Chapitre (S2) 4 Parcours de graphes non pondérés & Applications

1 Parcours en largeur et profondeur .....
2 Piles et files .....
3 Algorithmes de parcours .....
4 Applications .....

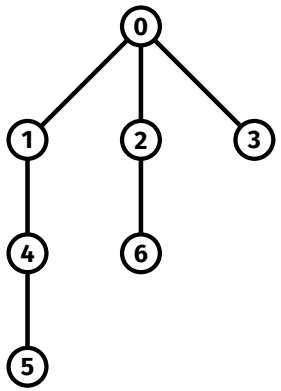
Objectifs

- Connaître les deux types de parcours d'un graphe : parcours en profondeur, parcours en largeur.
- Savoir utiliser les structures de données piles et files à l'aide du module `deque`.
- Savoir mettre en oeuvre les algorithmes de parcours de graphes en utilisant une pile et une file.
- Savoir adapter les algorithmes de parcours pour déterminer un chemin, la distance entre deux sommets, déterminer la connexité et détecter un cycle dans un graphe.

1 PARCOURS EN LARGEUR ET PROFONDEUR

Les tableaux et les listes sont des structures de données *séquentielles* dans le sens où les informations qui y sont stockées peuvent être lues ou modifiées en les *parcourant* les unes à la suite des autres, suivant un *ordre* défini par l'implémentation. En pratique, on se soucie guère de savoir comment les parcourir. Les langages de programmation proposent des instructions qui facilitent ces opérations. C'est le cas des instructions `for` et `while` qui permettent de répéter des blocs d'instruction. Or, parcourir les données d'une structure, c'est justement les passer en revue en répétant une même série d'opérations. Et ces deux instructions le font en suivant l'*ordre* spécifique d'organisation des données dans la structure.

Dans un graphe, les données ne sont plus organisées de manière séquentielle mais de manière *relationnelle* : il existe des liens entre certaines données. Dès lors, comment les parcourir ? Dans quel ordre ? Les réponses à ces questions ne sont plus uniques mais dépendent du point de départ dans le graphe[1] et de l'objectif à atteindre : visiter tous les sommets du graphe, les modifier, en extraire seulement certains. C'est pourquoi deux *parcours de graphes* existent : le *parcours en profondeur*[2] (DFS) et le *parcours en largeur*[3] (BFS). Nous allons les présenter ci dessous.



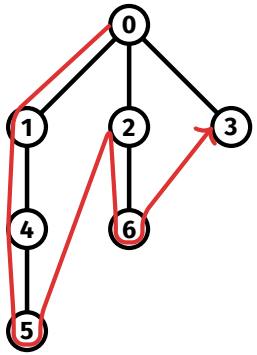
OBJECTIFS. On souhaite définir deux algorithmes permettant de parcourir le graphe précédent de « manière intelligente », selon les principes suivants et à partir d'un certain sommet de départ fixé.

- **[Principe 1 : en profondeur]** Une fois que l'on s'engouffre dans une branche, on poursuit jusqu'à blocage, avant de revenir au premier sommet disponible où il n'y a plus de blocage. Ce principe est utile par exemple pour les problèmes de sortie de labyrinthe, une fois ledit labyrinthe traduit en graphe (chaque case du labyrinthe étant un sommet du graphe).

1. Choix d'un sommet particulier
2. Depth First Search (DFS) en anglais, pour parcours en profondeur d'abord.
3. Breadth First Search (BFS) en anglais, pour parcours en largeur d'abord.

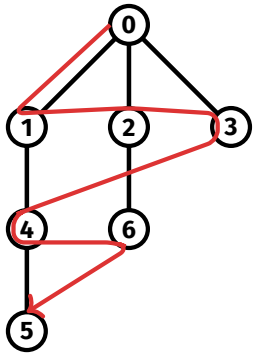
- **[Principe 2 : en largeur]** On parcourt les sommets par « distances croissantes » à partir du sommet d'origine (donc d'abord les sommets à distance 1, puis distance 2, *etc.*). Ce principe nous sera utile pour calculer la distance minimale entre deux sommets, ainsi qu'un chemin minimal.

Si l'on considère commet sommet de départ le sommet 0, cela donnerait les parcours suivants.



Ordre souhaité : $(0,1,4,5,2,6,3)$

(a) Parcours en profondeur



Ordre souhaité : $(0,1,2,3,4,6,5)$

(b) Parcours en largeur

FIGURE 1 : Deux types de parcours

1.1 Parcours en profondeur ou DFS

PRÉSENTATION GÉNÉRALE Dans un parcours en profondeur, à partir d'un sommet de départ, on parcourt des sommets en progressant le plus loin possible dans le graphe suivant un certain *chemin*, tant que cela est possible (on ne peut pas passer plus d'une fois par un sommet donné). Lorsqu'on est bloqué, on revient à un *embranchement* pour parcourir d'autres sommets là encore le plus loin possible dans le graphe, et ainsi de suite jusqu'à avoir parcouru tous les sommets accessibles depuis le sommet de départ. Précisons le vocabulaire :

- à partir d'un sommet courant sur le lequel on se trouve, les sommets voisins sont des sommets *découverts*,
- un sommet que l'on quitte est un sommet *visité*.

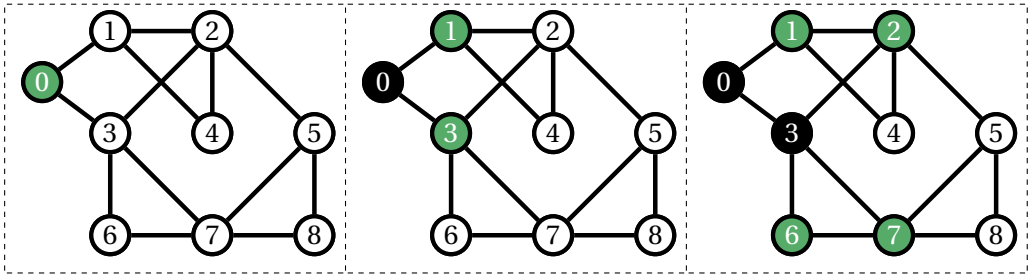
Quand un sommet est marqué comme visité, le parcours ne peut plus passer par lui. Dès lors, soit il est possible d'aller plus avant dans le graphe en découvrant un autre sommet, soit il faut rebrousser chemin jusqu'à revenir à un sommet qui possède des sommets adjacents découverts mais non encore visités.

EXEMPLES Illustrons le *parcours en profondeur* avec le graphe non orienté ci-dessous en adoptant les conventions graphiques de marquages suivantes des sommets : un sommet *non encore découvert* est sur fond blanc, un sommet *découvert* est sur fond vert, un sommet *visité* est sur fond noir.

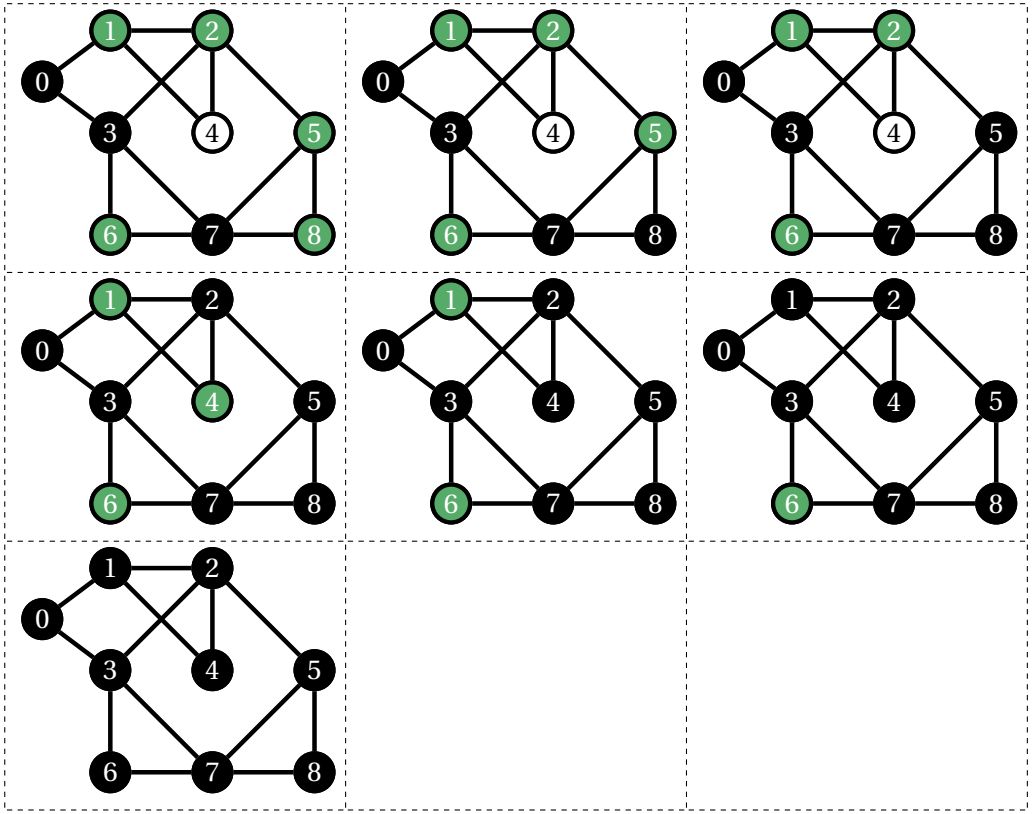
Partant de 0, ce sommet est marqué comme visité. Puis le parcours découvre les sommets adjacents : le sommet 1 et le sommet 3. Choisissons de passer par le sommet découvert de plus grande étiquette[4]. Donc, après 0, le sommet visité est 3. En poursuivant ainsi, les sommets successivement visités sont 7, 8, 5, 2, 4 et 1. À ce niveau du parcours, il n'existe plus de chemin possible. Il faut revenir au dernier sommet visité qui comporte des voisins non encore visités, en l'occurrence le sommet 7. Le parcours reprend en passant le dernier sommet restant 6 et le marque comme visité. Tous les sommets étant alors visités, le parcours est terminé. La figure 2 illustre les différentes étapes de ce parcours.

Avec les choix adoptés, ce parcours fait découvrir les sommets dans l'ordre défini par la liste suivante : `[0, 3, 7, 8, 5, 2, 4, 1, 6]`. Cette liste contient tous les sommets accessibles depuis le sommet de départ (soit ici tous les sommets du graphe), mais elle ne dit pas le chemin qui a été emprunté pour aller du sommet de départ à l'un des sommets de la liste. Pour représenter cette dernière information, on peut représenter l'ensemble des sommets accessibles depuis le sommet de départ, en faisant figurer uniquement les arêtes qui ont permis de passer d'un sommet à l'autre *lors du parcours*. Une telle représentation graphique est appelée *arbre couvrant* et permet de visualiser ce parcours (figure 5).

Figure 2 – PARCOURS EN PROFONDEUR D'UN GRAPHE NON ORIENTÉ.

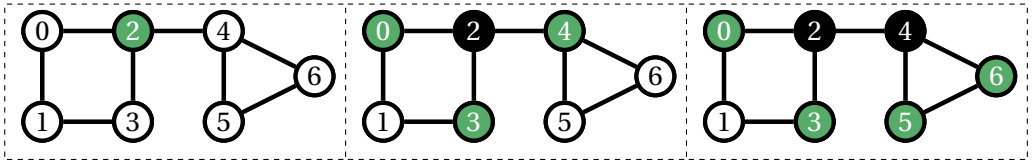


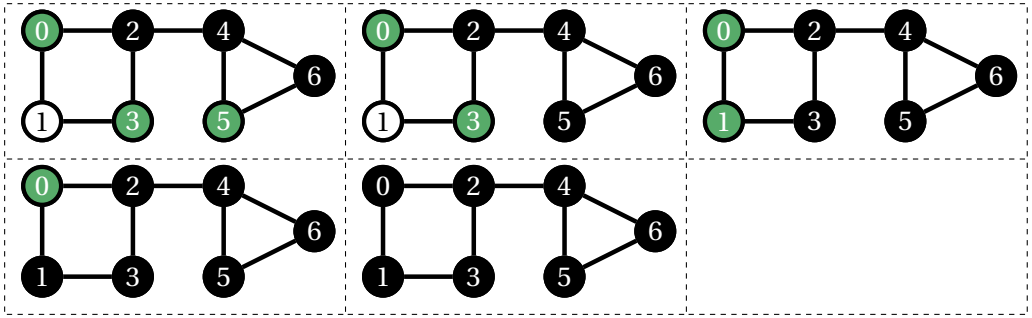
4. Ce choix est purement conventionnel mais a des conséquences sur l'ordre des visites des sommets.



La figure 3 illustre le parcours en profondeur d'un autre graphe non orienté en partant du sommet 2. Les mêmes conventions d'ordre de découverte des sommets sont adoptées. Le lecteur est invité à vérifier que la liste `[2, 4, 6, 5, 3, 1, 0]` définit l'ordre de découverte des sommets. L'arbre associé à ce parcours est donné en figure 5.

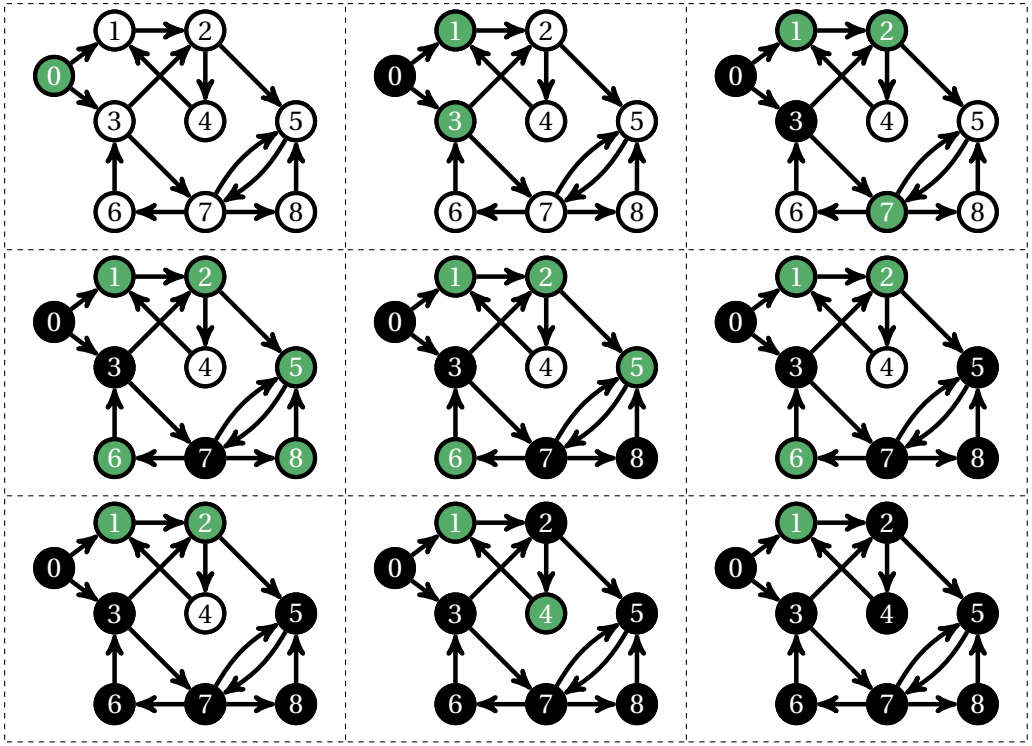
Figure 3 – PARCOURS EN PROFONDEUR D'UN GRAPHE NON ORIENTÉ.



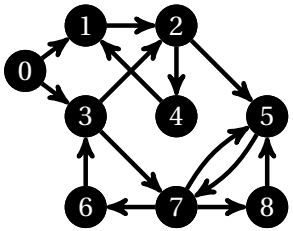


Considérons à présent une version *orientée* du premier graphe introduit dans cette partie. La présence d'orientations contraint à présent les découvertes. La figure 4 illustre le parcours en profondeur depuis le sommet 0 en suivant les règles de découvertes énoncées plus haut. L'arbre associé à ce parcours est donné en figure 5.

Figure 4 – PARCOURS EN PROFONDEUR D'UN GRAPHE ORIENTÉ.

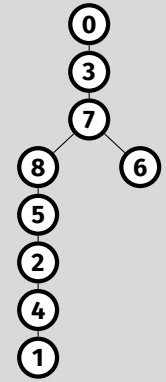


PARCOURS EN PROFONDEUR D'UN GRAPHE ORIENTÉ.

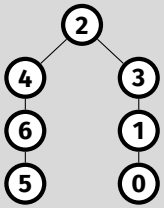


Les figures suivantes présentent les arbres couvrants qui permettent de visualiser les parcours en profondeur relatifs aux trois graphes précédents.

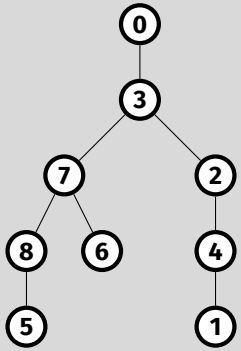
Figure 5 – ARBRES COUVRANTS.



PARCOURS DE LA FIGURE 2



PARCOURS DE LA FIGURE 3



PARCOURS DE LA FIGURE 4

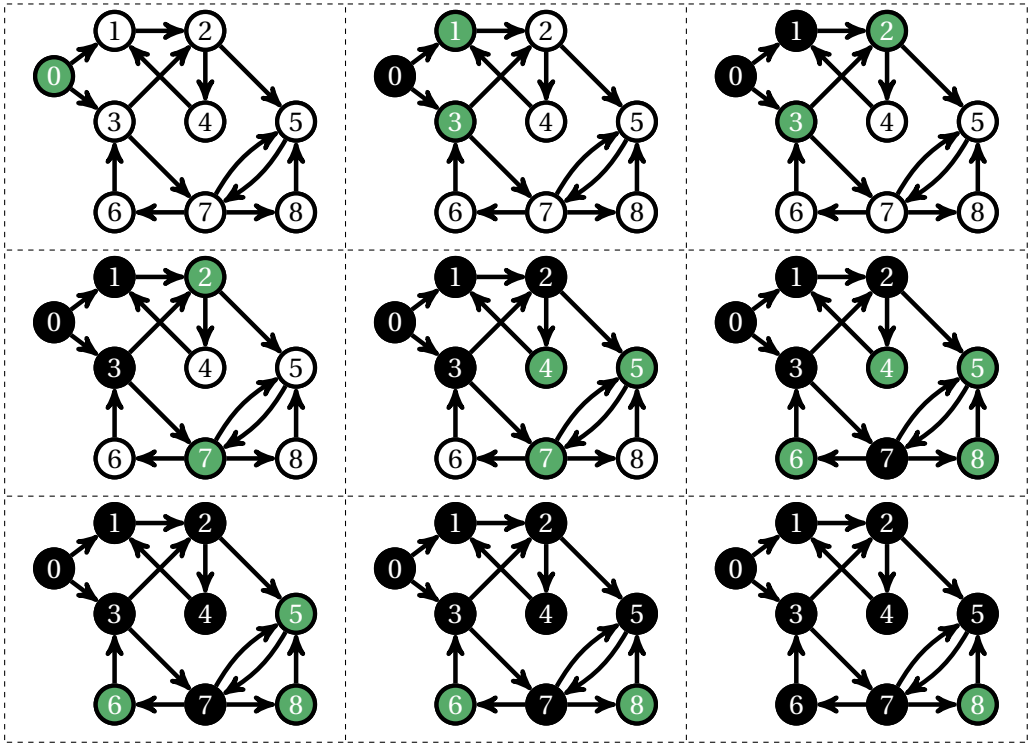
C'est un arbre où chaque sommet n'a qu'un « parent » : le sommet d'où l'on provenait au moment de sa visite.

1.2 Parcours en largeur ou BFS

PRÉSENTATION GÉNÉRALE Le parcours en largeur procède d'une autre façon pour parcourir le graphe. A partir d'un sommet de départ, on découvre les sommets voisins, puis on visite chacun des sommets découverts. Pour chaque nouveau sommet visité, on peut alors découvrir de nouveaux sommets, et l'on itère le processus jusqu'à ce qu'il n'y ait plus de sommets à visiter accessibles depuis le sommet de départ[5].

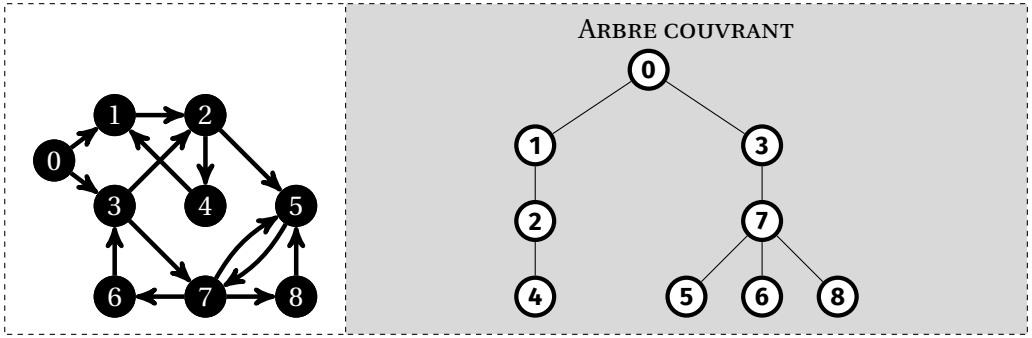
EXEMPLES Pour illustrer le *parcours en largeur*, on reprend le graphe orienté précédent. Partant du sommet 0, la découverte des sommets se fait en découvrant d'abord tous ses sommets voisins. Là encore, un choix doit être fait sur l'ordre de découverte des sommets. On choisit ici de découvrir les sommets non découverts par étiquettes croissantes. Ainsi, le sommet découvert après le sommet 0 est le sommet 1. Puis vient le sommet 3. On doit ensuite visiter les sommets voisins des sommets 1 et 3 non encore découverts. Ce qui mène à la découverte des sommets 2 et 7 respectivement. Et ainsi de suite jusqu'à découvrir tous les sommets de graphe. La figure 6 illustre ce parcours (la cellule grisée correspond à l'arbre couvrant associée au parcours).

Figure 6 – PARCOURS EN LARGEUR D'UN GRAPHE ORIENTÉ



5. On pourrait qualifier BFS de parcours en *pelures d'oignon*.

PARCOURS EN LARGEUR D'UN GRAPHE ORIENTÉ



1.3 Liste d'attente des sommets à visiter

Pour chacun des parcours précédents, on voit que l'on progresse dans le graphe en découvrant des sommets, puis en choisissant de visiter ces sommets découverts selon une logique qui dépend du parcours. Il est donc nécessaire de disposer une *liste d'attente des sommets à parcourir*, qui permettra notamment de gérer les retours en arrière (ou embranchements) dans le parcours en profondeur, et de parcourir les sommets par « couches successives » pour le parcours en largeur. Cette liste d'attente peut être réalisée par deux structures de données nouvelles, à savoir les *piles* et les *files*.

2 PILES ET FILES

2.1 Généralités

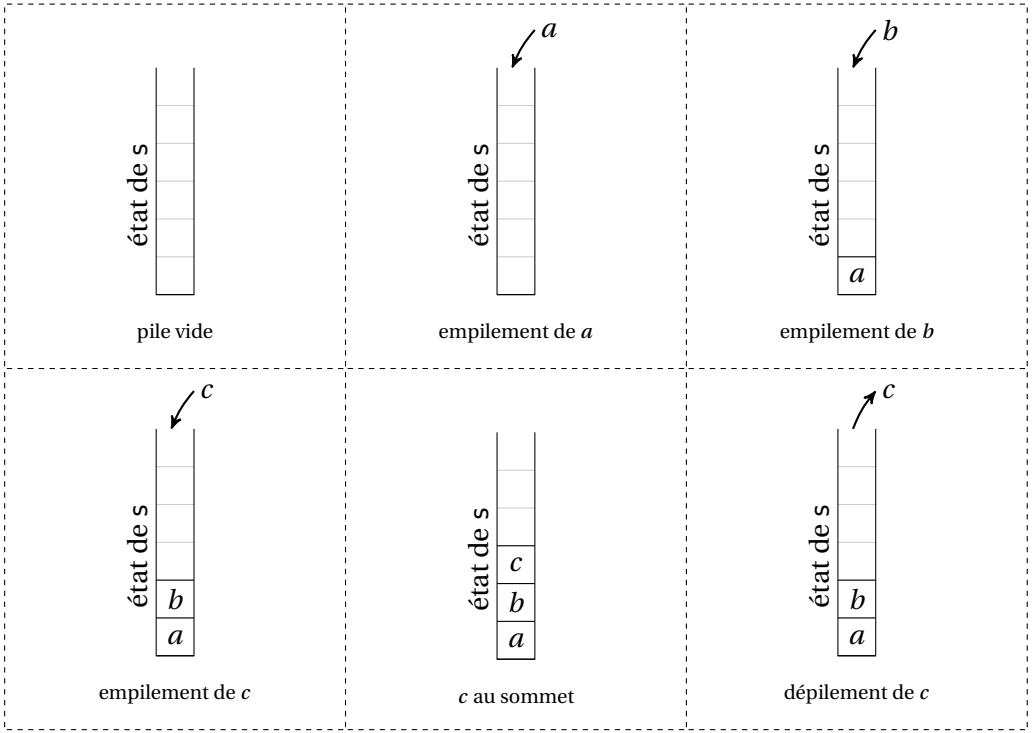
Les *piles* *(« Stack » en anglais)* et les *files* *(« Queue » en anglais)* sont des structures de données linéaires pouvant contenir des données de types divers. Mais à la différence des tableaux ou des listes, les données ne peuvent être ajoutées ou supprimées de ces structures qu'en suivant des procédures spécifiques.

SUR LES PILES Dans les *piles*, une information ne peut être ajoutée ou retirée qu'au niveau de ce qu'on appelle le *sommet* de la pile. En *représentant* une pile comme une superposition de cases mémoires (figure 7), seule celle située au sommet de cette représentation est accessible. Toutes les autres ne sont pas directement accessibles mais le deviennent dès que celles situées « au-dessus » ont été retirées.

On dit qu'une pile suit le principe *LIFO* : *Last In First Out*[6]. Les opérations d'ajout et de suppression d'une information sont appelées *empilement* et *dépilement*.

On peut s'interroger sur l'intérêt réel d'une telle structure de données mais certains composants électroniques ne font rien d'autres que ces opérations élémentaires. Et ils le font de manière très efficace. La figure 7 illustre la représentation d'une pile et les opérations d'empilement et de dépilement.

Figure 7 – REPRÉSENTATION D'OPÉRATIONS D'EMPILEMENT ET DE DÉPILEMENT

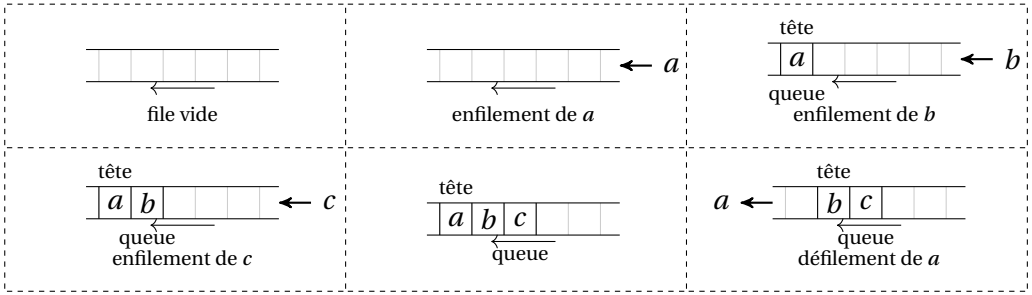


SUR LES FILES. Dans les *files*, une information ne peut être ajoutée qu'au niveau de ce qu'on appelle la *queue* de la file et ne peut être retirée qu'au niveau de la *tête* de la file. Là encore, une *représentation* (figure 8) peut aider à fixer les idées. Si une file est représentée comme une succession de cases mémoires, seules celles situées à ses deux extrémités sont accessibles, l'une pour y ajouter une information, l'autre pour en retirer une. Toutes les autres ne sont pas directement accessibles mais le deviennent dès que celles situées « avant »[7] ont été retirées. On dit qu'une file suit

6. Dernier entré, premier sorti.
7. C'est-à-dire situées entre la tête de la file et l'élément à retirer.

le principe *FIFO* : *First In First Out*[8]. Les opérations d'ajout et de suppression d'une information sont appelées *enfilement* et *défilement*.

Figure 8 – REPRÉSENTATION D'OPÉRATIONS D'ENFILEMENT ET DE DÉFILEMENT



Des fonctions, autres que celles d'ajout ou de retrait d'une information sont disponibles, comme créer une structure *vide*, savoir si elle est vide ou éventuellement pleine[9], ou encore connaître son nombre d'éléments.

Il importe de retenir que ces structures de données linéaires sont *non indexées* : il n'est pas possible d'accéder à une information à l'aide d'un indice, cette dernière notion n'ayant d'ailleurs pas vraiment de sens dans un tel contexte. L'*interface*[10] de chaque structure pourrait être définie de manière simplifiée comme suit.

Pile	File
Création d'une pile vide	Création d'une file vide
Empilement	Enfilement
Dépilement	Défilement
Tester si une pile est vide	Tester si une file est vide

La suite de cette partie présente différentes implémentations des ces deux structures de données en Python. Deux implémentations utilisent la structure de données préexistante du langage Python : les listes Python. Deux autres utilisent un module spécifique qui optimise les complexités des traitements.

8. Premier entré, premier sorti.
9. Dans le cas où l'espace mémoire alloué est imposé.
10. À savoir, la description des fonctions de définition et de manipulation d'une structure de données.

2.2 Implémentation par listes Python

Les listes Python disposent de fonctions qui répondent au cahier des charges précédent. En fait, les listes Python font bien plus que répondre aux besoins. Il s'agit ici d'un « détournement » de leur rôle premier en vue de répondre à des besoins spécifiques. Les complexités temporelles sont données en fonction du nombre n d'éléments présents dans la structure.

CAS DES PILES. L'implémentation d'une *pile* correspond à l'utilisation d'une liste Python dont la dernière case remplie définit le sommet de la pile. Toutes les opérations désirées sont alors de complexité constante[11].

Pile	Instruction	Complexité
Création	`s = []`	$O(1)$
Empilement	`s.append('a')`	$O(1)$
Dépilement	`s.pop()`	$O(1)$
Test pile vide	`s == []`	$O(1)$

CAS DES FILES. L'implémentation d'une *file* correspond à l'utilisation d'une liste Python dont la première case correspond à la tête de la file, la dernière case à la queue de la file[12].

File	Instruction	Complexité
Création	`q = []`	$O(1)$
Enfilement	`q.append('a')`	$O(1)$
Défilement	`q.pop(0)`	$O(n)$
Test file vide	`q == []`	$O(1)$

On note que l'opération de défilement telle que choisie ici est de complexité non constante, linéaire en la taille de la liste (en effet, les éléments d'une liste étant indexés, la suppression de l'élément indexé par 0 entraine de fait la ré-indexation de tous les éléments suivants, soit $O(n)$ opérations). Ce point constitue un obstacle à

11. En toute rigueur, il conviendrait de préciser que le coût de l'ajout d'un élément est de complexité *amortie* constante, c'est-à-dire presque toujours constante excepté lorsque la liste doit être redimensionnée pour pouvoir stocker plus d'éléments qu'elle ne le pouvait initialement.
12. Il ne s'agit que d'un choix particulier. Définir la première case de la liste comme la queue de la file et sa dernière case comme la tête de la file est un choix tout aussi pertinent.

l'utilisation d'une telle implémentation pour une file. Idéalement, une complexité constante de défilement est préférable.

2.3 Module spécifique et « Double Ended Queue »

Le module `collections` de Python définit des objets `deque` *(pour « Double Ended Queue »)* qui implémentent une structure de données linéaire généralisant les piles et les files dans laquelle l'ajout et le retrait d'une information peuvent se faire à coût pratiquement constant aux deux extrémités de la structure. Par construction,

- elle permet l'implémentation d'une pile en définissant l'une des extrémités comme le sommet de la pile.
- Elle permet l'implémentation d'une file en définissant l'une des extrémités comme la tête de la file et l'autre extrémité comme sa queue. Pour importer la structure, le code doit comporter la ligne suivante.

```
from collections import deque
```

Les fonctions de manipulation utiles pour nos besoins sont les suivantes.

Deque	Instruction	Complexité
Création	`d = deque()`	O(1)
Ajout à gauche	`d.appendleft('a')`	O(1)
Ajout à droite	`d.append('b')`	O(1)
Retrait à gauche	`d.popleft()`	O(1)
Retrait à droite	`d.pop()`	O(1)
Test vide	`len(d) == 0`	O(1)

Dans la suite de ce chapitre, nous adoptons les conventions suivantes de définition des piles et des files.

Pile	Instruction
Création	`s = deque()`
Empilement	`s.append('a')`
Dépilement	`s.pop()`
Test pile vide	`len(s) == 0`

File	Instruction
Création	`q = deque()`
Enfilement	`q.append('a')`
Défilement	`q.popleft()`
Test file vide	`len(q) == 0`

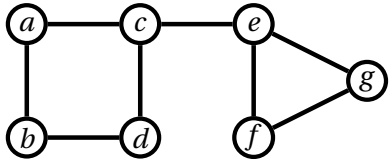
Remarque 1 Python propose un module natif (on dit souvent un *built-in module* pour préciser que le module fait partie intégrante des fonctionnalités fournies par défaut avec le noyau Python) nommé `queue` qui permet la définition d'objets `Queue` pour les files et d'objets `LifoQueue` pour les piles. Le lecteur intéressé peut lire la documentation sur le site officiel de Python (`https://docs.python.org/3/library/queue.html`).

3 ALGORITHMES DE PARCOURS

La mise en oeuvre informatique des parcours précédents requiert l'utilisation des piles et des files présentées précédemment. En stockant les sommets découverts, ces dernières donnent la possibilité de « mémoriser » les sommets sur lesquels il convient de revenir pour avancer dans un parcours.

3.1 Parcours en profondeur « DFS »

Illustrons notre propos avec le *parcours en profondeur* du graphe non orienté ci-dessous étiqueté par des caractères, implémenté par un dictionnaire `grph`.




```
grph = {
    'a' : ['b', 'c'], 'b' : ['a', 'd'],
    'c' : ['a', 'd', 'e'], 'd' : ['b', 'c'],
    'e' : ['c', 'f', 'g'], 'f' : ['e', 'g'],
    'g' : ['e', 'f']}
```

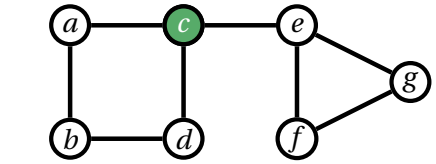
FIGURE 9 : Graphe non orienté et son dictionnaire d'adjacence.

On crée un *dictionnaire des sommets visités*, dont les clés sont les étiquettes du graphe et dont les valeurs initiales sont **`False`**, et qui permet de marquer un sommet visité en passant sa valeur à **`True`**.

Lors du parcours du graphe, les sommets voisins à un sommet et non encore visités sont stockés dans une pile dans l'ordre de leur lecture dans la liste. Cet empilement correspond à l'étape de *découverte* d'un sommet. Chaque fois qu'un sommet est dépilé, il est marqué comme *visité*, s'il ne l'est pas déjà.

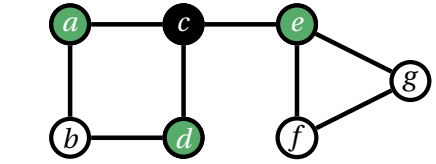
```
# dictionnaire de booléens des sommets visités
visited = {x : False for x in grph}
# création d'une pile vide
s = deque()
```

Choisissons le sommet c comme sommet de départ d'un parcours en profondeur. Reprenant le vocabulaire des parties précédentes, ce sommet est *découvert* et son étiquette est *empilée* dans `s`. Le schéma suivant montre l'état de la pile et du graphe à ce moment du parcours.

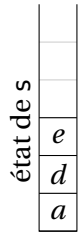


sommets visités : `[]`

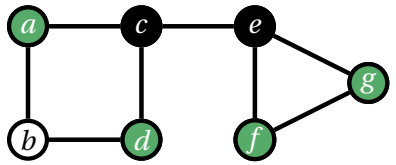
L'étape suivante dépile c de `s` et le marque comme *visité*. Puis ses sommets adjacents sont découverts et empilés dans l'ordre de lecture séquentielle de la liste : a d'abord, d ensuite, e enfin. Ce qui mène à l'état de la pile représenté ci-contre.



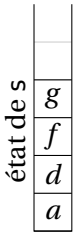
sommets visités : `['c']`



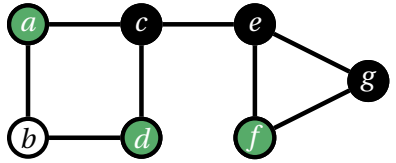
Le sommet e est alors dépilé et marqué comme *visité*. Ses sommets voisins non encore visités sont empilés. Le sommet c n'est donc pas empilé. Les sommets f et g le sont dans cet ordre.



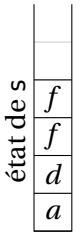
sommets visités : `['c','e']`



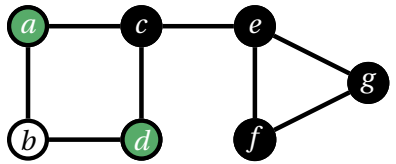
Le sommet g est ensuite dépilé et marqué comme *visité*. Ses sommets voisins non encore visités sont empilés. Il n'y a qu'un seul sommet, déjà découvert mais non encore visité, à empiler : le sommet f.



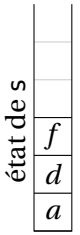
sommets visités : `['c','e','g']`



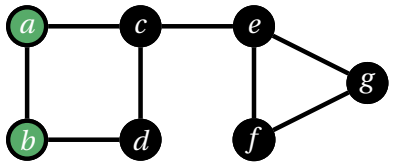
Le sommet f est dépilé et marqué comme *visité*. Il ne comporte aucun sommet adjacent non encore visité. Rien n'est empilé.



sommets visités : `['c','e','g','f']`

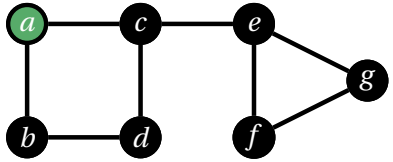


Le sommet f est dépilé à nouveau. Comme il a déjà été visité, on passe au sommet suivant de la pile qui est le sommet d. Ce dernier est dépilé et marqué comme visité. Ses sommets adjacents non encore visités sont découverts et empilés. Seul le sommet b est empilé.



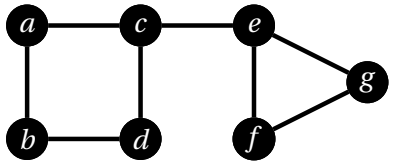
sommets visités : `['c','e','g','f','d']`

Le sommet b est dépilé et marqué comme visité. Le sommet a est empilé, seul sommet voisin de b non encore visité.



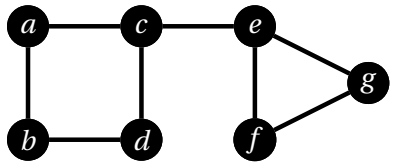
sommets visités : `['c','e','g','f','d','b']`

Le sommet a est dépilé. Comme il ne comporte aucun sommet voisin non encore visité, aucun autre sommet n'est empilé.



sommets visités : `['c','e','g','f','d','b','a']`

Le sommet a est de nouveau dépilé. Étant déjà marqué somme visité, rien n'est fait. La pile est alors vide. Il n'y a plus aucun sommet à découvrir. Le parcours du graphe est terminé.

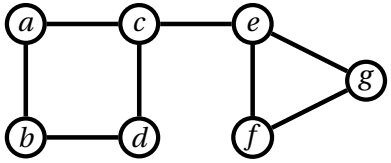


sommets visités : `['c','e','g','f','d','b','a']`

Exercice 1 **Exemple de parcours DFS** [Sol 1] Sur le même principe, appliquer un parcours DFS au graphe orienté présenté en figure 4, en partant du sommet 0 et en précisant l'état de la pile et la liste des sommets visités à chaque itération.

3.2 Parcours en largeur « BFS »

Pour le parcours en largeur, on applique la même méthode en remplaçant la pile servant à stocker les sommets découverts par une file. On peut à nouveau illustrer le fonctionnement du parcours en largeur sur le graphe ci-dessous, et en précisant l'état de la file à chaque itération.



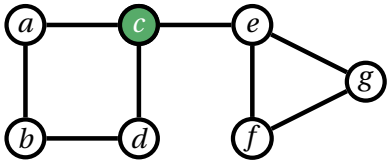

```
grph = {
    'a' : ['b', 'c'], 'b' : ['a', 'd'],
    'c' : ['a', 'd', 'e'], 'd' : ['b', 'c'],
    'e' : ['c', 'f', 'g'], 'f' : ['e', 'g'],
    'g' : ['e', 'f']}
```

FIGURE 10 : Graphe non orienté et son dictionnaire d'adjacence.

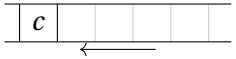
Comme précédemment, on crée un dictionnaire des sommets visités, et on initialise une file vide, nécessaire pour ce parcours en largeur.

```
# dictionnaire de booléens des sommets visités
visited = {x : False for x in grph}
# création d'une file vide
q = deque()
```

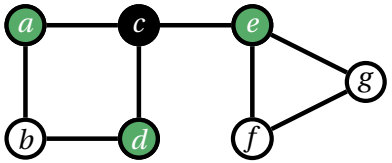
On part du sommet c, ce sommet est *découvert* et son étiquette est *enfilée* dans q. Le schéma suivant montre l'état de la file et du graphe à ce moment du parcours.



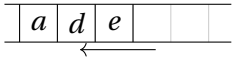
sommets visités : [ ]



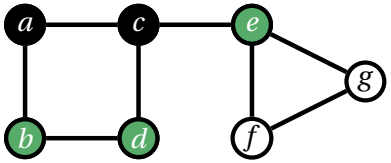
L'étape suivante défile c de q et le marque comme *visité*. Puis ses sommets adjacents sont découverts et enfilés dans l'ordre de lecture séquentielle de la liste : a d'abord, d ensuite, e enfin. On obtient donc le résultat suivant



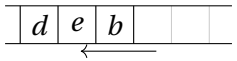
sommets visités : ['c']



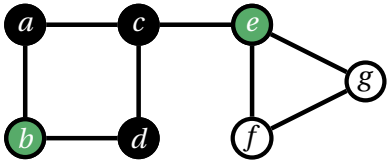
On défile le sommet a et on le marque comme *visité*. On enfile les voisins de a qui n'ont pas encore été visités : seul b est enfilé.



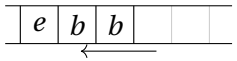
sommets visités : ['c','a']



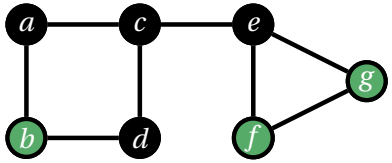
Le sommet d est défilé et marqué comme visité. Le sommet b, voisin de d et non encore visité, est à nouveau enfilé.



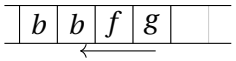
sommets visités : ['c','a','d']



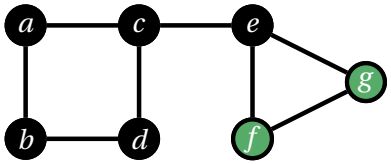
Le sommet e est défilé et marqué comme visité. Les sommets f et g sont enfilés.



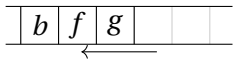
sommets visités : ['c','a','d','e']



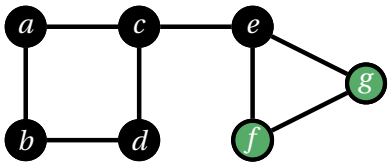
Le sommet b est défilé et marqué comme visité. Il n'a plus de sommets voisins non visités, aucun sommet n'est rajouté dans la file.



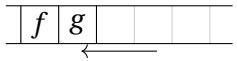
sommets visités : ['c','a','d','e','b']



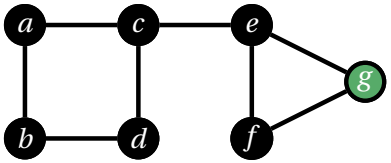
Le sommet b est à nouveau défilé, mais comme il est déjà marqué comme visité, on ne fait rien de plus à cette étape.



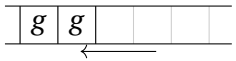
sommets visités : ['c','a','d','e','b']



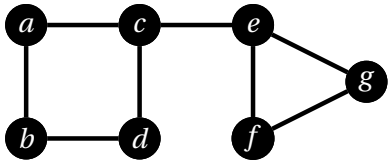
Le sommet f est défilé, on le marque comme visité, et on enfile à nouveau le sommet g.



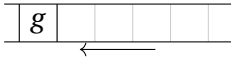
sommets visités : ['c','a','d','e','b','f']



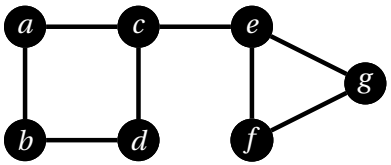
Le sommet g est défilé, on le marque comme visité. Aucun autre sommet n'est enfilé.



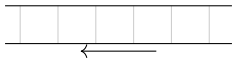
sommets visités : ['c','a','d','e','b','f','g']



Le sommet g est à nouveau défilé, comme il est déjà visité, on ne fait rien d'autre. La file est vide, le parcours est terminé.



sommets visités : ['c','a','d','e','b','f','g']



Exercice 2 **Exemple de parcours BFS** [Sol 2] Sur le même principe, appliquer un parcours BFS au graphe orienté présenté en 6, en partant du sommet 0 et en précisant l'état de la file et la liste des sommets visités à chaque itération.

3.3 Exemples de codes

En utilisant une représentation d'un graphe par un dictionnaire d'adjacence, on peut proposer les codes ci-dessous pour les fonctions `dfs` et `bfs` qui mettent en oeuvre les algorithmes présentés ci-dessus. Une liste `lst_visited` initialement vide est progressivement remplie avec les sommets visités. Elle permet de mémoriser l'ordre de visite des sommets. Un dictionnaire `visited` est défini comme indiqué plus haut. Dans la fonction `dfs`, une pile `s` sert à stocker les sommets au fur et mesure de leur découverte, alors que dans la fonction `bfs`, c'est une file `q` qui joue ce rôle. On peut remarquer la similarité dans la structure des codes.

```
def dfs(grph, v):
    s = deque()                                  création d'une pile vide
    visited = {x : False \
        for x in grph}                           dictionnaire de booléens des sommets visités
    lst_visited = []                             liste des sommets visités
    s.append(v)                                  empilement du sommet de départ
    while len(s) > 0:                            parcours des sommets non visités
        w = s.pop()                              dépilement du sommet de s
        if not visited[w]:                       si w non déjà visité
            visited[w] = True                    w marqué comme visité
            lst_visited.append(w)                ajout de w à la liste des sommets visités
            for u in grph[w]:                    parcours des voisins u de w
                if not visited[u]:               si u non déjà visité
                    s.append(u)                  empilement de u
    return lst_visited
```

```python
def bfs(grph, v):
    q = deque()                                  # création d'une file vide
    visited = {x : False \
        for x in grph}                           # dictionnaire de booléens des sommets visités
    lst_visited = []                             # liste des sommets visités
    q.append(v)                                  # enfilement du sommet de départ
    while len(q) > 0:                            # parcours des sommets non visités
        w = q.popleft()                          # défilement du sommet de q
        if not visited[w]:                       # si w non déjà visité
            visited[w] = True                    # w marqué comme visité
            lst_visited.append(w)                # ajout de w à la liste des sommets visités
            for u in grph[w]:                    # parcours des voisins u de w
                if not visited[u]:               # si u non déjà visité
                    q.append(u)                  # enfilement de u
    return lst_visited
```

VISION UNIFIÉE DES DEUX PARCOURS On peut regrouper les deux parcours d'un graphe $G = (S, A)$ à partir d'un sommet s en les considérant comme deux variantes d'un même algorithme, ce dernier utilisant une structure de donnée générale pour stocker les sommets découverts, notée ici Z.

Parcours de graphe

Données : Le graphe $G = (S, A)$ et un sommet de départ s

Résultat : La liste des sommets visités
$Z \leftarrow s$,
tant que $Z \neq \emptyset$

- retirer un sommet w de Z
- si w n'est pas visité, le marquer comme visité
- pour chaque voisin u de w non visité, ajouter u à Z

renvoyer la liste des sommets visités.

Pour un parcours en profondeur, Z aura une structure de pile, alors que pour un parcours en largeur, Z aura une structure de file.

VARIANTES DES ALGORITHMES PRÉCÉDENTS Les parcours de graphe présentés renvoient ici uniquement la liste `lst_visited` des sommets visités dans l'ordre de leur visite. Dans cette liste, tous les sommets de départ sont accessibles depuis le point de départ. Mais cette liste ne permet pas de retrouver le chemin qui conduit du sommet de départ à l'un des sommets de la liste. Si cette dernière information nous intéresse, on pourra chercher à créer, lors du parcours, un *dictionnaire des prédécesseurs* `pred` associé au parcours donné d'un graphe, tel que `pred[u]` corresponde au sommet `w` par lequel on arrive sur le sommet `u` lors du parcours.

Tel qu'il a été présenté ci-dessus, l'algorithme DSF (*resp.* BFS) peut empiler (*resp.* enfiler) plusieurs fois un même sommet. Cela peut constituer une difficulté si la pile (*resp.* file) devient de grande taille. On peut chercher à construire un algorithme qui évite cet empilement multiple.

On peut aussi choisir de représenter le graphe par un tableau contenant la matrice d'adjacence du graphe. Dans ce cas, il faut légèrement modifier le code de chaque parcours pour tenir compte de ce choix.

De manière plus générale, les deux parcours présentés doivent être considérés comme deux façons de parcourir les sommets d'un graphe donné, et constituent en quelque sorte l'équivalent d'une boucle `for` pour une liste, un tableau ou toute structure séquentielle. On peut ensuite, à partir de ces parcours, effectuer tout type d'opérations sur les graphes.

3.4 Implémentation du graphe et complexité

Il peut être intéressant de déterminer la complexité temporelle de ces deux algorithmes, ou du moins sa complexité asymptotique (suffisante pour déterminer l'efficacité des algorithmes). Pour un graphe $G = (S, A)$, on note ici $n = |S|$ le nombre de sommets et $m = |A|$ le nombre d'arêtes (ou d'arcs). La complexité des parcours va dépendre de la façon dont le graphe étudié est implémenté en mémoire.

GRAPHE REPRÉSENTÉ PAR LISTE D'ADJACENCE. C'est le cas des exemples développés précédemment. D'après les algorithmes décrits, pour chaque sommet w parcouru, on a un petit nombre d'opérations, puis, pour chaque arête (ou arc) issue de ce sommet, on réalise encore un petit nombre d'opérations. À la fin de l'algorithme, on aura parcouru une fois chaque sommet et parcouru une fois chaque arête. On a donc dans ce cas : $\boxed{C = O(n+m)}$.

GRAPHE REPRÉSENTÉ PAR MATRICE D'ADJACENCE Ici la situation est différente, car lors de l'exploration des voisins de w, on réalise n tests, l'information « u est un voisin de w » étant stockée dans une ligne de taille n. On a donc à la fin de l'algorithme réalisé n parcours de sommets et n^2 tests sur les arêtes. On a donc : $\boxed{C = O(n^2)}$.

BILAN Pour tout graphe orienté, on a $m_{\max} = n(n-1)$. Pour un graphe non orienté, $m_{\max} = n(n-1)/2$.

- Le cas des graphes *peu denses* correspond à la situation $m \ll m_{\max}$ de sorte que $O(n+m) = O(n)$. L'implémentation par liste d'adjacence est donc d'autant plus efficace en terme de complexité pour ces parcours que le nombre d'arêtes par sommet est faible.
- Pour les *graphes denses*, on a $m = O(n^2)$ de sorte que $O(n+m) = O(n^2)$; les deux complexités sont équivalentes.

4 APPLICATIONS

Les parcours de graphe font l'objet de nombreuses applications. On en présente rapidement trois ci-dessous qui seront étudiées en TP.

4.1 Existence de chemins

Partant d'un sommet d'un graphe, les parcours en profondeur ou en largeur visitent tous les sommets accessibles depuis ce sommet. Il est alors simple de vérifier l'existence ou non d'un chemin de ce sommet de départ à tout autre sommet du graphe. Dans le graphe de la Figure 11, il existe au moins un chemin de a vers i. Le chemin $a \to b \to e \to g \to j \to i$ en est un ; le chemin $a \to b \to c \to f \to h \to k \to \ell \to j \to i$ en est un autre. En revanche, il n'existe aucun chemin de k vers e.

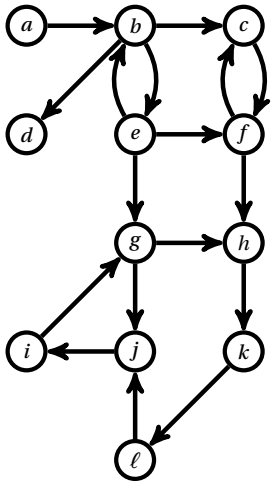


FIGURE 11 : Graphe orienté et chemins.

4.2 Calcul de la distance entre deux sommets

Les *parcours* de graphes visitent tous leurs sommets une seule fois. Seul l'ordre de leur visite diffère selon la nature du parcours, en *largeur* et en *profondeur*, mais également selon l'ordre dans lequel les sommets sont *découverts*. D'un point de vue informatique, ce dernier point est directement lié à la façon dont les voisins d'un sommet sont stockés dans la liste des voisins. On peut s'interroger sur la possibilité d'utiliser ces parcours pour déterminer des longueurs de chemin, voire des distances entre sommets. Considérons à nouveau le graphe G_1 introduit dans le Chapitre (S2) 3.

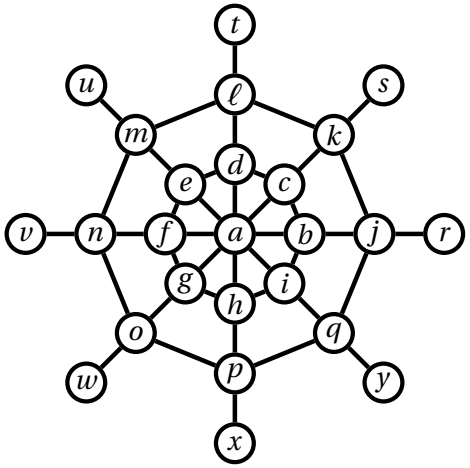
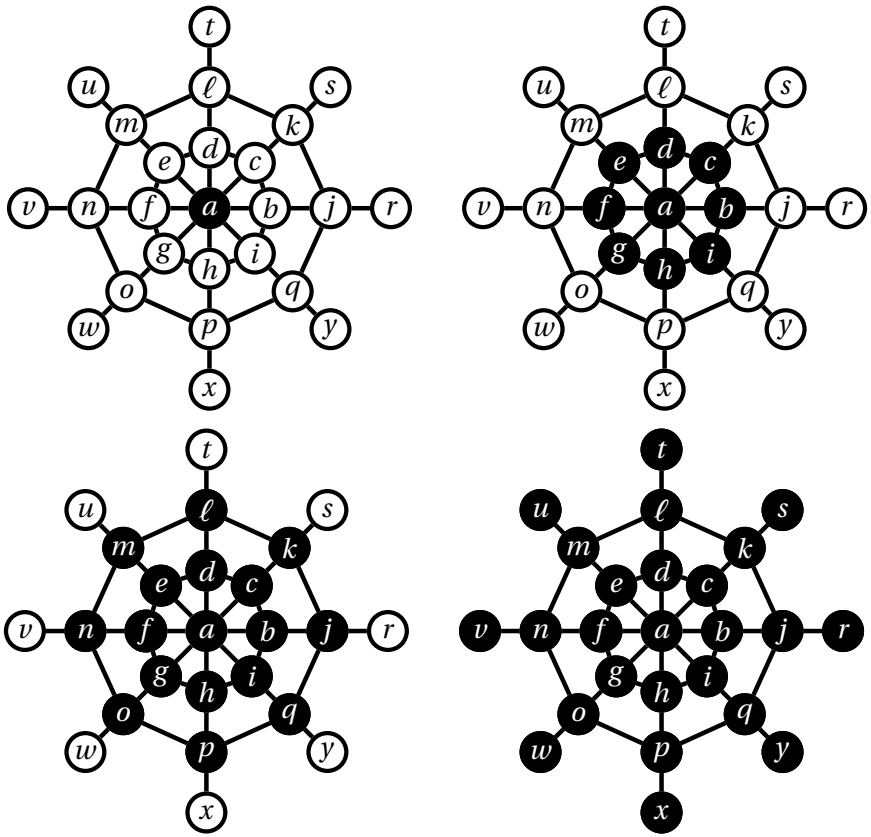


FIGURE 12 : Graphe G_1

Si on considère un *parcours en largeur* du graphe G_1 définie, on sait que les sommets sont visités par distance croissante depuis un sommet de départ donné. Si l'ordre de leur découverte importe pour préciser l'ordre de visite des sommets, il est totalement inutile pour calculer la longueur d'un chemin liant un sommet de départ à chaque sommet visité. La topologie en cercles concentriques du graphe G_1 est particulièrement adaptée pour illustrer les sommets visités lors d'un parcours en largeur issu du sommet a.

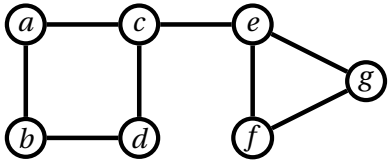


Une simple adaptation du parcours en largeur doit donc permettre de calculer les longueurs des chemins liant un sommet de départ à tout autre sommet du graphe. Et qui plus est, en raison de la nature même du parcours en largeur, ces longueurs sont aussi les distances du sommet de départ aux autres sommets.

Le code suivant met en oeuvre le parcours en largeur et calcule, dans un dictionnaire, les distances d'un sommet de départ v à tous les autres sommets d'un graphe g.

```
def bfs_dist_essai(grph, v):
    q = deque()                                  déclaration d'une file vide
    visited = {x: False for x in grph}           dictionnaire des sommets visités
    dist = {x : 0 for x in grph}                 dictionnaire des distances
    q.append(v)                                  le sommet de départ est enfilé
    while len(q) > 0:                            visite de tous les sommets
        w = q.popleft()                          défilement du sommet w
        if not visited[w]:                       si w non déjà visité
            visited[w] = True                    w est marqué comme visité
            for u in grph[w]:                    parcours des voisins de w
                if not visited[u]:
                    q.append(u)                  et enfilé
                    if dist[u] == 0:             dans le cas d'une première découverte
                        dist[u] = 1 + dist[w]    sa distance à v est mise à jour
    return dist
```

Voici par exemple un essai sur le graphe `grph` défini en début de section.




```
grph = {
    'a' : ['b', 'c'], 'b' : ['a', 'd'],
    'c' : ['a', 'd', 'e'], 'd' : ['b', 'c'],
    'e' : ['c', 'f', 'g'], 'f' : ['e', 'g'],
    'g' : ['e', 'f']}
```

```
>>> bfs_dist_essai(grph, "a")
{'a': 0, 'b': 1, 'c': 1, 'd': 2, 'e': 2, 'f': 3, 'g': 3}
```

AMÉLIORATION POSSIBLE. On peut améliorer la version précédente, et s'économiser les tests sur les distances. En effet, on peut marquer les sommets comme visités dès leur enfilement, puisqu'une fois qu'on les a découvert une première fois, on a leur distance depuis le sommet d'origine ; il ne faut pas y retoucher ensuite.

Note | *On pourrait améliorer ainsi de la même façon le code de `bfs`, mais la présentation faite en cours a le mérite de l'unification avec `dfs`.*

```
def bfs_dist(grph, v):
    q = deque()                                  déclaration d'une file vide
    visited = {x : False for x in grph}          dictionnaire des sommets visités
    dist = {x : 0 for x in grph}                 dictionnaire des distances
    q.append(v)                                  le sommet de départ est enfilé
    visited[v] = True                            et marqué comme visité
    while len(q) > 0:                            visite de tous les sommets
        w = q.popleft()                          défilement du sommet w
        for u in grph[w]:                        parcours des voisins de w
            if not visited[u]:                   si un sommet n'a pas été visité
                visited[u] = True                il est marqué comme visité
```

```
                q.append(u)                          et enfilé
                dist[u] = 1 + dist[w]                sa distance à v est mise à jour
    return dist
```

Voici par exemple un essai sur le graphe `grph` défini en début de section. C'est bien cohérent avec la première version, mais est plus économique en calculs.

```
>>> bfs_dist(grph, "a")
{'a': 0, 'b': 1, 'c': 1, 'd': 2, 'e': 2, 'f': 3, 'g': 3}
```

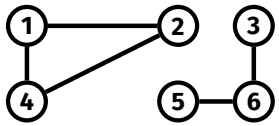
Exercice 3 [Sol 3] Adapter le code précédent au cas d'un graphe codé par matrice d'adjacence.

Remarque 2 La distance entre deux sommets, fournie par le parcours en largeur, ne donne en revanche pas un chemin de longueur minimale. Nous verrons en TP comment en obtenir un.

4.3 Composantes connexes

À la fin d'un parcours d'un graphe non orienté, tous les sommets rencontrés depuis un sommet donné appartiennent à une même composante connexe du graphe. Pour déterminer les éventuelles autres composantes connexes, on peut effectuer un nouveau parcours à partir d'un des sommets non encore visités, et ce jusqu'à ce qu'il n'y ait plus aucun sommet non visités.

Le graphe représenté ci-contre est non connexe et comporte deux composantes connexes qu'on peut définir par les ensembles de sommets $\{1,2,4\}$ et $\{3,5,6\}$.



Remarque 3 Une notion de connexité peut également être définie pour les *graphes orientés* mais elle s'écarte un peu de la notion de connexité simple. En effet, un graphe orienté peut être d'un seul tenant mais il n'existe pas toujours de chemin entre deux sommets du graphe en raison des orientations. Les sommets qui sont liés par des chemins forment des *composantes fortement connexes*. Ce sujet étant hors-programme, il ne sera pas développé.

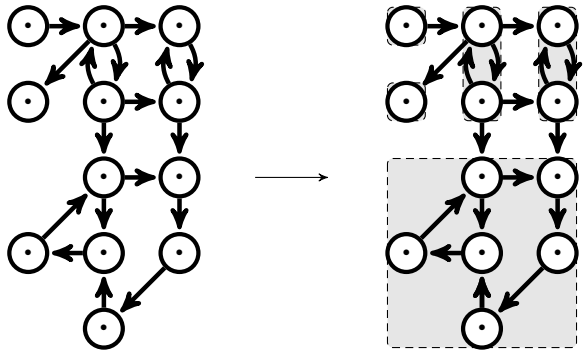
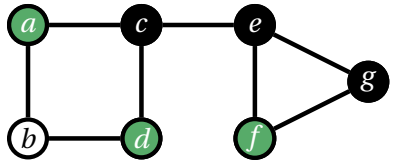


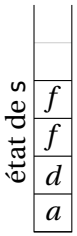
FIGURE 13 : Composantes fortement connexes d'un graphe orienté.

4.4 Détection de cycle

GRAPHES NON ORIENTÉS. Pour un graphe non orienté, si lors d'un parcours on découvre deux fois le même sommet `v`, alors il existe un cycle contenant `v`. Par exemple, lors d'un des parcours en profondeur étudiés précédemment, on a :



sommets visités : `['c','e','g']`



Le sommet f est présent deux fois dans la pile, car il a été découvert une première fois à partir de e, puis une deuxième fois à partir de g. Ceci est bien associé à l'existence du cycle e, f, g.

GRAPHES ORIENTÉS L'algorithme de parcours en profondeur permet également la détection de *cycles* dans un graphe orienté, c'est-à-dire l'existence de chemins qui se referment. Le graphe représenté en figure 14 présente de nombreux cycles parmi lesquels $b \to e \to b$, $g \to j \to i \to g$.

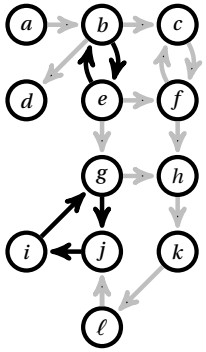


FIGURE 14 : Deux cycles dans un graphe orienté.

L'algorithme de parcours en profondeur marque comme visités les sommets découverts. Rencontrer deux fois le même sommet ne suffit cependant pas à affirmer l'existence d'un cycle. Sur la figure 14, au moins deux chemins relient e à h : le chemin $e \to f \to h$ et le chemin $e \to g \to h$. Un algorithme de parcours qui visite les sommets depuis le sommet e trouvera donc le premier de ces chemins et marquera le sommet h comme visité. Par le second de ces chemins, le sommet h est de nouveau rencontré et marqué comme visité. Mais la suite des sommets e, f, h, g ne forme pas un cycle. Il conviendra donc de trouver un moyen de distinguer les sommets visités selon qu'aucun autre parcours ne les visitera de nouveau ou selon qu'ils sont susceptibles d'être encore visités.

Solution 3

```
def bfs_dist(A, v):
    n = len(A)                                  nombre de sommets
    q = deque()                                 déclaration d'une file vide
    visited = {i : False for i in range(n)}     dictionnaire des sommets
visités
    dist = {i : 0 for i in range(n)}            dictionnaire des distances
    q.append(v)                                 le sommet de départ est
enfilé
    visited[v] = True                           et marqué comme visité
    while len(q) > 0:                           visite de tous les sommets
        i = q.popleft()                         défilement du sommet i
        for j in range(n):                      parcours des voisins de i
          if not visited[j] and A[i,j]:         si un sommet n'a pas été
visité
            visited[j] = True                   il est marqué comme
visité
            q.append(j)                                 et enfilé
            dist[j] = 1 + dist[i]               sa distance à v est mise à
jour
    return dist
```