Parcours d'un graphe

1. Principes généraux

Un parcours de graphe est un algorithme consistant à explorer les sommets d'un graphe de proche en proche à partir d'un sommet initial. Parcourir simplement le dictionnaire ou la matrice d'un graphe n'est pas un parcours de graphe. Tous les parcours suivent plus ou moins le même algorithme de base.

On visite un sommet s_1. On note S l'ensemble des voisins de s_1.
Tant que S n'est pas vide :
- on choisit un sommet s de S (*)
- on visite s
- on ajoute à S tous les voisins de s pas encore visités

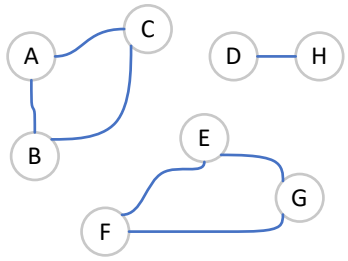
S est l'ensemble des sommets découverts. Le type de parcours dépend du choix effectué lors de l'étape (*) et donc de la structure de l'ensemble S.

Complexité
On suppose que chaque visite est en temps constant. C'est bien le cas si on affiche l'étiquette d'un sommet, mais pas si on calcule la distance du sommet visité aux autres sommets du graphe par exemple. On suppose aussi que chaque test (visités ou pas), chaque ajout à S, chaque choix de sommet (avec son retrait de S) sont en temps constant.
Dans un graphe connexe à n sommets et m arêtes, on visite chaque sommet une fois, on teste chaque sommet une fois par arête incidente. La complexité du parcours est alors $O(n + m)$ qui est un $O(m)$, sauf cas particulier d'un graphe sans arête.

Marquage/mémorisation des sommets visités/découverts
La dernière étape implique que l'on retienne les sommets déjà visités ou découverts. Pour cela :
- On utilise un marquage visité/découvert pour chaque sommet. Cela peut être fait en associant à chaque sommet une couleur, par exemple noir pour visité, gris pour découvert, blanc pour non découvert. La liste de sommets peut être sous la forme [(A, 'blanc'), (B, 'gris'), (C, 'blanc'), (D, 'noir')]. On peut aussi avoir une liste avec les noms et une liste avec les couleurs.

  ou
- On crée une liste ou, plus efficace, un dictionnaire des sommets déjà visités/découverts. Pour un parcours simple, on ne s'intéresse alors qu'aux clés du dictionnaire et les valeurs associées sont sans importance (None, True …), mais on peut aussi stocker comme valeur le précédent sommet dans le parcours, ce qui permettra de reconstituer un chemin.

Connexité
Un parcours de graphe ne visitera que les sommets reliés à s_1 par un chemin. Il permet de visiter la **composante connexe** du graphe contenant s_1.



Ce graphe a trois composantes connexes.

Un parcours au départ de E ne visitera que E, F, G.

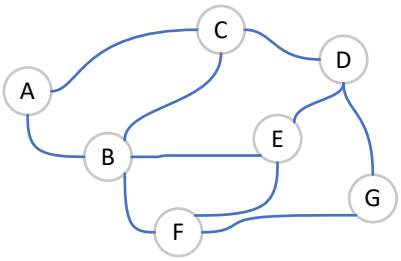
En particulier, un parcours de graphe permet de déterminer si un graphe est connexe. Il suffit de compter le nombre de sommets distincts parcourus pour voir si tous les sommets du graphe sont connectés.

2. Parcours en largeur (BFS, Breadth First Search)

On utilise une **file** pour S, les sommets enregistrés dans S en premier vont être visités les premiers. On va donc visiter d'abord les sommets les plus prêts de s_1 (ceux à distance 1, puis ceux à distance 2, puis 3…). On va utiliser ce parcours pour trouver le sommet le plus prêt de s_1 vérifiant une condition donnée ou le chemin le plus court entre s_1 et un autre sommet.

Une file est une structure de données linéaire, comme les listes et les tableaux du type FIFO (First In First Out, comme une file d'attente) munie des opérations suivantes :

- file_vide() : renvoie une file vide
- enfiler(f, e) : ajoute l'élement e à la queue de la file f
- defiler(f) : supprime l'élément de tête de la file f et le renvoie
- est_vide(f) : renvoie True si la file f est vide, False sinon



Sur ce graphe A-B-C-F-E-D-G est un parcours en largeur au départ de A. B-A-C-E-F-D-G et B-A-F-E-C-G-D sont des parcours en largeur au départ de B.

On marque les sommets découverts afin que chaque sommet ne soit enfilé qu'une fois. Ainsi, lorsqu'on défile un sommet, on sait qu'il n'a pas encore été visité.

```
parcours_largeur(G, s):
    f = file_vide()
    enfiler(f, s)
    découverts = {s}
    tant que f n'est pas vide :
        s = défiler(f)
        pour v dans voisins(G, s):
            si v n'est pas dans découverts :
                ajouter v à découverts
                enfiler(f, v)
```

On pourrait écrire un parcours en largeur en marquant les sommets visités, mais l'algorithme serait moins performant puisqu'un même sommet pourrait être enfiler/défiler plusieurs fois. Il faudrait alors tester si un sommet que l'on défile n'est pas déjà visité.

Une version en Python, en supposant qu'on dispose d'une fonction d'entête `def voisins(G, s):` renvoyant la liste des voisins du sommet s dans un graphe G :

```
from collections import deque   # import de deque pour simuler une file
                                # car enfiler/defiler en temps constant
def parcours_largeur(G, s):
    f = deque([])
    f.appendleft(s)
    decouverts = {s: None}     # utilisation d'un dictionnaire pour
    while f:                   # recherche en temps constant
        s = f.pop()
        print(etiquette(s))    # remplacer print par l'action sur le sommet
        for v in voisins(G, s):
            if v not in decouverts :
                decouverts[v] = s    # stocke le prédécesseur de v pour
                f.appendleft(v)      # pouvoir construire un chemin
```

3. Parcours en profondeur (DFS, Depth First Search)

On utilise une **pile** pour S, les sommets enregistrés en dernier vont être visités en premier : on parcourt le graphe en visitant à chaque fois un voisin du dernier sommet, sauf si celui-ci n'a pas de voisin non visité, auquel cas on remonte au dernier sommet ayant un voisin non visité. C'est le parcours utilisé naturellement par une personne qui explore un labyrinthe.

Une pile est une structure de données linéaire du type LIFO (Last In First Out, comme une pile d'assiette) munie des opérations suivantes :

- pile_vide() : renvoie une pile vide
- empiler(p, e) : ajoute l'élément e sur le sommet de la pile p
- depiler(p) : supprime l'élément du sommet de la pile p et le renvoie
- est_vide(p) : renvoie True si la pile p est vide, False sinon

Sur le graphe précédent, A-B-C-D-E-F-G et F-B-C-D-G-E-A sont des parcours en profondeur.
F-B-C-D-G-A-E n'en est pas un (E a été empilé après A, donc sera dépilé avant).

On marque les sommets visités. Un même sommet peut être empilé plusieurs fois.
En effet, un parcours en profondeur doit permettre de passer d'un sommet à un sommet voisin, et si ce sommet a déjà été empilé en début de parcours, il doit être empilé de nouveau pour devenir immédiatement accessible.

```
parcours_profondeur(G, s):
    p = pile_vide()
    empiler(p, s)
    visités = {}
    Tant que p n'est pas vide :
        s = depiler(p)
        si s n'est pas déjà visité :
            ajouter s à visités
            pour v dans voisins(G, s):
                si v n'est pas dans visités :
                    empiler(p, v)
```

Les listes Python peuvent être utilisées comme des piles, les méthodes pop et append permettant d'empiler et de dépiler en temps constant.
L'opération lst.pop(0) est en temps linéaire, car toute la liste est décalée si on enlève le premier élément, ce qui justifie l'utilisation des deque pour implémenter les files, mais lst.pop() supprime et renvoie le dernier élément en temps constant.

Voici une version Python du parcours en profondeur :

```
def parcours_profondeur(G, s):
    p = [s]
    visites = {}
    while p != []:
        s = p.pop()
        if s not in visites:
            print(etiquette(s))    # ou autre action
            visites[s] = None
            for v in voisins(G, s):
                if v not in visites:
                    p.append(v)
```

Parcours récursif

parcours_recursif(G, s, visités):
  ajouter s à visités
  pour v dans voisins(G, s):
    si v n'est pas dans visités:
      parcours_recursif(G, v, visités)

Version Python :

```
def parcours_recursif(G, s, visites=None):
    if visites == None:
        visites = {}
    visites[s] = None
    print(etiquette(s))
    for v in voisins(G, s):
        if v not in visites:
            parcours_recursif(G, v, visites)
```

C'est un parcours en profondeur. La condition d'arrêt est implicite : si tous les sommets sont déjà visités, les appels s'interrompent.

Ce parcours récursif diffère légèrement de la version itérative. En effet, le premier voisin va être visité d'abord, alors que la version itérative visite le dernier voisin d'abord.

Quelques exemples de codes sur la recherche de cycles

- **Dans un graphe orienté**

Dans un graphe orienté, on parle plutôt de circuit. Un circuit peut être de longueur 2 (deux sommets reliés entre eux dans les deux sens). Pour rechercher un circuit, il suffit de parcourir le graphe en cherchant si le sommet de départ est dans les successeurs d'un sommet. Avec un parcours en largeur par exemple :

```
def recherche_circuit(G, sdepart):
    "Recherche un circuit passant par un sommet sdepart"
    file = deque([sdepart])
    decouverts = {sdepart: None}
    while file:
        s = file.popleft()
        for v in successeurs(G, s):
            if v == sdepart:
                return True
            if v not in decouverts:
                decouverts[v] = None
                file.append(v)
    return False
```

L'expression any(recherche_circuit(G, s) **for** s **in** sommets(G)) va alors renvoyer un booléen indiquant la présence d'un circuit dans le graphe G avec une complexité en O(n(n + m)) où n est le nombre de sommets du graphe et m le nombre d'arêtes.
Un parcours en profondeur fournit une solution plus efficace. La présence d'un circuit se traduit par une arête vers un sommet déjà présent sur le chemin en cours d'exploration. Pour savoir si un sommet déjà visités est sur le chemin en cours d'exploration ou pas, on peut colorier en gris les sommets visités et en noir les sommets qui n'ont plus de successeurs (et ne font plus partie du chemin en cours).

```
def parcours_recursif(G, s, visites, presence_circuit):
    visites[s] = 'gris'
    for v in successeurs(G, s):
        if v not in visites:
            parcours_recursif(G, v, visites, presence_circuit)
        else:
            if visites[v] == 'gris':
                presence_circuit[0] = True
    visites[s] = 'noir'

def acyclique(G):
    s = sommets(G)[0]
    presence_circuit = [False]
    visites = {}
    parcours_recursif(G, s, visites, presence_circuit)
    return not presence_circuit[0]
```

Le code ci-dessus effectue une recherche de circuit dans un graphe connexe avec un seul parcours, donc en O(n + m). Si le graphe n'est pas connexe, on doit parcourir les sommets du graphe pour relancer le parcours sur les sommets non visités.

- **Dans un graphe non orienté**

Un cycle est au moins de longueur 3. Les algorithmes précédents ne sont plus valables puisqu'ils valideraient un aller-retour entre deux sommets. On doit s'assurer qu'on ne provient pas du même sommet en comparant les voisins d'un sommet avec son prédécesseur ou en vérifiant que le cycle est au moins de longueur 3.

Le code suivant va afficher tous les cycles élémentaires (ne passant pas deux fois par un même sommet à part aux extrémités) d'un graphe non orienté G passant par le sommet s :

```
def parcours_cycle(G, s, chemin=None, visites=None):
    if chemin == None:
        chemin = [s]
    if visites == None:
        visites = {s: None}
    for v in voisins(G, s):
        if v == chemin[0] and len(chemin) > 2:
            print(chemin + [v])
        if v not in visites:
            chemin.append(v)
            visites[v] = None
            parcours_cycle(G, v, chemin, visites)
            chemin.pop()
            del visites[v]
```

On peut penser que le dictionnaire visites fait double-emploi avec la liste chemin mais il permet d'effectuer un test d'appartenance en O(1) au lieu de O(n). La complexité de cette fonction s'obtient en ajoutant celle du parcours et celle de l'affichage des cycles, donc O($n + m + nk$) où k est le nombre de cycles.

Si les sommets sont désignés par un numéro, on utilisera un tableau de booléens plutôt qu'un dictionnaire.

Le code suivant indique si un graphe connexe non orienté possède un cycle avec la complexité du parcours, en O($n + m$).

```
def contient_cycle(G, sommet_actif=None, visites=None, predecesseur=None):
    if sommet_actif == None:
        sommet_actif = sommets(G)[0]
    if visites == None:
        visites = {}
    visites[sommet_actif] = 'gris'
    for v in voisins(G, sommet_actif):
        if v not in visites:
            if contient_cycle(G, v, visites, sommet_actif):
                return True
        else:
            if visites[v] == 'gris' and v != predecesseur:
                return True
    visites[sommet_actif] = 'noir'
    return False
```

Dans le cas d'un graphe non connexe, la fonction contient_cycle est à appliquer sur chaque composante connexe, en choisissant un sommet_actif dans les différentes composantes lors de l'appel initial.

L'expression `sommets(G)[0]` renvoie le premier sommet du graphe G. Dans le cas d'une implémentation avec matrice d'adjacence, si les sommets ne sont pas nommés, on la remplace par l'indice du premier sommet, 0 ou 1. Si le graphe est implémenté par un dictionnaire, on peut la remplacer par `list(G)[0]` ou `next(iter(G))`. Cette dernière instruction est en O(1) alors que la première est en O(*n*), ce qui ne change pas l'ordre de complexité final du code.

Exemples de codes sur la connexité

```
def est_connexe(G):
    sdepart = sommets(G)[0]  # à adapter selon l'implémentation
    pile = [sdepart]
    visites = {}
    while pile:
        s = pile.pop()
        if s not in visites:
            visites[s] = None
            for v in voisins(G, s):
                pile.append(v)
    return len(visites) == len(sommets(G))   # ou len(G) selon implémentation
```

Une variante :

```
def est_connexe(G):
    sdepart = sommets(G)[0]
    pile = [sdepart]
    decouverts = {sdepart: None}
    while pile:
        s = pile.pop()
        for v in voisins(G, s):
            if v not in decouverts:
                pile.append(v)
                decouverts[v] = None
    return len(decouverts) == len(sommets(G))
```

Deux questions pour le lecteur : le premier code est un parcours en profondeur, pas le deuxième. Pourquoi ? Pourquoi utiliser deux structures pile et decouverts qui vont contenir les mêmes éléments ?

Un parcours permet de déterminer les composantes connexes d'un graphe non orienté. On peut utiliser un parcours en largeur ou en profondeur.

```
def parcours_profondeur(G, s, visites):
    visites[s] = None
    for v in voisins(G, s):
        if v not in visites:
            parcours_profondeur(G, v, visites)
```

```
def composantes_connexes(G):
    composantes = []
    sommets = {s: None for s in G}  # valable si implémentation dictionnaire
    while sommets:
        s = next(iter(sommets))     # valable si implémentation dictionnaire
        visites = {}
        parcours_profondeur(G, s, visites)
        for s in visites:
            del sommets[s]
        composantes.append(list(visites))
    return composantes
```