

Cours 7

Files d'attente – Piles Graphes non orientés

Jean-Jacques.Levy@inria.fr
<http://jeanjacqueslevy.net>
tel: 01 39 63 56 89

secrétariat de l'enseignement:
Catherine Bensoussan
cb@lix.polytechnique.fr
Aile 00, LIX
tel: 01 69 33 34 67

<http://www.enseignement.polytechnique.fr/informatique/>

Plan

1. Files d'attente
2. Piles
3. Graphes
4. Représentation des graphes
5. Parcours en profondeur d'abord
6. Parcours en largeur d'abord
7. Arbres de recouvrement
8. Connexité et sortie de labyrinthe
9. Biconnexité.

Files d'attente représentée par un tampon circulaire

Premier arrivé, premier servi (**First In, First Out**). Structure de données très fréquente dans les programmes: par exemple dans les OS, le hardware, le temps-réel, etc.

Une première méthode compacte consiste à gérer un **tampon circulaire**.

```
class FIFO {
    int          debut, fin;
    boolean      pleine, vide;
    int[]        contenu;

    FIFO (int n) {
        debut = 0; fin = 0; pleine = false; vide = true;
        contenu = new int[n];
    }
}
```

File d'attente représentée par un tampon circulaire

```
static void ajouter (int x, FIFO f) {
    if (f.pleine)
        throw new Error ("File Pleine.");
    f.contenu[f.fin] = x;
    f.fin = (f.fin + 1) % f.contenu.length;
    f.vide = false; f.pleine = f.fin == f.debut;
}

static int supprimer (FIFO f) {
    if (f.vide)
        throw new Error ("File Vide.");
    int res = f.contenu[f.debut];
    f.debut = (f.debut + 1) % f.contenu.length;
    f.vide = f.fin == f.debut; f.pleine = false;
    return res;
}
```

Belle symétrie de ces procédures. Complexité en $O(1)$.

File d'attente représentée par une liste

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

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

    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;
        }
    }
}
```

Pile représentée par un tableau

Une pile correspond à la stratégie LIFO (**Last In, First Out**). On peut la représenter par un tableau ou une liste.

```
class Pile {
    int hauteur ;
    int contenu[];
    Pile (int n) {hauteur = 0; contenu = new int[n]; }

    static void empiler (int x, Pile p) {
        ++p.hauteur;
        if (p.hauteur >= p.contenu.length)
            throw new Error ("Pile pleine.");
        p.contenu[p.hauteur - 1] = x;
    }

    static int depiler (Pile p) {
        if (p.hauteur <= 0)
            throw new Error ("Pile vide.");
        return p.contenu [--p.hauteur];
    }
}
```

Pile représentée par une liste

```
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;
  }
}
```

Récurusif = Itératif + Pile

(Randell et Russel, 1960 ⇒ compilateur).

Idem pour l'évaluation des expressions arithmétiques avec une pile, par un parcours suffixe de l'ASA, et en empilant les valeurs et opérateurs rencontrés et en faisant des opérations entre le sommet et le sous-sommet de la pile.

Graphe

Un graphe $G = (V, E)$ a un ensemble de **sommets** V et d'**arcs** $E \subset V \times V$. Un arc $v = (n_1, n_2)$ a pour origine $org(v) = n_1$ et pour extrémité $ext(v) = n_2$.

Exemples: les rues de Paris, le plan du métro.

$G = (N, V)$ est un graphe non **orienté** ssi $(n_1, n_2) \in V$ implique $(n_2, n_1) \in V$. Par exemple, les couloirs de l'X.

Un **chemin** est une suite d'arcs v_1, v_2, \dots, v_n , telle que $org(v_{i+1}) = ext(v_i)$ pour $1 \leq i < n$, où $n \geq 0$. Un **circuit** (ou cycle) est un chemin où $ext(v_n) = org(v_1)$.

Les **dag** (*directed acyclic graphs*) sont des graphes orientés sans cycles. Exemple: le graphe des dépendances inter modules pour la création d'un projet informatique. (*Makefile*)

Un arbre est un dag. Une forêt (ensemble d'arbres disjoints) est un dag.

Exemple de graphe

	000	
100		001
	010	
	101	
110		011
	111	

Graphe de de Bruijn

Exemple de graphe



Graphe des diviseurs

Représentations d'un graphe

En gros, deux méthodes:

- **Matrice de connexion** M , où $m_{i,j} = 1$ ssi (n_i, n_j) est un arc du graphe. Matrice symétrique pour un graphe non orienté.

```
class Graphe {
    boolean[][] m;
    Graphe (int n) {
        m = new boolean[n][n];
        for (int i = 0; i <= n; ++i)
            for (int j = 0; j <= n; ++j)
                m[i][j] = false;
    }

    static void ajouterArc (Graphe g, int x, int y) { g.m[x][y] = true; }
```

- **Tableau de listes de successeurs:**

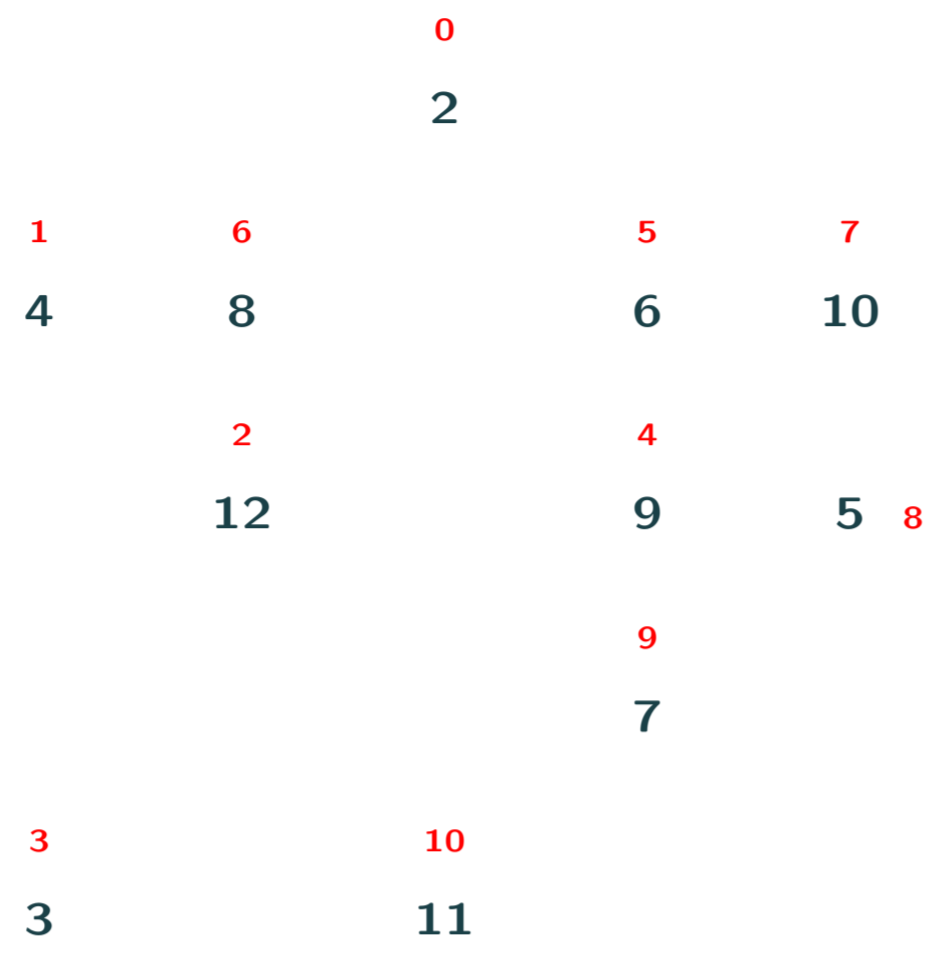
```
class Graphe {
    Liste[] succ;
    Graphe (int n) { succ = new Liste[n]; }

    static void ajouterArc (Graphe g, int x, int y) {
        g.succ[x] = new Liste (y, g.succ[x]);
    }
}
```

Entrée du graphe

```
public static void main (String[] args) {
    BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
    try {
        String s = in.readLine(); int n = Integer.parseInt(s);
        Graphe g = new Graphe (n);
        while ((s = in.readLine()) != null) {
            StringTokenizer st = new StringTokenizer(s);
            int x = Integer.parseInt(st.nextToken());
            int y = Integer.parseInt(st.nextToken());
            if (0 <= x && x < n && 0 <= y && y < n) {
                ajouterArc(g, x, y);
                ajouterArc(g, y, x);
            }
        }
        trouverArticulations (g);
        imprimerArticulations (g);
    } catch (IOException e) {
        System.err.println(e);
    }
}
```

Parcours en profondeur d'abord, Depth-First Search



Parcours en profondeur d'abord, Depth-First Search

Tarjan (1972) a démontré l'utilité de cette méthode.

```
static int id; static int[] num;

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

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

Son temps est en $O(V + E)$.

Aussi appelé arborescences de Trémaux dans le polycopié.

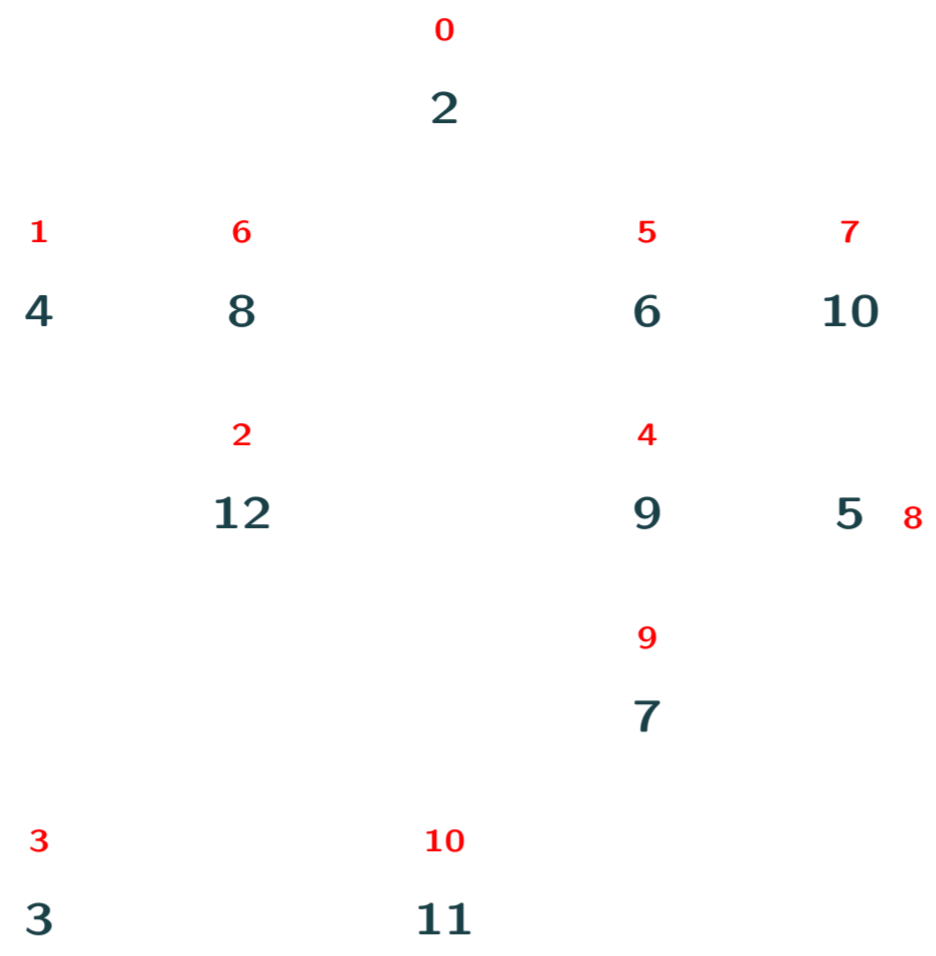
Arbres de recouvrement

L'arbre des appels récursifs à `dfs` est un arbre de recouvrement (spanning tree). Les arcs d'un arc sont de 4 sortes:

- d'un noeud de l'arbre à un de ses fils dans l'arbre
- d'un noeud de l'arbre à un de ses ancêtres dans l'arbre
- d'un noeud de l'arbre à un de ses descendants non direct dans l'arbre
- d'un noeud à un de ses cousins dans l'arbre

[Dans `dfs` d'un graphe non orienté, il n'y a que les arcs du type 1-2.]

Arbre de recouvrement – *spanning tree*



Parcours en largeur d'abord

```
      0
      2
    1  2
    4  8
      5
      12
    3  4
    6 10
      8
      9
      5 6
      9
      7
    7 10
    3 11
```

Parcours en largeur d'abord, Breadth-First Search

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

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

Itératif et plus compliqué que le parcours en profondeur d'abord.

Quelques problèmes classiques

- Sortie de labyrinthe (dans un graphe non-orienté)
- Test de non-cyclicité dans un graphe orienté
- Tri topologique dans un graphe orienté (Makefiles)
- Planarité (Tarjan)
- Connexité dans un graphe non orienté
- Plus court chemin
- Coloriage de graphe
- Optimisation de flot

Sortie de labyrinthe

On cherche un chemin de d à s .

```
static Liste chemin (Graphe g, int d, int s) {
    num[d] = ++id;
    if (d == s) return new Liste (d, null);
    for (Liste ls = g.succ[d]; ls != null; ls = ls.suivant) {
        int x = ls.val;
        if (num[x] == -1) {
            Liste r = chemin (g, x, s);
            if (r != null) return new Liste (d, r);
        }
    }
    return null;
}
```

Exercice 1 Calculer le chemin le plus court vers la sortie.

Exercice 2 Imprimer tous les chemins vers la sortie.

Bi-connectivité

- Dans un graphe non orienté, existe-t-il 2 chemins différents entre toute paire de noeuds?
- Un point d'articulation est un point qui sépare le graphe en parties non connexes si on l'enlève.
- Un graphe sans point d'articulation est un graphe **bi-connexe**.
- Exemples: les rues de Paris, la distribution d'électricité, la topologie d'un réseau informatique, etc

Bi-connectivité

Un **point d'attache** de x est le plus petit y tel qu'un chemin relie x à y contenant au plus un seul arc de retour $z \rightarrow z'$ (tel que $z > z'$)

La fonction suivante (en dfs) calcule les points d'articulation.

```
static int attache (Graphe g, int k) {
    num[k] = ++id; int min = id;
    for (Liste ls = g.succ[k]; ls != null; ls = ls.suivant) {
        int x = ls.val;
        if (num[x] == -1) {
            int m = attache (g, x);
            if (m < min) min = m;
            if (m >= num[k]) articulation[k] = true;
        } else
            if (num[x] < min) min = num[x];
    }
    return min;
}
```

Bi-connectivité – bis

Mais correction pour un noeud racine d'un arbre de recouvrement n'est articulation que s'il a au moins deux successeurs.

```
static int id; static int[] num;
static boolean[] articulation;

static void trouverArticulations (Graphe g) {
    int n = g.succ.length;
    id = -1; num = new int[n];
    articulation = new boolean[n]; int r;
    for (int x = 0; x < n; ++x) { num[x] = -1; articulation[x] = false; }
    for (int x = 0; x < n; ++x)
        if (num[x] == -1) {
            r = attache(g, x);
            if (articulation[x] && g.succ[x].suivant == null)
                articulation[x] = false;
        }
}
```

En TD – problèmes

- Recherche de la plus courte sortie dans un labyrinthe
- Recherche de la plus longue sortie dans un labyrinthe
- Recherche de toutes les sorties dans un labyrinthe
- Enumérer tous les arbres de recouvrement
- Enumérer tous les chemins (élémentaires) issus du noeud.
(Chemin élémentaire ne passe pas deux fois par un même noeud).