

## Graphes et leurs applications Quelques définitions et propriétés

ADT Graphe

Liste d'arêtes

Liste d'adjacence

Map d'adjacence

Matrice d'adjacence

Implémentation de Graph.py

Parcours en profondeur

Parcours en largeur

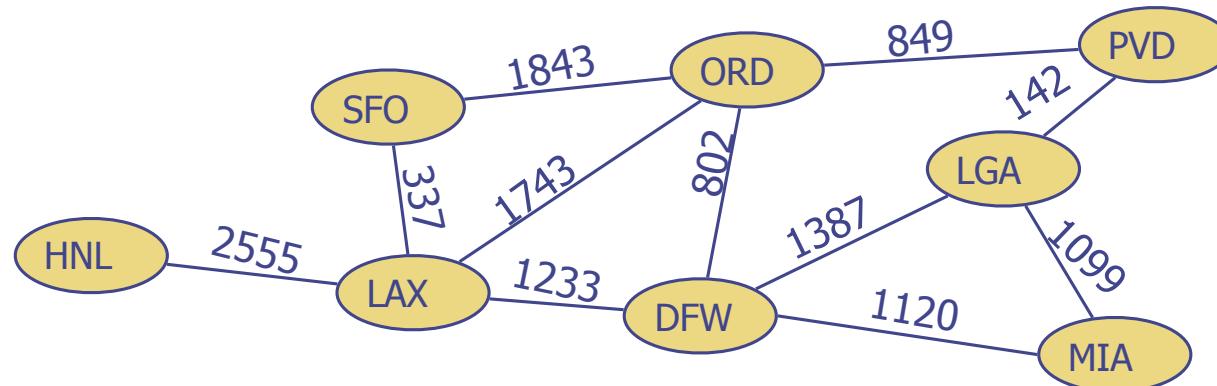
# Graphes

Un **graphe** est une paire  $(V, E)$ , où

- $V$  est un ensemble de nœuds, appelés sommets et
- $E$  est un ensemble de paires de sommets, appelés arêtes
- Les sommets et les arêtes sont des positions qui stockent des éléments

Exemple :

- Un sommet représente un aéroport et stocke son code à trois lettres
- Une arête représente un vol entre deux aéroports et stocke la distance de sa route (ici en miles ; 1 mile = 1.6 km)



# Types d'arêtes

## Arête dirigée/orientée

- paire ordonnée de sommets ( $u, v$ )
- le premier sommet  $u$  est l'origine
- le deuxième sommet  $v$  est la destination
- par exemple, un vol de Chicago à Providence
- étiquette #vol American Airlines 1206



## Arête non dirigée/orientée

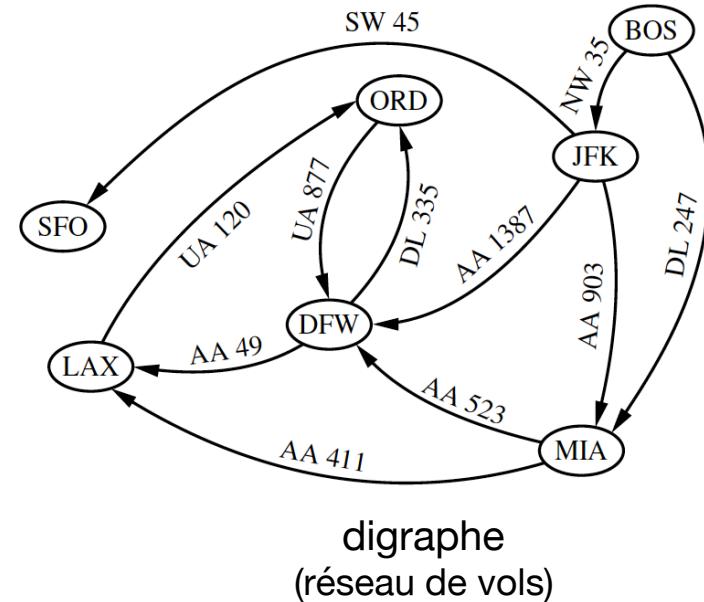
- paire non ordonnée de sommets ( $u, v$ )
- par exemple, la route d'un vol
- étiquette distance de la route 849 miles (1358 km)

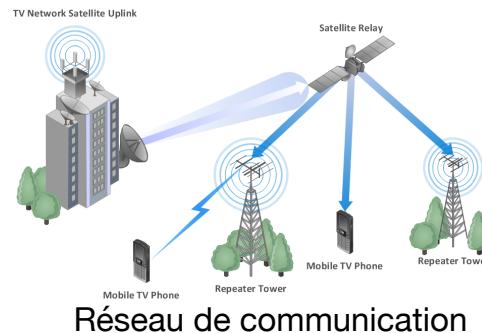
## Graphe dirigé/orienté (digraphe)

- toutes les arêtes sont dirigées
- par exemple, un réseau de vols

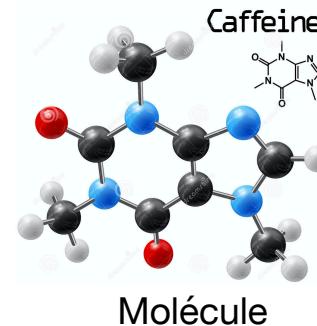
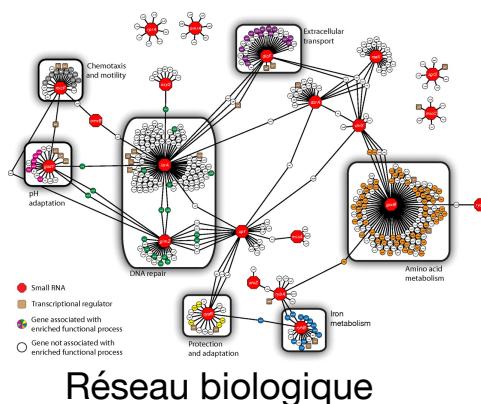
## Graphe non-dirigé/non-orienté

- toutes les arêtes sont non dirigées
- par exemple, un réseau de routes





## Applications



Molécule

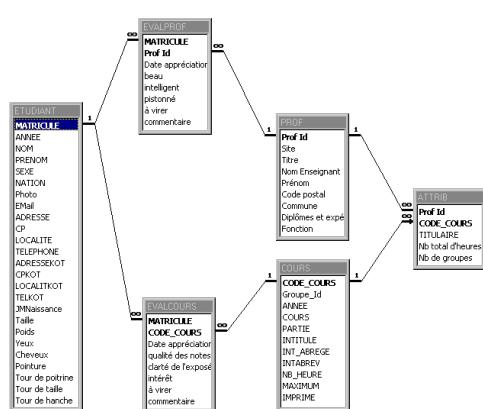
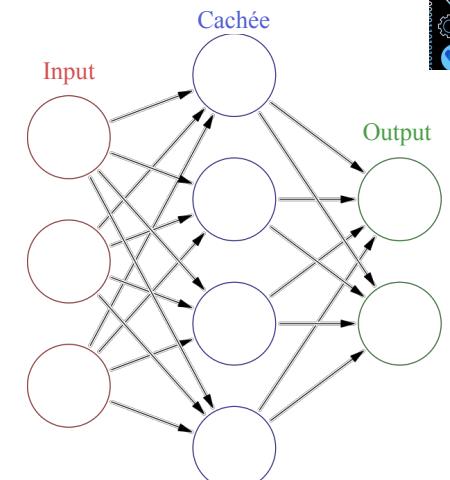
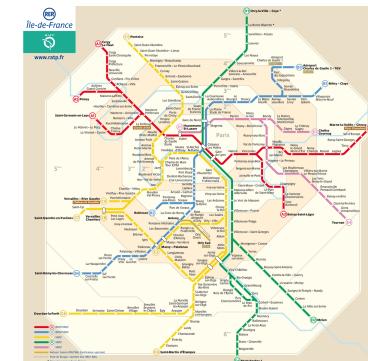
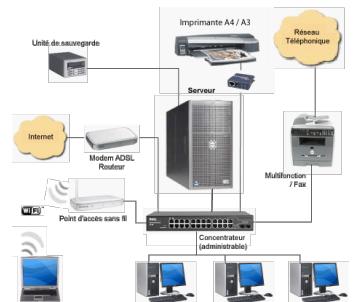


Diagramme entité-relation



Réseau de neurones artificiels



Réseau informatique

# Terminologie des sommets et arêtes

Sommets terminaux (ou extrémités) d'une arête

- $U$  et  $V$  sont les extrémités de l'arête  $a$

Arêtes adjacentes/propres à un sommet

(si elles rejoignent ce sommet)

- $a$ ,  $d$  et  $b$  sont propres à  $V$

Sommets adjacents (si connectés par une arête)

- $U$  et  $V$  sont adjacents

Degré d'un sommet : nombre d'arêtes adjacentes

(cas particulier pour les boucles qui ajoutent +2)

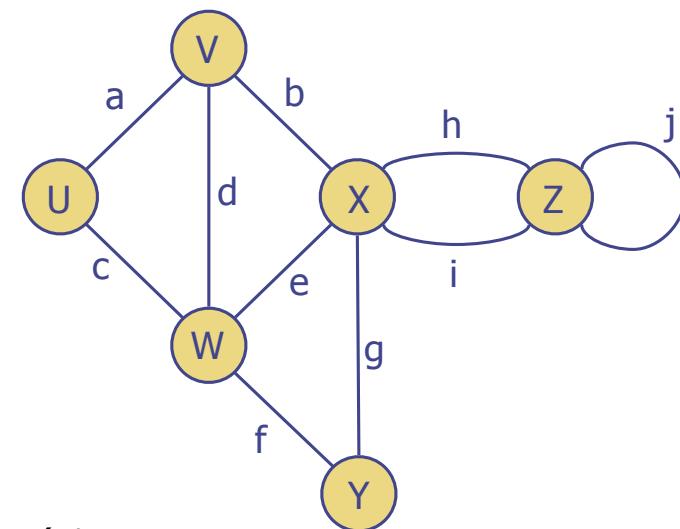
- $X$  a un degré 5 ;  $Y$  a un degré 2 ;  $Z$  a un degré 4

Arêtes parallèles (si elles connectent les mêmes extrémités)

- $h$  et  $i$  sont des arêtes parallèles

Auto-boucle : une arête qui connecte une extrémité à elle-même

- $j$  est une auto-boucle



# Terminologie des chemins

## Chemin

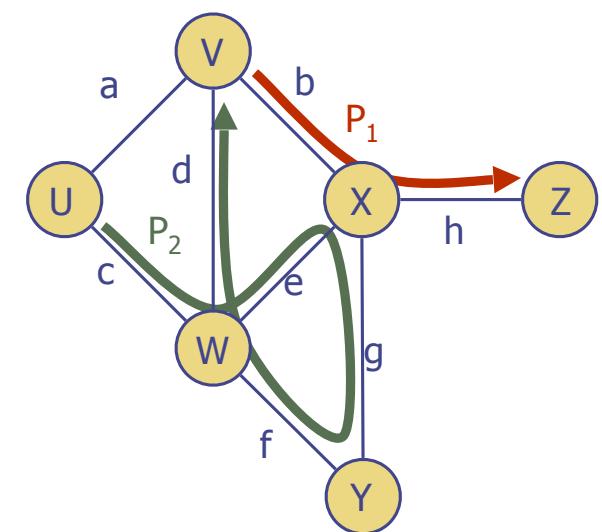
- séquence de sommets et d'arêtes alternés qui :
  - commence par un sommet
  - se termine par un sommet
- chaque arête est précédée et suivie de ses extrémités

## Chemin simple

- chemin tel que tous ses sommets et ses arêtes sont distincts

## Exemples :

- $P_1 = (V, b, X, h, Z)$  est un chemin simple
- $P_2 = (U, c, W, e, X, g, Y, f, W, d, V)$  est un chemin qui n'est pas simple



# Terminologie des cycles

## Cycle

- séquence circulaire de sommets et d'arêtes alternés
- chaque arête est précédée et suivie de ses extrémités

## Cycle simple

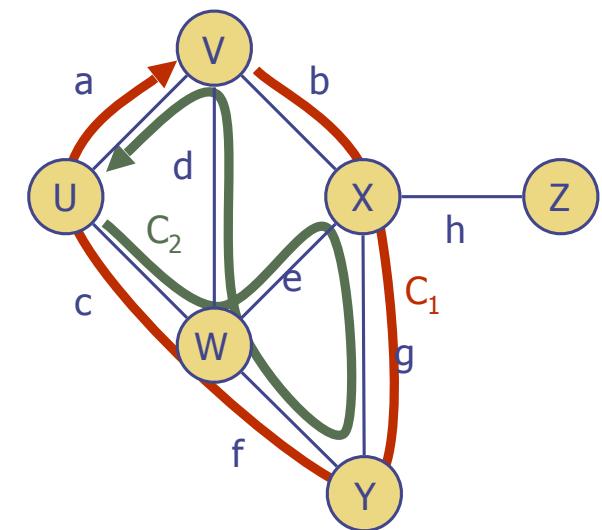
- cycle tel que tous ses sommets et ses arêtes sont distincts  
(sauf pour les sommets de départ et d'arrivée ; on peut aussi utiliser un symbole particulier pour marquer le retour au sommet de départ, e.g.  $\leftarrow$ )

## Exemples :

- $C_1 = (V, b, X, g, Y, f, W, c, U, a, \leftarrow)$  est un cycle simple
- $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, \leftarrow)$  est un cycle qui n'est pas simple

## Graphe acyclique

- un graphe dirigé est acyclique (DAG) s'il ne possède aucun cycle



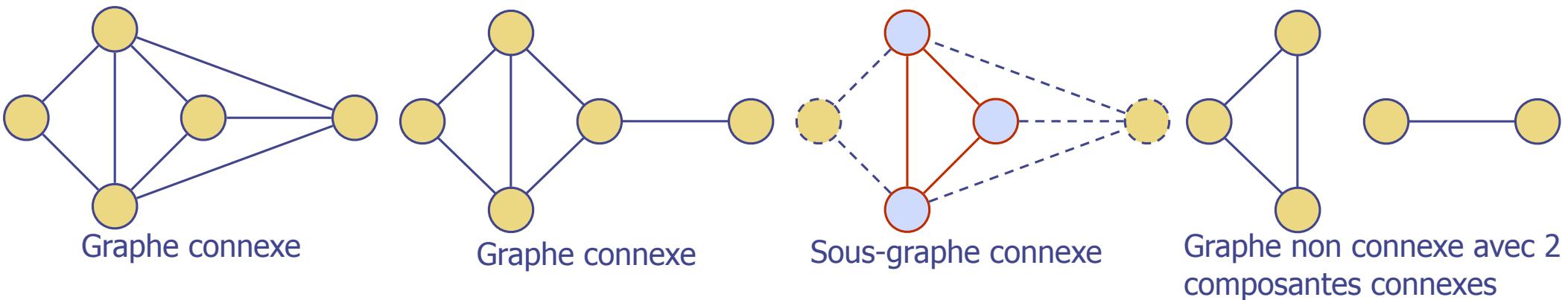
# Autres définitions

Étant donné les sommets  $u$  et  $v$  d'un graphe (dirigé)  $G$ , on dit que  $u$  rejoint/atteint  $v$ , et que  $v$  est accessible depuis  $u$ , si  $G$  possède un chemin (dirigé) de  $u$  vers  $v$ .

Dans un graphe non orienté, la notion d'accessibilité est symétrique, c'est-à-dire que  $v$  est accessible de  $u$  si et seulement si  $u$  est aussi accessible de  $v$ . Cependant, dans un graphe orienté, il est possible qu'on puisse rejoindre/atteindre  $v$  de  $u$  mais que  $v$  ne permette pas de rejoindre  $u$ , car un chemin dirigé doit être traversé selon les directions de ses arêtes.

Un graphe est connexe si, pour deux sommets quelconques, il y a un chemin entre eux. Un graphe orienté,  $G$ , est fortement connexe si pour toute paire de sommets  $u$  et  $v$ ,  $u$  rejoint  $v$  et  $v$  rejoint  $u$ .

Un sous-graphe d'un graphe  $G$  est un graphe  $H$  dont les sommets et les arêtes sont des sous-ensembles des sommets et des arêtes de  $G$ , respectivement.

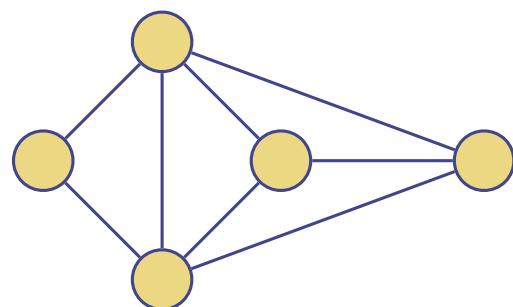


# Autres définitions

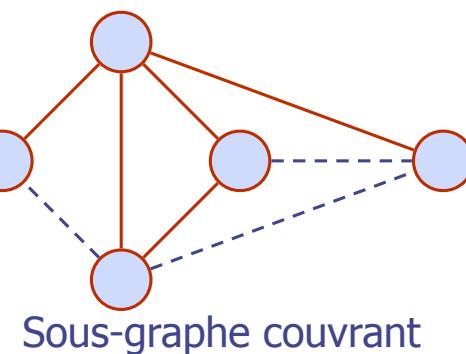
Un sous-graphe couvrant de  $G$  est un sous-graphe de  $G$  qui contient tous les sommets de  $G$ . Si un graphe  $G$  n'est pas connexe, ses sous-graphes maximaux connexes sont appelés les composantes connexes de  $G$ .

Une forêt est un graphe sans cycles. Un arbre est une forêt connexe, c'est-à-dire un graphe connexe sans cycle.

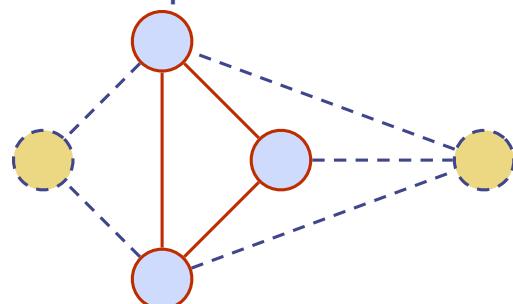
Un arbre couvrant d'un graphe est un sous-graphe couvrant qui est un arbre.



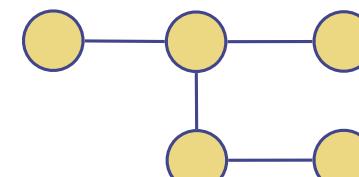
Graphe connexe



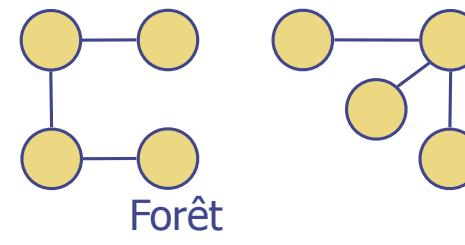
Sous-graphe couvrant



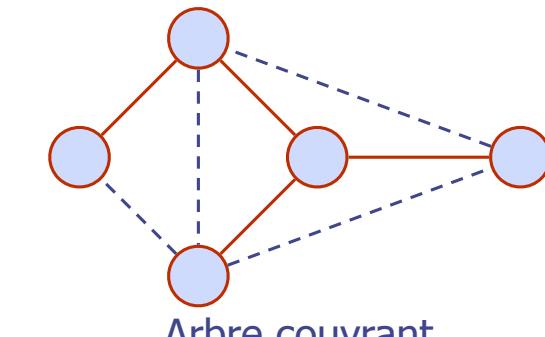
Sous-graphe connexe



Arbre

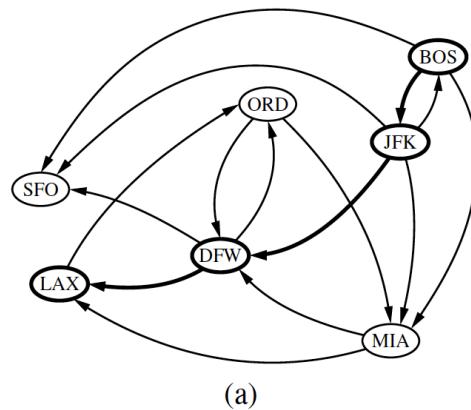


Forêt

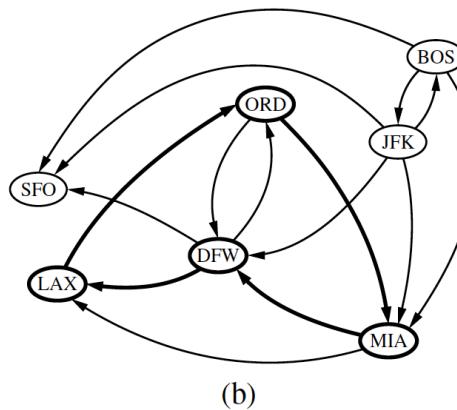


Arbre couvrant

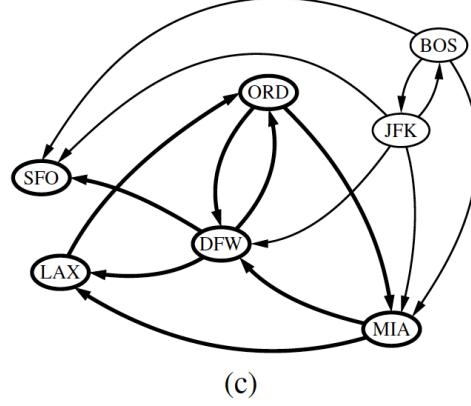
# Exemples



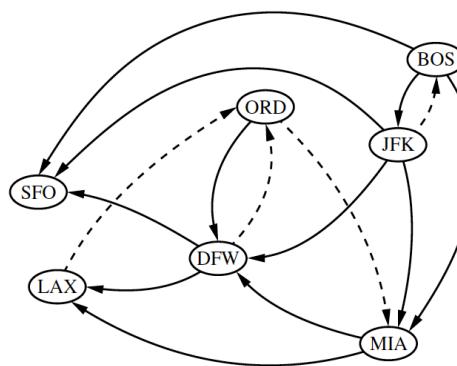
(a)



(b)



(c)



(d)

- (a) un chemin dirigé de BOS vers LAX est montré en gras
- (b) un cycle dirigé (ORD, MIA, DFW, LAX, ORD) est montré en gras
- (c) le sous-graphe des sommets et arêtes accessibles de ORD est montré en gras
- (d) la suppression des arêtes en pointillés résulte en un graphe dirigé acyclique.

# Propriétés

## Propriété

$$\sum_v \deg(v) = 2m$$

Preuve : chaque arête est comptée deux fois

Exemple :  $\sum_v \deg(v) = 3 + 3 + 3 + 3 = 2 \times 6 = 12$

PS. On peut spécialiser la définition pour un graphe dirigé :

$$\sum_v \text{indeg}(v) = \sum_v \text{outdeg}(v) = m$$

## Propriété

Dans un graphe (simple) non orienté sans auto-boucle et sans arêtes multiples

$$m \leq n(n - 1) / 2$$

Preuve : chaque sommet a un degré d'au plus  $(n - 1)$

Exemple :  $m \leq n(n - 1) / 2$

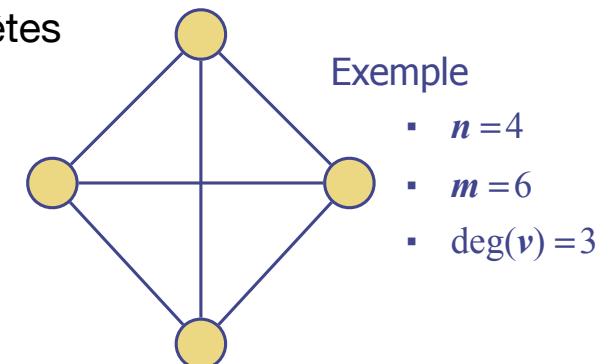
$$6 \leq 4(4 - 1) / 2$$

Implication : un graphe simple de  $n$  sommets possède dans  $O(n^2)$  arêtes.

**Q** : Quelle est la borne sur  $m$  pour un graphe dirigé ?

Dans un graphe dirigé, on peut avoir jusqu'à 2 arêtes pour connecter 2 sommets, donc  $m \leq n(n - 1)$

$n$ :	nombre de sommets
$m$ :	nombre d'arêtes
$l$ :	nombre d'arêtes-boucles
$\deg(v)$ :	degré du sommet $v$
$\text{indeg}(v)$ :	nombre d'arêtes entrantes
$\text{outdeg}(v)$ :	nombre d'arêtes sortantes



Exemple

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

# L'ADT Graphe : Sommets et arêtes

Un graphe est une collection de sommets et d'arêtes.

Nous modélisons l'abstraction d'un graphe comme une combinaison de trois types de données: *Sommets*, *Arêtes* et *Graphe*.

Un sommet est un objet qui stocke un élément arbitraire fourni par l'utilisateur (par exemple, un code d'aéroport)

*On suppose qu'il supporte une méthode, `element()`, pour récupérer l'élément stocké à un sommet*

Une arête stocke un objet associé (par exemple, un numéro de vol, une distance de route, un coût)

*On suppose qu'il supporte aussi une méthode, `element()`, pour récupérer l'élément stocké à une arête*

De plus, on suppose qu'une arête prend en charge les méthodes suivantes :

*`endpoints()` : retourne un tuple  $(u, v)$  tel que  $u$  est l'origine de l'arête et  $v$  sa destination ; pour un graphe non orienté, l'orientation est arbitraire*

*`opposite(v)` : assumant que  $v$  est une extrémité de l'arête (son origine ou sa destination), retourne l'autre extrémité*

# L'ADT Graphe : méthodes

On suppose qu'un graphe peut être non dirigé ou dirigé, et que cette information est déclarée lors de son instantiation

Un graphe mixte peut être représenté comme un graphe orienté en modélisant l'arête  $\{ u, v \}$  comme une paire d'arêtes dirigées  $( u, v )$  et  $( v, u )$ .

L'ADT Graphe comprend les méthodes suivantes :

`vertex_count()` : Renvoie le nombre de sommets du graphe.

`vertices()` : Renvoie un itérateur sur les sommets du graphe.

`edge_count()` : Renvoie le nombre d'arêtes du graphe.

`edges()` : Renvoie un itérateur sur les arêtes du graphe.

`get_edge( u, v )` : Renvoie l'arête connectant le sommet  $u$  au sommet  $v$  s'il en existe un. Sinon, renvoie `None`. Pour un graphe non orienté, il n'y a pas de différence entre `get_edge( u, v )` et `get_edge( v, u )`.

`degree( v, out = True )` : Pour un graphe non orienté, renvoie le nombre d'arêtes adjacents au sommet  $v$ . Pour un graphe orienté, renvoie le nombre d'arêtes sortantes (respectivement entrantes) adjacentes au sommet  $v$ , tel que désigné par le paramètre optionnel `out`.

`incident_edges( v, out = True )` : Renvoie un itérateur sur les arêtes adjacentes au sommet  $v$ . Dans le cas d'un graphe orienté, l'itérateur est sur les arêtes sortantes par défaut, sinon sur les arêtes entrantes, selon le paramètre optionnel `out`.

`insert_vertex( x = None )` : Crée et retourne un nouveau sommet avec la valeur  $x$  (`None` par défaut).

`insert_edge( u, v, x = None )` : Crée et retourne une nouvelle arête du sommet  $u$  au sommet  $v$  avec la valeur  $x$  (`None` par défaut).

`remove_vertex( v )` : Supprime le sommet  $v$  et toutes ses arêtes propres.

`remove_edge( e )` : Supprime l'arête  $e$ .

# Structures de données pour les graphes

On présente 4 structures de données pour représenter un graphe de  $n$  sommets et  $m$  arêtes. Chacune maintient une collection pour stocker les sommets. Cependant, les 4 diffèrent grandement dans la façon d'organiser les arêtes.

- Dans une liste d'arêtes, on conserve une liste non ordonnée de toutes les arêtes. Ceci est suffisant, mais ne permet pas de localiser efficacement une arête particulière,  $(u, v)$ , ou l'ensemble de toutes les arêtes propres à un sommet  $v$ .
- Dans une liste d'adjacence, on conserve, pour chaque sommet, une liste séparée des arêtes propres au sommet. L'ensemble complet de toutes les arêtes se trouve en prenant l'union des ensembles. Cette organisation nous permet de trouver plus efficacement (qu'avec une liste d'arêtes) une arête en particulier ou toutes les arêtes propres à un sommet donné.
- Une Map d'adjacence est très similaire à la liste d'adjacence, mais on utilise une Map (implémentée avec hachage) pour les arêtes propres à un sommet, plutôt qu'une liste, avec le sommet adjacent comme clé. Cela permet d'accéder à une arête spécifique  $(u, v)$  en  $O(1)$  attendu.
- Une matrice d'adjacence fournit un accès à une arête spécifique,  $(u, v)$ , en  $O(1)$  en pire cas en maintenant une matrice  $n \times n$  pour un graphe de  $n$  sommets. Chaque entrée est dédiée au stockage d'une référence à l'arête  $(u, v)$  pour une paire particulière des sommets  $u$  et  $v$ . Si aucune arête n'existe, l'entrée est *None*.

# Liste d'arêtes

On utilise une liste positionnelle doublement chaînée, qui nous donne `remove_edge( e )` en  $O(1)$ , lorsqu'on a accès par sa position. Utilise un espace mémoire dans  $O(n+m)$ .

## Objet Sommet

- référence à un élément
- référence à sa position dans la séquence de sommets,  $V$

## Objet Arete

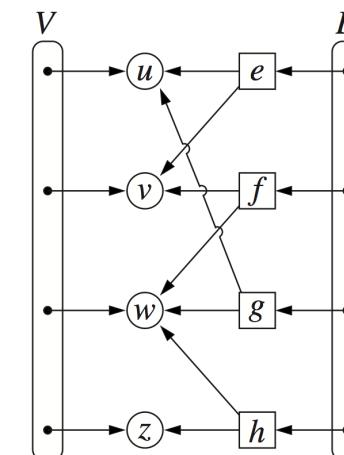
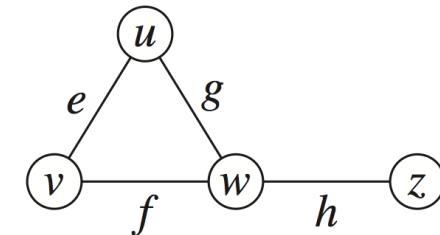
- référence à un élément
- référence au sommet d'origine
- référence au sommet de destination
- référence à sa position dans la séquence d'arêtes,  $E$

## Séquence de Sommet ( $V$ )

- séquence d'objets de type Sommet

## Séquence d'Arete ( $E$ )

- séquence d'objets de type Arete



Opérations pour $n$ sommets et $m$ arêtes	Temps d'exécution (pire cas)
<code>vertex_count()</code> , <code>edge_count()</code>	$O(1)$
<code>vertices()</code>	$O(n)$
<code>edges()</code>	$O(m)$
<code>get_edge( u, v )</code> , <code>degree( v )</code> , <code>incident_edges( v )</code>	$O(m)$
<code>insert_vertex( x )</code> , <code>insert_edge( u, v, x )</code> , <code>remove_edge( e )</code>	$O(1)$
<code>remove_vertex( v )</code> *	$O(m)$

\*car il faut aussi supprimer toutes ses arêtes

# Liste d'adjacence

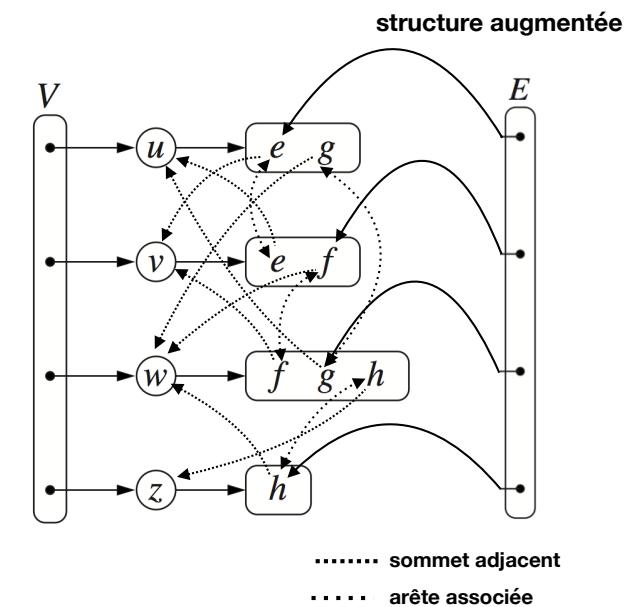
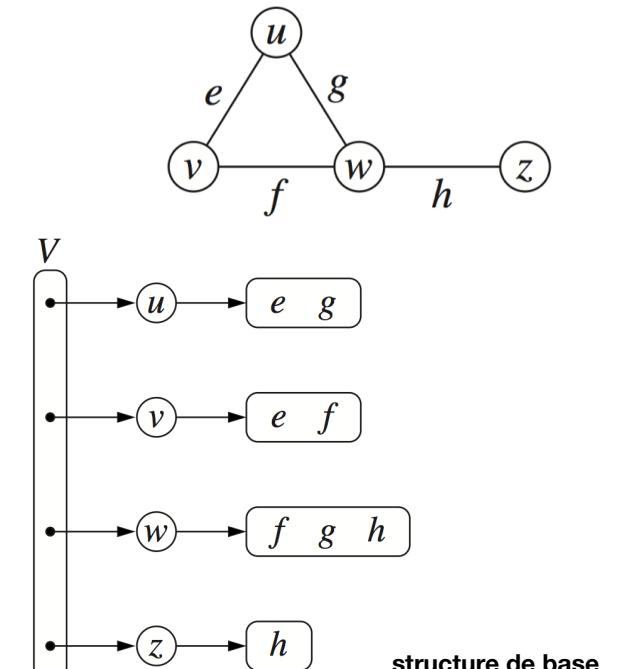
**Structure de base.** On a une liste d'adjacence pour chaque sommet de  $V$ . Comme pour la liste d'arêtes, utilise un espace mémoire dans  $O(n+m)$ .

**Structure augmentée.** Pour `edges()` dans  $O(m)$  et `edge_count()` dans  $O(1)$ , on peut maintenir une liste auxiliaire pour les arêtes, soit une liste de références aux objets de type `Arete`,  $E$

On peut aussi augmenter les objets de type `Arete` avec :

- une référence au sommet adjacent
- une référence à sa 2ème instance, assurant `remove_edge(e)` en  $O(1)$ )

Opérations pour $n$ sommets et $m$ arêtes	Temps d'exécution (pire cas)
<code>vertex_count()</code> , <code>edge_count()</code>	$O(1)$
<code>vertices()</code>	$O(n)$
<code>edges()</code>	$O(m)$
<code>get_edge(u, v)</code>	$O(\min(\text{degree}(u), \text{degree}(v)))$
<code>degree(v)</code>	$O(1)$
<code>incident_edges(v)</code>	$O(\text{degree}(v))$
<code>insert_vertex(x)</code> , <code>insert_edge(u, v, x)</code>	$O(1)$
<code>remove_edge(e)</code>	$O(1)$
<code>remove_vertex(v)</code>	$O(\text{degree}(v))$



# Map d'adjacence

**Structure de base.** Pour permettre d'améliorer `get_edge( u, v )`, on utilise une `HashMap` d'adjacence pour les arêtes propres à chaque sommet.

- l'autre extrémité de l'arête est utilisée comme **clé** et la référence à l'instance `Arete` comme **valeur**

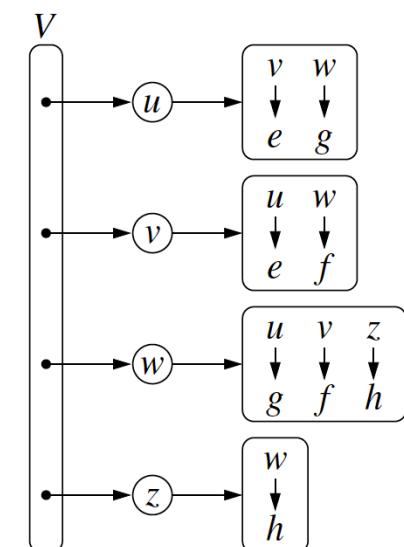
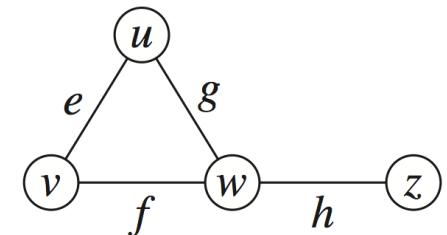
Pas de changement dans l'espace :  $O(n+m)$ .

**Structure augmentée.** Comme pour la liste d'adjacence, pour assurer que `edges()` et `edge_count()` soient efficaces, on peut maintenir une liste auxiliaire pour les arêtes.

Objets de type `Arete` augmentés

- même principe que pour liste d'adjacence

Opérations pour $n$ sommets et $m$ arêtes	Temps d'exécution (pire cas)
<code>vertex_count()</code> , <code>edge_count()</code>	$O(1)$
<code>vertices()</code>	$O(n)$
<code>edges()</code>	$O(m)$
<code>get_edge( u, v )</code>	$O(1)$ attendu
<code>degree( v )</code>	$O(1)$
<code>incident_edges( v )</code>	$O(\text{degree}( v ))$
<code>insert_vertex( x )</code>	$O(1)$
<code>insert_edge( u, v, x )</code>	$O(1)$ attendu
<code>remove_edge( e )</code>	$O(1)$ attendu
<code>remove_vertex( v )</code>	$O(\text{degree}( v ))$



# Matrice d'adjacence

Pour garantir `get_edge( u, v )` dans  $O(1)$  en pire cas, on utilise une matrice d'adjacence.

**Structure de base.** Matrice d'adjacence à 2 dimensions (2D)

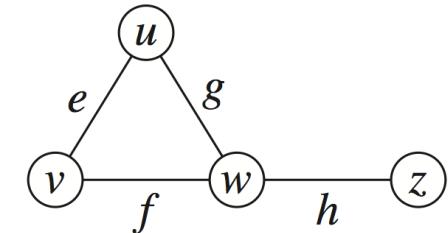
- Espace dans  $O(n^2)$
- Références aux objets de type `Arete` pour chaque sommet adjacent ou `None` pour les sommets non adjacents

Cela déteriorie `insert_vertex( x )` et `remove_vertex( v )` qui change la taille de la matrice (nécessitant des recopierages). Des stratégies d'extension dynamique sont intéressantes à explorer.

**Structure augmentée.** Structure de liste d'arêtes,  $V$ , d'objets Sommet augmentés par :

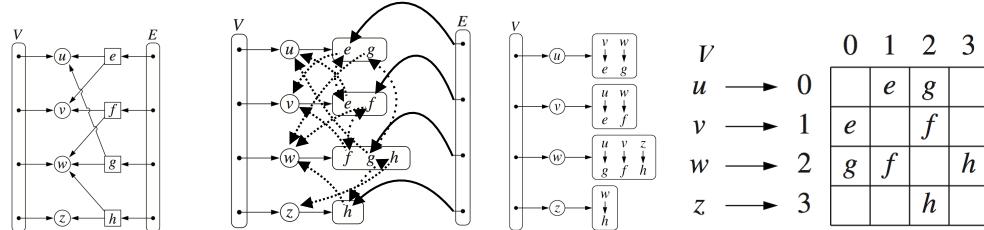
- Clé entière (index) associée au sommet

Opérations pour $n$ sommets et $m$ arêtes	Temps d'exécution (pire cas)
<code>vertex_count()</code> , <code>edge_count()</code>	$O(1)$
<code>vertices()</code>	$O(n)$
<code>edges()</code>	$O(m)$
<code>get_edge( u, v )</code>	$O(1)$
<code>degree( v )</code> , <code>incident_edges( v )</code>	$O(n)$
<code>insert_vertex( x )</code> , <code>remove_vertex( v )</code>	$O(n^2)$
<code>insert_edge( u, v, x )</code> , <code>remove_edge( e )</code>	$O(1)$



$V$	0	1	2	3
$u \rightarrow 0$		<i>e</i>	<i>g</i>	
$v \rightarrow 1$	<i>e</i>		<i>f</i>	
$w \rightarrow 2$	<i>g</i>	<i>f</i>		<i>h</i>
$z \rightarrow 3$			<i>h</i>	

## Résumé des performances en pire cas des opérations sur les graphes



- $n$  sommets,  $m$  arêtes
- pas d'arêtes parallèles
- pas d'auto-loop

	Liste d'arêtes	Liste d'adjacence	Map d'adjacence	Matrice d'adjacence
Espace	$O(n + m)$	$O(n + m)$	$O(n + m)$	$O(n^2)$
<code>vertex_count( )</code>	$O(1)$	$O(1)$	$O(1)$	$O(1)$
<code>edge_count( )</code>	$O(1)$	$O(1)$	$O(1)$	$O(1)$
<code>vertices( )</code>	$O(n)$	$O(n)$	$O(n)$	$O(n)$
<code>edges( )</code>	$O(m)$	$O(m)$	$O(m)$	$O(m)$
<code>get_edge( u, v )</code>	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ att.	$O(1)$
<code>degree( v )</code>	$O(m)$	$O(1)$	$O(1)$	$O(n)$
<code>incident_edges( v )</code>	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
<code>insert_vertex( x )</code>	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
<code>remove_vertex( v )</code>	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
<code>insert_edge( u, v, x )</code>	$O(1)$	$O(1)$	$O(1)$ att.	$O(1)$
<code>remove_edge( e )</code>	$O(1)$	$O(1)$	$O(1)$ att.	$O(1)$

# Implémentation de Graph.py

On utilise une **Map d'adjacence**.

Pour chaque sommet  $v$ , on utilise un dict Python pour maintenir sa liste d'arêtes propres, *outgoing*, et un 2ème dans le cas d'un digraphe, *incoming*.

La liste  $V$  est remplacée par le dict *outgoing* qui utilise les sommets comme clés et leurs dict d'arêtes propres correspondants comme valeurs.

On parcourt tous les sommets en générant l'ensemble des clés du dict *outgoing*.

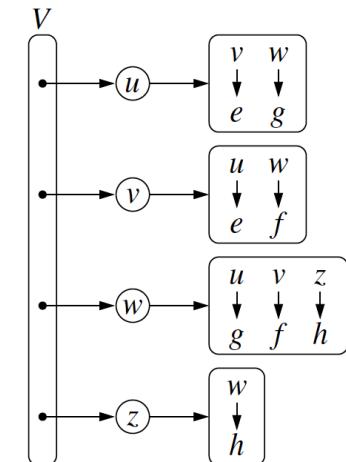
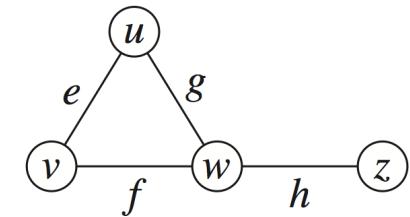
Un sommet n'a pas besoin de maintenir explicitement une référence à sa position dans *outgoing* car on peut le déterminer en O(1)-attendu.

```

outgoing = {
    u : { v : (u,v), w : (u,w) },
    v : { u : (u,v), w : (v,w) },
    w : { u : (v,w), v : (u,w), z : (w,z) },
    z : { w : (w,z) }
}

incoming = outgoing

```



```

#classe Graph
class Graph:

    #classe Vertex pour les Sommets
    #avec une référence à un élément
    class Vertex:
        __slots__ = '_element'

        #constructeur avec élément optionnel
        def __init__( self, x = None ):
            self._element = x

        #accès à l'élément
        def element( self ):
            return self._element

        #prettyprint d'un Sommet
        def __str__( self ):
            return str( self._element )

        #la référence à self comme hashcode
        def __hash__( self ):
            return hash( id( self ) )

    #classe Edge pour les Arêtes
    #avec les Sommets origine et destination et une référence à un élément
    class Edge:
        __slots__ = '_origin', '_destination', '_element'

        #constructeur avec élément optionnel
        def __init__( self, u, v, x = None ):
            self._origin = u
            self._destination = v
            self._element = x

        #retourne les Sommets origine et destination dans un tuple
        def endpoints( self ):
            return( self._origin, self._destination )

        #retourne le Sommet opposé à v
        def opposite( self, v ):
            return self._destination if v is self._origin else self._origin

        #accès à l'élément
        def element( self ):
            return self._element

        #prettyprint d'une Arête
        def __str__( self ):
            return str( self._element )

        #la référence au tuple (origin, destination) comme hashcode
        def __hash__( self ):
            #hachage du tuple (origin, destination)
            return hash( ( self._origin, self._destination ) )

```

# La fonction Python interne `hash()` pour  
# les opérations membres des collections hachables  
# telle que `dict`.

```
#constructeur d'un Graphe vide et non orienté par défaut
def __init__( self, directed = False ):
    #dictionnaire vide pour les dictionnaires
    #d'arêtes propres sortantes de chaque Sommet
    self._outgoing = {}
    #crée un 2ème dictionnaire dans le cas d'un digraphe,
    #pour les arêtes arrivantes de chaque Sommet.
    #par convention, pour un graphe non orienté
    #on utilise un alias vers outgoing (direction indiscernable)
    self._incoming = {} if directed else self._outgoing

#prettyprint d'un graphe, G( V{...}, E{...} )
def __str__( self ):
    s = "G( V{ "
    for v in self.vertices():
        s += str( v ) + " "
    s += "}, E{ "
    for e in self.edges():
        s += str( e ) + " "
    s += " } )"
    return s

#dit si le graphe est dirigé. on utilise le fait que
#incoming est distinct de outgoing dans ce cas
def is_directed( self ):
    return self._incoming is not self._outgoing

#retourne le nombre de Sommets, soit la taille de outgoing
def vertex_count( self ):
    return len( self._outgoing )

#itérateur des sommets, soit les clés de outgoing
def vertices( self ):
    return self._outgoing.keys()

#retourne le nombre d'arêtes en calculant la somme des tailles
#des dictionnaires d'arêtes de chaque sommet, divisée par 2 si digraphe.
def edge_count( self ):
    total = sum( len( self._outgoing[v] ) for v in self._outgoing )
    return total if self.is_directed() else total // 2
```

```
#itérateur sur les arêtes du graphe
def edges( self ):
    #on utilise un ensemble vide
    result = set()
    #et pour chaque dict d'arêtes
    for secondary_dict in self._outgoing.values():
        #on les ajoute à result
        result.update( secondary_dict.values() )
    #set assure une seule instance d'arête pour chacune
    return result

#retourne la valeur stockée dans l'arête (u, v)
def get_edge( self, u, v ):
    return self._outgoing[u].get( v )

#retourne le degré (sortant par défaut) du sommet v,
#la taille du dict associé
def degree( self, v, outgoing = True ):
    adj = self._outgoing if outgoing else self._incoming
    return len( adj[v] )

#itérateur sur les arêtes propres (sortantes par défaut) du sommet v,
#soit les valeurs du dict associé
def incident_edges( self, v, outgoing = True ):
    adj = self._outgoing if outgoing else self._incoming
    for edge in adj[v].values():
        yield edge

#insertion d'un sommet avec élément optionnel
def insert_vertex( self, x = None ):
    #création du Sommet
    v = self.Vertex( x )
    #et de son dict vide outgoing
    self._outgoing[v] = {}
    #et si digraphe incoming
    if self.is_directed():
        self._incoming[v] = {}
    return v

#insertion d'une arête avec élément optionnel
def insert_edge( self, u, v, x = None ):
    e = self.Edge( u, v, x )
    #ajout des 2 arêtes associées
    #origine u vers destination v
    self._outgoing[u][v] = e
    #vers la destination v de l'origine u, pour un digraphe
    #ou on double l'arête non dirigée pour un graphe non orienté
    self._incoming[v][u] = e
```

# Parcourir les noeuds des graphes

Parcourir/traverser un graphe est une procédure systématique pour visiter tous les sommets et toutes les arêtes d'un graphe. Un parcours est efficace si il visite tous les sommets et toutes les arêtes dans un temps proportionnel à leurs nombres, c'est-à-dire en temps linéaire.

Les algorithmes pour parcourir un graphe sont essentiels pour répondre à des questions fondamentales impliquant la notion d'accessibilité, c'est-à-dire, pour "voyager" d'un sommet à un autre en parcourant les chemins du graphe.

Dans la suite de ce module, on présente deux algorithmes efficaces pour parcourir un graphe : la recherche en profondeur et la recherche en largeur.

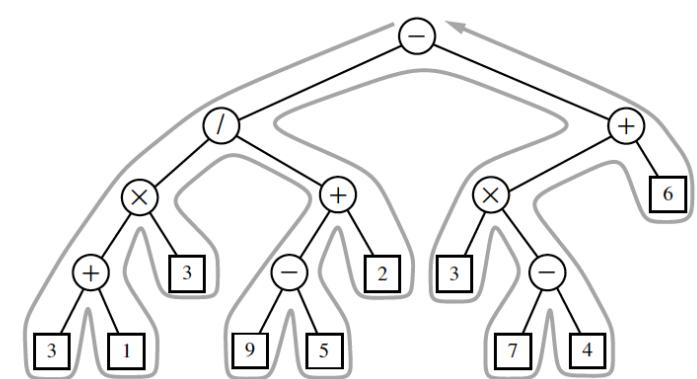
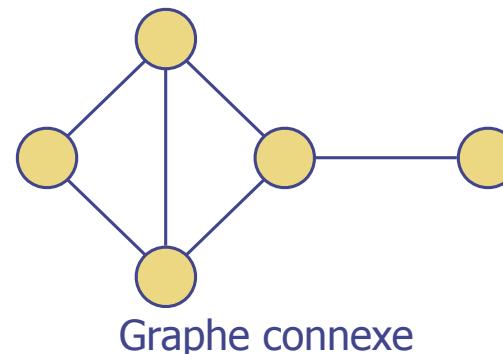
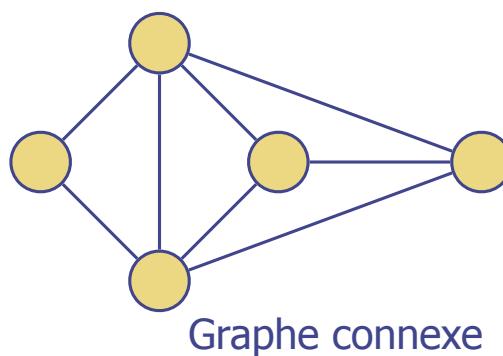
# Recherche en profondeur

La recherche en profondeur (DFS : Depth-First-Search) est une technique générale pour traverser un graphe. Le parcours DFS d'un graphe  $G$  :

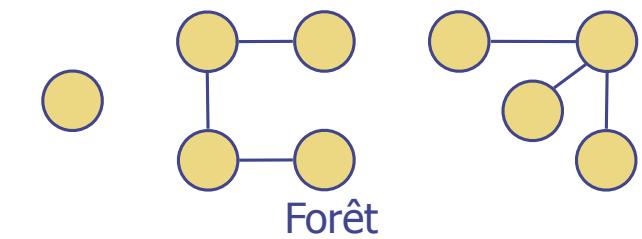
- Visite tous les sommets et arêtes de  $G$
- Détermine si  $G$  est connexe
- Calcule les composantes connexes de  $G$
- Calcule une forêt couvrante de  $G$

La DFS sur un graphe avec  $n$  sommets et  $m$  arêtes s'exécute dans  $O(n + m)$

La DFS est au graphe ce que le parcours d'Euler est aux arbres binaires



```
Algorithme parcoursEuler( T, p ):
    action avant visite pour p
    pour chaque enfant c de T.enfants( p ) faire:
        parcoursEuler( T, c )
    action après visite pour p
```



# Algorithme DFS

La DFS dans un graphe  $G$  est analogue à se promener dans un labyrinthe avec une ficelle et une boîte de peinture.

On débute à un point de départ, par exemple le sommet  $s$ , qu'on initialise en y fixant une extrémité de notre ficelle et en peignant  $s$  "visité".

Le sommet  $s$  est le sommet "courant",  $u = s$ .

On traverse  $G$  en considérant une arête (arbitraire) adjacente à  $u$ ,  $(u, v)$ . Si le sommet  $v$  est déjà peint "visité", on ignore cette arête. Si, au contraire,  $v$  n'a pas été visité, alors on y va en déroulant notre ficelle. Arrivé à  $v$ , on le peint "visité", et on en fait notre sommet "courant".

Éventuellement, on arrive à une "impasse", c'est-à-dire à un sommet pour lequel toutes les arêtes adjacentes mènent à des sommets déjà visités. Pour se sortir de cette impasse, on suit notre ficelle qui nous a mené à  $v$  vers l'arrière (on "backtrack") pour revenir à  $u$ . On continue à explorer les arêtes adjacentes à  $u$  qu'on a pas encore considérées. Si toutes ses arêtes conduisent à des sommets déjà visités, on utilise le même truc pour retourner au sommet précédent, et on poursuit notre quête !

On continue ce "backtracking" jusqu'à ce qu'on trouve un sommet qui a une ou des arêtes encore inexplorées. On continue ainsi jusqu'à ce qu'on revienne au sommet de départ  $s$  et que toutes les arêtes aient été explorées.

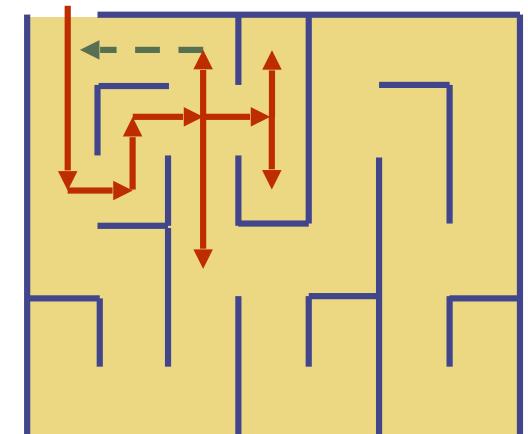
**Algorithme DFS(  $G, u$  ):**

**Input:** Un graphe  $G$  et un sommet  $u$  de  $G$

**Output:** Une collection de sommets accessibles de  $u$  et leurs arêtes y menant  
**pour toute** arête sortante  $e = (u, v)$  de  $u$  **faire**

**si** sommet  $v$  n'a pas été visité **alors**

        marquer sommet  $v$  visité (via l'arête  $e$ )  
         appeler récursivement DFS(  $G, v$  )



# Les types d'arêtes parcourues par DFS

On utilise DFS pour analyser la structure d'un graphe.

La DFS identifie naturellement ce qu'on appelle l'arbre de recherche en profondeur (arbre DFS) enraciné au sommet de départ  $s$ .

Chaque fois qu'une arête  $e = (u, v)$  est empruntée pour découvrir un nouveau sommet  $v$ , cette arête nous y menant est dite découverte (ou arête de l'arbre DFS).

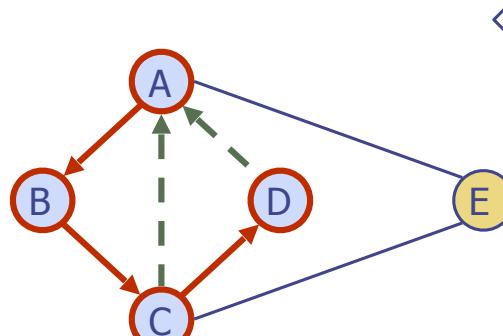
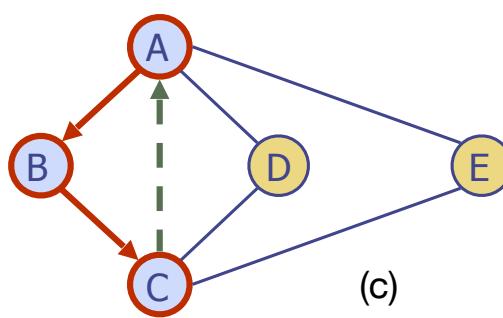
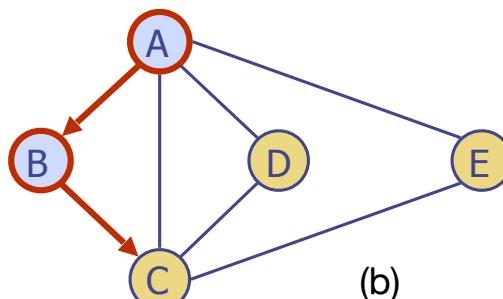
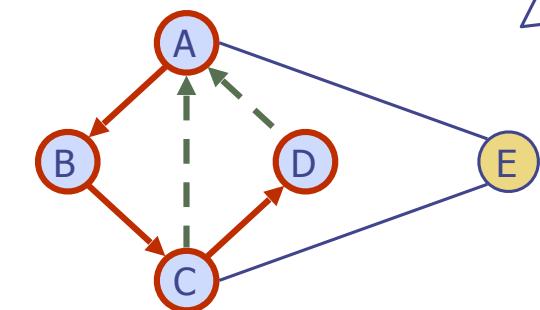
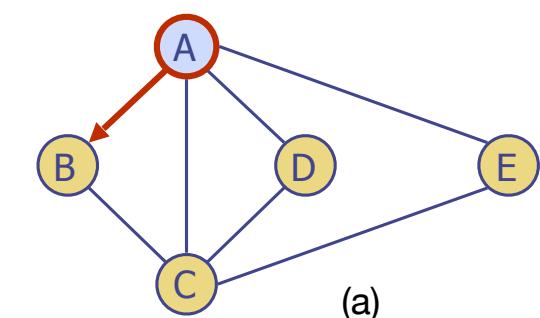
- arête **découverte**, relie deux sommets dans l'arbre DFS

Toutes les autres arêtes nous mènent à un sommet déjà visité et ne font pas partie de l'arbre DFS. Ces arêtes peuvent être de trois types :

- arête **arrière**, relie un sommet à un ancêtre de l'arbre DFS
- arête **avant**, relie un sommet à un descendant dans l'arbre DFS
- arête **croisée**, relie un sommet à un autre qui n'est ni ancêtre ni descendant.

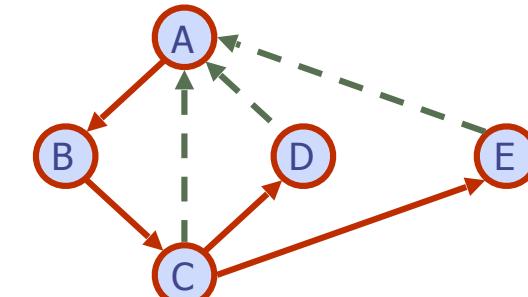
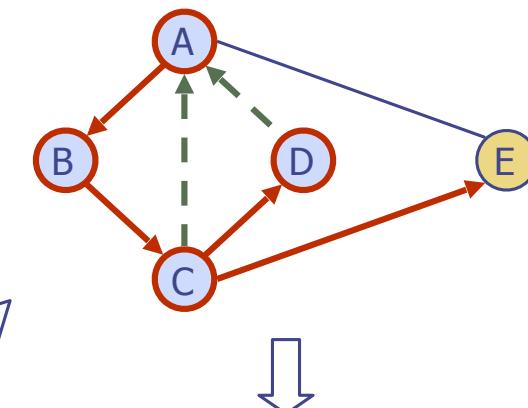
# Exemple de DFS dans un graphe non-orienté

-  sommet inexploré
-  sommet visité
- arête inexploitée
- arête découverte
- -> arête non DFS

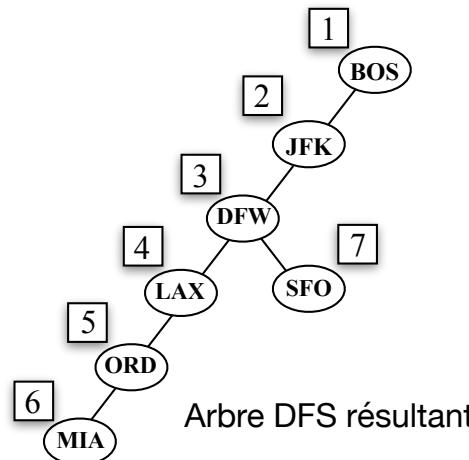
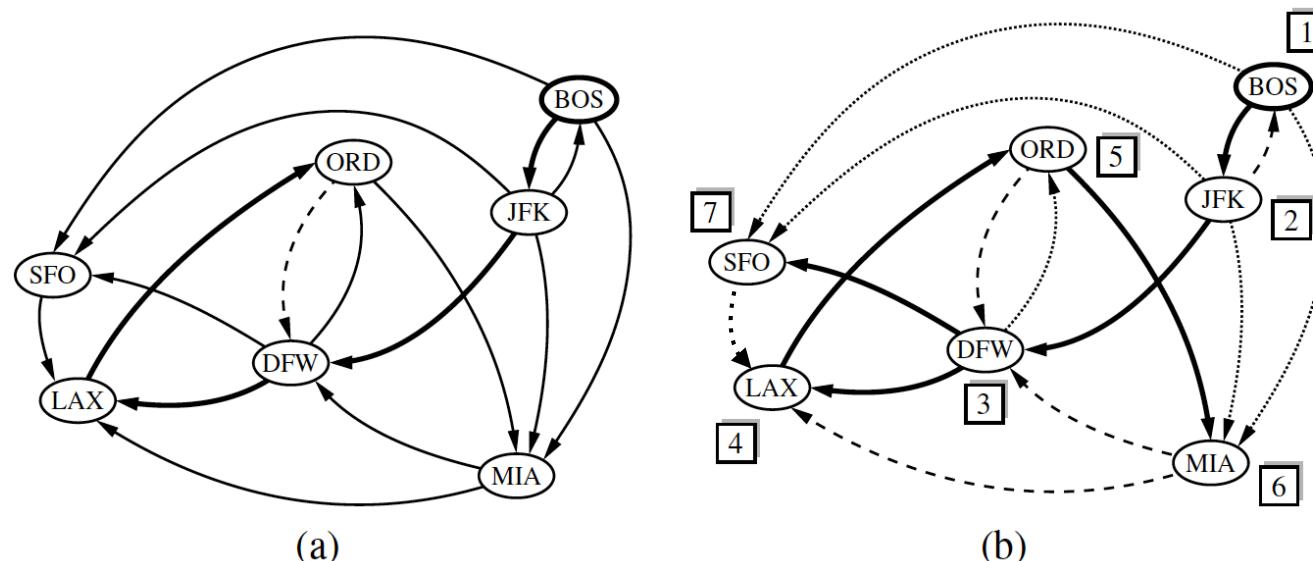


À partir du sommet A :

- Après avoir parcouru l'arête AB. Le sommet B n'avait pas été parcouru, donc l'arête AB fait partie de l'arbre DFS.
  - Après avoir parcouru l'arête BC. C n'ayant pas été parcouru auparavant, l'arête BC fait partie de l'arbre DFS.
  - Le sommet A a déjà été parcouru. L'arête CA ne fait donc pas partie des arêtes découvertes.
- Et ainsi de suite...



# Exemple de DFS dans un graphe orienté

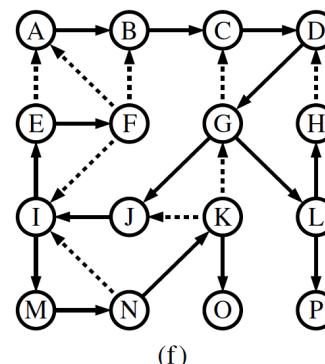
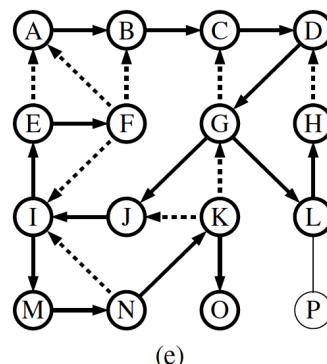
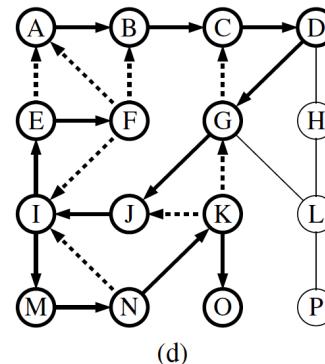
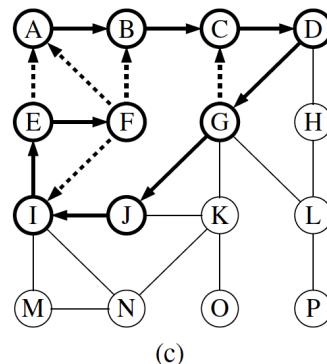
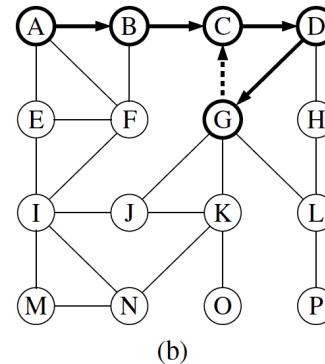
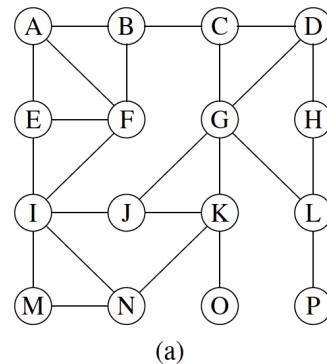


À partir du sommet BOS :

- (a) **Étape intermédiaire.** Pour la première fois, une arête considérée conduit à un sommet visité (DFW);
- (b) **DFS complétée.** L'ordre de visite des sommets est indiqué par une étiquette. L'arête (ORD, DFW) est une **arête arrière**, mais (DFW, ORD) est une **arête avant**. Seule l'arête (SFO, LAX) est une **arête croisée**.

- découverte
- - - → arrière
- ..... → avant
- .... → croisée

# Autre exemple de DFS dans un graphe non orienté



À partir du sommet *A* et en supposant que les sommets sont dans l'ordre alphabétique :

Les sommets visités et les arêtes explorées sont mis en évidence : les **arêtes découvertes** sont dessinées avec des lignes continues en gras ; et, les **arêtes non DFS** avec des lignes pointillées :

- Graphe initial.
- Trajectoire des arêtes de l'arbre DFS, tracée de *A* jusqu'à l'arête arrière (*G, C*).
- Rejoindre *F* représente une impasse.
- Après avoir backtracké vers *I*, on reprend avec l'arête (*I, M*), et on frappe une autre impasse à *O*.
- Après avoir backtracké vers *G*, on continue avec l'arête (*G, L*), et on frappe une autre impasse à *H*.
- Résultat final.

Arêtes non DFS sont toutes arrières.

# Propriétés de la DFS

Propriété Arbre Couvrant : Soit  $G$  un graphe non-orienté sur lequel on lance la DFS à partir du sommet  $s$ . Tous les sommets de la composante connexe incluant  $s$  avec les arêtes découvertes forment un arbre couvrant de cette composante.

Justification:

**Aucun cycle n'est formé.** Puisque nous parcourons seulement des arêtes découvertes, c'est-à-dire qui mènent à des sommets non visités, nous ne formerons jamais de cycle avec ces arêtes. Par conséquent, les arêtes découvertes forment un sous-graphe connexe sans cycle, c'est-à-dire un arbre.

De plus, c'est un arbre couvrant, car la DFS visite tous les sommets dans la composante connexe qui inclut  $s$ .

**Tous les sommets sont visités.** Si la recherche n'a pas visité tous les sommets, un ensemble de sommets ont été visités et un autre ensemble de sommets n'ont pas été visités. Le graphe étant connecté, il existe au moins une arête entre un sommet visité et un sommet non visité. Mais lorsque l'algorithme a atteint ce sommet visité, il a visité successivement chacun de ses voisins, revenant chaque fois après un nombre fini d'étapes, et a donc également visité le sommet non visité. Cette contradiction montre que tous les sommets ont été visités.

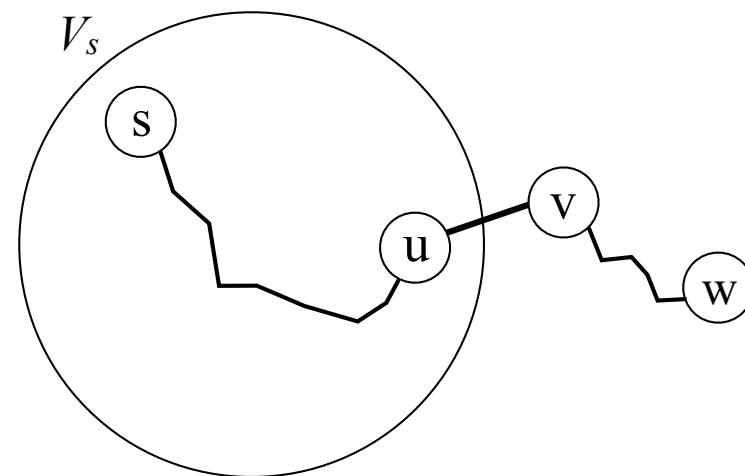
# Propriétés de la DFS

Propriété Accessibilité et chemins dirigés : Soit  $G$  un graphe orienté. Une DFS qui débute au sommet  $s$  visite tous les sommets de  $G$  qui y sont accessibles. L'arbre DFS contient un chemin dirigé de  $s$  vers chaque sommet accessible de  $s$ .

Justification:

**Visite de tous les sommets accessibles.** Soit  $V_s$  le sous-ensemble des sommets de  $G$  visités par la DFS à partir du sommet  $s$ . On veut montrer que  $V_s$  contient  $s$  et chaque sommet accessible depuis  $s$ . Supposons maintenant, par contradiction, qu'il existe un sommet accessible depuis  $s$ ,  $w$ , qui n'est pas dans  $V_s$ . Considérons un chemin dirigé de  $s$  vers  $w$  et  $(u, v)$  la première arête nous menant hors de  $V_s$  ( $u$  est dans  $V_s$  mais pas  $v$ ). Lorsque la DFS rejoint  $u$ , elle explore toutes les arêtes sortantes de  $u$ , et donc rejoint le sommet  $v$  via l'arête  $(u, v)$ . Par conséquent,  $v$  devrait être dans  $V_s$ , et nous avons une contradiction. Ainsi,  $V_s$  doit contenir tous les sommets accessibles de  $s$ .

**L'arbre DFS contient tous les chemins dirigés de  $s$  vers chaque sommet accessible de  $s$ .** Par induction sur les étapes de la DFS, on revendique que chaque fois qu'une arête découverte  $(u, v)$  est identifiée, il existe un chemin dirigé de  $s$  à  $v$  dans l'arbre DFS, puisque  $u$  a déjà été visité, il existe un chemin de  $s$  à  $u$ . On ajoute l'arête  $(u, v)$  à ce chemin pour obtenir un chemin dirigé de  $s$  à  $v$ .



# Performances de la DFS

La DFS est une méthode efficace pour parcourir un graphe. Elle est appelée au plus une fois sur chaque sommet (puisque'ils sont marqués lors de la première visite) et chaque arête est considérée au plus deux fois (pour un graphe non-orienté ; une fois de chaque extrémité), et au plus une fois dans un graphe orienté (à partir de son sommet d'origine).

Si on définit  $n_s \leq n$  comme le nombre de sommets accessibles depuis un sommet  $s$ , et  $m_s \leq m$  le nombre d'arêtes adjacentes à ces sommets, la DFS commençant à  $s$  exécute dans  $O(n_s + m_s)$ , si on a les conditions suivantes :

- L'opération `incident_edges(v)` est dans  $O(\deg(v))$ , et `opposite(v)` est dans  $O(1)$ , réalisés avec une liste d'adjacence, par exemple, mais pas avec une matrice d'adjacence.
- "Marquer" un sommet ou une arête et tester si un sommet ou une arête ont été explorés est dans  $O(1)$ .

Ainsi, on peut résoudre un certain nombre de problèmes intéressants de manière efficace :

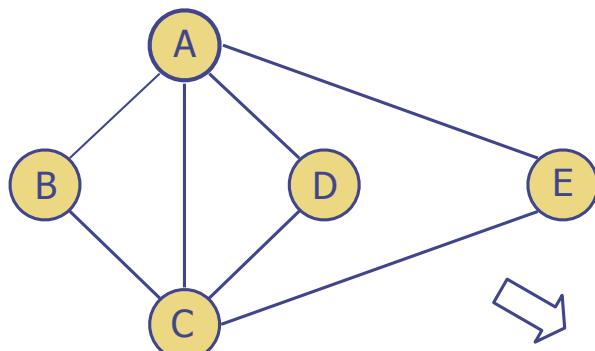
*Dans un graphe G non-orienté de n sommets et m arêtes, la DFS résout les problèmes suivants dans O(n + m) :*

- Calculer un chemin entre deux sommets de G, s'il en existe un
- Tester si G est connexe
- Calculer un arbre couvrant de G, si G est connexe
- Calculer les composantes connexes de G
- Trouver un cycle dans G ou déterminer que G ne contient pas de cycle

*Dans un graphe G orienté avec n sommets et m arêtes, la DFS résoud les problèmes suivants dans O(n + m) :*

- Calculer un chemin dirigé entre deux sommets de G, s'il en existe un
- Calculer l'ensemble des sommets de G qui sont accessibles depuis un sommet donné s
- Tester si G est fortement connexe
- Trouver un cycle dirigé dans G, ou déterminer que G est acyclique
- Calculer la fermeture transitive de G (à voir en IFT2125)

```
#DFS à partir du sommet u
def DFS( g, u, discovered ):
    #pour chaque arête propre au sommet u
    for e in g.incident_edges( u ):
        #accéder au sommet opposé v
        v = e.opposite( u )
        #si non visité
        if v not in discovered:
            #l'ajouter aux sommets visités
            discovered[v] = e
            #résoudre DFS à partir du sommet v
            DFS( g, v, discovered )
```

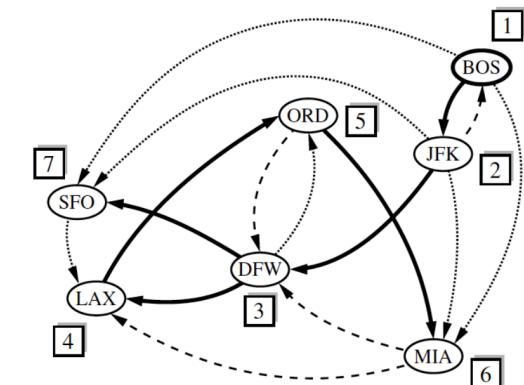
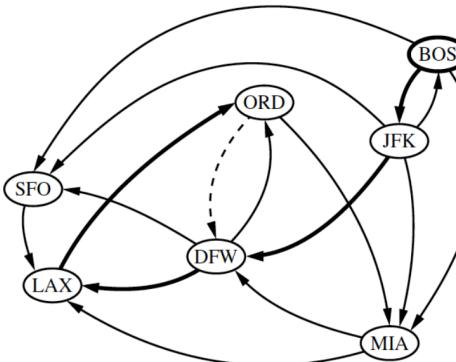


```
g = Graph()
aa = g.insert_vertex( 'A' )
bb = g.insert_vertex( 'B' )
cc = g.insert_vertex( 'C' )
dd = g.insert_vertex( 'D' )
ee = g.insert_vertex( 'E' )

ab = g.insert_edge( aa, bb, 'AB' )
ac = g.insert_edge( aa, cc, 'AC' )
ad = g.insert_edge( aa, dd, 'AD' )
ae = g.insert_edge( aa, ee, 'AE' )
bc = g.insert_edge( bb, cc, 'BC' )
cd = g.insert_edge( cc, dd, 'CD' )
ce = g.insert_edge( cc, ee, 'CE' )
```

Graph unit testing...  
GC V{ A B C D E }, E{ AB AC AD AE BC CD CE }  
DFS du sommet A :  
A -  
B AB  
C BC  
D CD  
E CE

GC V{ BOS JFK MIA ORD DFW SFO LAX }, E{ BOS->SFO JFK->BOS BOS->JFK JFK->DFW JFK->MIA ORD->MIA  
ORD->DFW DFW->SFO DFW->ORD SFO->LAX LAX->ORD BOS->MIA JFK->SFO MIA->DFW MIA->LAX DFW->LAX }  
DFS de BOS :  
BOS -  
JFK BOS->JFK  
DFW JFK->DFW  
LAX DFW->LAX  
ORD LAX->ORD  
MIA ORD->MIA  
SFO DFW->SFO



g148 = Graph( True )

```
bos = g148.insert_vertex( 'BOS' )
jfk = g148.insert_vertex( 'JFK' )
mia = g148.insert_vertex( 'MIA' )
chi = g148.insert_vertex( 'ORD' )
dfw = g148.insert_vertex( 'DFW' )
sfo = g148.insert_vertex( 'SFO' )
lax = g148.insert_vertex( 'LAX' )
```

```
bosjfk = g148.insert_edge( bos, jfk, 'BOS->JFK' )
bosmia = g148.insert_edge( bos, mia, 'BOS->MIA' )
bossfo = g148.insert_edge( bos, sfo, 'BOS->SFO' )
jfkbos = g148.insert_edge( jfk, bos, 'JFK->BOS' )
jfkdfw = g148.insert_edge( jfk, dfw, 'JFK->DFW' )
jfkmia = g148.insert_edge( jfk, mia, 'JFK->MIA' )
jfksfo = g148.insert_edge( jfk, sfo, 'JFK->SFO' )
chimia = g148.insert_edge( chi, mia, 'ORD->MIA' )
chidfw = g148.insert_edge( chi, dfw, 'ORD->DFW' )
sfolax = g148.insert_edge( sfo, lax, 'SFO->LAX' )
laxchi = g148.insert_edge( lax, chi, 'LAX->ORD' )
dfwlax = g148.insert_edge( dfw, lax, 'DFW->LAX' )
dfwsfo = g148.insert_edge( dfw, sfo, 'DFW->SFO' )
dfwchi = g148.insert_edge( dfw, chi, 'DFW->ORD' )
miadfwi = g148.insert_edge( mia, dfw, 'MIA->DFW' )
mialax = g148.insert_edge( mia, lax, 'MIA->LAX' )
```

# Trouver un chemin entre deux sommets

```

#construire un chemin des sommets u à v
#discovered est un dict d'arêtes
def construct_path( g, u, v, discovered ):
    #initialise chemin vide
    path = []
    #on part du sommet destination, v
    #et on remonte vers l'origine, u
    #si v est dans l'arbre DFS
    if v in discovered:
        #on l'ajoute au chemin
        path.append( v )
        #on marque le sommet de destination
        walk = v
        #tantque la destination n'est pas atteinte
        while walk is not u:
            #on prend l'arête dont walk est l'origine
            e = discovered[walk]
            #si elle existe
            if e is not None:
                #on prend le sommet opposé
                parent = e.opposite( walk )
                #on ajoute ce sommet dans le chemin
                path.append( parent )
                #on poursuit la boucle à partir de ce sommet
                walk = parent
        #comme on a construit de la destination à l'origine
        #on inverse la solution
        path.reverse()
    return path

```

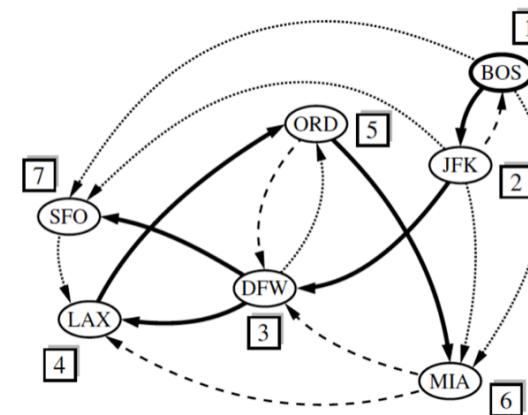
La DFS identifie un chemin (dirigé) entre les sommets  $u$  et  $v$ , si  $v$  est accessible de  $u$ . Ce chemin peut être reconstruit à partir des informations enregistrées dans le dictionnaire des arêtes découvertes. La fonction `construct_path(g, u, v, discovered)` produit une liste ordonnée des sommets sur le chemin entre  $u$  et  $v$ .

Pour reconstruire le chemin, on commence à la destination.

Tant que le sommet origine n'est pas identifié, on remonte les arêtes en prenant le sommet opposé qu'on ajoute au chemin. Et on recommence à partir de ce dernier sommet.

Une fois que nous avons tracé le chemin de  $v$  à  $u$ , il suffit de l'inverser pour obtenir le chemin de  $u$  à  $v$ .

Ce processus prend un temps proportionnel à la longueur du chemin, donc dans  $O(n)$  en pire cas, en plus du temps passé dans la DFS pour déterminer les arêtes découvertes.



Chemin entre BOS et ORD :  
 BOS  
 JFK  
 DFW  
 LAX  
 ORD

# Déterminer la connectivité d'un graphe

La DFS détermine si un graphe  $G$  de  $n$  sommets et  $m$  arêtes est connexe.

Cas d'un graphe non-orienté :

On lance la DFS à partir d'un sommet arbitraire  $v$  et ensuite on teste si  $\text{len}(\text{discovered})$  est égale à  $n$ . Si  $G$  est connexe, alors, d'après la proposition **Arbre Couvrant**, tous les sommets accessibles auront été parcourus et, inversement, si  $G$  n'est pas connexe, il y aura au moins un sommet qui n'est pas accessible depuis  $v$ .

Cas d'un graphe orienté :

On voudrait savoir si  $G$  est fortement connexe, càd si pour chaque paire de sommets  $u$  et  $v$ , on a que  $u$  est accessible de  $v$  et  $v$  est accessible de  $u$ .

On peut lancer la DFS de chaque sommet de  $G$ , mais lancer  $n$  fois la DFS nous coûterait dans  $O(n(n + m))$ .

On peut le faire avec seulement 2 appels à DFS :

La première à partir d'un sommet arbitraire  $s$ . S'il y a un sommet qui n'est pas visité, càd qui n'est pas accessible de  $s$ , alors  $G$  n'est pas fortement connexe.

Sinon, pour montrer que  $G$  est fortement connexe on doit s'assurer que  $s$  est accessible de tous les autres sommets. Conceptuellement, on copie  $G$  et inverse la direction de toutes ses arêtes, créant le graphe  $G'$ . Si  $G$  est fortement connexe, la DFS à partir de  $s$  dans  $G'$  rejoindra tous les sommets pouvant rejoindre  $s$  dans  $G$ .

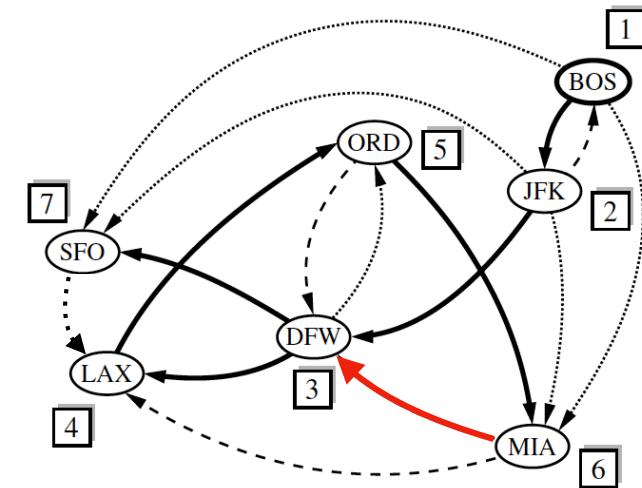
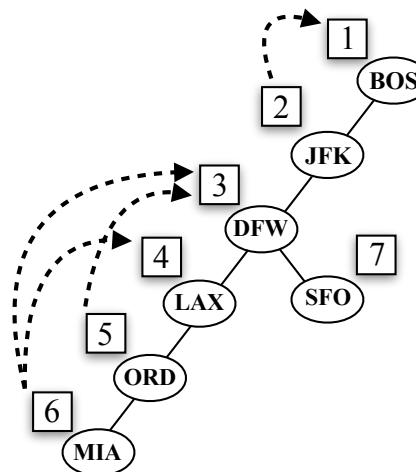
En pratique, plutôt que de créer un nouveau graphe, on réimplémente une version de la DFS qui parcourt les arêtes entrantes vers le sommet courant, plutôt que toutes les arêtes sortantes. Puisque cet algorithme ne fait que lancer la DFS 2 fois, il s'exécute dans  $O(n + m)$ .

## Trouver un cycle dans un graphe, ou déterminer qu'un digraphe est acyclique

Dans un graphe non orienté, toute arête qui ne se retrouve pas dans l'arbre DFS indique un cycle.

Pour un graphe orienté, un cycle existe si et seulement si une arête arrière existe. On voit facilement la création d'un tel cycle en ajoutant une arête arrière à l'arbre DFS. Inversement, si un cycle existe dans le graphe, c'est qu'il doit y avoir une arrête arrière.

Dans le cas d'un graphe orienté, il faut modifier DFS pour identifier correctement une arête arrière. Quand une arête dirigée mène à un sommet déjà visité, il faut **reconnaitre si ce sommet est un ancêtre du sommet actuel dans l'arbre DFS**.



Exemple de détection d'un cycle dans un graphe orienté. L'arête arrière, MIA->DFW en rouge, ainsi que plusieurs autres, MIA->LAX, ORD->DFW et JFK->BOS, indiquent la présence de cycles dans ce graphe. Ces arêtes vont d'un sommet vers un ancêtre dans l'arbre DFS.

Ce n'est pas le cas des arêtes avant ou croisées !

# Calculer les composantes connexes d'un graphe

Quand un graphe n'est pas connexe, on peut identifier ses composantes connexes ou ses composantes fortement connexes, dans un graphe non-orienté et orienté, respectivement.

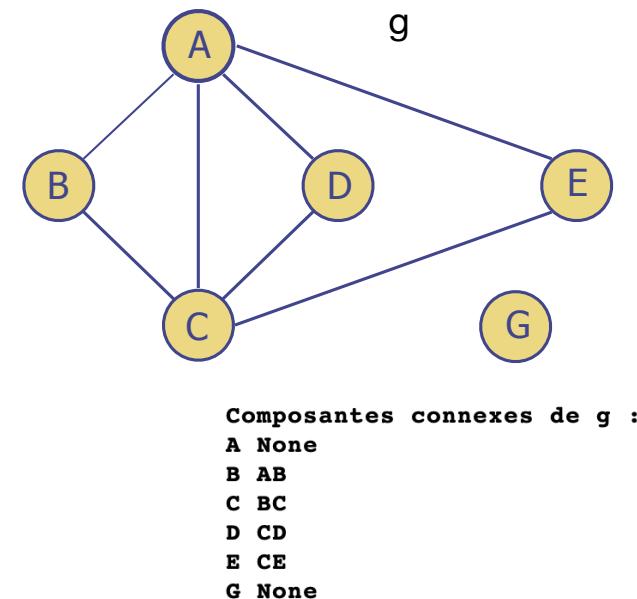
Dans le cas non-orienté, si un appel à DFS ne parvient pas à rejoindre tous les sommets, on la redémarre simplement à l'un de ces sommets :

```
#retourne une forêt de composantes connexes
#dans un dict en utilisant DFS.
def DFS_complete( g ):
    #initialisation du dict de la forêt
    forest = {}
    #pour tous les sommets du graphe
    for u in g.vertices():
        #si le sommet n'est pas dans la forêt
        if u not in forest:
            #on calcule l'arbre DFS contenant u
            forest[u] = None
            DFS( g, u, forest )
    return forest
```

Bien que cette fonction effectue plusieurs appels à DFS, le temps total est dans  $O(n + m)$ .

Cette limite dans  $O(n + m)$  s'applique également au cas d'un graphe dirigé, même si les ensembles des sommets atteignables ne sont pas nécessairement disjoints. Le dictionnaire des arêtes découvertes est passé en argument à chaque appel à DFS, alors elles seront utilisées au plus 2 fois pour un graphe non-orienté et les arêtes sortantes une seule fois pour un digraphe.

Le dictionnaire des arêtes découvertes retourné représente une forêt DFS du graphe. C'est une forêt qui contient un arbre DFS pour chaque composante connexe du graphe. Le nombre de composantes connexes peut être déterminé par les sommets dans le dictionnaire ayant des entrées *None*, car elles représentent la racine des arbres DFS de ces sommets.



# La recherche en largeur

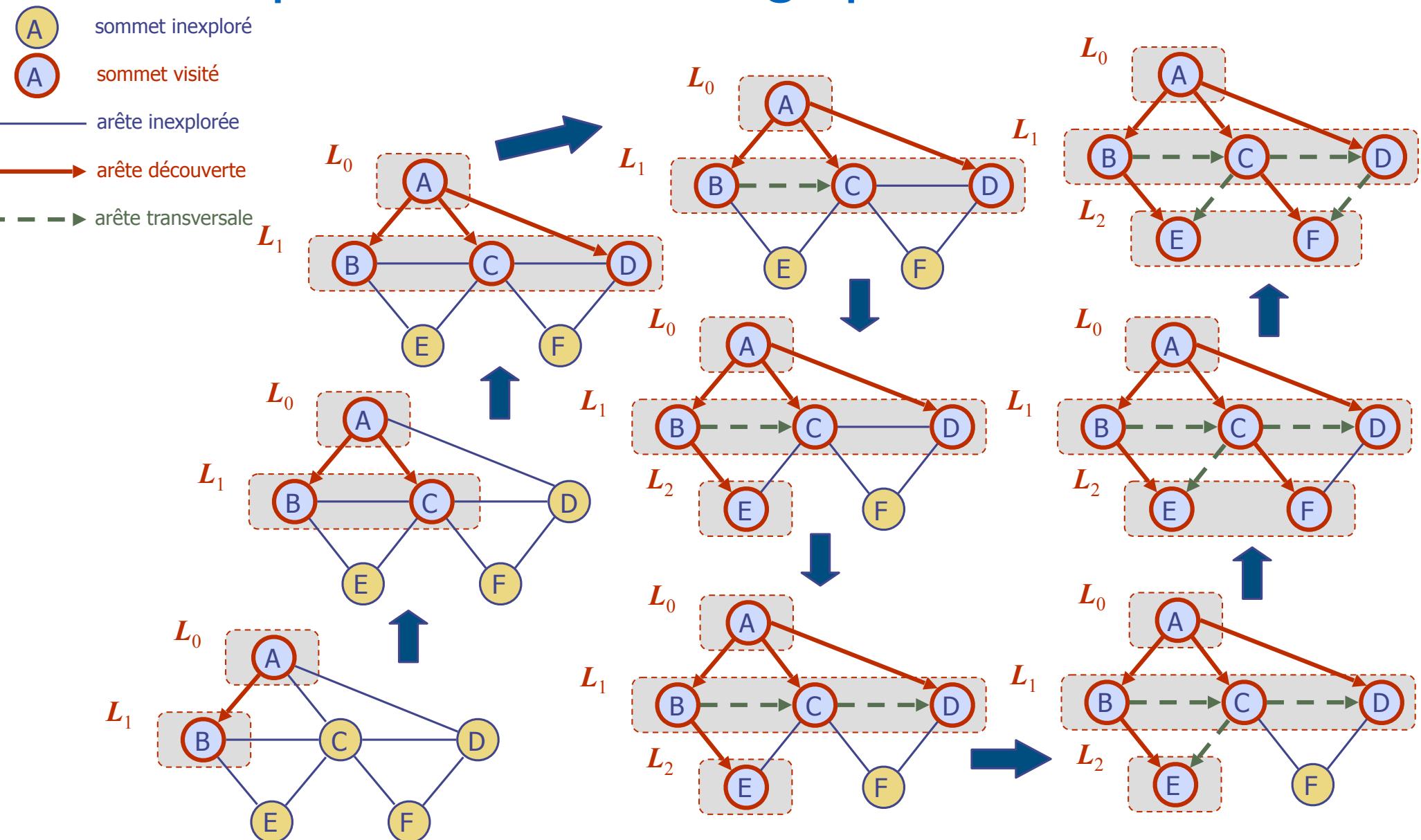
Un autre algorithme pour traverser une composante connexe d'un graphe est la recherche en largeur (BFS : Breadth-First-Search).

L'algorithme BFS est proche de l'envoi dans tous les sens de nombreux explorateurs qui traversent collectivement un graphe de manière coordonnée.

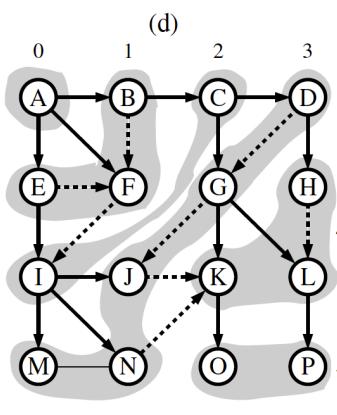
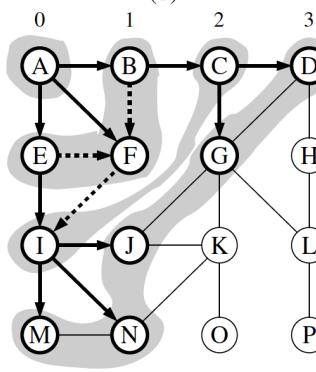
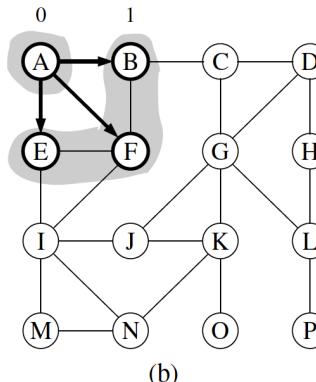
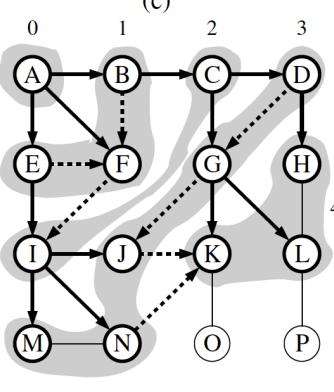
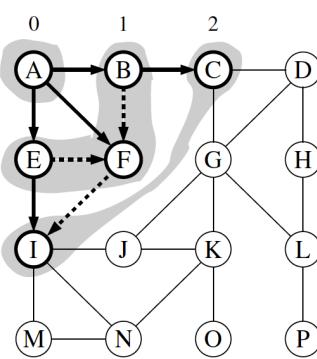
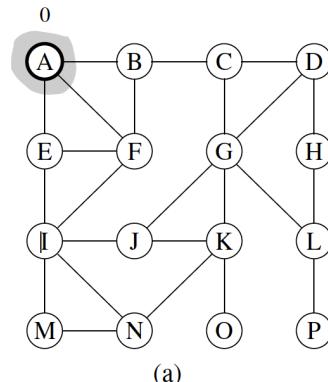
La BFS procède en rondes et subdivise les sommets en niveaux. La BFS démarre au sommet  $s$ , qui est au niveau 0. Au premier tour, on peint "visité" tous les sommets adjacents au sommet  $s$ . Ces sommets sont à un pas du début et sont placés dans le niveau 1. Dans le deuxième tour, on permet à tous les explorateurs d'aller à deux pas du sommet de départ. Ces nouveaux sommets, qui sont adjacents aux sommets de niveau 1 et qui n'ont pas été visités au niveau 1, sont placés dans le niveau 2 et marqués "visité". Ce processus se poursuit de manière similaire et se termine quand aucun nouveau sommet n'est trouvé pour créer un nouveau niveau.

```
#BFS à partir du sommet s
def BFS( g, s, discovered ):
    #on place s dans les sommets à explorer au niveau 0
    level = [s]
    #tantqu'il y des sommets à explorer
    while len( level ) > 0:
        #on initialise les sommets du prochain niveau à vide
        next_level = []
        #pour chaque sommet u dans le niveau courant
        for u in level:
            #pour chaque arête propre à u
            for e in g.incident_edges( u ):
                #prendre le sommet opposé, v
                v = e.opposite( u )
                #s'il n'a pas encore été découvert
                if v not in discovered:
                    #ajouter l'arête correspondante
                    discovered[v] = e
                    #ajouter v aux sommets du prochain niveau
                    next_level.append( v )
        #passer au niveau suivant
        level = next_level
```

# Exemple de BFS dans un graphe non-orienté



# Autre exemple de BFS dans un graphe non-orienté

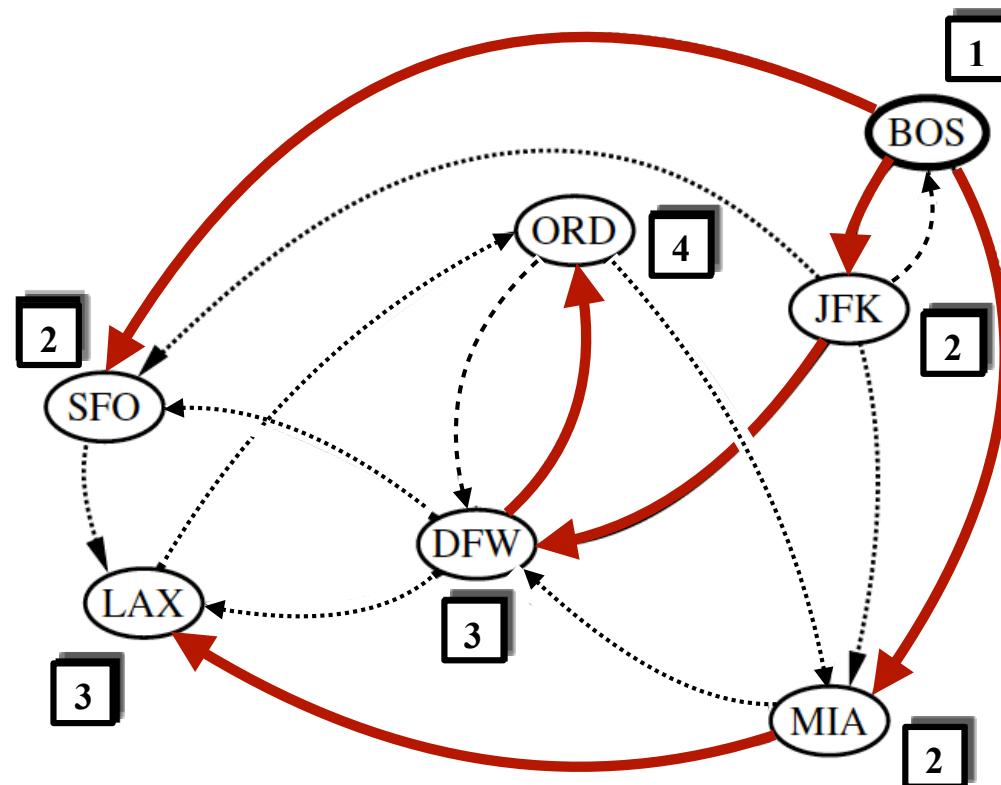


À partir du sommet A et en supposant que les sommets sont dans l'ordre alphabétique :

Les arêtes découvertes sont représentées par des lignes pleines et les arêtes croisées par des lignes pointillées :

- (a) Début.
- (b) Découvertes du niveau 1
- (c) Découvertes de niveau 2
- (d) Découvertes du niveau 3
- (e) Découvertes du niveau 4
- (f) Découvertes du niveau 5.

# Exemple de BFS dans un digraphe

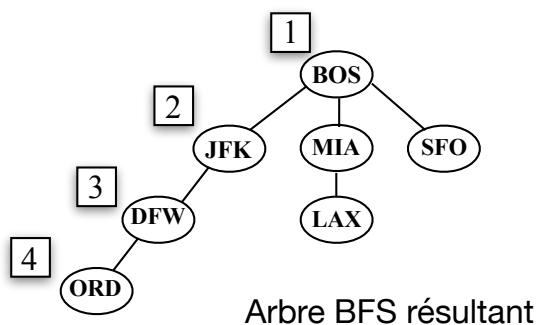


Les niveaux de la BFS sont indiqués par les chiffres dans les boîtes

L'arbre BFS est indiqué par les arêtes rouges

BFS de BOS :  
 BOS -  
 JFK BOS->JFK  
 MIA BOS->MIA  
 SFO BOS->SFO  
 DFW JFK->DFW  
 LAX MIA->LAX  
 ORD DFW->ORD  
 Chemin entre BOS et ORD :  
 BOS  
 JFK  
 DFW  
 ORD

- découverte
- -> arrière
- ....> croisée



# Propriétés et performances de BFS

Pour BFS sur un graphe non-orienté, toutes les arêtes qui ne sont pas dans l'arbre BFS sont des arêtes croisées.

Sur un graphe orienté, toutes les arêtes qui ne sont pas dans l'arbre sont soit des arêtes arrière ou croisées.

La BFS a un certain nombre de propriétés intéressantes.

Plus particulièrement, un chemin de l'arbre BFS reliant sa racine  $s$  à tout autre sommet  $v$  est garanti d'être le plus court chemin de  $s$  à  $v$  en termes de nombre d'arêtes à parcourir.

Proposition : Soit  $G$  un graphe non-orienté ou orienté sur lequel on a lancé la BFS à partir du sommet  $s$ , alors :

- Le parcours visite tous les sommets de  $G$  qui sont accessibles de  $s$
  - Pour chaque sommet  $v$  au niveau  $i$ , le chemin de l'arbre BFS entre  $s$  et  $v$  a  $i$  arêtes, et tout autre chemin de  $s$  à  $v$  a au moins  $i$  arêtes
  - Si  $(u, v)$  est une arête qui n'est pas dans l'arbre BFS, alors le niveau de  $v$  peut être au plus de 1 supérieur au niveau de  $u$ .

L'analyse de l'exécution de la BFS est similaire à celle de la DFS, soit dans  $O(n + m)$ , ou plus précisément dans  $O(n_s + m_s)$ , si  $n_s \leq n$  est le nombre de sommets accessibles du sommet  $s$ , et  $m_s \leq m$  est le nombre d'arêtes adjacentes à ces sommets.

Pour explorer le graphe entier, le processus peut être redémarré à un autre sommet, similairement à la DFS.

Aussi, le chemin entre les sommets  $u$  et  $v$  peut être reconstruit en utilisant la fonction `construct_path`(  $q$ ,  $u$ ,  $v$ ,  $discovered$  ).