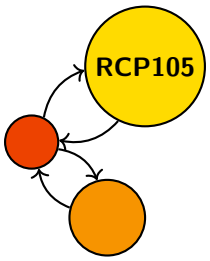
Sommaire



Arborescences

Préambule

Comment représenter les liens de dépendance au sein d'une base de données ?

Comment parcourir tous les sommets d'un graphe sans boucler ?

1. Parcours d'un graphe

1.1. Arborescences

Définition 1 (Arborescence)**.**

Une ***arborescence*** (ou ***arbre***) (X,A,r) de ***racine*** r est un graphe (X,A), où r est un élément de X tel que, pour tout sommet x, il existe un ***unique chemin*** d'origine r et d'extrémité x.

C'est-à-dire, $\forall x, \exists !\{y_0, y_1, ..., y_p\}$ tel que $y_0 = r, y_p = x, \forall i, 0 \leqslant i < p,\ (y_i, y_{i+1}) \in A$.

L'entier p est appelé la ***profondeur*** (nombre d'arcs) du sommet x dans l'arborescence.

Dans un ***arbre***, les ***fils*** d'un sommet sont ordonnés (on distingue le fils gauche du fils droit).

L'***unique prédécesseur*** d'un sommet (différent de r) est appelé son ***père***. L'ensemble des $y_0, y_1, ...y_p$, formant le chemin de $r = y_0$ à $x = y_p$, est appelé l'ensemble des ***ancêtres*** de x. Les successeurs de x sont appelés ses ***fils***. L'ensemble des sommets extrémités d'un chemin d'origine x est l'ensemble des ***descendants*** de x.

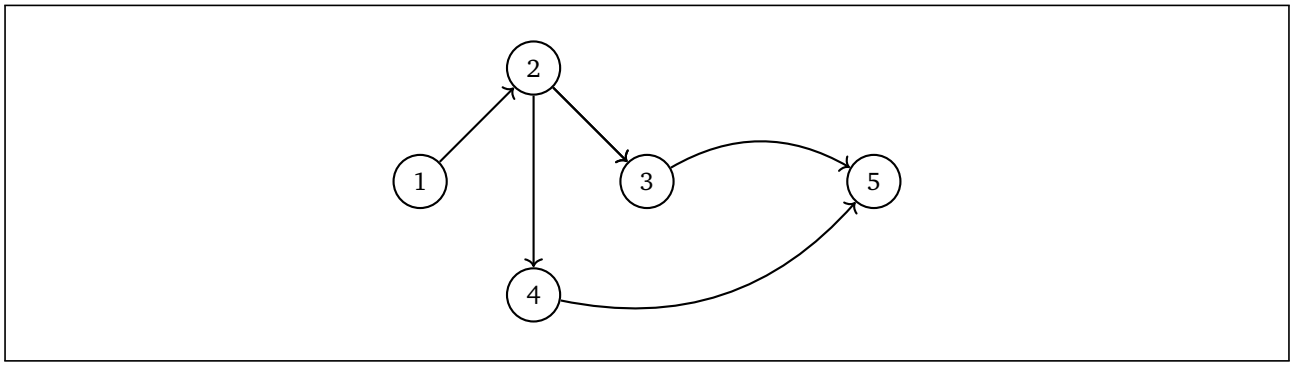


FIGURE 1.1 – Ce graphe n'est par un arbre, car 2 chemins mènent au sommet 5

Propriété 1 (Arborescence)**.**
Etant donné un **graphe symétrique** *G comportant* **n sommets**, *les propriétés suivantes sont équivalentes pour caractériser si G est un* **arbre** *:*

1) *Il existe une chaîne et une seule entre 2 sommets quelconques de G.*

2) G est sans cycle et possède **n-1 arêtes**.

3) G est sans cycle, et en ajoutant une arête, on crée un et un seul cycle élémentaire.

4) G est connexe -cf chapitre connexité- et sans cycle.

5) G est connexe -cf chapitre connexité- et admet n-1 arêtes.

Mini-exercices.

1) Montrez qu'une arborescence a nécessairement au moins 1 sommet de degré 1, sinon il y aurait 1 circuit.
2) Montrez que si G a (n-1) arêtes pour n sommets, alors au moins 1 de ses sommets est de degré 1.
3) Démontrez les propriétés précédentes.
   - 1 $\Longrightarrow$ 2 : par contraposée pour acyclique et récurrence sur n pour le nombre d'arêtes et Q1.
   - 2 $\Longrightarrow$ 1 : par récurrence sur n et Q2.

```
1  class Arborescence {
2    int[ ] pere;
3    Arborescence(int n) { pere = new int[n];}
4
5    Arborescence(Graphe g, int r) {
6     int n=g.succ.length;
7     pere=new int[n];
8     pere[r]=r;
9     for (int x=0;x<n;++x)
10              for (Liste ls=g.succ[x];ls!=null;ls=ls.suivant) {
11                  int y=ls.val;
12                  pere[y]=x;
13              }
14   }
15 }
```

FIGURE 1.2 – Algorithme - Classe arborescence

1.2. Adressage IP dans un réseau - Numérotation préfixe

Comment distribuer les adresses IP d'un réseau informatique ?

Notation 1 (Numérotation des sommets)**.**
Dans la suite, les sommets des graphes sont numérotés de 0 et n-1, avec n nombre de sommets.

Définition 2 (Arborescence préfixe)**.**
Une arborescence est dite ***préfixe*** si, pour tout sommet i, l'ensemble des descendants de i est un intervalle de l'ensemble des entiers dont le plus petit élément est i.
Dans une arborescence préfixe, les intervalles de descendants s'emboîtent les uns dans les autres comme des systèmes de parenthèses.
On parle de ***numérotation préfixe*** : la racine est initialisée à 0, incrémenter la numérotation en parcourant le ***sous-arbre droit***, puis le ***sous-arbre gauche***.

Remarque 1 (Autres numérotations)**.**
Il existe aussi des numérotations :

Infixe : parcourir le sous-arbre droit, traiter la racine, parcourir les sous-arbre gauche.

Suffixe (appelé aussi ***postfixe***) : parcourir le sous-arbre droit, le sous-arbre gauche, puis traiter la racine.

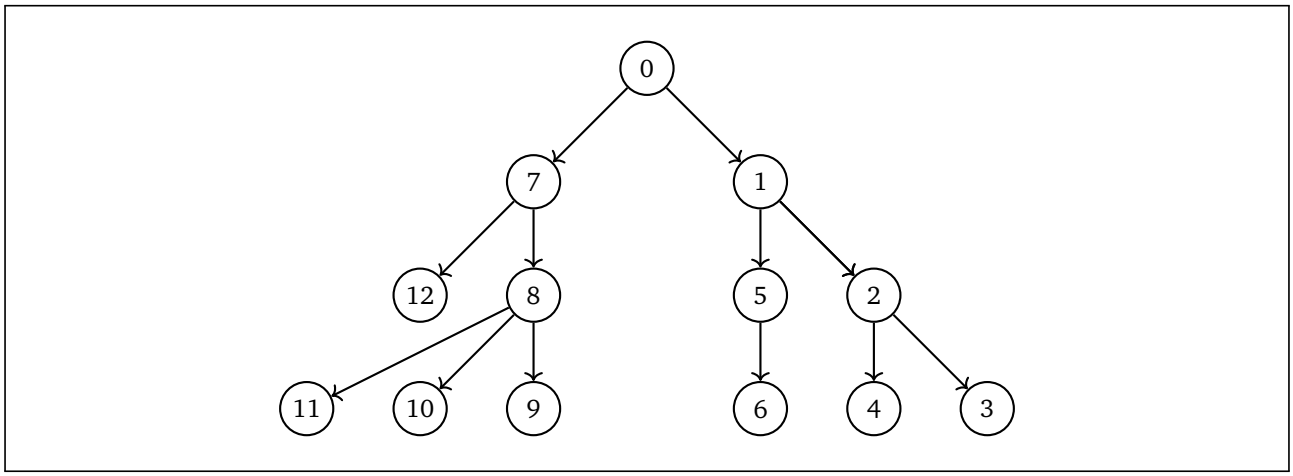


FIGURE 1.3 – Arborescence préfixe

Remarque 2 (Sous-branches)**.**

Dans la figure 1.3 :

- Descendants de $0 = [1, 11]$
- Descendants de $1 = [2, 6]$
- Descendants de $7 = [8, 12]$
- ...

Dans la numérotation préfixe, les fils ont un numéro plus grand que le père de leur branche.

Proposition 1 (Numérotation préfixe d'une arborescence - Graphe orienté)**.**
[2] *Pour toute arborescence (X,A,r), il existe une re-numérotation des éléments de X qui la rend* **préfixe**.

Démonstration. Pour trouver cette numérotation, on applique l'algorithme récursif suivant :

- La racine est numérotée 0.
- Le premier fils x_1 de la racine est numéroté 1.
- L'arborescence des descendants de x_1 est numérotée par appels récursifs de l'incrémentation, on obtient ainsi des sommets numérotés de 1 à p_1.
- Le deuxième fils de la racine est numéroté $p_1 + 1$ et les descendants de ce fils sont numérotés récursivement de $p_1 + 1$ à p_2.
- On procède de même et successivement pour tous les autres fils de la racine. La démonstration que la numérotation obtenue est bien préfixe se fait par récurrence sur le nombre de sommets de l'arborescence.

□

```
1  // Sommets numerotes de 1 a n
2  int[] numPrefixe(Graphe g, int r) {
3     int cpt=1;
4     int n=g.succ.length;
5     int[ ] num=new int[n];
6     for (int i=0;i<n;++i) {num[i]=-1;}
7     numPrefixe1(g,r,num,cpt);
8     return num;
9  }
10
11 void numPrefixe1(Graphe g,int x,int[ ] num,int k) {
12    num[x]=k;
13    cpt=cpt+1;
14    // Parcours en profondeur
```

```
15      for (Liste ls=g.succ[x];ls!=null;ls=ls.suivant) {
16              int y=ls.val;
17              numPrefixe1(g,y,num,cpt);
18      }
19  }
20
21  // Pour changer la numerotation des peres en numerotation prefixe
22  void appliquerNum(int[ ] pere,int[ ] num) {
23      int n=pere.length;
24      for (int i=0;i<n;++i)
25              pere[num[i]]=num[pere[i]];
26  }
27
28  Arborescence arboPrefixe(Graphe g,int r) {
29          Arborescence a=new Arborescence(g,r);
30          appliquerNum(a.pere,numPrefixe(g, r));
31  return a;
32  }
```

FIGURE 1.4 – Algorithme numérotation préfixe

Complexité : Les parcours en ***profondeur*** passent une fois par sommet et par arc. La complexité de tels parcours dans un graphe G=(X,A) est donc en $O(|X|+|A|)$, noté $O(X+A)$ ou $O(n+p)$.

Mini-exercices.
Si on applique la fonction `numPrefixe(g=Graphe,r=2)` ci-dessus à l'arbre G de la figure 1.5 :

1) Que vaut le vecteur num[], quand x dans « *numPrefixe1* » vaut 6, en supposant que les successeurs sont classés par ordre croissant de numéro ? Donnez l'état de la pile des appels récursifs.
2) Donnez la numérotation préfixe complète du graphe.

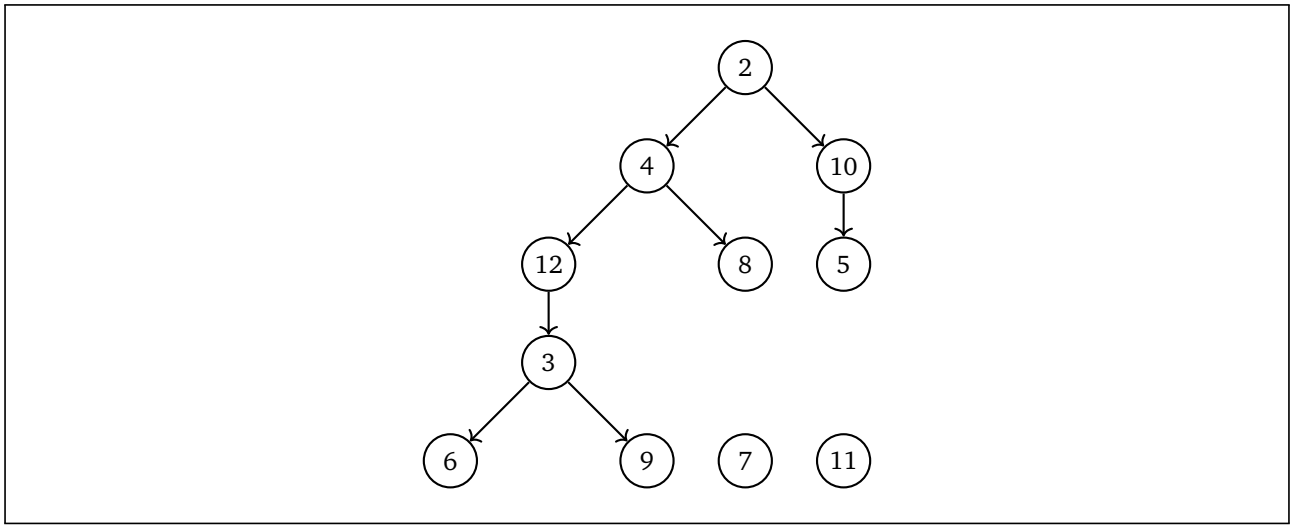


FIGURE 1.5 – Arborescence

1.3. Parcours d'un labyrinthe - Arborescence de Trémaux (F) [2]

Comment parcourir un graphe avec un minimum d'arcs et de sommets sans reboucler ?

L'***algorithme de Trémaux*** du nom de l'ingénieur ***Trémaux*** (1818-1895), dénommé aussi ***Depth First Search*** (***parcours en profondeur***) par ***Tarjan*** (1948), permet de parcourir sur le principe du fil d'Ariane, un graphe suivant une ***arborescence*** à partir d'un sommet x ***racine***.

Définition 3.
L'***arborescence de Trémaux*** (Y,T,x) d'un graphe G=(X,A) de ***racine*** x, est un ***sous-graphe*** de G sans cycle qui recouvre les sommets du graphe G à partir de x. Il est appelé ***arbre couvrant*** ou ***arbres de recouvrement*** (***spanning tree***). $Y \subset X, T \subset A$.

Pour calculer l'***arborescence de Trémaux*** (Y, T, x), une numérotation ***préfixe*** de racine x est mise en oeuvre, et en même temps le graphe G est élagué par suppression des arcs dits de ***descente***, de ***retour*** et ***transverses***.

L'ensemble des arcs du graphe G=(X,A) qui ne sont pas dans l'arborescence de Trémaux (Y,T,x) sont de 4 types cf figure 1.6 :

1) Les arcs dont l'origine n'est pas dans Y, ce sont les arcs issus d'un sommet qui n'est ***pas atteignable*** à partir de x ;
2) Les ***arcs de descente***. Il s'agit des arcs de la forme (y, z) ou z est un descendant de y dans (Y,T,x), mais n'est pas un de ses successeurs (immédiats) dans cette arborescence : arc vers un « *petit-fils* ».
3) Les ***arcs de retour***. Il s'agit des arcs de la forme (y,z) où z est un ancêtre de y dans (Y,T,x) : arc vers un « *grand-parent* ».
4) Les ***arcs transverses***. Il s'agit des arcs de la forme (y, z) où z n'est pas un ancêtre, ni un descendant de y dans (Y,T,x) : arc vers un « *cousin* ».

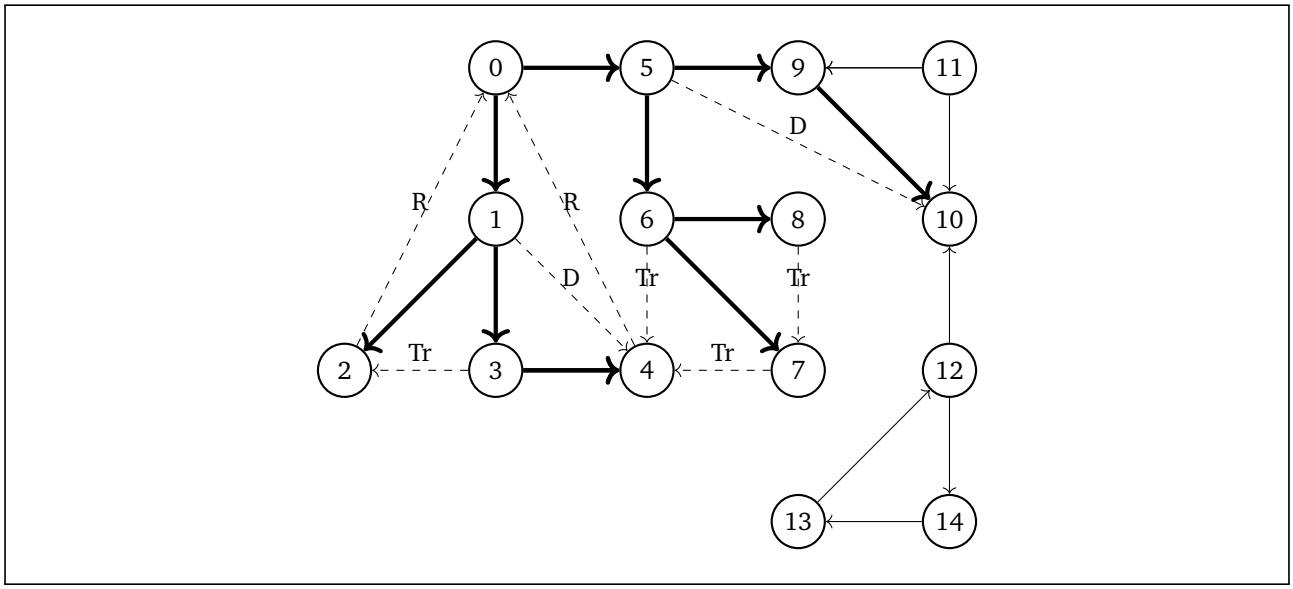


FIGURE 1.6 – Numérotation préfixe d'une arborescence de Trémaux en gras

```
1   Arborescence arboTremaux(Graphe g, int r) {
2     int n=g.succ.length;
3     Arborescence a=new Arborescence(n);
4     for (int i=0;i<n;++i) a.pere[i]=-1;
5     a.pere[r]=r;
6     tremaux(g, r, a);
7     return a;
8   }
9
10  void tremaux(Graphe g,int x,Arborescence a) {
11    for (Liste ls=g.succ[x];ls!=null;ls=ls.suivant) {
12      int y=ls.val;
13      if (a.pere[y]==-1) {
14        a.pere[y]=x;
15        tremaux(g,y,a);
16      }
17    }
```

```
18  }
```

FIGURE 1.7 – Algorithme arbre de Trémaux

Remarque 3 (Récursivité).
Parcours en ***profondeur*** $\Leftrightarrow$ récursivité.

Mini-exercices.
Si on applique la fonction « *arboTremaux(g,0)* » ci-dessus à l'arbre de la figure 1.6 :

1) Que vaut a.pere[] quand x=6, en supposant les successeurs classés par ordre croissant de numéro ?
2) Donnez l'état de la pile des appels récursifs à `Trémaux()`.

2. Plus courts chemins entre 2 sommets - Parcours en largeur

Définition 4.
Dans un graphe G=(X,A), pour chaque sommet x, le graphe ***des plus courts chemins*** de racine x est une arborescence (Y, B, x) telle que :

1) Un sommet y appartient à Y si et seulement si il existe un chemin d'origine x et d'extrémité y.
2) La longueur du plus court chemin de x à y dans G est égale à la profondeur de y dans l'arborescence (Y,B,x).

Remarque 4 (Arborescence des plus courts chemins).

Un plus court chemin est par définition unique, ce qui suffit à définir l'arborescence.

Si $a_1, a_2, \dots a_p$ est un plus court chemin entre a_1 et a_p, alors le chemin $a_1, a_2, \dots a_i$ est aussi un plus court chemin entre a_1 et a_i pour tout i vérifiant $1 \leqslant i \leqslant p$.

Théorème 1 (Plus courts chemins).
[2] Pour tout graphe G=(X,A) et tout sommet x de G, il existe une **arborescence des plus courts chemins** *de racine x.*

Démonstration. On considère la suite d'ensembles de sommets construite de la façon suivante :

- $Y_0 = \{x\}$.
- Y_1 est l'ensemble des successeurs de x, duquel il faut éliminer x si le graphe possède un arc ayant x pour origine et pour extrémité.
- Y_{i+1} est l'ensemble des successeurs d'éléments de Y_i qui n'appartiennent pas à $\bigcup_{k=1,i} Y_k$

D'autre part pour chaque $Y_i (i > 0)$, on construit l'ensemble d'arcs B_i contenant, pour chaque $y \in Y_i$, un arc ayant comme extrémité y et dont l'origine est dans Y_{i-1}. On pose ensuite : $Y = \bigcup_{k=1,i} Y_k$, $B = \bigcup_{k=1,i} B_k$. ***Le graphe (Y, B) est par construction une arborescence. C'est une arborescence des plus courts chemins.*** □

Remarque 5 (Parcours en largeur).

L'arborescence des plus courts chemins correspond à un ***parcours en largeur*** (***Breadth First Search***). Elle n'est pas ***unique***.

Le calcul des chemins les plus ***longs*** serait possible sur des graphes sans cycle.

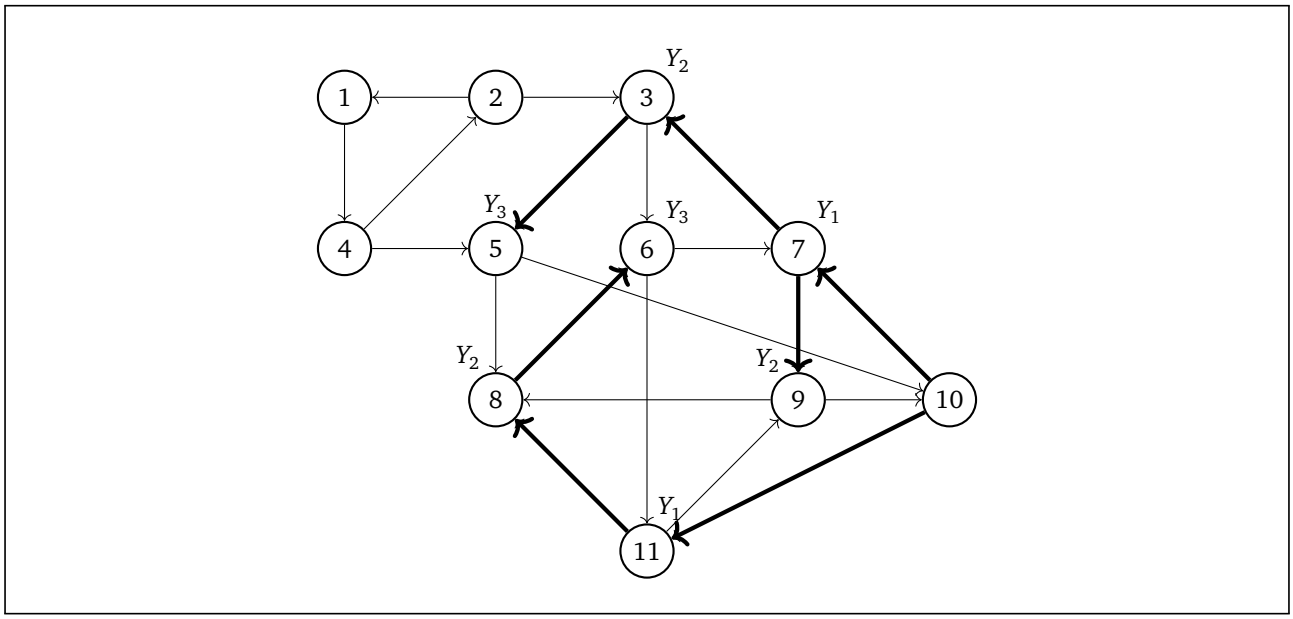


FIGURE 1.8 – Arborescence en gras des plus courts chemins de racine 10

```
1  static Arborescence arboPlusCourt(Graphe g,int x){
2    int n=g.succ.length;
3    Arborescence a=new Arborescence (n);
4    for (int i=0;i<n;++i) {a.pere[i]=-1;}
5    FIFO f=new FIFO(n);
6    a.pere[x]=x; // Racine x
7    FIFO.ajouter(f,x);//Push ajoute x
8    while (!FIFO.estVide(f)){
9        x=FIFO.supprimer(f);//Pull
10       // Parcours en largeur
11       for (Liste ls=g.succ[x];ls!=null;ls=ls.suivant){
12          int y=ls.val;
13          if (a.pere[y]==-1){
14             a.pere[y]=x;
15             FIFO.ajouter(f,y);//Push ajoute y
16          }
17       }
18   }
19   return a;
20 }
```

FIGURE 1.9 – Algorithme - Arborescence des chemins les plus courts à compléter cf exercice

Complexité : Les parcours en ***largeur*** passent une fois par sommet et par arc. La complexité de tels parcours dans un graphe G=(X,A) est donc en $O(|X|+|A|)$, noté $O(X+A)$ ou $O(n+p)$.

Mini-exercices.
On applique la fonction « *arboPlusCourt(g=G,x=10)* » à l'arbre G de la figure 1.8, en supposant les sommets traités par ordre croissant de numéro.

1) Quel est l'état de la FIFO quand x=3 dans la boucle while ?
2) Modifier le code pour calculer en plus la distance au sommet origine, en utilisant un tableau `int dist[n]` mise à jour avec `dist[y]=dist[x]+1`.

Remarque 6 (Graphes symétriques)**.**
Le parcours en ***largeur*** fonctionne aussi pour calculer les chemins les plus courts des ***graphes symétriques***.

3. Détection de cycles - Trémaux

Comment router les messages dans un réseau informatique pour adresser une seule fois les machines ?

Définition 5 (Cycle)**.**
Les graphes ***sans circuit*** sont dits ***acycliques***, ***Directed Acyclic Graphs*** en anglais.

Le parcours de graphes en ***profondeur*** permet de traverser un graphe sans boucler, ni passer plus d'une fois par chaque sommet.

Le graphe est parcouru suivant l'algorithme de ***Trémaux*** et les sommets rencontrés sont coloriés suivant un ***code de 3 couleurs*** :

- ***Blanc*** pour les sommets non encore explorés.
- ***Gris*** pour un sommet partiellement traité (racine d'une branche pas encore explorée).
- ***Noir*** pour un sommet complètement traité (racine d'une branche déjà explorée).

- Si le successeur est ***Blanc***, il s'agit d'un arc de l'***arborescence***.
- Si le successeur est ***Gris***, il s'agit d'un arc de ***retour*** d'un cycle.
- Si le successeur est ***Noir***, il s'agit d'un arc ***transverse*** ou de descente.

```
1  int BLANC=0,GRIS=1,NOIR=2;
2  int numOrdre=0;
3  int[] tremauxPrefixe(Graphe g, r){
4    int n=g.succ.length;
5    int[]couleur=new int[n],num=new int[n];
6    for(int x=0;x<n;++x) {couleur[x]=BLANC;num[x]=-1;}
7    tremauxPrefixe(g,r,couleur,num);
8  return num;
9  }
10
11 void tremauxPrefixe(Graphe g,int x,int[] couleur,int[] num){
12   couleur[x]=GRIS;
13   num[x]=++numOrdre;
14   // Parcours en profondeur
15   for(Liste ls=g.succ[x];ls!=null;ls=ls.suivant){
16      int y=ls.val;
17      if(couleur[y]==BLANC) tremauxPrefixe(g,y,couleur,num);
18   }
19   couleur[x]=NOIR;
20 }
```

FIGURE 1.10 – Algorithme - Trémaux - Détection des arcs de descente, retour et transverses

Remarque 7 (Récursivité et complexité)**.**

- Parcours en ***profondeur*** $\Leftrightarrow$ récursivité.
- A la question, existe-t-il 1 chemin entre 2 sommets, l'algorithme de ***Trémaux*** y répond avec une complexité en O((n+p)*n), à comparer à la complexité en $O(n^3)$ de la ***fermeture transitive*** et en $O(n^4)$ de l'algorithme de dénombrement des ***chemins de longueur k***.

Mini-exercices.
Si on applique la fonction `tremauxPrefixe(g,r)` à l'arbre de la figure 1.11 :

1) Que vaut `couleur[]`, quand x=2 et y=6 ?
2) Donnez l'état de la pile des appels récursifs à `TrémauxPrefixe()`.

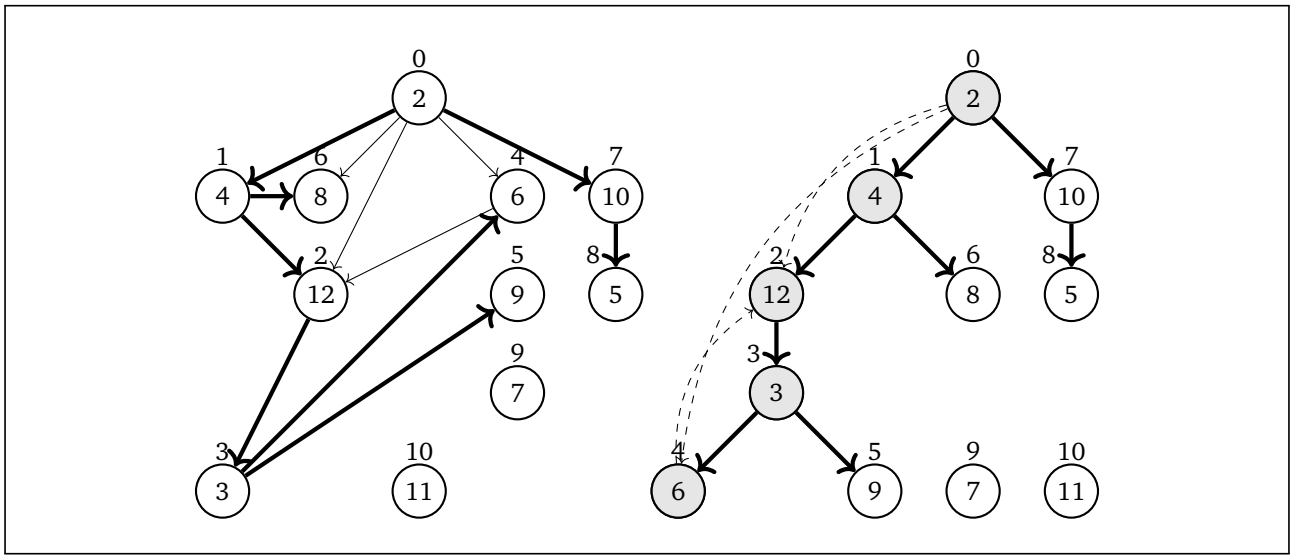


FIGURE 1.11 – Graphe avec circuits et son arbre de recouvrement en gras de racine x=2 - Numérotation préfixe au dessus du cercle

Remarque 8 (Détection possible des cycles avec Trémaux)**.**

L'algorithme de ***Trémaux*** de calcul d'un arbre de recouvrement à partir d'un sommet x permet aussi de ***détecter les circuits*** à partir des ***arcs*** de ***retour***. Si les successeurs sont toujours ***blancs*** ou ***noirs***, il n'y a pas de cycles.

Sur la figure 1.11, un arc part du sommet 6 vers le sommet gris 12. C'est la preuve de l'existence d'un cycle (6,12,6).

```
1  boolean acyclique(Graphe g,r){
2    int n=g.succ.length;
3    int[] couleur=new int[n];
4    for(int x=0;x<n;++x)couleur[x]=BLANC;
5    return cycleEn(g,r,couleur));
6  }
7
8  //Extremite arc retour grise =>circuit
9  //Extremite arc descente noire =>ras
10 //Extremite arc transverse noire =>ras
11 //Extremite arc arborescence blanche =>ras
12 //Les branches sans circuit sont noires.
13 boolean cycleEn(Graphe g,int x,int[] couleur){
14   couleur[x]=GRIS;
15   // Parcours en profondeur
16   for(Liste ls=g.succ[x];ls !=null;ls=ls.suivant){
17     int y=ls.val;
18     if((couleur[y]==GRIS)||((couleur[y]==BLANC)&&cycleEn(g,y,couleur)) return true;
19   }
20   //Plus de successeur ou arc descente ou transverse.
21   couleur[x]=NOIR;
22   return false;
23 }
```

FIGURE 1.12 – Algorithme - Détection des cycles d'un graphe

Mini-exercices.

Si on applique la fonction `acyclique(g)` à l'arbre de la figure 1.11 :

1) Que vaut x en sortie de boucle et que vaut couleur[] ?

4. Tri topologique - Trémaux

Pour lire le chapitre 6 d'un livre, il faut avoir lu les chapitres 3, 12, 4 et 2 (comme modélisable dans l'arbre de recouvrement de la fig 1.11). En revanche les chapitres 8 et 9 ne sont pas utiles. Quelle est la liste des chapitres nécessaires à la lecture du chapitre 6 ?

Remarque 9 (Circuits)**.**
Ce problème n'a pas de solution si le graphe de dépendance des chapitres contient un ***circuit***.

Définition 6.
Le ***tri topologique*** est l'opération qui consiste à mettre en ordre les sommets d'un graphe orienté ***sans circuit***, telle que si il existe un chemin de u à v dans le graphe, alors le numéro de u est inférieur à celui de v.

Algorithme 1 (tri topologique)**.**
[2] Pour un sommet terminal s donné extrémité d'une ***arborescence***, il faut construire une liste formée de tous les sommets origines d'un chemin d'extrémité s. Pour cela, on applique l'algorithme de descente en ***profondeur*** (Trémaux) sur le ***graphe inverse*** vers la ***racine*** de l'arborescence. (Au lieu de considérer les successeurs succ[x] du sommet x, on considère ses prédécesseurs pred[x].) Au cours de cette recherche, quand on a fini de visiter un sommet, on le met en tête de liste. En fin d'algorithme, on calcule l'image mirroir de la liste en inversant la liste.

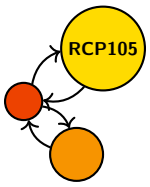
Exemple 1.
Par exemple, le tri topologique pour le sommet 9 de la figure 1.11 donne la suite de sommets 2-4-12-3-9.

Complexité : L'algorithme est du type recherche en ***profondeur*** en $O(X+A)$, suivi de l'image miroir qui ne dépasse pas $O(A)$. Au total la ***complexité*** est en $O(X+A)$.

```
1  // Si le sommet terminal est connu, un parcours en profondeur suffit.
2  // Sinon il faut executer calculerTerminaux(g).
3  int BLANC=0,GRIS=1,NOIR=2;
4  Liste triTopologique(Graphe g,int s){
5   int n=g.preced.length;
6   int[] couleur=newint[n];
7   for(int x=0;x<n;++x) etat[x]=BLANC;
8  //Inverse ordre de la liste
9   return Liste.mirroir(DFS(g,s,couleur,null));
10 }
11
12 //Depth First Search
13 //-g:grapheinverse
14 //-r:tritopologique
15 //-couleur: detectioncycles
16 Liste DFS(Graphe g,int x,int[] couleur,Liste r){
17  etat[x]=GRIS;
18  for(Liste ls=g.preced[x];ls !=null;ls=ls.suivant){
19   int y=ls.val;
20   if(couleur[y]==GRIS) throw new Error("Le graphe a un cycle");
21   if(couleur[y]==BLANC){
22      r=DFS(g,y,couleur,r);
23   }
24 }
25 return return new Liste(x,r);
26 }
27
28 //Pour trouver les sommets terminaux
29 // un parcours lineaire des arcs suffit.
30 boolean[] calculerTerminaux(Graphe g){
31   int n=g.succ.length;
32   int[] terminal=new boolean[n];
```

```
33      for(int x=0;x<n;++x) terminal[x]=true;
34      for(int x=0;x<n;++x){
35          for(Liste ls=g.preced[x];ls!=null;ls=ls.suivant){
36              int y=ls.val;
37              terminal[y]=false;
38      }
39  }
40  return terminal;
41  }
```

FIGURE 1.13 – Algorithme - Tri topologique



Index

Auteurs du chapitre
Rédaction : Hervé Bailly

C'est une synthèse qui emprunte largement aux cours suivants :

- « *Initiation à la théorie des graphes* » de Thierry Brugère.[1]
- « *Informatique Fondamentale* » de Jean-Jacques Lévy. [2]

Design : « *Exo7* »

Version 3.5 – Avril 2025