Semestre 2 / Cours 4 - Parcours de graphes non pondérés & Applications

ITC MPSI & PCSI – Année 2023-2024

Sommaire

Notions de parcours

Objectifs

- Connaître les deux types de parcours d'un graphe : parcours en profondeur, parcours en largeur,

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`,

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 œuvre les algorithmes de parcours de graphes en utilisant une pile et une file,

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 œuvre les algorithmes de parcours de graphes en utilisant une pile et une file,
- savoir adapter les algorithmes de parcours pour déterminer un chemin, déterminer la connexité et détecter un cycle dans un graphe.

Notion de parcours

- Tableaux et listes : données organisées de manière séquentielle (indexation),

Notion de parcours

- Tableaux et listes : données organisées de manière séquentielle (indexation),
- parcours à l'aide des instructions `for` et `while`.

Notion de parcours

- Tableaux et listes : données organisées de manière séquentielle (indexation),
- parcours à l'aide des instructions **`for`** et **`while`**.
- Graphes : données organisées de manière relationnelle (arêtes ou arcs entre noeuds),

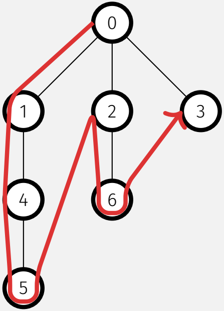
Notion de parcours

- Tableaux et listes : données organisées de manière séquentielle (indexation),
- parcours à l'aide des instructions `for` et `while`.
- Graphes : données organisées de manière relationnelle (arêtes ou arcs entre noeuds),
- nécessité de concevoir des manières de parcourir ces données.

- Tableaux et listes : données organisées de manière séquentielle (indexation),
- parcours à l'aide des instructions `for` et `while`.
- Graphes : données organisées de manière relationnelle (arêtes ou arcs entre noeuds),
- nécessité de concevoir des manières de parcourir ces données.
- Deux parcours fondamentaux :

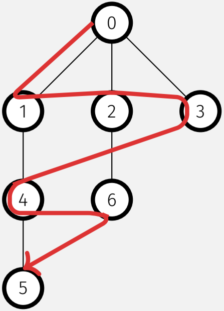
- Tableaux et listes : données organisées de manière séquentielle (indexation),
- parcours à l'aide des instructions **`for`** et **`while`**.
- Graphes : données organisées de manière relationnelle (arêtes ou arcs entre noeuds),
- nécessité de concevoir des manières de parcourir ces données.
- Deux parcours fondamentaux :
  - le <u>parcours en profondeur</u> ou DFS (Deep First Search) : on explore le graphe à partir d'un sommet s en allant « le plus loin possible » pour chaque chemin,

- Tableaux et listes : données organisées de manière séquentielle (indexation),
- parcours à l'aide des instructions **`for`** et **`while`**.
- Graphes : données organisées de manière relationnelle (arêtes ou arcs entre noeuds),
- nécessité de concevoir des manières de parcourir ces données.
- Deux parcours fondamentaux :
  - le parcours en profondeur ou DFS (Deep First Search) : on explore le graphe à partir d'un sommet s en allant « le plus loin possible » pour chaque chemin,
  - le parcours en largeur ou BFS (Breadth First Search) : on explore le graphe à partir d'un sommet s en explorant « niveau par niveau ».



Ordre souhaité : $(0, 1, 4, 5, 2, 6, 3)$

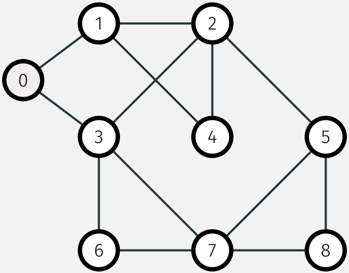
(a) Parcours en profondeur



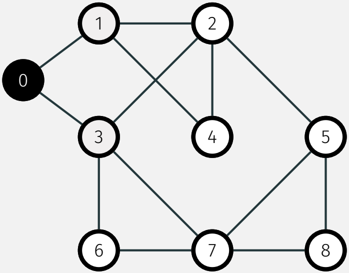
Ordre souhaité : $(0, 1, 2, 3, 4, 6, 5)$

(b) Parcours en largeur

Figure 1 – Deux types de parcours

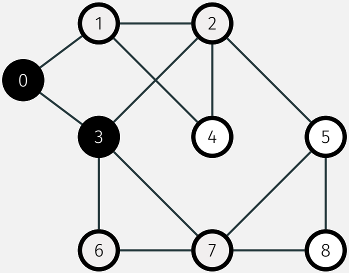


Sommets visités : $L = []$



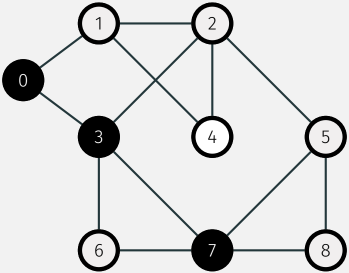
Sommets visités : $L = [0]$

Premier exemple DFS : graphe non orienté



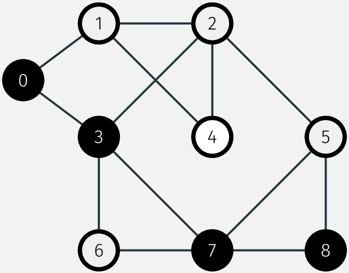
Sommets visités : $L = [0, 3]$

Premier exemple DFS : graphe non orienté



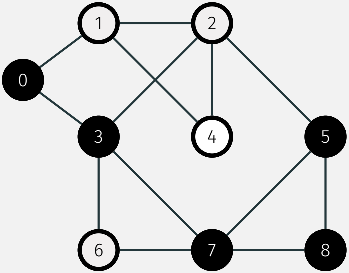
Sommets visités : $L = [0, 3, 7]$

Premier exemple DFS : graphe non orienté



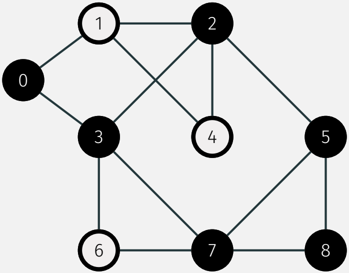
Sommets visités : $L = [0, 3, 7, 8]$

Premier exemple DFS : graphe non orienté



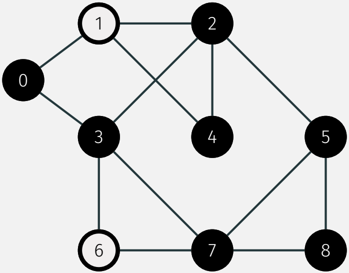
Sommets visités : $L = [0, 3, 7, 8, 5]$

Premier exemple DFS : graphe non orienté



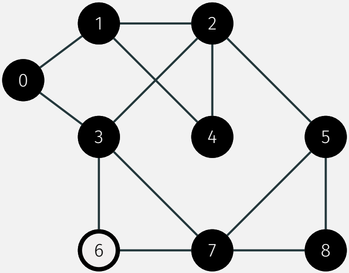
Sommets visités : $L = [0, 3, 7, 8, 5, 2]$

Premier exemple DFS : graphe non orienté



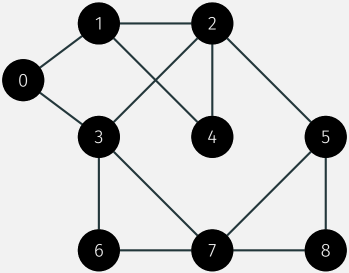
Sommets visités : $L = [0, 3, 7, 8, 5, 2, 4]$

Premier exemple DFS : graphe non orienté



Sommets visités : $L = [0, 3, 7, 8, 5, 2, 4, 1]$

Premier exemple DFS : graphe non orienté



Sommets visités : $L = [0, 3, 7, 8, 5, 2, 4, 1, 6]$

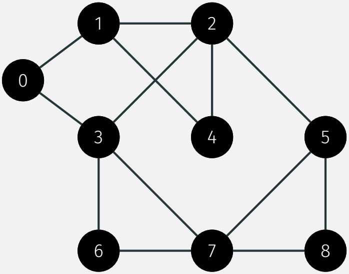


Figure 2 – Parcours DFS d'un graphe non orienté à partir du sommet 0.

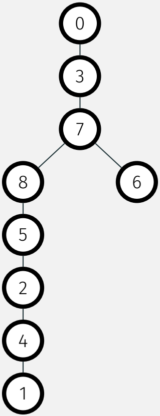
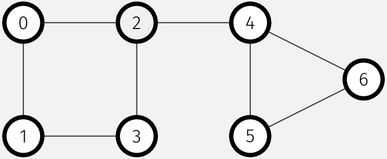


Figure 3 – Arbre des sommets parcourus

Sommets visités :

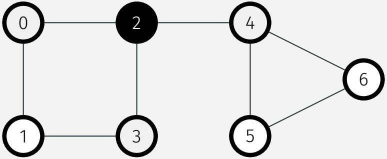
$$L = [0, 3, 7, 8, 5, 2, 4, 1, 6]$$

Deuxième exemple DFS : graphe non orienté



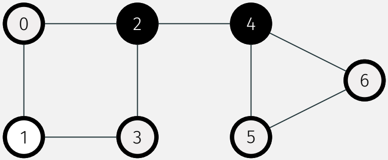
Sommets visités : $L = []$

Deuxième exemple DFS : graphe non orienté



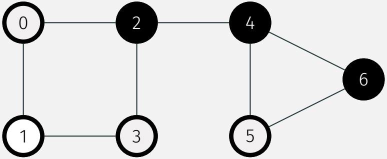
Sommets visités : $L = [2]$

Deuxième exemple DFS : graphe non orienté



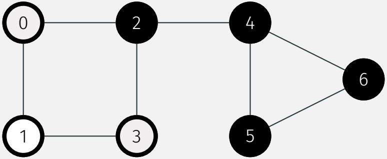
Sommets visités : $L = [2, 4]$

Deuxième exemple DFS : graphe non orienté



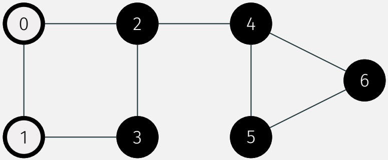
Sommets visités : $L = [2, 4, 6]$

Deuxième exemple DFS : graphe non orienté



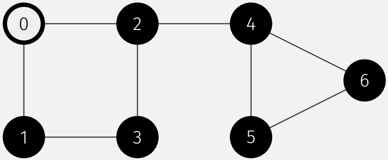
Sommets visités : $L = [2, 4, 6, 5]$

Deuxième exemple DFS : graphe non orienté

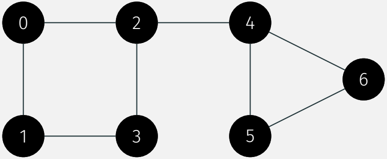


Sommets visités : $L = [2, 4, 6, 5, 3]$

Deuxième exemple DFS : graphe non orienté



Sommets visités : $L = [2, 4, 6, 5, 3, 1]$



Sommets visités : $L = [2, 4, 6, 5, 3, 1, 0]$

Deuxième exemple DFS : graphe non orienté

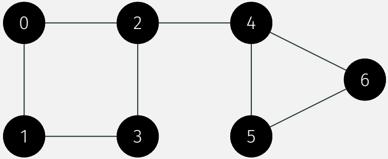


Figure 4 – Parcours DFS d'un graphe non orienté à partir du sommet 0.

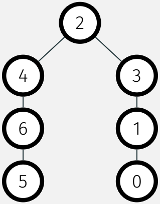
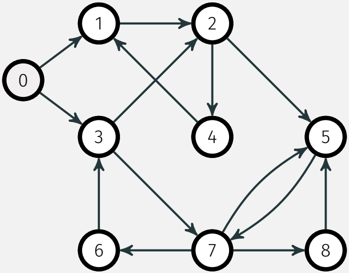
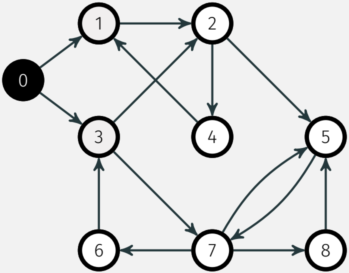


Figure 5 – Arbre des sommets parcourus

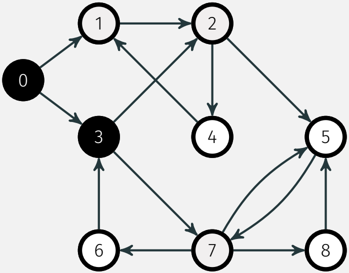
Sommets visités :
$$L = [2, 4, 6, 5, 3, 1, 0]$$



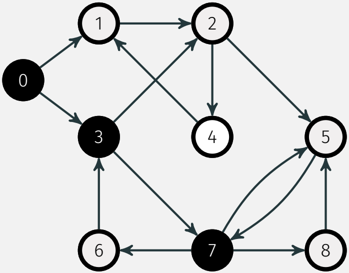
Sommets visités : $L = []$



Sommets visités : $L = [0]$

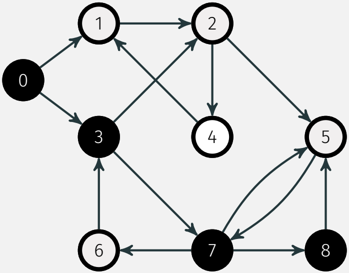


Sommets visités : $L = [0, 3]$

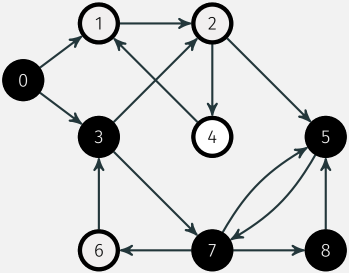


Sommets visités : $L = [0, 3, 7]$

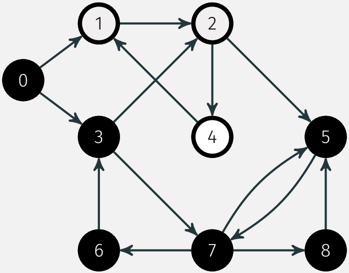
Troisième exemple DFS : graphe orienté



Sommets visités : $L = [0, 3, 7, 8]$

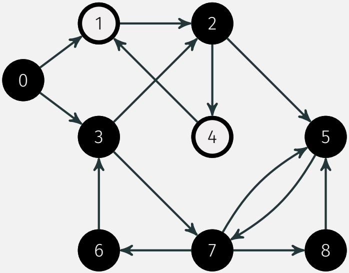


Sommets visités : $L = [0, 3, 7, 8, 5]$



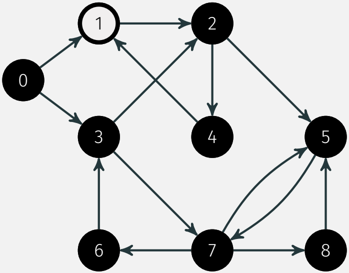
Sommets visités : $L = [0, 3, 7, 8, 5, 6]$

Troisième exemple DFS : graphe orienté



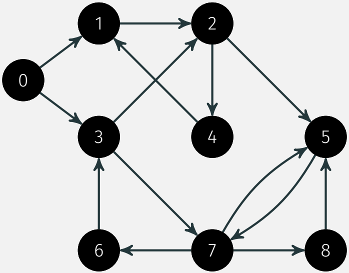
Sommets visités : $L = [0, 3, 7, 8, 5, 6, 2]$

Troisième exemple DFS : graphe orienté



Sommets visités : $L = [0, 3, 7, 8, 5, 6, 2, 4]$

Troisième exemple DFS : graphe orienté



Sommets visités : $L = [0, 3, 7, 8, 5, 6, 2, 4, 1]$

Troisième exemple DFS : graphe orienté

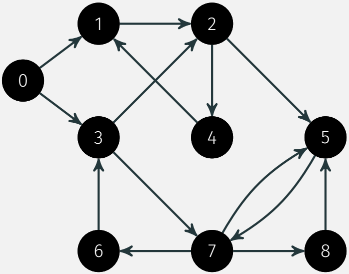


Figure 6 – Parcours DFS d'un graphe orienté à partir du sommet 0.

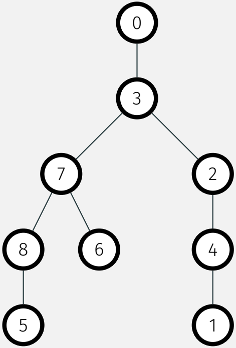
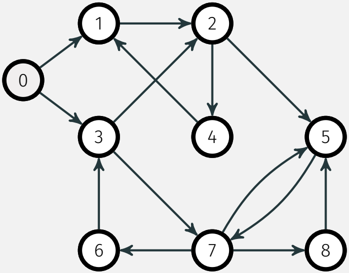


Figure 7 – Arbre des sommets parcourus par DFS

Sommets visités :

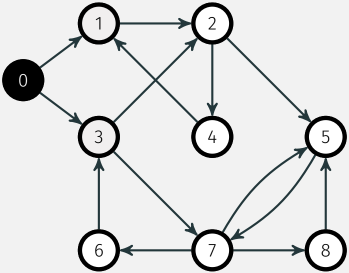
$$L = [0, 3, 7, 8, 5, 6, 2, 4, 1]$$

Exemple BFS : graphe orienté



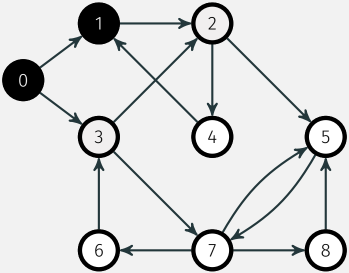
Sommets visités : $L = []$

Exemple BFS : graphe orienté



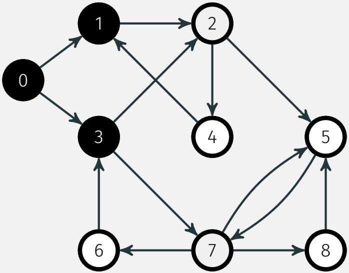
Sommets visités : $L = [0]$

Exemple BFS : graphe orienté

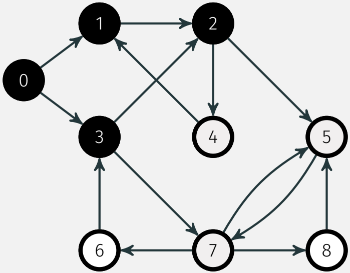


Sommets visités : $L = [0, 1]$

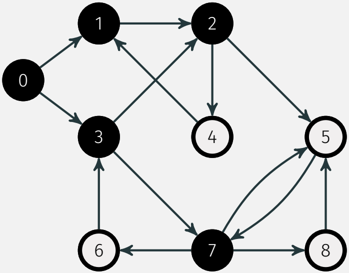
Exemple BFS : graphe orienté



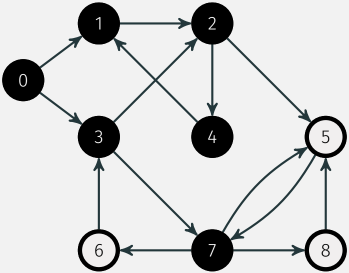
Sommets visités : $L = [0, 1, 3]$



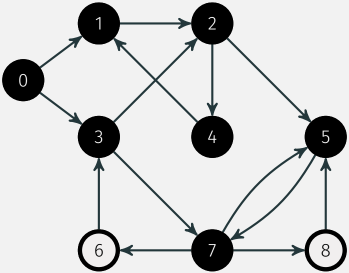
Sommets visités : $L = [0, 1, 3, 2]$



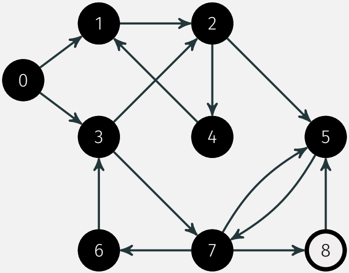
Sommets visités : $L = [0, 1, 3, 2, 7]$



Sommets visités : $L = [0, 1, 3, 2, 7, 4]$

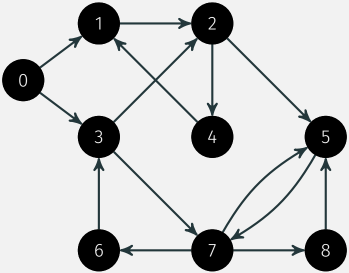


Sommets visités : $L = [0, 1, 3, 2, 7, 4, 5]$



Sommets visités : $L = [0, 1, 3, 2, 7, 4, 5, 6]$

Exemple BFS : graphe orienté



Sommets visités : $L = [0, 1, 3, 2, 7, 4, 5, 6, 8]$

Exemple BFS : graphe orienté

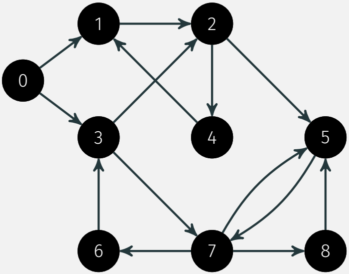


Figure 8 – Parcours BFS d'un graphe orienté à partir du sommet 0.

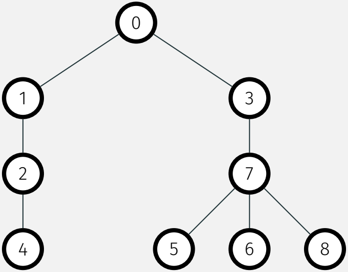


Figure 9 – Arbre des sommets parcourus par BFS

Sommets visités :
$L = [0, 1, 3, 2, 7, 4, 5, 6, 8]$

Liste d'attente des sommets à visiter

- à chaque sommet, découverte des sommets voisins,

Liste d'attente des sommets à visiter

- à chaque sommet, découverte des sommets voisins,
- choix du sommet à visiter parmi les sommets découverts,

Liste d'attente des sommets à visiter

- à chaque sommet, découverte des sommets voisins,
- choix du sommet à visiter parmi les sommets découverts,
- gestion d'une <u>liste d'attente des sommets à visiter</u> adaptée au parcours.

Liste d'attente des sommets à visiter

- à chaque sommet, découverte des sommets voisins,
- choix du sommet à visiter parmi les sommets découverts,
- gestion d'une liste d'attente des sommets à visiter adaptée au parcours.

Nécessité de nouvelles structures de données : les piles et les files.

Piles et files

Notion de pile (stack)

- Structure de données élémentaire ne permettant qu'un petit nombre d'opérations, optimisée pour ces opérations.

Notion de pile (stack)

- Structure de données élémentaire ne permettant qu’un petit nombre d’opérations, optimisée pour ces opérations.
- Représentation verticale possible par un empilement de données.

NOTION DE PILE (STACK)

- Structure de données élémentaire ne permettant qu'un petit nombre d'opérations, optimisée pour ces opérations.
- Représentation verticale possible par un empilement de données.

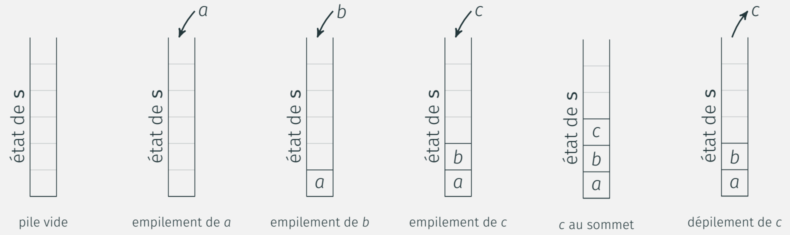


Figure 10 – Représentation d'opérations d'empilement et de dépilement.

Principe LIFO : « Last In First Out »

Notion de file (queue)

- Structure de données élémentaire ne permettant qu'un petit nombre d'opérations, optimisée pour ces opérations.

Notion de file (queue)

- Structure de données élémentaire ne permettant qu'un petit nombre d'opérations, optimisée pour ces opérations.
- Représentation horizontale possible par un glissement de données.

- Structure de données élémentaire ne permettant qu'un petit nombre d'opérations, optimisée pour ces opérations.
- Représentation horizontale possible par un glissement de données.

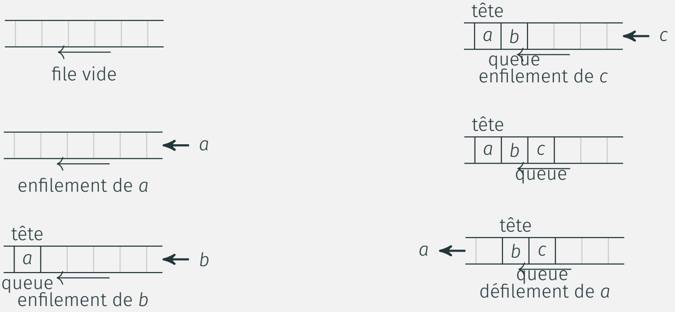


Figure 11 – Représentation d'opérations d'enfilement et de défilement.

Principe FIFO : « First In First Out »

Opérations souhaitées

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

Pile implémentée par une liste dont le dernier élément est le sommet de la pile (*stack* en Anglais, nous les appellerons donc `s` en Python) :

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

- File implémentée par une liste dont le premier élément est la tête et le dernier élément est la queue (*queue* en Anglais, nous les appellerons donc `s` en Python) de la file :

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

- Complexité en $O(n)$ pour le défilement $\Longrightarrow$ choix non retenu.

- nouvelle structure de données `deque` à importer :

```python
from collections import deque
```

Double Ended Queue : présentation

- nouvelle structure de données `deque` à importer :

```python
from collections import deque
```

- ajout et retrait de données aux deux extrémités en $O(1)$,

Double Ended Queue : présentation

- nouvelle structure de données `deque` à importer :

```
from collections import deque
```

- ajout et retrait de données aux deux extrémités en $O(1)$,
- implémentation d'une pile $\implies$ choix d'une extrémité comme sommet,

Double Ended Queue : présentation

- nouvelle structure de données `deque` à importer :

```
from collections import deque
```

- ajout et retrait de données aux deux extrémités en $O(1)$,
- implémentation d'une pile $\Longrightarrow$ choix d'une extrémité comme sommet,
- implémentation d'une file $\Longrightarrow$ choix d'une extrémité comme tête et l'autre comme queue,

Double Ended Queue : présentation

- nouvelle structure de données **deque** à importer :

```
from collections import deque
```

- ajout et retrait de données aux deux extrémités en $O(1)$,
- implémentation d'une pile $\implies$ choix d'une extrémité comme sommet,
- implémentation d'une file $\implies$ choix d'une extrémité comme tête et l'autre comme queue,
- fonctions de manipulation utiles :

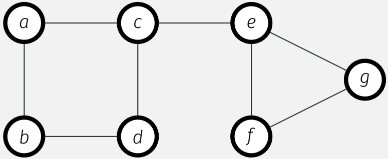
Deque	Instruction	Complexité
Création	`d = deque()`	$O(1)$
Ajout à gauche	`d.appendleft('a')`	$O(1)$
Ajout à droite	`d.append('b')`	$O(1)$
Retrait à gauche	`d.popleft()`	$O(1)$
Retrait à droite	`d.pop()`	$O(1)$
Test vide	`len(d) == 0`	$O(1)$

Double Ended Queue : utilisation

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`

Algorithmes de parcours

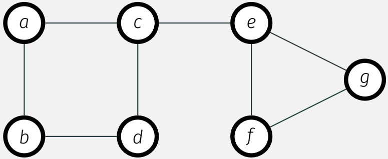



```
grph = {
    'a' : ['b', 'c'], \
    ↪ 'b' : ['a', 'd'],
    'c' : ['a', 'd', \
    ↪ 'e'], 'd' : \
    ↪ ['b', 'c'],
    'e' : ['c', 'f', \
    ↪ 'g'], 'f' : \
    ↪ ['e', 'g'],
    'g' : ['e', 'f']}
```

Figure 12 – Graphe non orienté et son dictionnaire d'adjacence.

- Début du parcours : sommet c,

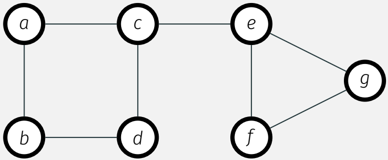
Implémentation : parcours DFS et pile




```
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 12 – Graphe non orienté et son dictionnaire d'adjacence.

- Début du parcours : sommet *c*,
- utilisation d'une pile **s** contenant les sommets découverts (à partir du sommet courant) et non encore visités,




```
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 12 – Graphe non orienté et son dictionnaire d'adjacence.

- Début du parcours : sommet *c*,
- utilisation d'une pile `s` contenant les sommets découverts (à partir du sommet courant) et non encore visités,
- utilisation d'un dictionnaire indiquant les sommets visités

Lors du parcours,

- chaque sommet dépilé et non encore visité est marqué comme visité,

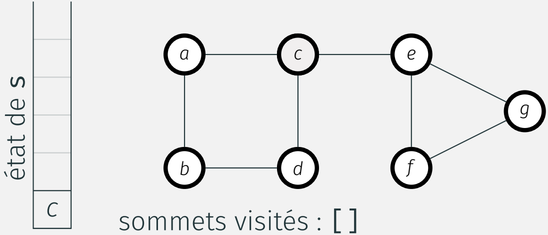
Lors du parcours,

- chaque sommet dépilé et non encore visité est marqué comme visité,
- chaque sommet découvert et non encore visité est empilé.

Lors du parcours,

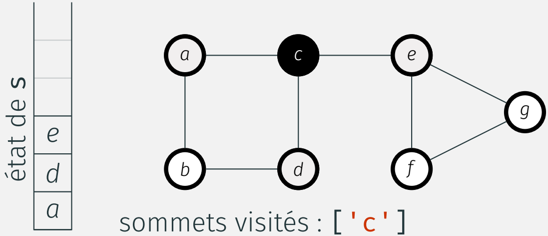
- chaque sommet dépilé et non encore visité est marqué comme visité,
- chaque sommet découvert et non encore visité est empilé.

Initialisation : on empile le sommet de départ, ici $'c'$.



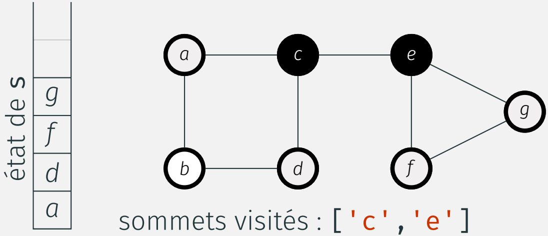
sommets visités : [ ]

Implémentation : parcours DFS et pile



sommets visités : ['c']

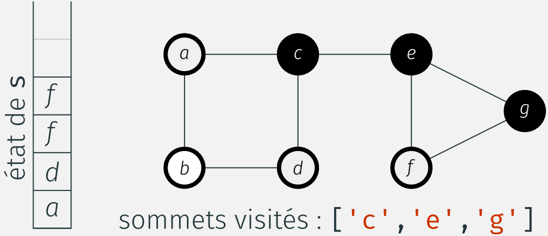
sommets découverts : 'a','d','e'



sommets visités : `['c','e']`

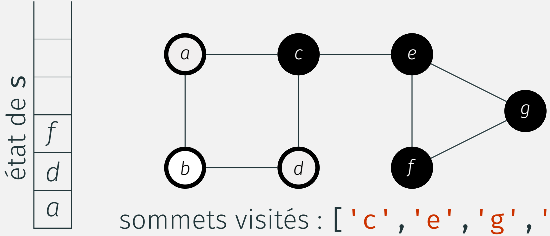
sommets découverts : `'f','g'`

Implémentation : parcours DFS et pile



sommets visités : `['c','e','g']`

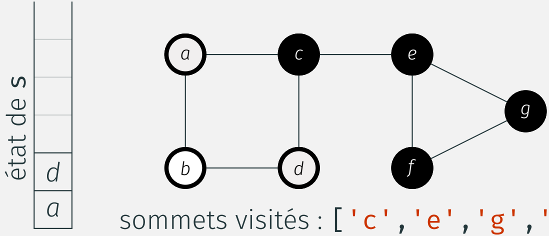
sommets découverts : `'f'`



sommets visités : `['c','e','g','f']`

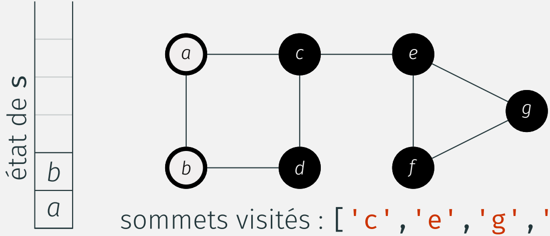
sommets découverts : $\emptyset$

Implémentation : parcours DFS et pile



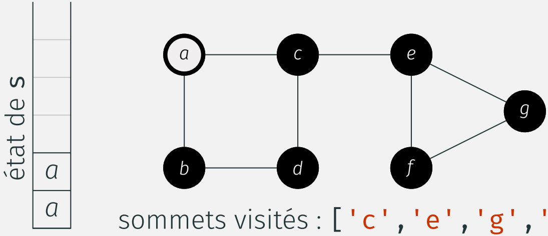
sommets visités : `['c','e','g','f']`

sommets découverts : $\emptyset$



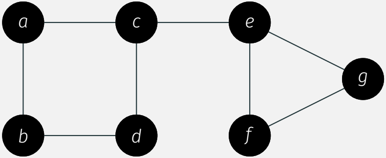
sommets visités : `['c','e','g','f','d']`

sommets découverts : `'b'`



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

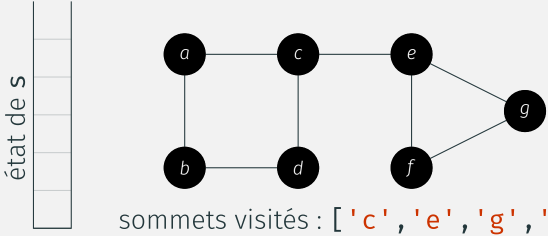
sommets découverts : 'a'



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

sommets découverts : $\emptyset$

Implémentation : parcours DFS et pile

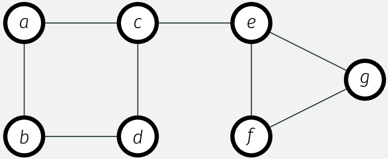


sommets visités : `['c','e','g','f','d','b','a']`

sommets découverts : $\emptyset$

La pile est vide : fin du parcours.

Exercice : parcours BFS et file

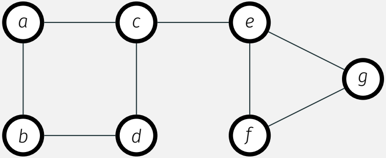



```
grph = {
    'a' : ['b', 'c'], \
    ↪ 'b' : ['a', 'd'],
    'c' : ['a', 'd', \
    ↪ 'e'], 'd' : \
    ↪ ['b', 'c'],
    'e' : ['c', 'f', \
    ↪ 'g'], 'f' : \
    ↪ ['e', 'g'],
    'g' : ['e', 'f']}
```

Figure 13 – Graphe non orienté et son dictionnaire d'adjacence.

- Début du parcours : sommet c,

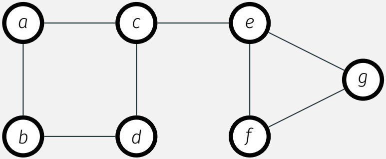
Exercice : parcours BFS et file




```
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 13 – Graphe non orienté et son dictionnaire d'adjacence.

- Début du parcours : sommet c,
- utilisation d'une file **q** contenant les sommets découverts (à partir du sommet courant) et non encore visités,




```
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 13 – Graphe non orienté et son dictionnaire d'adjacence.

- Début du parcours : sommet c,
- utilisation d'une file **q** contenant les sommets découverts (à partir du sommet courant) et non encore visités,
- utilisation d'un dictionnaire mémorisant les sommets visités

Lors du parcours,

- chaque sommet défilé et non encore visité est marqué comme visité,

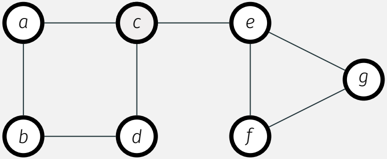
Lors du parcours,

- chaque sommet défilé et non encore visité est marqué comme visité,
- chaque sommet découvert et non encore visité est enfilé.

Lors du parcours,

- chaque sommet défilé et non encore visité est marqué comme visité,
- chaque sommet découvert et non encore visité est enfilé.

Initialisation : on enfile le sommet de départ, ici c.

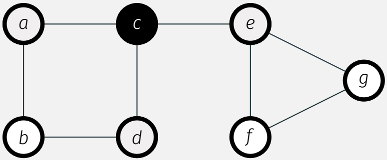


sommets visités : [ ]

On défile le sommet c, on l'indique comme visité puisqu'il ne l'était pas déjà. On analyse les voisins de c non visités, et on les enfile.

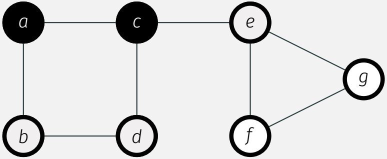
Exercice : parcours BFS et file

On défile le sommet c, on l'indique comme visité puisqu'il ne l'était pas déjà. On analyse les voisins de c non visités, et on les enfile.



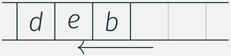
sommets visités : ['c']

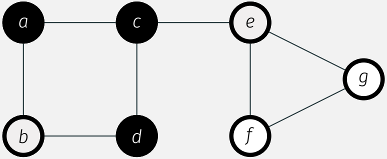
sommets découverts : 'a','d','e'



sommets visités : `['c','a']`

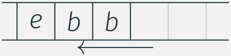
sommets découverts : `'b'`

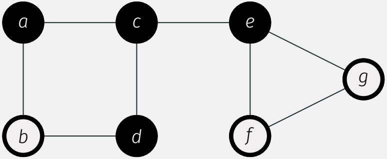




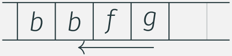
sommets visités : `['c','a','d']`

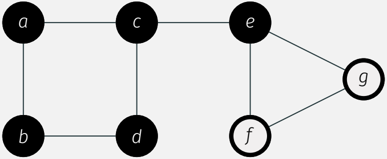
sommets découverts : `'b'`





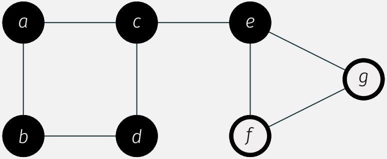
sommets visités : `['c','a','d','e']`
sommets découverts : `'f','g'`





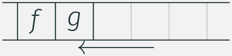
sommets visités : ['c','a','d','e','b']

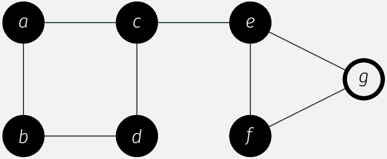
sommets découverts : $\emptyset$



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

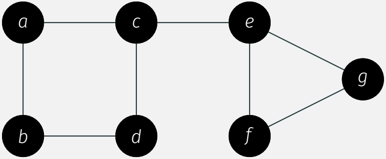
sommets découverts : $\emptyset$





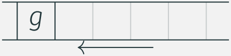
sommets visités : `['c','a','d','e','b','f']`

sommets découverts : `'g'`

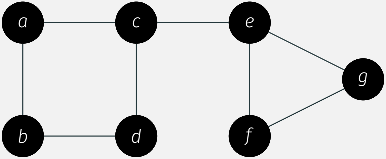


sommets visités : `['c','a','d','e','b','f','g']`

sommets découverts : $\emptyset$



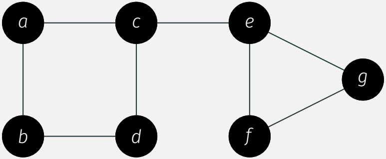
Exercice : parcours BFS et file



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

sommets découverts : $\emptyset$

Exercice : parcours BFS et file



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

sommets découverts : $\emptyset$

La file est vide : fin du parcours.

On crée les deux fonctions :

- `dfs(grph:dict,v)->list`
- `bfs(grph:dict,s)->list`

avec les caractéristiques suivantes :

Exemples de codes

On crée les deux fonctions :

- `dfs(grph:dict,v)->list`
- `bfs(grph:dict,s)->list`

avec les caractéristiques suivantes :

- variables d'entrée :

On crée les deux fonctions :

- `dfs(grph:dict,v)->list`
- `bfs(grph:dict,s)->list`

avec les caractéristiques suivantes :

- variables d'entrée :
  - ◇ `grph`, dictionnaire d'adjacence du graphe étudié,

On crée les deux fonctions :

- `dfs(grph:dict,v)->list`
- `bfs(grph:dict,s)->list`

avec les caractéristiques suivantes :

- variables d'entrée :
  - `grph`, dictionnaire d'adjacence du graphe étudié,
  - `v`, étiquette du sommet de départ (pouvant être par exemple de type `str` ou `int`),

On crée les deux fonctions :

- `dfs(grph:dict,v)->list`
- `bfs(grph:dict,s)->list`

avec les caractéristiques suivantes :

- variables d'entrée :
  - ◇ `grph`, dictionnaire d'adjacence du graphe étudié,
  - ◇ `v`, étiquette du sommet de départ (pouvant être par exemple de type `str` ou `int`),
- renvoie `lst_visited`, liste des sommets visités dans l'ordre de leur visite,

On crée les deux fonctions :

- `dfs(grph:dict,v)->list`
- `bfs(grph:dict,s)->list`

avec les caractéristiques suivantes :

- variables d'entrée :
  - `grph`, dictionnaire d'adjacence du graphe étudié,
  - `v`, étiquette du sommet de départ (pouvant être par exemple de type `str` ou `int`),
- renvoie `lst_visited`, liste des sommets visités dans l'ordre de leur visite,
- utilise les variables locales suivantes :

Exemples de codes

On crée les deux fonctions :

- `dfs(grph:dict,v)->list`
- `bfs(grph:dict,s)->list`

avec les caractéristiques suivantes :

- variables d'entrée :
  - `grph`, dictionnaire d'adjacence du graphe étudié,
  - `v`, étiquette du sommet de départ (pouvant être par exemple de type `str` ou `int`),
- renvoie `lst_visited`, liste des sommets visités dans l'ordre de leur visite,
- utilise les variables locales suivantes :
  - pour DFS (resp. BFS), une pile `s` (resp. file `q`) contenant les sommets découverts à partir du dernier sommet visité,

On crée les deux fonctions :

- `dfs(grph:dict,v)->list`
- `bfs(grph:dict,s)->list`

avec les caractéristiques suivantes :

- variables d'entrée :
  - `grph`, dictionnaire d'adjacence du graphe étudié,
  - `v`, étiquette du sommet de départ (pouvant être par exemple de type `str` ou `int`),
- renvoie `lst_visited`, liste des sommets visités dans l'ordre de leur visite,
- utilise les variables locales suivantes :
  - pour DFS (resp. BFS), une pile `s` (resp. file `q`) contenant les sommets découverts à partir du dernier sommet visité,
  - un dictionnaire `visited` : pour un sommet `w`, `visited[w]` vaut `True` si `w` a été visité, `False` sinon.

Code dfs

```
def dfs(grph, v):
    s = deque()
    visited = {x : False \
        for x in grph.keys()}
    lst_visited = []
    s.append(v)
    while len(s) > 0:
        w = s.pop()
        if not visited[w]:
            visited[w] = True
            lst_visited.append(w)
            for u in grph[w]:
                if not visited[u]:
                    s.append(u)
    return lst_visited
```

création d'une pile vide
dictionnaire de booléens des sommets visités
liste des sommets visités
empilement du sommet de départ
parcours des sommets non visités
dépilement du sommet de s
si w non déjà visité
w marqué comme visité
ajout de w à la liste des sommets visités
parcours des voisins u de w
si u non déjà visité
empilement de u

Code bfs

```python
def bfs(grph, v):
    q = deque()
    visited = {x : False \
        for x in grph.keys()}
    lst_visited = []
    q.append(v)
    while len(q) > 0:
        w = q.popleft()
        if not visited[w]:
            visited[w] = True
            lst_visited.append(w)
            for u in grph[w]:
                if not visited[u]:
                    q.append(u)
    return lst_visited
```

création d'une file vide

dictionnaire de booléens des sommets visités

liste des sommets visités

enfilement du sommet de départ

parcours des sommets non visités

défilement du sommet de q

si w non déjà visité

w marqué comme visité

ajout de w à la liste des sommets visités

parcours des voisins u de w

si u non déjà visité

enfilement de u

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.

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.

La structure de donnée Z servant à stocker les sommets découverts est :

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.

La structure de donnée Z servant à stocker les sommets découverts est :

- une pile pour le parcours en profondeur dfs,
- une file pour le parcours en largeur bfs.

Les algorithmes précédent peuvent être modifiés pour répondre à différentes situations et volontés :

- Création d'un <u>dictionnaire des prédécesseurs</u> `pred` tel que `pred[u]` soit le sommet `w` par lequel on arrive sur le sommet `u` lors du parcours.

Les algorithmes précédent peuvent être modifiés pour répondre à différentes situations et volontés :

- Création d'un dictionnaire des prédécesseurs `pred` tel que `pred[u]` soit le sommet `w` par lequel on arrive sur le sommet `u` lors du parcours.Permet de reconstituer le chemin entre deux sommets lors du parcours.

Les algorithmes précédent peuvent être modifiés pour répondre à différentes situations et volontés :

- Création d'un dictionnaire des prédécesseurs `pred` tel que `pred[u]` soit le sommet `w` par lequel on arrive sur le sommet `u` lors du parcours.Permet de reconstituer le chemin entre deux sommets lors du parcours.
- Empilement ou enfilement multiple du même sommet peut poser des soucis.

Les algorithmes précédent peuvent être modifiés pour répondre à différentes situations et volontés :

- Création d'un dictionnaire des prédécesseurs `pred` tel que `pred[u]` soit le sommet `w` par lequel on arrive sur le sommet `u` lors du parcours.Permet de reconstituer le chemin entre deux sommets lors du parcours.
- Empilement ou enfilement multiple du même sommet peut poser des soucis.
- Adaptation nécessaire au cas où le graphe est implémenté par matrice d'adjacence.

- 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),

Complexité

- 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),
- si G est implémenté par une liste d'adjacence :
  - ◇ pour chaque sommet w parcouru, on a un petit nombre d'opérations,

Complexité

- 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),
- si G est implémenté par une liste d'adjacence :
  - pour chaque sommet w parcouru, on a un petit nombre d'opérations,
  - pour chaque arête (ou arc) issue de ce sommet, on réalise encore un petit nombre d'opérations,

Complexité

- 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),
- si G est implémenté par une liste d'adjacence :
  - pour chaque sommet w parcouru, on a un petit nombre d'opérations,
  - pour chaque arête (ou arc) issue de ce sommet, on réalise encore un petit nombre d'opérations,

  On a donc $C = \mathrm{O}(n + m)$.

- 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),
- si G est implémenté par une liste d'adjacence :
  - ◇ pour chaque sommet w parcouru, on a un petit nombre d'opérations,
  - ◇ pour chaque arête (ou arc) issue de ce sommet, on réalise encore un petit nombre d'opérations,

  On a donc $C = \mathrm{O}(n + m)$.
- Si G est implémenté par une matrice d'adjacence :

- 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),
- si G est implémenté par une liste d'adjacence :
  - ◇ pour chaque sommet w parcouru, on a un petit nombre d'opérations,
  - ◇ pour chaque arête (ou arc) issue de ce sommet, on réalise encore un petit nombre d'opérations,

  On a donc $C = \mathrm{O}(n + m)$.
- Si G est implémenté par une matrice d'adjacence :
  - ◇ pour chaque sommet w parcouru, on a un petit nombre d'opérations,

Complexité

- 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),
- si G est implémenté par une liste d'adjacence :
  - pour chaque sommet w parcouru, on a un petit nombre d'opérations,
  - pour chaque arête (ou arc) issue de ce sommet, on réalise encore un petit nombre d'opérations,

  On a donc $C = \mathrm{O}(n + m)$.
- Si G est implémenté par une matrice d'adjacence :
  - pour chaque sommet w parcouru, on a un petit nombre d'opérations,
  - pour explorer les arêtes issues de ce sommet, on doit parcourir la ligne correspondante dans la matrice, soit n opérations,

Complexité

- 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),
- si G est implémenté par une liste d'adjacence :
  - pour chaque sommet w parcouru, on a un petit nombre d'opérations,
  - pour chaque arête (ou arc) issue de ce sommet, on réalise encore un petit nombre d'opérations,

  On a donc $C = \mathrm{O}(n + m)$.
- Si G est implémenté par une matrice d'adjacence :
  - pour chaque sommet w parcouru, on a un petit nombre d'opérations,
  - pour explorer les arêtes issues de ce sommet, on doit parcourir la ligne correspondante dans la matrice, soit n opérations,

  On a donc $C = \mathrm{O}\left(n^2\right)$.

Complexité

- 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),
- si G est implémenté par une liste d'adjacence :
  - pour chaque sommet w parcouru, on a un petit nombre d'opérations,
  - pour chaque arête (ou arc) issue de ce sommet, on réalise encore un petit nombre d'opérations,

  On a donc $C = \mathrm{O}(n + m)$.
- Si G est implémenté par une matrice d'adjacence :
  - pour chaque sommet w parcouru, on a un petit nombre d'opérations,
  - pour explorer les arêtes issues de ce sommet, on doit parcourir la ligne correspondante dans la matrice, soit n opérations,

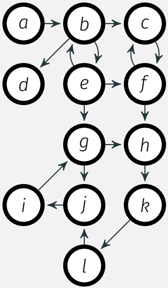
  On a donc $C = \mathrm{O}\left(n^2\right)$.
- On a toujours $m \leq n(n-1)/2$, donc l'implémentation par liste est meilleure d'un point de vue complexité.

Applications

Recherche d'un chemin

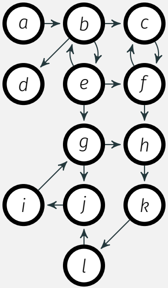
- Les parcours dfs et bfs visitent tous les sommets accessibles depuis le sommet de départ `v` : `w` appartient à la liste des sommets visités depuis `v` $\implies$ il existe au moins un chemin de `v` vers `w`.

- Les parcours dfs et bfs visitent tous les sommets accessibles depuis le sommet de départ `v` : `w` appartient à la liste des sommets visités depuis `v` $\implies$ il existe au moins un chemin de `v` vers `w`.



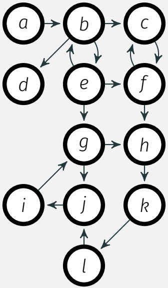
Recherche d'un chemin

- Les parcours dfs et bfs visitent tous les sommets accessibles depuis le sommet de départ `v` : `w` appartient à la liste des sommets visités depuis `v` $\Longrightarrow$ il existe au moins un chemin de `v` vers `w`.



- dfs depuis `a` : `L_visited=['a','b','e','g','j','i']`. Il existe un chemin de `a` vers `i`.

- Les parcours dfs et bfs visitent tous les sommets accessibles depuis le sommet de départ `v` : `w` appartient à la liste des sommets visités depuis `v` $\implies$ il existe au moins un chemin de `v` vers `w`.

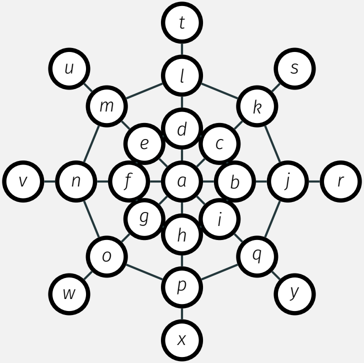


- dfs depuis `a` : `L_visited=['a','b','e','g','j','i']`. Il existe un chemin de `a` vers `i`.
- dfs depuis `k` : `L_visited=['k','l','j','i','g','h']`. Il n'existe pas de chemin de `k` vers `e`.

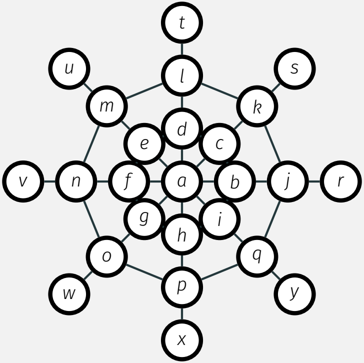
Calcul de la distance entre deux sommets

Nous allons dans cette section adapter le parcours en largeur afin de pouvoir calculer la distance entre deux sommets.

Nous allons dans cette section adapter le parcours en largeur afin de pouvoir calculer la distance entre deux sommets.
Considérons le graphe suivant :



Nous allons dans cette section adapter le parcours en largeur afin de pouvoir calculer la distance entre deux sommets.
Considérons le graphe suivant :



Vérifions que dans un parcours en largeur, les sommets sont visités par distance croissante depuis un sommet de départ donné.

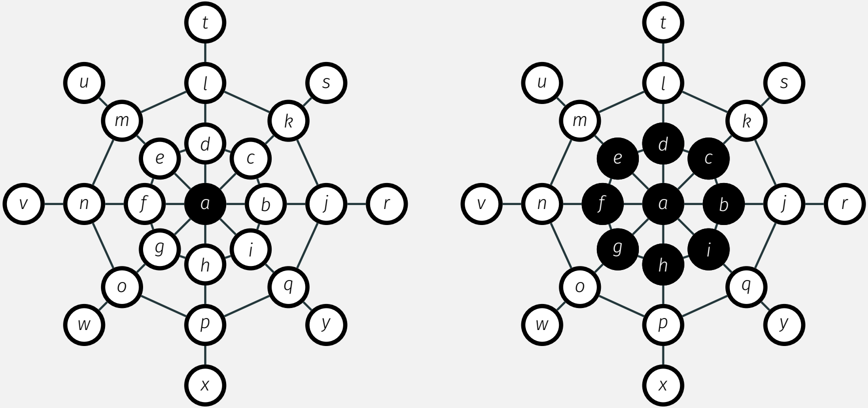
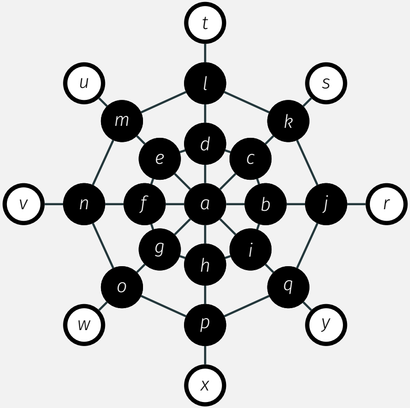
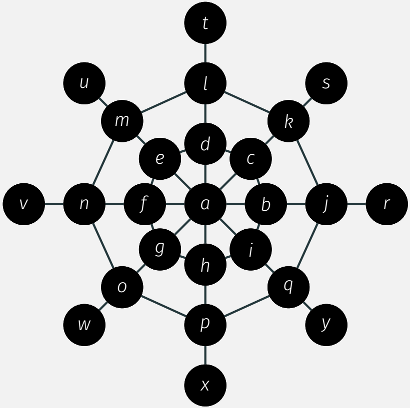


Figure 14 – Étapes du parcours en largeur du Graphe G_1


t
u
l
s
m
k
e
d
c
v
n
f
a
b
j
r
g
h
i
o
q
w
p
y
x


t
u
l
s
m
k
e
d
c
v
n
f
a
b
j
r
g
h
i
o
q
w
p
y
x

Voici le code qui crée le dictionnaire des distances de `v` aux sommets du graphe `g` :

Calcul de la distance entre deux sommets

Voici le code qui crée le dictionnaire des distances de v aux sommets du graphe g :

```
def bfs_dist(g, v):
    q = deque()                                  # déclaration d'une file vide
    visited = {x : False for x in g.keys()}      # dictionnaire des sommets visités
    dist = {x : 0 for x in g.keys()}             # 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 g[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
```

Exercice : calcul de la distance entre deux sommets

Adapter le code précédent au cas d'un graphe codé par matrice d'adjacence.

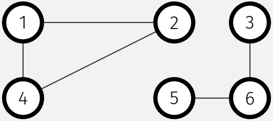
Exercice : calcul de la distance entre deux sommets

Adapter le code précédent au cas d'un graphe codé par matrice d'adjacence.

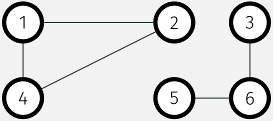
```
def bfs_dist(A, v):
    n = len(A)
    q = deque()
    visited = {i : False for i in range(n)}
    dist = {i : 0 for i in range(n)}
    q.append(v)
    visited[v] = True
    while len(q) > 0:
        i = q.popleft()
        for j in range(n):
        if not visited[j] and M[i,j]:
            visited[j] = True
            q.append(j)
            dist[j] = 1 + dist[i]
    return dist
```

nombre de sommets
déclaration d'une file vide
dictionnaire des sommets visités
dictionnaire des distances
le sommet de départ est enfilé
et marqué comme visité
visite de tous les sommets
défilement du sommet i
parcours des voisins de i
si un sommet n'a pas été visité
il est marqué comme visité
et enfilé
sa distance à v est mise à jour

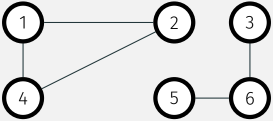
Composantes connexes



- Tous les sommets visités par un parcours à partir d'un sommet v appartiennent à la même composante connexe,



- Tous les sommets visités par un parcours à partir d'un sommet v appartiennent à la même composante connexe,
- S'il reste des sommets non visités, le graphe comporte plusieurs composantes connexes. On peut alors effectuer de nouveaux parcours à partir de sommets non visités, jusqu'à avoir visité tous les sommets.

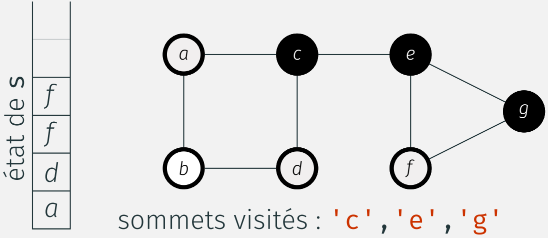


- Tous les sommets visités par un parcours à partir d'un sommet `v` appartiennent à la même composante connexe,
- S'il reste des sommets non visités, le graphe comporte plusieurs composantes connexes. On peut alors effectuer de nouveaux parcours à partir de sommets non visités, jusqu'à avoir visité tous les sommets.
- Exemple : parcours depuis 1 : `lst_visited=[1,4,2]`, puis parcours depuis 3 : `lst_visited=[3,6,5]`.

Détection de cycle

- 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,

- 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,



- pour un graphe orienté, la détection de cycle est plus délicate, mais s'appuie sur des parcours en profondeur.