Algorithmes sur les graphes

Dans ce chapitre nous allons voir plusieurs algorithmes s'appliquant sur des graphes. Leur implémentation en Python utilisée dans ce document se base sur la représentation des graphes par leurs listes d'adjacence, sous forme de dictionnaires. Le but est de compléter la classe « **`Graphe`** » correspondante avec ces algorithmes, et de les tester. Vous pourrez ensuite adapter leur implémentation à la représentation des graphes par leurs matrices d'adjacence. Vous obtiendrez alors deux classes fonctionnelles dont les interfaces seront identiques du point de vue de l'utilisateur.

I./ Parcours en largeur d'un graphe :

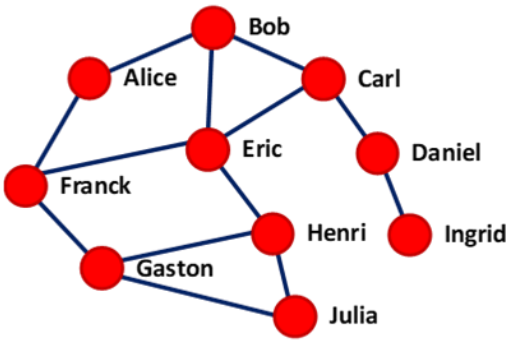
Parcourir un graphe consiste à lister tous ses nœuds une seule fois. Il y a deux façons principales de parcourir un graphe. La première est le *parcours en largeur*.

Le principe de base du parcours en largeur repose sur l'utilisation d'une *file* de la manière suivante :

1. On part d'un nœud de départ.
2. On visite ses voisins directs et on les enfile s'ils ne sont pas déjà présents dans la file.
3. On défile (c'est-à-dire qu'on enlève la tête de la file).
4. On recommence à partir du point 2 tant que la file n'est pas vide.

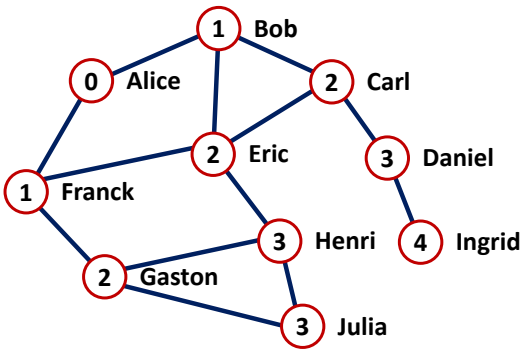
(On remarquera l'utilisation d'une structure de type file comme dans l'algorithme de parcours en largeur d'un arbre binaire).

On va voir le principe de ce type de parcours sur l'exemple suivant :



On choisit tout d'abord un nœud de départ, comme par exemple le nœud 'Alice', qu'on va placer dans la liste des nœuds parcourus. On cherche ensuite les voisins de ce nœud de départ : dans notre exemple il s'agit de 'Bob' et 'Franck'. On place ces nœuds dans la liste des nœuds parcourus. On continue en cherchant les voisins de 'Bob' et de 'Franck' qui ne soient pas déjà dans la liste des nœuds parcourus : on obtient 'Carl', 'Eric' et 'Gaston', etc…

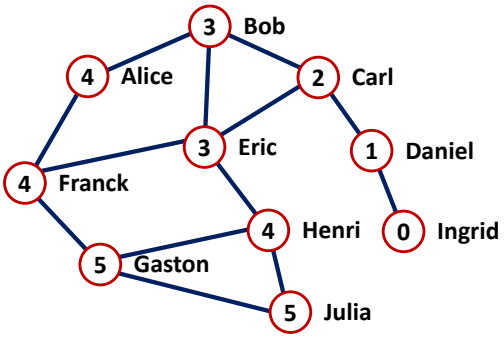
Si on note « 0 » le nœuds de départ, « 1 » ses voisins, « 2 » les voisins suivants, etc… on obtient ceci :



Le parcours de ce graphe à partir du nœud **`'Alice'`** donne donc le résultat suivant :

`['Alice','Bob','Franck','Carl','Eric','Gaston','Daniel','Henri','Julia','Ingrid']`.

Autre exemple : le parcours de ce graphe à partir du nœud **`'Ingrid'`** donne donc le résultat suivant :



`['Ingrid','Daniel','Carl','Bob','Eric','Alice','Franck','Henri','Gaston','Julia']`

Reprendre la classe « **`Graphe`** » tapée précédemment, et rajouter la méthode « **`parcours_en_largeur`** » :

```
    def parcours_en_largeur(self, noeud):
        """Parcours en largeur du graphe à partir du noeud précisé en argument"""
        if not(noeud in self.__noeuds.keys()):
            return None
        F = [noeud]
        resultat = []
        while len(F) != 0:
            S = F[0]
            for voisin in self.voisins(S):
                if not(voisin in resultat) and not(voisin in F):
                    F.append(voisin)
            resultat.append(F.pop(0))
        return resultat
```

Créer le graphe correspondant à l'exemple, et tester le parcours en largeur en commençant par le nœud 'Alice', puis en commençant par le nœud 'Ingrid', afin de vérifier qu'on retrouve bien le même résultat que dans les exemples.

II./ Parcours en profondeur d'un graphe :

L'idée de base du *parcours en profondeur* d'un graphe repose sur l'utilisation d'une *pile* de la manière suivante :

1. On part d'un nœud que l'on empile.
2. On dépile et on ajoute le nœud dépilé dans la liste des nœuds parcourus.
3. On empile chacun des voisins du nœud dépilé qui ne sont pas déjà dans la pile et qui n'ont pas été déjà été traités.
4. On recommence à partir du point 2 tant que la pile n'est pas vide.

On reprend notre exemple et on part du nœud 'Alice'. On empile 'Alice', puis on dépile ce nœud pour le mettre dans la liste des nœuds parcourus. On cherche les voisins du nœud 'Alice' : on trouve 'Bob' et 'Franck', et on empile ces deux nœuds. On dépile le premier élément de la pile, ici 'Franck', puis on l'ajoute à la liste des nœuds parcourus. On cherche les voisins de Franck, et on trouve 'Eric' et 'Gaston', que l'on empile. On dépile le premier élément de la pile, ici 'Gaston', puis on l'ajoute à la liste des nœuds parcourus. On cherche les voisins de 'Gaston', et on trouve 'Henri' et 'Julia', que l'on empile. On dépile le premier élément de la pile, ici 'Julia', puis on l'ajoute à la liste des nœuds parcourus. Les voisins de 'Julia' ont déjà été rencontrés, donc on dépile l'élément suivant, ici 'Henri', que l'on ajoute à la liste des nœuds parcourus. Les voisins de 'Henri' ont déjà été rencontrés, donc on dépile l'élément suivant, ici 'Eric', que l'on ajoute à la liste des nœuds parcourus. On cherche les voisins de 'Eric' n'ayant pas été traités : on trouve 'Carl', que l'on empile. On dépile le

premier élément de la pile, ici 'Carl' et on l'ajoute à la liste des nœuds parcourus. On cherche les voisins de 'Carl' n'ayant pas été traités : on trouve 'Daniel', que l'on empile. On dépile le premier élément de la pile, ici 'Daniel' et on l'ajoute à la liste des nœuds parcourus. On cherche les voisins de 'Daniel' n'ayant pas été traités : on trouve 'Ingrid', que l'on empile. On dépile le premier élément de la pile, ici 'Ingrid' et on l'ajoute à la liste des nœuds parcourus. Les voisins de 'Ingrid' ont déjà été rencontrés, donc on dépile l'élément suivant, ici 'Bob', que l'on ajoute à la liste des nœuds parcourus. La pile étant maintenant vide, le parcours du graphe est terminé.

On obtient donc, en parcourant en profondeur le graphe de l'exemple à partir du nœud 'Alice' :

['Alice','Franck','Gaston','Julia','Henri','Eric','Carl','Daniel','Ingrid','Bob']

Vérifier sur l'exemple que si on effectue le parcours en profondeur sur l'exemple à partir du nœud 'Ingrid', on obtient :

['Ingrid','Daniel','Carl','Eric','Henri','Julia','Gaston','Franck','Alice','Bob']

Rajouter la méthode « **parcours_en_profondeur** » :

```
    def parcours_en_profondeur(self, noeud):
        """Parcours en largeur du graphe à partir du noeud précisé en argument"""
        if not(noeud in self.__noeuds.keys()):
            return None
        P = [noeud]
        resultat = []
        while len(P) != 0:
            S = P.pop()
            resultat.append(S)
            for voisin in self.voisins(S):
                if not(voisin in resultat) and not(voisin in P):
                    P.append(voisin)
        return resultat
```

Tester le parcours en profondeur en commençant par le nœud 'Alice', puis en commençant par le nœud 'Ingrid', afin de vérifier qu'on retrouve bien le même résultat que dans les exemples.

III./ Recherche de chemins :

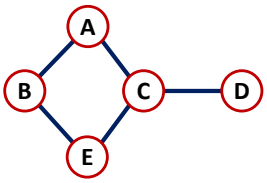
Les algorithmes suivants ne seront pas détaillés. Les première méthode, nommée « **cherche_chemin** », permet de trouver un *chemin entre deux nœuds*, c'est-à-dire la liste des nœuds rencontrés lorsqu'on se déplace du nœud de départ au nœud d'arrivée. La deuxième méthode, nommée « **cherche_tous_chemins** », permet de donner la liste de tous les chemins existants entre le nœud de départ et le nœud d'arrivée.

Rajouter ces deux méthodes à la classe « **Graphe** » :

```
    def cherche_chemin(self, depart, arrivee):
        """Recherche d'un chemin dans un graphe depuis un sommet de départ vers un sommet d'arrivée"""
        if not(depart in self.__noeuds.keys()) or not(arrivee in self.__noeuds.keys()):
            return None
        pile = [(depart, [depart])]
        chemin = []
        while len(pile) != 0:
            noeud, chemin = pile.pop()
            liste_nouveaux_noeuds_voisins = [voisin for voisin in self.voisins(noeud) if not(voisin in chemin)]
            for voisin in liste_nouveaux_noeuds_voisins:
                if voisin == arrivee:
                    return chemin + [arrivee]
                pile.append((voisin, chemin + [voisin]))
        return None
```

```
    def cherche_tous_chemins(self, depart, arrivee):
        """Recherche tous les chemins dans un graphe depuis un sommet de départ vers un sommet d'arrivée"""
        chemins = []
        if not(depart in self.__noeuds.keys()) or not(arrivee in self.__noeuds.keys()):
            return None
        pile = [(depart, [depart])]
        chemin = []
        while len(pile) != 0:
            noeud, chemin = pile.pop()
            liste_nouveaux_noeuds_voisins = [voisin for voisin in self.voisins(noeud) if not(voisin in chemin)]
            for voisin in liste_nouveaux_noeuds_voisins:
                if voisin == arrivee:
                    chemins.append(chemin + [arrivee])
                pile.append((voisin, chemin + [voisin]))
        return chemins
```

Le nombre de chemins existants entre deux nœuds d'un graphe augmente très rapidement avec le nombre de nœuds de ce réseau. Pour tester ces deux méthodes, nous allons donc utiliser un réseau plus simple. Soit le graphe suivant :



Quels sont les chemins permettant de relier 'A' et 'B'. Même question pour les nœuds 'A' et 'E', ainsi que 'A' et 'D' ? A l'aide de la classe « **Graphe** », créer le graphe correspondant à cet exemple, et vérifier vos réponses en utilisant les méthodes correspondantes.

IV./ Recherche de cycles :

Les algorithmes suivants ne seront pas détaillés. Un *cycle* est un chemin particulier où le nœud de départ est aussi le nœud d'arrivée. Les première méthode, nommée « **cherche_cycles** », permet de trouver tous les cycles passant par un nœud particulier. La deuxième méthode, nommée « **cherche_tous_cycles** », permet de donner la liste de tous les cycles existant dans le graphe. En reprenant l'exemple simple précédent, trouver tous les cycles passant par le nœud 'A'. Déterminer ensuite tous les cycles existants dans le graphe. Vérifier vos réponses à l'aide des méthodes correspondantes.

Attention : dans un cycle, le sens de parcours est important. Par exemple, les cycles **[B,C,E,B]** et **[B,E,C,B]** sont deux cycles différents.

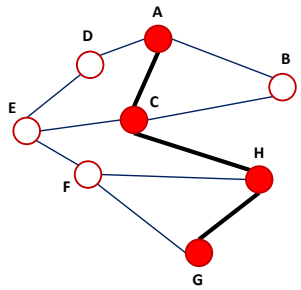
```
    def cherche_cycles(self, noeud):
        """Recherche tous les cycles partant d'un certain sommet dans le graphe"""
        cycles = []
        for voisin in self.voisins(noeud):
            if voisin in self.voisins(noeud):
                chemins = self.cherche_tous_chemins(voisin, noeud)
                for chemin in chemins:
                    if len(chemin) > 2:
                        cycles.append([noeud] + chemin)
        return cycles

    def cherche_tous_cycles(self):
        """Recherche tous les cycles dans le graphe"""
        tous_cycles = []
        for noeud in self.__noeuds.keys():
            cycles = self.cherche_cycles(noeud)
            if cycles != []:
                tous_cycles.append(cycles)
        return tous_cycles
```

V./ Une application pratique : la recherche du chemin optimal dans un graphe :

Il peut être intéressant de connaître le *chemin optimal* existant entre deux nœuds d'un graphe. Par exemple si on veut déterminer la plus courte distance à parcourir entre deux points d'un réseau routier : c'est le principe du guidage par GPS. Le caractère optimal du chemin dépend bien évidemment du critère choisi : cela peut être la distance en kilomètres, le temps en minutes, etc… En règle générale on va vouloir le chemin dont la somme des poids des arêtes parcourues est le plus petit possible.

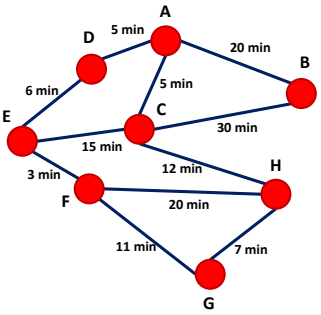
Nous avons vu précédemment des méthodes permettant de trouver un chemin ou tous les chemins possibles entre deux nœuds d'un réseau. Les chemins renvoyés sont des listes contenant les étiquettes des nœuds rencontrés successivement lorsqu'on parcourt ces chemins. Exemple : pour le graphe suivant, un chemin possible pour aller de 'A' à 'G' est : ['A', 'C', 'H', 'G']. Ce chemin n'est pas unique, on a par exemple le chemin suivant : ['A', 'B', 'C', 'E', 'F', 'H', 'G'].



Ecrire une méthode « **`longueur_chemin`** » qui renvoie la longueur du chemin, dans le sens somme des poids des arêtes rencontrées, du chemin passé en argument sous forme de liste.

Ecrire ensuite une méthode « **`cherche_chemin_optimal`** » qui attend en argument deux nœuds du graphe et qui renvoie le chemin optimal existant entre ces deux nœuds. Le chemin optimal correspondra au chemin dont la somme des poids des arêtes est le plus petit. Pour ce faire, on pourra utiliser la méthode « **`cherche_tous_chemins`** » pour obtenir la liste de tous les chemins possibles entre les deux nœuds. On utilisera alors la méthode « **`longueur_chemin`** » pour déterminer lequel de ces chemins a la plus petite longueur.

Prenons l'exemple suivant : (voir chapitre précédent)



La méthode « **`cherche_chemin_optimal`** » avec 'B' comme point de départ et 'E' comme point d'arrivée va nous renvoyer : ['B', 'A', 'D', 'E']. La méthode « **`longueur_chemin`** » appliquée au chemin ['B', 'A', 'D', 'E'] va nous renvoyer 31. On peut vérifier qu'aucun autre chemin ne permet d'aller de 'B' à 'E' en moins de 31 minutes.

Remarques :

- Plusieurs chemins peuvent être optimaux. On peut éventuellement modifier la méthode pour qu'elle les renvoie tous.
- Il existe donc d'autres algorithmes plus efficaces que la simple recherche exhaustive comme *l'algorithme de Dijkstra* dont on a notamment parlé dans le chapitre sur le routage. Cet algorithme n'est pas au programme, mais vous pouvez voir le principe dans cette vidéo : https://www.youtube.com/watch?v=JPeCmKFrKio

2025
Bruno PEYROT

CC BY NC SA