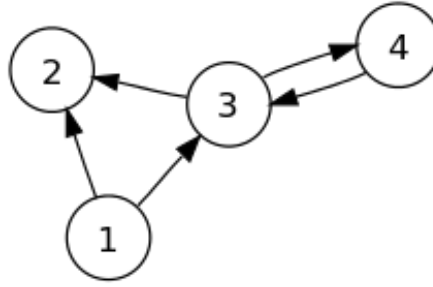


## I Numérisation des graphes

### Exercice 1

Décrire le dictionnaire d'adjacence du graphe.



### Exercice 2

On donne le dictionnaire d'adjacence suivant. Représenter le graphe correspondant.

```

1 d = {
2   'a' : ['b', 'c'],
3   'b' : ['c', 'd', 'e'],
4   'c' : ['e'],
5   'd' : [],
6   'e' : [],
7 }
```

Est-ce un graphe orienté ?

### Exercice 3

Ces graphes sont-ils connexes ? L'un de deux ne comporte pas de cycle. Lequel ?

## II Détection de cycle

Dans cette partie, on considère seulement des graphes non-orientés (ainsi, la matrice d'adjacence est symétrique).



Vous devez placer les fichiers **tp5.py** et **graphe.py** dans un même dossier et utiliser `runfile` pour que l'import de **graphe** fonctionne

### Exercice 4

On souhaite savoir si un tel graphe possède un cycle, c'est à dire une suite d'au moins deux arêtes toutes distinctes telles que le sommet de départ et le sommet final sont égaux. Vu notre numérisation des arêtes, nous devons prendre garde à distinguer deux arêtes distinctes (une même arête du graphe correspond à deux entrées dans notre dictionnaire)

On choisit d'utiliser un parcours en largeur adapté à notre problème :

- pour chaque nouveau sommet découvert, on note en plus depuis quel sommet il a été découvert (on l'appelle sommet parent)
- on détecte la présence d'un cycle lorsqu'on découvre un sommet déjà traité qui n'est pas le parent du sommet courant.

Les sommets parents seront stockés dans un dictionnaire, où les clés et valeurs seront des sommets et on associe à chaque sommet (clé) la valeur du sommet depuis lequel il a été découvert.

Compléter la fonction `cycle_depuis` du fichier **tp4.py**. Cette fonction prend comme arguments un graphe donné par son dictionnaire d'adjacence ainsi qu'un sommet de départ `s` et retourne un booléen indiquant si le parcours en largeur depuis le sommet `s` détecte un cycle par l'algorithme précédent.

## III Étude du graphe des transports en communs de la CRÉA

Dans le fichier **graphes.py** (que vous n'avez pas à ouvrir), se trouve un graphe `G` représentant le réseau Astuce.

Plus précisément `G` est :

- un dictionnaire dont les clés sont des chaînes (qui sont les noms des arrêts)
- les valeurs sont des listes d'arêtes représentées par des *tuples* de longueur 3 : (sommet, étiquette, distance).  
Pour chaque arête, on précise en plus la ligne reliant les deux arrêts (il s'agit de l'étiquette) ainsi que le temps de parcours moyen entre les deux arrêts.

De plus, le module **graphe** contient une liste `stops` qui contient le nom des arrêts.

On y accède grâce à la syntaxe suivante :

```

1 import graphe as g # déjà présent
2 g.stops # liste des arrêts
3 g.G # graphe

```

### Exercice 5

Trouver les arrêts accessibles depuis l'arrêt "Saint-Sever" (ou tout autre arrêt vous convenant, mais prenez garde à l'orthographe), en utilisant des lignes de codes et pas vos yeux.

Choisir un de ces arrêts, identifier la ligne utilisée pour y accéder et poursuivre votre parcours sur cette ligne en utilisant le graphe (vous avez 3 minutes pour aller le plus loin possible du point de départ).

### Exercice 6

Pour trouver des itinéraires dans ce graphe, nous allons utiliser un algorithme classique appelé algorithme de Dijkstra dont voici une implémentation en python

```

1 def dijkstra(G, depart):
2     """
3     G est un graphe annoté : les clés sont les sommets et les valeurs des tuples dont le premier élément
4     est le sommet associé, le deuxième une étiquette et le troisième la distance entre la clé
5     et le sommet cible.
6
7     Retourne un dictionnaire d contenant les distances minimales à s de chaque sommet
8     accessible depuis s ainsi qu'un dictionnaire chemin permettant de reconstituer le chemin à parcourir.
9     Plus précisément, les clés du dictionnaire chemin sont les sommets du graphes et les valeurs associées
10    le prédécesseur dans le plus court chemin depuis s
11    """
12    chemins = {}
13    distances = {k: inf for k in G} # distance infinie avant le parcours
14    distances[depart] = 0 # le sommet de depart est à distance nulle de lui même
15    a_traité = c.deque()
16    a_traité.append(depart)
17    vus = {}
18
19    while len(a_traité) > 0:
20        sommet = a_traité[0]
21        M = distances[sommet]
22        for s in a_traité:
23            if distances[s] < M:
24                M = distances[s]
25                sommet = s
26        for cible in G[sommet]:
27            s, e, d = cible # sommet etiquette distance
28            if s not in vus:
29                if distances[s] == inf:
30                    a_traité.append(s)
31                if distances[s] > distances[sommet] + d:
32                    distances[s] = distances[sommet] + d
33                    chemins[s] = sommet
34            vus[sommet] = True
35            a_traité.remove(sommet)
36    return distances, chemins

```

Expliquer rapidement l'utilité des lignes 20 à 25. Quelle est la différence avec nos méthodes de parcours habituelles ? Le reste du traitement consiste à mettre à jour les distances minimale entre le départ et les sommets découverts de puis le sommet que l'on traite actuellement.

### Exercice 7

Tester la fonction `dijkstra` disponible dans le module `graphe` pour trouver un plus court chemin entre l'arrêt choisi à l'exercice 5 et un autre arrêt de votre choix.

Le résultat n'est pas très lisible. En re-lisant la signification des valeurs de retour, trouver "à la main" (dans la console) la suite d'arrêts donnant le plus court chemin.

### Exercice 8

Grâce au résultat précédent, montrer que le graphe étudié n'est pas connexe (il y a des arrêts non accessibles depuis votre point de départ).

Trouver la liste des arrêts non accessibles depuis votre point de départ, puis vérifier grâce au graphe quelles sont les lignes qui y passent.

### Exercice 9

Compléter la fonction `plus_court_chemin(G, s1, s2)` qui prend comme argument un graphe `G` ainsi que deux sommets et retournant la liste `L` qui donne la suite des sommets à parcourir pour atteindre `s2` en partant de `s1` par le plus court chemin. On aura donc forcément `L[0] == s1` et `L[-1] == s2`.

**Exercice 10**

Compléter la fonction `trajet` qui prend comme argument deux sommets du graphe du réseau astuce et retourne un trajet à suivre. On souhaite en plus de la liste des arrêts (par leurs noms) obtenir les lignes à emprunter (voire en plus les temps de parcours moyens).

**Exercice 11**

Quels sont les deux arrêts les plus éloignés du réseau ? La distance sera calculée en temps minimal de parcours.

## Annexe

**Fonctions et méthodes sur les dictionnaires**

Opération	Exemple	Complexité
Création	<code>d = {cle : valeur}</code>	$O(1)$
Test d'appartenance d'une clé	<code>cle in d</code>	$O(1)$
Ajout d'un couple clé/valeur	<code>d[cle] = valeur</code>	$O(1)$
Valeur correspondant à une clé	<code>d[cle]</code>	$O(1)$
Itération sur les clés	<code>for cle in d:</code>	$O(n)$ , le nombre de paires clé/valeur