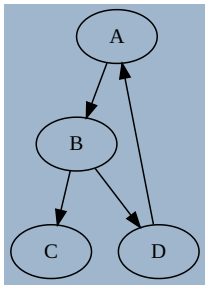
Parcours de graphes et applications

1. Parcours en profondeur

1.1. Comparaison avec un arbre

Le **parcours en profondeur** d'un graphe (*Depth First Search* en anglais), c'est-à-dire un parcours où on explore chaque chemin jusqu'à son extrémité finale, est équivalent à celui pour un arbre comme présenté dans le chapitre idoine, à une subtilité prêt : dans un graphe il est possible de trouver des boucles, ce qui pourrait amener à un chemin infini :

le cas d'une boucle



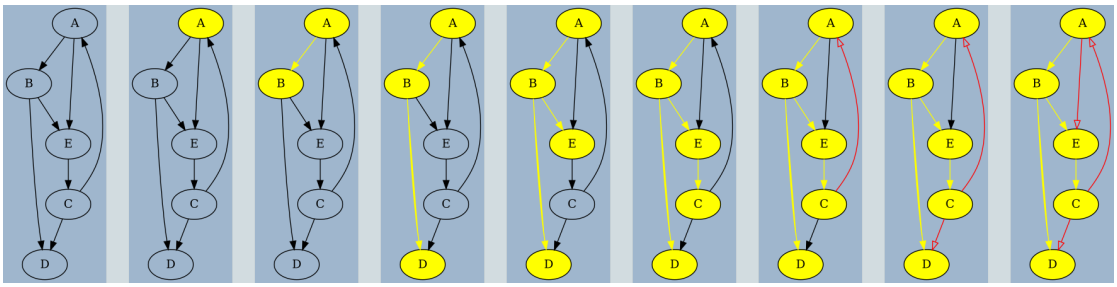
Dans un parcours DFS d'un graphe comme ci-dessus, et en partant de A, on aura un parcours dans l'ordre suivant (en considérant les voisins dans l'ordre alphabétique) :

- On ira de A à B puis de B à C ;
- Une fois en C, il faut remonter au dernier sommet visité, donc B, et regarder si il existe d'autres voisins, donc on passera ensuite à D, puis à A, puis à B, etc.

On obtient donc un parcours infini $A->B->C->D->A->...$, et l'algorithme ne s'arrête pas.

1.2. Algorithme en langage naturel

Pour que l'algorithme puisse fonctionner, il faudra donc *marquer* les sommets déjà visités, comme sur l'exemple suivant :

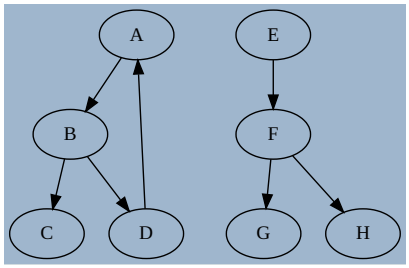


On considèrera qu'il existe une méthode pour *marquer* les sommets, et donc qu'on n'explore plus un sommet déjà marqué.

```
fonction explore_graphe(G, s) :
    Marquer(s)
    pour chaque voisin v de s :
        si v n'est pas marqué :
            explorer_graphe(G,v)
```

Graphes avec de multiples composantes connexes

Cette fonction ne permet pas d'explorer tout le graphe si celui-ci comporte pluiseurs composantes connexes, car on explore uniquement de voisins en voisins :



Sur un graphe tel que ci-dessus, la fonction précédente appelée sur A n'explorera que les sommets A, B, C et D dans cet ordre.

Il faut donc encapsuler la fonction précédente dans une autre fonction :

```
fonction parcours_profondeur(G) :
    pour chaque sommet s de G :
        si s n'est pas marqué :
            explorer_graphe(G,s)
```

1.3. Implémentation en Python

Pour pouvoir implémenter le parcours de graphe en profondeur, il nous faut utiliser une structure permettant de conserver les sommets visités, tout en gardant leur ordre. Une solution évidente est d'utiliser une *liste Python*, et d'utiliser en particulier sa propriété de **mutabilité**.

Hors programme : paramètre mutable d'une fonction

Considérons le code suivant et son exécution dans PythonTutor :

L'objectif est de comprendre la différence entre objets **mutables** et **non-mutables** :

- un objet de type `list` est mutable, et possède une méthode `.append` qui permet de lui ajouter un élément ;
- un objet de type `tuple` est non-mutable. Pour ajouter un élément on est obligé de créer un nouvel objet, qu'on réaffecte au nom `tpl` :

  ```
  tpl = *tpl, n
  ```

  On utilise ici l'opérateur `*` pour `unpacker` (décompacter) les éléments constituant le tuple `tpl`, puis ajouter l'élément `n` au nouveau tuple.

A la fin de l'exécution du code, alors que les méthodes semblent similaires, on constate que les objets *tuples* créés ont disparu à la fin de l'exécution de chaque instance de la fonction `fibonacci`, alors que l'objet de type `list` a lui été modifié au fur et à mesure, et **que ces modifications sont conservées !**

Le paramètre `lst` de la fonction `fibonacci` est donc un **argument mutable**, ce qui :

- est parfois fort pratique ;
- est souvent un **générateur d'effets de bords indésirés** !

Totalement hors programme : utilisation d'une liste en paramètre non-mutable

En réalité, ce n'est pas tant la structure de liste utilisée qui est importante dans l'exemple ci-dessus, mais bien l'utilisation d'une méthode **en place** comme `append`. On peut en effet utiliser une liste comme un paramètre pseudo non-mutable comme dans l'exemple ci-dessous, avec la concaténation de listes :

Au vu de mon expérience, un conseil : **NE FAITES SURTOUT PAS CA** ! Vous vous éviterez des heures de débuggage frustrantes !

Si vous tenez à conserver un paramètre non-mutable, utilisez **TOUJOURS** une structure qui est construite en ce sens...

Implémentation en Python

Enoncé

1. Ajouter à la classe `Graph` (implémentation au choix) une méthode `get_vertices()` renvoyant la liste des sommets *dans l'ordre lexicographique*.
2. Créer une fonction `explore_graph(G, s, explored)` qui explore récursivement le graphe `G` à partir du sommet `s`, connaissant un objet de type `list` `explored` contenant la liste des sommets déjà explorés.
3. Créer une fonction `DFS(G)` renvoyant la liste des sommets explorés, dans l'ordre d'exploration lexicographique (dans l'exemple du graphe à deux composantes connexes ci dessu, on obtiendra `[A, B, C, D, E, F, G, H]`, et dans l'exemple présenté exhaustivement, on aura `[A, B, D, E, C]`).

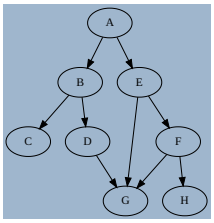
Corrigé

A venir !

2. Parcours en largeur

2.1. Comparaison avec un arbre

Le **parcours en largeur** d'un graphe (*Breadth First Search* en anglais), c'est-à-dire un parcours où on explore chaque chemin jusqu'à son extrémité finale, est équivalent à celui pour un arbre comme présenté dans le chapitre idoine, tout en ajoutant le même problème que pour le parcours en profondeur : il faut marquer les sommets déjà visités.



Ainsi, dans le graphe ci-dessus, l'ordre de parcours des sommets est : A, B, E, C, D, F, G, H.

2.2. Algorithme en langage naturel

L'algorithme itératif s'implémente à l'aide d'une file :

```
fonction explore_largeur(G, s):
       Créer une file f
       Enfiler s dans f
       marquer s
       tant que la file est non vide
            defiler f dans s
            afficher s
            pour tout voisin t de s dans G
                si t non marqué
                    Enfiler t dans f
                    marquer t
```

De même que pour un parcours en profondeur, la fonction ci-dessus ne permet pas un parcours de graphe non-connexe. On complètera donc cette fonction par :

```
fonction parcours_largeur(G) :
    Pour chaque sommet s de G
        si s n'est pas marqué
            explore_largeur(G,s)
```

2.3. Implémentation en Python

Percours en largeur

Enoncé

1. Créer une fonction `explore_width(G,s)` qui effectue un parcours en largeur à partir du sommet `s` du graphe `G` et renvoie les sommets dans l'ordre de visite. Pour éviter d'avoir à réimplémenter nous-même une classe `File`, nous pouvons utiliser le module `queue` de Python et les commanes suivantes :
   - `F = queue.Queue()`, pour créer une file vide ;
   - `F.put(item)` : enfile `item` dans la file `F` ;
   - `item = F.get()` : défile la file `F` et stocke dans `item` ;
2. Créer une fonction `BFS(G)` qui renvoie les sommets dans l'ordre de visite du parcours en largeur.

Corrigé

A venir !

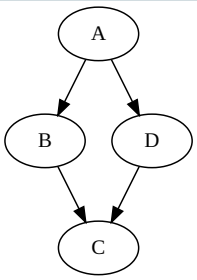
3. Application : Recherche de cycle dans un graphe

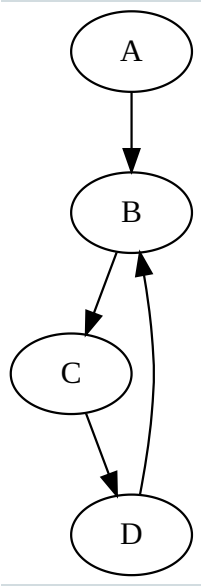
Il est parfois nécessaire de détecter dans un graphe la présence d'un cycle, comme par exemple :

- pour déterminer la longueur de cycle d'un générateur de nombre pseudo-aléatoires ;
- en cryptographie, en particulier pour détecter des collisions dans des fonctions de hachage cryptographiques (CHF) ;
- pour détecter des boucles infinies dans certains programmes, en utilisant une représentation du programme sous la forme d'un graphe (Méthodes formelles) ;
- etc.

Le parcours en profondeur est adapté aux recherches de cycles dans un graphe, mais il faudra lui apporter quelques modifications. En effet, dans le parcours en profondeur, on **marque un noeud avant d'explorer ses voisins**. Quand on retombe sur un noeud déjà marqué, on ne peut pas forcément savoir si c'est à cause de la présence d'un cycle. Prenons comme exemple les deux graphes ci-dessous :

Tips





- Dans le cas du graphe de gauche, on va explorer `A`, puis `B` puis `C`, et donc `C` sera marqué. Puis on retombera sur `C` en venant de `D`, mais sans cycle puisqu'il s'agit de chemins parallèles.
- Dans le cas du graphe de droite, on va explorer `A`, puis `B` puis `C`, et donc `C` sera marqué. Puis on retombera sur `C` en venant de `D`, mais *par un cycle* !

La différence entre les deux situations est que dans le premier cas, *la visite des voisins de `C` est terminée*, alors qu'elle est *toujours en cours dans le deuxième cas*. On va donc devoir **différencier ces deux situations**.

Algorithme de détection de cycles

Nous adopterons une solution en utilisant **3 couleurs** pour marquer ces sommets : blanc, gris et noir. Initialement, tous les sommets seront de **couleur blanche**.

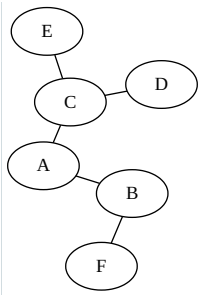
Lorsqu'on visitera un sommet `s` :

- s'il est blanc :
  a. on colore le sommet `s` en gris ;
  b. on visite tous les voisins de `s` récursivement ;
  c. on colore le sommet `s` en noir une fois toutes les étapes précédentes franchies.
- s'il est gris, c'est qu'on vient de découvrir un cycle ;
- s'il est noir, on ne fait rien.

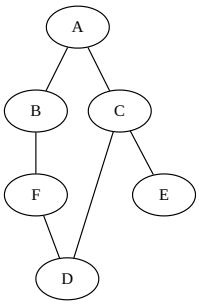
Application

Pour chacun des graphes ci-dessous, appliquez l'algorithme de détection d'un cycle au graphe ci-dessous (vous partirez du sommet de votre choix).

Graphe 1



Graphe 2



Codage en Python

```python
BLANC, GRIS, NOIR = 1, 2, 3

### Un dictionnaire est mutable, donc on peut le modifier par appels récursifs.

def parcours_cycle(graphe : Graph, couleur : dict , s : Sommet) -> boolean :
    if couleur[s] == ... :
        return True
    if couleur[s] == ... :
        return False
    couleur[s] = ...
    for v in graphe.get_neighbours(s) :
        got_cycle = parcours_cycle(g, couleur, s)
        if  got_cycle :
            return True
    couleur[s] = ...
    return False

def cycle(graphe) :
    couleur = {}
    for s in ... :
        couleur[s] = BLANC
    for s in ... :
        if parcours_cycle(graphe, couleur, s) :
            return True
    return False
```