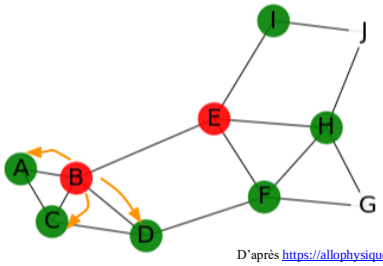
| TNSI | Algorithmique<br>Graphes | TP – **Algorithmes sur les graphes** |
|---|---|---|

Objectifs :

- ⇨ Découvrir les algorithmes de parcours de graphe
- ⇨ Mettre en œuvre un algorithme simple de coloration de graphe




D'après https://allophysique.com

# I – Visualisation d'un graphe

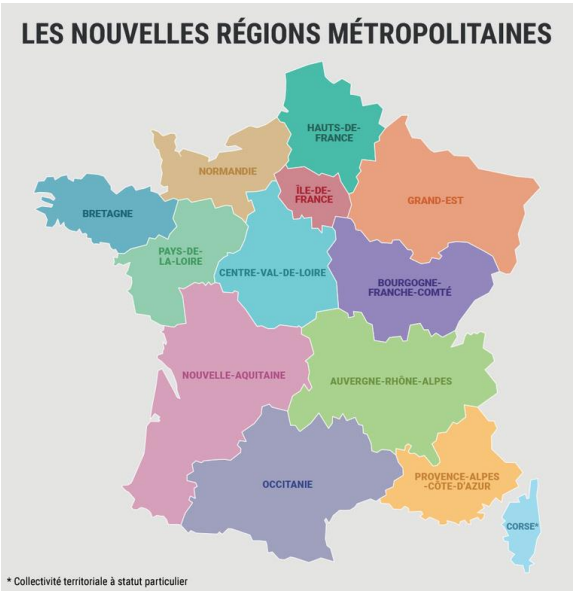
Pour cd TP, nous allons utiliser la bibliothèque graphviz qui permet de créer, manipuler et visualiser simplement des graphes non-orienté (Graph) et orientés (Digraph).

Les méthodes de base pour les graphes sont :

| Instruction | Effet |
|---|---|
| `g=Graph (engine=type)` | Crée une instance de graphe. `engine` est optionnel est peut valoir : circile, dot, fdp, neato, osage, patchwork, sfdp, twopi. |
| `g.node('sommet', 'label', fillcolor="#26c9b3")` | Créer un sommet. Le label est optionnel s'il n'est pas rempli le nom du sommet sera utilisé. `fillcolor` est optionnel et représente la couleur de remplissage (composantes RVB en hexadécimal). **Attention aux types de données, ce sont des types** `str` |
| `g.edge('sommet1', 'sommet2', 'poids')` | Créer une arête ou un arc entre deux sommets. Attention le poids est une chaine de caractère et est optionnel. |
| `print(g)` | Affiche le graphe en mode texte (type dictionnaire d'adjacence). |
| `g.view()` | Pour afficher le graphe en mode graphique. Dans ce cas il faut que l'exécutable de graphviz ait été installé. Celui-ci génère un fichier pdf qui est affiché avec le programme de visualisation par défaut du système. |

Par la suite, on va travailler sur le graphe non-orienté des régions françaises. On donne ci-dessous le dictionnaire d'adjacence de ce graphe :

```
regions = {'Haut': {'Normandie', 'IdF','Est'},
        'Normandie': {'Bretagne', 'Loire', 'Centre', 'IdF', 'Haut'},
        'IdF': {'Normandie', 'Haut', 'Est', 'Bourgogne', 'Centre'},
        'Est': {'Haut', 'IdF', 'Bourgogne'},
        'Bretagne': {'Normandie', 'Loire'},
        'Loire': {'Bretagne', 'Normandie', 'Centre', 'Aquitaine'},
        'Centre': {'Loire', 'Normandie', 'IdF', 'Bourgogne', 'Auvergne',
'Aquitaine'},
        'Bourgogne': {'Centre', 'IdF', 'Est', 'Auvergne'},
        'Aquitaine': {'Loire', 'Centre', 'Auvergne', 'Occitanie'},
        'Auvergne': {'Aquitaine', 'Centre', 'Bourgogne', 'Provence',
'Occitanie'},
        'Occitanie': {'Aquitaine', 'Auvergne', 'Provence'},
        'Provence': {'Occitanie', 'Auvergne'},
        'Corse': {}}
```

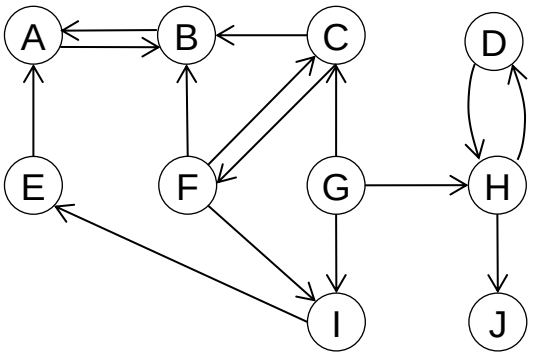



**Question 1** :

1) Ecrire une fonction `succ_vers_Graph` qui prend en argument un dictionnaire de successeurs et renvoie le graphe non-orienté graphviz correspondant en créant les sommets et les arêtes nécessaires. Visualiser le graphe résultant.
2) Le graphe conserve-t-il la position géographique des régions ?
3) Pourquoi les régions sont-elles reliées par deux traits à chaque fois ? Comment peut-on y remédier ?

**Question 2** :
1) Compléter la liste de successeurs représentant le graphe 2 (voir plus bas).
2) Ecrire une fonction `succ_vers_Digraph` qui prend en argument un dictionnaire de successeurs et renvoie le graphe orienté graphviz correspondant en créant les sommets et les arêtes nécessaires. Tester votre fonction avec le graphe 2.

```
graphe2 = {'A': {'B'},
           'B': {'A'},
           'C': {'B','F'},
           'D': {'H'},
           'E': {'A'},
           'F': {'B','C','I'},
           …
           …
           …
           …}
```



Graphe 2

# II - Parcours d'un graphe

Parcourir un graphe signifie passer par tous ses sommets. On peut effectuer le parcours pour de multiples raisons, comme par exemple déterminer s'il existe un chemin menant d'un sommet A vers un sommet B, pour connaître le sommet dont l'étiquette a la plus grande valeur ou pour trouver le chemin le plus court entre deux sommets.

On va donc explorer différents chemins, mais comme on a vu que les chemins pouvaient faire des boucles, il faut s'assurer de ne pas tourner en rond. Pour cela il sera nécessaire de marquer les sommets déjà visités ou de faire une liste des sommets par lesquels on est déjà passés.

Il existe principalement deux façons de parcourir les graphes : le parcours en profondeur (ou parcours en profondeur d'abord : *DFS pour **D**epth-**F**irst **S**earch*) et le parcours en largeur (ou parcours en largeur d'abord : *BFS pour **B**readth-**F**irst **S**earch*).

## 1) Parcours en profondeur

Le principe du parcours en profondeur est le suivant :

1. On part d'un sommet quelconque
2. On marque ce sommet comme visité
3. On fait la liste de tous ses successeurs
4. On visite chaque successeur :
   - Si le sommet a déjà été visité, on passe au suivant
   - Si le sommet n'a pas été visité, on le marque comme visité et on fait la liste de ses successeurs pour les visiter récursivement.

L'algorithme s'arrête quand il n'y a plus de successeur non visité.

Pour la suite du TP, on utilisera le fichier « graphes.py » qui contient la classes `Graphe` les graphes orientés et non-orientés. On se référera à la documentation de la classe pour plus de détails.

**Question 3** :
On souhaite vérifier qu'il existe un chemin entre 2 sommets d'un graphe.

1) Ecrire une fonction `parcours_profondeur` qui prend en argument un graphe `g` et un sommet `s` et renvoie une ensemble contenant tous les sommets accessibles à partir du sommet `s`. Tester votre fonction avec le graphe des régions.
2) Quelle est la complexité de cet algorithme ?

On veut maintenant trouver un chemin entre deux sommets s1 et s2 du graphe.

3) Comment modifier l'algorithme précédent pour pouvoir obtenir le chemin entre deux sommets ? On attend ici une idée, une stratégie et pas (encore) sa mise en œuvre.

Une des méthodes classiques pour obtenir le chemin à partir d'un parcours en profondeur est de mémoriser à chaque sommet son sommet précédent. Ainsi lorsqu'on atteint le sommet de destination, il suffit de « remonter la piste » jusqu'au sommet de départ.
On peut le faire de manière récursive, mais il est possible de le faire de manière itérative en utilisant une pile. L'algorithme devient alors :

```
FONCTION chemin(g : Graphe, s1 : sommet, s2 :sommet)
DEBUT FONCTION
    predecesseurs ← dictionnaire vide
    p ← pile vide
    empiler s1 dans p
    trouve ← faux
    TANT QUE trouve est faux ET p n'est pas vide FAIRE
        s ← élément dépilé du sommet de p
        POUR v parmi les voisins de s FAIRE
            SI v n'est pas dans predecesseurs FAIRE
                empiler v dans p
                ajouter la clé v au dictionnaire predecesseurs avec la valeur s
                SI v  égal s2 FAIRE
                    trouve ← vrai
                FIN SI
            FIN SI
        FIN POUR
    FIN TANT QUE

    SI trouve est vrai FAIRE
        parcours ← liste contenant s2
        s ← s2
        TANT QUE s est différent de s1 FAIRE
            s ← valeur de la clé s du dictionnaire predecesseurs
            ajouter s à la liste parcours
        FIN TANT QUE
        RENVOYER liste parcours inversée
    RENVOYER None
FIN FONCTION
```

4) Ecrire la fonction `chemin(g, s1, s2)` en suivant l'algorithme défini plus haut. Tester avec différents graphes. On pourra importer le fichier « piles.py » qui contient une implémentation de pile.
5) Quel est le chemin indiqué par l'algorithme entre la région Ile-de-France et l'Occitanie dans le graphe des régions ? Quelle est sa longueur ? Existe-t-il un chemin plus court ?

### 2) Parcours en largeur

Dans le parcours en profondeur, on choisit un voisin un peu au hasard puis on part dans cette direction ce qui peut amener à faire des détours. Si on cherche un chemin plus direct, il faut envisager un parcours en largeur : On va visiter en premier les sommets voisins du départ (la *source* s1), donc à une distance de 1. Puis on procède par cercles concentriques autour de la source : les sommets à une distance de 2, puis ceux à une distance de 3, … jusqu'à ce que tous les sommets atteignables aient été visités.

**Question 4** :

1) Quelle stratégie pourrait-on utiliser pour réaliser un parcours en largeur ? On attend ici une idée, une stratégie et pas (encore) sa mise en œuvre.

L'information n'étant pas présente dans le graphe, il va nous falloir construire au fur et à mesure une liste des distances entre le sommet source et les autres sommets. Cela nous permettra également de garder la trace des sommets qu'on a déjà visité.
Comme dans le parcours en largeur des arbres, on va utiliser une file pour gérer l'ordre de traitement des sommets.
Ainsi le principe général est :

1. Enfiler la source s1 dans une file vide. Ce sommet a une distance de 0.
2. Défiler le premier sommet s de la file.
3. Pour chaque voisin de s, s'il n'a pas été visité ou que sa distance est plus grande que la distance de s + 1, alors on l'enfile en lui donnant comme distance la distance de s + 1.
4. Tant que la file n'est pas vide, on recommence au 2.

2) Ecrire une fonction `parcours_largeur` qui prend en argument un graphe `g` et un sommet `s` et renvoie une ensemble contenant tous les sommets accessibles à partir du sommet `s` ainsi que leur distance. Tester votre fonction avec le graphe des régions.

3) En se servant de la fonction précédente, implémenter une fonction `distance(g, s1, s2)` qui renvoie la plus petite distance entre les deux sommets `s1` et `s2` du graphe `g` ou `None` si ces deux sommets ne peuvent être reliés.

## III – Coloration d'un graphe

Colorier un graphe, c'est associer une couleur à chaque sommet. Les problèmes de coloration consistent généralement à associer une couleur différente à chaque sommet adjacent, un peu comme la coloration d'une carte du monde où les pays voisins sont représentés avec une couleur différente pour les distinguer.
Quel est le nombre minimal de couleurs pour colorier un graphe ?
Ce problème n'est pas forcément simple et il a fallu utiliser l'informatique pour prouver le théorème des 4 couleurs : tout graphe peut se colorier avec un minimum de 4 couleurs.
Nous allons ici voir un algorithme simple qui ne réalise pas forcément une coloration optimale du graphe, l'algorithme de Welsh-Powel :
Les couleurs sont numérotées 1, 2, 3 ... (Les couleurs sont les naturels non nuls).

1) Dresser une liste des sommets (dans un ordre décroissant des degrés).
2) Tant que tous les sommets ne sont pas coloriés :
   Choisir le premier sommet non encore colorié de la liste et lui affecter la plus petite couleur possible, non déjà affectée à un sommet adjacent).

Remarque : En modifiant l'ordre des sommets de la liste, on peut parfois obtenir un nombre moindre de couleurs.

**Question 5** :

En reprenant le graphe des régions, réaliser « à la main » la coloration du graphe en suivant l'algorithme de Welsh-Powel.