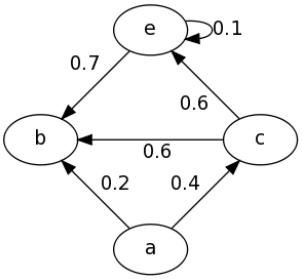
– Implémentation des graphes –



Nous avons déjà rencontré les graphes. Pour rappel : un graphe est constitué de **sommets** et d'**arêtes** (*nodes* et *edges* en anglais).

Dans le graphe ci-contre :

- ⇨ a, b, c et e sont des sommets
- ⇨ Le « segment » $e \rightarrow b$ est une arête

Un graphe peut être orienté ou non. Ci-contre est représenté un graphe orienté : il existe une arête $a \rightarrow b$ mais pas d'arête $b \rightarrow a$.

Deux sommets sont adjacents, ou voisins, si une arête les relie.

Un graphe **non-orienté** est connexe si tous les sommets sont accessibles en suivant un *chemin* partant de n'importe quel autre. Dans le cas des graphes orientés, on utilise la même définition mais sans tenir compte de l'orientation des arêtes. Ainsi, le graphe ci-dessus est connexe.

I. Implémentation

Il existe deux façons classiques d'implémenter un graphe en informatique :

- ⇨ *La liste d'adjacence* : on donne pour chaque sommet la liste de ses voisins (dans un dictionnaire par exemple)

```
Liste = { "a" : [ "b", "c" ],
          "b" : [],
          "c" : [ "b", "e" ],
          "e" : [ "b", "e" ]
          }
```

- ⇨ *La matrice d'adjacence* : on fournit un tableau, une matrice, dans lequel chaque ligne/colonne correspond à un sommet et la valeur d'une case reprend la valeur associée à l'arête joignant les sommets :

	a	*b*	*c*	*e*
a	0	0,2	0,4	0
b	0	0	0	0
c	0	0,6	0	0,6
e	0	0,7	0	0,1

Les méthodes primitives associées au type de données Graphe sont :

renvoie_sommets : Aucun Argument → Liste:
"""
Renvoie la liste des noms des sommets
"""

ajouter_sommet : nom (chaîne) → None:
"""
Ajoute le sommet indiqué par son nom au graphe
"""

ajouter_arete : s1 (chaîne) et s2 (chaîne) → None:
"""
Ajoute une arête entre les sommets de noms s1 et s2
La valeur de l'arête est par défaut 1
"""

adjacent : s1 (chaîne) et s2 (chaîne) → booléen :
"""
Renvoie True si s2 est voisin avec s1
"""

voisins : s (chaîne) → liste:
"""
Retourne la liste des voisins de s
"""

retirer_sommet : s (chaîne) → None:
"""
Retire le sommet s du graphe ainsi que les arêtes associées
"""

retirer_arete : s1 (chaîne) et s2 (chaîne) → None:
"""
Retire l'arête s1-s2 du graphe
"""

Dans le cas de l'implémentation avec la matrice, la méthode *ajouter_arete* donnera par défaut la valeur 1 aux arêtes. On ajoutera de plus les méthodes suivantes :

renvoie_valeur_arete : s1 (chaîne) et s2 (chaîne) → nombre:
"""
Retourne la valeur associée à l'arête s1-s2
(qui peut être différente de celle associée à s2-s1 si on a utilisé set_valeur_arete)
"""

modifie_valeur_arete : s1 (chaîne) et s2 (chaîne) et valeur (nombre) → None:
"""
Affecte la valeur à l'arête s1-s2
"""

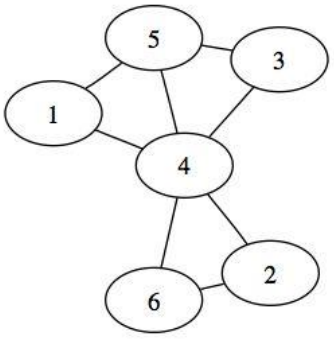
Dans les deux cas une méthode *dessiner* permet de représenter le graphe.

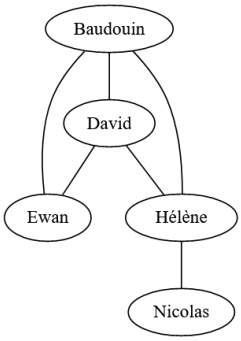
Le fichier graphes.py contient les classes `Graph_as_list` et `Graph_as_matrix` partiellement codées. On fournit en effet :

- ⇨ Le constructeur de chaque classe
- ⇨ L'accesseur `renvoie_liste` et `renvoie_matrice`
- ⇨ Les vérifications des *préconditions* pour chaque méthode

1. Finir le codage des deux classes.

2. Créer les graphes ci-dessous :

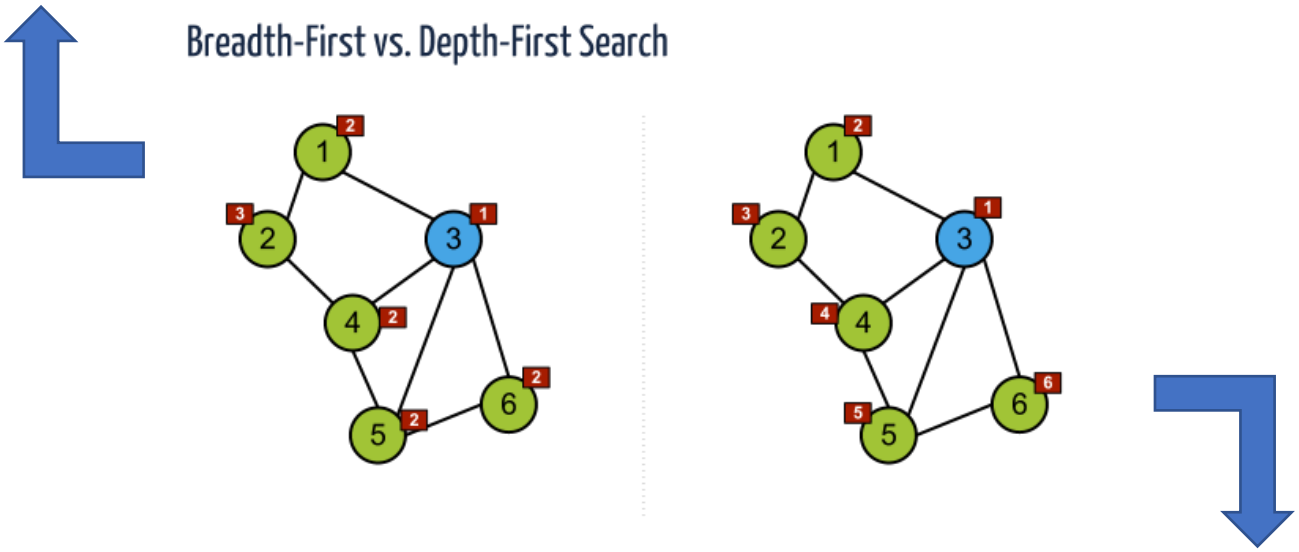




II. Parcours

1. Compléter le fichier `graphes.py` en codant le parcours en largeur d'abord (*breadth-first*) :

```
largeur : G (graphe) et depart (chaîne) → None
    """
    Affiche les sommets visités lors du parcours en largeur d'abord du graphe partant de depart
    """
    f est une File vide
    Enfiler depart dans f
    Marquer depart comme visité
    Tant que f est non vide :
        Défiler le sommet s
        Afficher s
        Pour tous les voisins v de s dans G :
            Si v n'a pas été visité :
                Enfiler v dans f
                Marquer v comme visité
```



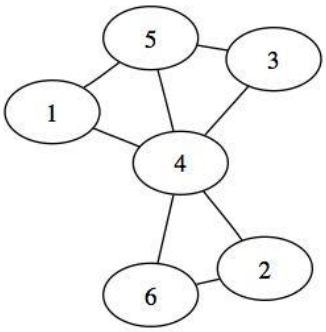
2. Faire de même avec le parcours en profondeur d'abord (*depth-first*) :

```
profondeur : G (graphe) et depart (chaîne) → None
    """
    Affiche les sommets visités lors du parcours en profondeur d'abord du graphe partant de depart
    """
    Marquer depart comme visité
    Afficher depart
    Pour tous les voisins v de depart :
        Si v n'a pas été visité :
            Profondeur(G, v)
```

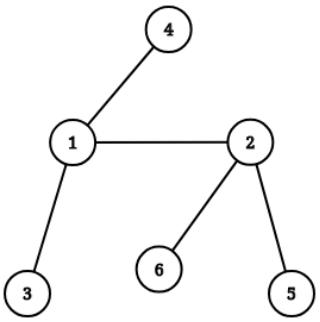
III. Recherche de cycles

La recherche de cycle dans un graphe peut être utile dans le cas où le graphe décrit les dépendances de tâches à effectuer

Graphe cyclique



Graphe acyclique



Une approche classique est d'utiliser un parcours en profondeur « modifié »…