



ASD 2

Algorithmique et Structure de Données

Marc Gaëtano

`gaetano@polytech.unice.fr`

Polytech'Nice-Sophia

Département Sciences Informatiques

930 route des Colles

06903 Sophia Antipolis - France





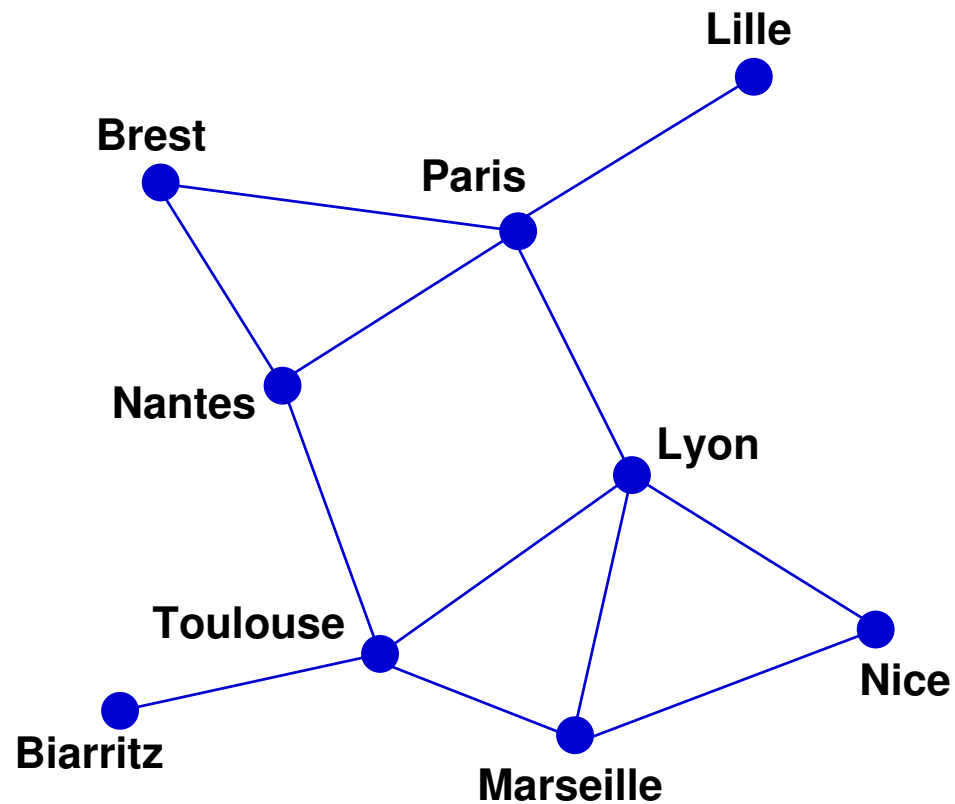
Introduction aux Graphes

- Définitions, propriétés et représentations
- Parcours en profondeur et en largeur
- Tri topologique
- Connexité
- Arbres couvrants



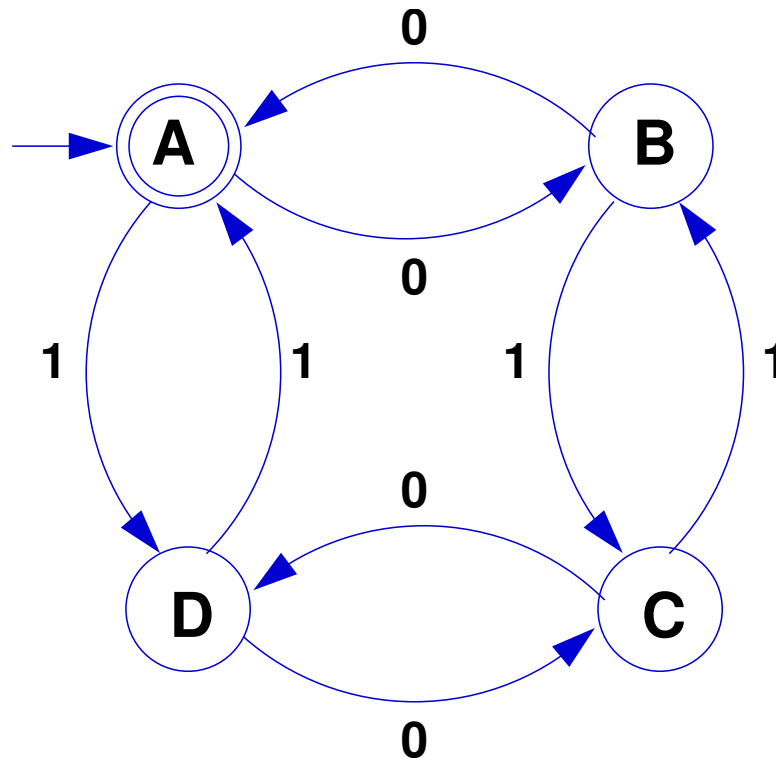
Graphe : exemple

Réseaux



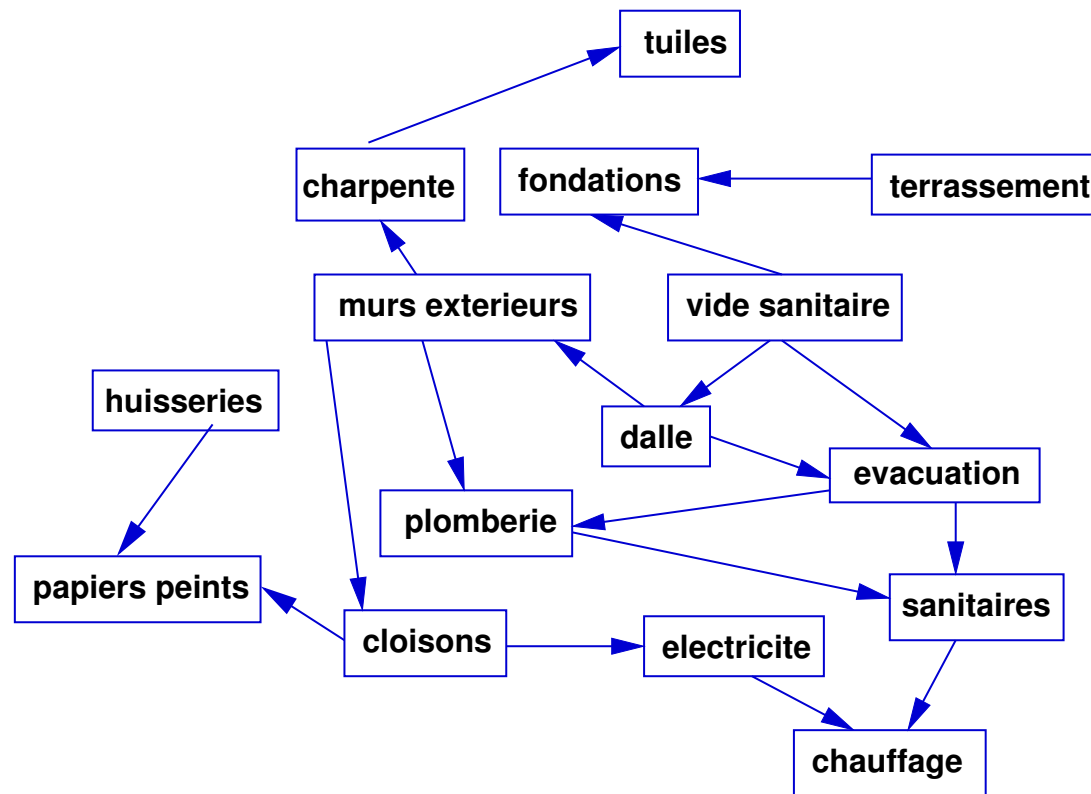
Graphe : exemple

Processus



Graphe : exemple

Ordonnancement



Graphe : définition

$G = (S, A)$ est un graphe

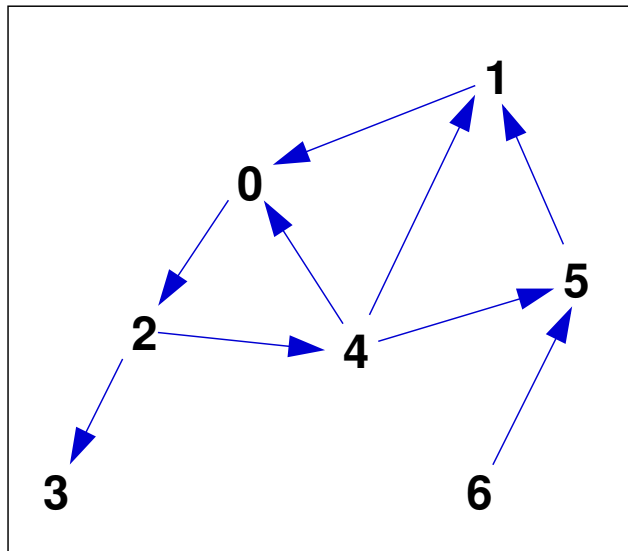
- S est un ensemble fini (les sommets)
- $A \subseteq S \times S$ est une relation binaire sur S

Terminologie

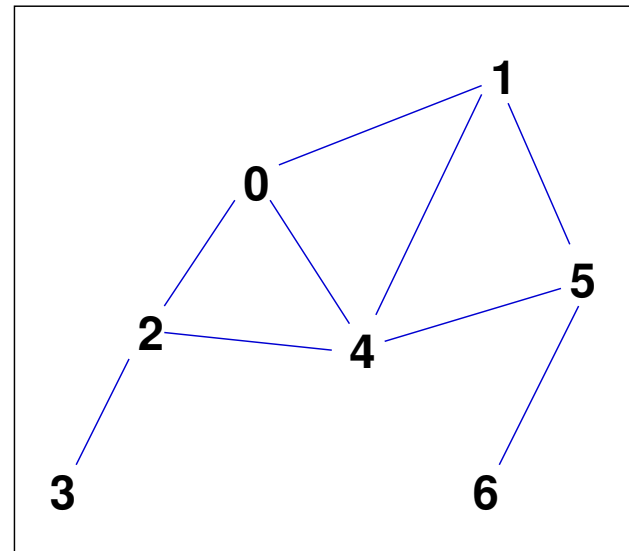
- si A est symétrique, G est *non orienté* et $a = (u, v) \in A$ est une *arête*, sinon G est *orienté* et a est un *arc*
- $|S|$ est l'*ordre* de G

Graphe : exemple

Graphe orienté et non orienté



G1



G2

Graphe : définition

Adjacence

Soit $(u, v) \in A$ un arc ou une arête

- (u, v) part de u et arrive à v
- v est adjacent à u
- (u, v) est incident à u et à v

Si G est *non orienté*, alors $(u, v) \in A \implies u \neq v$, sinon $(u, u) \in A \implies (u, u)$ est une *boucle*

Graphe : définition



Degré

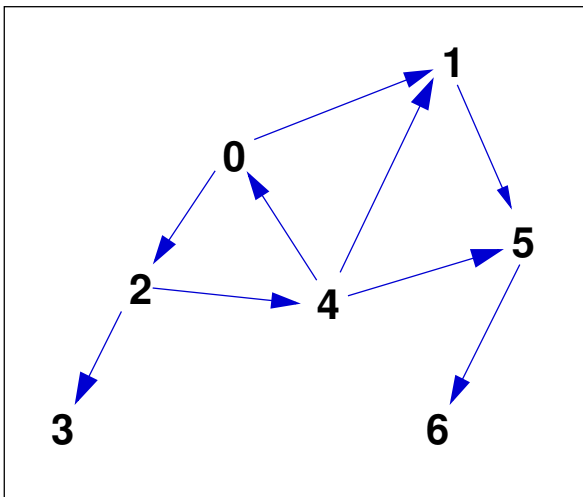
Soit $u \in S$ un sommet de G et $d(u)$, le *degré* de u

- $d(u)$ est le nombre d'arcs ou d'arêtes incidents à u
- si G est orienté :
 - $d_e(u)$ (le degré *entrant*) est le nombre d'arcs qui arrivent à u
 - $d_s(u)$ (le degré *sortant*) est le nombre d'arcs qui partent de u
 - $d(u) = d_e(u) + d_s(u)$

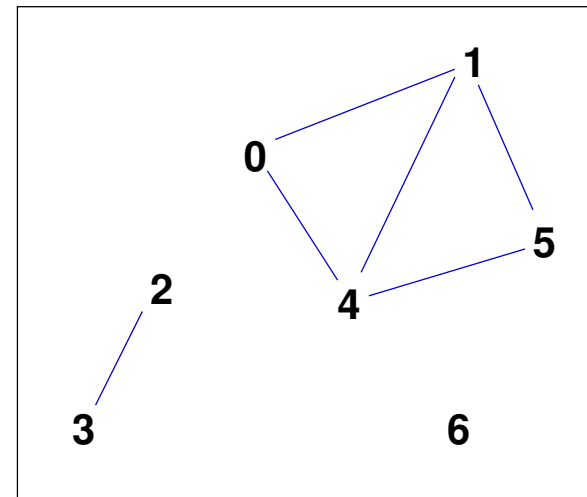


Graphe : exemple

Degrés des sommets



G1



G2

- dans **G1** : $d_e(4) = 1$, $d_s(4) = 3$ et $d(4) = 4$
- dans **G2** : $d(4) = 3$ et $d(6) = 0$

Graphe : définition

Chemin

Une suite de sommets $\langle u_0, u_1, \dots, u_k \rangle$ tels que $(u_i, u_{i+1}) \in A, \forall i \in [0..k[$ est un *chemin* de u_0 à u_k de longueur k

- si tous les sommets du chemin sont distincts, le chemin est *élémentaire*
- si $\exists$ un chemin de u à v dans G , on dit que v est *accessible* par u dans G , soit $u \rightarrow_G v$

Graphe : définition



Cycle et circuit

Un chemin $\langle u_0, u_1, \dots, u_k \rangle$ est un :

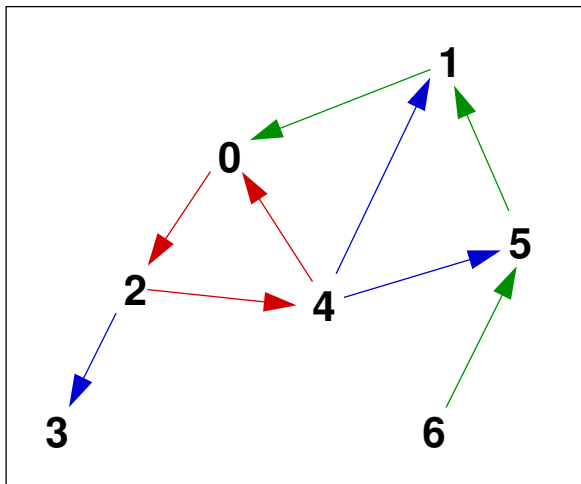
- *circuit* dans un graphe G orienté, si $u_0 = u_k$ et si le chemin contient au moins un arc (une boucle)
- *cycle* dans un graphe G non orienté, si $u_0 = u_k$ et si le chemin contient au moins trois arêtes

Un graphe sans cycle (circuit) est *acyclique*

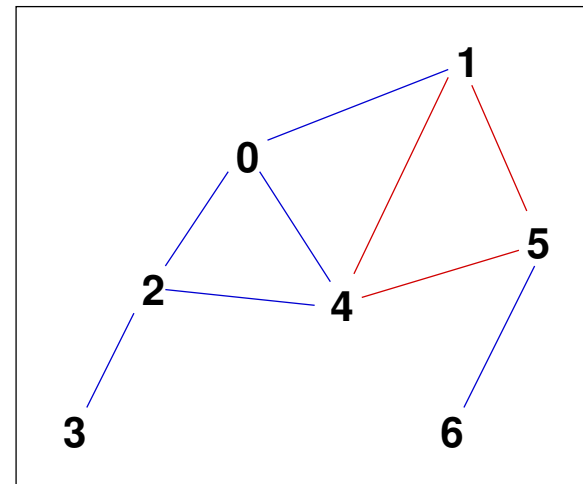


Graphe : exemple

Chemin, circuit et cycle



G1



G2

- dans **G1** : $\langle 0, 2, 4, 0 \rangle$ et $\langle 1, 0, 2, 4, 5, 1 \rangle$ sont des circuits
- dans **G2** : $\langle 0, 2, 4, 0 \rangle$ et $\langle 1, 4, 5, 1 \rangle$ sont des cycles

Graphe : définition



Connexité

La relation $\leftrightarrow_G$ définie sur S l'ensemble des sommets d'un graphe G par $u \leftrightarrow_G v \iff u \rightarrow_G v \text{ et } v \rightarrow_G u$ est une relation d'équivalence, et

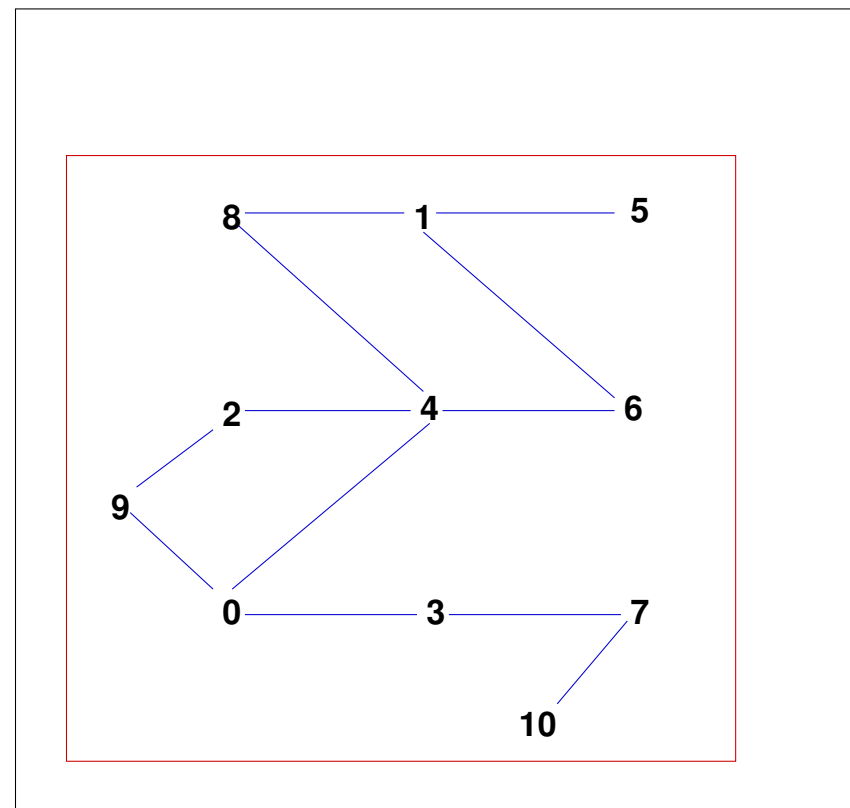
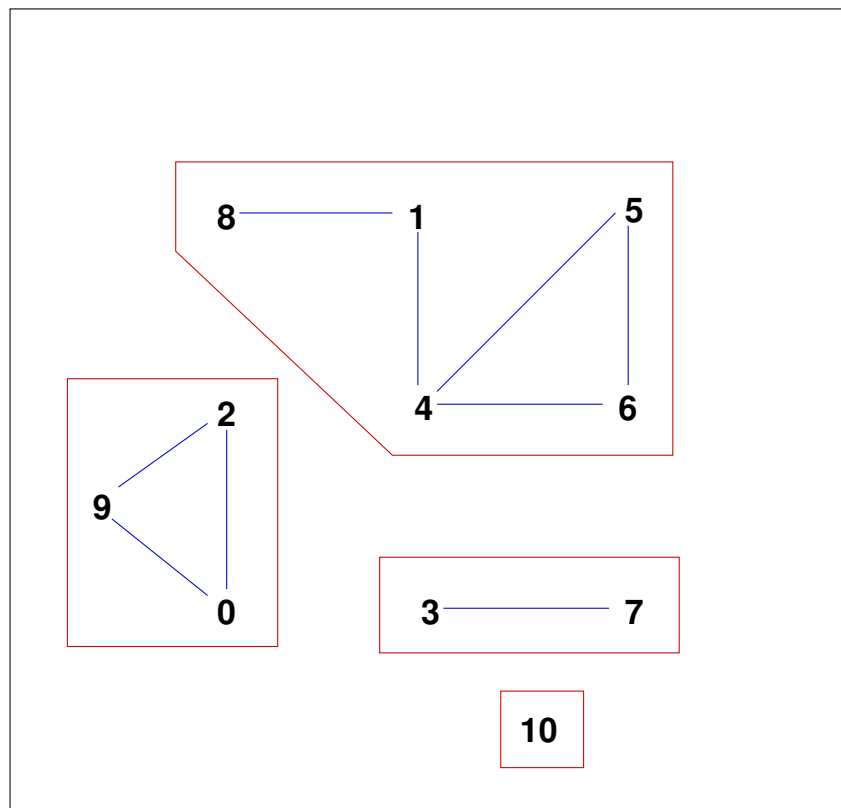
- les classes d'équivalence de $\leftrightarrow_G$ dans S sont les composantes (fortement) connexes de G
- G est (fortement) connexe si et seulement si $\leftrightarrow_G$ n'a qu'une seule classe d'équivalence dans S

L'adverbe "*fortement*" s'emploie quand G est orienté



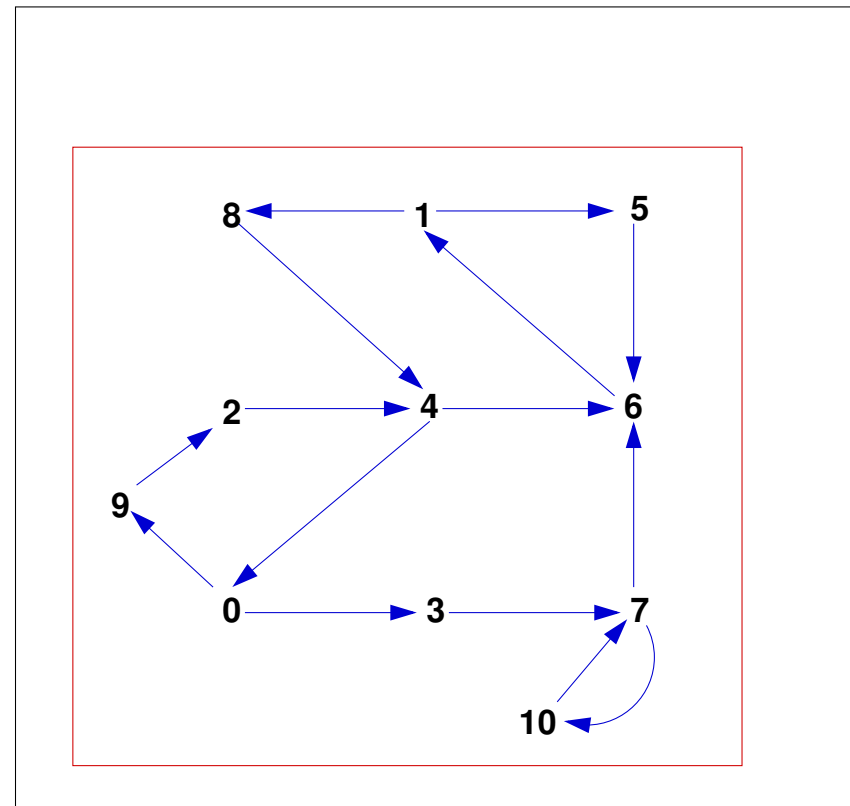
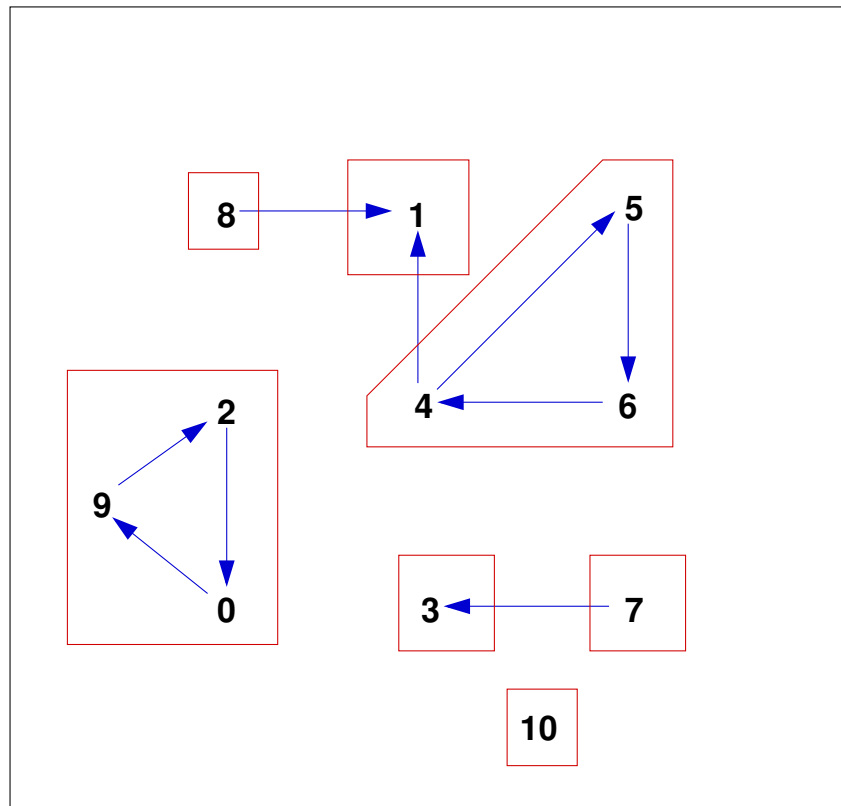
Graphe : exemple

Connexité des graphes non orientés



Graphe : exemple

Connexité des graphes orientés



Graphe : définition

Sous-graphe

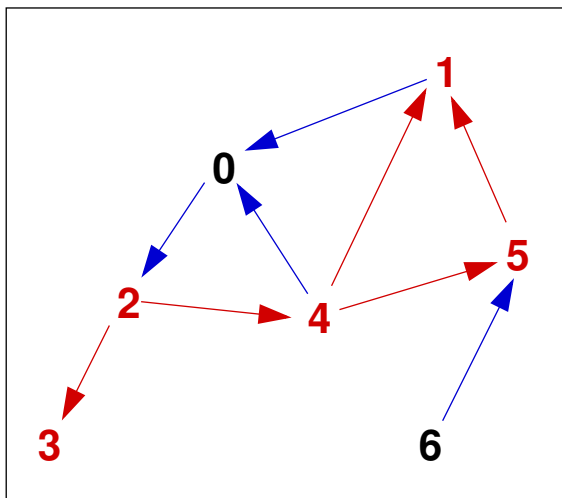
$G' = (S', A')$ est un sous-graphe du graphe $G = (S, A)$ si et seulement si $S' \subseteq S$ et $A' \subseteq A$. Si $S' = S$, on dit que G' est *couvrant*

Soit $G = (S, A)$ un graphe et $S' \subseteq S$. Le sous-graphe $G' = (S', A')$ de G avec $A' = \{(u, v) \in A \mid u, v \in S'\}$ est appelé le sous-graphe de G *engendré* par S'

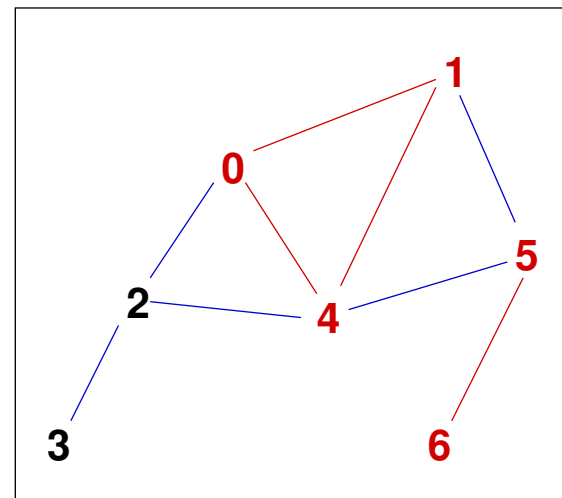
De même, si $A' \subseteq A$, le sous-graphe $G' = (S', A')$ de G avec $S' = \{u \in S \mid \exists v \in S, (u, v) \text{ ou } (v, u) \in A'\}$ est appelé le sous-graphe de G *engendré* par A'

Graphe : exemple

Sous-graphes



G1



G2

En **rouge**, le sous-graphe de **G1** engendré par les sommets $\{1, 2, 3, 4, 5\}$ et le sous-graphe de **G2** engendré par les arêtes $\{(0, 1), (0, 4), (1, 4), (5, 6)\}$

Graphe : définition

Isomorphisme

Deux graphes de même nature $G = (S, A)$ et $G' = (S', A')$ sont *isomorphes* si et seulement si $\exists b$ bijective, $b : S \longrightarrow S'$ telle que

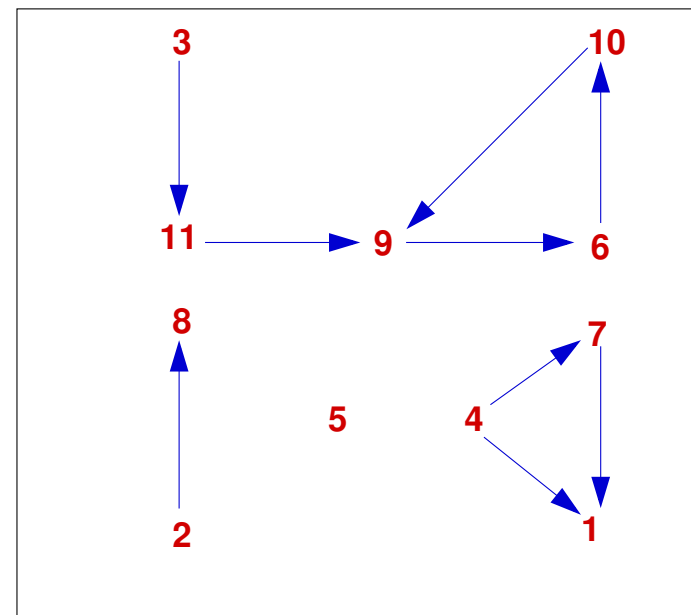
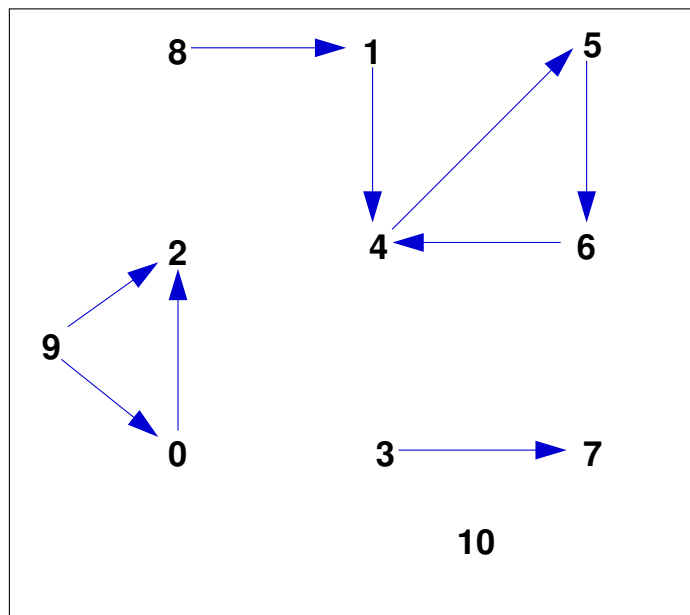
$$\forall u, v \in S, (u, v) \in A \iff (b(u), b(v)) \in A'$$

Autrement dit, G est isomorphe à G' si je peux *renommer* les sommets de G par les sommets de G'

$\implies$ les sommets d'un graphe G d'ordre n sont toujours notés $0, 1, \dots, n - 1$

Graphe : exemple

Graphes isomorphes



La bijection b : $b(0)=7$, $b(1)=11$, $b(2)=1$, $b(3)=2$, $b(4)=9$,
 $b(5)=6$, $b(6)=10$, $b(7)=8$, $b(8)=3$, $b(9)=4$ et $b(10)=5$

Graphe : définition



Arbre

Un graphe non orienté connexe et acyclique est un *arbre*

Arbre enraciné

Etant donné un arbre, le choix d'un sommet particulier appelé *racine* enrachine l'arbre, et détermine récursivement les sous-arbres enracinés

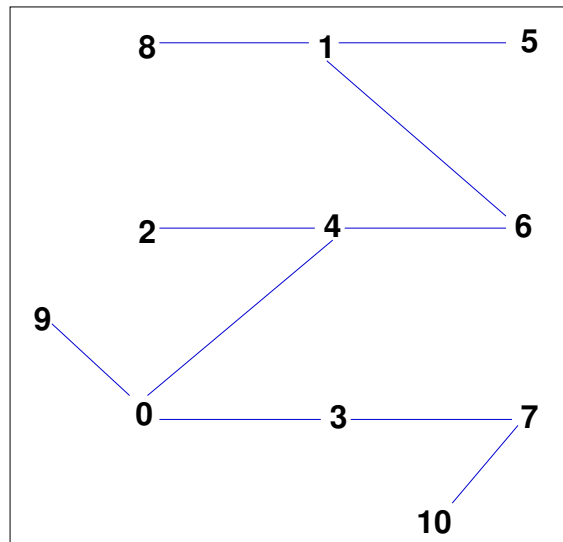
Forêt

Un graphe non orienté acyclique est une *forêt* (un ensemble d'arbres)

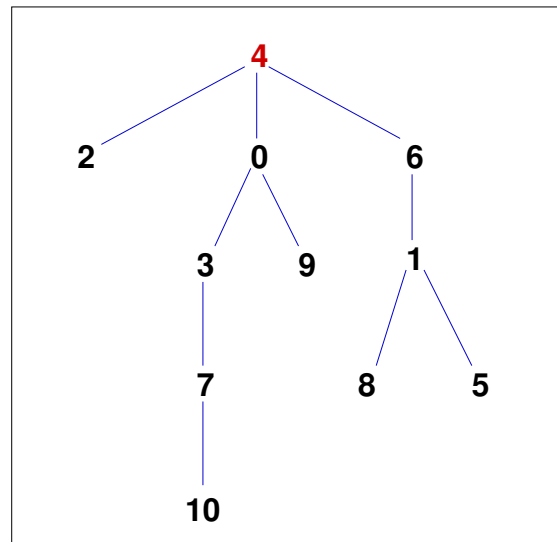


Graphe : exemple

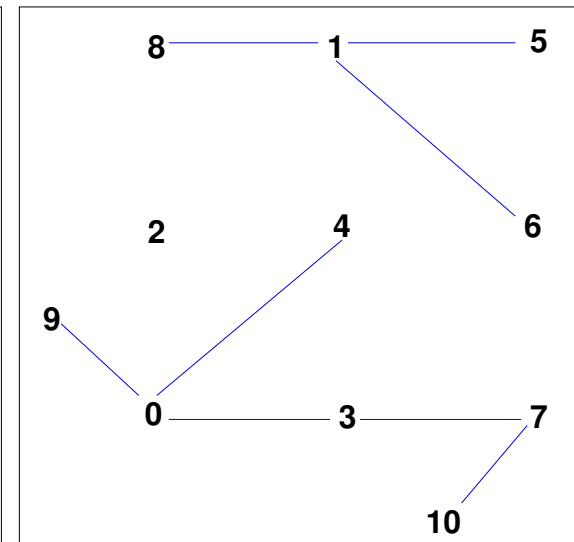
Arbres et forêt



A



E



F

A est un arbre, **E** l'arbre **A** enraciné avec **4** comme racine et **F** une forêt à 3 arbres

Graphe : représentation

Soit $G = (S, A)$ un graphe d'ordre n

Matrice d'adjacence

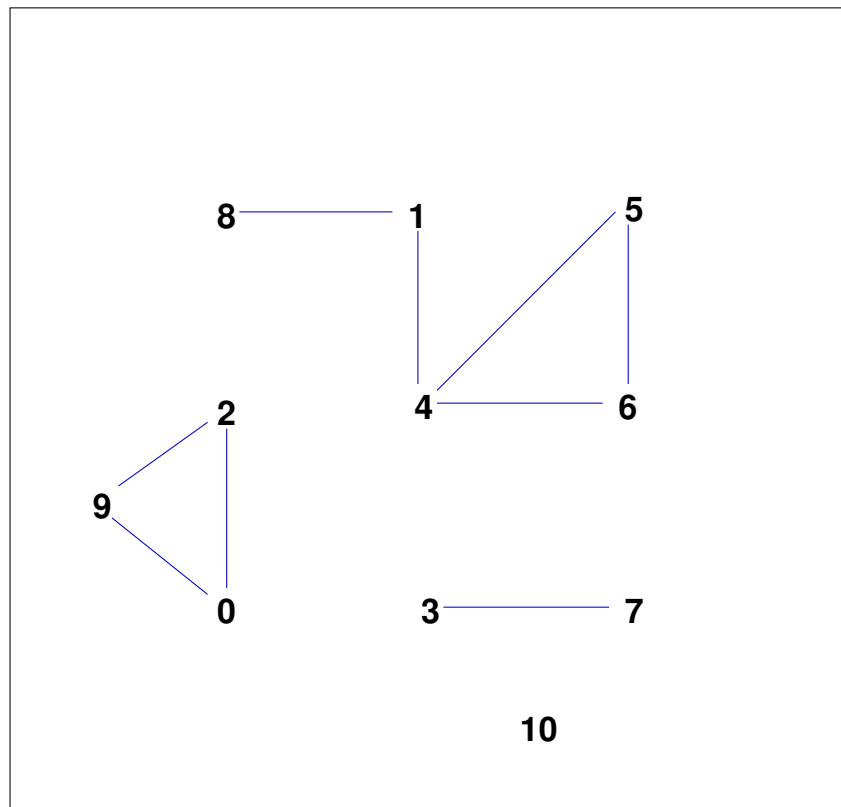
$S = \{0, 1, \dots, n-1\}$ et A est une matrice $n \times n$ de booléens telle que $\forall u, v \in S, (u, v) \in A \iff A[u][v] = \text{vrai}$

Listes d'adjacence

$S = \{0, 1, \dots, n-1\}$ et Adj est une fonction de S vers $\langle S \rangle$ (les listes finies d'éléments de S) telle que pour tout $u \in S$, $\text{Adj}(u) = \langle v_0, v_1, \dots, v_{k-1} \rangle$ la liste des k sommets adjacents à u

Graphe : représentation

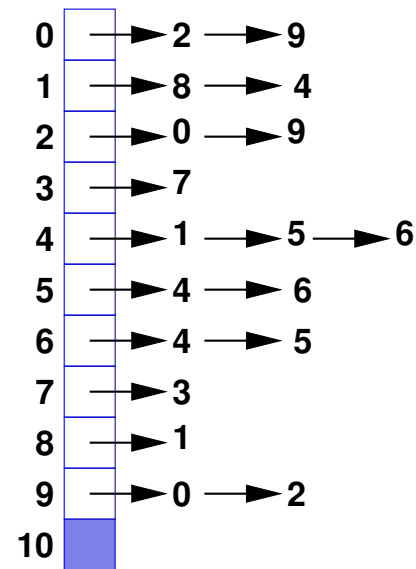
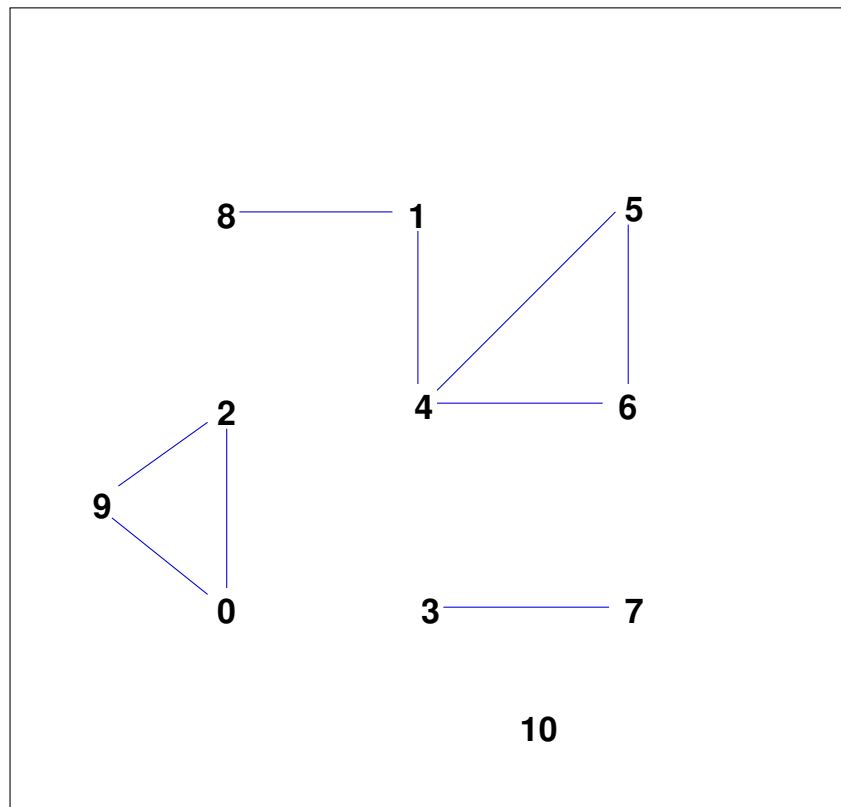
Graphe non orienté et matrice d'adjacence



	0	1	2	3	4	5	6	7	8	9	10
0	■		■							■	
1		■			■				■		
2	■		■							■	
3				■				■			
4		■			■	■	■				
5					■	■	■				
6					■	■	■				
7				■				■			
8		■						■			
9	■		■							■	
10											■

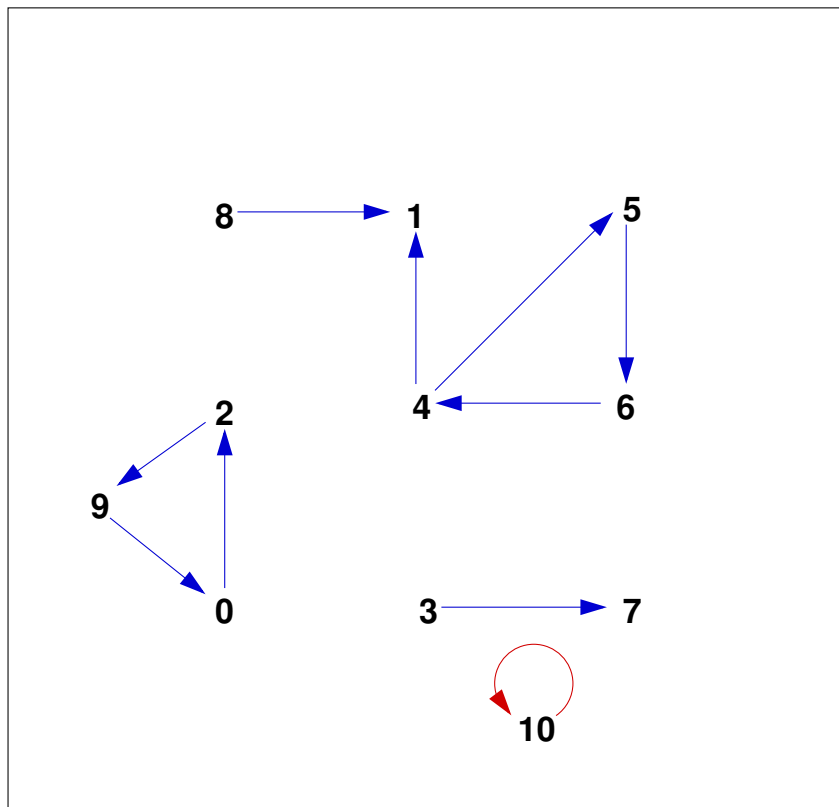
Graphe : représentation

Graphe non orienté et liste d'adjacence



Graphe : représentation

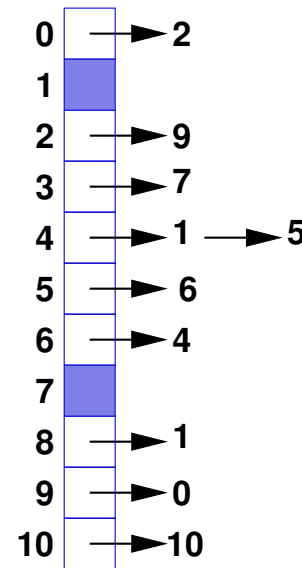
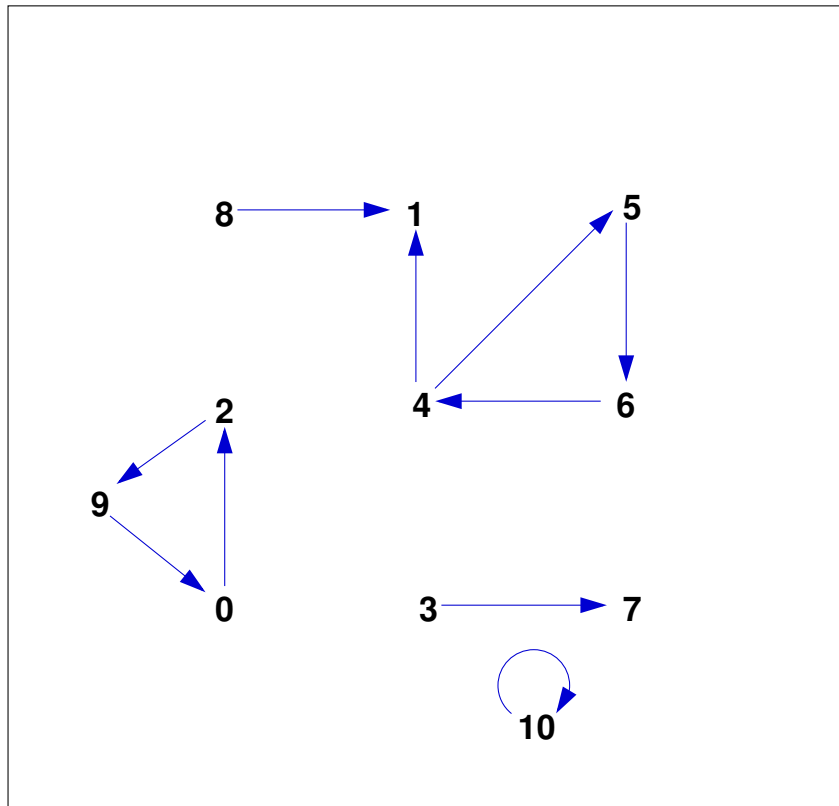
Graphe orienté et matrice d'adjacence



	0	1	2	3	4	5	6	7	8	9	10
0			■								
1											
2										■	
3								■			
4	■				■						
5						■					
6					■						
7											
8		■									
9	■										
10											■

Graphe : représentation

Graphe orienté et liste d'adjacence



Graphe : parcours



Parcours en profondeur (PP)

- exploration des sommets "*en profondeur*" d'abord
- explore tout le graphe
- base de nombreux algorithmes sur les graphes

Parcours en largeur (PL)

- exploration des sommets "*en largeur*" d'abord
- n'explore qu'une partie du graphe
- moins puissant que le PP



PP : présentation



Deux parties distinctes

Le parcours des sommets

- on cherche les sommets non explorés
- parcours linéaire itératif de la liste des sommets

L'exploration du graphe

- on cherche les sommets non explorés accessibles à partir d'un sommet donné
- parcours récursif vers les sommets adjacents



PP : attributs



Attributs

- le **père** : $p[u]$, le sommet à partir duquel le sommet u à été découvert
- la **couleur** : $c[u]$, un attribut dynamique indiquant l'état du sommet u durant le parcours
- le **temps** : le temps global (un entier) qui s'incrémente à chaque étape
- les **dates** : $d[u]$ et $f[u]$, les dates de début et de fin de découverte (ou traitement) du sommet u



PP : relations



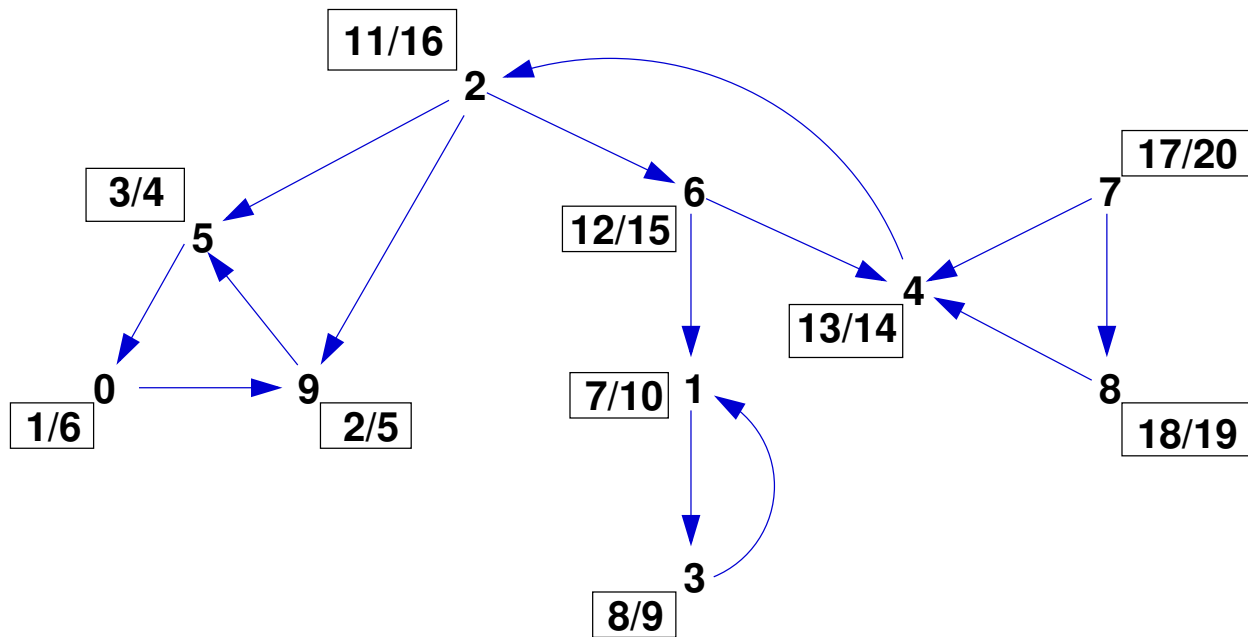
Relations entre les attributs

- $\forall u \in S, 1 \leq d[u] < f[u] \leq 2|S|$
- la couleur $c[u]$ du sommet u est
 - *blanc* durant l'intervalle de temps $[0, d[u][$
 - *gris* durant l'intervalle de temps $[d[u], f[u][$
 - *noir* durant l'intervalle de temps $[f[u], \infty[$
- au début du parcours, tous les sommets sont *blancs*
- à la fin du parcours, tous les sommets sont *noirs*



PP : exemple

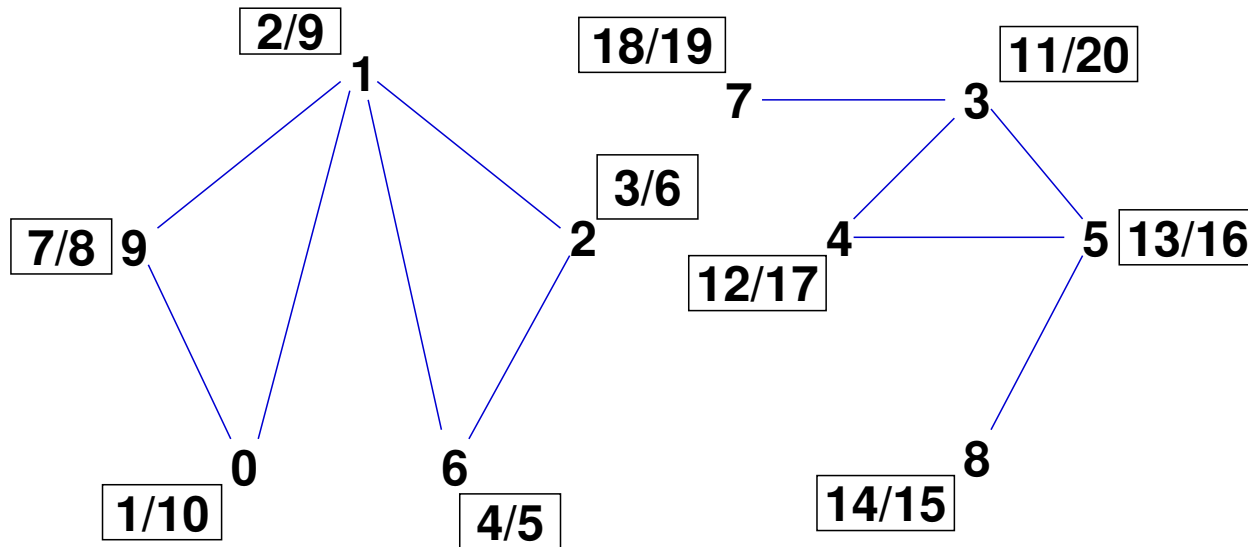
PP d'un graphe orienté, date de début et de fin



Les sommets sont choisis par ordre croissant

PP : exemple

PP d'un graphe non orienté, date de début et de fin



Les sommets sont choisis par ordre croissant

PP : graphe de liaison

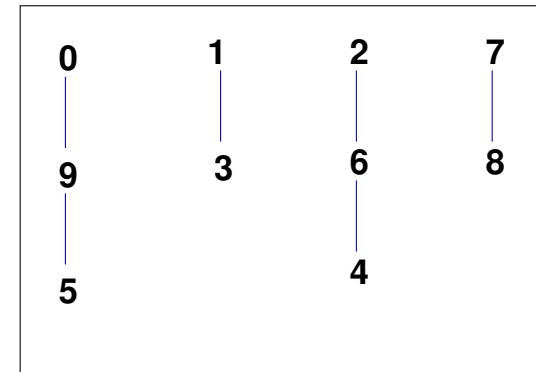
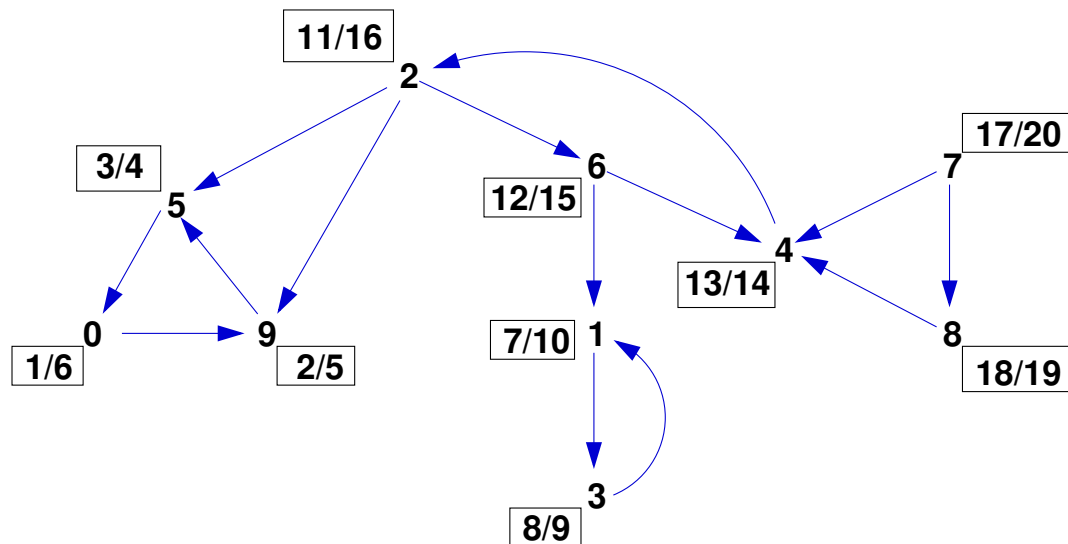
Soit $G = (S, A)$ un graphe (orienté ou non) et π un PP particulier de G

Graphe de liaison

- $G_\pi = (S, A_\pi)$ où $A_\pi = \{(p[u], u) \mid u \in S, p[u] \neq -1\}$
- G_π est une forêt dont chaque arbre possède un *unique* sommet r tel que $p[r] = -1$ (la racine de l'arbre)
- si v est dans un sous-arbre de G_π de racine u , v est un *descendant* de u et u est un *ancêtre* de v dans G_π

PP : exemple

PP d'un graphe orienté et graphe de liaison

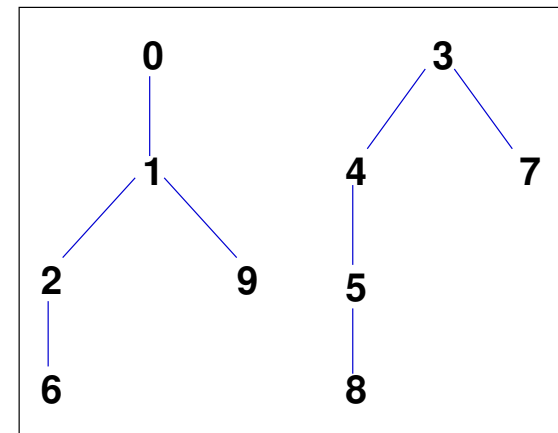
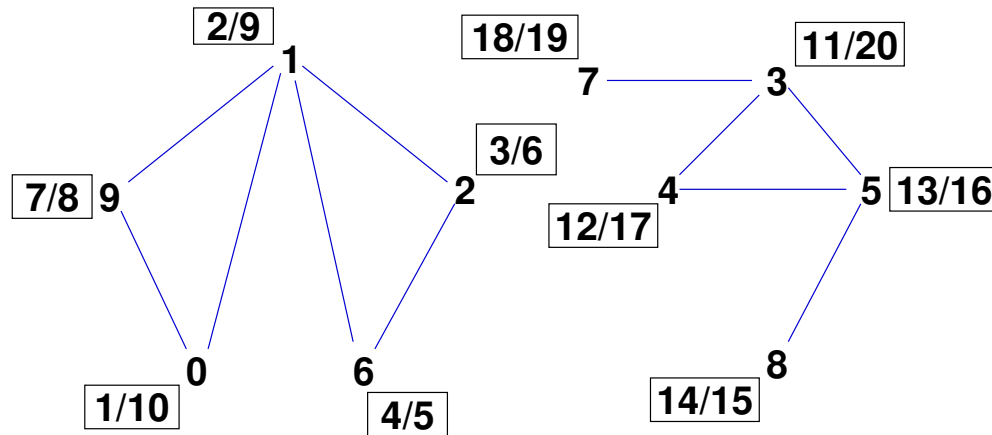


graphe de liaison

Les sommets sont choisis par ordre croissant

PP : exemple

PP d'un graphe non orienté et graphe de liaison



graphe de liaison

Les sommets sont choisis par ordre croissant

PP : classification



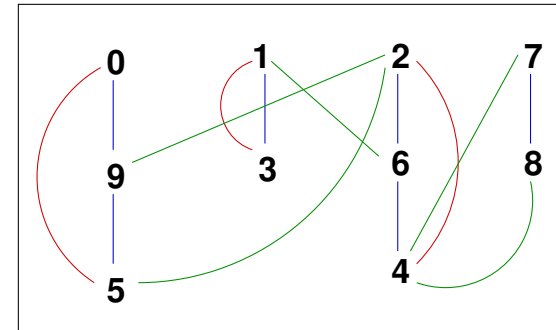
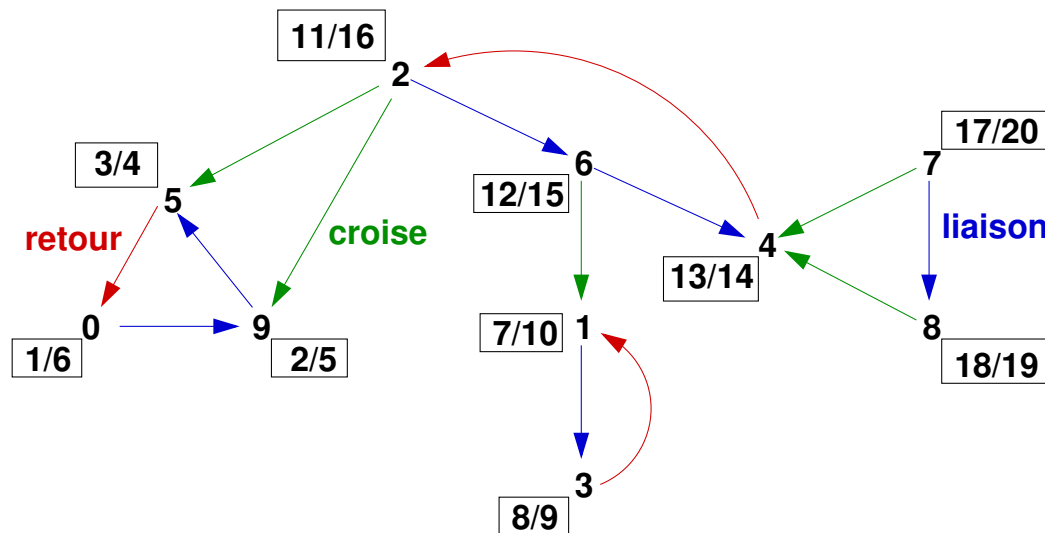
Classification des arcs (cas orienté)

- **arcs de liaison** : les arcs de G_π
- **arcs retour** : les arcs $(u, v) \in A$ où u est un descendant de v dans G_π
- **arcs croisés** : les arcs $(u, v) \in A$ où u et v $\in$ à deux arbres distincts de G_π
- **arcs couvrants** : tous les autres arcs



PP : exemple

PP d'un graphe orienté et classification des arcs



Les sommets sont choisis par ordre croissant

PP : classification



Classification des arêtes (cas non orienté)

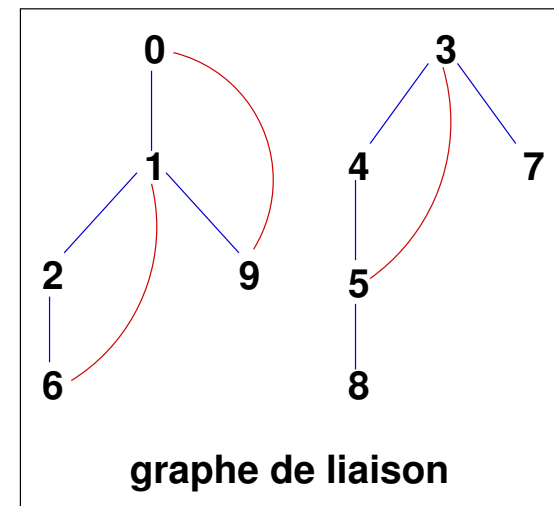
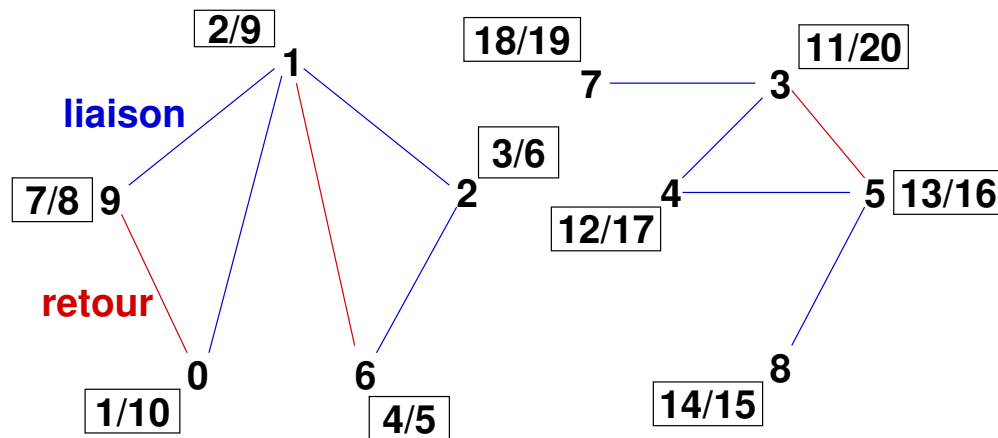
- **arêtes de liaison** : les arêtes de G_π
- **arêtes retour** : les arêtes $(u, v) \in A$ où u est un descendant de v dans G_π
- **aucun** autre type d'arête

Si on découvre v depuis u durant le PP et que v est noir, alors (v, u) est une arête retour dans π



PP : exemple

PP d'un graphe non orienté et classification des arêtes



Les sommets sont choisis par ordre croissant

PP : algorithme



La méthode publique

```
public void parcourir() {  
    t = 1;  
    for ( int i = 0; i < G.ordre(); i++ ) {  
        c[i] = blanc;  
        p[i] = -1;  
    }  
    for ( int i = 0; i < G.ordre(); i++ ) {  
        if ( c[i] == blanc )  
            visiter(i);  
    }  
}
```



PP : algorithme

Visite en profondeur d'un graphe orienté

```
private void visiter(int i) {
    c[i] = gris; d[i] = t++; Adjacents adj = G.adjacents(i);
    while ( adj.nonVide() ) {
        int s = adj.suivant();
        switch ( c[s] ) {
            case blanc :
                afficher(i,s,"de_liaison");p[s]=i;visiter(s);break;
            case gris :
                afficher(i,s,"retour"); break;
            case noir :
                afficher(i,s,"croisé/couvrant"); }
    }
    c[i] = noir; f[i] = t++;
}
```

PP : algorithme

Visite en profondeur d'un graphe non orienté

```
private void visiter(int i) {
    c[i] = gris; d[i] = t++; Adjacents adj = G.adjacents(i);
    while ( adj.nonVide() ) {
        int s = adj.suivant();
        if ( c[s] == blanc ) {
            afficher(i,s,"de_liaison"); p[s] = i; visiter(s);
        }
        else {
            if ( c[s] == gris && s != p[i] )
                afficher(i,s,"retour");
        }
    }
    c[i] = noir; f[i] = t++;
}
```

PP : propriétés

Propriété des parenthèses

Soit $G = (S, A)$ un graphe (orienté ou non) et π un PP *particulier* de G . Pour tout u et $v \in S$, une et une seule des trois conditions suivantes est vérifiée

- $[d[u], f[u]]$ et $[d[v], f[v]]$ sont disjoints
- $[d[v], f[v]] \subset [d[u], f[u]]$ et v est un descendant de u dans G_π
- $[d[u], f[u]] \subset [d[v], f[v]]$ et u est un descendant de v dans G_π

PP : propriétés

Propriété du chemin blanc

Soit $G = (S, A)$ un graphe (orienté ou non) et π un PP *particulier* de G .

un sommet v est un descendant d'un sommet u dans G_π



durant le parcours π à la date $d[u]$
 $\exists$ un chemin allant de u à v constitué de sommets blancs

PP : applications



Très nombreuses, notamment :

- tri topologique (graphe orienté)
- connexité (graphe non orienté)
- 2-connexité (graphe non orienté)
- forte connexité (graphe orienté)
- voir les exercices de la feuille de TD...



PL : présentation

- parcours itératif d'une partie graphe à l'aide d'une file
- explore le graphe à partir d'un sommet particulier s (la source)
- algorithme identique pour les graphes orientés et non orientés
- calcule $\delta(s, u)$ pour tout sommet u , la distance de plus court chemin en nombre d'arcs ou d'arêtes entre la source s et un sommet u quelconque

PL : attributs

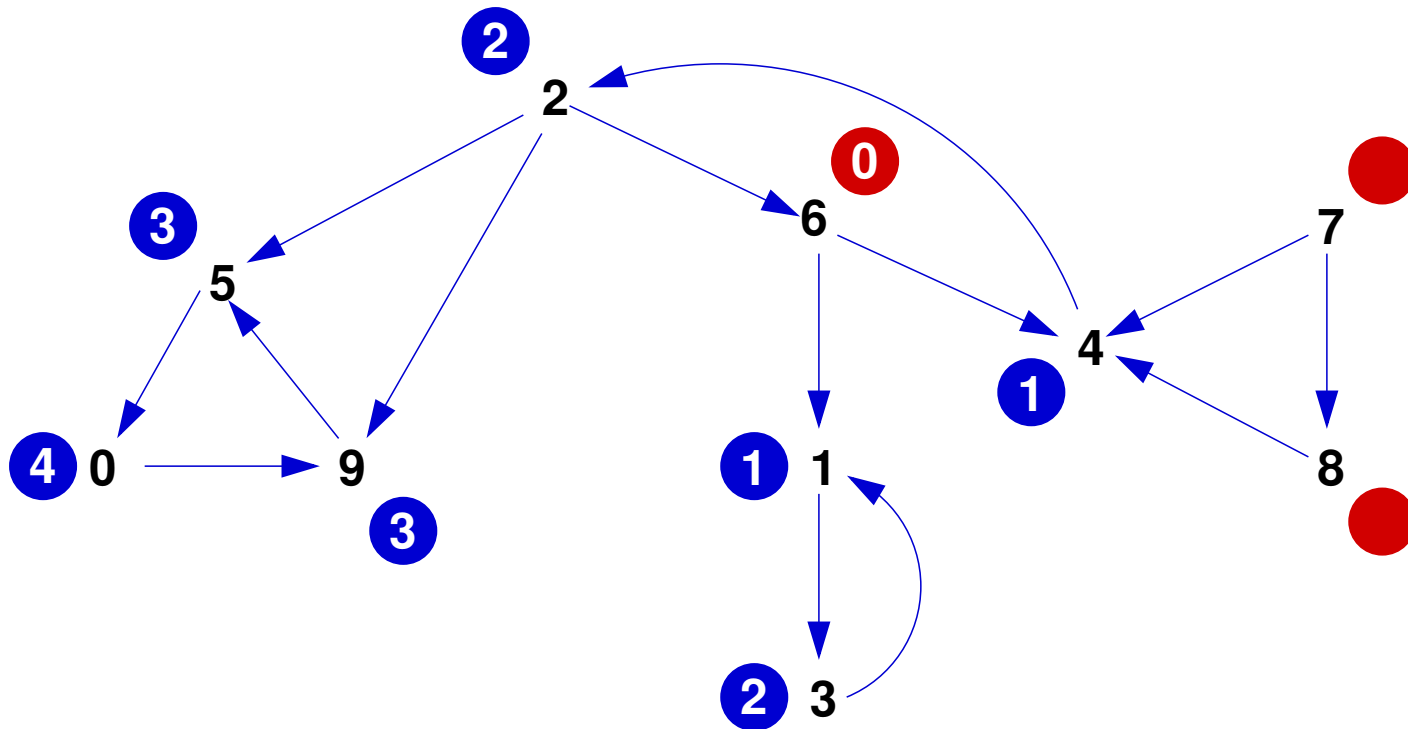


Attributs

- la **source** : le sommet s à partir duquel on explore le graphe
- le **père** : $p[u]$, le sommet à partir duquel le sommet u à été découvert
- la **couleur** : $c[u]$, indique l'état du sommet u durant le parcours. Les sommets sont blancs, gris ou noirs
- la **distance** : $D[u]$, la distance courante en nombre d'arcs ou d'arêtes entre s et u



PL : exemple



x = D[u] et source = 6

PL : algorithme

```
public int[] parcourir(int s) {  
    for( int i = 0; i < G.ordre(); i++ ) {  
        c[i] = blanc; D[i] = -1; p[i] = -1; }  
    FileStatique F = new FileStatique(G.ordre());  
    c[s] = gris; D[s] = 0; ajouter(s,F);  
    while ( nonVide(F) ) {  
        int u = enlever(F); Adjacents adj = G.adjacents(u);  
        while ( adj.nonVide() ) {  
            int v = adj.suivant();  
            if ( c[v] == blanc ) {  
                c[v]=gris; D[v]=D[u]+1; p[v]=u; ajouter(v,F); }  
            }  
        c[u] = noir;  
    }  
    return D;  
}
```

PL : validité



Invariant

- u est gris $\iff u \in F$
- u est gris ou noir $\iff D[u] = \delta(s, u)$
- u est gris ou noir et v blanc $\implies \delta(s, v) > \delta(s, u)$
- u est blanc et $s \rightarrow_G u \implies \exists w$ gris, et un chemin constitué de sommets blancs de w à u

Variant

- $(n_b, |F|)$ décroît strictement et est borné par 0, où n_b est le nombre de sommets blancs



PL : synthèse



Complexité

Au pire, chaque sommet passe une et une seule fois dans la file, et tous ses adjacents sont énumérés $\implies O(|S| + |A|)$

Applications

- calcul de $\delta(s, u)$ pour un sommet s donné
- reconstitution du chemin le plus court avec $p[u]$
- tester si un graphe non orienté est bi-parti
- calculer le diamètre d'un arbre



Tri topologique : définition

Soit $G = (S, A)$ un graphe orienté *acyclique* et la relation binaire $\rightarrow_G$ sur S définie par :

$$u \rightarrow_G v \iff \exists \text{ un chemin de } u \text{ à } v \text{ dans } G$$

- $\rightarrow_G$ est réflexive : $\forall u \in S, u \rightarrow_G u$
- $\rightarrow_G$ est anti-symétrique : $u \neq v$ et $u \rightarrow_G v \implies v \not\rightarrow_G u$
- $\rightarrow_G$ est transitive : $u \rightarrow_G v$ et $v \rightarrow_G w \implies u \rightarrow_G w$
- $\rightarrow_G$ est partielle : $\exists u, v \in S$, avec $u \not\rightarrow_G v$ et $v \not\rightarrow_G u$

La relation $\rightarrow_G$ est une relation d'ordre partiel sur S

Tri topologique : définition

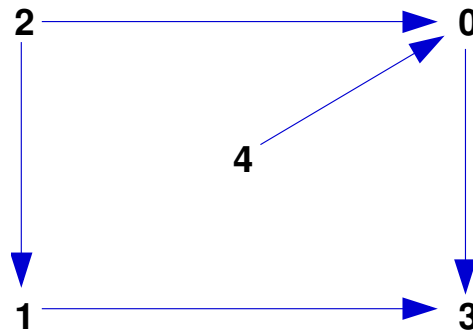
Un tri topologique de G est une suite *ordonnée* $\langle u_0, u_1, \dots, u_{n-1} \rangle$ de tous les sommets de S telle que, pour tout i et j , $0 \leq i < j < n$:

- soit u_i et u_j ne sont pas comparables par $\rightarrow_G$
- soit $u_i \rightarrow_G u_j$

Si $u_i \rightarrow_G u_j$, on dit que u_j *dépend* de u_i (emprunté à l'**ordonnancement** de tâches)

Tri topologique : exemple

Ordonnancement de tâches



On peut effectuer les 5 tâches dans différents ordres :

• 2, 4, 1, 0 et 3

• 4, 2, 1, 0 et 3

• 4, 2, 0, 1 et 3

.....

Tri topologique : algorithme

Principe

Effectuer un PP de G et présenter les sommets dans l'ordre suivant leur date de fin décroissante

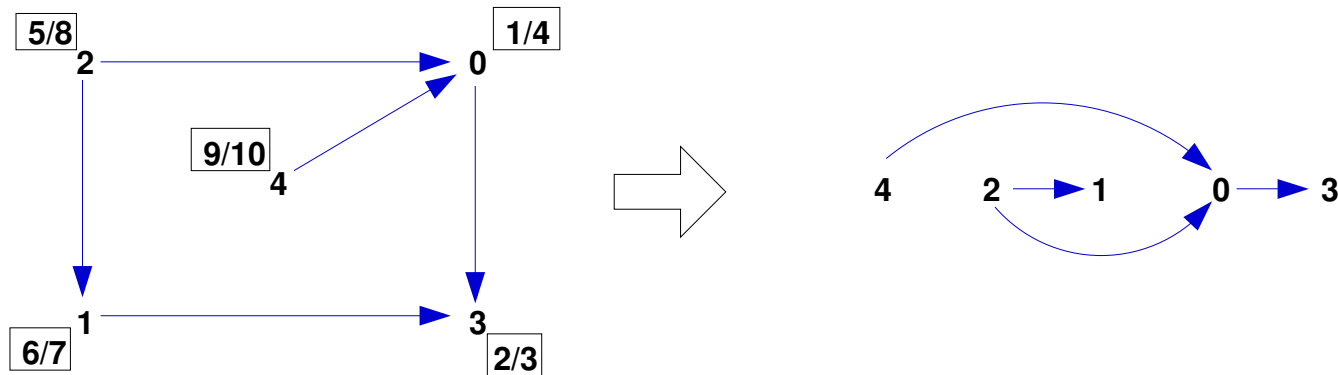
Validité

Si le résultat du tri topologique est $\langle u_0, u_1, \dots, u_{n-1} \rangle$, alors $\forall i, j, 0 \leq i < j < n, f[u_i] > f[u_j]$ et :

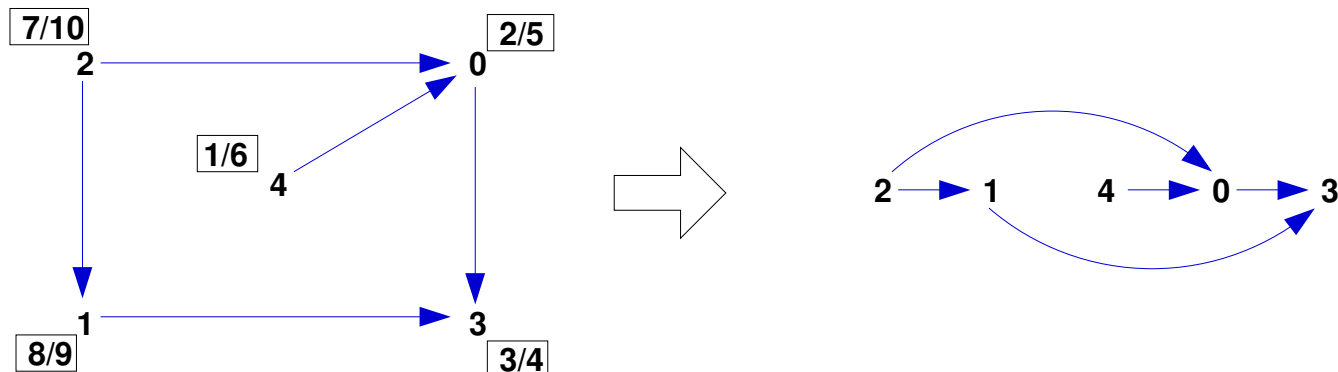
- soit il n'y pas de chemin entre u_i et u_j
- soit $u_i \rightarrow_G u_j$ et $u_j \not\rightarrow_G u_i$ (sinon u_i et $u_j \in$ à un cycle)
- $u_j \rightarrow_G u_i$ est impossible car $f[u_i] > f[u_j]$

Tri topologique : algorithme

Sommets dans l'ordre croissant



Sommets dans l'ordre décroissant



Tri topologique : algorithme

Méthode publique, adaptation de "parcourir"

```
public PileStatique trier() {  
    PileStatique pile = new PileStatique(G.ordre());  
    for ( int i = 0; i < G.ordre(); i++ ) {  
        c[i] = blanc;  
    }  
    for ( int i = 0; i < G.ordre(); i++ ) {  
        if ( c[i] == blanc ) {  
            visiter(i, pile);  
        }  
    }  
    return pile;  
}
```

Tri topologique : algorithme

Méthode privée, adaptation de "visiter"

```
private void visiter(int i, PileStatique pile) {
    c[i] = gris; Adjacents adj = G.adjacents(i);
    while ( adj.nonVide() ) {
        int s = adj.suivant();
        if ( c[s] == blanc ) {
            visiter(s, pile);
        }
    }
    c[i] = noir;
    pile.empiler(new Integer(i));
}
```

Connexité : problèmes



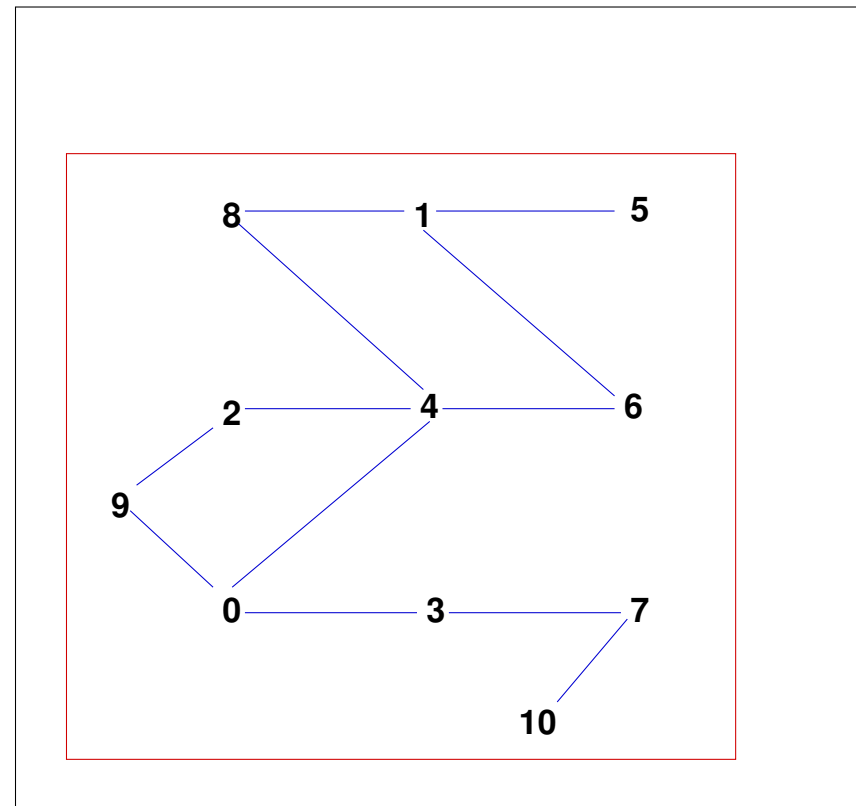
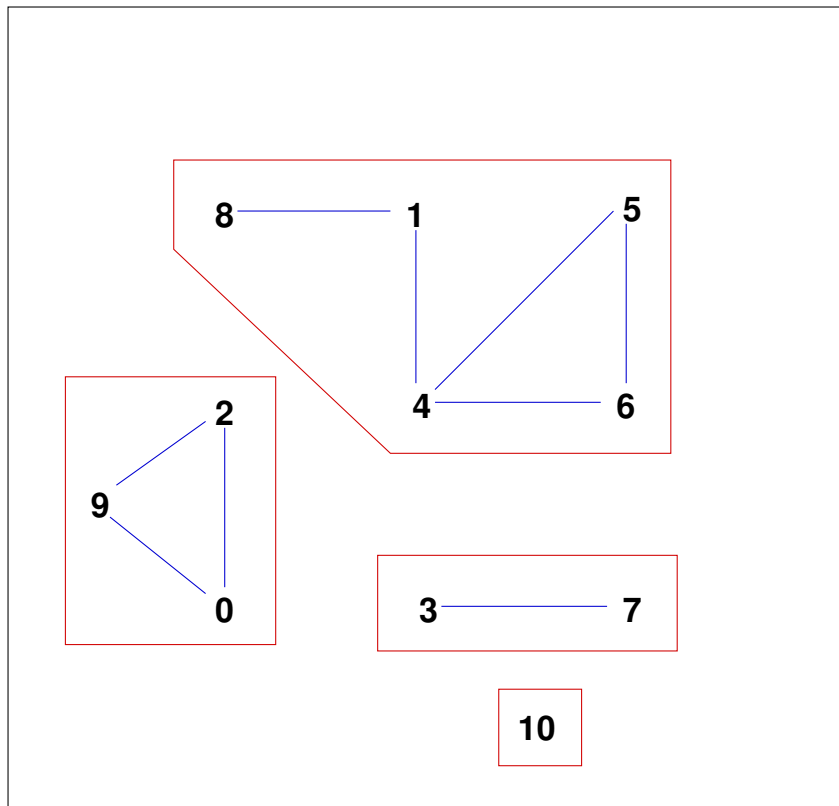
Etant donné un graphe $G = (S, A)$ non orienté

- décider s'il est connexe ou non
- s'il n'est pas connexe, donner le nombre k de ses composantes connexes
- fabriquer ses composantes connexes, c.a.d fabriquer un tableau CC tel que $CC[i]$ est le numéro de la composante connexe à laquelle i appartient ($0 \leq CC[i] < k$)



Connexité : problèmes

Composantes connexes d'un graphe non orienté



Connexité : algorithme



Principe

Effectuer un PP de G et identifier chaque arbre de la forêt en profondeur à une composante connexe

Validité

La propriété du chemin blanc assure qu'une fois le premier sommet découvert, tous ceux de sa composante connexe seront atteints. Il suffit alors de recommencer avec un sommet non atteint, et ce jusqu'à ce que tous les sommets aient été visités



2-Connexité : définitions

Soit $G = (S, A)$ un graphe non orienté connexe avec $|S| \geq 3$

Point d'articulation

$u \in S$ est un point d'articulation $\iff$ le sous-graphe de G engendré par $S \setminus \{u\}$ est non connexe

Pont

$(u, v) \in A$ est un pont de $G \iff$ le sous-graphe G' de G , $G' = (S, A \setminus \{(u, v)\})$, est non connexe

2-Connexité : propriétés

2-Connexité

G est 2-connexe $\iff G \not\ni$ un point d'articulation

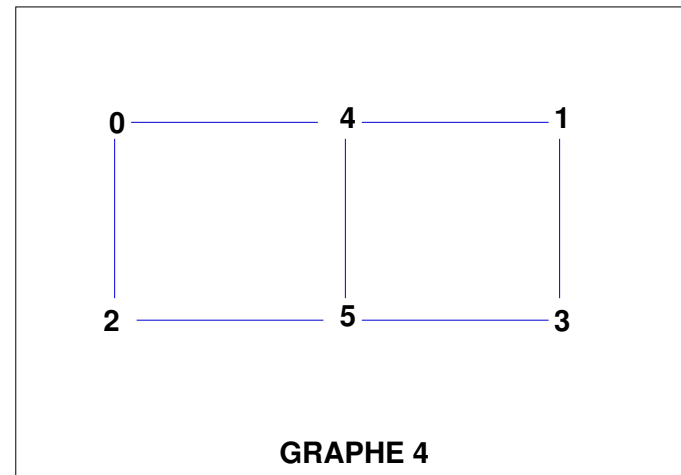
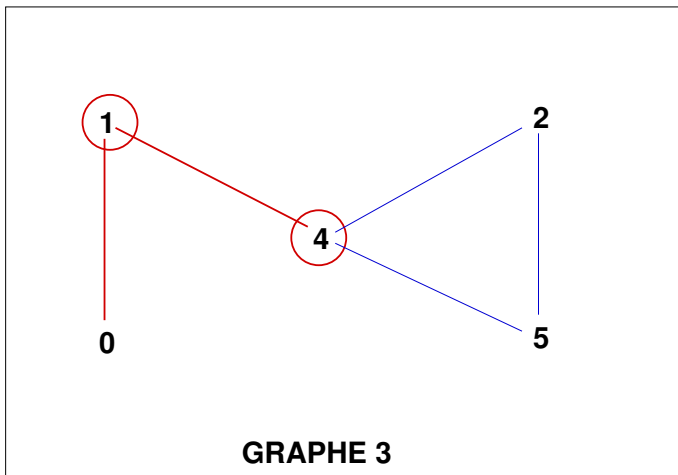
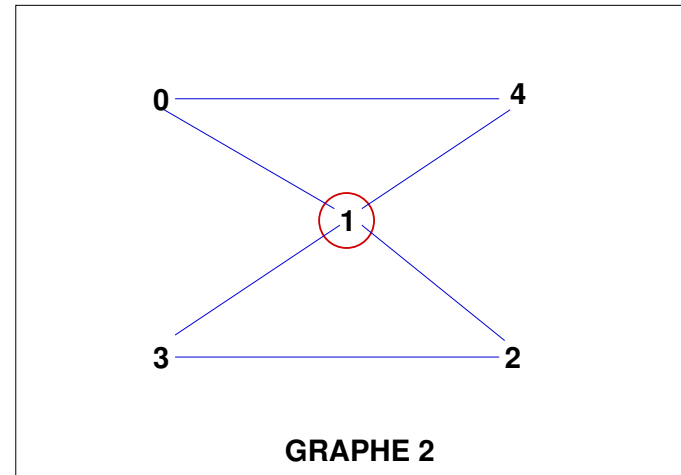
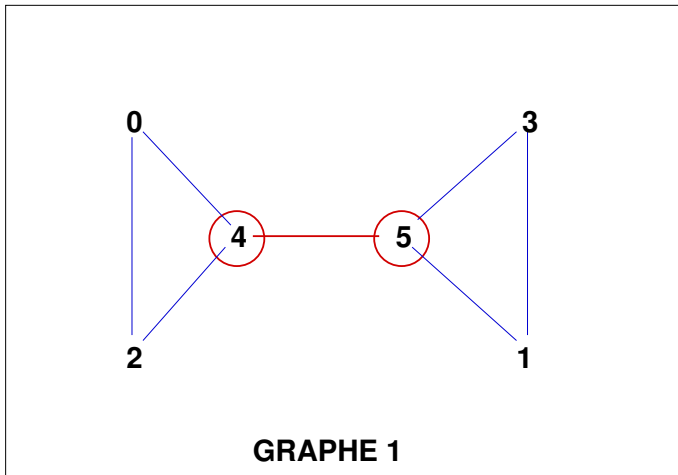
Points d'articulation et ponts

$G \ni$ un pont $\implies G \ni$ un point d'articulation

Composante 2-connexe

- une composante 2-connexe de G = un sous-graphe engendré maximal 2-connexe
- les composantes 2-connexes de G partitionnent A

2-Connexité : exemple



2-Connexité : algorithme

Base

Effectuer π , un PP de G . Comme G est connexe, G_π est un arbre

Définition

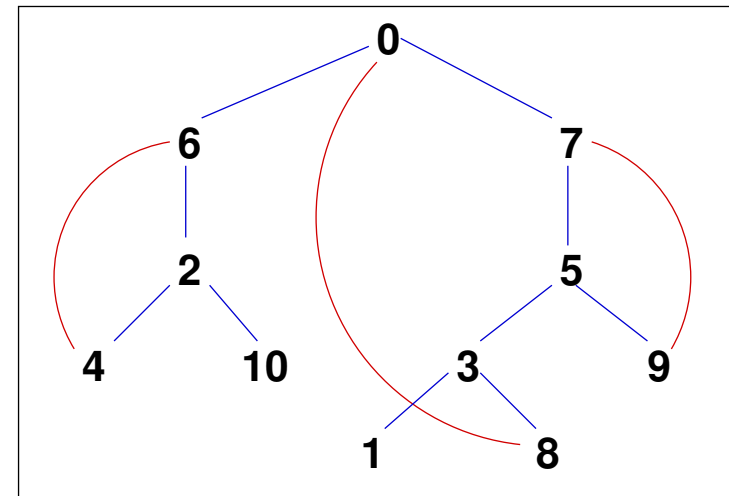
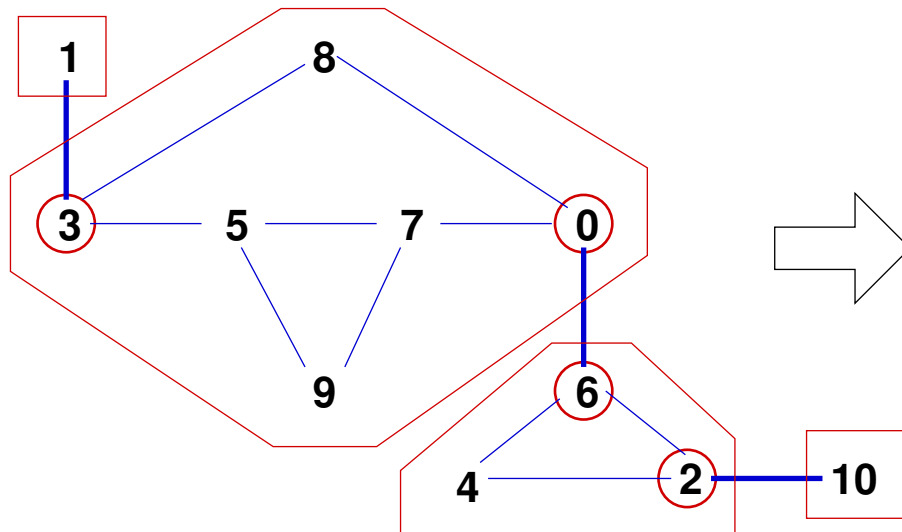
$\forall u \in S, u \neq$ de la racine de G_π :

Le sommet u *remonte* (strictement) $\iff \exists$ une arête retour dans G_π partant de u ou d'un de ses descendants dans G_π , et arrivant à un ancêtre (propre) de $p[u]$

Remarque : $\forall u \in S, u$ est un ancêtre de u

2-Connexité : exemple

2-connexité et PP d'un graphe non orienté



GRAPHE DE LIAISON

Les sommets sont choisis en ordre croissant

2-Connexité : algorithme

Propriétés

- $r \in S$, r racine de G_π est un point d'articulation $\iff r$ a au moins deux fils dans G_π
- $u \in S$, $u \neq$ de la racine de G_π , est un point d'articulation $\iff u$ a au moins un fils dans G_π qui ne remonte pas strictement
- $(u, v) \in A$ est un pont $\iff p[\alpha] = \beta$ et α ne remonte pas dans G_π ($\alpha = u$ et $\beta = v$ ou $\alpha = v$ et $\beta = u$)

2-Connexité : algorithme



Attribut

- $\mathcal{D}(u) = \{u\} \cup \{v \in S \mid v \text{ descendant de } u \text{ dans } G_\pi\}$
- $\mathcal{R}(u) = \{w \in S \mid \exists v \in \mathcal{D}(u), (v, w) \text{ arête retour dans } G_\pi\}$
- $m[u] = \min\{d[v] \mid v \in \mathcal{D}(u) \cup \mathcal{R}(u)\}$

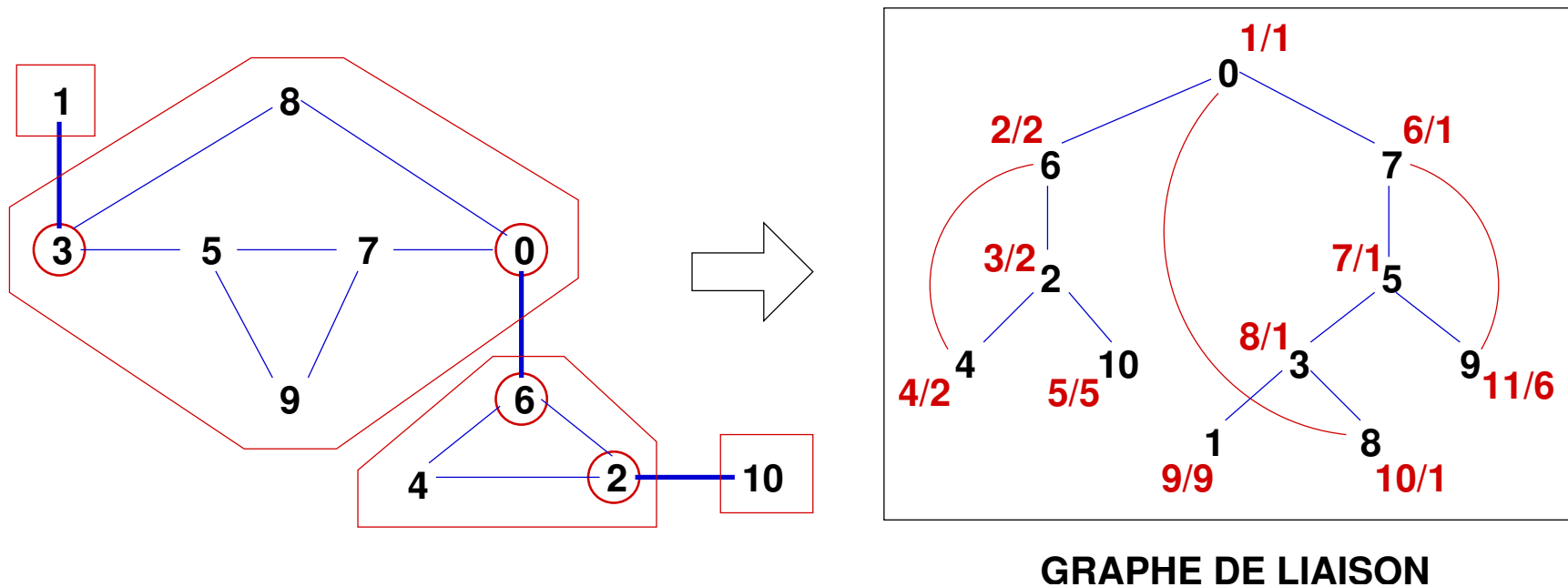
Propriétés

- $u \text{ remonte dans } G_\pi \iff m[u] < d[u]$
- $u \text{ remonte strictement dans } G_\pi \iff m[u] < d[p[u]]$



2-Connexité : exemple

PP d'un graphe non orienté et attribut $d[u]/m[u]$



Les sommets sont choisis en ordre croissant

Forte connexité : problèmes

Etant donné un graphe $G = (S, A)$ orienté

- décider s'il est fortement connexe ou non
- s'il n'est pas fortement connexe, donner le nombre k de ses composantes fortement connexes (CFC)
- fabriquer ses CFC, c.a.d fabriquer un tableau CC tel que $CC[i]$ est le numéro de la CFC à laquelle le sommet i appartient ($0 \leq CC[i] < k$)
- fabriquer le graphe réduit de G

Forte connexité : définition

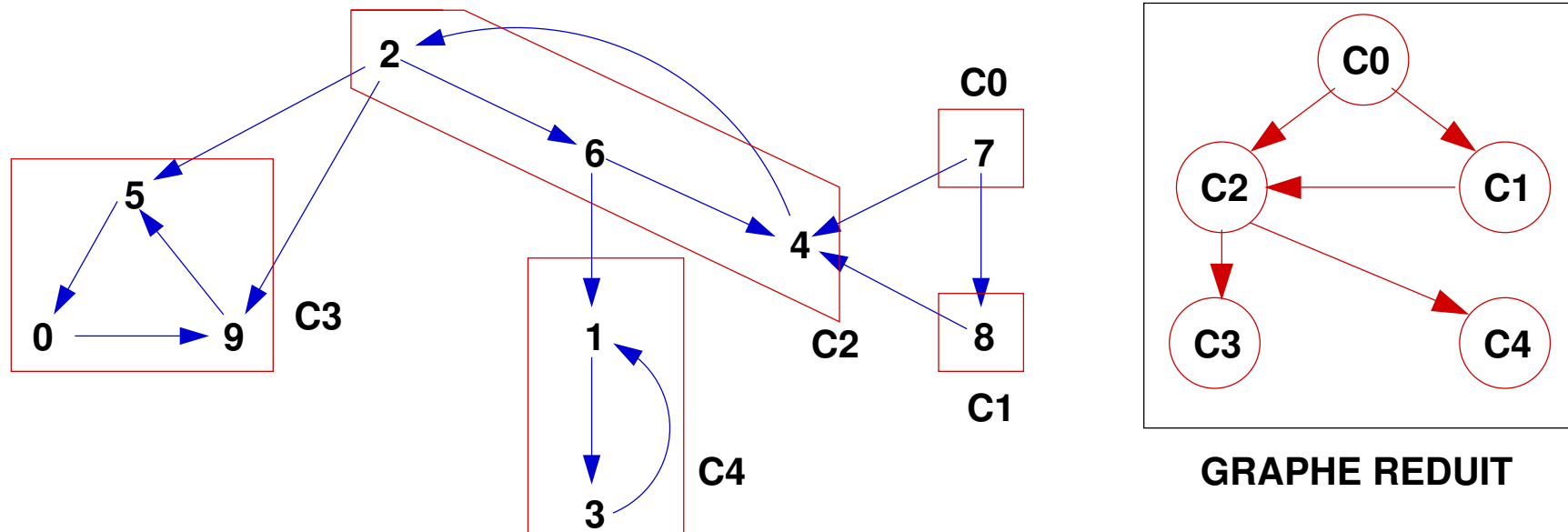
Graphe réduit d'un graphe orienté $G = (S, A)$

- G_r , le graphe réduit de G , est un graphe orienté
- G_r est le graphe des CFC de G
- $G_r = (S_G, A_G)$ avec $S_G = \{C \mid C \text{ est une CFC de } G\}$ et $A_G = \{(C, C') \mid C, C' \in S_G, \exists u \in C, u' \in C', (u, u') \in A\}$
- G_r est acyclique

G_r est une représentation synthétique de G

Forte connexité : exemple

Composantes fortement connexes et graphe réduit



Forte connexité : algorithme



Idée de base

Peut-on s'inspirer du cas non orienté ?

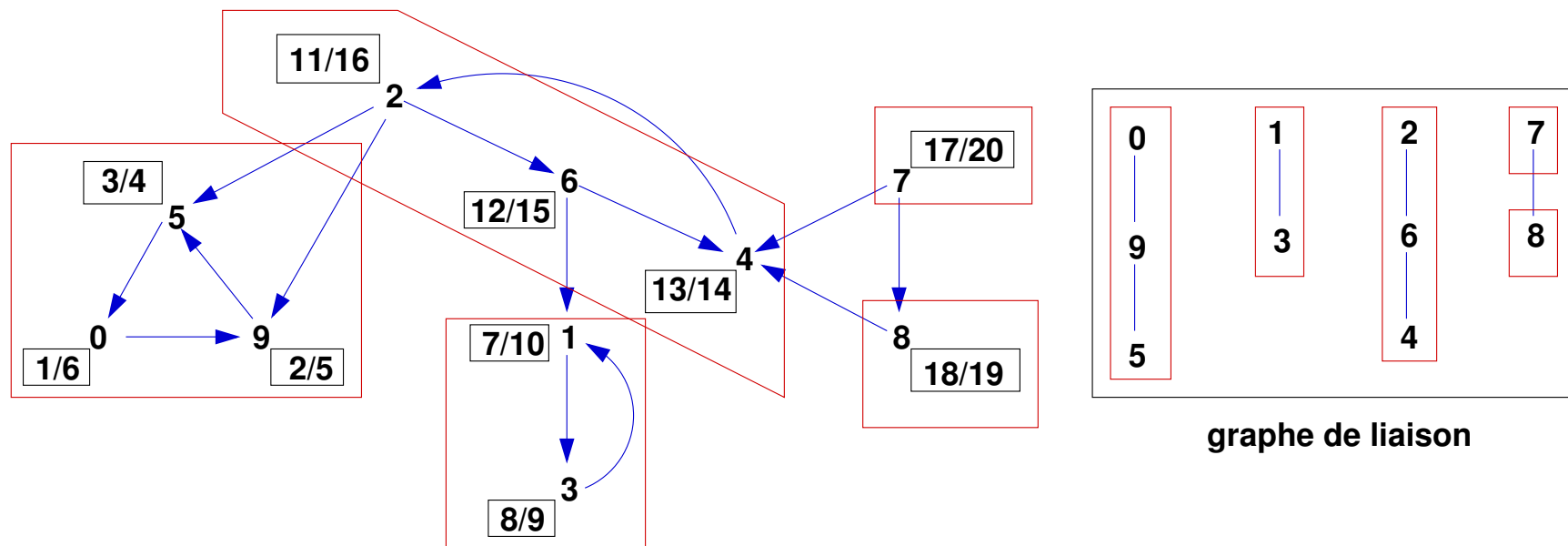
Problèmes

- cas non orienté : **tous** les PP de G donnent les même composantes connexes (les arbres de G_π)
- cas orienté : seulement **certains** PP de G donnent une forêt en profondeur dont les arbres s'identifient exactement avec les CFC



Forte connexité : exemple

Composantes fortement connexes et PP



Les sommets sont choisis dans l'ordre croissant

Forte connexité : algorithme



Idée de base améliorée

Existe-t-il un prétraitement de G qui donne l'ordre dans lequel il faut **choisir** les sommets pour qu'un PP de G identifie les arbres de la forêt en profondeur avec les CFC ?

Intuition

Le tri topologique classe les sommets d'un graphe (acyclique) en fonction de l'existence de chemin entre ses sommets



Forte connexité : algorithme

Propriété

Soit $G = (S, A)$ et $G_r = (S_G, A_G)$ son graphe réduit. On effectue un PP de G et on considère $f[u], \forall u \in S$. Si $(C, C') \in A_G$ alors $\exists u \in C \mid \forall u' \in C', f[u] > f[u']$

Conséquences

- un tri "topologique" de G donne la liste des sommets dans l'ordre où il ne faut **pas** les choisir !
- inverser la liste précédente est faux car n'importe quel sommet du graphe peut terminer en premier

Forte connexité : algorithme

Solution

Ca marche sur tG !

Définition et propriétés

- Soit $G = (S, A)$ un graphe orienté. Le graphe orienté ${}^tG = (S, {}^tA)$, le transposé de G , est défini par tA :
$${}^tA = \{(v, u) \mid v, u \in S, (u, v) \in A\}$$
- G et tG ont les mêmes CFC
- ${}^t(G_r) = ({}^tG)_r$

Forte connexité : algorithme



Principe

- effectuer un PP de G , collecter les sommets dans P suivant leur date de fin décroissante et fabriquer tG
- effectuer π , un PP de tG en choisissant les sommets dans P : chaque arbre de ${}^tG_\pi$ est une CFC de G

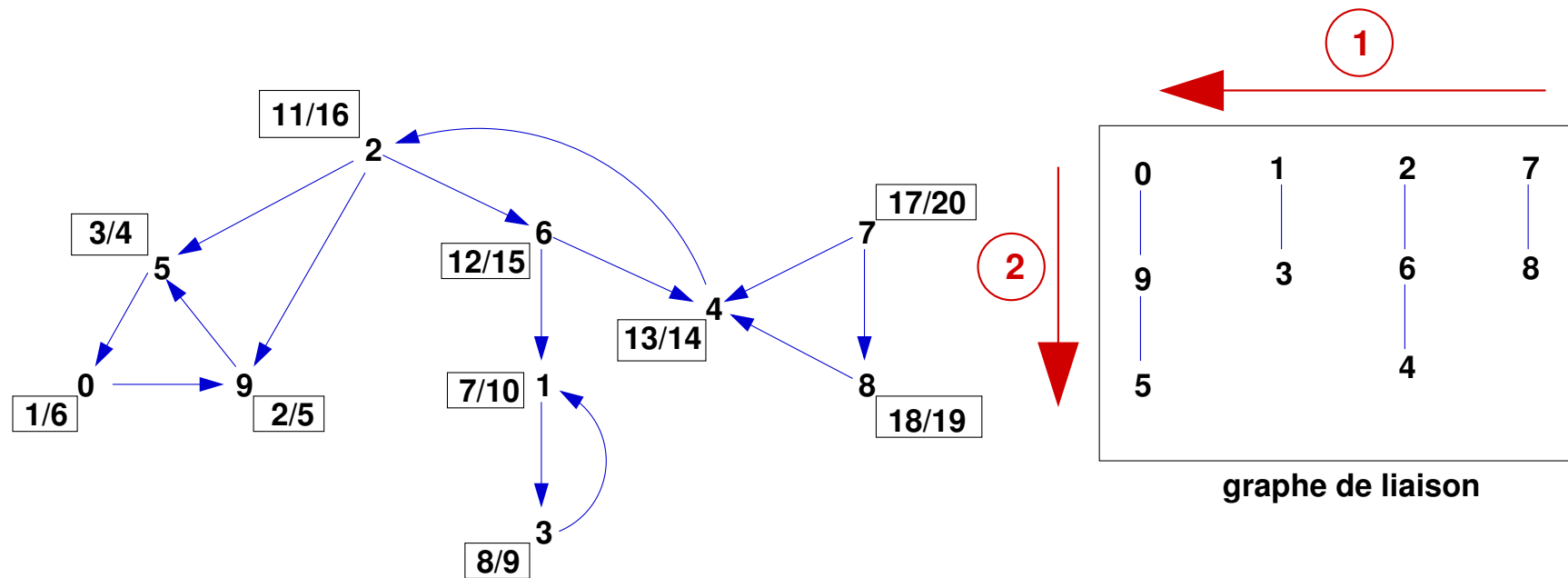
Complexité

- deux parcours en profondeur = $\Theta(|S| + |A|)$ en temps
- le graphe annexe ${}^tG = \Theta(|S| + |A|)$ en espace



Forte connexité : exemple

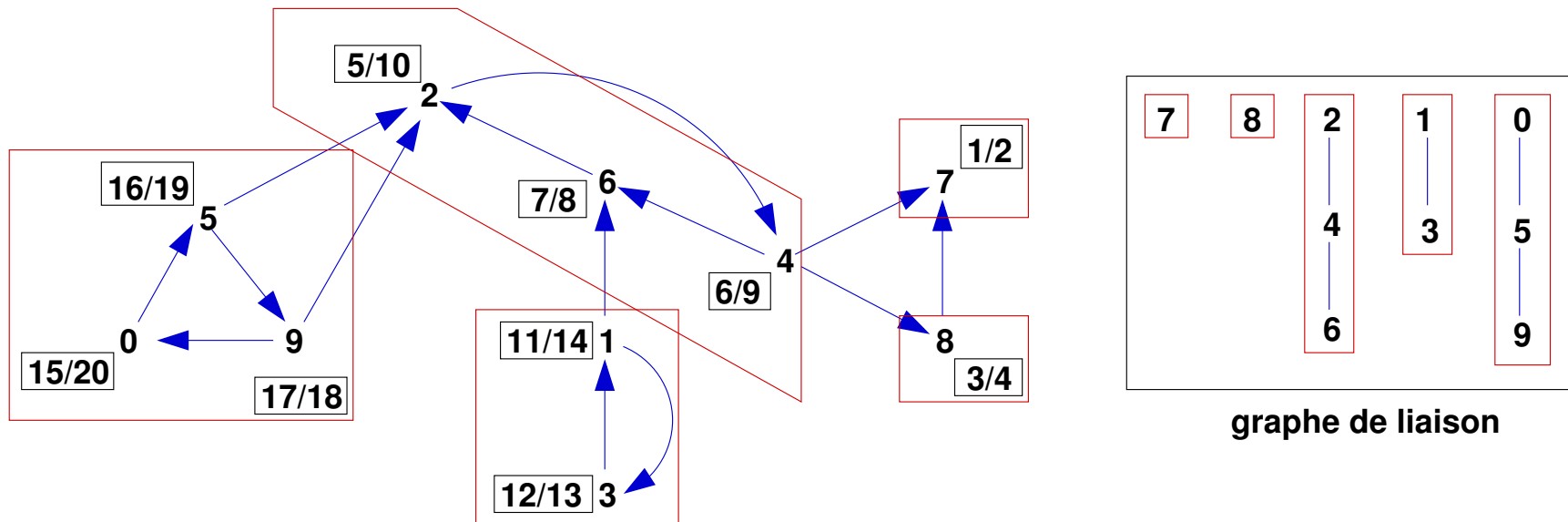
Premier PP et tri "topologique" des sommets



Les sommets sont choisis dans l'ordre croissant

Forte connexité : exemple

Deuxième PP sur le graphe transposé



Les sommets sont choisis dans l'ordre résultant du "tri topologique" précédent, soit 7, 8, 2, 6, 4, 1, 3, 0, 9, 5

Forte connexité : algorithme

Méthode publique

```
public int[] cfc() {  
    return cfc(topo());  
}
```

Deuxième parcours

```
private int[] cfc(PileStatique pile) {  
    int n = 1; int[] cfc = new int[G.ordre()];  
    .....  
    while ( nonVide(pile) ) {  
        int i = depiler(pile);  
        if ( c[i] == blanc ) visiter(i,n++,cfc);  
    }  
    return cfc;  
}
```

Forte connexité : algorithme

Seconde visite en profondeur et marquage des CFC

```
private void visiter(int i, int n, int[] cfc) {  
    c[i] = noir;  
    cfc[i] = n;  
    Adjacents adj = TG.adjacents(i);  
  
    while ( adj.nonVide() ) {  
        int s = adj.suivant();  
        if ( c[s] == blanc ) {  
            visiter(s,n,cfc);  
        }  
    }  
}
```

Forte connexité : algorithme

Premier parcours et fabrication de la pile

```
private PileStatique topo() {  
    PileStatique pile = new PileStatique();  
  
    for ( int i = 0; i < G.ordre(); i++ ) {  
        c[i] = blanc;  
    }  
  
    for ( int i = 0; i < G.ordre(); i++ ) {  
        if ( c[i] == blanc ) {  
            visiter(i, pile);  
        }  
    }  
    return pile;  
}
```

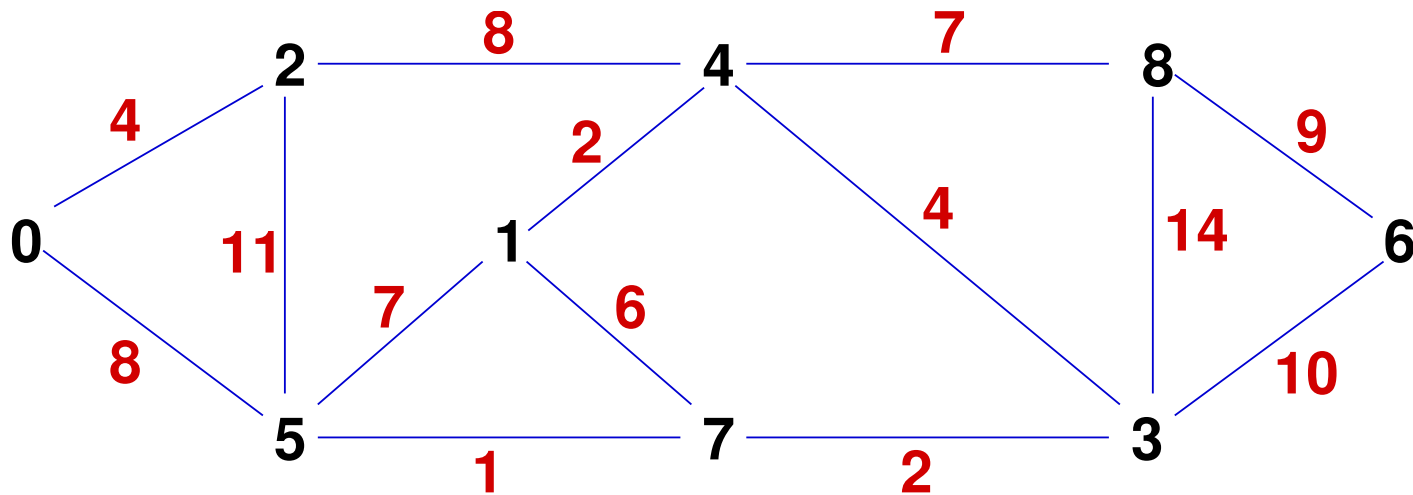
Forte connexité : algorithme

Première visite en profondeur, remplissage de la pile et fabrication du graphe transposé

```
private void visiter(int i, PileStatique pile) {  
    c[i] = noir;  
    Adjacents adj = G.adjacents(i);  
    while ( adj.nonVide() ) {  
        int s = adj.suivant();  
        TG.connecter(s, i);  
        if ( c[s] == blanc ) {  
            visiter(s, pile);  
        }  
    }  
    empiler(i, pile);  
}
```

Arbres couvrants : présentation

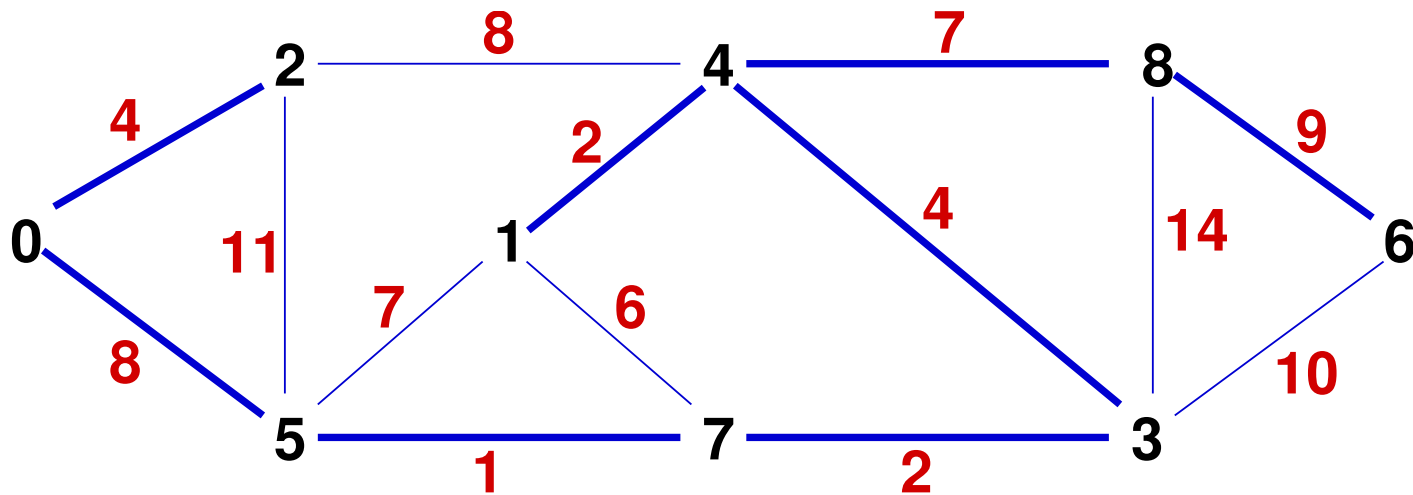
Connecter à moindre coût des éléments à un réseau



Les arêtes du graphe sont les connections possibles avec leur coût (en rouge)

ACM : exemple

Connecter à moindre coût des éléments à un réseau



Les arêtes en gras forment les connections qui minimisent le coût total de la connexité du réseau

ACM : définitions

Soit $G = (S, A)$ un graphe non orienté connexe

- $\omega : A \longrightarrow \mathbb{N}^*$ est une *valuation* de G (on dit que G est ω -valué)
- $\forall G' = (S', A')$, sous-graphe de G , on définit $\omega(G')$, le poids de G' , par $\omega(G') = \sum_{(u,v) \in A'} \omega(u, v)$
- $T = (S, A_T)$, $A_T \subseteq A$, est un arbre **couvrant** de poids minimum (ACM) de $G \iff T$ est un arbre **couvrant** et $\omega(T) = \min\{\omega(T') \mid T' \text{ arbre couvrant de } G\}$

ACM : définitions

- $E \subseteq A$ est un sous-ACM de $G \iff \exists T = (S, A_T)$ ACM de G tel que $E \subseteq A_T$
- $\forall (u, v) \in A \setminus E, (u, v)$ est une arête *sûre* pour $E \iff E \cup \{(u, v)\}$ est un sous-ACM de G
- $\forall S' \subset S, S' \neq \emptyset, S'$ est appelée une *coupure* de G
- $(u, v) \in A$ *traverse* une coupure S' de $G \iff u \in S'$ et $v \in S \setminus S'$ ou bien $u \in S \setminus S'$ et $v \in S'$
- une coupure S' de G *respecte* E , un sous-ACM de $G \iff \forall (u, v) \in E, (u, v)$ ne traverse pas S'

ACM : propriétés

Soit $G = (S, A)$ un graphe non orienté connexe ω -valué, et soit E un sous-ACM de G . Les deux propriétés suivantes sont appelées *propriétés des ACM* :

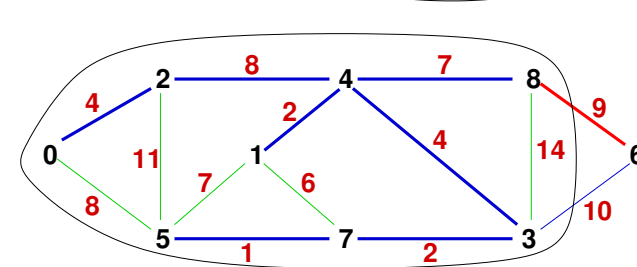
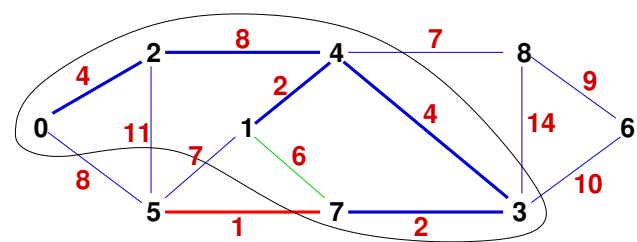
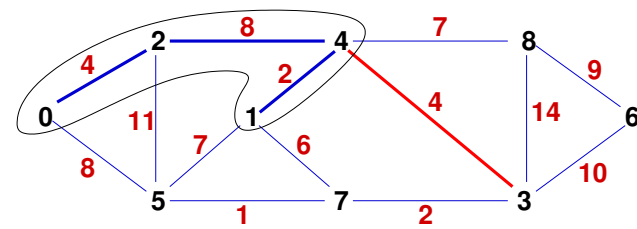
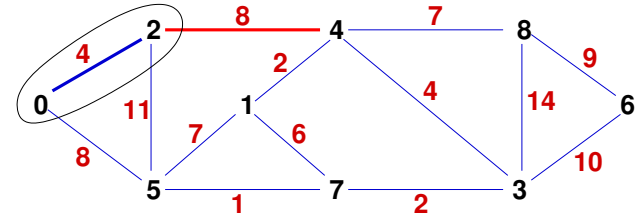
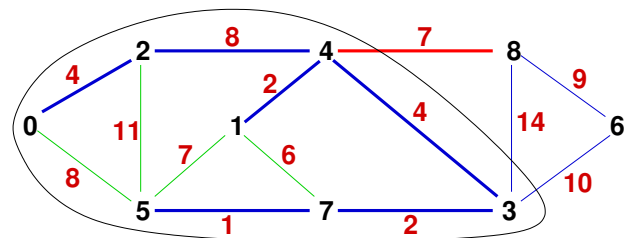
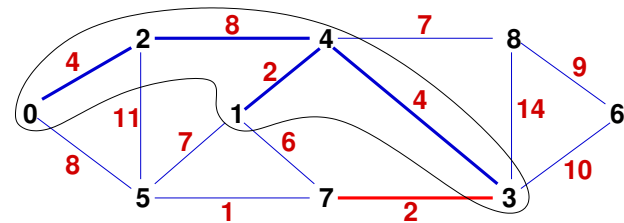
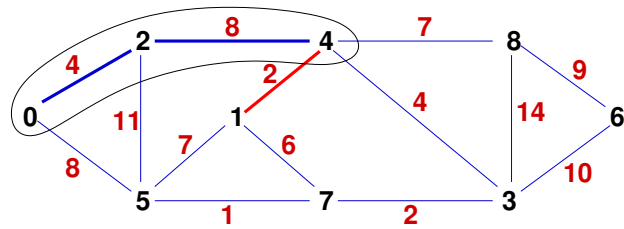
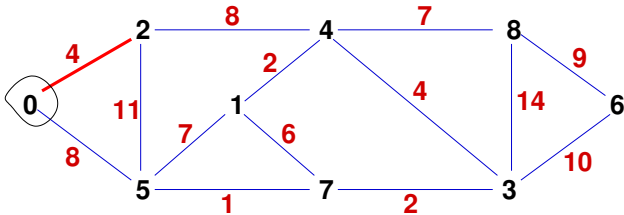
- $\forall S'$ une coupure de G qui respecte E , toute arête (u, v) qui traverse S' et telle que $\omega(u, v) = \min\{\omega(u', v') \mid (u', v') \text{ traverse } S'\}$ est une arête sûre pour E
- $\forall (u, v) \in A \setminus E$ de poids minimum, telle que u et v sont dans des composantes connexes distinctes du graphe $G' = (S, E)$, (u, v) est une arête sûre pour E

Prim : principe

Déduit de la première propriété des ACM

- à l'initialisation, $E = \emptyset$ et $S' = \{u\}$ avec u un sommet quelconque de G
- tant que $G_E = (S', E)$ ne couvre pas G faire :
 - $E \leftarrow E \cup \{(u, v)\}$ où (u, v) est une arête sûre pour E qui traverse S'
 - $S' \leftarrow S' \cup \{u, v\}$
- G_E reste connexe tout au long de l'algorithme

Prim : exemple



Prim : implémentation

Représentation des différents éléments

- $G_E = (S', E)$: un graphe non orienté initialement sans arête, auquel on ajoute les arêtes sûres successives
- S' : un tableau de booléen pour marquer les sommets de S'
- E : simplement les arêtes de G_E
- l'ensemble des arêtes traversant S' : un tas croissant, la clef d'une arête étant son poids. Le minimum du tas est l'arête sûre qui traverse S'

Prim : algorithme

Méthode publique : initialisations

```
public GrapheNonOriente acm() {  
    TasCroissant T = new TasCroissant(G.aretes());  
    GrapheNonOriente Ge = new GrapheNonOriente(G.ordre());  
    boolean[] dansSprime = new boolean[G.ordre()];  
  
    dansSprime[0] = true; int m = 1;  
  
    Adjacents adj = G.adjacents(0);  
    while ( adj.nonVide() ) {  
        int s = adj.suivant();  
        ajouter(0,s,G.poids(0,s),T);  
    }  
    .....  
}
```

Prim : algorithme

Méthode publique : fabrication de E

```
public GrapheNonOriente acm() {  
    .....  
    while ( m < G.ordre() ) {  
        Arete uv = retirerMinimum(T);  
        int u = uv.origine; int v = uv.extremite;  
        if ( ! dansSprime[v] ) {  
            Ge.connector(u,v); dansSprime[v] = true; m++;  
            adj = G.adjacents(v);  
            while ( adj.nonVide() ) {  
                int s = adj.suivant();  
                if ( ! dansSprime[s] ) ajouter(v,s,G.poids(v,s),T);  
            }  
        }  
        return Ge;  
    }  
}
```

Prim : validité



Invariant

- E est un ACM du sous-graphe engendré par S'
l'ensemble des sommets marqués avec $|S'| = m$
- $(u, v) \in A, u \in S' \text{ et } v \notin S' \implies (u, v) \in T$

Variant

- $(m, -|T|)$ croît et est borné par $(G.\text{ordre}, 0)$

Complexité

- $\Theta(|A| \lg |A|)$

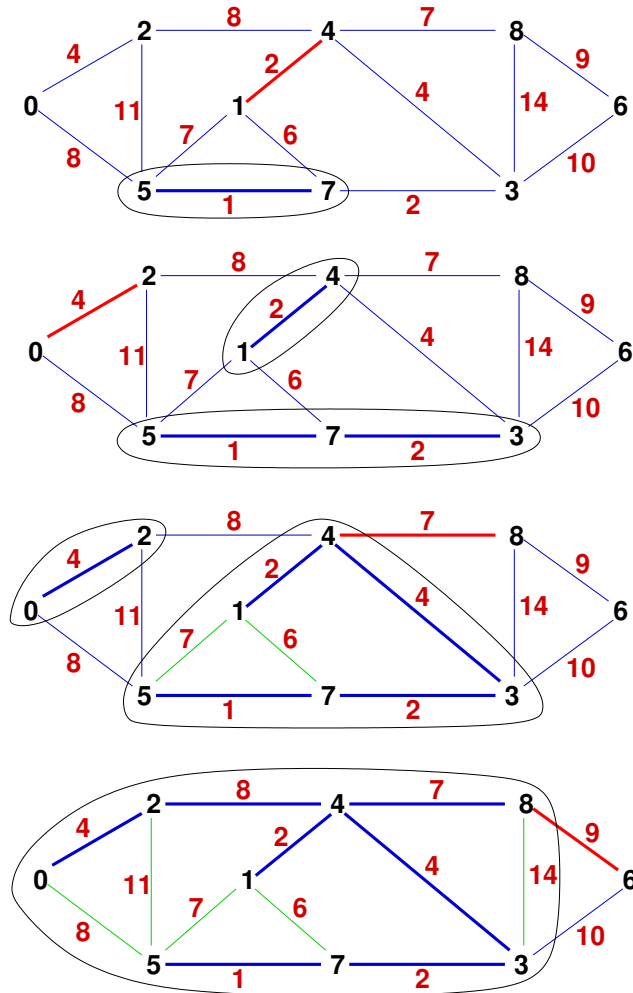
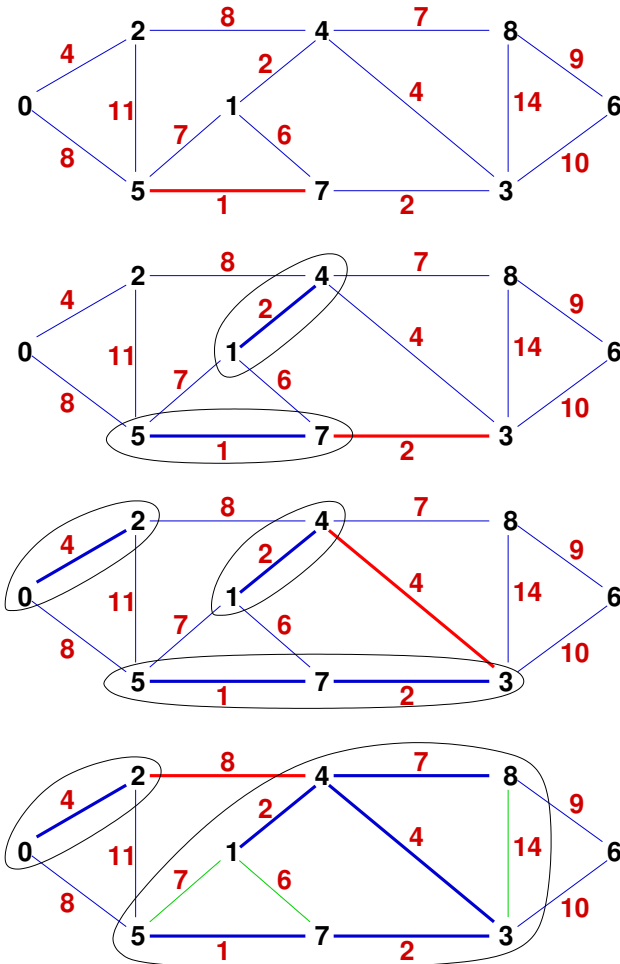


Kruskal : principe

Déduit de la deuxième propriété des ACM

- à l'initialisation, $E = \emptyset$
- tant que $G' = (S, E)$ n'est pas connexe faire :
 - $E \longleftarrow E \cup \{(u, v)\}$ où $(u, v) \in A \setminus E$ est une arête sûre pour E parmi toutes les arêtes reliant deux composantes connexes de G'
- G' n'est pas connexe durant l'algorithme
- problème : gérer les composantes connexes de G' !

Kruskal : exemple



Kruskal : implémentation

Représentation des différents éléments

- $A \setminus E$: on y choisit à chaque itération l'arête de poids minimum $\implies T$, un tas croissant !
- CC , les composantes connexes de $G' = (S, E)$: il faut gérer des ensembles de sommets et pouvoir
 - tester si u et v sont dans la même composante
 - fusionner les composantes de u et v en une seule
- E : simplement les arêtes du graphe G' (noté G_p) en construction

Kruskal : algorithme

Méthode publique

```
public GrapheNonOriente acm() {  
    GrapheNonOriente Gp = new GrapheNonOriente(G.ordre());  
    TasCroissant T = new TasCroissant(G.aretes()); initialiser(T);  
    Composante CC = new Composante(G.ordre()); int k = G.ordre();  
    while ( k > 1 ) {  
        Arete uv = (Arete) T.extraireMinimum().info();  
        int u = uv.origine(); int v = uv.extremite();  
        if ( CC.separes(u,v) ) {  
            Gp.connecter(u,v); CC.reunir(u,v); k--;  
        }  
    }  
    return Gp;  
}
```

Kruskal : validité



Invariant

- $G' = (S, E)$ est acyclique de poids minimum, parmi tous les graphes $G'' = (S, X)$ de G avec $|X| = |E|$
- k est le nombre de composantes connexes de G'
- $k > 1 \iff \exists C_1, C_2$ deux composantes de G' , telles que $\exists u_1 \in C_1, u_2 \in C_2$ avec $(u_1, u_2) \in T$

Variant

- $(k, |T|)$ décroît et est minoré par $(1, 0)$



Kruskal : composantes

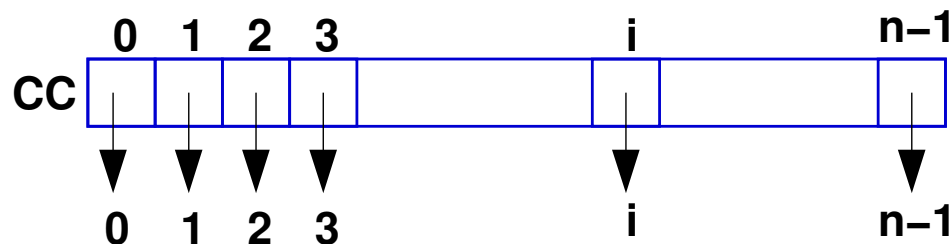
Gestion des composantes connexes de G'

- une composante = un ensemble d'entiers $i, 0 \leq i < n$
où n est l'ordre du graphe
- $\forall i, j$ sommets de G , $\text{separés}(i, j)$ doit tester si i et j
sont dans deux ensembles différents
- $\forall i, j$ sommets de G tels que $\text{separés}(i, j)$ est vrai,
 $\text{reunir}(i, j)$ doit fusionner les deux composantes
auxquelles $\ni i$ et j en une seule composante

Kruskal : composantes

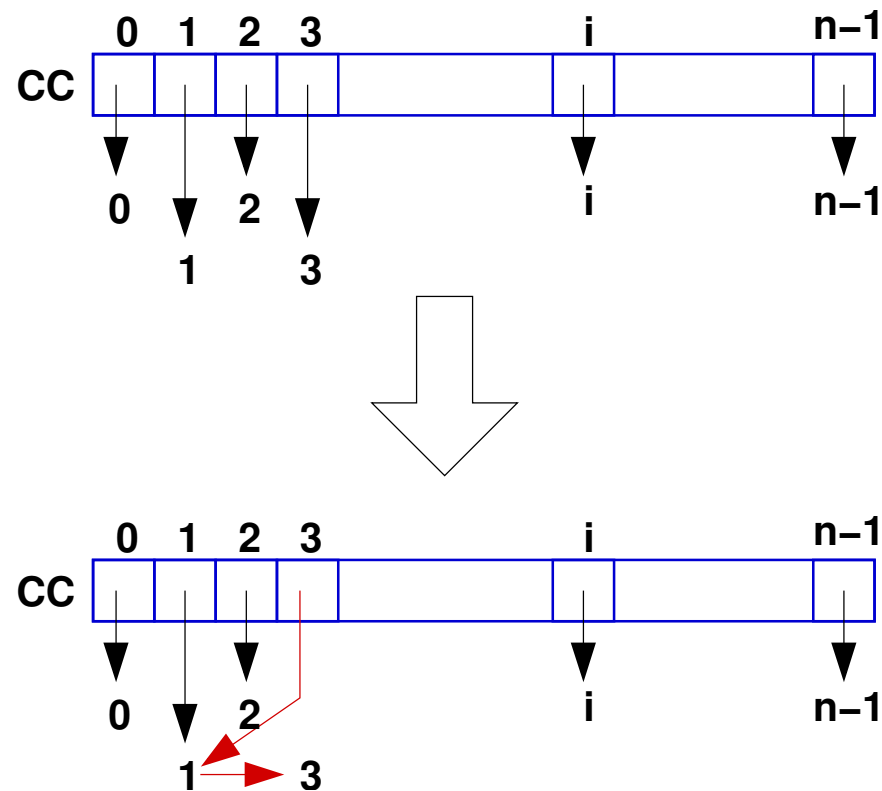
On gère CC , un tableau contenant des pointeurs sur des entiers (les sommets). L'opération $separés(i, j)$ est équivalente à $CC[i] = CC[j]$, et $reunir(i, j)$ consiste à fusionner des listes $CC[i]$ et $CC[j]$, en fusionnant la plus petite dans la plus grande

A l'initialisation



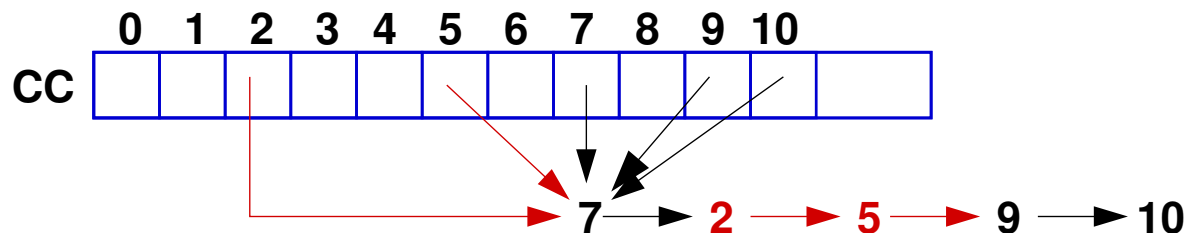
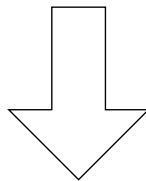
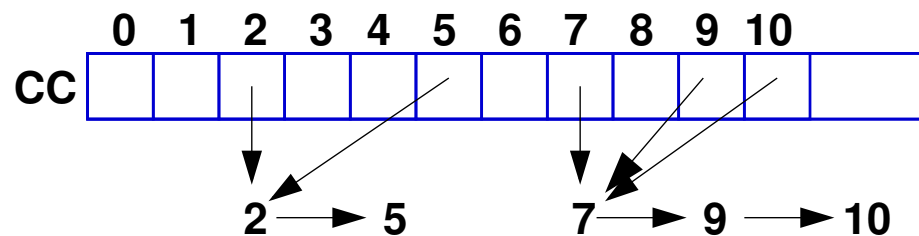
Kruskal : composantes

Réunir les composantes de 1 et 3



Kruskal : composantes

Réunir les composantes de 2 et 7



Kruskal : composantes

Complexité de la gestion des composantes

- complexité de $\text{reunir}(i, j) = \min(|CC[i]|, |CC[j]|)$
- dans le pire des cas on a 2^k sommets et on les réunit 2 par 2, 4 par 4, etc..., et la complexité est
$$n/2 + n/4 + \dots + n/(2^i) = \frac{n}{2} \times \lg(n) = O(n \lg n)$$

La complexité de la réunion de tous les sommets de G est en $O(|S| \lg |S|)$

Kruskal : complexité

Fabrication des composantes connexes de G'

- réunion de $|S|$ singleton en un unique ensemble de taille $|S|$: $\Theta(|S| \lg |S|)$

Gestion du tas T

- on extrait les $|A|$ éléments du tas : $\Theta(|A| \lg |A|)$

Algorithme de Kruskal

- $\Theta(|S| \lg |S|) + \Theta(|A| \lg |A|) + \Theta(|A|) = \Theta(|A| \lg |A|)$ car $|A| \geq |S| - 1$ (graphe connexe)

ACM : synthèse



Deux algorithmes basés sur les mêmes propriétés

- algorithmes de **Prim** et **Kruskal**
- complexité en $\Theta(|A| \lg |A|)$ en utilisant un tas

Algorithmes dérivés

- recherche du deuxième ACM
- mise à jour de l'ACM après ajout ou suppression d'une arête

