Graphes

Eric Goubault

CEA/Saclay

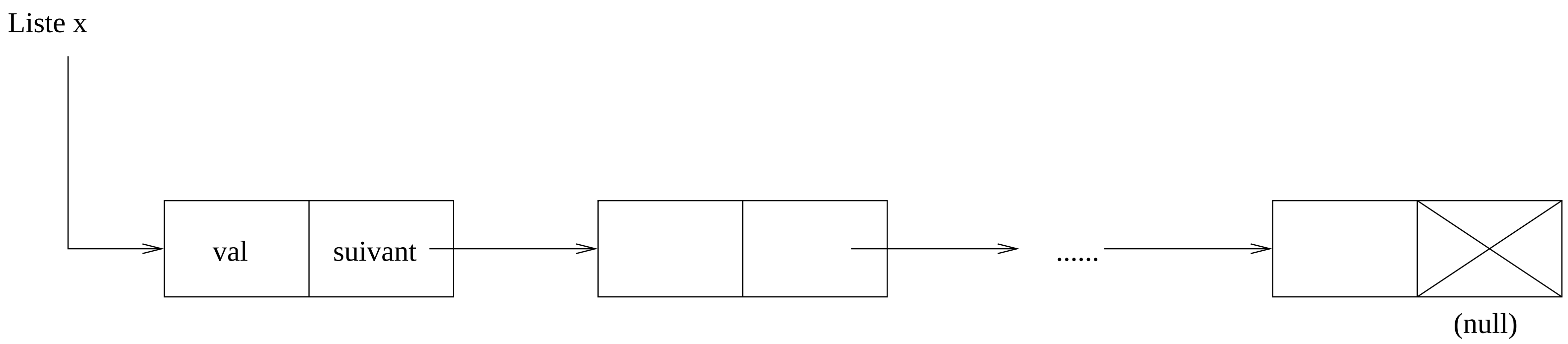
`Eric.Goubault@cea.fr`

`http://www.di.ens.fr/~goubault`

Rappel: listes

```
class Liste {
  int val;
  Liste suivant;
  Liste(int x, Liste ls) { val = x; suivant = ls; }
}
```

Listes



Remarques sur les “locations mémoire” et constructeurs

- `new Liste();` crée une adresse mémoire valide, pouvant contenir un champs entier `val` et un champs “pointeur” ou “référence” sur une case de type `Liste`
- `new Liste(x,l);` où `l` est une liste et `x` est un entier appelle le constructeur défini plus haut, construit une case, et range les bons éléments à leurs places
- `int tab[]=new int[3];` construit une référence sur un tableau à trois entiers (c’est à dire une zone mémoire contigue de trois entiers)

Remarques sur le passage de paramètres en JAVA

- Le passage de paramètre “par valeur” est la règle en JAVA; cad que,

  ```
  int f(int x) { return x++;}
  int g(int x) { int y = f(x); return x;}
  ```

  l’appel à `g(3)` renvoie `3` (et non `4`)

- Le passage de la valeur d’une référence = passage par référence, permet de modifier les objets:

  ```
  int f(int x[]) { return x[0]++; }
  int g(int x[]) { int y = f(x); return x[0]; }
  ```

  l’appel à `g({0,0})` renvoie `1` (et non `0`)

- Ou encore, l’appel à `g(l)` où `l` est une `Liste` permet à priori de modifier en place tous les éléments de la liste `l`.

Exercice de rappel...

Ecrire une méthode (de la classe `Liste`):

```
void mapincr(void)
```

qui applique "en place" la fonction incrémentation à chaque élément de la liste (sur laquelle la méthode s'applique)

Ou si vous préférez:

```
static void mapincr(Liste l)
```

(qui s'applique sur l'argument `l`)

Solution...

```
public void mapincr() {
  val++;
  if (suivant != null) suivant.mapincr(); }
```

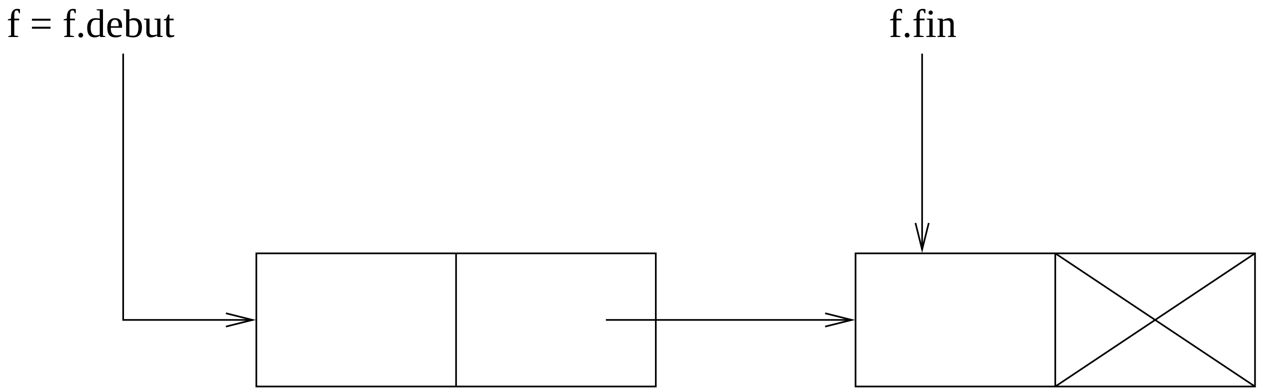
(attention une méthode ne s'applique qu'à un objet non nul!)

Files d'attente (ou "First In First Out")

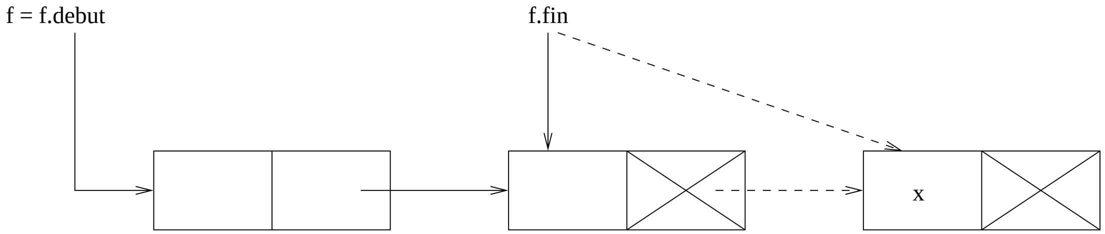
```
class FIFO {
  Liste debut, fin;
  FIFO() { debut = null; fin = null; }

  static void ajouter(FIFO f, int x) {
    if (f.fin == null) f.debut = f.fin = new Liste(x);
    else {
      f.fin.suivant = new Liste(x);
      f.fin = f.fin.suivant;
    }
  }
```

Exécution



Exécution



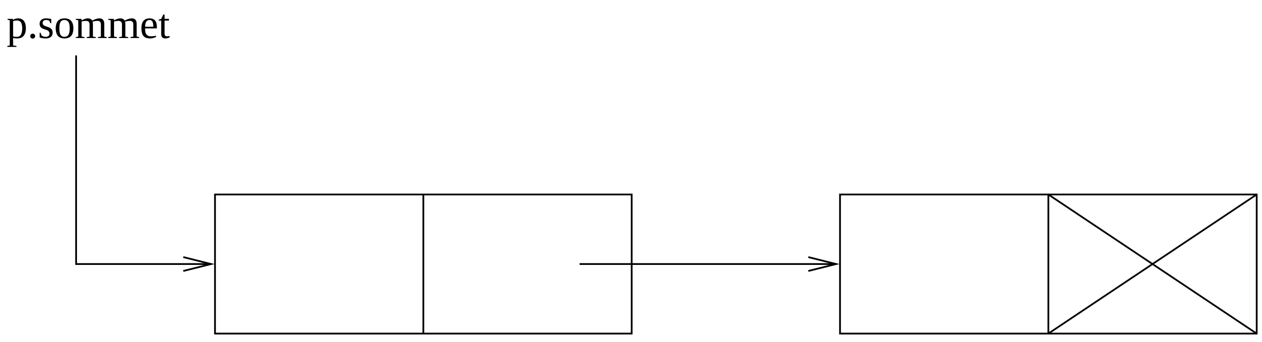
Files d’attente

```
  static int supprimer(FIFO f) {
    if (f.debut == null) throw new Error(``File Vide'');
    else {
      int res = f.debut.val;
      if (f.debut == f.fin) f.debut = f.fin = null;
      else f.debut = f.debut.suivant;
      return res;
    }
  }
}
```

Piles (ou “Last In First Out”)

```
class Pile {
  Liste sommet;
  Pile() { sommet = null; }
  static void empiler(int x, Pile p) {
    p.sommet = new Liste(x, p.sommet);
  }
  static int depiler(Pile p) {
    if (p.sommet == null) throw new Error(``Pile Vide'');
    int res = p.sommet.val;
    p.sommet = p.sommet.suivant;
    return res; } }
```

Exécution



Exécution

Graphes: définitions

- $G = (X, A)$, X ensemble de sommets, $A \subseteq X \times X$ ensemble des arcs de G
- $n = \mid X \mid$, $m = \mid A \mid$ (en $O(n^2)$ au pire)
- Utile pour la modélisation de nombreux phénomènes
  - Modélisation de relations
  - Discrétisation de formes unidimensionnelles - géométrie (connexité etc.)
  - Systèmes dynamiques (graphes orientés)

Programme récursif vers programme itératif

Exercice 1: Soit le programme récursif suivant:

```
class fibonnacci {
  static int calcule(int n) {
    if ((n == 0) || (n == 1)) return 1;
      return calcule(n-1)+calcule(n-2);
  }

  public static void main(String args[]) {
    System.out.println(calcule(Integer.parseInt(args[0])));
```

Quel est son graphe d'appel?

Solution...

```
fibo(5)
  fibo(4)
    fibo(3)
      fibo(2)
        fibo(1)
        ->1
        fibo(0)
        ->1
      ->2
      fibo(1)
      ->1
    ->3
```

Solution...

```
  fibo(2)
    fibo(1)
    ->1
    fibo(0)
    ->1
  ->2
->5
fibo(3)
  fibo(2)
    fibo(1)
    ->1
    fibo(0)
    ->1
  ->2
```

Solution...

```
      fibo(1)
      ->1
    ->3
  ->8
```

Programme récursif vers programme itératif

- Exercice 2: Quel est le problème? Donner un programme récursif équivalent, plus efficace
- Exercice 3: Donner un programme itératif équivalent

Solution...

Exercice 2: replier le graphe sur les mêmes appels en mémorisant tout ce qui a déjà été calculé.

```
class fibonnacci3 {
  static int memo[];
  static int calcule(int n) {
    if (memo[n-1] == 0)
      memo[n-1] = calcule(n-1); // calcule(n-2) fait!
    return memo[n-1]+memo[n-2]; }
  public static void main(String args[]) {
    int n = Integer.parseInt(args[0]);
    if (n < 2) { System.out.println(1); return; }
    memo = new int[n]; memo[0] = 1; memo[1] = 1;
    System.out.println(calcule(n)); } }
```

Exécution

```
[goubault@cygne13 Tronc-commun]$ time java fibonnacci 40
165580141
5.440u 0.040s 0:05.56 98.5% 0+0k 0+0io 2840pf+0w
[goubault@cygne13 Tronc-commun]$ time java fibonnacci3 40
165580141
0.110u 0.030s 0:00.15 93.3% 0+0k 0+0io 2840pf+0w
```

Solution plus efficace...

On a besoin que d'une mémoire en fait (fenêtre de 2 arguments):

On remarque que la suite de Fibonnacci peut s'écrire également, $(u(n+2), u(n+1)) = (u(n+1) + u(n), u(n+1))$. On utilisera:

```
public class pair {
    public int x1;
    public int x2;
    public static pair makepair(int i,int j) {
        pair x=new pair();
        x.x1=i;
        x.x2=j;
        return(x); } }
```

Solution plus efficace...

```
public class fibonnacci5 {
  static pair calcule(int n) {
    int res;
    pair res2;
    if (n==0) return(pair.makepair(1, 0));
    if (n==1) return(pair.makepair(1, 1));
    res2=calcule(n-1);
    res=res2.x1+res2.x2;
    res2=pair.makepair(res,res2.x1);
    return(res2); }

  public static void main (String[] args) {
    System.out.println(calcule(Integer.parseInt(args[0])).x
```

Graphe d'appel

```
fibo(5)
  fibo(4)
    fibo(3)
      fibo(2)
        fibo(1)
        ->1
      ->2
    ->3
  ->5
->8
```

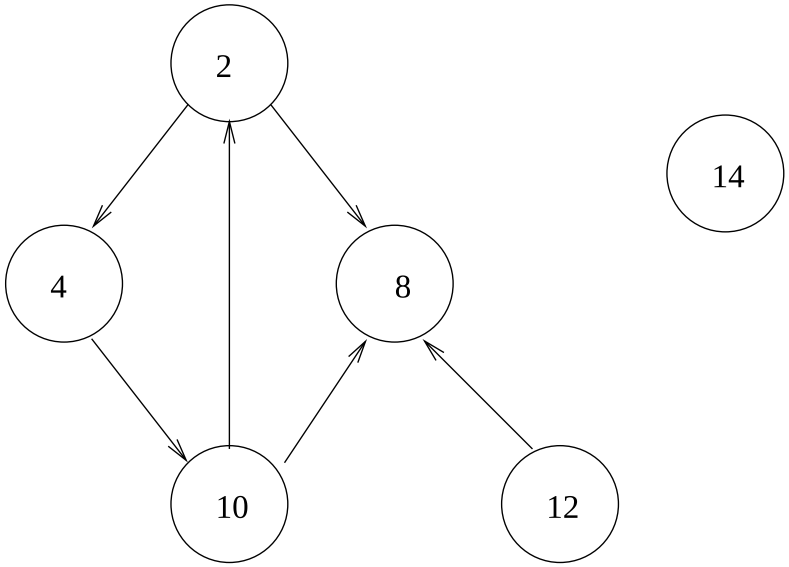
Version itérative de la version efficace

```
class fibonnacci6 {
  static int calcule(int n) {
  int u, v, w;
  u = v = w = 0;
  for (int i=0; i<=n; i++) {
    if (i == 0) { u = 0; v = 1; }
    else if (i == 1) { u = 1; v = 1; }
    else { w = u; u = v; v = w+v; }
  }
  return v; }

  public static void main(String args[]) {
    System.out.println(calcule(Integer.parseInt(args[0])));
```

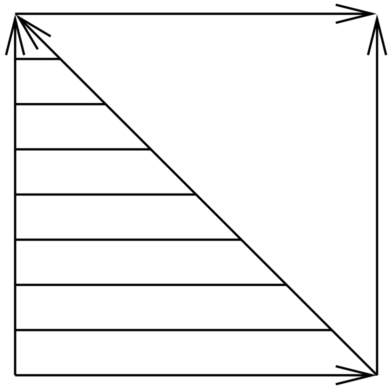
Exemple et généralisations

- graphes valués
- multigraphes, hypergraphes
- ensembles simpliciaux etc.



Ensembles (semi-)simpliciaux

- $M = (M_n, \{d_i^n : M_n \to M_{n-1} \mid 0 \leq i \leq n\})_{n \in N}$
- Si $M = (M_0, M_1)$ alors graphe
- "Modélisation" des espaces topologiques



Représentation des graphes

- Matrice d'adjacence $M_{i,j} = \begin{cases} 0 & \text{si } (i,j) \notin E \\ 1 & \text{si } (i,j) \in E \end{cases}$
- M est symétrique si le graphe est non-orienté (auquel cas on dit arête au lieu d'arc)
- Représentation en n^2

Implémentation des graphes orientés en JAVA

```
class GraphMat {
  boolean[][] m;
  GrapheMat(int n) {
    m = new boolean[n][n],
  }
}
```

Représentation par liste de successeurs

En fait: représentation creuse de la matrice d'adjacence:

- Pour chaque sommet, on représente la liste de ses successeurs
- On a un tableau de taille n pour les débuts de liste
- Deux variantes:
  - (en supposant un nombre de successeurs à peu près égal pour tous les sommets) `GrapheSucc` est un vecteur, le dernier élément est signalé par une valeur particulière `Omega=-1`
  - une vraie structure de liste `GrapheListe`
- Représentation en $m + n$

Implémentation en JAVA

```
class Graphe {
  Liste[] succ;
  Graphe(int n) { succ = new int(n); }
```

Passage d’une représentation à une autre

```
Graphe(GrapheMat g) {
  int n = g.m.length;
  succ = new Liste[n];
  for (int i=0; i<n; i++)
    for (int j=n-1; j>=0; j--)
      if (g.m[i][j])
        succ[i] = new Liste(j,succ[i]);
}
```

Passage d’une représentation à une autre

```
GrapheMat(Graphe g) {
  int n = g.succ.length;
  m = new boolean[n][n];
  for (int x=0; x<n; x++)
    for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant)
      int y = ls.val;
      m[x][y] = true;
    }
}
```

Propriété des graphes

- Chemins: suites finies de sommets immédiatement successeurs dans le graphe (orienté ou non)
- Cycles: chemins avec mêmes extrémités
- Connexité: il existe un chemin entre tous les sommets du graphe non-orienté sous-jacent. Composante connexe
- Forte connexité (dans un graphe orienté): pour toute paire de sommets x, y, il existe un chemin de x vers y et un chemin de y vers x. Composantes fortement connexes

Autre représentations: matrices d'incidence

- incidence sommet-arc: pour $G = (X, U)$, c'est la matrice $A = (a_{iu})$, $i = 1, \ldots, N$, $u = 1, \ldots, M$ avec: si u est un arc de i vers j dans G alors $a_{iu} = 1$ et $a_{ju} = -1$ et le reste de la colonne est nul
- Exercice: A est totalement unimodulaire: toutes ses sous-matrices carrées sont unimodulaires (de déterminant 1, 0 ou -1)
- incidence sommet-arête: on ne s'intéresse plus à l'orientation des arcs, donc aux signes des entrées de la matrice

Exercice

- Différentes représentations du graphe plus haut?
- Tous les graphes sont-ils représentables par matrice d'incidence sommet-arc, sommet-arête?
- Comment faire pour représenter efficacement les matrices d'incidence sommet-arc? Comment résoudre le problème plus haut? Quelle est alors la propriété des boucles?

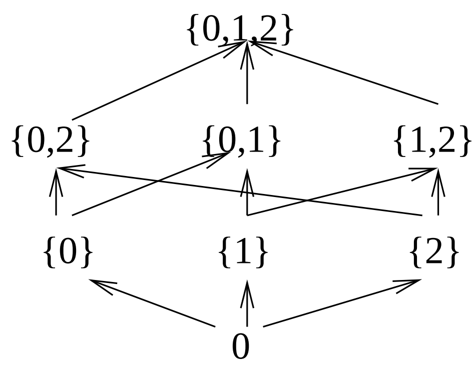
Quelques cas particuliers

- DAG (graphes sans cycles orientés)
- Arbres (non-orientés):
  - graphe connexe et sans cycle
  - graphe connexe, minimal pour cette propriété
  - graphe connexe et #arêtes=#sommets-1
  - graphe sans cycle et maximal pour cette propriété
  - graphe sans cycle et #arêtes=#sommets-1
  - toute paire de sommets est reliée par un chemin élémentaire et un seul

Exercice: prouver les équivalences plus haut.

Graphes particuliers

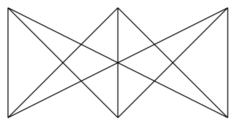
- Graphes de Hasse (représentation d'un ordre partiel)



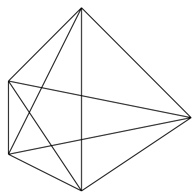
- etc.

Quelques observations géométriques et combinatoires

- ~~Planarité: un graphe est planaire si il est "représentable" dans~~ R^2 sans aucune intersection entre arêtes distinctes
- notion de subdivision, graphes de Kuratowski $K_{3,3}$ et K_5:



K3,3



K5

- notion de face d'un graphe (topologique): composante connexe de $R^2 \backslash G$
- caractéristique d'Euler-Poincaré: pour les graphes planaire, $\#sommets - \#arêtes + \#faces = 2$.

Exercice: vérifier que $K_{3,3}$ (problème des 3 villas et des 3 usines) ne peut être planaire

Quelques notions utiles

- Degré d'un sommmet dans un graphe orienté: nombre d'arcs en sortant plus nombre d'arcs y entrant
- Points d'articulation: un point tel que si on l'enlève, on crée une nouvelle composante connexe
- Isthme: une arête telle que si on l'enlève, on augmente le nombre de composantes connexes
- etc.

Exercice: Isthme et points d'articulation du graphe plus haut?

Un peu de raisonnement sur les graphes

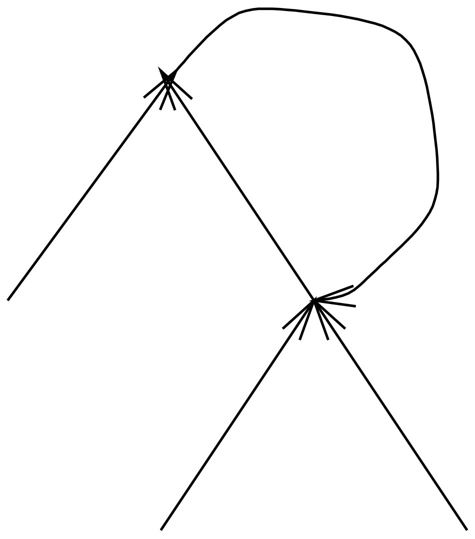
On peut raisonner en général:

- par récurrence sur le nombre de sommets et d'arcs (Application: prouver le résultat sur la caractéristique d'Euler-Poincaré)
- par induction "structurelle" dans des cas particuliers de graphes, par exemples les arbres n-aires (ou les listes)
- combinatoirement sur relations entre nombre de sommets/arcs (invariants également topologiques...)
- topologiquement: sur les composantes connexes, fortement connexes, points d'articulation etc.

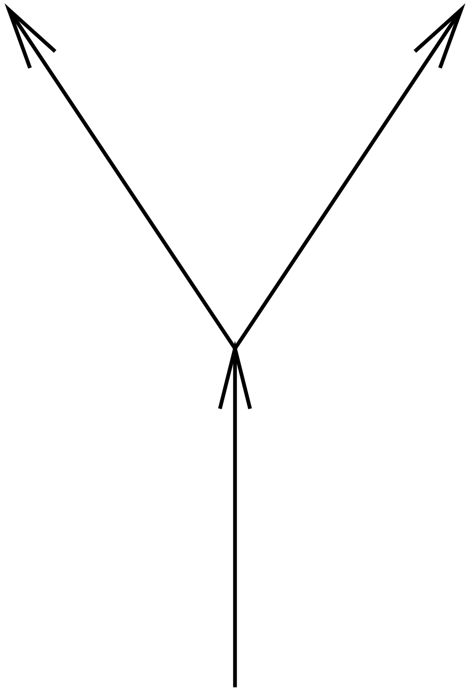
Exercice: les pseudo-forêts

- pseudo-forêt = graphe *orienté* tel que tout noeud ait au plus un voisin immédiat, cad, pour tout noeud n de la pseudo-forêt $G = (V, E)$, le cardinal de $\{n' \mid (n, n') \in E\}$ est inférieur ou égal à un: le nombre d'arcs sortant de tout noeud est au plus un.

- Pseudo-forêt:



Ceci n’est pas une pseudo-forêt:



Exercice

- arbre = graphe *orienté* avec une seule racine (c'est-à-dire un seul point qui n'a pas de voisin immédiat), et un unique chemin entre tout noeud du graphe et cette racine (à l'envers de d'habitude...)
- graphe fonctionnel G = graphe orienté fabriqué à partir d'une fonction $f : V \to V$, où V est un ensemble fini, $G = (V, E)$ (le même V) où $E = \{(v, f(v)) \mid v \in V\}$.

Questions

- Est-ce que tous les graphes sont des graphes fonctionnels?
- Montrer que les graphes fonctionnels sont des pseudo-forêts, que les pseudo-forêts sont des unions disjointes d'arbres et de pseudo-arbres et que les pseudo-forêts qui sont des graphes fonctionnels sont des unions de pseudo-arbres.

Solution...

- Le deuxième graphe n'est pas un graphe fonctionnel: le sommet intermédiaire aurait deux images par une fonction dans ce cas!
- Un graphe fonctionnel est tel que chaque sommet v a un unique arc sortant: c'est $(v, f(v))$.

Solution...

Soit G une pseudo-forêt. Considérons T une (quelconque) des composantes connexes de G.

Remarquons déjà que T ne peut avoir qu'une composante fortement connexe ayant un cardinal (nombre de sommets) strictement supérieur à un, au plus.

En effet, supposons que l'on ait par exemple 2 composantes fortement connexes C_1 et C_2 de cardinal strictement supérieur à un dans T connexe, alors il existe un arc a de $u \in C_1$ vers $v \in C_2$ (ou le contraire, peu importe).

Mais il existe forcément un arc sortant de u, restant dans C_1, donc distinct de a, donc u a au moins deux arcs sortant, impossible car T est un sous-graphe d'une pseudo-forêt.

Solution...

Si T n'a aucune composante fortement connexe de cardinal strictement supérieur à un, cela veut dire que T ne comporte aucun cycle. Dans ce cas, il ne peut y avoir deux chemins d'un sommet u vers un sommet v dans T, car sinon un sommet intermédiaire aurait au moins deux arcs sortant. Donc T est un arbre.

Supposons maintenant que T a une composante fortement connexe C non réduite à un point. Alors C est forcément un cycle simple (c'est à dire un chemin dont les deux extrémités sont identifiées) - sinon il y aurait plus d'un arc sortant dans cette composante fortement connexe.

Solution...

Soit r l'un quelconque des sommets. Enlevons l'unique arc sortant b de r. Alors $C \backslash b$ est connexe non fortement connexe, et $T \backslash b$ est une pseudo-forêt connexe, dont toutes les composantes fortement connexes sont réduites à un point.

Par ce que l'on vient de voir, c'est un arbre: résultat, T est un arbre auquel on a adjoint un arc de la racine vers un sommet de T.

Enfin, un arbre ne peut pas être un graphe fonctionnel, car il n'y aurait pas d'image de la racine.

Parcours de graphe - versions récursives

- En profondeur d'abord ("DFS"): composantes connexes etc.
- En largeur d'abord ("BFS"): sortie de labyrinthes, chemins les plus courts etc.
- Algorithmes en $O(\#\text{sommets}+\#\text{arêtes})$
- Par rapport aux arbres: il faut faire attention à la où on est passé!
  - `BLANC`: non-traité
  - `GRIS`: en cours de traitement
  - `NOIR`: traité

Parcours en profondeur d'abord

```
final static int BLANC = 0, GRIS = 1, NOIR = 2;
static int[] couleur;

static void visiter(Graphe g) {
  int n = g.succ.length; couleur = new int[n];
  for (int x = 0; x < n; ++x) couleur[x] = BLANC;
  for (int x = 0; x < n; ++x)
    if (couleur[x] == BLANC)
      dfs(g,x);
}
```

Parcours en profondeur d'abord

```
static void dfs(Graphe g, int x) {
  couleur[x] = GRIS;
  Pour tout successeur de x dans G faire
    if (couleur[y] == BLANC)
      dfs(g,y);
  couleur[x] = NOIR;
}
```

Exercice

Que donne le parcours en profondeur d'abord du graphe plus haut, en supposant une orientation trigonométrique des successeurs des noeuds?

Meilleure implémentation...

```
static int numOrdre;

static void visiter(Graphe g) {
  numOrdre = -1;
  int n = g.succ.length; int[] num = new int[n];
  for (int x = 0; x<n; ++x) num[x] = -1;
  for (int x = 0; x<n; ++x)
    if (num[x] == -1)
      dfs(g,x,num);
}
```

Meilleure implémentation...

```
static void dfs(Graphe g, int x, int[] num) {
  num[x] = ++numOrdre;
  for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
    int y = ls.val;
    if (num[y] == -1)
      dfs(g,y,num);
  }
}
```

Exercice: comment supprimer numOrdre? Comment utiliser `dfs` pour calculer l'ensemble des composantes connexes?

Exercices

- Comment calculer les points d'articulation? (élément de réponse: par un parcours en profondeur...)
- Comment calculer les composantes fortement connexes (idem)?

Parcours en largeur d'abord

```
static void visiter(Graphe g) {
  int n = g.succ.length;
  numOrdre = -1; FIFO f = new FIFO(n);
  for (int x = 0; x<n; ++x) num[x] = -1;
  for (int x = 0; x<n; ++x)
    if (num[x] == -1) {
      FIFO.ajouter(f,x); num[x] = -2;
      bfs(g,f);
    }
}
```

Que se passe t-il si on change la `FIFO` en `Pile`?

Parcours en largeur d'abord

```
static void bfs(Graphe g, FIFO f) {
  while (!f.vide()) {
    int x = FIFO.supprimer(f);
    num[x] = ++numOrdre;
    for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant)
      int y = ls.vak;
      if (num[y] == -1) {
        FIFO.ajouter(f,y); num[y] = -2;
} } } }
```

Arboresces

(X, A, r) arborescence de racine r est un graphe (X, A) avec,

- $r \in X$
- pour tout x de X il existe un unique chemin d'origine r et d'extrémité x

Une arborescence est un graphe particulier, et peut être donnée par une relation sommet, père du sommet (unique, comme pour tout arbre)

Changement de représentation d’une arborescence

```
class Arborescence {
  int[] pere;

  Arborescence(Graphe g, int r) {
    int n = g.succ.length;
    pere = new int[n];
    pere[r] = r;
    for (int x=0; x<n; x++)
      for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant
        int y = ls.val;
        pere[y] = x;
      }
  }
}
```

Arborescences de Trémaux

```
static Arborescence arboTremaux(Graphe g, int x) {
  int n = g.succ.length;
  Arborescence a = new Arborescence(n);
  for (int i=0; i<n; i++) a.pere[i]=-1;
  a.pere[x] = x;
  tremaux(g,x,a);
  return a;
}
```

Arborescences de Trémaux

```
static void tremaux(Graphe g, int x, Arborescence a) {
  for (Liste ls=g.succ[x]; ls!=null; ls=ls.suivant) {
    int y = ls.val;
    if (a.pere[y] == -1) {
      a.pere[y] = x;
      tremaux(g,y,a);
    }
  }
}
```

Version non-récursive

- principe général, simuler la pile d'appel par une vraie pile
- ici empiler et dépiler les sommets parcourus

Version non-récursive

```
static Arborescence arboTremauxPile(Graphe g, int x) {
  int n = g.succ.length;
  Arborescence a = new Arborescence(n);
  for (int i=0; i<n; i++) a.pere[i] = -1;
  Pile p = new Pile(n);
  a.pere[x] = x;
  Pile.ajouter(p,x);
  while (!Pile.estVide(p)) {
    x = Pile.supprimer(p);
    for (Liste ls=g.succ[x]; ls!=null; ls=ls.suivant) {
      int y = ls.val;
      if (a.pere[y] == -1) {
        a.pere[y] = x;
        Pile.ajouter(p,y); } } } }
```