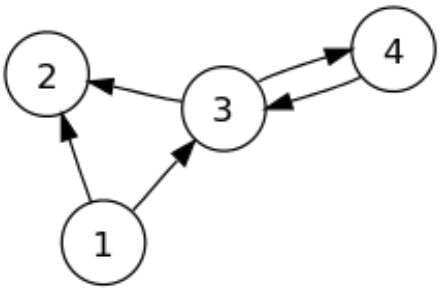
I Numérisation des graphes

Exercice 1
Décrire le dictionnaire d'adjacence du graphe.



Exercice 2
On donne le dictionnaire d'adjacence suivant. Représenter le graphe correspondant.

```
d = {
'a' : ['b', 'c'],
'b' : ['c', 'd', 'e'],
'c' : ['e'],
'd' : [],
'e' : [],
}
```

Est-ce un graphe orienté ?

Exercice 3
Ces graphes sont-ils connexes ? L'un de deux ne comporte par 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.

Compléter la fonction `cycle_depuis` du fichier **tp5.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 des "identifiants", nous en reparlerons lors du cours sur les bases de données)
- 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 un dictionnaire `stops` dont les clés sont les identifiants des arrêts et les valeurs associées le nom de l'arrêt.
On y accède grâce à la syntaxe suivante :

```python
import graphe as g  # déjà présent
g.stops # dictionnaire des arrêts
g.G # graphe
```

Exercice 5
Trouver l'identifiant de 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.

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

```python
def dijkstra(G, depart):
    """
    G est un graphe annoté : les clés sont les sommets et les valeurs des tuples dont le premier élément
    est le sommet associé, le deuxième une étiquette et le troisième la distance entre la clé
    et le sommet cible.

    Retourne un dictionnaire d contenant les distances minimales à s de chaque sommet
    accessible depuis s ainsi qu'un dictionnaire permettant de reconstituer le chemin à parcourir.
    Plus précisément, les clés du dictionnaire chemin sont les sommets du graphes et les valeurs associées
    le prédécesseur dans le plus court chemin depuis s
    """
    chemins = {}
    distances = {k: inf for k in G}  # distance infinie avant le parcours
    distances[depart] = 0 # le sommet de depart est à distance nulle de lui même
    a_traite = c.deque()
    a_traite.append(depart)
    vus = {}

    while len(a_traite) > 0:
        sommet = a_traite[0]
        M = distances[sommet]
        for s in a_traite:
            if distances[s] < M:
                M = distances[s]
                sommet = s
        for cible in G[sommet]:
            s, e, d = cible # sommet etiquette distance
            if s not in vus:
                if distances[s] == inf:
                    a_traite.append(s)
                if distances[s] > distances[sommet] + d:
                    distances[s] = distances[sommet] + d
                    chemins[s] = sommet
        vus[sommet] = True
        a_traite.remove(sommet)
    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
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 9
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 10
Reprendre l'exercice précédent, mais cette fois les arguments sont les noms des arrêts.

Exercice 11
Quel est le plus long chemin qu'on puisse trouver dans notre graphe ? La distance sera calculée en temps de parcours cumulés.