Compte-rendu du projet d'Algorithmique de graphes

DJADEL Céline, PIRABAKARAN Thanushan, SANTINI Maya

Avril 2022

Plan :

Introduction :

Le but de ce projet est d'écrire un programme informatique permettant de déterminer le plus court itinéraire pour aller d'une station à une autre dans le métro parisien. Le programme est écrit en python.

I - Récupération des données :

Voici les différentes données utilisées, y compris celles qui ne proviennent pas du document txt:

- Les imports :
  `import sys` : nous permettra de modifier la récursivité maximale.
  `import tkinter as tk` : nous permettra de créer l'interface graphique.
  `from tkinter import ttk` : nous permettra d'utiliser des widgets de tkinter spéciaux (*exemple : Combobox)*.

- Les variables :
  `arretes` : dictionnaire contenant les sommets de départ et d'arrivée de E. (connexité)
  `bloque` : liste indiquant où le graphe n'est pas connexe. (connexité)
  `choix` : liste des stations utilisées pour le menu déroulant. (interface graphique)
  `dicE` : dictionnaire contenant les arêtes entre tous les sommets et le temps nécessaire pour aller d'un sommet à l'autre { station_depart : [station_arrivee : temps] } de notre graphe. (dijkstra)
  `dicgare`: dictionnaire contenant toutes les stations de notre graphe sous la forme {'numéro de station': ['Nom de la station', 'métro correspondant', "Terminus "]. (dijkstra, interface graphique)
  `dicoregroupe`: dictionnaire utilisé pour le menu déroulant qui permettra d'éviter d'obtenir des doublons de gare dans le menu (exemple : Arts et Métiers apparaissant plusieurs fois car plusieurs métros passent par cette station). (interface graphique)
  `noeuds` : liste de toutes les stations. (connexité)
  `path` : récupération du fichier txt sur lequel nous allons travailler, provenant du metro.txt mais modifié. (récupération de données)
  `visites` : liste des sommets visités du graphe pour vérifier sa connexité. (connexité)

- La récupération des données du txt :
  Avant toute autre chose, on commence par récupérer les données présentes dans notre fichier txt de base.
  Pour cela, on commence par ouvrir notre fichier metro.txt en utilisant l'encoding 'utf8' afin de pouvoir afficher les caractères spéciaux sur n'importe quel système d'exploitation.
  On récupère les 375 stations et on les ajoute à notre dictionnaire dicgare grâce aux deux méthodes split( ) et readline().
  On fait de même pour les 472 connexions, ainsi que leur temps respectifs présents dans notre fichier et on les ajoute au dictionnaire dicE.
  En ce qui concerne le dictionnaire aretes, il est similaire au dicE sauf qu'on y ajoute seulement les connexions entre les stations sans le temps.

- Les fonctions utilisées :
  `def dico_adj():` Cette fonction nous permet de renvoyer un dictionnaire dic_adj représentant le graphe qui a en clés les différents sommets du graphe et en valeurs les sommets adjacents. Nous allons par la suite utiliser ce dictionnaire afin de vérifier la connexité de notre graphe.

  `def bidirectionnel():`
  La fonction bidirectionnel parcourt tous les éléments présents dans la liste noeuds et si un élément n'est pas présent dans dicE, on parcourt cette fois-ci le dicE à la recherche d'une station de départ possédant l'élément comme station d'arrivée. On rajoute cette élément comme clé du dicE et en valeur on aura la station de départ correspondante ainsi que le temps.
  En ce qui concerne les lignes de métro 7bis et 10, dont certaines stations ne sont pas bidirectionnelles, il nous faut donc les supprimer manuellement.

  `def ligne(i):`
  Cette fonction nous permet de créer la ligne de métro d'une station donnée.
  Pour cela, on récupère le terminus de la station et son numéro correspondant dans le dictionnaire dicgare.
  En partant de ce terminus, on parcourt toute la ligne de métro en suivant les connexions présentes dans dicE si les stations appartiennent à la même ligne de métro.
  Cette fonction retourne les stations présentes dans la ligne de métro ainsi que son numéro.

```
def cas_particulier(ligne,numero):
```

Cette fonction concerne les lignes de métros 7bis, 7, 10 et 13. Elle crée une liste imbriquée contenant les deux directions possibles pour chaque ligne et la retourne à la fin.

II - Vérification de la connexité :

Afin de vérifier la connexité, nous avons créé deux fonctions:

- ```
  def connexe(sdepart,dic_adj):
  ```

  Cette première fonction est une fonction récursive. Elle prend en argument sdepart, la station de départ, et dic_adj. En variables globales, on a noeuds et visites, la liste qui va venir être complétée au fur et à mesure de la fonction lorsqu'un sommet/station est visité.
  Dans un premier temps, si la station de départ n'est pas dans la liste des sommets visités, on l'ajoute à celle-ci. Ensuite, on va parcourir avec i les stations reliées à la station de départ grâce au dictionnaire dic_adj, et si la station i n'est pas dans la liste des sommets, on rappelle la fonction connexe avec en argument (i, dic_adj), i étant désormais la station de départ.
  Cela permettra de vérifier la connexité pour tous les sommets : si le graphe n'est pas connexe, la fonction bloquera à ce point et n'ajoutera pas les sommets où ce n'est pas connexe à la liste visites.

- ```
  def verif_connexe_fin():
  ```

  Cette deuxième fonction permet de vérifier, après exécution de la première fonction, si le graphe est connexe. Pour cela, on a les variables globales noeuds, visites et bloque, cette dernière permettra d'indiquer là où le graphe n'est pas connexe.
  D'abord, nous avons comparé la longueur de la liste visites au nombre de stations total avec un if : en effet, si elles sont égales, c'est qu'on peut aller de n'importe quelle station à une autre, et donc que le graphe est connexe. On print ensuite 'Le graphe est connexe.'
  Avec un else, donc si il n'y a pas le même nombre de stations que de stations dans visites, on va venir voir où est-ce que le graphe n'est pas connexe : on parcourt noeuds avec elt, puis on vérifie si elt est dans visites. Si il ne l'est pas, c'est que le sommet n'a pas été traité, et on l'ajoute à bloque. Lorsque noeuds a été parcouru entièrement, on s'arrête et on print 'Le graphe n'est

pas connexe à :', suivi de bloque pour savoir à quelles stations le graphe n'est pas connexe.

Avant d'arriver à cette solution, nous avons essayé plusieurs méthodes : des listes imbriquées, une matrice adjacente.. ainsi que la récursivité. Au départ, la récursivité ne fonctionnait pas mais nous avons pu, grâce à sys, changer le maximum d'appels récursifs et faire en sorte que ça fonctionne.

Ⅲ- Le plus court itinéraire :

Pour calculer et indiquer à l'utilisateur le meilleur itinéraire pour aller d'une station à une autre, nous avons décidé d'implémenter l'algorithme de Dijkstra en python. Plusieurs fonctions, complémentaires ou non, ont donc été créées:

```
def Dijkstra(SDepart,SArrive):
```

Cette fonction nous permet d'obtenir le temps ainsi que le chemin le plus court entre deux stations données . Pour cela, on a les variables SDepart et SArrive qui correspondent respectivement à la station de départ et la station d'arrivée.
On commence par créer trois dictionnaires :

- dist : correspond au temps final afin d'arriver à SArrive ainsi qu'à toutes les stations adjacentes aux stations traitées. Au format {station traitée: temps final afin d'arriver à la station traitée}.
- pred : contient tous les sommets traités en partant de la station SDepart. Au format {station traitée : prédécesseur du sommet traité}.
- stock : dictionnaire intermédiaire qui permet d'obtenir le dictionnaire dist dont le premier élément est {SDepart:0}. Au format {station traitée: temps final afin d'arriver à la station traitée}.

Ainsi qu'une liste nommée Traites qui comporte en premier lieu SDepart.
D'abord, on parcourt tous les éléments présents dans la liste Traites et si l'élément correspond à SArrive on interrompt le parcours avec un `break.` Dans le cas contraire, on ajoute les stations adjacentes aux stations déjà traitées.
On calcule ensuite le temps nécessaire afin d'aller de la station traitée aux stations adjacentes puis on ajoute une condition qui permet de vérifier si la station a déjà été traitée et ajoutée aux dictionnaire stock et de la rajouter dans le cas ou elle n'y est pas encore ainsi que son prédécesseur.

Au final, nous avons comme résultat le temps nécessaire ainsi que le chemin le plus court nous permettant d'arriver à SArrive.

```
def est_chemin(p, SDepart, SArrive):
```

Nous avons créé cette fonction afin de pouvoir récupérer le chemin le plus court que l'on renvoie dans la fonction précédente.
Elle prend en argument p, SDepart et SArrive qui correspondent respectivement au dictionnaire contenant les prédécesseurs récupérés durant l'exécution de la fonction `Dijkstra`, la station de départ et la station d'arrivée.
Tant que le prédecesseur de Sarrive est différent de Sdepart, on ajoute le prédecesseur de Sarrive dans une liste appelée chemin qui contient déjà Sarrive. SArrive prend la valeur de son prédécesseur et on obtient donc le chemin le plus court en inversant la liste chemin.

```
def ShortestDijkstra(c):
```

Cette fonction nous permet de récupérer uniquement le début ainsi que la fin d'un même métro dans un chemin.
Elle prend en argument c, qui est le chemin le plus court obtenu grâce à notre fonction `Dijkstra`.

```
def Intermediaire(SDepart, SArrive):
```

Elle prend en argument SDepart et SArrive.
Si la station de départ ou d'arrivée possède plusieurs numéros, on applique la fonction `Dijkstra` pour chacune des possibilités et on prend le chemin le plus court entre ceux proposés.
A l'intérieur de cette fonction, on utilise également la fonction `ShortestDijkstra` sur le chemin le plus court obtenu afin de simplifier les calculs pour l'affichage.

```
def TempsTrajet(p):
```

Cette fonction nous permet simplement de convertir le temps obtenu par `Dijkstra` sous la forme : {heure} heure et {min} minutes et {seconde} secondes.

Ⅳ- Interface graphique :

Une fenêtre s'affiche, permettant de sélectionner sa station de départ et sa station d'arrivée à l'aide de barres déroulantes. L'utilisateur appuie ensuite sur le bouton "Recherche", qui agrandit la fenêtre actuelle pour laisser paraître le chemin le plus court qui doit être pris.
Pour cela, nous avons utilisé l'interface graphique tkinter et les widgets qu'elle propose. A la suite, les fonctions créées sont détaillées:

`def affichage(tps, longdic, gare, end):` Permet d'afficher le texte selon les différents cas possibles. Cette fonction est très longue, mais cela est nécessaire si l'on veut traiter tous les cas imaginables. Elle prend en entrée:

- tps : chemin renvoyé par `ShortestDijkstra`
- longdic : si `ShortestDijkstra` renvoie un chemin où, entre deux stations, `Dijkstra` aurait renvoyé x stations en plus, longdic indique x sous forme de dictionnaire {Numéro du métro : Taille} *(ex : {M2 : 7})*
- gare : comme longdic, si `ShortestDijkstra` renvoie un chemin où, entre deux stations, `Dijkstra` aurait renvoyé x stations en plus, il indique la gare de proximité donc celle qui suit la gare actuelle dans `Dijkstra`, même si ce n'est pas la gare qui suit dans `Dijkstra`
- end : temps renvoyé par `TempsTrajet`

En variable locale, on a un compteur compt qui ne sera utile que pour la conversion des numéros de stations sous format '0000' si ce n'est pas déjà le cas, pour traiter toutes les stations du chemin.
On récupère ensuite dans metro et numero ce que renvoie `ligne`. On initialise une variable ter qui nous permettra dans la seconde partie de la fonction d'insérer des conditions aux cas particuliers.
Pour la première partie de la fonction, on exclut les cas particuliers donc les métros 7, 7bis, 10 et 13 avec un if. On vérifiera dans quel sens nous allons pour indiquer la bonne direction en comparant notre station actuelle et la station qui suit, notamment grâce à dicgare. Selon les différents cas, différentes phrases s'afficheront à l'écran : si on part de la première gare, seulement "Prenez" sera affiché tandis que dans les autres cas, "Changez et prenez" sera affiché. Si un changement sur la même gare seulement doit être effectué, uniquement "Changez" sera affiché. La fonction s'arrête uniquement quand nous arrivons à la fin du chemin du `ShortestDijkstra` et renvoie end, le temps à afficher.
Pour la deuxième partie de la fonction, on va faire au cas par cas : les métros 7 et 13 ont 3 terminus, et les métros 7bis et 10 ont des boucles. On commence par définir les différents terminus et add, comme pour la première partie, qui nous permettra de

définir par la suite dans quel sens on va. terminus1 et terminus2 pour les métros à double terminus, et terminus pour le terminus de la ligne droite. On va ensuite traiter selon différents cas : nous les avons dessinés et vous les présenterons lors de la soutenance, et la feuille avec les dessins expliquant les cas sera donnée en format papier en annexe du compte rendu. Selon chaque cas, l'affichage sera différent car le terminus voulu sera différent. Il y a 4 cas possibles : l'affichage du terminus, du terminus1, du terminus2, ou du terminus1 et terminus2 en même temps. De plus, il faudra aussi traiter les cas différemment s' il s'agit des métros avec des boucles, d'où la longueur de cette partie de code. Des `break` sont utilisés quand des boucles sont sollicitées. La fonction s'arrête de la même manière que dans la première partie de la fonction.

`def Stations_Menu():` Permet de remplir la liste choix ne contenant alors que les stations qui seront affichées dans la barre déroulante dans l'ordre alphabétique. C'est ici que les doublons seront évités sur l'affichage du menu déroulant, notamment grâce au dictionnaire dicoregroupe. Elle sera rappelée lors de la création des widgets.

`def relie(D,A):` Permet de relier la fonction press, appelée par le bouton, à `Intermediaire` et donc Dijkstra. Elle affiche la première phrase, qui indique la gare de départ, et la dernière phrase, qui indique la station d'arrivée ainsi que le temps calculé par l'algorithme de Dijkstra et ajusté par la fonction `TempsTrajet`. Si l'utilisateur a sélectionné la même gare de départ et d'arrivée, elle indiquera ce fait. Elle vérifiera également si la gare choisie est desservie par plusieurs métros grâce à dicoregroupe : si tel est le cas, une liste sera renvoyée à `Intermediaire` qui la traitera par la suite, sinon, une liste avec un seul élément contenant le numéro de la station lui sera transmis. Grâce à cette dernière, nous récupérons le temps du trajet.

`def clearTextInput():` Permet de supprimer l'affichage présent si l'utilisateur appuie sur le bouton pour calculer un nouveau trajet.

`def press():` Permet de relier le bouton de recherche à la fonction `relie`*(en transmettant les stations choisies par l'utilisateur)*, qui elle permettra l'affichage du trajet le plus court sur l'interface graphique. C'est également cette fonction qui permet à la fenêtre de s'agrandir pour laisser paraître le chemin le plus court. Elle appelle `clearTextInput` à chaque fois qu'elle est appelée, et modifie l'affichage du texte du bouton et du choix des stations.

En dehors de ces fonctions, à la fin du code, nous avons également créer l'interface de base, nommée `racine`, à laquelle nous avons attribué nom, taille et autres attributs comme la couleur. Nous avons également créé une zone de texte pour l'affichage du chemin, deux labels pour indiquer où choisir la station de départ et d'arrivée, deux combobox pour le choix des stations, un bouton qui permet de lancer le processus, et une icône de métro.

Par DJADEL Céline, PIRABAKARAN Thanushan, et SANTINI Maya