The folder called “Dijkstra” contains “londonconnections.csv”, “londonstations” and “londonlines.csv”, which all are the csv files that will be used to read the data about the London tube system in the program.

The files “londonModel.py”, “londonController.py” and “londonView” contain the actual program.

“londonModel.py” is where I created the two classes for the nodes modelled as stations and the other class called Graph, which will put all the stations as nodes to the graph. It also reads the csv file, which contains all the information required for a station.

“londonController.py” is the file, which makes Dijkstra’s algorithm come into action.

“londonView” is the file, which contains the tkinter monules and runs a short demo version of what the algorithm can do.

The final program and user interface will not look like this at the end; this is just a demonstration of it.

The folder “APIs” contains all of the API calls foe arrivals of all the lines.

The folder “network” contains a second version of the initial program. It is coded using the Python Jupyter Notebook, which uses the module Networkx, which makes a graph of all the stations and performs Dijkstra’s algorithm without me needing to implement the algorithm myself since Networkx has already implemented it inside the module. It uses the same files again to read all the required information for a station. This was inspired by [n2\_2\_weighted\_and\_directed\_graphs (staging--tsl-website.netlify.app)](https://staging--tsl-website.netlify.app/tf/60008_21/n2_2_weighted_and_directed_graphs.html) and [The London Tube as a Graph (markd.ie)](http://markd.ie/2016/04/10/The-London-Tube-as-a-Graph/). All of this second version makes my implementation of Dijkstra’s algorithm redundant since I will use the second version from now on to proceed with my technical solution.

The folder “status-checker” contains “status-checker.py” where I demonstrate a little version of what I will be implementing in my technical solution using TFL’s API. This program checks the status of all lines, disruptions and plans a journey.

|  |  |  |  |
| --- | --- | --- | --- |
| **Group** | **Technique** | **Evidence in code** | **Page in document** |
| **A** | * **Graph/Tree Traversal** * **Complex user-defined algorithms (eg optimisation, minimisation, scheduling, pattern matching) or equivalent difficulty** * **Dynamic generation of objects based on complex user-defined use of OOP model** * **Complex user-defined use of object-orientated programming (OOP) model, eg classes, inheritance, composition, polymorphism, interfaces** * **Recursive algorithms** | * **The Graph Class** * **findRoute – method in controller module** * **findRoute - method in GUI class** * **Dijkstra’s Algorithm** * **Creating objects to output the tkinter – app = GUI()** * **Inheritance in the GUI class from tk.Frame** * **In the “undergound” files to check status of lines and journey planner** | * **P.1** * **P.4, P.7** * **P.12** * **P.7** * **P.11** |
| **B** | * **Writing and reading from files** * **Dictionaries** | * **In the controller module reading csv files** * **E.g. having the stations from the csv file in a dictionary with the respective ID** | * **P.4, P.11** * **P.3** |

londonModel.py:

import csv

#Creating a class for all the nodes

class Vertex:

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

self.id = id

self.adjacent = {}

self.dist = float('inf')

self.previousVertex = False

self.path = {}

def \_\_str\_\_(self):

return str(self.id) + ' adjacent: ' + str([x.id for x in self.adjacent])

def addNeighbour(self, neighbour, weight=0):

self.adjacent[neighbour] = weight

def getConnections(self):

return self.adjacent.keys()

def getId(self):

return self.id

def getWeight(self, neighbour):

return self.adjacent[neighbour]

def setDist(self, dist):

self.dist = dist

def getDist(self):

return self.dist

def setPreviousVertex(self, vertex):

self.previousVertex = vertex

def getPreviousVertex(self):

return self.previousVertex

def setPath(self, key, path):

self.path[key] = path

def getPath(self, key):

return self.path[key]

#Creating a class for the graph of nodes(stations)

class Graph:

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

self.vertDict = {}

self.numVertices = 0

def \_\_iter\_\_(self):

return iter(self.vertDict.values())

def addVertex(self, id):

self.numVertices += 1

newVertex = Vertex(id)

self.vertDict[id] = newVertex

return newVertex

def getVertex(self, n):

if n in self.vertDict:

return self.vertDict[n]

else:

return None

def getVertices(self):

return self.vertDict.keys()

def addEdge(self, frm, to, cost=0):

if frm not in self.vertDict:

self.addVertex(frm)

if to not in self.vertDict:

self.addVertex(to)

self.vertDict[frm].addNeighbour(self.vertDict[to], cost)

self.vertDict[to].addNeighbour(self.vertDict[frm], cost)

#Read csv file with the data for stations, connections and lines

#Put all of this data into a dictionary

def readCSV(file):

currentLine = 0

output = {}

headers = []

#File is opened here and read - looping through each station in the csv

with open(file, 'r') as openedFile:

for line in csv.reader(openedFile):

currentWord = 0

#Looping through every word

for word in line:

if (currentLine == 0):

output[word] = []

headers.append(word)

else:

#Add station to the dictionary under the correct category

#{"station1": [1]}

output[headers[currentWord]].append(word)

currentWord += 1

currentLine += 1

output['headers'] = headers # -> titles in the dictionary

return output

#Connecting lines with IDs from the lines file with the connections file

def formatLines(lines):

formattedLines = {}

for x in range(len(lines[lines['headers'][0]])):

temp = {}

for y in range(1, len(lines['headers'])):

temp[lines['headers'][y]] = lines[lines['headers'][y]][x]

formattedLines[lines[lines['headers'][0]][x]] = temp

return formattedLines

#Connecting stations with IDs from stations file with the connections file

def formatStations(stations):

formattedStations = {}

for x in range(len(stations['id'])):

formattedStations[stations['id'][x]] = {

'id' : stations['id'][x],

'latitude' : stations['latitude'][x],

'longitude' : stations['longitude'][x],

'name' : stations['name'][x],

'display\_name' : stations['display\_name'][x],

'zone' : stations['zone'][x],

'total\_lines' : stations['total\_lines'][x],

'rail' : stations['rail'][x],

}

return formattedStations

londonController.py:

import londonModel

#Reading the files

lines = londonModel.formatLines(londonModel.readCSV('londonlines.csv'))

stations = londonModel.formatStations(londonModel.readCSV('londonstations.csv'))

connections = londonModel.readCSV('londonconnections.csv')

graph = None # -> The graph is a global variable

#Get all of the stations - a getter method

def getStations():

return stations

#Dijkstra's algorithm in action from the londonModel

def findFastestRoute(start, end, unavailableStations):

global graph

#Graph is creted and initialised

graph = londonModel.Graph()

response = validateStations(start, end, unavailableStations)#Checking inputs vor validity

if response:

return response

#Get ID From The File

start = getStationIdFromName(start)

end = getStationIdFromName(end)

#Stations are added to the graph

addStations(unavailableStations)

visited = []

unvisited = list(graph.getVertices())

#Current node has a distance of 0 from itself to itself

for vertex in graph:

if (start == vertex.getId()):

vertex.setDist(0)

start = vertex

nextNode = getNextNode(unvisited)

checkNeighbours(nextNode)

#Delete if station is unvisited

del unvisited[unvisited.index(nextNode.getId())]

visited.append(nextNode.getId())

#Rerurned to starting node

setPaths(start)

path = getPath(start, end)

#Return the path if the route is valid

if path != []:

return path

else:

#Otherwise output an error message

response = {

'error': True,

'msg': 'No possible route between stations'

}

return response

#Check chosen stations by the user are valid - if route exixts, if only start or end is chosen; or if start and destination are the same

#and output a respective message

def validateStations(start, end, unavailableStations):

response = {

'error': True,

'msg': ''

}

if (start == '' or end == ''):

response['msg'] = 'Please select a starting and ending station'

return response

if (start == end):

response['msg'] = 'Starting and ending stations are the same'

return response

if start in unavailableStations or end in unavailableStations:

response['msg'] = 'No possible route between stations'

return response

return None

#Get the ID number from the connections

def getStationIdFromName(target):

for id in stations:

if (stations[id]['name'] == target):

return id

#Stations are added to the graph in order to add them to the graph and the respective lsit. Check if both start and destiantion are in the unavailable list

#and add to the graph if not

def addStations(unavailable):

for x in range(len(connections['station1'])):

if (unavailable == None):

graph.addEdge(connections['station1'][x], connections['station2'][x], int(connections['time'][x]))

else:

if (stations[connections['station1'][x]]['name'] not in unavailable and stations[connections['station2'][x]]['name'] not in unavailable):

graph.addEdge(connections['station1'][x], connections['station2'][x], int(connections['time'][x]))

def getNextNode(unvisited):

nextNode = False

for vertex in graph:

if (vertex.getId() in set(unvisited)):

if (nextNode == False):

nextNode = vertex

if (vertex.getDist() < nextNode.getDist()):

nextNode = vertex

return nextNode

#See the weights of the adjacent neighbours and add to the current node

def checkNeighbours(currentNode):

neighbours = []

neighbours = currentNode.getConnections()

for vertex in neighbours:

tentativeDist = currentNode.getDist() + vertex.getWeight(currentNode)#-> Tentative list is the new distance and if it is less than the current distance

#to that node, update the shortest distance. If notm do not update

if (tentativeDist < vertex.getDist()):

vertex.setDist(tentativeDist)

vertex.setPreviousVertex(currentNode)

#Set path for every node in the graph from starting node

def setPaths(start):

for vertex in graph:

path = pathToVertex(vertex, [])

if vertex.getDist() != float('inf'):

vertex.setPath(start.getId(), path[::-1])

#Set the path to the node from the starting node

def pathToVertex(vertex, path):

if (vertex.getPreviousVertex()): # If this node has a previous node, add the current node to the path.If there is no previous node, return self.

path.append(vertex.getId())

pathToVertex(vertex.getPreviousVertex(), path)

else:

path.append(vertex.getId())

return path

#Path from start node to destination

def getPath(start, end):

vertex = graph.getVertex(end)

path = []

if vertex == None:

return path

if vertex.getDist() == float('inf'):# -> No possible route since the distance is infinte

return path

for x in vertex.getPath(start.getId()):

path.append(stations[x]['name'])

return path

londonView.py:

import tkinter as tk

import londonController

#Put the dictionary into a list of stations

def getStationList():

stations = []

stationList = londonController.getStations()

for x in stationList:

stations.append(stationList[x]['name'])

stations.sort()

return stations

#This will not the final project or the user interface. This is just a demo to show what the program does at this stage of working on it.

class GUI(tk.Frame):

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

tk.Frame.\_\_init\_\_(self)

self.stationList = getStationList()

self.startingStation = ''

self.endingStation = ''

self.unavailableStations = []

self.grid()

self.createWidgets()

def createWidgets(self):

self.errorLabel = tk.Label(self, text="", width=90) #Output for errors

self.errorLabel.grid(row = 0, column = 0, columnspan = 3)

#Stations in a list box for the user to pick from

self.stationLabel = tk.Label(self, text="Stations", width=6)

self.stationLabel.grid(row = 1, column = 0)

#Activate station list box

self.stations = tk.Listbox(self, width = 40, height = 5)

self.stations.grid(row = 2, column = 0)

#Add every and each station to that list box

for station in self.stationList:

self.stations.insert(tk.END, station)

#Output message for a station that is not available

self.unavailableStationLabel = tk.Label(self, text="Unavailable Stations", width=20)

self.unavailableStationLabel.grid(row = 1, column = 2)

#Activate and initialise

self.unavailableStationsListBox = tk.Listbox(self, width = 40, height = 5)

self.unavailableStationsListBox.grid(row = 2, column = 2)

#The button to choose a start station

self.startingStationButton = tk.Button(self, text = 'Set Start', command = self.setStart, width = 15, padx = 0)

self.startingStationButton.grid(row = 2, column = 1, sticky = tk.N)

#Initialising the start button

self.endingStationButton = tk.Button(self, text = 'Set End', command = self.setEnd, width = 15, padx = 0)

self.endingStationButton.grid(row = 2, column = 1)

#Button for destination station

self.setUnavailable = tk.Button(self, text = 'Set Status', command = self.changeStatus, width = 15, padx = 0)

self.setUnavailable.grid(row = 2, column = 1, sticky = tk.S)

#Initialising the butoon for destiantion station

self.routeLabel = tk.Label(self, text="", width=50)

self.routeLabel.grid(row = 3, column = 0, columnspan = 2, sticky = tk.W)

#Button to confirm

self.confirmRoute = tk.Button(self, text = 'Confirm Route', command = self.findRoute, width = 15)

self.confirmRoute.grid(row = 3, column = 2, sticky = tk.E)

#Initialise button

self.routeText = tk.Text(self, height = 5, width = 80)

self.routeText.grid(row = 4, column = 0, columnspan = 3, padx = 5)

#Make the start start button command

def setStart(self):

index = self.stations.curselection()[0]

self.startingStation = self.stations.get(index)

if self.endingStation != '':

updatedText = "From %s to %s" % (self.startingStation, self.endingStation)

else:

updatedText = "From %s" % (self.startingStation)

self.routeLabel.config(text = updatedText)

#Commnd for the destination button

def setEnd(self):

index = self.stations.curselection()[0]

self.endingStation = self.stations.get(index)

if self.startingStation != '':

updatedText = "From %s to %s" % (self.startingStation, self.endingStation)

else:

updatedText = "%s to %s" % (self.routeLabel['text'], self.endingStation)

self.routeLabel.config(text = updatedText)

#Change the occupation of a station - if in stations list, move to unavailbale and vice versa

def changeStatus(self):

if (self.stations.curselection()):

station = self.changeStationStatus(self.stations, self.unavailableStationsListBox)

self.unavailableStations.append(station)

if (self.unavailableStationsListBox.curselection()):

station = self.changeStationStatus(self.unavailableStationsListBox, self.stations)

self.unavailableStations.remove(station)

#A temporary list of all active stations that are in the list box, clear current list of stations in the taregt, insert into respective list box

#remove from previous list box and return the moved station

def changeStationStatus(self, moveFrom, moveTo):

index = moveFrom.curselection()[0]

station = moveFrom.get(index)

tempList = list(moveTo.get(0, tk.END))

tempList.append(station)

tempList.sort()

moveTo.delete(0, tk.END)

for item in tempList:

moveTo.insert(tk.END, item)

moveFrom.delete(index)

return station

#Dijkstra in action imported from the other module, if route found, output it, if not, error message is outputted.

def findRoute(self):

route = londonController.findFastestRoute(self.startingStation, self.endingStation, self.unavailableStations)

if isinstance(route, list):

self.setRouteText(route)

self.displayResponse({'error': False, 'msg': 'found'})

else:

self.setRouteText(None)

self.displayResponse(route)

#Output the route, count how many stations are in the list , append to the output list. Dont add "-" if the station is last.

def setRouteText(self, route):

self.routeText.delete(1.0, tk.END)

if route:

stationCount = len(route)

self.routeText.insert(tk.END, "Suggested Route: ")

for station in route:

if str(station) == str(route[stationCount-1]):

station = "%s" % station

else:

station = "%s - " % station

self.routeText.insert(tk.END, station)

#Output messages respectively

def displayResponse(self, response):

if response:

background = 'Green'

if response['error']:

background = 'Red'

self.errorLabel.config(background=background, text=response['msg'])

#Create object to output the demo

def initiateView():

app = GUI()

app.master.title('London tube')

app.mainloop()

initiateView()

Status-checker.py:

import requests

import json

import datetime

appId = '0d3ca992cf024a1b933ad36e42ca42d9'

appKey = '7cb4785379de41e5a653e984990bb861'

def api():

try:

stationStart = input("Please enter the station you want to travel from ")

print(f"You have selected {stationStart}")

destination = input("Please enter your destination station ")

print(f"You have selected{destination}")

print("Now fetching route, please wait...")

getRequest = requests.get(f"https://api.tfl.gov.uk/journey/journeyresults/{stationStart}/to/{destination}&app\_id={appId}&app\_key={appKey}")

data = getRequest.json()

start = data['fromLocationDisambiguation']['disambiguationOptions'][0]['parameterValue']

end = data['toLocationDisambiguation']['disambiguationOptions'][0]['parameterValue']

journey = requests.get(f"https://api.tfl.gov.uk/journey/journeyresults/{start}/to/{end}&app\_id={appId}&app\_key={appKey}")

journeyJson = journey.json()

journeyTime = journeyJson['journeys'][0]['duration']

print(f"Journey is {journeyTime} minutes")

print("This is the journey...\n")

route = journeyJson['journeys'][0]['legs']

for information in route:

print("From ", information['arrivalPoint']['commonName'])

print(information['instruction']['detailed'])

print("Arrive at ", information['arrivalPoint']['commonName'])

disruptions = information['isDisrupted']

if disruptions == True:

for disruption in information['disruptions']:

print("\n=======Disruption:=======")

print(disruption['description'])

else:

print("No disruptions available\n")

print("\n")

except KeyError as e:

print("Error (KeyError)! Please try again!")

api()

Version2.py

import colorsys

from multiprocessing import connection

import pandas as pd

import networkx as nx

import matplotlib.pyplot as plt

lines = pd.read\_csv("london.lines.csv", index\_col=0)

stations = pd.read\_csv("underground\_stations.csv", index\_col=0)

connections = pd.read\_csv("london.connections.csv")

stations.drop('display\_name', axis=1, inplace=True)

graph = nx.Graph()

for connection\_id, stationID in connections.iterrows():

lineName = lines.loc[stationID["line"]]['name']

station1\_name = stations.loc[stationID["station1"]]["name"]

station2\_name = stations.loc[stationID["station2"]]["name"]

node1 = (station1\_name, lineName)

node2 = (station2\_name, lineName)

graph.add\_edge(node1, node2, time=stationID["time"])

graph.add\_edge(node1, station1\_name, time=1)

graph.add\_edge(node2, station2\_name, time=1)

for neighbor in graph.neighbors(station1\_name):

graph.add\_edge(node1, neighbor, time = 0.5)

for neighbor in graph.neighbors(station2\_name):

graph.add\_edge(node2, neighbor, time = 0.5)

graph.add\_edge('Bank', 'Monument', time = 1)

nx.shortest\_path(graph, 'Oxford Circus', 'Turnpike Lane', weight='time')