make unvisited node black

if (btn.text() == self.destI.text()): #the dest

btn.setStyleSheet("color:lime;background-color:transparent;") #make the dest node lime

elif(btn.text() == n): #change the source node

btn.setStyleSheet("color:red;background-color:transparent;") #make the source node red

self.srcI.setText(n) # add the text to scr indicator

self.isS = False #disable the source assignment

self.isD = True #enable the dest node to be assigned

elif(self.isD): # dest now is allow

for btn in self.buttons:#loop throught all the nodes

btn.setStyleSheet("color:black;background-color:transparent;")#make unvisited node black

if (btn.text() == self.srcI.text()): #testing source

btn.setStyleSheet("color:red;background-color:transparent;")#make the source node red

elif(btn.text() == n): #change the dest node

btn.setStyleSheet("color:lime;background-color:transparent;") #set the color to lime of the dest node

self.destI.setText(n)# add the text to dest indicator

self.isS = True #allow the source

self.isD = False# disable the dest

def setUp(self): #create the method

self.init\_graph = {} # create a graph tuple

for node in self.validNodes: # loop over the nodes

self.init\_graph[node] = {} # adding node to a graph

# initilasing all the node edges

self.init\_graph["A"]["B"] = 3

self.init\_graph["A"]["D"] = 4

self.init\_graph["A"]["S"] = 7

self.init\_graph["B"]["A"] = 3

self.init\_graph["B"]["D"] = 4

self.init\_graph["B"]["H"] = 1

self.init\_graph["B"]["S"] = 2

self.init\_graph["C"]["L"] = 2

self.init\_graph["C"]["S"] = 3

self.init\_graph["D"]["A"] = 4

self.init\_graph["D"]["B"] = 4

self.init\_graph["D"]["F"] = 5

self.init\_graph["F"]["D"] = 5

self.init\_graph["F"]["H"] = 3

self.init\_graph["G"]["H"] = 2

self.init\_graph["G"]["T"] = 2

self.init\_graph["H"]["B"] = 1

self.init\_graph["H"]["F"] = 3

self.init\_graph["H"]["G"] = 2

self.init\_graph["I"]["J"] = 6

self.init\_graph["I"]["K"] = 4

self.init\_graph["I"]["L"] = 4

self.init\_graph["J"]["I"] = 6

self.init\_graph["J"]["K"] = 4

self.init\_graph["J"]["L"] = 4

self.init\_graph["K"]["I"] = 4

self.init\_graph["K"]["J"] = 4

self.init\_graph["K"]["T"] = 5

self.init\_graph["L"]["C"] = 2

self.init\_graph["L"]["I"] = 4

self.init\_graph["L"]["J"] = 4

self.init\_graph["S"]["A"] = 7

self.init\_graph["S"]["B"] = 2

self.init\_graph["S"]["C"] = 3

self.init\_graph["T"]["G"] = 2

self.init\_graph["T"]["K"] = 5

self.graph = self.getGraph() #assign the graph

def calculate(self, start\_node, target\_node): # this cals the algorithm and calculate the shortest path and the path it take

previous\_nodes, shortest\_path = self.dijkstra\_algorithm(start\_node=start\_node) # invoke the dijkstra\_algorithm method

path = [] # create the list of paths

node = target\_node # assign the variable node with the destination node

while node != start\_node: # loop until we reach the destination node

path.append(node) # add the node into the path list

node = previous\_nodes[node] # record the previous node and assign it no the node variable

path.append(start\_node) # Add the start node manually

text = " +-+ ".join(reversed(path)) # format the output of the path

self.costI.setText(f" cost = {str(shortest\_path[target\_node])}".upper())

self.pathI.setText(str(text).upper()) # add the output to the app

for node in self.buttons:#loop through all the nodes

if(node.text() == self.srcI.text() or node.text() == self.destI.text()):

pass

else:

node.setStyleSheet("color:black;background-color:transparent;")#make unvisited node black

if(node.text() in path): #check the node whether is within the path

if(node.text() == self.srcI.text() or node.text() == self.destI.text()):

pass

else:

node.setStyleSheet("color:cyan;background-color:transparent;")#make the in between nodes cyan

def dijkstra\_algorithm(self, start\_node):

unvisited\_nodes = list(self.validNodes) # create list of unvisited node

# We'll use this dict to save the cost of visiting each node and update it as we move along the graph

shortest\_path = {} # create the distionary of the shortest path

# We'll use this dict to save the shortest known path to a node found so far

previous\_nodes = {} # create the distionary of previous nodes

# We'll use max\_value to initialize the "infinity" value of the unvisited nodes

max\_value = sys.maxsize # assign infinity value

for node in unvisited\_nodes: # loop through all the unvisited nodes

shortest\_path[node] = max\_value # assign all the nodes shortest path to infinity

shortest\_path[start\_node] = 0 # However, we initialize the starting node's value with 0

while unvisited\_nodes: # The algorithm executes until we visit all nodes

# The code block below finds the node with the lowest score

current\_min\_node = None # create the current minimum node

for node in unvisited\_nodes: # Iterate over the nodes

if current\_min\_node == None: # test whether the current minimum node is not assign

current\_min\_node = node # assign the current minimumnode to the current node

elif shortest\_path[node] < shortest\_path[current\_min\_node]: # if is already assign compare the cost of the nodes

current\_min\_node = node # select the minimum path

# The code block below retrieves the current node's neighbors and updates their distances

neighbors = self.get\_outgoing\_edges(current\_min\_node) # get the neighbours of the current minimum node

for neighbor in neighbors: # loop over the neighbours

tentative\_value = shortest\_path[current\_min\_node] + self.graph[current\_min\_node][neighbor] # get the total cost/distance

if tentative\_value < shortest\_path[neighbor]: # check the total path with the neighbour path

shortest\_path[neighbor] = tentative\_value # update the shortest path of the neightbour

# We also update the best path to the current node

previous\_nodes[neighbor] = current\_min\_node # update the previous node of the neighbour

unvisited\_nodes.remove(current\_min\_node) # After visiting its neighbors, we mark the node as "visited"

return previous\_nodes, shortest\_path # return the result

def get\_outgoing\_edges(self, node):

# Returns the neighbors of a node.

connections = [] # an empty list of connections

for out\_node in self.validNodes: # loop through all nodes

if self.graph[node].get(out\_node, False) != False: # check if the node is the neighbour

connections.append(out\_node) # add the neighbour node to the list of connections

return connections # return the list of neighbours/connections

def getGraph(self):

graph = {} # create a empty dictionary of a graph

for node in self.validNodes: # loop through all the nodes

graph[node] = {} # create a empty dictionary for a node

graph.update(self.init\_graph) # undate the graph

for node, edges in graph.items(): # get all the node and all the adges and loop through all of them

for adjacent\_node, value in edges.items(): # from the edges get the cost/distance from the node to each adjacent node

if graph[adjacent\_node].get(node, False) == False: # chech the node adjacent

graph[adjacent\_node][node] = value # add the cost/distance /value between the node and the node neighbour

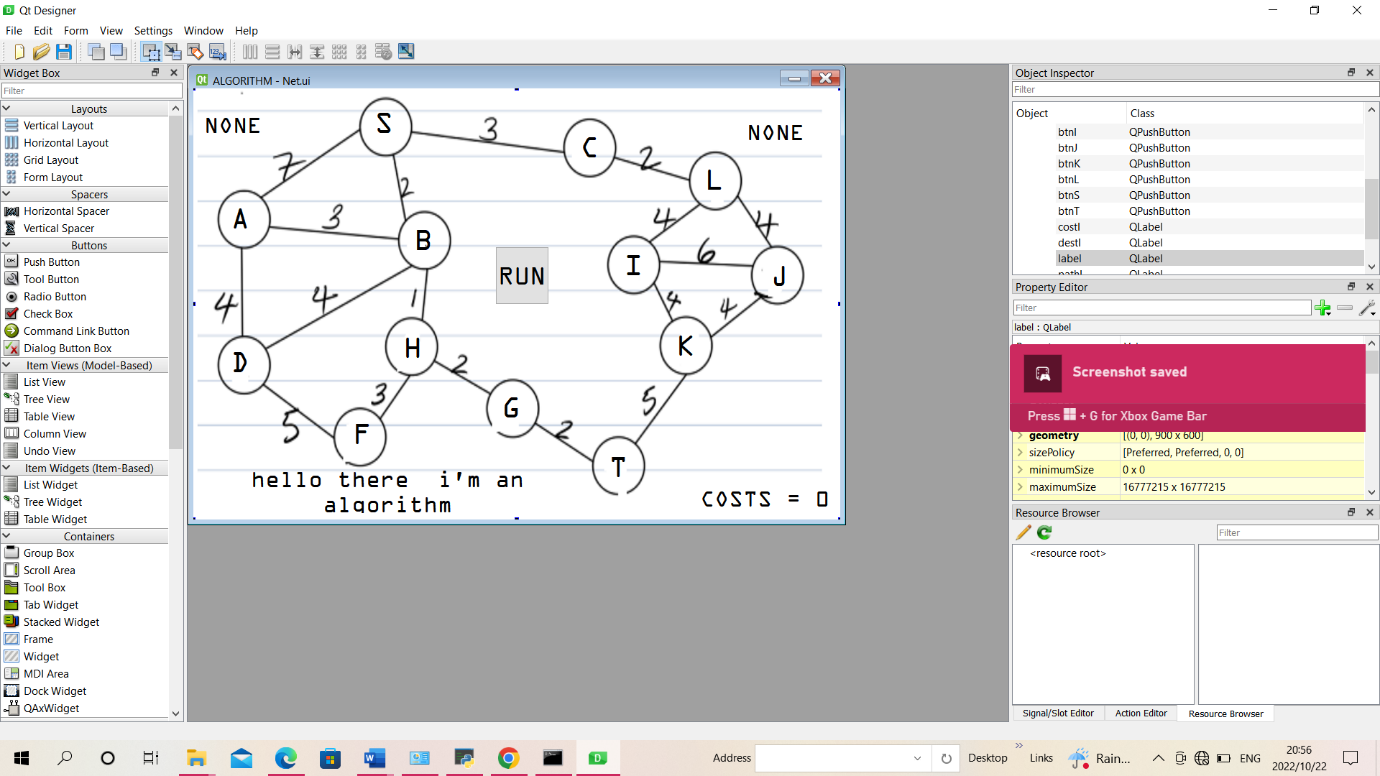
return graph # return the full updated graph

app = QApplication(sys.argv)

main = Run()

sys.exit(app.exec\_())

3. Hand-in a flip file folder with the Problem statement of this group project, a hardcopy of the Python source code, and output results; and answers to Questions on Figure 1.



The outputs

