Université Nice SOPHIA ANTIPOLIS

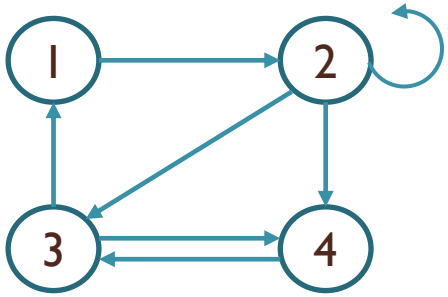
LSVIII-BIM Algorithmie et structures de données
P. Coquillard , 2015 m.à j. 2019

Théorie des graphes et opérations sur les graphes

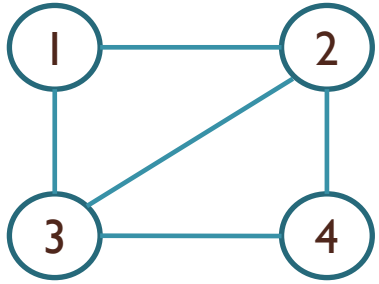
1. Eléments de la théorie
2. Représentation numérique
3. Opérations sur les graphes
    1. Algorithme BFS
    2. Algorithme DFS

I. Eléments de la théorie des graphes

Un graphe orienté G, noté G = (S, A), est un ensemble fini S de sommets (ou nœuds) et d'arcs (A) (en anglais Vertices et Edges) . A est un sous-ensemble du produit cartésien S X S . Les arcs sont appelés des arêtes et relient 2 sommets: $a = \{u,v\}$ où l'arrête *a* part de *u* et arrive en *v*. *v* est dit *adjacent* à *u*.



I. Gr. orienté



II. Gr. non orienté

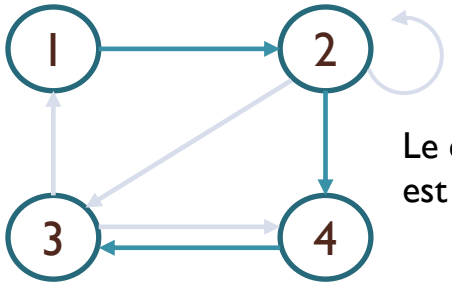
Un graphe est non orienté si A représente un ensemble de paires non orientées de sommets. Dans un tel graphe, l'arc $a = \{u,v\}$ est incident à la fois à *u* et à *v*. L'adjacence entre sommets reliés est alors symétrique.

Le degré d'adjacence d'un sommet dans un graphe non orienté est le nombre d'arrêtes qui lui sont incidentes. Dans un graphe orienté on distingue le degré sortant du degré entrant. Ex dans le Gr.I, le sommet 2 à un degré sortant de 3 et un degré entrant de 2.

Dans certains cas, les arêtes portent des poids : débits (diagramme de flot), distances entre villes (réseaux) , probabilités de transition (chaines de Markov), couts (graphes d'ordonnancement) , etc. Ces graphes sont utiles pour répondre à des problèmes précis (optimisation de la distribution, de flux, des couts de transport…)

I. Éléments de la théorie des graphes

Un chemin dans un graphe orienté (= chaine dans un graphe non orienté) de longueur K du sommet *u* vers *u'* est une séquence de sommets.



Le chemin de u_0 =1 vers *u'*=3 est de longueur K = 3

Orienté		Non orienté
arc	↔	arête
chemin	↔	chaine
circuit	↔	cycle

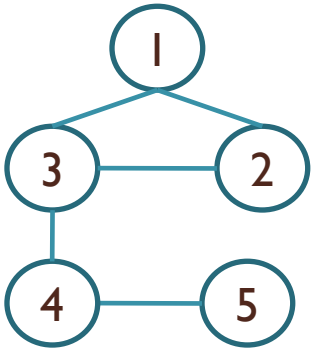
La longueur du chemin (chaine) est égal au nombre d'arcs (arêtes).
Un sommet *u'* est accessible depuis *u* s'il existe une chemin de *u* à *u'*, noté ($u \rightarrow u'$). Le chemin est élémentaire si tous ses sommets sont distincts.

Un chemin est un **circuit** si le sommet de départ est égal à celui d'arrivée.
Un circuit est élémentaire s'il ne passe pas deux fois par le même sommet.
Une **boucle** est un circuit de longueur = 1

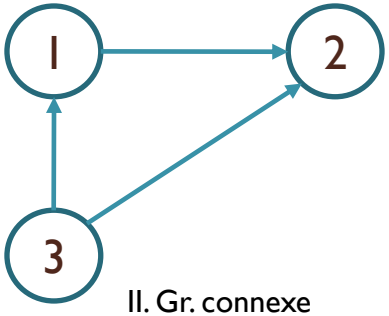
Dans un graphe NO, une chaine forme un cycle élémentaire si $K \geq 2$ et si tous les sommets sont distincts entre eux.
Un graphe sans cycle est dit acyclique.

I. Éléments de la théorie des graphes

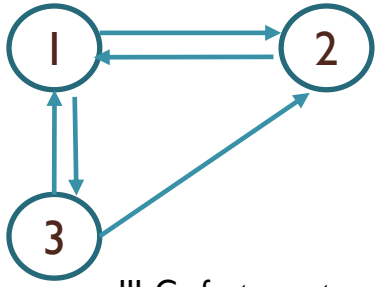
- Un **graphe** NO est **connexe** si pour tout couple de sommets il existe une chaine les reliant. On note : $\forall(x, y) \in S, \exists\ x\Re y$, où $\Re$ signifie « relation d'équivalence ». NB : il existe toujours $x\Re x$. *Un graphe connexe possède au moins n-1 arcs. S'il en possède exactement n-1, c'est un arbre.*
- Un **graphe** orienté est **fortement connexe** si chaque sommet est accessible depuis n'importe quel autre sommet.
- Un graphe connexe acyclique est un **arbre.**



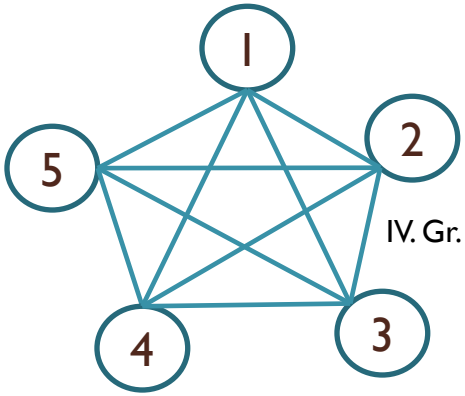
I. Gr. connexe



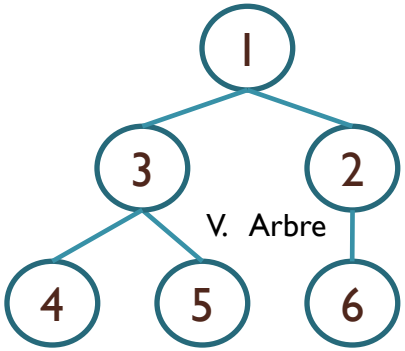
II. Gr. connexe



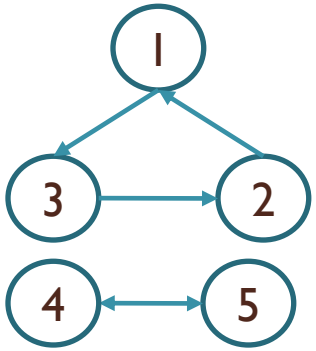
III. Gr. fortement connexe



IV. Gr. complet



V. Arbre



VI. Gr. Non connexe à 2 composantes fortement connexes

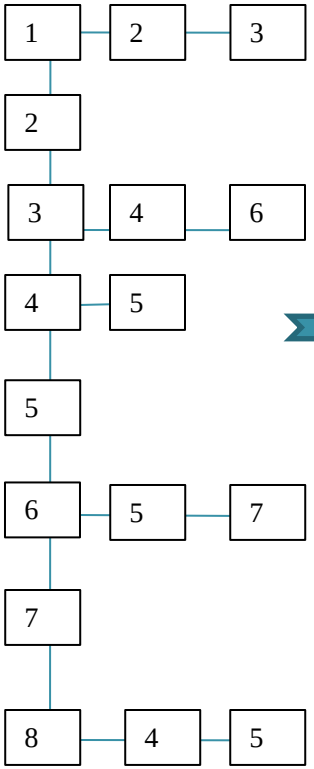
II. Graphes : représentation informatique (1)

Graphes orientés

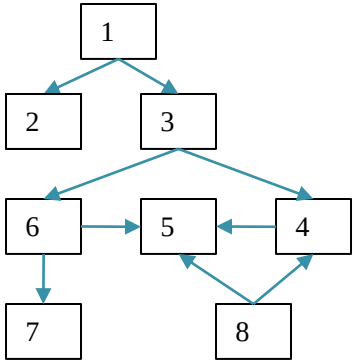
Un graphe peut aisément se représenter par une matrice

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	1	0	1	0	0
4	0	0	0	0	1	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	1	0	1	0
7	0	0	0	0	0	0	0	0
8	0	0	0	1	1	0	0	0

Ou une liste d'adjacence



Graphe *G*



Représentation informatique (Scilab)

G = list(list(2,3), list(), list(4,6), list(5), list(), list(7,5), list(), list(4,5))

La complexité mémoire d'une liste est $O(|S|+|A|)$, celle de la matrice est $O(S^2)$

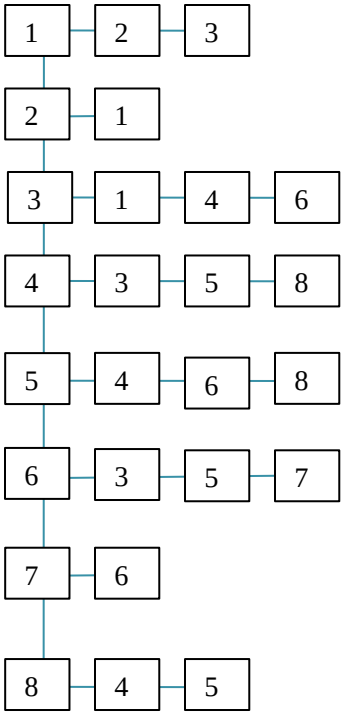
II. Graphes : représentation informatique (2)

Graphes non orientés

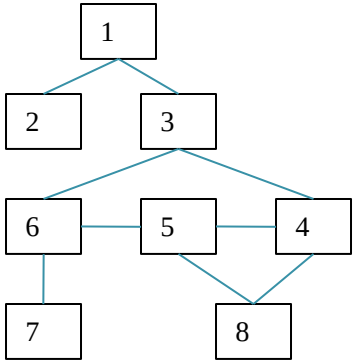
La matrice triangulaire suffit en raison de la symétrie

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	0	0	0	0	0	0
3	1	0	0	1	0	1	0	0
4	0	0	1	0	1	0	0	1
5	0	0	0	1	0	1	0	1
6	0	0	1	0	1	0	1	0
7	0	0	0	0	0	1	0	0
8	0	0	0	1	1	0	0	0

liste d'adjacence



Graphe *G*



G = list(list(2,3), list(1), list(1,4,6), list(3,5,8), list(4, 6, 8), list(3,5,7), list(6), list(4,5))

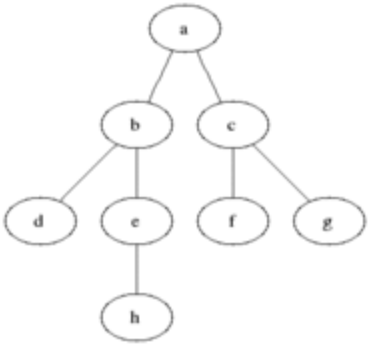
Mais on peut aussi utiliser la liste précédente en considérant que : list(a,b) ⇔ list(b,a)

Les listes sont préférées pour les graphes peu denses (|A| □ |S|, donc |A| + |S| □ |S|²)

III. Opérations sur les graphes. BFS.

BFS : Pseudo-code très simplifié d'exploration en largeur à partir du sommet *s*

```
ParcoursLargeur(Sommet s):
{
f = CreerFile();
f.enfiler(s);              // on stocke et marque s
marquer(s);
TANT-QUE  (NON f.vide() ) FAIRE
     s = f.defiler();      // on enlève s ; aux tours suivant on enlèvera les voisins un à un…
     afficher(s);
     POUR TOUT voisin de s FAIRE
          SI (voisin non marqué )
                    FAIRE f.enfiler(voisin);     // On stocke les voisins dans f
                    marquer(voisin);
          FIN SI
     FIN POUR TOUT
FIN TANT QUE
}
```



Donc à la fin de cet algorithme, tous les voisins de s on été explorés et marqués.

III. Opérations sur les graphes

BFS, plus détaillé

Entrée : un graphe ***G***

Sortie : marquage des arcs,
partition des nœuds de ***G***

I. Algorithme *BFS(G)*

Pour tout ***u*** ∈ ***G.noeuds***()
marquer(u, INEXPLORE)

Pour tout ***e*** ∈ ***G.arcs***()
marquer(e, INEXPLORE)

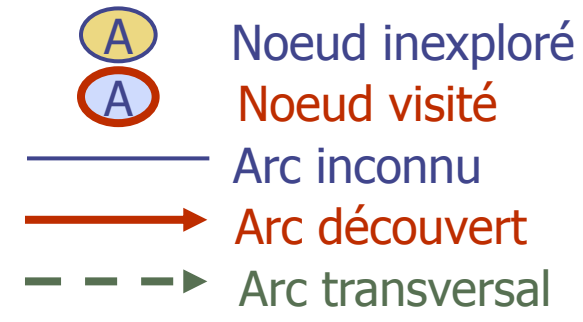
Pour tout ***v*** ∈ ***G.noeuds***()
si ***getMarque*** (***v***) == ***INEXPLORE***
alors ***BFS(G, v)*** *// voir page suivante*

III. Opérations sur les graphes

```
II. Algorithme BFS(G, s){
L0 ← nouvelle liste vide
L0 ← insérer(s)
marquer(s, VISITÉ)
i ← 0
    TantQue Li n'est pas vide faire
            Li+1 ← nouvelle liste vide
            PourTout v ∈ Li.elements()
                    PourTout e ∈ G.arcIncident(v)
                            If getMarquage(e) = INEXPLORÉ
                                    w ← opposé(v,e)
                                    If getMarquage(w) = INEXPLORÉ
                                            setMarquage(e, DECOUVERT)
                                            setMarquage(w, VISITÉ)
                                            Li+1.inserer(w)
                                    sinon
                                            setMarquage(e, TRANSVERSAL)
                                    FinIf
                            FinIf
                    FinPourTout
            FinPourTout
            i ← i +1
    FinTantQue
    }
```

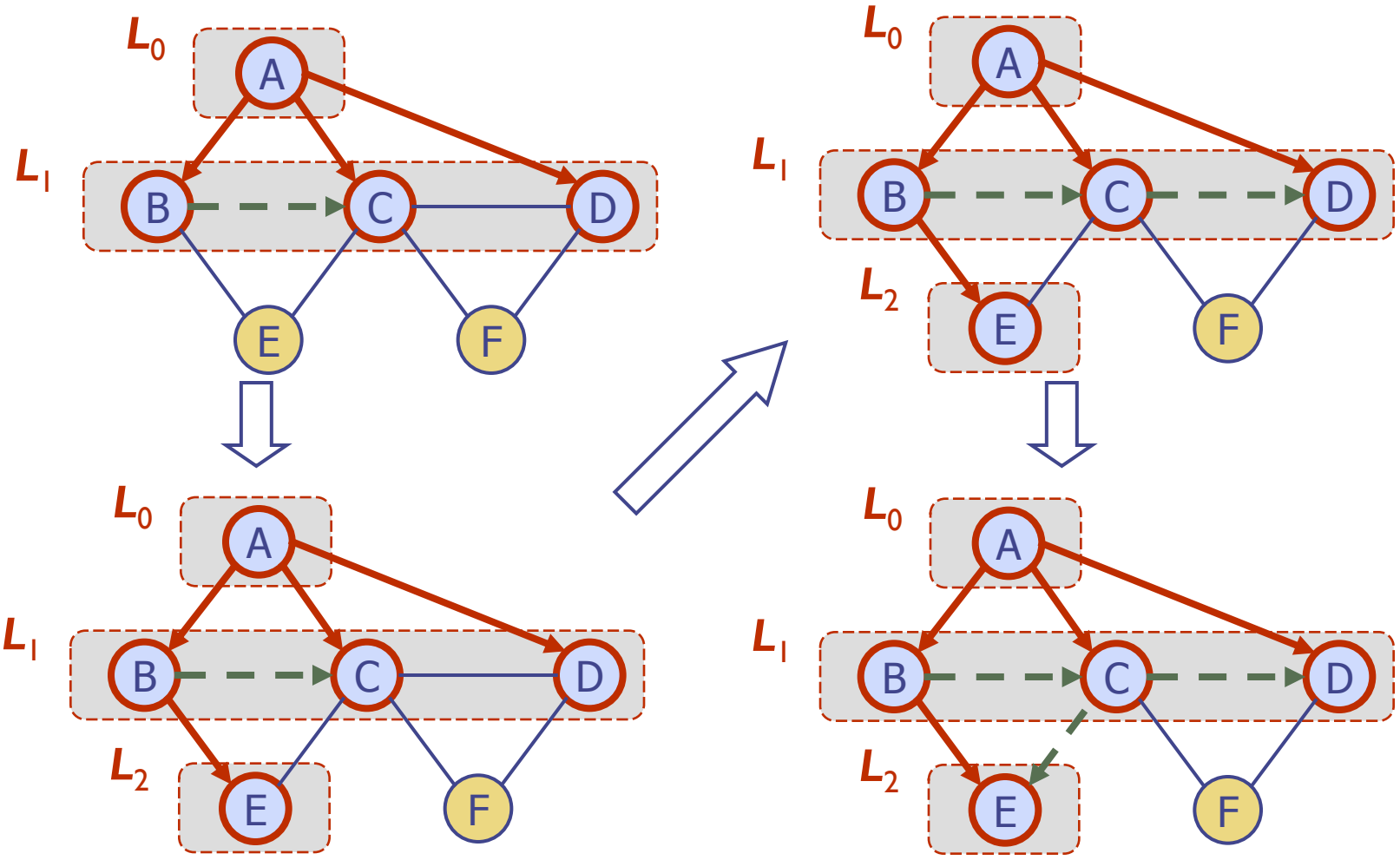
III. Opérations sur les graphes

Exemple



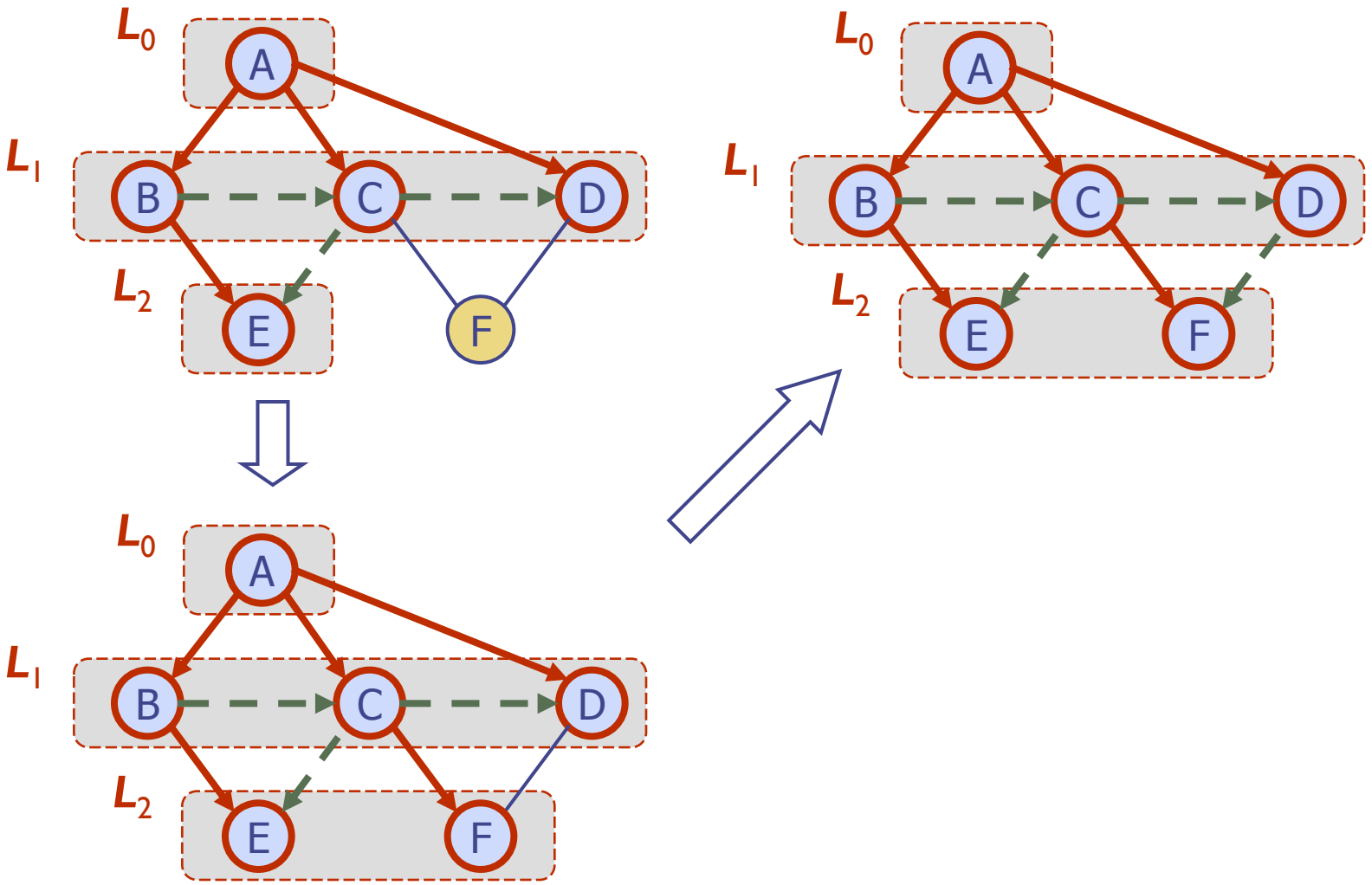
III. Opérations sur les graphes

Exemple (suite)



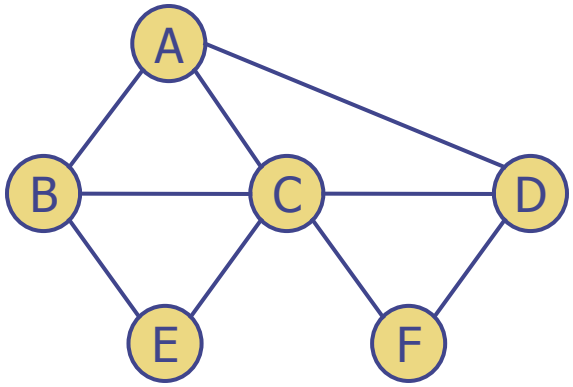
III. Opérations sur les graphes

Exemple (fin)



III. Opérations sur les graphes

Propriétés



Pr. 1

***BFS*(*G, s*)** visite tous les noeuds et arcs de G_s

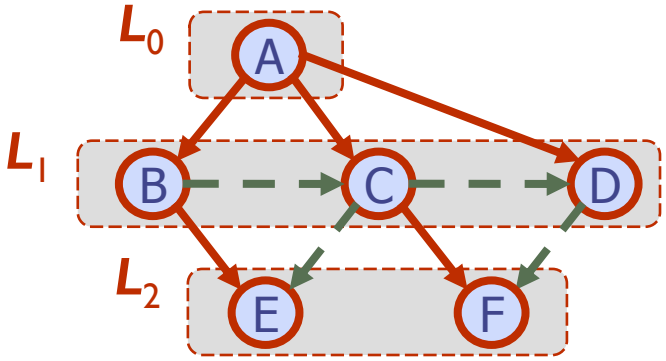
Pr. 2

Si tous les arcs ont le même poids, la découverte des arcs labellisés par ***BFS*(*G, s*)** constitue **un arbre couvrant minimal** T_s de G_s enraciné sur A dans l'exemple ci-contre.

Pr. 3

Pour chaque noeud ***v*** dans L_i

- Le chemin dans T_s allant de ***s* à *v*** possède ***i*** arcs
- Chaque chemin allant de ***s*** à ***v*** dans G_s possède au moins ***i*** arcs.



Pr. 4

- Complexité enfiler et défiler = $O(1)$, « S » fois : $O(S)$
- Examiner tous les arcs : $O(A)$
- Complexité = $O(|S| + |A|)$

III. Opérations sur les graphes. DFS.

DFS : Pseudo-code très simplifié d'exploration en profondeur à partir du sommet *s*

En entrée : un graphe G. dont aucun sommet n'est marqué, un sommet s
En sortie : une partition des arcs du graphe : ceux qui construisent l'arbre des sommets accessibles depuis s, et ceux qui n'en font pas partie.

```
1.  Parcours_profondeur DFS(G, s)
2.  {
3.      Marquer(s)*
4.      Pour tous éléments  fils de voisins(s) faire
5.          Si NonMarqué (fils)**  alors  DFS(G, fils)
6.          FinSi
7.      FinPour
8.  }
```

* *Mettre sa couleur à « noir »*
* * *C'est-à-dire si sa couleur est blanche*

Attention : cet algorithme contient un auto-appel (ligne 5).
C'est un cas de récursivité (appel récursif).

III. Opérations sur les graphes. DFS.

DFS détaillé

En entrée un graphe G. En sortie un arbre

```
Algorithme DFS(G){

Pour tous les sommets u ∈ G faire                    // initialisation
    couleur(u) = blanc
    pere(u)    = NIL
FinPour

temps = 0

Pour tous les sommets u ∈ G faire
    Si couleur(u) == blanc alors
        DFS-VISIT(u)                                 // voir en page suivante
    FinSi
FinPour
}
```

III. Opérations sur les graphes. DFS.

Algorithme DFS-VISIT(*u*){
couleur(u) = gris
temps = temps +1
decouverte[u] = temps

Pour ***tous v adjacent à u*** **faire** *// = appartenant à la liste d'adjacence de u*
Si ***couleur(v) = blanc*** **alors**
père(v) = u
DFS-VISIT(v) *//*  *récursivité !*
FinSi
FinPour

Couleur(u) = noir *// on a exploré tous les descendants*
temps = temps +1
final[u] = temps *// on enregistre le temps total*
}

On utilise :
decouverte[*u*] = instant t de découverte de *u*
final[*u*] = instant t_i de fin d'exploration de *u* = *temps de retour en u*
Avec ces données on construit l'arbre d'exploration du graphe. Il résume tous les sommets accessibles depuis u (et seulement ceux-ci).

III. DFS en Python


```python
class DFS:
    def __init__(self, node,edges):
        self.node = node
        self.edges = edges
        self.color=['W' for i in range(0,node)] # W for White
        self.graph =color=[[False for i in range(0,node)] for j in range(0,node)]
        self.parent =[-1 for u in range(0,node)]

        # Start DFS
        self.construct_graph()
        self.dfs_traversal()

    def construct_graph(self):
        for u,v in self.edges:
            self.graph[u][v], self.graph[v][u] = True, True

    def dfs_visit(self, u):
        self.color[u]='G' # G for Gray
        for i in range(0, self.node):
            if self.graph[u][i]==True and self.color[i]=='W':
                self.parent[i]=u
                self.dfs_visit(i)
        self.color[u]='B' # B for black

    def dfs_traversal(self):
        for i in range(0,self.node):
            if self.color[i]=='W': # W for white
                self.dfs_visit(i)

    def print_path(self, source, destination):
        if destination==source:
            print destination,
        elif self.parent[destination] == -1:
            print "No Path"
        else:
            self.print_path(source, self.parent[destination])
            print "-> ",destination,

node = 8 # 8 nodes from 0 to 7
edges =[(0,1),(0,3),(1,2),(1,5),(2,7),(3,4),(3,6),(4,5),(5,7)] # bi-directional edge

dfs = DFS(node, edges)
dfs.print_path(0, 7)
print ""
dfs.print_path(2, 5)
print ""
dfs.print_path(0, 4)
```

code par Abu Zahed Jony, 2012