|  |  |
| --- | --- |
| EFREI 2024 – LSI 1  Projet « Métro, Boulot, Dodo » |  |
| **Rapport Théorie des Graphes** |  |



**Par Cérine RAHMANI, Rihem BOUTEMINE, Bushra HOSSAIN, Anis MOSBAHI**

Table des matières

[**I- Introduction 3**](#_Toc149510547)

[**II- Modification du fichier metro.txt & Connexité 4**](#_Toc149510548)

[**III- Structures de données utilisées (explication du graphe) 6**](#_Toc149510549)

[**IV- Algorithme Bellman (explication du code) 9**](#_Toc149510550)

[**V- Algorithme PRIM (explication du code) 12**](#_Toc149510552)

[**VII- Code- annexe 15**](#_Toc149510553)

# Introduction

Paris est chaque jours parcourue par des millions de parisiens, de visiteurs, de salariés et de toute sortes d’individus souhaitant se rendre d’un point A à un point B. Il a donc été nécessaire de développer un réseau de transport en commun poussé dans son fonctionnement permettant des déplacements simples et rapides : le métro. C’est dans ce contexte que nous essayerons, à travers notre projet Graph Explorer, d’utiliser nos connaissances en théorie des graphes pour explorer le métro parisien.

Ce projet, développé en Python, nous amènera à appliquer et développer des concepts vus en cours pour résoudre différents problèmes dont les habitants de la capitale sont confrontés au quotidien.

Comment trouver le chemin le plus court entre deux stations ? Peut-on atteindre toutes les stations à partir de chacune des stations ? Comment rejoindre une station le plus rapidement possible, et comment optimiser ce temps ?

Pour répondre à ces questions essentiels, cœurs de notre projet, nous analyserons et apporterons les modifications nécessaires aux fichiers sur les stations et les lignes afin de les représenter sous la forme d’un graphe ; nous vérifierons la connexité de ce dernier.

Nous verrons ensuite quelles structures de données nous avons utilisé pour traiter le graphe, en expliquant comment nous avons représenté et traité le graphe à l’aide de python.

Nous finirons par les algorithmes de Bellman et Prim qui permettent d’optimiser le temps des trajets, puis nous conclurons.

# Modification du fichier metro.txt & Connexité

1. **Analyse du fichier metro.txt**

Pour commencer le graphe, nous avons analysé le fichier ***« metro.txt » .*** Celui-ci contient des informations sur les stations et lignes.

Après analyse, nous constatons qu’il existe plusieurs correspondances entre différentes stations et lignes ou depuis des terminus car certaines stations apparaissent plusieurs fois dans le « metro.txt » comme par exemple « Chatelêt ». On doit donc vérifier à partir de là, ***la connexité.***

Une image contenant texte, Police, capture d’écran, typographie

Description générée automatiquement

1. **Vérification de la connexité du Graphe**

L’un des aspects primordial est de savoir si on peut aller d’un sommet, c’est-à-dire d’une station à une autre à savoir, si le graphe est bien connexe ou non. Ainsi, nous avons dû développer un module qui vérifie cette condition basé sur l’algorithme du parcours en largeur pour explorer les différentes stations de métro à partir d’un sommet.

Nous avons implémenté dans le module ***« Graph »*** la fonction ***« is\_connected »*** :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

1. **Explication de l’algorithme utilisé**

La fonction ***is\_connected*** comme par choisir une station de métro à partir du fichier metro.txt. On crée une queue, soit une file d’attente en ajoutant cette station et on explore tous les stations accessibles depuis ce point en les ajoutant à notre file jusqu’à avoir exploré toutes les stations du réseau.

On remarque donc que le graphe sera bien connexe si l’algorithme explore toutes les stations, sinon nous devons ajouter des jointures si des stations sont innaccessible depuis notre point de départ.

Ainsi, si la fonction ***is\_connected*** retourne **« True »,** le graphe est connexte. Sinon, si ça retourne **« False »** cela signifie qu’il manque des liaisons entre les sommets.

Dans le cas du **False,** on remarque que le programme assure la connexité en créant des liaisons entre les stations sans arrêtes.

# Structures de données utilisées (explication du graphe)

Pour ce projet en python, nous avons utiliser deux types de structures de données :

* Un dictionnaire où se trouve les stations de métro
* Une liste (dictionnaires imbriqués)
* Classe

1. **Dictionnaire pour les stations**

Voici les éléments composant le dictionnaire :

èLa Clé : numéro du sommet (l’ID)

èUne liste avec les données sont structurés comme tels :

Une image contenant texte, Police, Graphique, capture d’écran

Description générée automatiquement

* **V (Sommet)** – **Station ID** – **Nom de la station** – **Numéro de ligne** – **Un booléen** si c’est un terminus ou non avec TRUE/FALSE – **Un indicateur de branchement** avec comme valeur 0, 1 ou 2…. Pour une station commune

1. **Dictionnaire pour les arêtes**

Voici les éléments composant le dictionnaire :

èLa Clé : Un couple de sommets liés par une arête : Sommet 1 (départ) et Sommet 2 (arrivée)

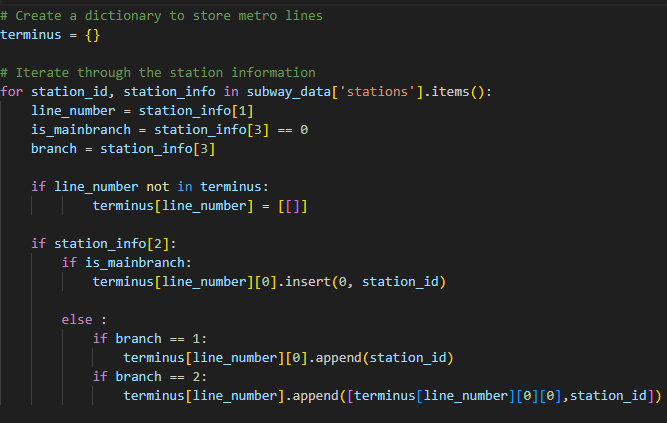
èLe temps en secondes pour aller d’un sommet à l’autre lié par une arête



* **E(Arête)**  – **Sommet 1** – **Sommet 2**- **poids** entre les deux arrêtes (soit le temps en secondes)

1. **Dictionnaire ‘terminus’**

Dans le graphe, chaque station est un nœud du graphe et les arêtes représentent les jointures entre les stations. Ainsi ce dictionnaire sert à stocker les informations des lignes de métro et des terminus. Voici un exemple de notre code :



Ainsi, cette structure de donnée permet également de faciliter la recherche des terminus de chaque ligne de métro et vérifier la connexité.

1. **Classe « Graph »**

Cette classe en python sert à crée le graphe du réseau métro. Elle est composée du dictionnaire ***« subway\_data »***

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

èPermet de faire une liaison entre les stations en créeant des arêtes et stocker les poids.

Cette classe permet donc :

* De verifier la connexité
* Calculer les itinéraires
* Appliquer l’algorithme de Bellman et Prim pour trouver un arbre couvrant de poids minimal.

|  |  |
| --- | --- |
|  | Algorithme Bellman (explication du code)L'idée est d'utiliser le [Algorithme de Bellman-Ford](https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm) pour calculer les chemins les plus courts d'un seul sommet source à tous les autres sommets dans un digraphe pondéré donné. L'algorithme initialise la distance à la source pour 0 et tous les autres nœuds à INFINITY. Ensuite, pour tous les bords, si la distance jusqu'à la destinatison peut être raccourcie en prenant le bord, la distance est mise à jour à la nouvelle valeur inférieure. A chaque itération i que les bords sont balayés, l'algorithme trouve tous les chemins les plus courts d'au plus longueur i bords. Puisque le chemin le plus long possible sans cycle peut être V-1 bords, les bords doivent être scannés V-1 fois pour s'assurer que le chemin le plus court a été trouvé pour tous les nœuds.  Voici le pseudocode de Bellman-Ford et son équivalent sur le code Python de notre application :  fonction Bellman\_Ford(liste de sommets, liste d'arêtes, sommet source, distance[], parent[], dest)    /\* Étape 1 - Initialiser le graphe. Au début, tous les poids des arêtes sont fixés à l'infini, le parent est initialisé à nul, et la source (le début) est mis à 0. \*/    pour chaque sommet v dans liste de sommets  distance[v] = INFINI  parent[v] = NULL  distance[source] = 0  Insertion de l’image...    // Étape 2 - relâcher les sommets de manière récursive  pour i de 1 à V-1  // V - Nombre d'arêtes pour chaque sommet (u, v) avec le poids w.  si (distance[u] + w) est inférieur à distance[v] alors  distance[v] = distance[u] + w  parent[v] = u    // Étape 3 - Detection du plus court chemin et le stocker dans une liste  Tantque dest est different de Null  Stocker la destination dans pcc[] ainsi que les parents de la destination dans la destination (on a utilisé un dictionnaire et une liste imbriquée)  retourner distance[], pcc[] |
|  |

# Algorithme PRIM (explication du code)

Comme l’algorithme de Kruskal, ou encore Djikstra, l’algorithme de Prim est également un algorithme glouton. Cela commence par un arbre couvrant vide. L’idée est de conserver deux ensembles de sommets. Le premier groupe contient des sommets qui ont été inclus dans Minimum spaning tree (l’arbre mininmal couvrant) et l’autre groupe contient des sommets qui ne sont pas encore inclus.

À chaque étape, il considérera tous sommets déjà parcouru et l’arête avec le poids le plus faible parmi ces arêtes. Il vérifie notamment que l’arrête ajouter ne crée pas un cycle dans notre arbre existent. Après avoir sélectionné l’arête, il déplace l’autre extrémité de l’arête vers l’ensemble contenant l’arête.  
L’idée de l’algorithme de Prim est très simple. Un arbre couvrant signifie que tous les sommets doivent être connectés.

Voici le pseudocode de Prim et son équivalent sur le code Python de notre application :

Fonction Prim(self, start\_node, list\_vertices=[], tree=[]):

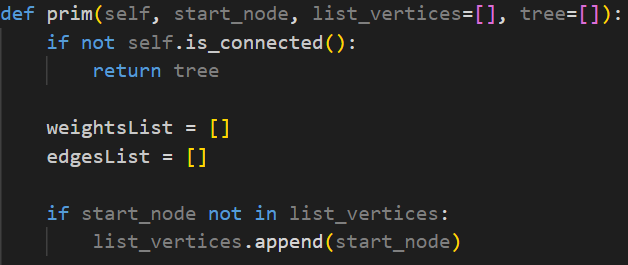
Si le graphe n'est pas connecté:

Retourner l'arbre actuel (tree)

Créer une file de priorité F contenant les sommets de G avec cout[.] comme priorité

Si start\_node n'est pas dans list\_vertices:

Ajouter start\_node à list\_vertices



// Étape 2 - Création d'une file de priorité pour extraire les sommets en fonction de leur coûts

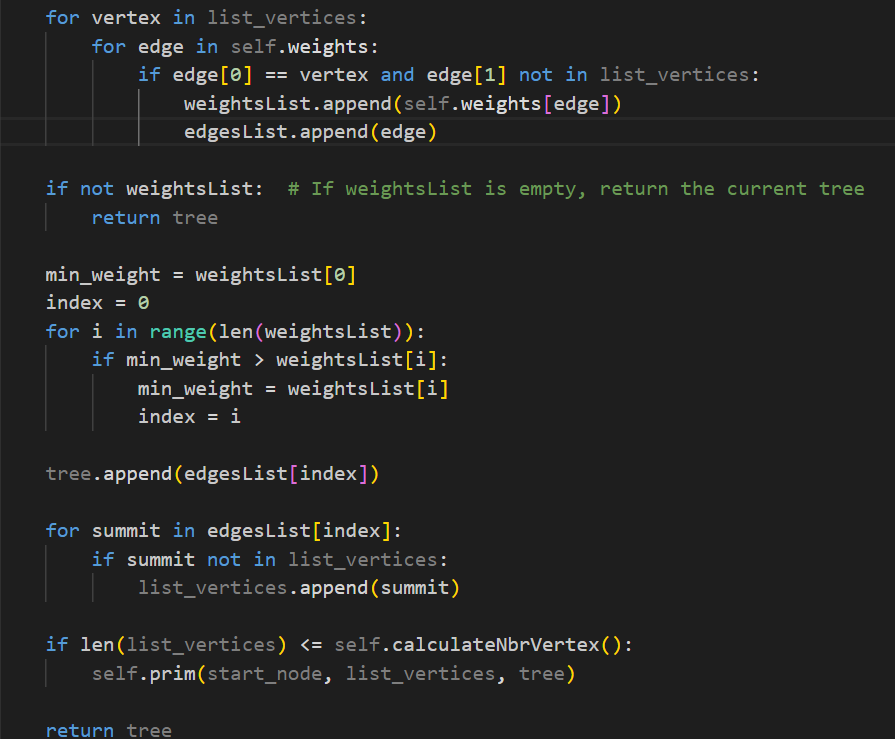
Tant que F n'est pas vide:

T := F.extraire\_le\_sommet\_avec\_coût\_minimum() // Extrait le sommet T de F

Pour chaque arête (T--U) dans G:

Si U est dans F et le poids de l'arête est inférieur ou égal à cout[U]:

pred[U] := T // Met à jour le prédécesseur de U Mettre à jour le coût de U dans F



/\* Une fois que la boucle se termine, on aura construit l'arborescence couvrante de poids minimum et le tableau pred contiendra l’arborescence finale.\*/  
   
 **retourner** l’arbre final (tree)

1. **Conclusion**

Ce projet nous a permis d’appliquer plusieurs aspects clés de la théorie des graphes a un cas réel. Nous avons pu voir comment des algorithmes tels que Bellman et Prim peuvent être utilisés pour résoudre des problèmes spécifiques au réseau de transports en commun, comme la recherche du plus court itinéraire ou l’optimisation de l’extension du réseau, et comment les graphes peuvent offrir des solutions pratiques pour améliorer la planification, la gestion et l’efficacité des systèmes de transport en commun.

En développant le sujet et en approfondissant l’application de la théorie des graphes dans le domaine des transports, en prenant en compte les contraintes spécifiques aux transports en commun, on pourrait développer une réelle application comme City mapper ou RATP mais qui se baserait entièrement sur la théorie des graphes.

Les graphes offrent des outils puissants pour la résolution de problèmes spécifiques, l’informatique, la biologie ou encore la géographie ne représentent qu’un échantillon de tous les domaines où on y trouve une application.

# Code- annexe

**DATA.PY**

line\_colors = {

'1': '#F6D046',

'2': '#2962AB',

'3': '#9F993B',

'3bis': '#A4D2E0',

'4': '#B24A8E',

'5': '#E49352',

'6': '#90C195',

'7': '#E8A7BA',

'7bis': '#90C195',

'8': '#9F993B',

'9': '#BAB33A',

'10': '#DBB448',

'11': '#866034',

'12': '#387F53',

'13': '#A4D2E0',

'14': '#5E287E'

}

*# subway\_data = {*

*# 'stations' : {*

*# 'numstation : [*

*# station name,*

*# ligne*

*# terminus Oui : 1 / Non : 0*

*# branchement principale : 0, 1er branche : 1, 2e branche : 2],*

*# 'join' : {*

*# 's1': [],*

*# 's2': [],*

*# 'lon': []*

*# }*

*# }*

subway\_data = {

'stations' : {},

'join' : {

'summit1': [],

'summit2': [],

'lon': []

}

}

*# -------- 1) Description du stockage des données ------------*

*# format pour les sommets :*

*# V num\_sommet nom\_sommet numéro\_ligne si\_terminus branchement (0 stations en commun, 1 pour la direction 1, 2 pour la direction 2, ainsi*

*# de suite ...)*

*# format pour les arêtes :*

*# E num\_sommet1 num\_sommet2 temps\_en\_secondes*

*# récupérer les diférentes lignes et branches*

with open('metro.txt', 'r', encoding="UTF-8") as metro:

for line in metro:

if line[0] == 'V':

*# station id -> station name,*

*# ligne*

*# terminus True or False*

*# branchement principale : 0, 1er branche : 1, 2e branche : 2*

subway\_data['stations'][int(line.split()[1])] = [' '.join(line.split()[2:-3]),

line.split()[-3][1:],

False if line.split()[-2][1:] == 'False' else True,

0 if line.split()[-1]== '0' else 1 if line.split()[-1] == '1' else 2]

if line[0] == 'E':

subway\_data['join']['summit1'].append(int(line.split()[1:][0]))

subway\_data['join']['summit2'].append(int(line.split()[1:][1]))

subway\_data['join']['lon'].append(int(line.split()[1:][2]))

*# Create a dictionary to store metro lines*

terminus = {}

for station\_id, station\_info in subway\_data['stations'].items():

line\_number = station\_info[1]

is\_mainbranch = station\_info[3] == 0

branch = station\_info[3]

if line\_number not in terminus:

terminus[line\_number] = [[]]

if station\_info[2]:

if is\_mainbranch:

terminus[line\_number][0].insert(0, station\_id)

else :

if branch == 1:

terminus[line\_number][0].append(station\_id)

if branch == 2:

terminus[line\_number].append([terminus[line\_number][0][0],station\_id])

*# Create a dictionary to store metro's positions*

position = {}

with open('pospoints.txt', 'r', encoding='utf-8') as file:

for line in file:

line = line.strip().split(';')

if len(line) == 3:

x\_coord, y\_coord, station\_name = line

x\_coord = int(x\_coord)

y\_coord = int(y\_coord)

position[(x\_coord, y\_coord)] = int(station\_name) *# Replace @ with spaces*

*# Create an array to store metro's lines*

*# Reconstitution des lignes de métro*

from graph import Graph

graph = Graph()

lines = []

for line in terminus:

temp\_lines = [line]

for branch in terminus[line]:

temp = graph.bellman\_ford(branch[0], branch[1])

temp\_lines.append(temp[1])

lines.append(temp\_lines)

**GRAPH.PY**

from collections import defaultdict, deque

from data import subway\_data

*# Création du graphe où on ajoute les arrêtes et les poids*

class Graph():

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

*# Create a graph instance*

*# struct of graph edges :{0: [238, 159], 238: [0, 322, 239], .....}*

*# struct of graph weight = {(s1, S2) : w1, ... , (sk, sn) : wp }*

*self*.edges = defaultdict(list)

*self*.weights = {}

*self*.add\_edge(subway\_data)

def add\_edge(self, subway\_data):

for i in range(len(subway\_data['join']['summit1'])):

from\_node = subway\_data['join']['summit1'][i]

to\_node = subway\_data['join']['summit2'][i]

weight = subway\_data['join']['lon'][i]

*self*.edges[from\_node].append(to\_node)

*self*.edges[to\_node].append(from\_node)

*self*.weights[(from\_node, to\_node)] = weight

*self*.weights[(to\_node, from\_node)] = weight

def calculateNbrVertex(self):

max\_vertex = max(*self*.edges.keys())

return max\_vertex

def is\_connected(self):

if not *self*.edges:

return False

starter = list(*self*.edges.keys())[0]

visited = set()

queue = deque([starter])

while queue:

sommet = queue.popleft()

visited.add(sommet)

for neighbor in *self*.edges[sommet]:

if neighbor not in visited and neighbor not in queue:

queue.append(neighbor)

return len(visited) == len(*self*.edges)

*# Bellman-Ford*

*# En entrée : - G=(V,E) graphe orienté pondéré et*

*# - r un sommet de G*

*# En sortie : Pour chaque sommet u de G un chemin de poids minimal de r vers u*

*# Step 1 Initialisations*

*# step 2 Pour i:= 1 à n-1 /\* n = nbre de sommets \*/*

*# Relacher tous les arcs de G*

*# Step 3 Vérifier qu'il n'y a pas de circuit négatif*

*#Initialisation (step 1)*

*# Mettre une etiquette inf sur tous les sommets et une etiquette 0 sur r*

*# Relâchement(u,v)(step 2)*

*# Si d[v] > d[u] + w(u,v) alors*

*# d[v] := d[u] + w(u,v)*

*# Parent[v] := u*

def bellman\_ford(self, start\_node, destination):

*# Initialize distances and parents*

*self*.d = {node: float('inf') for node in *self*.edges}

*self*.d[start\_node] = 0

parents = {node: None for node in *self*.edges}

*#ONE TO ALL*

*# Number of vertices*

nbrVertex = *self*.calculateNbrVertex()

for i in range(1, nbrVertex - 1):

for from\_node, to\_nodes in *self*.edges.items():

for to\_node in to\_nodes:

if *self*.d[to\_node] > *self*.d[from\_node] + *self*.weights[(from\_node, to\_node)]:

*self*.d[to\_node] = *self*.d[from\_node] + *self*.weights[(from\_node, to\_node)]

parents[to\_node] = from\_node *# Update the parent*

*# RETRIEVE PATH ONE TO ONE*

shortest\_path = []

while destination is not None:

shortest\_path.insert(0, destination)

destination = parents[destination]

return (*self*.d, shortest\_path)

def calculateTime(self, start\_node, destination):

temp = *self*.bellman\_ford(start\_node, destination)

for key in temp[0]:

if destination == key:

return int(temp[0][key]/60)

*# A partir du chemin récupéré, je reconstitue les transferts avec les lignes*

def get\_transfert(self, start\_node, destination):

temp = *self*.bellman\_ford(start\_node, destination)

transfert = {}

for ID in temp[1]:

for stationID, statinfo in subway\_data['stations'].items() :

if stationID == ID :

if statinfo[1] in transfert:

transfert[statinfo[1]].append((ID, statinfo[0]))

else : transfert[statinfo[1]] = [(ID, statinfo[0])]

return transfert

*# Algo de prim : arbre couvrant de poid minimal*

*# On a un graphe connexe pondéré*

*# Input : G(V, E)*

*# Output : arbre contenant tous les sommets de poids minimun*

*# Algorithme :*

*# entrée : graphe, start\_node*

*# sortie : arbre couvrant*

*# We take any vertx of the graph.*

*# At each we will grow this tree.*

*# step 1 : add the edge of minimum weight linked to you vertex.*

*# The chosen edges will be the edges of minimum weights*

*# step 2 : Check every edge linked to your verteces in you tree*

*# and verify that the two verteces of this edge are not on the tree already*

*# add the edge if it's the edge of minimum weight*

*# if there is multiple minimum edge with the same weight, take a random one*

*# add weight in the total\_weight*

*# if there is no vertices left*

*# => End*

*# Return tree*

*# else recurrence*

def prim(self, start\_node, list\_vertices=[], tree=[]):

if not *self*.is\_connected():

return tree

weightsList = []

edgesList = []

if start\_node not in list\_vertices:

list\_vertices.append(start\_node)

for vertex in list\_vertices:

for edge in *self*.weights:

if edge[0] == vertex and edge[1] not in list\_vertices:

weightsList.append(*self*.weights[edge])

edgesList.append(edge)

if not weightsList: *# If weightsList is empty, return the current tree*

return tree

min\_weight = weightsList[0]

index = 0

for i in range(len(weightsList)):

if min\_weight > weightsList[i]:

min\_weight = weightsList[i]

index = i

tree.append(edgesList[index])

for summit in edgesList[index]:

if summit not in list\_vertices:

list\_vertices.append(summit)

if len(list\_vertices) <= *self*.calculateNbrVertex():

*self*.prim(start\_node, list\_vertices, tree)

return tree

**MAIN.PY**

from graph import Graph

from data import terminus, subway\_data, lines

*# Function to find directions for each shift of metro line during itinerary*

def find\_directions(transfert, lines):

direction, directions = [], []

for key in transfert :

for line in lines :

if key == line[0]:

for i in range(1,len(line)):

if transfert[key][-2][0] in line[i]:

index = line[i].index(transfert[key][-2][0])

if transfert[key][-1][0] == line[i][index + 1]:

direction.append(line[i][-1])

break

if transfert[key][-1][0] == line[i][index - 1]:

direction.append(line[i][0])

break

for ids in direction:

for key, values in subway\_data['stations'].items():

if ids == key:

directions.append(values[0])

return direction

*# Function to swap station id by name in a list*

def findname(directions):

convert = []

for id\_station in directions :

for station\_id, info in subway\_data['stations'].items():

if id\_station == station\_id:

convert.append(info[0])

return convert

*# To test the code manualy*

if \_\_name\_\_ == "\_\_main\_\_":

*# Create a graph instance*

graph = Graph()

*# # Check connectivity*

is\_connected = graph.is\_connected()

if is\_connected:

print("Le graphe est connecté.")

else:

print("Le graphe n'est pas connecté.")

start\_node = input("start : ")

destination = input("dest : ")

shortest\_path = graph.bellman\_ford(start\_node, destination)[1]

transfert = graph.get\_transfert(start\_node, destination)

time\_sp = graph.calculateTime(start\_node, destination)

*# Call the function to get directions for each line in your route*

route\_directions = find\_directions(transfert)

print(shortest\_path,"\n","\n",transfert,"\n","\n",time\_sp,"\n","\n",route\_directions,"\n",)

print(findname(route\_directions))

**APP.PY**

import tkinter as tk

from data import position, subway\_data, lines , line\_colors

from graph import Graph

from main import find\_directions, findname

graph = Graph() *# Your graph object*

*# Retrieving the id from the name*

def found\_ID(name):

for station\_id, info in subway\_data['stations'].items():

if name == info[0]:

return station\_id

*# Retrieving the coordinates of a station*

def get\_station\_coordinates(station):

for (x, y), station\_id in position.items():

if station\_id == station:

return x, y

return None, None

*# Change the color of the vertex and draw lines between each stations*

def change\_color(depart\_station, dest\_station):

reset\_stations\_color()

shortest\_path = graph.bellman\_ford(depart\_station, dest\_station)[1]

transfert = graph.get\_transfert(depart\_station, dest\_station)

route\_directions = findname(find\_directions(transfert, lines))

time\_sp = graph.calculateTime(depart\_station, dest\_station)

for i in range(len(shortest\_path) - 1):

current\_station = shortest\_path[i]

next\_station = shortest\_path[i + 1]

for key, info in subway\_data['stations'].items():

for line in line\_colors:

if key == current\_station and info[1] == line:

color1 = line\_colors[line]

if key == next\_station and info[1] == line:

color2 = line\_colors[line]

*# Initialize x1, y1, x2, y2 before the loop*

x1, y1 = get\_station\_coordinates(current\_station)

x2, y2 = get\_station\_coordinates(next\_station)

if x1 is not None and y1 is not None and x2 is not None and y2 is not None:

*# Draw a line between the stations*

line = canvas.create\_line(x1, y1, x2, y2, fill=color1, width=4)

*# Color the stations*

canvas.itemconfig(stations[current\_station], fill=color1)

canvas.itemconfig(stations[next\_station], fill=color2)

*# Store the line in a list to remove it later if needed*

lines\_between\_stations.append(line)

create\_transit\_map(transfert, route\_directions, time\_sp)

*# Color the graph to give the acpm*

def color\_acpm(depart\_station):

reset\_stations\_color()

acpm = graph.prim(depart\_station)

for i in range(len(acpm)):

current\_station = acpm[i][0]

next\_station = acpm[i][1]

*# Initialize x1, y1, x2, y2 before the loop*

x1, y1 = get\_station\_coordinates(current\_station)

x2, y2 = get\_station\_coordinates(next\_station)

if x1 is not None and y1 is not None and x2 is not None and y2 is not None:

*# Draw a line between the stations*

line = canvas.create\_line(x1, y1, x2, y2, fill="blue", width=4)

*# Color the stations*

canvas.itemconfig(stations[current\_station], fill="blue")

canvas.itemconfig(stations[next\_station], fill="blue")

*# Store the line in a list to remove it later if needed*

lines\_between\_stations.append(line)

*# Display a colorfull itinary*

def create\_transit\_map(transfert, route\_directions, time):

for key, direction in zip(transfert, route\_directions):

for line,color in line\_colors.items():

if key == line :

line\_circle = tk.Canvas(frame, width=40, height=40, highlightthickness=0)

line\_circle.create\_oval(5, 5, 35, 35, fill=color,outline= color )

line\_circle.create\_text(20, 20, text=line, fill='black')

line\_circle.pack(pady=5)

direction\_icon = tk.Label(frame, text=f"→ {direction}", font=("Arial", 20))

direction\_icon.pack()

stop = transfert[key][-1]

stop\_icon = tk.Label(frame, text=f"stop at {stop[1]}", font=("Arial", 20))

stop\_icon.pack()

time\_icon = tk.Label(frame, text=f"~{time} mn", font=("Arial", 70))

time\_icon.pack(ipady=50)

*# Destroy every widgets between each search*

def reset\_stations\_color():

for station\_id in stations.keys():

canvas.itemconfig(stations[station\_id], fill="black")

for line in lines\_between\_stations:

canvas.delete(line)

lines\_between\_stations.clear()

widgets\_to\_keep = [departure\_dropdown, destination\_dropdown, submit\_button ] *# Replace with your actual dropdowns*

for widget in frame.winfo\_children():

if widget not in widgets\_to\_keep:

widget.destroy()

*# Button*

def on\_submit():

change\_color(found\_ID(departure\_var.get()), found\_ID(destination\_var.get()))

def acpm\_submit():

color\_acpm(found\_ID(acpm\_root.get()))

*# Screen/ canvas / frame*

root = tk.Tk()

root.title("Station Map")

frame = tk.Frame(root, padx=8)

frame.pack(side=tk.LEFT)

canvas = tk.Canvas(root, width=1200, height=1000)

canvas.pack()

*# Graphic illustration of the stations*

stations = {}

lines\_between\_stations = []

for (x, y), station\_id in position.items():

station = canvas.create\_oval(x - 5, y - 5, x + 5, y + 5, fill="black")

stations[station\_id] = station

*# Drop-downs for selection*

departure\_var = tk.StringVar(root)

departure\_var.set("Select Departure")

departure\_values = [info[0] for info in subway\_data['stations'].values()]

departure\_dropdown = tk.OptionMenu(frame, departure\_var, \*departure\_values)

departure\_dropdown.pack(anchor='n', pady=5)

destination\_var = tk.StringVar(root)

destination\_var.set("Select Destination")

destination\_values = [info[0] for info in subway\_data['stations'].values()]

destination\_dropdown = tk.OptionMenu(frame, destination\_var, \*destination\_values)

destination\_dropdown.pack(anchor='n', pady=5)

*# Button*

submit\_button = tk.Button(frame, text="Submit", command=on\_submit)

submit\_button.pack(anchor='n', pady=5)

*# Create a frame to contain the new dropdown and button*

top\_frame = tk.Frame(root, padx= 8)

top\_frame.place(x=root.winfo\_screenwidth() - 950, y=0) *# Adjust the coordinates as needed*

*# Create the ACPM dropdown inside the new frame*

acpm\_root = tk.StringVar(root)

acpm\_root.set("Select a vertex")

acpm\_rootValues = [info[0] for info in subway\_data['stations'].values()]

acpm\_rootValues\_dropdown = tk.OptionMenu(top\_frame, acpm\_root, \*acpm\_rootValues)

acpm\_rootValues\_dropdown.pack()

*# Create a button inside the top frame*

acpm\_button = tk.Button(top\_frame, text="ACPM", command=acpm\_submit)

acpm\_button.pack(side=tk.RIGHT)

root.mainloop()