TD01-2 : Graphes-2
Réseau ferré RATP

Ce TP se propose d'explorer le graphe du réseau RATP.
Les entités du réseau RATP sont représentées par plusieurs dictionnaires, `Dlignes`, `Dstations`, `Darrets`.
Il existe trois types de lignes : métro, tramway ou funiculaire.
Les stations se distinguent des arrêts en ceci que les stations comportent un ou plusieurs arrêts, tandis que chaque arrêt est attaché à une station, dans laquelle il se trouve, l'arrêt en lui-même correspondant au quai à partir duquel on monte dans la rame.

Les clés des dictionnaires `Dlignes`, `Dstations` et `Darrets` ont pour clé un identifiant correspondant, respectivement, à une ligne, une station, ou un arrêt dans une station, associée à un dictionnaire.
Par exemple :

- à l'identifiant de ligne `'C01380'`, le dictionnaire `Dlignes`, associe le dictionnaire `{'nom': '10', 'type': 1, 'couleur': 'C9910D'}` ;
- à l'identifiant de station `'71972'`, le dictionnaire `Dstations`, associe le dictionnaire `{'nom': 'Ourcq', 'latitude': 48.8870329280158, 'longitude': 2.3866642761809373}` ;
- à l'identifiant d'arrêt `'463186'`, le dictionnaire `Darrets`, associe le dictionnaire `{'id_ligne': 'C01376', 'sens': '0', 'id_station': '71026'}` ;

On distingue trois types pour les lignes, donnés par le dictionnaire suivant :

```
>>> Dtypes_lignes
{0: 'tramway', 1: 'métro', 7: 'funiculaire'}
```

On distingue pour les arrêts, un sens, indiqué dans le dictionnaire décrivant l'arrêt, égal à `'0'` ou `'1'`, suivant le sens de parcours de la ligne sur laquelle l'arrêt se trouve.

Le réseau RATP est modélisé par un graphe orienté, que l'on nommera G, dont les sommets sont des identifiants de station ou d'arrêt, et les arcs sont des trajets directs possibles entre deux sommets.

Une liste des sommets du graphe est disponible, nommée `LS`, ainsi qu'une liste des arcs `Larcs`.
À chaque sommet est associé une étiquette indiquant s'il s'agit d'une station ou d'un arrêt, à laquelle on accède grâce au dictionnaire `Dsommet_etiquette`.
Par exemple :

```
>>> Dsommet_etiquette['462915'], Dsommet_etiquette['71370']
('arret', 'station')
```

À chaque arc est associé un dictionnaire-étiquette à auquel on accède grâce au dictionnaire `Darc_etiquette`.
Par exemple :

```
>>> Darc_etiquette[('22380', '463235')]
{'poids': 15, 'type': 0}
```

Le dictionnaire renvoyé indique un « poids » qui donne le temps de trajet direct entre les deux sommets en secondes, et un « type » qui indique le type du trajet direct.
Les différents types de trajets directs sont donnés par le dictionnaire suivant :

```
>>> Dtypes_arcs
{0: 'correspondance', 1: 'transport', 2: 'entree_station', 3:
'sortie_station'}
```

Le type 0 correspond à des trajets à pied, le type 1 à des trajets en empruntant une ligne, les types 2 et 3 correspondent, respectivement, au temps nécessaire pour aller de l'entrée de la station jusqu'au quai, et pour rejoindre l'extérieur de la station depuis le quai.

Enfin, le graphe orienté est représenté en machine par un dictionnaire, `Dadj`, associant à chaque sommet, la liste des sommets directement accessibles depuis ce sommet en empruntant un arc.
Par exemple :

```
>>> Dadj['463240']
['73671', '22031', '463068', '22117', '462988']
```

1 Préliminaires

Q1. Définir deux fonctions, `calc_ordre(dAdj)` et `calc_taille(dAdj)`, prenant en argument un graphe orienté décrit par un dictionnaire des listes d'adjacences de ses sommets. Déterminer, par appel à ces fonctions, l'ordre et la taille du graphe G.

Q2. Exécuter l'instruction `help(cherche_nomstation)` pour afficher l'aide à l'utilisation de la fonction `cherche_nomstation`, puis, à l'aide de cette fonction, afficher la liste des noms de toutes les stations dont le nom comporte la sous-chaîne `'ex'`.

Q3. La fonction `obt_idstation(nomstation)` est déjà définie et renvoie l'identifiant de la station dont on lui donne le nom. En utilisant également le dictionnaire des listes d'adjacence `Dadj`, obtenir les identifiants de tous les arrêts situés à la station Exelmans, puis afficher leurs caractéristiques à l'aide du dictionnaire `Darrets`.

Q4. Vérifier, à l'aide du dictionnaire `Dlignes`, que les arrêts de la station Exelmans se situent bien sur la ligne 9, mais dans des sens de parcours de la ligne différents.

L'un des arrêts à la station Exelmans a pour identifiant `'462956'`. On souhaite, à l'aide d'un parcours en profondeur du graphe, obtenir la liste, `L`, de tous les arrêts, situés sur la ligne 9, dans le même sens de parcours, depuis l'arrêt `'462956'`, jusqu'au terminus de la ligne 9.
Pour cela on souhaite définir le sous-graphe, H, induit par la restriction de l'ensemble des sommets (stations ou arrêts) à l'ensemble des arrêts.

Q5. **a.** Construire une représentation du **sous-graphe induit** H, sous la forme d'un dictionnaire de ses listes d'adjacence, `DadjH`, dont les sommets et les arcs correspondent, respectivement, à des arrêts et à des arcs entre des arrêts qui ne sont pas des correspondances (arcs de type 1). On utilisera les dictionnaires donnant les étiquettes des sommets et des arcs : `Dsommet_etiquette` et `Darc_etiquette`.

b. Déterminer l'ordre et la taille du graphe H.

On donne ci-dessous un script permettant d'obtenir, pour un graphe représenté par un dictionnaire, `dAdj`, de listes d'adjacence de ses sommets, à l'aide d'un parcours en profondeur, la liste, `accessibles_s0`, des sommets accessibles depuis un sommet `s0`.

```
s0 = ...
accessibles_s0 = [s0]

atteints = {s: False for s in LS}
pile = []
s = s0
pile.append(s)
atteints[s] = True
while len(pile) > 0:
    s = pile.pop()
    for succ in dAdj[s]:
        if not atteints[succ]:
            pile.append(succ)
            atteints[succ] = True
            accessibles_s0.append(succ)
```

Q6. **a.** Compléter et adapter ce script pour obtenir les listes des identifiants des arrêts accessibles depuis chaque arrêt situé à la station Exelmans (*cf.* **Q2**).

b. Afficher les identifiants des arrêts pour les deux listes suivis des noms des stations dans lesquelles ces arrêts sont situés, à l'aide de la fonction `obt_nomstation_arret(idarr)`.

c. Vérifier que les arrêts de la station Exelmans, ainsi que tous les arrêts accessibles depuis eux sont des sommets de degré sortant 1 ou 0. Interpréter ce fait.

Q7. **a.** Construire le dictionnaire `Darrets_degres_sortants`, dont les clés sont les sommets du graphe H, associés à leur degré.

b. Construire ensuite le dictionnaire `Ddegres_sortants`, dont les clés sont les différents degrés des sommets du graphe, associés à la liste des sommets présentant ce degré. Afficher les degrés sortants distincts dans le graphe et le nombre de sommets correspondants.

c. Comment interpréter une valeur de degré sortant d'un sommet supérieure strictement à 1 ?

d. Pour les sommets de degré sortant 2, faire appel à la fonction `obt_dinfo_arret(idarr)` pour afficher un dictionnaire donnant l'identifiant de l'arrêt, le nom et l'identifiant de la station correspondante, ainsi que le nom et l'identifiant de la ligne sur laquelle se trouve l'arrêt, ainsi que le sens de parcours de la ligne. Afficher les mêmes informations pour les successeurs de ces sommets de degré 2.

2 Exploration sous-graphe H (graphe induit par les arrêts)

2.1 Reconstruction des lignes de métro

On souhaite obtenir les terminus pour chacune des lignes du réseau.

Q8. Construire un dictionnaire, `Dlignes_terminus`, dont les clés sont des couples (nom de ligne, sens), associés à la liste des arrêts qui sont des terminus, sur cette ligne et dans ce sens. On utilisera la liste des noms de lignes, `liste_noms_lignes`, et la fonction `obt_nomligne(idligne)`, et le dictionnaire `DadjH`. On terminera en vérifiant, à l'aide de `assert`, qu'il existe bien, pour chaque ligne et dans les deux sens de parcours, un terminus, puis en affichant, lorsqu'il existe plusieurs terminus, la ligne concernée, le sens de parcours, et les terminus.

Q9. À partir du dictionnaire précédent, construire un dictionnaire, `Dterminus`, dont les clés sont les identifiants des arrêts trouvés précédemment comme terminus, avec pour valeur associée un dictionnaire dont les clés sont l'identifiant de la ligne sur laquelle le terminus se trouve, son nom, et le sens de parcours. On pourra utiliser la fonction `obt_idligne(nomligne)`. On aura par exemple :

```
>>> Dterminus['22233']
{'id_ligne': 'C01383', 'nom_ligne': '13', 'sens': '1'}
```

On souhaite maintenant obtenir, pour chaque ligne et dans les deux sens, les trajets, depuis le départ de la ligne jusqu'à un terminus.

Q10. Définir une fonction, `DFSaccessibles(dAdj, s0)`, sur la base du script utilisé en question **Q6**, et renvoyant la liste des sommets accessibles depuis un sommet `s0`, dans un graphe décrit par un dictionnaire des listes d'adjacence `dAdj`,

Q11. En prenant pour exemple le terminus de la ligne 9 parcourue dans le sens 0, dont l'identifiant, `'463233'`, a été obtenu à la question **Q6.a,** tandis que ses caractéristiques ont été affichées en **Q6.b,** justifier que l'appel à la fonction `DFSaccessibles` pour un terminus renvoie une liste à un élément.

Afin d'obtenir les trajets, le long de chaque ligne, depuis le départ jusqu'au terminus, on va construire le graphe inverse du graphe H. **Ce graphe inverse ne servira que pour les questions Q12 et Q13**.

Q12. Construire le dictionnaire `DadjH_transp`, décrivant le graphe transposé (ou inverse) H^T du graphe H, obtenu en conservant les sommets de H et en inversant tous les arcs de H, *i.e.*, si $H = (S, A)$, alors $H^T = (S, A^T)$ où $A^T = \{(s', s), (s, s') \in A\}$.

Q13. Caractériser, dans le graphe transposé, H^T, les arrêts au départ de chaque ligne, et construire deux dictionnaires `Dlignes_departs` et `Ddepart`, analogues des dictionnaires `Dlignes_terminus` et `Dterminus` construits aux questions **Q8** et **Q9**, mais cette fois pour les stations au départ de chaque ligne.

Q14. En considérant le départ et le terminus de la ligne 7bis parcourue dans le sens `'0'`, dont les identifiants sont, respectivement, `'24681'` et `'24684'`, déterminer, à l'aide de la fonction `DFSaccessibles`, les arrêts accessibles depuis l'arrêt de départ, dans le graphe H. Afficher les informations pour chacun de ces arrêts, à l'aide de la fonction `obt_dinfo_arret(idarr)`.

Q15. On donne l'identifiant, `'463200'`, de l'arrêt au départ de la ligne 7, parcourue dans le sens `'1'`, et on rappelle que cette ligne, parcourue dans ce sens admet deux terminus, d'identifiants respectifs, `'463220'` et `'463269'`. La méthode de la question **Q14** permet-elle d'obtenir un trajet depuis le départ de la ligne jusqu'à un de ses terminus ? Expliquer pourquoi.

Afin de déterminer des chemins vers les terminus des lignes, on va modifier le code de la fonction `DFSaccessibles(dAdj, s0)`, afin de garder trace des prédécesseurs de chaque sommet au moment de sa découverte.

Q16. Définir une fonction `DFSaccessibles1(dAdj, s0)` obtenue en modifiant le code de la fonction `DFSaccessibles(dAdj, s0)`, afin de construire et de renvoyer, au lieu d'une liste des sommets accessibles, un dictionnaire, `Dpreds`, dont les clés sont les sommets accessibles, associés à leur prédécesseur au moment de leur découverte. On associera le sommet `s0`, qui n'a pas de prédécesseur, à la valeur `None`. On pourra également utiliser ce dictionnaire pour se substituer au rôle du dictionnaire `atteints` utilisé jusque-là pour effectuer le parcours en profondeur.

Q17. Tester la fonction `DFSaccessibles1` avec le terminus de la ligne 7bis, parcourue dans le sens `'0'`, d'identifiant `'24684'`, donné par `Dlignes_terminus['7bis', '0']`. On devra obtenir :

```
>>> DFSaccessibles1_v0(DadjH_transp, s0='24684')
{'24684': None, '24682': '24684', '24686': '24682', '463156': '24686', '24685':
'463156', '24680': '24685', '24681': '24680'}
```

Q18. Définir une fonction `reconstruire_chemin(Dpreds, s1)`, permettant de reconstruire et de renvoyer, sous la forme d'une liste `chemin`, un chemin, depuis un sommet de départ `s0`, jusqu'à un sommet `s1` accessible par un parcours en profondeur depuis `s0`, à partir du dictionnaire de prédécesseurs, `Dpreds_s0`, associé à ce parcours. On vérifiera, à l'aide d'une assertion que le sommet `s1` est bien accessible.

Q19. Tester la fonction `reconstruire_chemin` avec le dictionnaire obtenu à la question **Q17** et le terminus correspondant. Reprendre ensuite l'exemple de la question **Q15**.

2.2 Recherche de circuits

Q20. On souhaite modifier la fonction `DFSaccessibles1(dAdj, s0)` en une fonction `DFScircuit(dAdj, s0)` détectant s'il existe un circuit passant par s_0. On utilisera que s_0 appartient à un cycle si et seulement si dans une exploration du graphe à partir de s_0, on trouve le sommet s_0 parmi les successeurs d'un sommet s_1 accessible depuis s. Dès qu'un tel sommet s_1 est trouvé, la fonction `DFScircuit` renverra le couple (s_0, s_1), sinon la fonction renverra `None`.

Q21. Définir une fonction `detecte_circuits(dAdj)` détectant s'il existe des circuits dans le graphe H. La fonction fera appel à la fonction `DFScircuit(dAdj, s0)` et renverra la liste des couples (s_0, s_1) renvoyés. La liste renvoyée sera vide s'il n'existe aucun circuit dans H. Est-on assuré de détecter tous les circuits du graphe ? Est-on assuré par cette méthode de détecter des circuits distincts ?

Q22. On a effectué une copie, H', du graphe H dans laquelle quatre arcs ont été ajoutés. Une description de cette copie est donnée sous la forme d'un dictionnaire de listes d'adjacence, `DadjHC`, qui peut être importée dans le notebook avec l'instruction `from annexe_TD10_graphes import *`. Vérifier que cette copie comporte deux circuits distincts, de longueurs 7 et 51.

Q23. Vérifier que dans le graphe H, il n'existe aucun circuit.

3 Exploration du symétrisé de H

On considère ici le graphe non orienté H^S obtenu en rendant bidirectionnels tous les arcs du graphe H, *i.e.* si $H = (S, A)$, alors $H^S = (V, A^S)$ où $A^S = \{(s, s'), (s, s') \in A \text{ ou } (s', s) \in A\}$.

Q24. Construire le dictionnaire `DadjHS`, décrivant le graphe H^S. On vérifiera préliminairement qu'aucun arc de H n'est bidirectionnel, *i.e.* que si (s, s') est un arc de H, alors (s', s) n'est pas un arc de H.

3.1 Étude de la connexité de H^S

Q25. Grâce à un appel à la fonction `DFSaccessibles(dAdj, s0)`, vérifier que le graphe H^S n'est pas connexe.

Q26. Modifier le code de la fonction `DFSaccessibles(dAdj, s0)`, afin d'obtenir une fonction `DFScompoconn(dAdj)` renvoyant, si le dictionnaire `dAdj` décrit un graphe non orienté, une liste de listes dont chaque sous-liste contient les sommets d'une composante connexe du graphe.

Q27. Vérifier que pour le graphe H^S, on obtient 38 composantes connexes correspondant aux 19 lignes du réseau, parcourues soit dans le sens `'0'`, soit dans le sens `'1'`.

3.2 Recherche de cycles

Pour la recherche de cycles, on considère le graphe H_C^S obtenu en fusionnant les arrêts situés à la même station et sur la même ligne, en un seul sommet, étiqueté par un couple, (nom de ligne, nom de station). On obtient alors un graphe non orienté d'ordre 444, comportant 429 arêtes, décrit par le dictionnaire `DadjHSC`, que l'on pourra importer avec l'instruction `from annexe_TD10_graphes import *`.

Q28. Vérifier que pour le graphe H_C^S, comporte 19 composantes connexes correspondant aux 19 lignes du réseau.

Dans un graphe non orienté, un cycle est nécessairement de longueur au moins trois, sinon il s'agit d'une arête, parcourue dans un sens, puis dans l'autre. Par ailleurs, tous les chemins peuvent être parcourus dans les deux sens.
Ainsi, dans un parcours à partir d'un sommet s_0, lorsque l'on découvre un sommet s à partir d'un sommet s'' dont il est voisin, si l'un des successeurs de s, s', différent de s'', a déjà été atteint, alors il existe un chemin c de s_0 à s et un chemin c' de s_0 à s'. Si l'on note $c = (s_0, s_1, \dots, s_{p-3}, s_{p-2} = s'', s)$ et $c' = (s'_0 = s_0, s'_1, \dots, s'_{q-2}, s')$, et k_0 le plus grand k tel que $s_k = {s_k}'$, alors le chemin

$$(*) \quad (s_k, \dots, s_{p-2} = s'', s, s', s'_{q-2}, \dots s_k)$$

est un cycle.
Réciproquement, s'il existe, dans un graphe non orienté, un cycle c, que l'on note $c = (s_1, s_2, \dots, s_n)$, en convenant que le nom s_n est attribué au dernier sommet du cycle qui est découvert, alors, lorsque l'on examine le voisinage de s_n, on y trouvera le sommet s_1 ou le sommet s_{n-1}, qui aura déjà été découvert, sans être pour autant le prédécesseur de s_n. On découvrira donc le cycle c au moment de la découverte de s_n ou, à tout le moins, on découvrira un cycle auquel appartient s_n.
Ainsi, la présence d'un cycle dans le graphe se signale par la découverte d'un sommet dont un successeur a déjà été atteint et qui n'est pas le prédécesseur de ce somment dans le parcours mis en œuvre.

Q29. Définir une fonction `detecte_cycle(dAdj)` effectuant un parcours en profondeur d'un graphe non orienté décrit par `dAdj`, et renvoyant la liste de couples (s, s'), ces couples correspondant à tous les sommets, s, qui, au moment de leur découverte, ont un voisin, s', déjà découvert qui n'est pas leur prédécesseur dans le parcours. La fonction renverra également le dictionnaire des prédécesseurs de chaque sommet construit au fil du parcours du graphe. On se basera sur l'implémentation de la fonction `DFScompoconn(dAdj)`, en construisant un dictionnaire des prédécesseurs et en l'utilisant pour remplacer le dictionnaire `atteints`. Tester la fonction avec le dictionnaire `DadjHSC` et constater que l'on obtient une liste de huit couples (s, s').

Q30. Définir une fonction `reconstruire_cycle`, prenant en argument un dictionnaire des prédécesseurs, et deux sommets `s1` et `s2`, correspondant à un couple (s, s') renvoyés par la fonction `detecte_cycle`, et renvoyant un cycle reconstitué selon le principe décrit ci-avant (*cf.* $(*)$).

Q31. Utiliser la fonction `reconstruire_cycle` pour reconstituer huit cycles dans le graphe H_C^S, chacun correspondant à l'un des couples renvoyés par l'appel à `detecte_cycle(DadjHSC)`. On pourra faire appel à la fonction `reconstruire_chemin`. Donner le nombre de cycles distincts détectés.

4 Plus courts chemins dans le graphe G

Le graphe G décrit l'ensemble du réseau RATP, avec ses stations et les arrêts dans les stations.
Les stations correspondent aux points d'accès au réseau depuis l'extérieur, tandis que les arrêts correspondent aux quais dans les stations. Les stations et les arrêts sont les sommets du graphe, et la liste des sommets du graphe sont les identifiants de ces stations et de ces arrêts.
Les arcs du graphe sont de trois types :

- type 0 : trajet à pied entre deux quais (arrêts), il s'agit de « correspondances » ;
- type 1 : trajet direct entre deux arrêts ;
- type 2 : trajet à pied depuis l'entrée d'une station jusqu'à l'un de ses quais (arrêt) ;
- type 3 : trajet à pied depuis un quai (arrêt) jusqu'à la sortie de la station, à l'extérieur.

On rappelle que les étiquettes des sommets (`"station"` ou `"arret"`) sont données par le dictionnaire `Dsommet_etiquette`, dont les clés sont les identifiants des sommets, et que les étiquettes des arcs, donnant, sous la forme d'un dictionnaire, le type d'arc et le temps de trajet associé, en secondes, sont données par le dictionnaire `Darc_etiquette`, dont les clés sont les identifiants des extrémités de l'arc.
On rappelle que le graphe est décrit par le dictionnaire, `Dadj`, des listes d'adjacence de ses sommets.

Q32. Déterminer l'ordre et la taille du graphe G. Déterminer le nombre de sommets qui sont des stations.

Q33. Déterminer les stations dans lesquelles se trouvent le plus grand nombre d'arrêts. Établir qu'elles comportent dix arrêts, et donner leurs noms.

4.1 Parcours en profondeur du graphe

Q34. Définir une fonction `DFSchemin(dAdj, s0, s1)` cherchant et renvoyant, s'il existe, un chemin depuis le sommet s_0 jusqu'au sommet s_1, dans un graphe décrit par un dictionnaire `dAdj`, obtenu par un parcours en profondeur. On adaptera la fonction `DFSaccessibles1`, afin de limiter l'exploration du graphe, à ce qui est nécessaire pour atteindre le sommet final. On s'appuiera sur un dictionnaire des prédécesseurs et on construira le chemin à l'aide de la fonction `reconstruire_chemin`. La fonction renverra le chemin (une liste vide si le chemin n'est pas trouvé) et le dictionnaire des prédécesseurs.

Q35. Tester la fonction entre les stations Rue de la Pompe (`"71292"`) et Delphine Seyrig (`"73800"`). Puis, avec les stations Javel - André Citroën (`"71150"`) et Saint-Fargeau (`"71860"`).

4.2 Parcours en largeur du graphe

On rappelle que l'on peut implémenter la structure de données abstraite de file à l'aide de listes, avec les quatre primitives suivantes :

```
def creer_file_vide():
    return []
def est_vide_file(f):
    return len(f) == 0
```

```
def enfiler(f, v):
    f.insert(0, v)
def defiler(f):
    return f.pop()
```

mais que ces quatre primitives sont de complexité constante, indépendante de n, le nombre d'éléments dans la file, à l'exception de la fonction `enfiler` qui est de complexité $O(n)$.
On préférera donc ici utiliser la structure de données `deque` du module `collections`, qui permet d'instancier une file vide `f` par l'instruction `f=deque()`, de tester si elle est vide par le test `len(f)==0`, d'enfiler une valeur `v` dans `f` par `f.appendleft(v)`, et de défiler une valeur par `f.pop()`.

Q36. Reprendre et adapter le script donné pour la question **Q6** pour définir une fonction `BFSexplore(dAdj, s0)`, renvoyant la liste des sommets accessibles depuis un sommet s_0 dans un parcours en largeur d'un graphe décrit par le dictionnaire de ses listes d'adjacence `dAdj`. Vérifier que tous les sommets du graphe H sont accessibles depuis la station Châtelet d'identifiant `"71264"`.

Q37. Modifier la fonction `BFSexplore(dAdj, s0)` en une fonction `BFSpreds(dAdj, s0)`, renvoyant un dictionnaire dont les clés sont les sommets accessibles depuis le sommet s_0, avec pour valeur associée leur prédécesseur dans le parcours en largeur du graphe représenté par `dAdj` depuis le sommet `s0`. On associera la valeur `None` au sommet `s0`.

Q38. Choisir deux stations au hasard et déterminer un chemin entre ces deux stations s'il existe. On pourra utiliser la variable `liste_ids_stations` et la fonction `reconstruire_chemin`.

On rappelle qu'un parcours en largeur permet d'atteindre les sommets dans l'ordre croissant de leur distance au sommet de départ, la distance étant mesurée en nombre d'arcs depuis le sommet de départ. Cette distance est le nombre d'arcs minimal le long d'un chemin depuis le sommet de départ jusqu'au sommet considéré.

Q39. Modifier la fonction `BFSpreds(dAdj, s0)` en une fonction `BFSpreds_dists(dAdj, s0)`, renvoyant, en plus du dictionnaire des prédécesseurs, un dictionnaire associant à chaque sommet atteint sa distance minimale en nombre d'arcs depuis le sommet `s0`.

Q40. Déterminer la station la plus éloignée de la station Châtelet d'identifiant `"71264"` pour cette distance.

Q41. Modifier la fonction `DFSchemin` (question **Q35**) en la renommant `BFSchemin(dAdj, s0, s1)`, pour l'adapter à un parcours en largeur. La tester avec les mêmes stations qu'en question **Q35**.

4.3 Plus court chemin en temps (algorithme de Dijkstra)

On rappelle qu'il a été défini un dictionnaire, `Dpoids`, ayant pour clés les couples (s, s') correspondants aux arcs du graphe G, associés au temps nécessaire pour aller du sommet s au sommet s' du graphe lorsque cela est possible.
On rappelle que l'algorithme de Dijkstra consiste à se donner un sommet-origine, puis à déterminer les uns après les autres les sommets les plus proches (ici en temps) du sommet-origine.
Pour cela, on peut maintenir un dictionnaire, `distances`, donnant la distance de tous les sommets au sommet-origine. Les clés de ce dictionnaire sont les sommets du graphe, associés chacun au plus petit temps connu pour atteindre ce sommet à partir du sommet-origine. En complément, on peut maintenir un dictionnaire, `Dpreds`, donnant le prédécesseur de chaque sommet dans le plus court chemin trouvé pour aller du sommet-origine à ce sommet. Ce dictionnaire permet de reconstituer un plus court-chemin depuis le sommet-origine jusqu'à un sommet donné.
Initialement, le sommet-origine est à la distance zéro de lui-même, tandis que tous les autres sommets sont à une distance infinie (figurée par la valeur flottante `float('inf')`) du sommet-origine. Comme il n'existe pas de prédécesseur pour atteindre le sommet-origine, on associe au sommet-origine la valeur `None` dans le dictionnaire `Dpreds`. Pour les autres sommets, on leur f
On met également en place un dictionnaire, `distances_min`, initialement vide, auquel on ajoutera, au fur et à mesure, les sommets pour lesquels une distance minimale au sommet-origine a été trouvée, en associant à chaque sommet cette distance minimale.
L'algorithme consiste ensuite, tant qu'il existe des sommets à atteindre (concrètement tant que le dictionnaire `distances` contient un sommet qui n'est pas à une distance infinie du sommet-origine), à :

- rechercher un sommet s à distance minimale du sommet-origine dans le dictionnaire `distances` ;
- enregistrer ce sommet dans le dictionnaire `distances_min` (il ne sera pas possible de l'atteindre par un chemin plus court car cela obligerait à passer par un autre sommet du graphe, nécessairement plus éloigné) ;
- examiner les successeurs de s dans le graphe ;
- pour chaque successeur s' de s : si le chemin depuis le sommet-origine jusque s prolongé par l'arc $s \to s'$, donne une distance inférieure à celle trouvée jusqu'ici, indiquée par la valeur `distance_min[s']`, alors on met à jour, d'une part, la valeur `distance_min[s']`, en la remplaçant par cette distance plus petite, et d'autre part le prédécesseur de s', en donnant à `Dpreds[s']` la valeur s.

On donne ici une fonction permettant de déterminer un sommet situé à distance minimale du sommet-origine dans un dictionnaire `distances` :

```
def get_smin(distances):
    smin, dmin = None, float('inf')
    for s, d in distances.items():
        if d < dmin:
            smin, dmin = s, d
    return smin, dmin
```

et une implémentation de l'algorithme de Dijkstra sous la forme d'une fonction, prenant en argument un dictionnaire des listes d'adjacence du graphe, `Dadj`, un dictionnaire des poids des arcs du graphe, `Dpoids`, un sommet-origine, `s1`, et un sommet-cible, `s2`. Cette implémentation met fin à l'algorithme si le sommet-cible est atteint.

```
def dijkstra(Dadj, Dpoids, s1, s2):
    distances = {s: float('inf') for s in Dadj}
    distances_min = dict()
    distances[s1] = 0
    Dpreds = {s1: None}
    while len(distances) > 0:
        sommetmin, distmin = get_smin(distances)
        if distmin == float('inf'): break
        distances_min[sommetmin] = distmin
        del distances[sommetmin]
        if sommetmin  == s2: break
        for s in Dadj[sommetmin]:
            if s not in distances_min:
                if distmin + Dpoids[(sommetmin, s)] < distances[s]:
                    distances[s] = distmin + Dpoids[(sommetmin, s)]
                    Dpreds[s] = sommetmin
    return Dpreds
```

On trouvera cette implémentation est assortie de commentaires dans le notebook.

Q42. Définir une fonction `chemin_dijkstra`, de mêmes paramètres que la fonction `dijkstra`, faisant appel aux fonctions `dijkstra` et `reconstruire_chemin`, renvoyant le chemin de `s1` à `s2` obtenu à l'aide de l'algorithme de Dijkstra et sa durée s'il existe et une liste vide et un temps infini sinon. Tester la fonction `dijkstra` avec les mêmes stations qu'en question **Q41**.

Q43. À l'aide du dictionnaire `Darc_etiquette`, écrire une fonction Python, `calcul_duree(chemin)`, renvoyant le temps de parcours d'un chemin dans le graphe. On autorise d'utiliser la fonction `sum`. Par appel à cette fonction comparer les temps de parcours des chemins obtenus par l'algorithme de Dijkstra et des chemins obtenus dans un parcours en largeur en **Q41**.

Il y a autant d'itérations de la boucle `while` qu'il y a de sommets accessibles depuis le sommet-origine, et à chaque itération on exécute un nombre borné par une constante d'opérations élémentaires, ainsi qu'un appel à la fonction `get_smin`, de complexité $O(n)$ (n l'ordre du graphe), et une exploration du voisinage du sommet défilé, dont la complexité est en $O(m_s)$ où m_s est le nombre de successeurs du sommet défilé, s. Les opérations préliminaires, dont la construction du dictionnaire distances, sont en $O(n)$. Ainsi la complexité de l'algorithme de Dijkstra est en

$$O(n) + n.O(n) + \sum_{s \in S} O(m_s) = O(n) + O(n^2) + O\left(\sum_{s \in S} m_s\right) = O(n^2) + O(m) = O(n^2 + m),$$

où n est l'ordre du graphe et m sa taille.
Comme $m \leq n^2$, on peut conclure à une complexité en $O(n^2)$.

Or il est possible d'implémenter l'algorithme de Dijkstra sur le modèle de l'implémentation itérative d'un parcours en profondeur ou en largeur, en remplaçant la structure de données de pile ou de file, par une **file de priorité**, pour laquelle les opérations d'ajout et de retrait sont, non pas de complexité constante comme pour les piles ou les files, mais de complexité logarithmique en le nombre d'éléments, ce qui est meilleur que la complexité de la fonction `get_smin`.

Le type file de priorité a été implémenté dans le fichier **TD10_annexe_file_priorite.py** que l'on pourra importer et permet les opérations suivantes :

- on crée une file de priorité vide par appel au constructeur `fp()` ;
- on peut tester si une file de priorité est vide à l'aide de la méthode `.est_vide()` ;
- on ajoute un élément dans la file de priorité sous la forme d'un couple (objet, priorité), en adjoignant à l'élément ajouté (objet) une priorité sous la forme d'une valeur (priorité) de type entier ou flottant, à l'aide de la méthode `.inserer(objet, priorite)` ;
- on retire un élément de la file de priorité à l'aide la méthode `.extraire_min()`. C'est alors nécessairement un élément de priorité la plus faible qui est automatiquement retiré de la file. La méthode renvoie la valeur de la priorité de l'élément retiré et l'élément retiré, dans cet ordre.

On dispose de plus des méthodes suivantes :

- `.obtenir_priorite(objet)` permettant d'obtenir la priorité d'un objet présent dans la file ;
- `.abaisser_priorite`, prenant en argument un objet présent dans la file et une nouvelle valeur de priorité, strictement inférieure à sa priorité actuelle, et lui affectant la nouvelle valeur de priorité.

Ces opérations sont en temps constant, à l'exception des opérations d'insertion et d'abaissement de la priorité qui sont en $O(p)$ où p est le nombre d'éléments dans la file de priorité.
Dans son utilisation pour implémenter l'algorithme de Dijkstra, le type file de priorité permettra de définir une file de priorité dont les éléments sont les sommets atteints du graphe, affectés chacun d'une priorité égale à la plus petite distance déjà déterminée le long d'un chemin depuis le sommet-origine.
On donne le squelette de fonction suivant afin de construire une implémentation de l'algorithme de Dijkstra à l'aide d'une file de priorité. Ce squelette se rapproche d'une implémentation possible de la fonction `BFSpreds_dists` de la question **Q39**. à ceci près que l'on interrompt la recherche si la distance minimale au sommet-cible, `s1`, est obtenue.
Le squelette proposé est construite sur le modèle de la fonction `dijkstra` du préambule du **4.3.** : les dictionnaires `distances_min`, et `Dpreds`, y jouent le même rôle, tandis que la file de priorité, `file_priorite`, se substitue au dictionnaire `distances`.

```
from TD10_annexe_file_priorite import *
```

```
def dijkstra_fp(dAdj, Dpoids, s0, s1):
    # initialisation de la file de priorité
    file_priorite = fp()
    for s in dAdj.keys():
        file_priorite.inserer(s, float('inf'))
    # initialisation du dictionnaire des distances minimales
    distances_min = {}
    # initialisation du dictionnaire des prédécesseurs pour les sommets atteints
    Dpreds = dict()
    # début de la recherche à partir de s0
    s = s0
    file_priorite.abaisser_priorite(s, 0)
    Dpreds[s] = None
    while ... : # terminaison de l'algorithme si toutes les distances minimales
                # pour les sommets accessibles ont été trouvées.
        distmin, sommetmin = ...
        # on termine l'algorithme si la distance minimale est infinie :
        ...
        # ajout du sommet défilé au dictionnaire distances_min
        ...
        # on termine l'algorithme si le sommet objectif est atteint
        ...


        # on explore le voisinage du sommet sélectionné, sommetmin
        for s in Dadj[sommetmin]:
            if s not in distances_min:
                distance_via_sommetmin = distmin + Dpoids[(sommetmin, s)]
                distance_deja_connue = file_priorite.obtenir_priorite(s)
                #mise à jour de la priorité et du prédécesseur de s si nécessaire
                ...
    return Dpreds, distances_min
```

Les opérations préliminaires sont en $O(n)$ (l'insertion des sommets dans la file est en $O(n)$ car on effectue n insertions, mais avec des valeurs de priorité toutes égales), et la boucle `while` est exécutée au plus n fois, mais ne donne lieu qu'à des opérations de complexité constante (l'extraction du minimum est en $O(1)$ et non plus en $O(n)$), à l'exception de l'exploration du voisinage du sommet s retiré de la file, qui, au pire, est en $m_s.O(\log n)$ si l'on abaisse la priorité de chaque successeurs de s, dans la file qui comporte au plus n éléments. Ainsi la complexité est, cette fois, en

$$O(n) + \sum_{s \in S} O(m_s.\log n) = O(n) + O\left(\log n\,.\sum_{s \in S} m_s\right) = O(n) + O(m.\log(n)) = O(n + m.\log(n)),$$

où n est l'ordre du graphe et m sa taille.
Si $n \leq m$, ce qui est le cas si tous les sommets admettent au moins un successeur, on peut conclure à une complexité en $O(m.\log n)$.

Q44. Compléter l'implémentation précédente et vérifier que l'on obtient bien les mêmes résultats qu'en question **Q42** . On définira pour cela une fonction `chemin_dijkstra_fp` analogue de la fonction `chemin_dijkstra`, mais faisant appel à la fonction `dijkstra_fp`.

4.4 Plus court chemin en temps (algorithme A^*)

L'algorithme A^* est une amélioration de l'algorithme de Dijkstra, dans laquelle on affecte une priorité aux sommets, égale à la somme de la distance au sommet-origine et d'une évaluation de la distance au sommet-cible, donnée par une fonction heuristique, qui évalue la distance en temps entre deux sommets au temps nécessaire à parcourir la distance qui sépare les deux stations auxquelles ils appartiennent à une vitesse `vmoy`.
Cette fonction est nommée `heuristique` et est définie dans le fichier **annexe_TD10_fonctions.py**.
On pourra l'importer avec l'instruction `from annexe_TD10_fonctions import *`.
On pourra appeler la fonction avec la syntaxe `heuristique(s1, s2, vmoy)`, où `s1` et `s2` sont deux sommets du graphe et `vmoy` sera utilisée en paramètre de la fonction heuristique et pourra être prise égale à de 20 km.h^{-1}.
On implémentera l'algorithme A^* sous la forme d'une fonction `a_etoile(dAdj, Dpoids, vmoy, s0, s1)`, la fonction heuristique étant utilisée comme une variable globale.
Afin qu'il soit assuré que l'algorithme A^* renvoie bien la distance minimale entre le sommet-origine et le sommet-cible, la fonction heuristique doit être **admissible**, *i.e.* ne doit jamais surestimer la distance

entre un sommet et le sommet-cible. Ce n'est pas le cas ici, car il existe des trajets entre deux arrêts qui se font à une vitesse supérieure à 20 $km.h^{-1}$.

Q45. Donner une implémentation de la fonction `a_etoile`, obtenue en modifiant le code de la fonction `dijkstra_fp`. On affectera initialement à tous les sommets une valeur de priorité infinie, sauf au sommet-origine qui aura une valeur de priorité égale à zéro, puis, au moment de réévaluer la priorité des successeurs d'un sommet défilé, on comparera leur précédente priorité à la somme de leur distance au sommet-origine calculée à partir de celle de leur prédécesseur et de l'évaluation de leur distance au sommet-cible calculée à l'aide de l'heuristique. On prêtera attention au fait que lorsqu'un sommet est extrait de la file, la méthode `.extraire_min` renvoie sa priorité (tenant compte de l'heuristique) et non pas sa distance au sommet-origine, qui faudra calculer à partir de celle de son prédécesseur, sauf pour le sommet-origine dont la distance à lui-même est égale à zéro.

Q46. Tester la fonction `a_etoile` sur les mêmes exemples qu'en questions **Q42** et **Q44**, et constater que, bien que l'heuristique ne soit pas admissible, on obtient les mêmes durées de trajet, donc les durées de trajet optimales, mais observer que le nombre de sommets explorés est moindre (comparer pour cela le nombre d'éléments des dictionnaires `distances_min` renvoyés).