gh

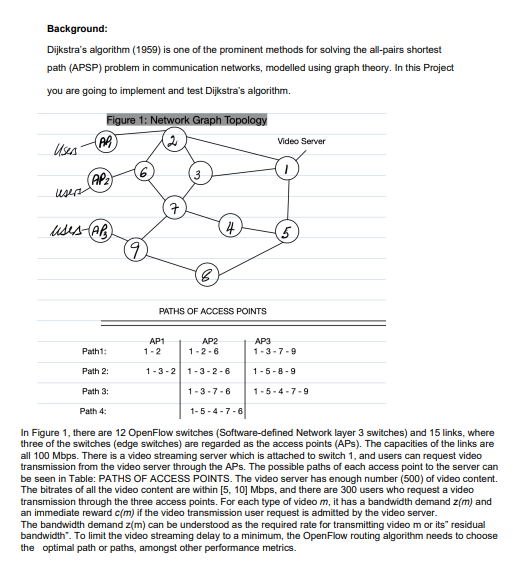
DSA30AT

DISTRIBUTED ALGORITHM PROJECT

22 April 2022

Member List

1. Simelane T 219230672
2. Zwane Q 218717667
3. Mtsweni BK 219886357
4. Thanjekwayo N 218104096
5. Mothiba LM 219365977

**QUESTION 1**

**Question 1 Answers**

1. If no other routing information is given, how would you model or describe the problem of Figure 1? (10 marks)

This network is conducted as a graph by locating each node to a unique vertex in the graph where links between network nodes are represented by edges connecting the corresponding vertices. Each edge can carry one or more weights; such weights may show cost, delay and bandwidth.

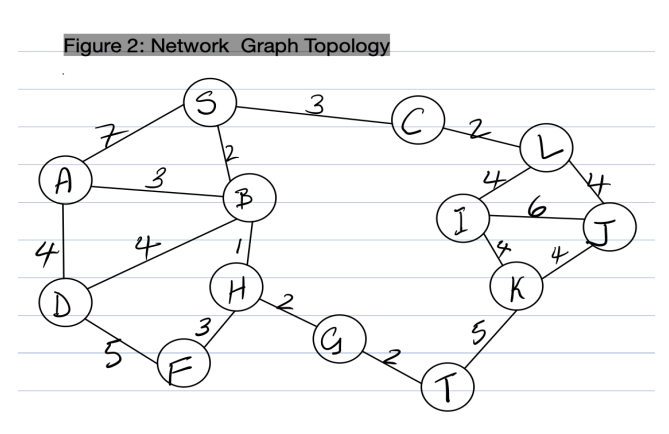
An undirected weighted graph is denoted as G (V, E, w), where V = {vi} is the set of vertices and E is the set of edges in G while w is a function that assigns a non-negative weight wij on every edge. The number of vertices edges is indicated as |V| or |E| in G. A path in G is a sequence of vertices, where every (v1, v2) is an edge in G for1 <= I <= k-1. The weight of path p, denoted as w (p), is the sum of the weights of all the edges in p. A path is simple if and only if there is no repeated vertex in p. The shortest path between vi and vj is a path with the minimum w (p) among all the paths between vi and vj. For simplicity, we use wi,j to denote the weight of the shortest path between vi and vj in G.

2. What is the bandwidth demand z(m) in the Figure 1 scenario? (2 marks)

5 Mbps

3. Deduce or infer the general formula that the traffic of every link at the time t in the Figure 1 scenario, should satisfy. (4 marks)

Bij/p(t) = min{bi l-m(t)}

**QUESTION 2**

**Questions on Figure 2:**

1. Write Dijkstra’s algorithm in Python ver 3.8 or latest version, to find the shortest path from node D to node J.

(15 marks)

2. Show comments in your Python code. (9 marks)

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. (10 marks)

**Question 2 Answers**

# import needed libraries for pyqt5 gui

import sys

from PyQt5.QtWidgets import \*

from PyQt5.uic import \*

class App(QMainWindow):

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

super(App, self).\_\_init\_\_()

self.setFixedSize(501, 354) # set the size of the gui interface

loadUi("app.ui", self)

self.route.setText("DSA30AT PROJECT APP")

self.nodes = ["A", "B", "C", "D", "F", "G", "H", "I", "J", "K", "L", "S", "T"] # create node list

self.done1 = False

self.done2 = False

self.source = ""

self.destination = ""

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

for node in self.nodes: # 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()

self.src.clicked.connect(self.setSrcNode)

self.dest.clicked.connect(self.setDestNode)

self.btn.clicked.connect(lambda: self.calculate(self.source, self.destination))

self.show()

def getGraph(self):

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

for node in self.nodes: # 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

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

# Returns the neighbors of a node.

connections = [] # an empty list of connections

for out\_node in self.nodes: # 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 calculate(self, start\_node, target\_node): # this cals the algorithm and calculate the shortest path and the path it take

try:

if self.source == "" or self.destination == "":

raise Exception

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)) + " = " + str(shortest\_path[target\_node]) # format the output of the path and costs

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

except KeyError:

self.route.setText(str(f"error invalid node selected[ {self.source},{self.destination} ]").upper())

except Exception:

self.route.setText(str("src and dest must be both selected").upper())

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

unvisited\_nodes = list(self.nodes) # 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 setSrcNode(self):

self.source, self.done1 = QInputDialog.getText(self, 'source', 'Enter the source node')

self.srcLbl.setText(str(f"SRC={self.source}").upper())

self.source = str(self.source).upper()

def setDestNode(self):

self.destination, self.done2 = QInputDialog.getText(self, 'destination', 'Enter the destination node')

self.destLbl.setText(str(f"SRC={self.destination}").upper())

self.destination = str(self.destination).upper()

app = QApplication(sys.argv)

cls = App()

app.exec\_()

