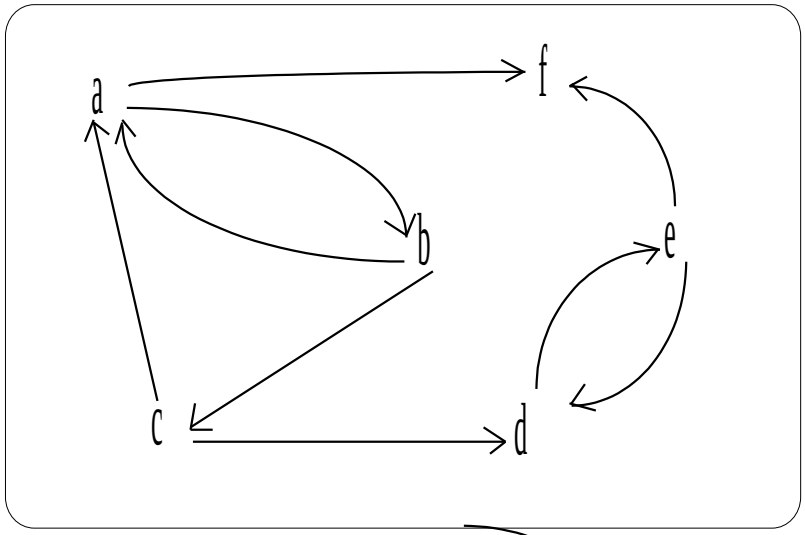
NFP136 – COURS 5 : INTRODUCTION AUX GRAPHES

PLAN

- Généralités et définitions
- Représentation d'un graphe
- Exploration d'un graphe
- Application : composantes connexes
- Automates, graphes d'état (facultatif)

5.1 GENERALITES ET DEFINITIONS

5-1-1 GRAPHES ORIENTES



EXEMPLE: **G1 = (X1 , U1)**

X1 = {<u>sommets</u>} = {a,b,c,d,e,f}

U1 = {<u>arcs</u>} = {(a,b),(b,a),(b,c),(c,a),(c,d),(a,f),
(e,f),(d,e),(e,d)}

G : $X \rightarrow P(X)$

$x \rightarrow G(x) =$ {successeurs de x}

EXEMPLE (suite) **G1(b)={a,c} G1(f)={Ø} G1(d)={e}**

Chemin : suite d'arcs telle que l'extrémité terminale d'un arc coïncide avec l'extrémité initiale de l'arc suivant

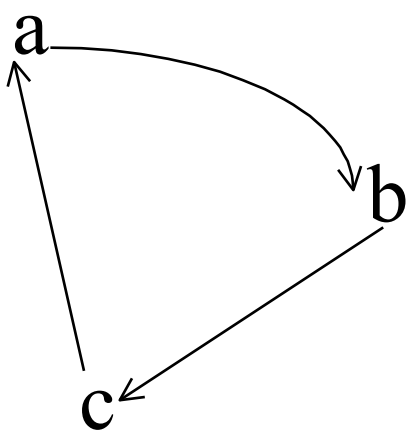
---

EXEMPLE (suite) ((a,b),(b,c),(c,d)) ou (a,b,c,d)

Circuit : chemin (sans répétition d'arcs) dont le premier sommet coïncide avec le dernier

---

EXEMPLE (suite) **(a,b,c,a) : circuit de G1**

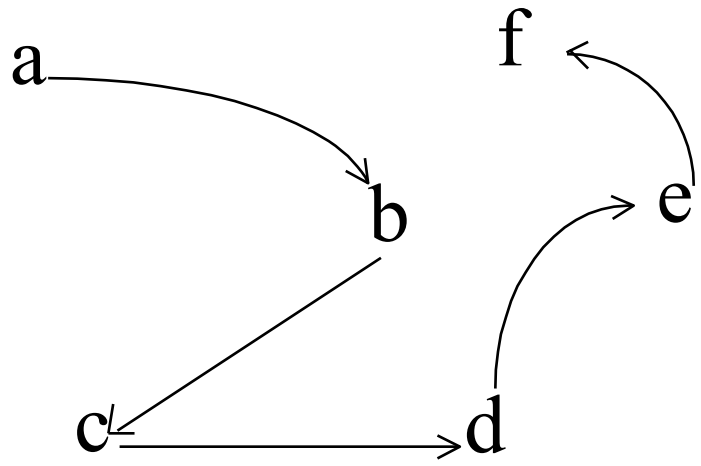


un circuit

Chemin hamiltonien : chemin qui passe une fois et une seule par chaque sommet.

Circuit hamiltonien : défini de façon similaire.

EXEMPLE (suite) **(a,b,c,d,e,f) :** chemin hamiltonien



un chemin hamiltonien

5-1-2 UTILISATION DES GRAPHES

- **Modélisation, représentation de problèmes :**

Exemple : plan de ville, réseaux routiers ou ferroviaires, arbre généalogique, états d'un système (automates – cf compléments facultatifs à la fin), etc.

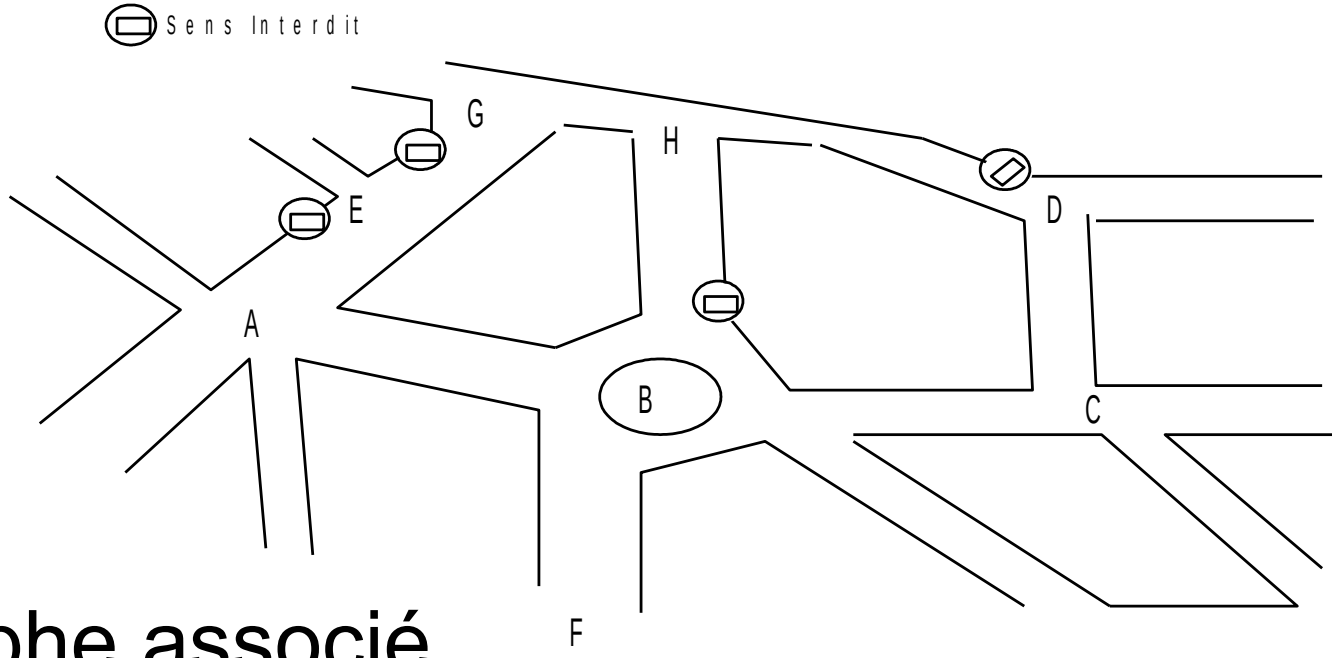
- **Résolution de problèmes (algos. de graphes) :**

Exemple : plus court chemin (GPS), ordonnancement (gestion de projets), flots (débit dans un réseau), etc.

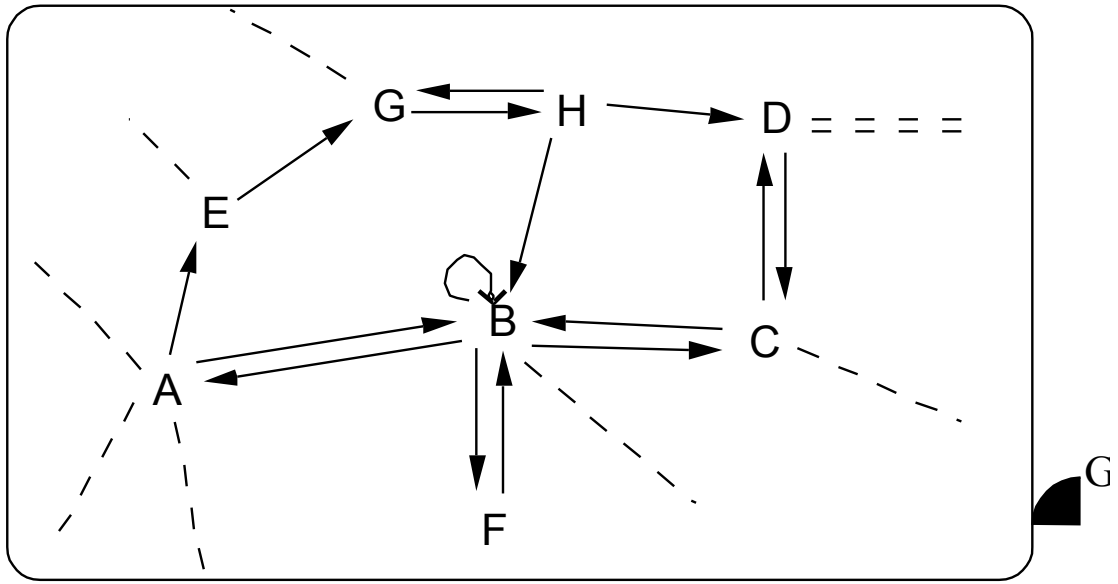
- **Outils :**

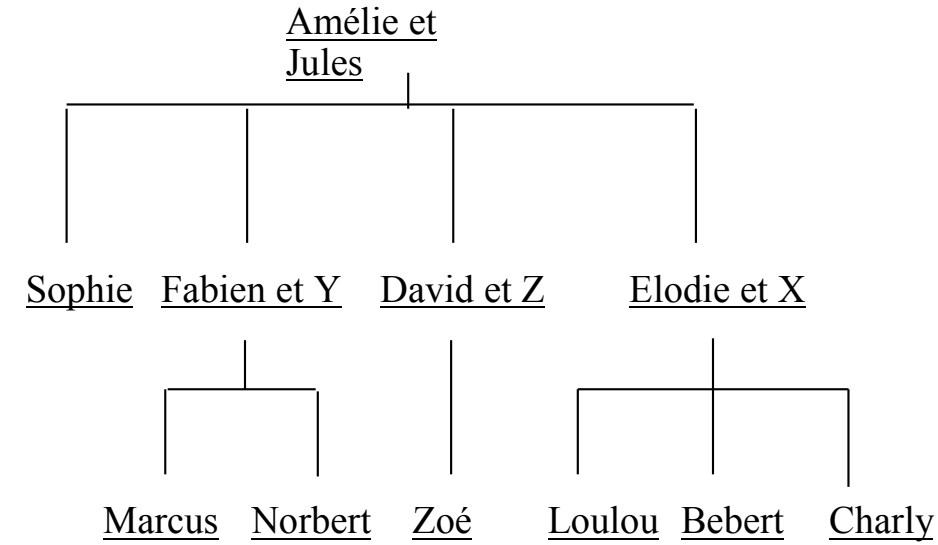
Exemple : structures de données, etc.

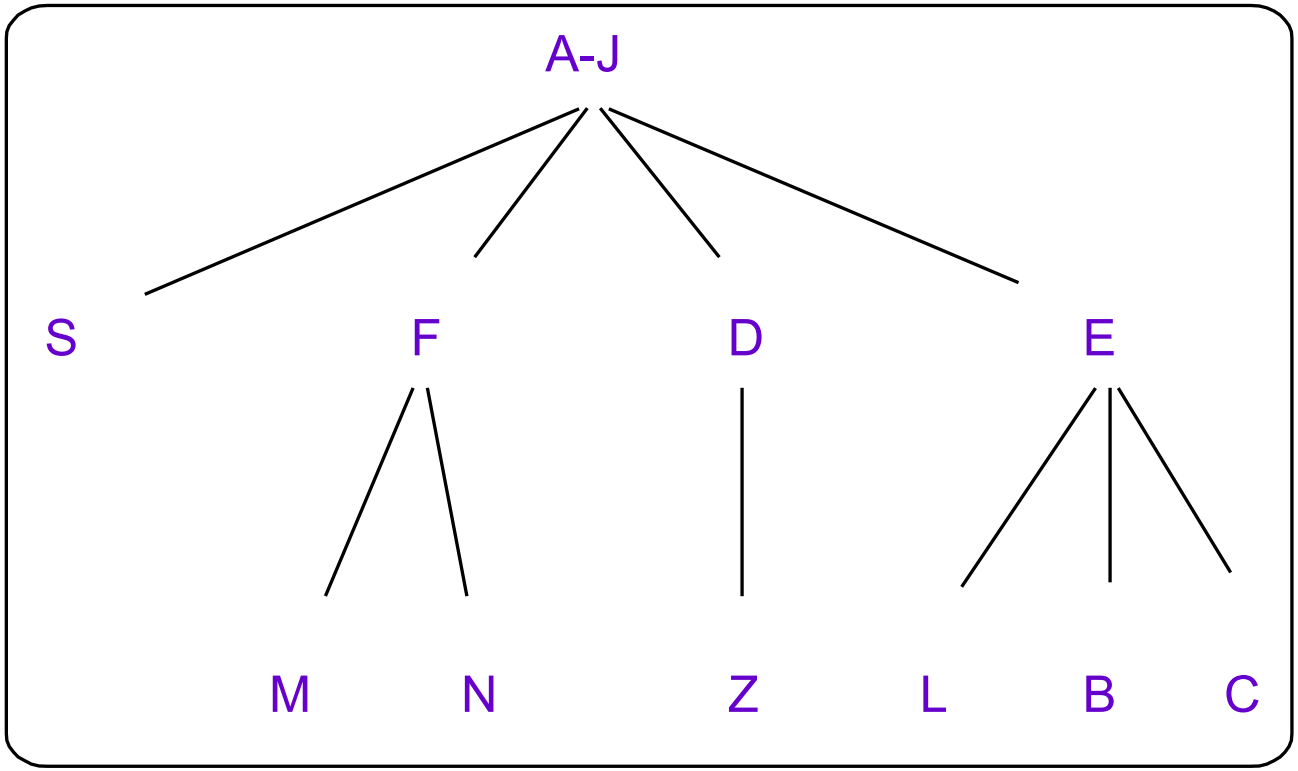
- **Etc.**



Graphe associé





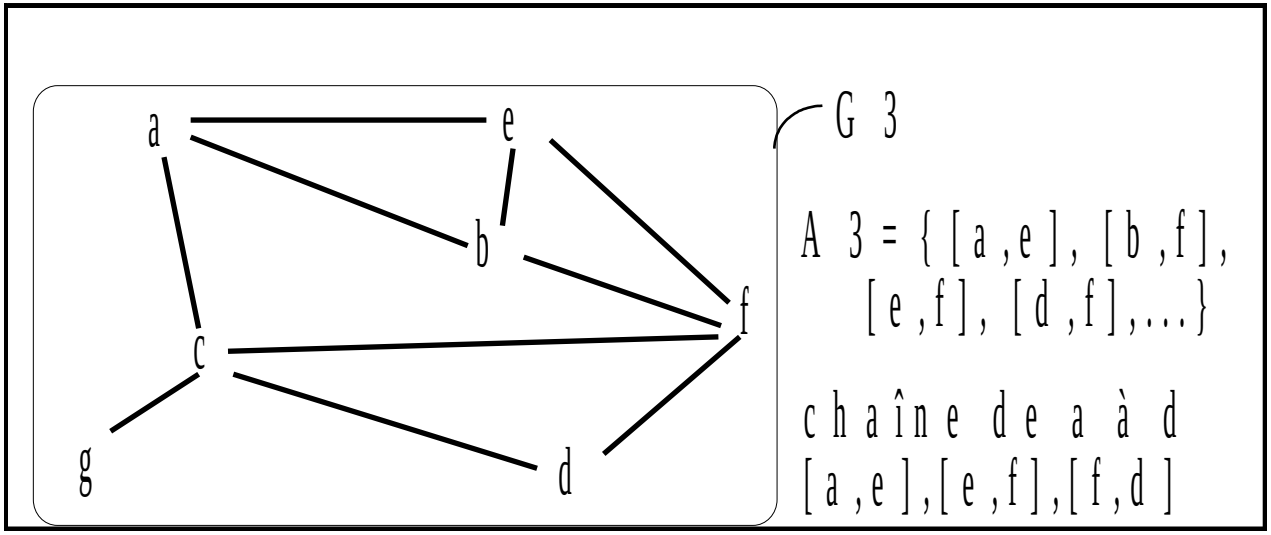


Graphe associé

5-1-3 GRAPHES NON ORIENTES

G= (X,A), où A est un ensemble d'arêtes

Arête : arc "sans orientation"



Chaîne : suite d'arêtes telle que toute arête de la suite a une extrémité commune avec l'arête précédente (sauf la première) et l'autre avec l'arête suivante (sauf la dernière).

Cycle : chaîne (sans arêtes répétées) dont les 2 extrémités coïncident (circuit « non orienté »)

EXEMPLE (suite) **cycle de G3 : [aefba]**

Les notions de chaîne et cycle peuvent aussi se définir aussi dans un graphe orienté (on ne tient plus compte de l'orientation)

EXEMPLE (suite) **cycle de G1 : (bacb)**

Connexité : un graphe est connexe si toute paire de sommets est reliée par une chaîne

---

EXEMPLE (suite) **G1 et G3 connexes**

---

degré **sommet $x \in X$**

d(x) = nombre de voisins de x

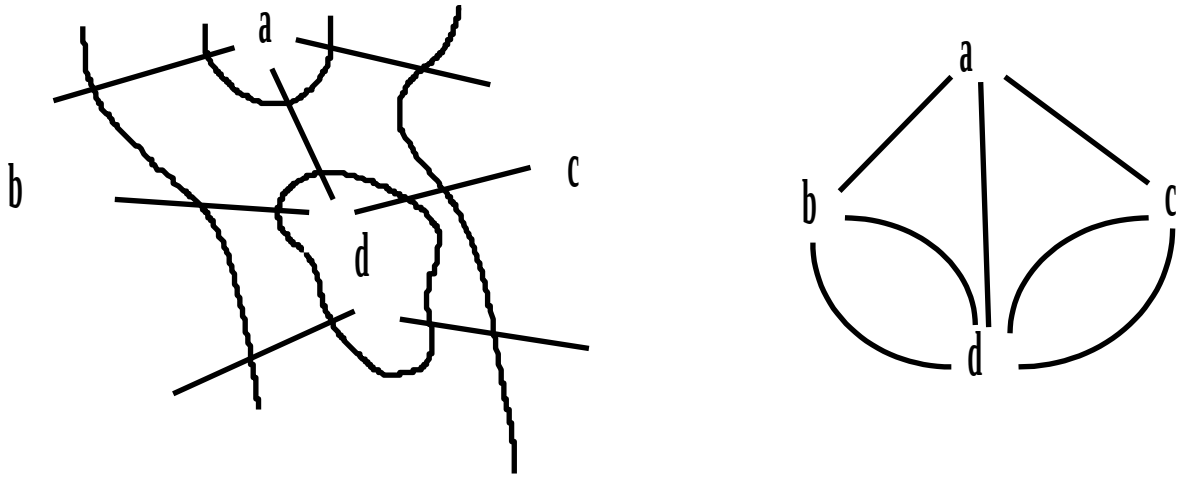
Remarque : la somme des degrés vaut 2m (m = nombre d'arêtes).

EXEMPLE (suite) **dans G3, d(c) = 4 et d(b) = 3**

Chaîne eulérienne : chaîne qui passe une fois et une seule par chaque arête

Théorème d'Euler

Un multigraphe connexe admet une chaîne eulérienne si et seulement si le nombre de sommets de degré impair est 0 ou 2



1766 : Les 7 ponts de Koenigsberg

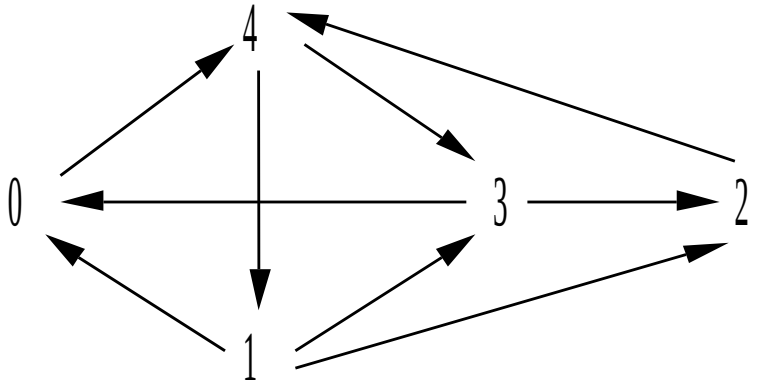
Chaîne eulérienne vs chaîne hamiltonienne :

- Chercher une chaîne eulérienne revient à calculer et à tester tous les degrés du graphe (théorème d'Euler),
- Chercher une chaîne hamiltonienne peut être très difficile dans le cas général.

==> DEUX PROBLEMES « PROCHES », DE COMPLEXITES TRES DIFFERENTES !

5.2 REPRESENTATION D 'UN GRAPHE

Exemple :

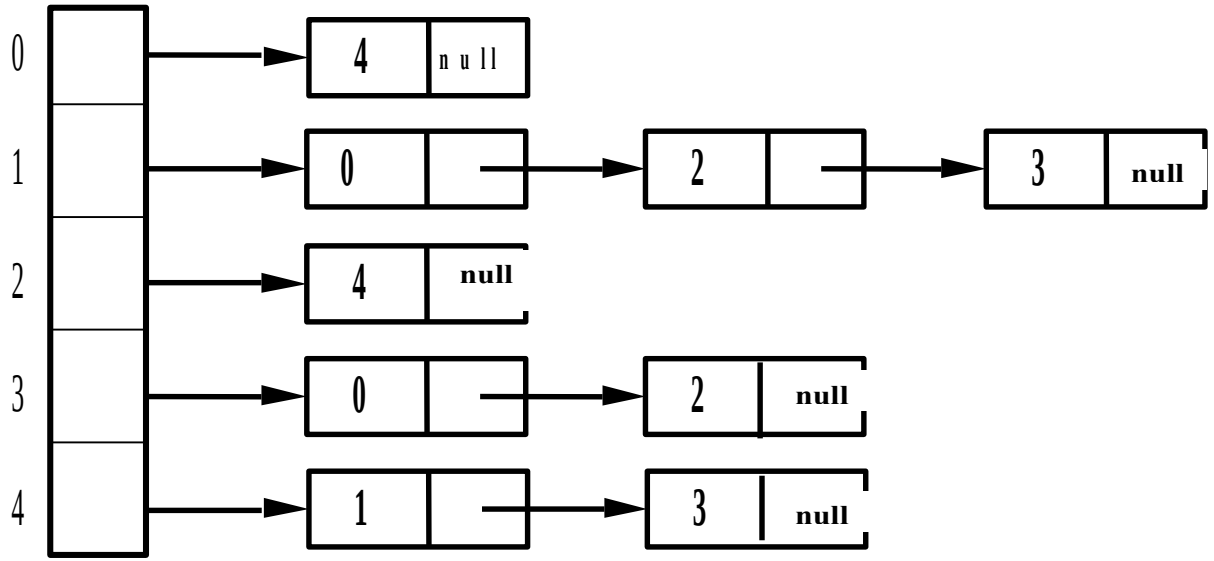


5-2-1 UNE MATRICE

	0	1	2	3	4
0	0	0	0	0	1
1	1	0	1	1	0
2	0	0	0	0	1
3	1	0	1	0	0
4	0	1	0	1	0

- balayages (successeurs, prédécesseurs),
- place mémoire importante,
- souple (évolution du graphe),
- accès direct+calculs matriciels possibles.

5-2-2 TABLEAU ADRESSANT DES LISTES CHAÎNEES DE SUCCESSEURS



Variante = liste de couples

- place mémoire optimisée,
- souple (évolution du graphe),
- peu pratique à manipuler (ex. : nombre de successeurs ou de prédécesseurs ?).

5-2-3 Une implémentation Java utilisant un tableau de listes de successeurs (illustration à lire)

D'abord une classe Sommet (qui utilise la classe Liste d'entiers) :

```
public class Sommet {
 private int id;
 private Liste listeSuccesseurs=null;
            //pointe vers la liste des numéros des successeurs du sommet

public Sommet(int id) {
 this.id=id;
}
public int idSommet() {
 return id;
}
public Liste successeurs() {
 return listeSuccesseurs;
}
```

Suite de la classe Sommet :

```
public void ajoutSuccesseur(int s) {
 listeSuccesseurs = new Liste(s, listeSuccesseurs);
}

public void affichage() {
 System.out.println("Sommet "+ id);
 System.out.print("Successeurs : ");
 Liste L=listeSuccesseurs;
 while(L != null){
     System.out.print (L.tete()+ " ");
     L=L.queue();
 }
 System.out.println("");
}
}//fin classe Sommet
```

Voici la classe Graphe :

```
public class Graphe{
 private Sommet[] tabSommets;

public Graphe(int nbSommets){
 //crée un graphe, étant donné un nombre de sommets
 //les sommets sont alors numérotés de 0 à nbSommets-1
 tabSommets = new Sommet[nbSommets];
 //creation des Sommets
 for (int i=0; i< tabSommets.length; i++){
    tabSommets[i]= new Sommet(i); //sommet numéro i
 }
}
```

Voici un deuxième constructeur pour la classe Graphe :

```
public Graphe(int[] tabSommetsNumerotes) {
//crée un graphe, étant donné un tableau de numéros de sommets
 tabSommets = new Sommet[tabSommetsNumerotes.length];
 //création des sommets
 for (int i=0; i< tabSommets.length; i++) {
    tabSommets[i]=new Sommet(tabSommetsNumerotes[i]);
                          //numéro du ième sommet
 }
}
```

Suite des méthodes de la classe Graphe :

```
public int trouverIndiceSommet (int id) {
 //trouve dans tabSommets l'indice du sommet de numéro id
 for (int i=0; i< tabSommets.length; i++) {
     if (tabSommets[i].idSommet()==id) return i;
 }
 return (-1); //échec : pas de sommet ayant ce numéro !
}
public void ajoutArc(int x, int y) {
 //d'abord, trouver l'indice de x
 int indice_x = trouverIndiceSommet(x);
 //ajouter y comme successeur du sommet tabSommets[indice_x];
 tabSommets[indice_x].ajoutSuccesseur(y);
}
public void affichage() {
 System.out.println("Début graphe");
 for (int i=0; i< tabSommets.length; i++) {
     tabSommets[i].affichage();
 }
 System.out.println("Fin graphe");
}
```

Mise en commun sur un exemple :

```
public class testGraphe {
   public static void main (String[] args) {
     int[] sommets= {5, 10, 15};
     Graphe G=new Graphe(sommets);
     G.ajoutArc(5,10);
     G.ajoutArc(5,15);
     G.affichage();
   }//fin main
}
```

```
> java testGraphe
Début graphe
Sommet 5
Successeurs : 15 10
Sommet 10
Successeurs :
Sommet 15
Successeurs :
Fin graphe
```

<== SORTIE

Mise en commun sur un autre exemple :

```
import java.io.*;
public class testGrapheBis {
    public static void main (String[] args) {
        Graphe G=new Graphe(6);
        G.ajouteArc (0,4);
        G.ajouteArc (0,1);
        G.ajouteArc (4,1);
        G.ajouteArc (4,3);
        G.ajouteArc (1,3);
        G.ajouteArc (1,5);
        G.ajouteArc (3,5);
        G.ajouteArc (2,3);
        G.ajouteArc (2,5);
        G.affichage();
    }
}
```

SORTIE :

```
> java testGrapheBis
Début graphe
Sommet 0
Successeurs : 1 4
Sommet 1
Successeurs : 5 3
Sommet 2
Successeurs : 5 3
Sommet 3
Successeurs : 5
Sommet 4
Successeurs : 3 1
Sommet 5
Successeurs :
Fin graphe
```

5.3 EXPLORATION (PARCOURS) D'UN GRAPHE

Problématique : étant donné un sommet x d'un graphe G à m arêtes, quels sommets de G peut-on atteindre à partir de x (= descendants de x) ?

Il existe plusieurs « stratégies » (= algorithmes) pour répondre à cette problématique, et effectuer ainsi un « parcours » du graphe à partir de x :

- Stratégie aveugle/naïve (peu efficace),
- Stratégie de *parcours en largeur* (d'abord),
- Stratégie de *parcours en profondeur* (d'abord).

Marquage des descendants d'un sommet : parcours en largeur

On marque tous les successeurs d'un sommet avant de traiter les autres sommets (on utilise pour cela une file FIFO).

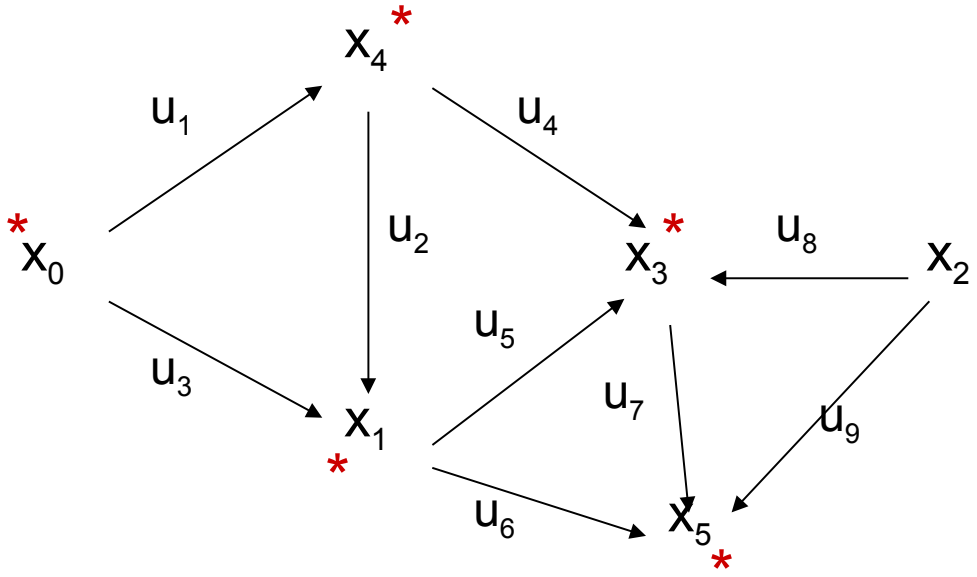
Parcours en largeur (nécessite l'utilisation d'une file) :

```
public void parcoursLargeur(int x, boolean[] marque, int[] pere, int[] Long) {
  File F=new File(tabSommets.length);
  int ind_x=trouverIndiceSommet(x);
  Long[ind_x]=0;
  marque[ind_x]=true;
  F.enfiler(ind_x);

  while(!F.estVide()) {
    ind_x=F.defiler();
    Liste L=tabSommets[ind_x].successeurs();
    while(L!=null) {
      int ind_y= trouverIndiceSommet(L.tete());
      if(!marque[ind_y]) {
        marque[ind_y]=true;
        pere[ind_y]=ind_x;
        Long[ind_y]=Long[ind_x]+1;
        F.enfiler(ind_y);
      }
      L=L.queue();
    }
  }//fin while
}//fin parcoursLargeur
```

Complexité *(pire cas)* **: O(m)**

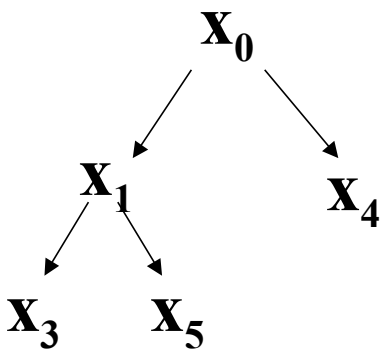
EXEMPLE



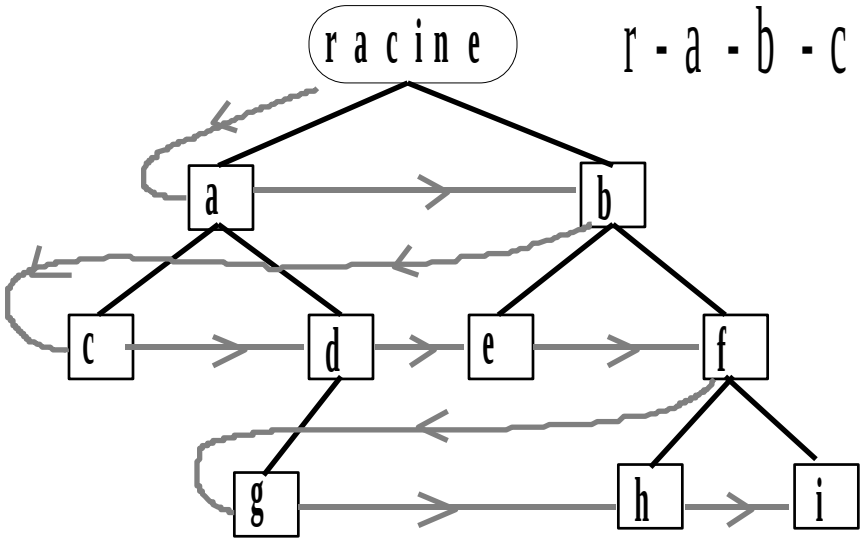
	x_1	x_2	x_3	x_4	x_5
père	x_0		x_1	x_0	x_1
dist(x)	**1**		**2**	**1**	**2**

REMARQUE

arborescence "en largeur" parcourue (arcs utilisés lors des marquages du parcours), qui donne les plus courts chemins issus de x_0



Parcours en largeur d'un arbre :



r - a - b - c - d - e - f - g - h - i

==> Niveau par niveau, de gauche à droite

Marquage des descendants d'un sommet : parcours en profondeur

On avance tant qu'on peut dans le graphe, et le parcours des chemins non explorés se fait lorsqu'on ne peut plus avancer.

Parcours en profondeur (version récursive) :

```
public void parcoursProfondeur(int x, boolean[] marque, int[] pere) {
 int ind_x=trouverIndiceSommet(x);
 marque[ind_x]=true;
 Liste L=tabSommets[ind_x].successeurs();
 while (L!=null) {
     int y=L.tete();
     int ind_y=trouverIndiceSommet(y);
     if (!marque[ind_y]) {
         pere[ind_y]=ind_x;
         parcoursProfondeur(y,  marque, pere);
     }
     L=L.queue();
 }//fin while
}//fin parcoursProfondeur
```

Complexité *(pire cas)* **: O(m)**

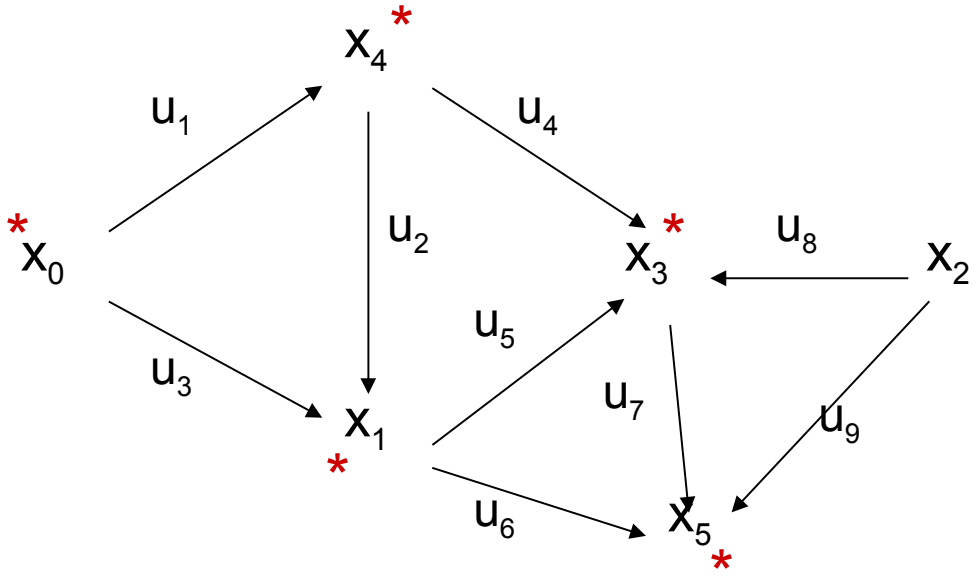
Parcours en profondeur (version itérative, qui utilise une pile) :

Pseudo-code :

```
Choisir un sommet arbitraire s (sommet initial)
Empiler et marquer s
Tant que la pile n'est pas vide faire
        –   Rechercher, parmi les voisins du sommet x qui se trouve
            sur le haut de la pile, un voisin y non marqué,
        –   Si y existe, alors empiler et marquer y (x est défini
            comme le père/prédécesseur de y dans ce parcours),
        –   Sinon, dépiler x.
```

Complexité *(pire cas)* **: O(m) également**

EXEMPLE

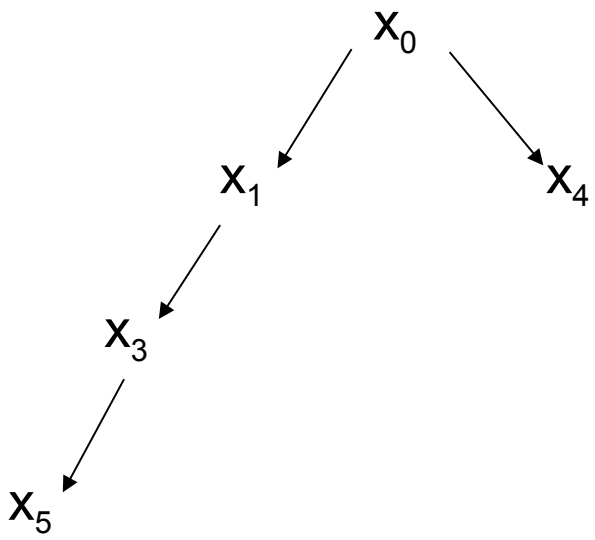


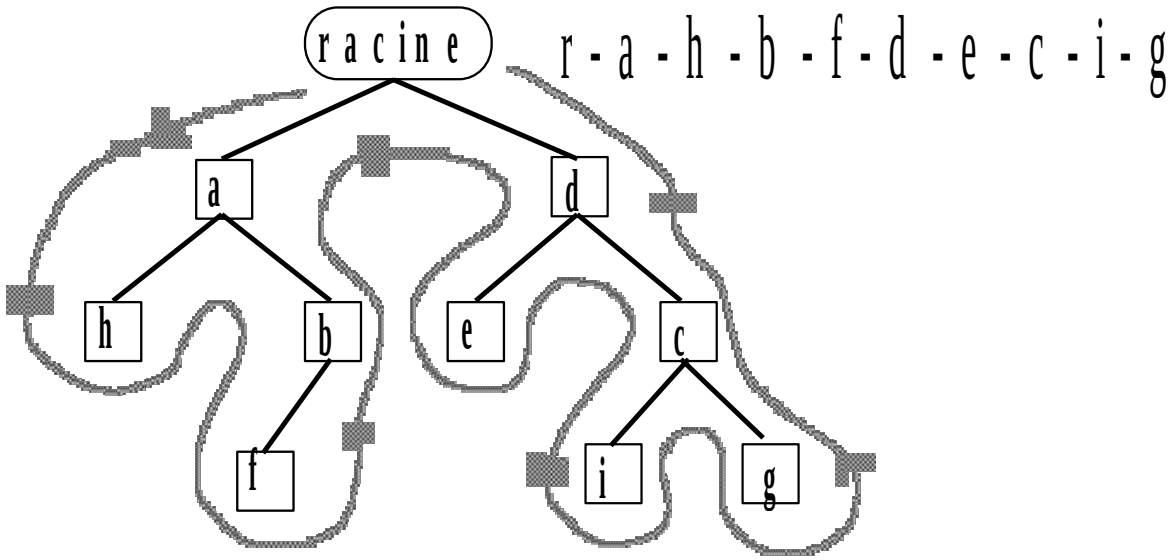
	x_1	x_2	x_3	x_4	x_5
père	x_0		x_1	x_0	x_3

EXEMPLE

	x_1	x_2	x_3	x_4	x_5
père	x_0		x_1	x_0	x_3

arborescence "en profondeur" parcourue :



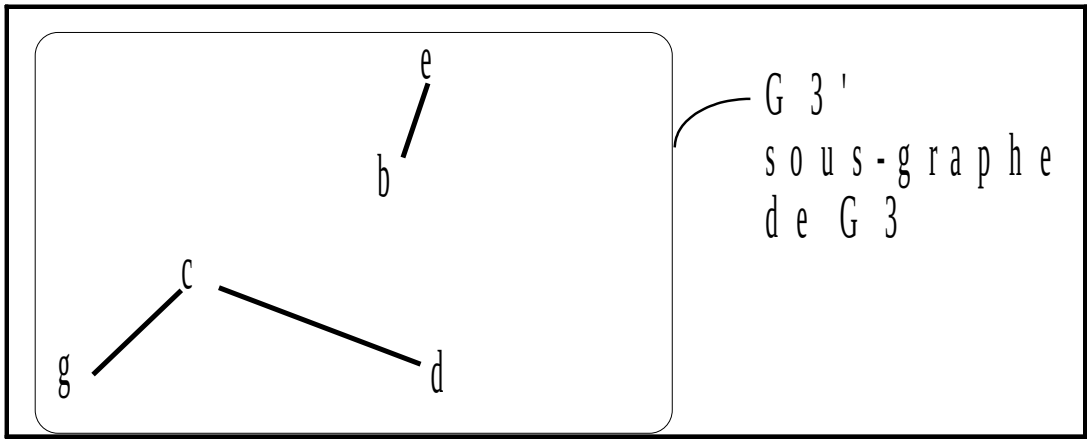


Parcours en profondeur d'un arbre : on explore les branches de gauche à droite (programmation de l'exploration facile, récursivement ou à l'aide d'une pile)

5.4 COMPOSANTE CONNEXE

Composante connexe de G : ensemble de tous les sommets accessibles par une chaîne à partir d'un sommet donné

EXEMPLE



G3' a 2 composantes connexes : {b,e} et {c,d,g}

DETERMINATION DES COMPOSANTES CONNEXES D'UN GRAPHE

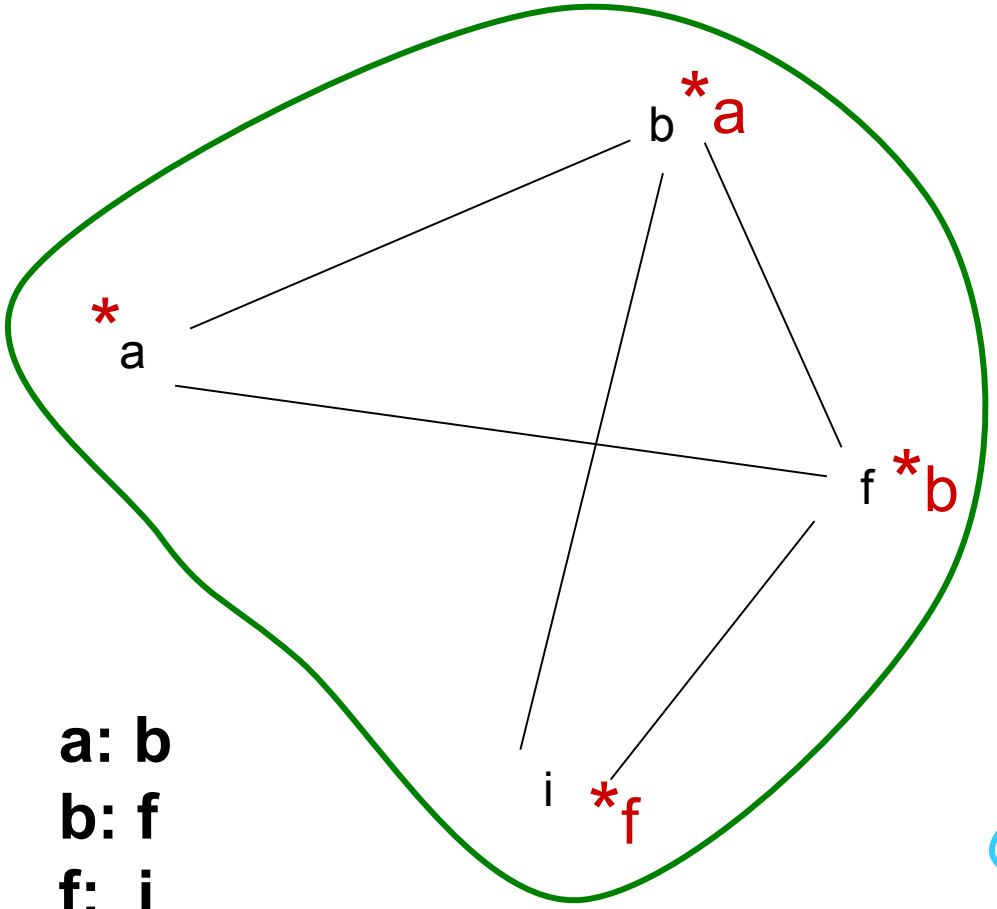
Entrée : graphe G = (X,A)

(G est non orienté ou on ne tient pas compte de l'orientation)

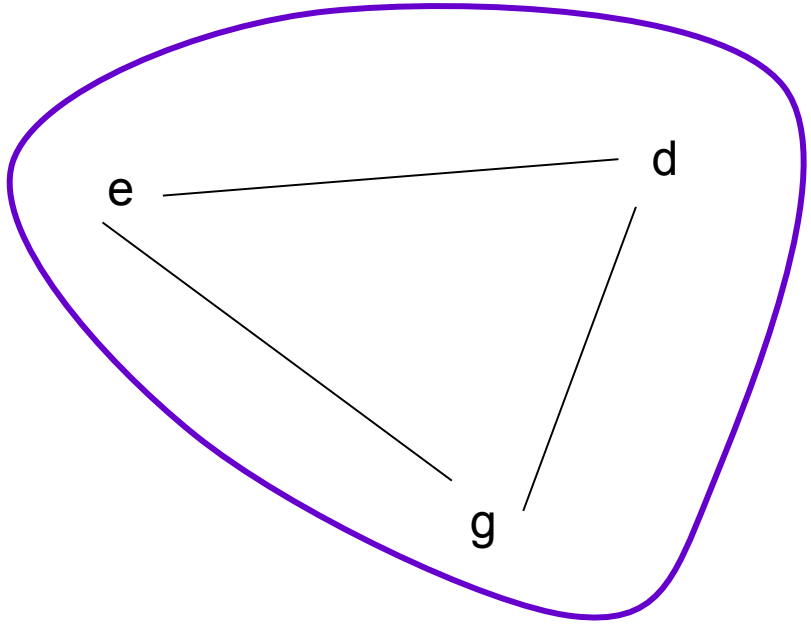
Sortie : liste des composantes

<u>Principe</u> :

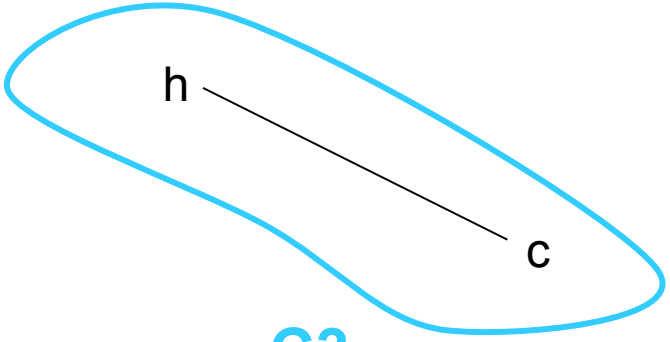
- Déterminer une composante connexe C à l'aide d'un parcours (en largeur ou en profondeur), en partant d'un sommet quelconque,
- Retirer C du graphe G et recommencer.



C1

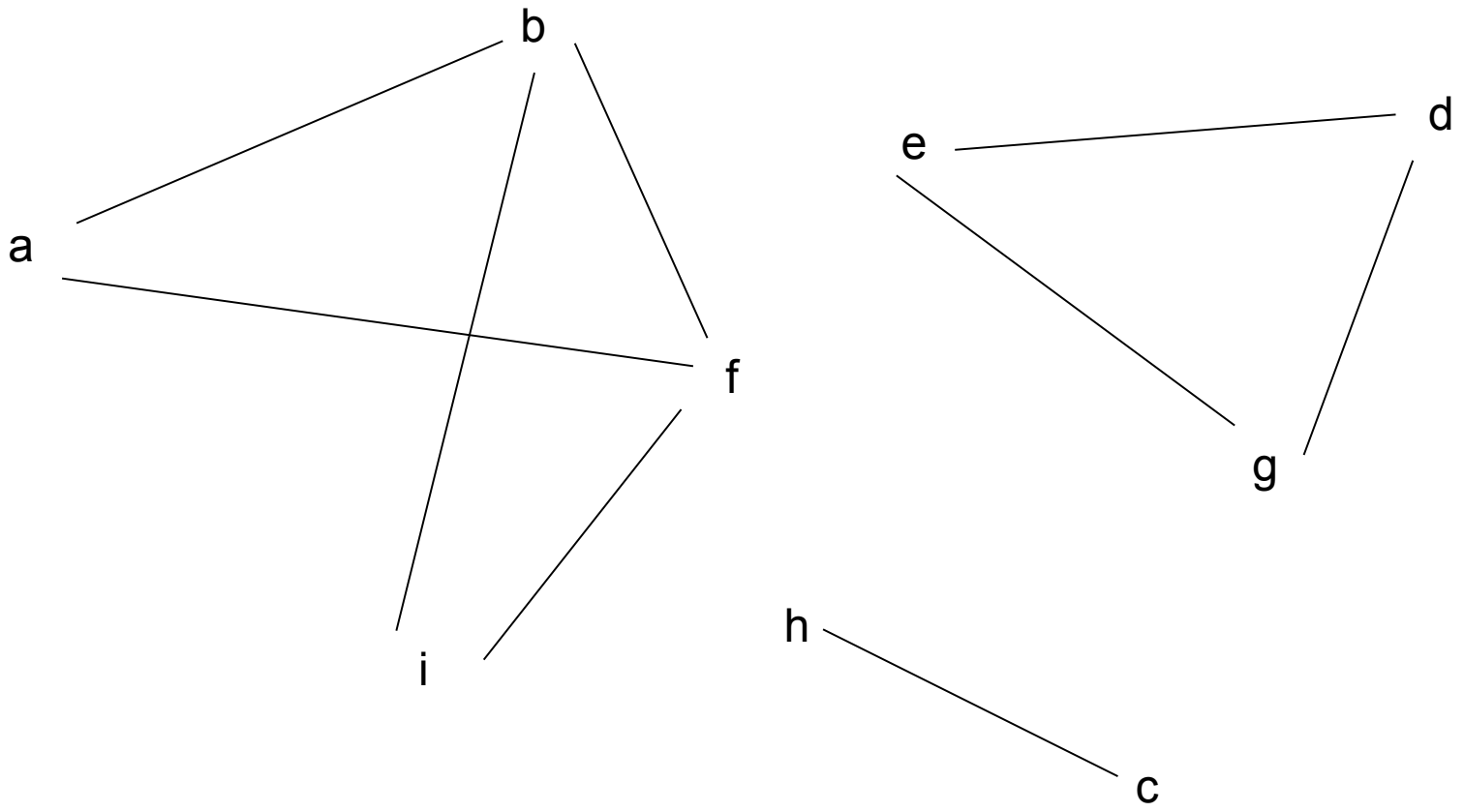


C2



C3

a: b
b: f
f: i
i: -
f: -
b: -
a: -



==> Complexité = O(nombre d'arêtes)

NFP136 : COMPLEMENTS SUR AUTOMATES, GRAPHES D'ETAT ET LANGAGES REGULIERS

PLAN

- Automates : définitions et usages (modélisation des états d'un système)
- Langages réguliers, automates, et expressions régulières

AUTOMATES : DEFINITIONS, USAGES

AUTOMATES

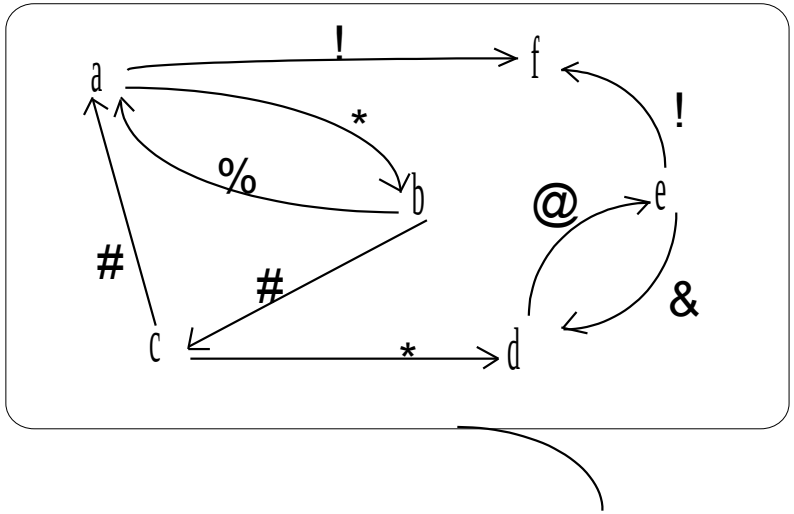
Un automate peut être vu comme un graphe orienté dont les arcs sont valués par des symboles, appelés **étiquettes**.

Les sommets d'un automate sont aussi appelés **états**, et sont partitionnés en trois ensembles :

- les états initiaux,
- les états transitoires,
- les états terminaux.

AUTOMATES ET MOTS

Les étiquettes d'un automate appartiennent à un ensemble donné de symboles appelé « alphabet », et un chemin de tout état initial vers tout état terminal définit alors un **mot** sur cet alphabet, en concaténant les symboles associés aux arcs de ce chemin dans l'ordre où ils sont parcourus.



Alphabet={#,%,*,!,@,&}
Etats initiaux={a}
Etats terminaux={f}
Le chemin a, b, a, b, c, d, e, f correspond au mot *%*#*@!

QUELQUES REMARQUES

1. En réalité, une étiquette peut être « vide » (c'est-à-dire ne correspondre à aucun symbole de l'alphabet).

2. Ainsi, on peut toujours se ramener au cas où il y a un unique état initial et un unique état terminal.

MODELISATION D'UN SYSTEME :

On peut modéliser la dynamique d'un système à événements discrets à l'aide de son *graphe des états* qui, dans son expression la plus simple, est un automate. Ainsi, chaque sommet (ou état) de l'automate correspond à un état du système, et les arcs de l'automate représentent les **transitions** possibles entre états du système.

Par exemple : si états={« inscription à l'UE » (E1, état initial), « examen session 2 » (E2), « validation de l'UE » (E3, état final), « non validation de l'UE » (E4, état final)}, alors l'automate représentant la dynamique des états du système « Parcours dans l'UE » a pour transitions (E1,E3), (E1,E2), (E2, E3), (E2, E4) (avec les étiquettes associées).

AUTOMATES, LANGAGES REGULIERS, ET EXPRESSIONS REGULIERES

LANGAGES REGULIERS ET AUTOMATES

Etant donné un alphabet, un **langage** (formel) est un ensemble de mots formés sur cet alphabet.

Etant donné un automate, on peut définir le langage de cet automate (on dit aussi «le langage **reconnu** par cet automate ») comme l'ensemble des mots obtenus en considérant tous les chemins entre les états initiaux et les états terminaux, et en concaténant pour chacun d'eux les symboles associés aux arcs de ce chemin dans l'ordre où ils sont parcourus.

Inversement, si un langage peut être reconnu par un automate fini (c'est-à-dire ayant un nombre fini de sommets), alors il est dit **régulier** (ou **rationnel**).

LANGAGES REGULIERS OU NON

Tous les langages ne sont pas réguliers : par exemple, le langage (= l'ensemble des mots) $\{a^n b^n$, pour $n>0\}$ ne peut pas être reconnu par un automate fini.

En revanche, tout langage fini (= contenant un nombre fini de mots) est trivialement régulier.

Pour décrire des langages (réguliers ou non), on utilise certaines conventions d'écriture et des parenthèses :

- La notation a* signifie un nombre indéfini, mais **positif ou nul**, de symboles a,
- La notation a+ signifie un nombre indéfini, mais **strictement positif**, de symboles a.

EXPRESSIONS REGULIERES

Un langage peut être « décrit » de différentes façons. Par exemple, l'ensemble des mots décrit par (a*b)* est le même que celui décrit par a*(a*b)*. Ces deux descriptions sont donc deux représentations différentes d'un même langage. Chacune de ces « descriptions », ou «représentations», s'appelle une « expression » : une expression qui représente un langage régulier s'appelle une **expression régulière**.

Le concept d'expression régulière est très utile/utilisé en informatique, pour tester la correspondance d'un « mot » (commande, instruction, paramètre, etc.) avec un « motif » de référence (ainsi, la ligne de commande « javac *.java » permet de compiler tous les fichiers .java du répertoire courant).