

**INF431**

**Algorithmes et Programmation:  
du séquentiel au distribué**

**École polytechnique**

**Partie I**

François Morain & Jean-Marc Steyaert







# Table des matières

<b>I</b>	<b>Programmation</b>	<b>13</b>
<b>1</b>	<b>Programmer avec Java</b>	<b>15</b>
1.1	Modularité . . . . .	15
1.1.1	Motivation . . . . .	15
1.1.2	Modules . . . . .	16
1.2	Programmation par objets . . . . .	18
1.2.1	Motivations . . . . .	18
1.2.2	L'héritage par l'exemple . . . . .	18
1.2.3	Propriétés de l'héritage . . . . .	21
1.2.4	Contrôle d'accès et sous-classes . . . . .	22
1.2.5	Un peu de typage . . . . .	22
1.3	Les interfaces de Java . . . . .	26
1.3.1	Spécialisations multiples . . . . .	29
1.4	Les génériques de Java . . . . .	30
1.4.1	Restreindre à certains types . . . . .	33
<b>2</b>	<b>Introduction au génie logiciel</b>	<b>35</b>
2.1	Généralités . . . . .	35
2.1.1	Quelques chiffres . . . . .	35
2.1.2	La chaîne de production logicielle . . . . .	35
2.1.3	Déboguer . . . . .	36
2.1.4	Tester . . . . .	37
2.1.5	Analyser les performances (benchmarks) . . . . .	37
2.1.6	Architecture détaillée . . . . .	37
2.1.7	Aspects organisationnels . . . . .	38
2.1.8	En guise de conclusion provisoire. . . . .	38
2.2	Génie logiciel en Java . . . . .	39
2.2.1	Paquetages et espace des noms . . . . .	39
2.2.2	Classe abstraite . . . . .	41
<b>3</b>	<b>Complexité et structures de données</b>	<b>45</b>
3.1	Complexité élémentaire . . . . .	45
3.1.1	Introduction . . . . .	45

3.1.2	Calculs élémentaires de complexité . . . . .	46
3.2	Structures de données . . . . .	47
3.2.1	Représentation des ensembles . . . . .	47
3.2.2	Piles, files, tas . . . . .	48
<b>II</b>	<b>Graphes</b>	<b>49</b>
<b>4</b>	<b>Propriétés élémentaires des graphes</b>	<b>51</b>
4.1	Introduction . . . . .	51
4.2	Définitions . . . . .	51
4.2.1	Graphes non orientés . . . . .	53
4.2.2	Sous-graphes, etc. . . . .	53
4.2.3	Degré . . . . .	54
4.2.4	Quelques graphes non orientés classiques . . . . .	56
4.3	Représentations en machine . . . . .	56
4.3.1	Représentation par une matrice . . . . .	57
4.3.2	Représentation par un tableau de listes . . . . .	57
<b>5</b>	<b>Accessibilité</b>	<b>59</b>
5.1	Problématique . . . . .	59
5.2	Définitions ; propriétés fondamentales . . . . .	59
5.3	Fermeture transitive . . . . .	60
5.3.1	Première solution par produit de matrice . . . . .	61
5.3.2	L'algorithme de Roy et Warshall . . . . .	62
5.4	Plus court chemin . . . . .	64
5.5	Arbres . . . . .	65
<b>6</b>	<b>Parcours</b>	<b>69</b>
6.1	Le cas non orienté . . . . .	69
6.1.1	Définitions et propriétés générales . . . . .	69
6.1.2	Arbre couvrant . . . . .	71
6.1.3	Les deux parcours les plus fréquents . . . . .	71
6.1.4	Parcours en largeur d'abord . . . . .	72
6.1.5	Parcours en profondeur d'abord ( <i>depth-first search</i> ) . . . . .	78
6.1.6	Construction de l'arbre couvrant . . . . .	80
6.2	Parcours dans le cas orienté . . . . .	82
6.2.1	Propriétés générales . . . . .	82
6.2.2	Propriétés du parcours en profondeur d'abord . . . . .	83
6.2.3	Graphes sans cycles . . . . .	88
6.2.4	Tri topologique . . . . .	88

<b>7</b>	<b>Topologie</b>	<b>91</b>
7.1	Forte connexité . . . . .	91
7.1.1	Définition, exemples . . . . .	91
7.1.2	Le retour de Trémaux . . . . .	92
7.1.3	Points d'attache . . . . .	94
7.1.4	Application à la forte connexité . . . . .	95
7.1.5	L'algorithme . . . . .	96
7.1.6	L'exemple détaillé . . . . .	98
7.1.7	Esquisse de la preuve . . . . .	101
7.2	Connexité dans les graphes non orientés . . . . .	102
7.2.1	Énumération des composantes connexes . . . . .	102
7.2.2	Points d'articulation . . . . .	103
7.3	Planarité . . . . .	107
7.3.1	Faces . . . . .	107
7.3.2	Graphe dual . . . . .	108
7.3.3	Relation d'Euler . . . . .	109
7.3.4	Un critère théorique de planarité . . . . .	112
7.3.5	Vers un algorithme de test de planarité . . . . .	112
7.3.6	Une idée d'algorithme . . . . .	113
7.4	Les programmes en Java . . . . .	116
7.4.1	Composantes fortement connexes . . . . .	116
7.4.2	Composantes connexes . . . . .	117
7.4.3	Points d'articulation . . . . .	118
<b>8</b>	<b>Euler et Hamilton</b>	<b>121</b>
8.1	Euler : fondateur de la théorie des graphes . . . . .	121
8.2	Hamilton (1805–1865) . . . . .	122
<b>9</b>	<b>Introduction à l'optimisation combinatoire</b>	<b>125</b>
9.1	L'algorithme de Dijkstra . . . . .	125
9.1.1	Théorie . . . . .	125
9.1.2	Implantations . . . . .	128
9.2	Arbre couvrant de poids minimal . . . . .	133
9.2.1	L'algorithme de Prim . . . . .	134
9.2.2	L'algorithme de Kruskal . . . . .	135
9.2.3	Ébauche de preuve . . . . .	137
9.2.4	Prim ou Kruskal? . . . . .	138
9.3	Les $n$ reines . . . . .	139
9.3.1	Prélude : les $n$ tours . . . . .	139
9.3.2	Des reines sur un échiquier . . . . .	139
9.4	Les ordinateurs jouent aux échecs . . . . .	142
9.4.1	Principes des programmes de jeu . . . . .	142
9.4.2	Algorithme minimax . . . . .	142
9.4.3	Principe de l'élagage $\alpha\beta$ . . . . .	142

9.4.4	Retour aux échecs . . . . .	143
<b>III</b>	<b>Langages</b>	<b>147</b>
<b>10</b>	<b>Langages rationnels</b>	<b>149</b>
10.1	Recherche de motifs dans un texte . . . . .	149
10.1.1	Un algorithme naïf . . . . .	150
10.1.2	L'algorithme KMP . . . . .	151
10.1.3	Multimotifs . . . . .	154
10.2	Automates finis . . . . .	156
10.2.1	Définitions . . . . .	156
10.2.2	Langage reconnu par un automate fini . . . . .	158
10.2.3	Combinaisons d'automates . . . . .	160
10.3	Expressions régulières et langages rationnels . . . . .	163
10.3.1	Langages rationnels : définitions . . . . .	163
10.3.2	La réciproque : le théorème de Kleene . . . . .	165
10.4	Déterminisme contre Non-Déterminisme . . . . .	166
10.4.1	Simulation du non-déterminisme . . . . .	167
10.4.2	Suppression des $\varepsilon$ -transitions . . . . .	167
10.4.3	Suppression du non-déterminisme . . . . .	168
10.5	Complexité et limites . . . . .	169
10.5.1	Déterminisation . . . . .	169
10.5.2	Automate minimal . . . . .	170
10.6	Langages non rationnels : le lemme d'itération . . . . .	172
10.7	Appendice . . . . .	173
<b>11</b>	<b>Parenthésages, Langages algébriques</b>	<b>177</b>
11.1	Exemples de structures parenthésées . . . . .	177
11.1.1	Notations . . . . .	177
11.2	Grammaires hors-contexte . . . . .	178
11.2.1	Automates, langages et grammaires . . . . .	179
11.2.2	Terminologie . . . . .	180
11.2.3	Ambiguïté . . . . .	182
11.3	Propriétés des langages algébriques . . . . .	182
11.4	Grammaires de Greibach et automates à pile . . . . .	185
11.4.1	Forme normale de Greibach . . . . .	185
11.4.2	Automates à pile . . . . .	188
11.5	Analyse par programmation dynamique . . . . .	191
11.5.1	Forme normale de Chomsky . . . . .	191
11.5.2	Reconnaissance déterministe des langages algébriques . . . . .	192
11.6	Algébricité . . . . .	196
11.6.1	Le lemme d'itération des langages algébriques . . . . .	196
11.6.2	Langage générateur . . . . .	198

<b>12 Analyse syntaxique et autres langages</b>	<b>203</b>
12.1 Éléments d'analyse syntaxique . . . . .	203
12.1.1 Analyse descendante . . . . .	204
12.1.2 Analyse ascendante . . . . .	205
12.2 Au-delà des algébriques . . . . .	210
12.2.1 L-systèmes . . . . .	210
12.2.2 Autres classes . . . . .	213
<b>13 Analyses lexicale et syntaxique par l'exemple</b>	<b>215</b>
13.1 Motivations . . . . .	215
13.2 Construction de l'analyseur lexical . . . . .	217
13.2.1 Utilisation des diagrammes de transition . . . . .	218
13.2.2 Le programme en Java . . . . .	218
13.2.3 Pour aller plus loin . . . . .	220
13.3 Analyse syntaxique . . . . .	221
13.3.1 Arbre de syntaxe abstraite . . . . .	221
13.3.2 Construction des ASA . . . . .	222
<b>IV Annexes</b>	<b>227</b>
<b>14 Quelques classes prédéfinies de Java</b>	<b>229</b>
14.1 La classe Collection . . . . .	229
14.1.1 La classe Vector . . . . .	230
14.1.2 La classe LinkedList . . . . .	230
14.1.3 La classe Hashtable . . . . .	230
14.1.4 La classe TreeSet . . . . .	231
<b>15 La classe Graphe</b>	<b>233</b>
15.1 Les choix . . . . .	233
15.2 grapheX comme exemple de package . . . . .	234
15.2.1 Comment ça marche ? . . . . .	234
15.2.2 Faire un .jar . . . . .	235
15.3 Sommets . . . . .	235
15.4 Sommets valués . . . . .	239
15.5 La classe Arc . . . . .	240
15.6 La classe abstraite Graphe et deux implantations . . . . .	242
15.6.1 Définitions génériques . . . . .	242
15.6.2 La classe Graphe . . . . .	244
15.6.3 Numérotation . . . . .	244
15.6.4 Implantation par matrice . . . . .	246
15.6.5 Implantation avec des listes . . . . .	249



# Présentation

## Les buts du cours

Dans une première partie, ce cours se veut la continuation de l'apprentissage de base de la programmation, auquel s'ajoute la description d'algorithmes sur les graphes. Dans un deuxième temps, on s'intéresse à la présentation de la théorie des réseaux, ainsi que des protocoles.

De nombreux ouvrages existent déjà sur la théorie des graphes, comme par exemple le polycopié de R. Cori et J.-J. Lévy [5]. Il est difficile d'être novateur sur un tel sujet. Nous avons toutefois privilégié une approche de la programmation utilisant les bibliothèques déjà programmées en Java, bénéficiant des types génériques. Par exemple, nous utiliserons les possibilités de simplification d'exposition des algorithmes en utilisant des itérateurs sur les sommets d'un graphe, plutôt que de rester à un codage classique des sommets par des entiers. De la même façon, une classe abstraite de graphes nous permettra de nous concentrer sur les algorithmes plutôt que sur l'implantation réelle, que nous reléguerons dans les implantations par matrice ou tableau de liste. Sans vouloir à tout crin programmer de façon outrancière, le code résultat nous semble à la fois plus lisible et plus proche du pseudocode dans lequel les algorithmes seront décrits.

## Remerciements

J.-J. Lévy nous a permis d'utiliser son poly pour la promotion X2004, ce dont nous le remercions chaleureusement.

P. Chassignet a prêté son concours à l'élaboration des classes de Graphe utilisées pour le poly. Il a également répondu à de multiples questions de tous ordres. Merci à P. Baptiste pour les discussions sur l'optimisation combinatoire. Merci aussi à J. Cervelle pour avoir répondu à des questions pointues sur l'utilisation de Java, version IIIe millénaire, ainsi que pour la relecture. Que les premiers lecteurs soient également remerciés : F. Pottier pour sa méticulosité, N. Sendrier, A. Cohen ; G. Schaeffer.

**Polycopié du cours INF431, Version 1.3c [14 janvier 2008].**

## Notations

Le cardinal d'un ensemble  $E$  sera noté  $|E|$ .

## Algorithme, pseudocode et programme

Quand on commence à programmer, on étudie et on implante des algorithmes simples, dont l'écriture coule de source. Arrivés à ce niveau de votre apprentissage, nous allons voir des algorithmes de plus en plus complexes, et se pose le problème de leur description indépendante d'un langage particulier et en négligeant dans un premier temps les détails d'implantation qui pourraient avoir tendance à compliquer inutilement la vue d'ensemble de l'algorithme. Pour résumer, nous décrirons un algorithme de manière synthétique et générique, à l'aide d'un langage intermédiaire, le *pseudocode*. La correction de l'algorithme proviendra souvent d'un théorème, l'étude de sa complexité sera esquissée à partir du pseudocode, et dépendant souvent de l'implantation choisie.

Quel pseudocode et quelles propriétés demander ? Nous modifions le pseudocode présenté de façon très classique dans [6, p. 17-18]. La syntaxe est très proche de langages de type Pascal, C, ou Java, et les principes du langage sont habituels en Java (cf. les objets).

1. Le symbole `//` indique que le reste de la ligne est un commentaire.
2. Les instructions seront terminées par un `;`.
3. L'indentation indique une structure de bloc ; on se passera, sauf cas particulier, d'indicateurs de type **début** ou **fin** pour alléger la lecture.
4. Les affectations entre type simple s'écriront `x <- 3` ; les tests d'égalité `==`. On utilisera les opérateurs de post et pré-incrémentation de Java à l'occasion.
5. On utilisera des boucles **tantque**, **pour**, **répéter**. Les variables d'itération continueront d'exister à la sortie des itérations.
6. Les variables sont locales aux procédures, nous n'emploierons pas de variable globale, sauf indication explicite.
7. `A[i]` désigne le  $i$ -ème élément du tableau `A`, et `A[2..j]` représente le sous-tableau de `A` d'indices compris entre 2 et  $j$ . Les tableaux seront numérotés à partir de 0. Les tableaux bi-dimensionnels seront notés `A[i][j]`.
8. Les données composées seront traitées comme des objets, constitués d'*attributs* ou *champs*. Comme en Java, un objet sera vu comme un pointeur sur les données. Si `f` est un champ de l'objet `x`, on notera `x.f`. La longueur d'un tableau `A` sera `longueur(A)`. Si `x` et `y` sont deux objets, faire `x <- y` fait que `x` et `y` pointent sur le même objet. Si un objet ne contient rien, il contient `NIL`. Dans le cas où on voudra recopier les objets champ par champ, on écrira `x := y`.
9. Les paramètres sont passés à une procédure *par valeur*. Les objets sont passés comme en Java.

10. Comme en Java, le *et* et le *ou* logiques sont *court-circuitants* (on dit aussi  *paresseux*) : quand on évalue  $x \text{ ET } y$ , on évalue  $x$ , et seulement s'il est vrai on évalue  $y$ .
11. Une liste sera écrite  $L = (x_1, \dots, x_n)$ , on écrira la concaténation

```
L # z = (x1, ..., xn, z)
```

ou

```
z # L = (z, x1, ..., xn)
```

Le premier élément d'une liste  $L = (x_1, x_2, \dots, x_n)$  sera récupéré comme  $t\hat{e}te(L)=x_1$ ; écrire

```
y <- t\hat{e}te(L);
```

fera que  $y$  contiendra  $x_1$  et il restera  $(x_2, \dots, x_n)$  dans  $L$ , ce que l'on pourra noter  $reste(L)$  si besoin est, de même que les éléments sauf le dernier seront désignés par  $d\acute{e}but(L)$ . La queue de  $L$  sera le dernier élément,  $queue(L)=x_n$ . On aura donc formellement :

```
L = t\hat{e}te(L) # reste(L) = d\acute{e}but(L) # queue(L)
```

12. Un ensemble sera noté  $\{a, b, \dots\}$ . Notons que des implantations spécifiques pourront être données quand nécessaire.
13. Si l'on veut afficher quelque chose à l'écran, on utilisera l'instruction *écrire*.

Donnons un exemple simple : recherchons le minimum d'un tableau. Notons que le type du tableau n'est pas explicité, et qu'on suppose implicitement qu'il existe un ordre sur les éléments du tableau.

```
rechercheMinimum(A)
  imin <- 0;
  pour i <- 1 à longueur(A)-1 faire
    si A[i] < A[imin] alors
      // on a trouvé un minimum local
      imin <- i;
  retourner A[imin];
```

Il sera facile de traduire ce pseudocode en Java, quel que soit le type de  $A$ .



Première partie  
Programmation



# Chapitre 1

## Programmer avec Java

Ce chapitre ne se propose pas comme une description exhaustive de Java, mais plutôt comme un guide à l'utilisation de certains traits avancés, comme l'héritage et les classes abstraites. Il ne saurait se substituer à un cours de langages de programmation, comme par exemple celui d'année 3. On trouvera des éléments complémentaires dans [9]. On pourra consulter également les tutoriels de Java sur la page de Sun<sup>1</sup>. Notre optique est d'être capable d'utiliser ces notions quand nous aborderons les graphes.

Plus philosophiquement, les langages modernes tournent autour de cette question : comment écrire au kilomètre des programmes corrects, lisibles et efficaces ? Une des façons de répondre est d'utiliser des classes prédéfinies, efficaces, standardisées, et nous en reparlerons au chapitre 14. Nous nous intéresserons plutôt ici aux mécanismes qui permettent la réutilisation et le partage de ces ressources.

### 1.1 Modularité

#### 1.1.1 Motivation

La modularité est un concept fondamental de la programmation. Dans un programme, on cherche toujours où on doit mettre quelles données. Pour s'y retrouver, il est beaucoup plus simple de pouvoir contrôler quelles données sont accessibles à qui (fonctions, classes, programmes, etc.). Dans certains langages, on a accès à essentiellement deux niveaux : une variable peut être *locale* à une fonction (ou plus généralement à un *bloc* de code) et n'est pas vue de l'extérieur de la fonction ; ou bien elle est *globale*, ce qui fait qu'elle est accessible à toutes les fonctions à la fois. Ces deux niveaux se révèlent rapidement insuffisants.

On est donc amené à introduire des contrôles plus stricts. On peut rassembler dans une même bibliothèque (appelée généralement *module* dans ce contexte), toutes les fonctions opérant sur des objets communs. L'idée derrière les modules est bien évidemment celui de la réutilisation et de la composition de petits modules pour en faire des plus gros, ce qui conduit à de gros programmes, etc.

---

<sup>1</sup><http://java.sun.com/docs/books/tutorial/index.html>

De même, on peut élargir les types des données pour fabriquer un type adapté à un cas spécifique, des types peuvent être locaux ou globaux, etc. En Java, types et modules se confondent dans la notion de classe, qui est à la fois type utilisateur et module contenant les fonctions associés à ce type.

### 1.1.2 Modules

Très généralement, un module se compose de deux parties :

- l'*interface*<sup>2</sup> accessible depuis l'extérieur du module ;
- l'*implémentation* accessible de l'intérieur du module.

Un des avantages de cette dichotomie est de permettre d'utiliser le principe d'*encapsulation*. On définit les fonctionnalités du module dans la partie interface (données partagées, types, fonctions), car c'est ce qui intéresse l'utilisateur, et on en réalise une implantation dans la partie implémentation. Prenons l'exemple classique de la file d'attente. On a besoin de savoir faire trois opérations : construction, ajouter, supprimer. Mais on peut implanter les files d'au moins deux façons : par un tableau géré de façon circulaire, ou bien par une liste. L'extérieur n'a pas besoin de savoir quelle représentation est utilisée, et parfois, on peut la changer sous ses pieds.

Dans certains langages (comme Pascal), un module contient une interface et une implantation, et on ne peut avoir plusieurs implantations correspondant à la même interface. En Java, l'utilisation des classes abstraites permet de résoudre ce problème (cf. section 2.2.2).

Reprenons le cas d'une file d'attente. On peut écrire :

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

    public FIFO(int n){
        debut = 0; fin = 0;
        pleine = n == 0; vide = true;
        contenu = new int[n];
    }
    public 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;
    }
    public static int supprimer(FIFO f){
        if(f.vide)
            throw new Error("File Vide.");
    }
}
```

<sup>2</sup>Attention : en Java, interface est un mot clef pour désigner autre chose, cf. section 2.2.2.

```

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;
}

```

On rend tous les champs privés (voir section 2.2.1) de sorte que l'utilisateur ne puisse utiliser directement l'implantation (ce qui rendrait son code sensible à tout changement de la définition de FIFO). Par contre, ce qui est public, c'est le constructeur, ainsi que les fonctions d'accès.

On pourrait également écrire une autre classe avec la même interface (et les mêmes signatures de fonction), mais cette fois ci utilisant une liste chaînée en interne :

```

public class FIFO{
  private Liste debut, fin;
  public FIFO(int n){ debut = null; fin = null; }
  public 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;
    }
  }
  public 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;
    }
  }
}

```

Ces deux exemples s'excluent mutuellement, si l'on cherche à garder le même nom à la classe. Nous verrons au chapitre 2 que les classes abstraites permettent de décrire plusieurs implantations réalisant les mêmes fonctionnalités. Nous pouvons également utiliser des interfaces, voir la section 1.3.

## 1.2 Programmation par objets

### 1.2.1 Motivations

Les premiers langages informatiques relèvent d'une programmation dite *procédurale* où on applique des fonctions (ou procédures) à des données. La programmation dirigée par les données (ou programmation par objet) est un paradigme plus récent. Pour simplifier, la première vision du monde est celle de calculs appliqués à des paramètres d'entrée (donc une approche très fonctionnelle), alors que la seconde considère les objets comme la quantité importante, et qu'un objet peut posséder ses propres fonctions de traitement.

En Java, un objet est une instance d'une classe. Il a un état (la valeur de ses champs de données) et un ensemble de méthodes attachées. Les données statiques (*static*) sont partagées par tous les objets de la classe.

Quels avantages a-t-on à programmer objet ? Même si la modularité n'impose pas la programmation objet, celle-ci bénéficie énormément de cette possibilité, tant les deux ont le même objectif de rassembler en un même endroit (du programme) les données et fonctions applicables à un objet. Un autre avantage est celui de l'héritage : on peut définir des classes, qu'on va étendre dans un but plus spécifique, et donc aboutir à une programmation incrémentale, qui nous rapproche de notre objectif de l'écriture de gros programmes. C'est le cas de l'AWT (*Abstract Windowing Toolkit*) qui permet de faire des dessins en Java.

On peut se poser diverses questions pour choisir : préfère-t-on contrôler les fonctions ou les données ? Les petits programmes peuvent se faire indifféremment dans les deux styles. Si le programme devient gros (disons au-delà de 10,000 lignes), la programmation objet a des avantages certains. Entrent alors en ligne de compte des considérations comme la stratégie de modification ou d'évolution du code, et du côté incrémental que cela peut représenter, ce qui est très prisé dans l'industrie.

La programmation objet est quasi impérative quand on utilise des classes prédéfinies déjà en syntaxe objet (AWT), ce que nous verrons plus loin.

Gardons la tête froide. En dernière analyse, c'est une affaire de goût. . . .

### 1.2.2 L'héritage par l'exemple

Nous allons considérer l'exemple ultra-classique des points et des points colorés. On commence par une classe `Point` qui code un point du plan par ses coordonnées `x` et `y` :

```
class Point{
    double x, y;
}
```

On décide de créer un nouveau type qui aura la particularité d'être un point avec une couleur :

```

class PointC extends Point{
    final static int JAUNE = 0, ROUGE = 1;
    int c;
    PointC(double x0, double y0, int col){
        x = x0; y = y0; c = col; // [1]
    }
    public static void main(String[] args){
        PointC pc = new PointC(3, 4, JAUNE);
    }
}

```

La classe `PointC` est une *sous-classe* de `Point`. Elle *hérite* des champs et des méthodes définis dans sa classe père, ce qui explique la syntaxe de la ligne [1] : à la création d'un `PointC`, on a déjà à sa disposition les deux champs `x` et `y` hérités de la classe `Point`.

Complétons l'exemple :

```

class Point{
    double x, y;
    static int NP = 1;
    static void afficherAbscisse(Point p){
        System.out.println("P: "+p.x);
    }
}
class PointC extends Point{
    final static int JAUNE = 0, ROUGE = 1;
    int c;
    PointC(double x0, double y0, int col){
        x = x0; y = y0; c = col;
    }
    public static void main(String[] args){
        PointC pc = new PointC(3, 4, JAUNE);
        afficherAbscisse(pc);
        System.out.println("NP="+NP);
    }
}

```

Il y a conversion *implicite* de `PointC` en `Point`, dans l'appel de la méthode `afficherAbscisse` de la classe `Point`. D'autre part, la variable statique `NP` est elle aussi accessible depuis `PointC`.

Dans la sous-classe, on ajoute ou on redéfinit des champs ou méthodes : on dit qu'on a *spécialisé* (en anglais, c'est l'*overriding*) ces champs ou méthodes. Par exemple, dans le code :

```

class Point{

```

```
...
    static void afficherAbscisse(Point p){
        System.out.println("Point: "+p.x);
    }
}
class PointC extends Point{
    ...
    static void afficherAbscisse(Point p){
        System.out.println("PointC: "+p.x);
    }
    public static void main(String[] args){
        PointC pc = new PointC(3, 4, JAUNE);
        afficherAbscisse(pc);
        Point.afficherAbscisse(pc);
    }
}
```

on a spécialisé dans PointC la méthode afficherAbscisse. On peut également spécialiser des méthodes d'objet :

```
class Point{
    ...
    void afficher(){
        System.out.println("p: "+x);
    }
}
class PointC extends Point{
    ...
    void afficher(){
        System.out.println("pc: "+x);
    }
    public static void main(String[] args){
        PointC pc = new PointC(3, 4, JAUNE);
        Point p = new Point();
        p.x = 1;

        pc.afficher();
        p.afficher();
    }
}
```

### 1.2.3 Propriétés de l'héritage

En Java, l'héritage est dit *simple*<sup>3</sup>, c'est-à-dire qu'une classe peut être sous-classe d'une autre classe et d'une seule. Par contre, une classe peut avoir plusieurs sous-classes. Les diagrammes d'héritage sont donc des arbres. Cela facilite grandement le choix de la méthode à appliquer à un objet, selon le principe suivant : on choisit *statiquement* la méthode la plus spécialisée, c'est-à-dire appartenant à la plus petite sous-classe connue contenant l'objet.

La classe père est accessible par le mot clef **super** (un peu comme si l'on écrivait (ClassePere)**this**); on ne peut l'utiliser que dans des méthodes d'objets : **super**.f est la méthode f de la classe parente; **super**() (avec d'éventuels arguments) est le constructeur (appelé par défaut) de la classe parente. Par exemple :

```
class PointC extends Point{
    final static int JAUNE = 0, ROUGE = 1;
    int c;
    PointC(double x0, double y0, int col){
        super(); // facultatif
        x = x0; y = y0; c = col;
    }
    public static void main(String[] args){
        PointC pc = new PointC(3, 4, JAUNE);
    }
}
```

Dans le cas d'un constructeur explicite pour la classe père, on est obligé d'appeler **super** :

```
class Point{
    double x, y;
    Point(double x0, double y0){
        x = x0; y = y0;
    }
}
class PointC extends Point{
    final static int JAUNE = 0, ROUGE = 1;
    int c;
    PointC(double x0, double y0, int col){
        super(x0, y0); // nécessaire
        c = col;
    }
    public static void main(String[] args){
        PointC pc = new PointC(3, 4, JAUNE);
    }
}
```

<sup>3</sup>contrairement à C++, Smalltalk ou Ocaml, dans lesquels l'héritage est *multiple*.

```
}

```

**Exemple.** En Java, on peut masquer des champs. Par exemple, le code suivant est (malheureusement ?) licite :

```
class PointC extends Point{
    final static int JAUNE = 0, ROUGE = 1;
    int x;
    int c;
    ...
}
```

Le nouveau `x` masque l'ancien, qui est toujours accessible par **super**.

### 1.2.4 Contrôle d'accès et sous-classes

Les champs ou méthodes d'une classe peuvent être :

- **public** pour permettre l'accès depuis toutes les classes ;
- **private** pour restreindre l'accès aux seules expressions ou fonctions de la classe courante ;
- par défaut pour autoriser l'accès depuis toutes les classes du même paquetage ;
- **protected** pour restreindre l'accès aux seules expressions ou fonctions de sous-classes de la classe courante.
- Un champ **final** ne peut être redéfini (on peut donc optimiser sa compilation).
- une classe **final** ne peut être spécialisée, c'est le cas de la classe `String` par exemple.
- une méthode peut être déclarée **final** (pas besoin de déclarer toute la classe **final**).

Les méthodes spécialisées doivent fournir au moins les mêmes droits d'accès que les méthodes originellement définies.

### 1.2.5 Un peu de typage

#### Théorie

Rappelons qu'un type est un ensemble de valeurs partageant une même propriété. En Java, il y a des types primitifs : **int**, **char**, etc ; une classe est un type. On a déjà rencontré des classes prédéfinies comme la classe `String`.

Comment le compilateur vérifie-t-il le typage ? Il s'agit d'une vérification statique (à la compilation) contrairement au typage dynamique (à l'exécution). L'intérêt du typage statique est de permettre la détection d'erreurs le plus tôt possible, mais aussi d'optimiser le code, et d'assurer une certaine sécurité (pas d'atteinte à la mémoire par exemple). Le typage dynamique peut être dans certains cas plus précis (prendre l'exemple du cas de `if(a) return 1; else return false;`).

Il n'est pas question ici de faire un cours de théorie des types. Nous renvoyons au cours Principes des Langages de Programmation d'année 3, ainsi qu'aux livres de Benjamin Pierce ou de Abadi-Cardelli.

### Et en Java

En Java, chaque objet possède un certain type lors de sa création. L'opérateur **instanceof** teste l'appartenance d'un objet à une classe. Ainsi :

```
if(p instanceof PointC)
    System.out.println ("couleur = " + p.c);
```

Donc l'expression `(PointC)p` équivaut à :

```
if(p instanceof PointC)
    p
else
    throw new ClassCastException();
```

On parle de sous-classe comme on parle de sous-type. On note  $x : t$  pour dire que  $x$  est de type  $t$  et  $t <: t'$  signifie  $x : t \Rightarrow x : t'$  pour tout  $x$ . On dit que  $t$  est un *sous-type* de  $t'$ . En Java, les types suivants sont des sous-types les uns des autres, et il y a conversion explicite si nécessaire :

```
byte <: short <: int <: long
float <: double
char <: int <: long
```

Une classe est un type; une sous-classe un sous-type. On propage la notation :  $C <: C'$  si  $C$  est une sous-classe de  $C'$ .

**Ex.** `PointC <: Point`.

On a déjà dit qu'en Java, l'héritage était simple. Il existe ainsi un arbre qui contient toute la hiérarchie des classes, dont la racine est la classe `Object` :

$$\forall C \quad C <: \text{Object}$$

( $C$  est une classe ou un tableau quelconque). Les méthodes de `Object` sont `clone`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `wait`, `equals`, `toString`.

Seules `equals`, `toString`, `finalize` peuvent être redéfinies. Comme conséquence de ce qui précède, *tous* les objets contiennent ces méthodes.

Comment se raccrochent les types primitifs dans ce tableau ? On convertit les scalaires **int**, **float**, **char**, ... en objets avec un « conteneur » (*autoboxing*) :

```
int x = 3;
Objet xobj = new Integer(x);
int y = xobj.intValue();
```

### Quelques exemples extrêmes

Le code qui suit est valide, mais est (souvent) déconseillé<sup>4</sup> :

```
class A{
    int x;

    A(){ x = 1; }
}
class B extends A{
    B(){ x = 2; }
    public static void main(String[] args){
        B b = new B();
        A a = new B();
        System.out.println(b.x);
        System.out.println(a.x);
    }
}
```

Le programme affiche :

```
2
2
```

Le *type créé* de `a` est `B`, mais son *type apparent* est `A`. La conversion `A a = new B()` marche puisque `B` est sous-classe de `A`.

Cette propriété permet d'écrire :

```
class Point{
    double x, y;
    static void translation(Point p,
                            double dx, double dy){
        p.x += dx; p.y += dy;
    }
}
class PointC extends Point{
    final static int JAUNE = 0, ROUGE = 1;
    int c;
    PointC(double x0, double y0, int col){
        x = x0; y = y0; c = col;
    }
    public static void main(String[] args){
        PointC pc = new PointC(3, 4, JAUNE);
        translation(pc, 1, 2);
    }
}
```

<sup>4</sup>Cette section peut être sautée en première lecture.

```
}

```

Par contre, le code suivant ne peut marcher :

```
class Point{
    ...
    static Point translation(Point p,
                             double dx, double dy){
        Point pt = new Point();
        pt.x = p.x + dx; pt.y = p.y + dy;
        return pt;
    }
}
class PointC extends Point{
    ...
    public static void main(String[] args){
        PointC pc = new PointC(3, 4, JAUNE);
        pc = (PointC)translation(pc, 1, 2);
    }
}
```

Il n'y a pas de conversion père → fils. La compilation marche (avec le cast), mais pas l'exécution (ClassCastException).

### Héritage et surcharge

La surcharge est déterminée à la *compilation*. On parle de liaison *retardée* : l'héritage permet d'appeler une méthode de la même classe sans savoir exactement quelle sous-classe va l'instancier.

Pour illustrer cette notion, on demande quelle est la valeur affichée par le programme suivant ?

```
class C{
    void f(){
        g();
    }
    void g(){
        System.out.println(1);
    }
}
class D extends C{
    void g(){
        System.out.println(2);
    }
    public static void main(String[] args){
        D x = new D();
    }
}
```

```

    x.f();
  }
}

```

La réponse est 2. En effet, que se passe-t-il? Le compilateur ne se plaint pas, car à la création,  $x$  étant de type  $D <: C$ , il possède bien une méthode  $f$ . À l'exécution, on utilise la règle suivant laquelle on applique la méthode de la plus petite classe possible contenant  $x$ , ici la classe  $D$ , qui a bien une méthode  $g$ , qui affiche 2!

Notons que ce genre de propriété est difficile à utiliser quand on débute. Il n'accroît pas la lisibilité du code non plus, et donc nous engageons le programmeur à ne l'utiliser que quand la situation l'exige, et elle peut très bien ne jamais se produire...

**Exercice 1.2.1** Quel est le résultat produit par le programme suivant quand  $T, T' \in \{A, B\}$  et  $U, U' \in \{A, B\}$  avec  $T' <: T$  et  $U' <: U$ ?

```

class A{
    public static void main(String[] args){
        T x = new T'(); U y = new U'();
        System.out.println (x.f(y));
    }

    int f(A y) { return 1; }
}

class B extends A{
    int f(B y) { return 2; }
}

```

### 1.3 Les interfaces de Java

Les *interfaces* à la Java permettent de passer des *contrats* entre programmeurs... Par exemple, on peut imaginer une interface décrivant ce qu'on attend d'une voiture, laissant une implantation particulière à chaque constructeur.

Une interface possède les propriétés suivantes :

- tous les champs sont indéfinis. Ses champs de données sont constants ; ses méthodes sont abstraites et publiques.
- Une interface peut *spécialiser* une autre interface.
- Une classe peut *implémenter* une ou plusieurs interfaces.
- Il n'y a pas de fonctions statiques, ou champs de données modifiables dans une interface.
- Les interfaces sont une bonne manière d'exiger la présence de certains champs dans une classe.

Reprenons l'exemple de la FIFO. On peut définir une interface qui décrit les propriétés attendues d'une FIFO :

```

interface FIFO{
    boolean estVide();
    void ajouter(int x);
    int supprimer();
}

```

On peut alors en réaliser deux implantations distinctes :

```

public class FIFO_T implements FIFO{
    private int debut, fin;
    private boolean pleine, vide;
    private int[] contenu;

    public FIFO_T(int n){
        debut = 0; fin = 0;
        pleine = n == 0; vide = true;
        contenu = new int[n];
    }
    public boolean estVide(){
        return vide;
    }
    public void ajouter(int x){
        if(pleine) throw new Error("File Pleine.");
        contenu[fin] = x;
        fin = (fin + 1) % contenu.length;
        vide = false; pleine = fin == debut;
    }
    public int supprimer(){
        if(vide)
            throw new Error("File Vide.");
        int res = contenu[debut];
        debut = (debut + 1) % contenu.length;
        vide = fin == debut; pleine = false;
        return res;
    }
}

```

et celle utilisant une liste :

```

public class FIFO_L implements FIFO{
    private Liste debut, fin;
    public FIFO(int n){ debut = null; fin = null; }
    public boolean estVide(){
        return (debut == null);
    }
}

```

```

public void ajouter(int x){
    if(fin == null)
        debut = fin = new Liste(x);
    else{
        fin.suivant = new Liste(x);
        fin = fin.suivant;
    }
}
public int supprimer(){
    if(debut == null) throw new Error("File Vide.");
    else{
        int res = debut.val;
        if(debut == fin)
            debut = fin = null;
        else debut = debut.suivant;
        return res;
    }
}
}
}

```

On peut alors écrire un algorithme général qui utilise les FIFO :

```

public class C{
    public static void Algo(FIFO f){
        while(! f.estVide()){
            int x = f.supprimer();
            System.out.println(x);
        }
    }
    public static void main(String[] args){
        FIFO_L L = new FIFO_L(10);
        for(int i = 0; i < 10; i++)
            L.ajouter(i);
        Algo(L);
        FIFO_T T = new FIFO_T(20);
        for(int i = 0; i < 10; i++)
            T.ajouter(i);
        Algo(T);
    }
}
}

```

En clair, on peut utiliser une **interface** comme un type.

### 1.3.1 Spécialisations multiples

Donnons encore un autre exemple, où on spécialise deux interfaces à la fois. On va se donner une interface de pile :

```
interface Pile{
    boolean estVide();
    void empiler(int x);
    void depiler();
}
```

On peut réaliser une classe de liste qui implante les deux et qui va utiliser un tableau :

```
public class MaListe implements FIFO, Pile{
    private int debut, fin;
    private boolean pleine, vide;
    private int[] contenu;

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

    boolean estVide(){ ... } // même signature pour FIFO et Pile!
    void ajouter(){ ... }
    int supprimer(){ ... }
    void empiler(int x){ ... }
    int depiler(int x){ ... }
}
```

On peut alors imaginer un programme de distribution de cartes :

```
public class Jouer{
    public static void main(String[] args){
        MaListe jeu = new MaListe(32);
        for(int i = 0; i < 32; i++){
            jeu.ajouter(i); // on enfile dans l'ordre 0, 1, ..., 31
        }
        while(! jeu.estVide()){
            // on affiche dans l'ordre 0, 1, ..., 31
            System.out.println(jeu.depiler());
        }
    }
}
```

Nous ne sommes plus très loin de la classe prédéfinie `LinkedList`.

## 1.4 Les génériques de Java

Java possède (depuis la version 5.0) une notion de polymorphisme, ce qui permet d'écrire une seule fois certaines classes, comme par exemple une classe de liste d'objets<sup>5</sup>.

Pour assurer la réutilisabilité du code, il est commode de fabriquer du code général, mais qui peut conduire à des problèmes, qui ne seront détectés qu'à l'exécution.

L'exemple de base est :

```
public class Box {

    private Object object;

    public void add(Object object) {
        this.object = object;
    }

    public Object get() {
        return object;
    }
}
```

qui se contente mettre un objet dans une boîte. On peut alors place n'importe quel type d'objet dans la boîte :

```
public class BoxDemo1 {

    public static void main(String[] args) {

        // ONLY place Integer objects into this box!
        Box integerBox = new Box();

        integerBox.add(new Integer(10));
        Integer someInteger = (Integer)integerBox.get();
        System.out.println(someInteger);
    }
}
```

Ce code ne nécessite pas de cast, puisque tout type est sous-type d'Object, donc on peut convertir un fils en son père sans problème.

Malheureusement, rien n'interdit d'écrire également :

```
public class BoxDemo2 {

    public static void main(String[] args) {
```

<sup>5</sup>Nous nous inspirons ici très fortement d'un des tutoriaux de Sun, <http://java.sun.com/docs/books/tutorial/java/generics/generics.html>.

```

// ONLY place Integer objects into this box!
Box integerBox = new Box();

// Imagine this is one part of a large application
// modified by one programmer.
integerBox.add("10"); // note how the type is now String

// ... and this is another, perhaps written
// by a different programmer
Integer someInteger = (Integer)integerBox.get();
System.out.println(someInteger);
}
}

```

qui va se compiler correctement (car String est sous-classe d'Object lui aussi). Mais à l'exécution :

```

Exception in thread "main"
    java.lang.ClassCastException:
        java.lang.String cannot be cast to java.lang.Integer
        at BoxDemo2.main(BoxDemo2.java:6)

```

Java implante un type de polymorphisme qui nous protège contre ce genre d'erreur :

```

public class Box<T> {

    private T t; // T stands for "Type"

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}

```

qui permet d'écrire :

```

public class BoxDemo3 {

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        Integer someInteger = integerBox.get(); // no cast!
    }
}

```

```

        System.out.println(someInteger);
    }
}

```

Remarquez la syntaxe

```
Box<Integer> integerBox = new Box<Integer>();
```

qui permet de définir une boîte d'Integer. En cas d'écriture erronée, on obtiendra une erreur à la compilation :

```

BoxDemo3.java:5: add(java.lang.Integer) in Box<java.lang.Integer>
cannot be applied to (java.lang.String)
    integerBox.add("10");
                  ^
1 error

```

Le symbole T est une sorte de paramètre de type, qui n'apparaît pas sous forme de **T.class** à aucun endroit.

On peut définir des boîtes à types multiples simplement en utilisant des symboles différents : `Box<T,U>` .

Par convention, on utilise des symboles mono-caractères en majuscule suivant le tableau

- \* E - Element (used extensively by the Java Collections Framework)
- \* K - Key
- \* N - Number
- \* T - Type
- \* V - Value
- \* S,U,V etc. - 2nd, 3rd, 4th types

On peut également écrire des interfaces génériques, comme :

```

public class LinkedList<E> ... {
    void add(int i, E x) { ... }
    E get(int i) { ... }
    E getFirst() { ... }
    ListIterator<E> listIterator(int i) { ... }
}
public interface ListIterator<E>
    extends Iterator<E> ... {
    boolean hasNext();
    E next();
}

```

Par exemple, une liste d'entiers sera simplement utilisée de la façon suivante :

```
LinkedList<Integer> L = new LinkedList<Integer>();  
L.add(1);
```

Signalons un inconvénient (expliqué dans la documentation de Sun). On ne peut écrire `E x = new E();` ou `E[] t = new E[10];` pour des raisons de typage. Cela nous conduira à quelques contorsions plus loin, mais ces classes sont très faciles à utiliser malgré tout.

### 1.4.1 Restreindre à certains types

Comment empêcher l'utilisation d'un type trop général ? Il suffit de donner une *borne supérieure* au type utilisable, comme dans

```
public class Box<T extends Number> {
```

On ne pourra alors utiliser la boîte que pour des sous-classes de la classe `Number`. On aurait la même syntaxe pour des interfaces.



## Chapitre 2

# Introduction au génie logiciel

Après avoir écrit un programme tout seul, il faut maintenant songer à l'étape suivante, l'écriture à 2, avant peut-être de participer à une aventure plus ambitieuse. Les quelques lignes qui suivent vous donneront quelques idées sur le sujet et nous reviendrons sur l'apport de Java à la fin du chapitre.

### 2.1 Généralités

#### 2.1.1 Quelques chiffres

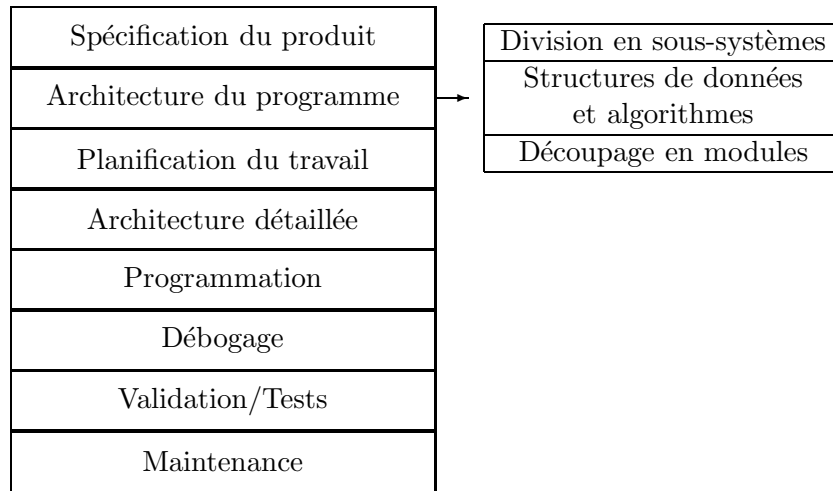
Le code source de Windows XP représente 50 millions de lignes de code, Linux environ 30 millions. Comment peut-on gérer autant de lignes de code, et autant de programmeurs supposés ?

Dans un article des *Communications of the ACM* (septembre 2006), des données rassemblées par le *Quantitative Software Management* sont présentées. L'étude a porté sur 564 projets récents, réalisés dans 31 entreprises dans 16 branches dans 16 pays. Il en ressort qu'un projet moyen requiert moins de 7 personnes pour une durée inférieure à 8 mois, pour un coût moyen inférieur à 58 homme-mois, avec comme langage encore majoritaire COBOL, en passe d'être détrôné par Java, représentant moins de 9,200 lignes de code.

Depuis l'avènement de l'informatique, de nombreux chercheurs et praticiens s'interrogent sur les aspects d'organisation des gros programmes en grosses (?) équipes. Une des bibles de référence est toujours [4].

#### 2.1.2 La chaîne de production logicielle

Le schéma qui suit permet de comprendre les différentes phases de la création d'un logiciel conséquent.



Comme on peut le constater, *un logiciel ne se résume pas à la programmation*. Malgré tout, la phase de programmation reste l'endroit où on a le plus de prise sur le produit.

La phase de spécification est importante et conditionne le reste. Les problèmes que l'on doit se poser sont généralement :

- Que doit faire le programme ? Qui doit l'utiliser ?
- Quelles sont les opérations spécifiques ?
- Quel doit être le temps de réponse ?

La documentation d'un programme (petit ou gros) est fondamentale. Il faut commencer à l'écrire dès le début, avec mise à jour à chaque fois qu'on écrit une fonction.

Une architecture excellente peut être gâtée par une programmation médiocre ; une excellente programmation ne peut pas rattraper complètement une architecture désastreuse.

Zoomons un peu sur cette phase, qui restera pour votre PI une des phases centrales. La division en sous-systèmes permet de se mettre d'accord sur l'interface (ici pris au terme d'entrée des données, formatage des sorties) en liaison avec le moteur du programmes (fabriquant les données de sortie). Cette phase recense également les bases de données, les problèmes de communications, l'aspect graphique ; etc.

C'est dans cette phase que la modularité s'exprime le mieux. On cherche toujours la simplicité, mais aussi une forme de protection contre les changements (surtout dans l'interface). Cela permet de réaliser une bonne division du travail et de minimiser les interactions.

### 2.1.3 Déboguer

Déboguer est un art qui demande patience, ingéniosité, expérience, un temps non borné et... du sommeil !

- Il est bon de se répéter les trois lois du débogage dès que tout va mal :
- tout logiciel complexe contient des bogues ;

- le bogue est probablement causé par la dernière chose que vous venez de modifier ;
- si le bogue n'est pas là où vous pensez, c'est qu'il est ailleurs ;
- Un bogue algorithmique est beaucoup plus difficile à trouver qu'un bogue de programmation pure.
- On ne débogue pas un programme qui marche !

Au-delà de ces boutades, déboguer relève quand même d'une démarche scientifique : il faut isoler le bogue et être capable de le reproduire. Déboguer est donc très difficile dans les programmes non déterministes (attention aux générateurs aléatoires – mieux vaut les débrancher au départ ; parallélisme, calculs distribués, etc.).

#### 2.1.4 Tester

Le test d'un programme est une activité prenante, et nécessaire, du moins quand on veut réaliser un programme correct, et non pas un produit à délivrer au client, avec les bogues corrigés s'il paie pour avoir les upgrades. . .

Les tests de fonctionnalité du programme sont impératifs, font partie du projet et sont de la responsabilité immédiate du programmeur. Plus généralement, on parle d'*alpha test* pour désigner les tests faits par l'équipe de développement ; le code est alors gelé, seuls les bogues corrigés. En phase de *beta test*, les tests sont réalisés par des testeurs sélectionnés et extérieurs à l'équipe de développement.

Écrire des tests n'est pas toujours facile. Ils doivent couvrir tous les cas normaux ou anormaux (boîte de verre, boîte noire, etc.). Souvent, tester toutes les branches d'un programme est tout simplement impossible.

#### 2.1.5 Analyser les performances (benchmarks)

Mesurer la vitesse de son programme est également une bonne idée. Même si tous les programmes du monde ne peuvent se terminer par retour chariot, essayer de comprendre où on passe sont temps est primordial, et un programme rapide est plus vendeur.

On peut écrire un programme de test qui affiche les paramètres pertinents. On peut tester 2 fonctions et produire deux courbes de temps, qu'il reste à afficher et commenter (*xgraphic* ou *gnuplot* en Unix).

Si l'algorithme théorique est en  $O(n^2)$ , on teste avec  $n$ ,  $2n$ ,  $3n$  et on regarde si le temps varie par un facteur 4, 9, . . . Si non, on regarde fonction par fonction.

Il ne reste plus qu'à commenter, déduire, etc. Cela exprime le côté expérimental de l'informatique.

#### 2.1.6 Architecture détaillée

Étudiez l'architecture soigneusement et vérifiez qu'elle marche avant de continuer.

La méthode généralement employée est celle de l'analyse descendante et du raffinement. Prenons l'exemple simple du comptage du nombre de mots dans un fichier. Les actions nécessaires sont :

- ouvrir le fichier ;
- aller au début du fichier ;

- tant que la fin du fichier n'est pas atteinte
  - lire un mot ;
  - incrémenter le nombre de mots ;
- afficher le nombre de mots ;
- fermer le fichier.

On peut encore raffiner : comment lire un mot, etc.

Rajoutons quelques règles :

- Toujours commencer par le commencement (!) ;
- oublier Java (temporairement) ;
- repousser les détails de programmation au plus tard possible ;
- ne pas descendre de niveau tant que vous n'êtes pas convaincu(e)s que le niveau actuel est satisfaisant ;
- si un problème apparaît, c'est sans doute qu'il trouve sa source au niveau immédiatement supérieur. Remonter et régler le problème.

### 2.1.7 Aspects organisationnels

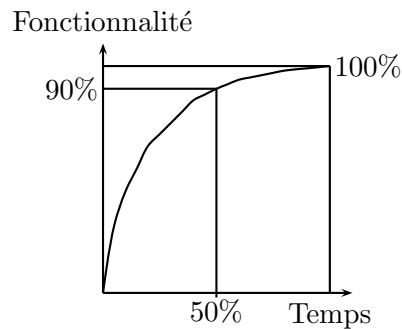
#### Planification du travail

Dans son bestseller, *The mythical Man-Month*, F. P. Brooks (qui a conçu le système d'exploitation de l'IBM 360, au début des années 1970), donne quelques règles empiriques pour un "bon" projet.

La répartition du temps devrait être celle-ci :

- 1/3 de spécification ;
- 1/6 de programmation ;
- 1/4 de test (alpha) ;
- 1/4 d'intégration et test (beta).

Il faut également garder en tête la fameuse courbe suivante, qui décrit l'état d'avancement vers le but final :



### 2.1.8 En guise de conclusion provisoire...

Un programme ressemble à un pont :

- Plus le projet est grand, plus il faut soigner l'architecture et le planning. Les problèmes humains ne peuvent être négligés.

- Découvrir les erreurs très vite est essentiel (ou *la découverte tardive est catastrophique*).
- Les erreurs peuvent être désastreuses (Ariane 5 – 1 milliard de dollars).
- Utiliser des préfabriqués permet de gagner du temps.

Un programme n'est pas un pont :

- Le logiciel est purement abstrait ; il est *invisible*, car il n'est vu que par son action sur un matériel physique.
- Le logiciel est écrit pour être changé, amélioré.
- Le logiciel est en partie réutilisable.
- Le logiciel peut être testé à tout moment de sa création.

## 2.2 Génie logiciel en Java

### 2.2.1 Paquetages et espace des noms

#### Définition et construction

En Java, il existe déjà de nombreuses structures de données = objets (ou tableaux). Cela fait beaucoup de classes, de fichiers `.class`, et un fort risque de collisions dans l'espace des noms.

D'où l'idée de rassembler les classes en *paquetages* (en anglais, *package*). La syntaxe d'utilisation est la suivante :

```
package ma_lib1;           package ma_lib2;
public class FIFO{...}   public class FIFO{...}
```

Pour utiliser un paquetage pour l'utiliser, on doit *l'importer* :

```
import ma_lib1.FIFO;
class Test{
  public static void main(String[] args){
    int n = Integer.parseInt(args[0]);
    FIFO f = new FIFO(n);
    FIFO.ajouter(1, f);
  }
}
```

L'instruction

```
package id1.id2...idk;
```

*qualifie* les noms des champs publics d'une unité de compilation, comme par exemple

- `ma_lib1.FIFO`, `ma_lib1.FIFO.ajouter`, `ma_lib1.FIFO.supprimer` (premier paquetage)
- `ma_lib2.FIFO`, `ma_lib2.FIFO.ajouter`, `ma_lib2.FIFO.supprimer` (deuxième paquetage)

On référence ces champs publics avec leurs *noms qualifiés* (classe `Test`) ou avec une *forme courte* si on importe la classe avant son utilisation

```
import id1.id2...idk.C;
```

où  $C$  est un nom de classe ou le symbole  $*$  pour importer toutes les classes d'un paquetage.

L'emplacement des paquetages dépend de deux paramètres :

- le *nom*  $id_1.id_2\dots id_k$  qui désigne l'emplacement  $id_1/id_2/\dots/id_k$  dans l'arborescence des répertoires du système de fichiers.
- la *variable d'environnement* CLASSPATH qui donne une suite de racines à l'arborescence des paquetages.

La valeur de CLASSPATH est fixée par une commande Unix :

```
setenv CLASSPATH ".:$HOME/inf431:/users/profs/info/chassignet/Jaxx"
```

(pour csh, tcsh) ou

```
export CLASSPATH=".:$HOME/inf431:/users/profs/info/chassignet/Jaxx"
```

(pour sh, bash).

Un fichier `.java` démarre souvent comme suit :

```
package ma_lib1;
import java.util.*;
import java.io.*;
import java.awt.*;
```

Cette unité de compilation fera partie du paquetage `ma_lib1` et importe les paquetages :

- `java.util` qui contient des classes *standards* pour les tables, les piles, les tableaux de taille variable, etc.
- `java.io` qui contient les classes d'*entrées-sorties*.
- `java.awt` qui contient les classes *graphiques*.
- `java.math` qui contient les classes *mathématiques*.

On renvoie à la liste des paquetages dans la page web du cours. Si aucun paquetage n'est précisé, l'unité de compilation fait partie du paquetage *anonyme* localisé dans le *répertoire courant*.

### Contrôle d'accès

Pour les membres d'une classe, il y a trois types d'accès :

- **public** pour permettre l'accès depuis toutes les classes ;
- **private** pour restreindre l'accès aux seules expressions ou fonctions de la classe courante ;
- par défaut pour autoriser l'accès depuis toutes les classes du même paquetage (cf. section 2.2.1).

Pour les membres d'un paquetage, il y a deux types d'accès :

- une *classe publique* peut être accédée de l'extérieur de son paquetage ;
- une classe sans qualificatif n'est accessible que depuis son paquetage ;
- de l'extérieur du paquetage, double-protection : classe et champ doivent être publics ;
- **Remarque** : le *class loader* ne vérifie pas cette discipline, car il accède à la méthode *main* sans que sa classe ne soit publique !
- *une seule* classe publique *C* par unité de compilation  
 ⇒ une seule classe publique par fichier *C.java*  
 (Certains compilateurs sont laxistes. Mais mieux vaut privilégier la portabilité)

### Utilisation des .jar (java archive)

Un fichier *.jar* est une forme compressée des répertoires. Un *.jar* contient une arborescence de paquetages en un seul fichier (cf. la commande Unix *jar*). Les fichiers *.zip* sont aussi utilisables.

#### 2.2.2 Classe abstraite

On veut permettre l'implantation de deux interfaces de modules de deux façons différentes. On sépare (comme dans le cas de la FIFO), la partie spécification, de la partie implantation. On pourra ainsi écrire (en utilisant une syntaxe objet) :

```
public abstract class FIFO{
    public abstract void ajouter(int x);
    public abstract int supprimer();
}
```

dont on aura deux implantations :

```
public class FIFOTAB extends FIFO{
    private int debut, fin;
    private boolean pleine, vide;
    private int[] contenu;

    public FIFO(int n){
        debut = 0; fin = 0;
        pleine = n == 0; vide = true;
        contenu = new int[n];
    }
    public void ajouter(int x){
        if(pleine)
            throw new Error("File Pleine.");
        contenu[fin] = x;
        fin = (fin + 1) % contenu.length;
        vide = false; pleine = fin == debut;
    }
}
```

```

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

```

et

```

public class FIFOLIB extends FIFO{
    private Liste debut, fin;
    public FIFO(int n){ debut = null; fin = null; }
    public void ajouter(int x){
        if(fin == null)
            debut = fin = new Liste(x);
        else{
            fin.suivant = new Liste(x);
            fin = fin.suivant;
        }
    }
    public int supprimer(){
        if(debut == null) throw new Error("File Vide.");
        else{
            int res = debut.val;
            if(debut == fin)
                debut = fin = null;
            else debut = debut.suivant;
            return res;
        }
    }
}

```

qu'on pourra utiliser sans problème dans :

```

public class TestFIFO{
    public static void main(String[] args){
        FIFOTAB f = new FIFOTAB(10);
        f.ajouter(1);
        FIFOLISTE g = new FIFOLISTE(10);
        g.ajouter(1);
        int x = g.supprimer();
    }
}

```

---

Énumérons quelques propriétés des classes abstraites :

- Une classe abstraite contient des champs indéfinis.
- On ne peut pas créer d'instances de classes abstraites.
- Permet d'utiliser des types disjonctifs (comparer avec les types somme de Caml, les variantes de Pascal ; les `union` de C).



## Chapitre 3

# Complexité et structures de données

Le but de ce chapitre est de rappeler quelques notions simples de complexité. Pour illustrer les concepts, nous rappellerons les structures de données élémentaires vues dans les cours précédents, en insistant sur leur complexité en terme d'accès, insertion ou suppression. Nous indiquerons également sous quelle forme elles sont disponibles en Java dans l'annexe 14.

### 3.1 Complexité élémentaire

#### 3.1.1 Introduction

La recherche en algorithmique porte sur l'élaboration d'algorithmes résolvant un problème donné. La comparaison des différentes méthodes proposées repose généralement sur le temps de calcul des différentes approches. On peut se pencher sur l'analyse exacte de l'algorithme, son coût asymptotique, son coût moyen ou dans le cas le pire, ou bien s'attacher à sa facilité d'implantation (c'est là plus subjectif). Le coût peut être un coût en temps aussi bien qu'en espace. Si  $T(n)$  est le coût d'un algorithme opérant sur des entrées de taille  $n$ , on appelle coût amorti la quantité  $T(n)/n$ . Nous verrons que beaucoup d'algorithmes sur les graphes ont un coût amorti  $O(1)$ .

Être capable d'estimer le coût théorique d'un algorithme est important. Quand on l'implante, cela permet de vérifier qu'il évolue bien en fonction de ce que prédit la théorie.

Trouver des paramètres pertinents pour analyser un algorithme n'est pas toujours simple et demande souvent un peu de métier. Dans les exemples que nous rencontrerons, nous justifierons le choix du ou des paramètres.

Un sujet encore plus poussé est celui du temps minimum pour résoudre un problème donné ; la recherche de bornes inférieures est très difficile et possible seulement dans un petit nombre de cas. On peut aussi essayer de classer les problèmes en fonction de leur difficulté, mais ce sujet sera abordé en année 3.

### 3.1.2 Calculs élémentaires de complexité

Ce que l'utilisateur veut généralement est le temps que va prendre son ordinateur pour résoudre un problème donné. Évidemment, entrent en ligne de compte les performances de la machine. On les abstrait généralement en les cachant dans les constantes d'implantation. On reste à un niveau théorique en remplaçant le temps par le nombre d'opérations élémentaires effectuées (affectations, comparaisons, +, \*, etc.). Il suffirait de mesurer le temps pris par celles-ci sur une machine donnée pour pouvoir extrapoler le temps pris sur l'ordinateur cible.

Énumérons quelques règles courantes :

1. La complexité d'une affectation, opération de lecture ou d'écriture est généralement de  $O(1)$ . C'est surtout vrai pour les types primitifs.
2. Le coût est considéré être additif : le coût de deux instructions indépendantes est  $T(P; Q) = T(P) + T(Q)$ . Le coût d'une itération est plus généralement :  $T(\text{for}(i = 0; i < n; i++) P(i);) = \sum_{i=0}^{n-1} T(P(i))$ .
3. Le coût d'une instruction conditionnelle est le maximum des coûts des deux branches (si le calcul de la condition est négligeable). On peut faire mieux si on connaît la probabilité de passer dans l'une ou l'autre branche pour évaluer le coût moyen.

Souvent, on compare le comportement asymptotique de deux algorithmes. Rappelons que si  $f, g$  sont deux fonctions à valeur  $\geq 0$ , on écrit  $f = O(g)$  si et seulement si le rapport  $f/g$  est borné à l'infini :

$$\exists n_0, \exists K, \forall n \geq n_0, 0 \leq f(n) \leq Kg(n).$$

De telles comparaisons asymptotiques conduisent généralement à isoler les coûts dominants dans l'algorithme.

Donnons un exemple, celui de la multiplication de matrices. Soient  $A$  et  $B$  deux matrices  $n \times n$  à coefficients entiers et  $C = AB$ . Rappelons que :

$$\forall i, j, c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}.$$

Le code Java correspondant est :

```

static int[][] mult(int[][] A, int[][] B){
    int n = A.length;
    int[][] C = new int[n][n];

    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            for(int k = 0; k < n; k++)
                C[i][j] += A[i][k] * B[k][j];
    return C;
}

```

Il est clair que le programme fait trois boucles **for** imbriquées, donc le coût est  $O(n^3)$  multiplications, et  $O(n^3)$  additions. Il est courant de supposer qu'une multiplication est plus coûteuse qu'une addition, donc on se contentera de considérer que la multiplication de matrices coûte  $O(n^3)$ .

Pour la petite histoire, Strassen a montré qu'il existe un algorithme qui demande  $O(n^{\log_2 7})$  multiplications. Il est classique de conjecturer que l'algorithme optimal a une complexité  $O(n^\omega)$  avec  $\omega \geq 2$ . Le record actuel est toujours 2.38 (D. Coppersmith et S. Winograd). Les optimistes pensent que  $\omega = 2$  est la bonne valeur (il y a des progrès récents liés à la théorie des groupes, qui iraient dans ce sens<sup>1</sup>). C'est un domaine de recherche actif de chercher à faire diminuer  $\omega$ . Notons que beaucoup d'algorithmes, notamment de calcul formel, peuvent se ramener à la multiplication de matrices, ce qui en fait une primitive incontournable.

Citons d'autres exemples en vrac. Les meilleurs algorithmes de tri de  $n$  objets par comparateur ont une complexité en  $O(n \log n)$  : c'est le cas de heap sort, merge sort, et quicksort (en moyenne). Trouver un élément dans une liste de taille  $n$  coûte  $O(n)$  opérations. Rechercher un élément dans un tableau trié coûte  $O(\log n)$  (par dichotomie). Énumérer tous les éléments de  $\{0, 1\}^n$  prend  $O(2^n)$ , ce qui fournit un exemple simple de problème difficile, dont le temps de calcul est doublement exponentiel en la taille de l'input et linéaire en la taille de la sortie.

## 3.2 Structures de données

Lors des cours précédents, des structures aussi simples que tableaux, listes et arbres ont été présentées, et des résultats prouvés sur leurs performances en temps d'accès. Nous allons prendre un point de vue dual. Soit  $E$  un ensemble de  $n$  objets (que nous supposons tous distincts). Quelles sont les représentations possibles de  $E$  et leurs différentes propriétés.

Dans la suite du cours, nous utiliserons ces structures de base pour représenter les graphes, mais également pour enrichir les graphes avec des propriétés particulières. Le choix d'une bonne structure d'accompagnement sera crucial dans la performance de certains algorithmes (Dijkstra par exemple).

### 3.2.1 Représentation des ensembles

On suppose que l'on veut stocker ou supprimer un élément, et tester si un élément appartient à  $E$ . Si on représente  $E$  par un tableau de taille  $n$ , ajouter un élément est facile, si on garde l'indice de la dernière case occupée. Retrouver (ou supprimer) un élément coûtera  $O(n)$ . Si un ordre sur les éléments est connu, on peut maintenir le tableau trié, ce qui nous donnera un coût de recherche ou suppression en  $O(\log n)$ . Il n'en reste pas moins que la gestion des places vides peut être compliquée.

Si on utilise une liste, les coûts d'insertion et suppression seront également en  $O(n)$ , mais la place sera toujours calculée au plus juste. Un ordre sur les éléments pourra

---

<sup>1</sup><http://www.cs.caltech.edu/~umans/papers/CKSU05.pdf>

aider un peu, au prix de la complication du code.

Utiliser un arbre est possible si un ordre est connu. Cela garantira un coût d'insertion/suppression/recherche en  $O(\log n)$ , en équilibrant les arbres (AVL) ou par d'autres technique (arbres bicolores, etc.). La taille du stockage augmente un peu (il faut stocker les pointeurs – liens des nœuds dans l'arbre). Les programmes résultants peuvent être compliqués.

La technique la plus efficace est celle du hachage. Elle permet d'avoir un coût d'insertion/suppression/recherche en  $O(1)$ , ce qui est optimal comme on s'en convaincra. L'idée consiste à associer de manière unique un entier à chaque objet, ce qui permettra de le repérer facilement dans un tableau. La difficulté est de trouver une bonne fonction de hachage. Une technique standard consiste à associer à un objet une représentation sous forme d'une chaîne de bits, puis à appliquer une fonction de transformation pour en tirer un petit entier. Il faut souvent ajouter un peu d'espace pour avoir du hachage efficace (disons une petite constante fois  $n$ ).

### 3.2.2 Piles, files, tas

Les piles et les files permettent des opérations rapides sur des cas particuliers d'ensemble. Une pile permet de stocker le dernier élément arrivé avec un coût  $O(1)$  et de le retirer avec le même coût. La file permet de stocker le dernier arrivé en  $O(1)$  et de retirer le premier arrivé en  $O(1)$  également.

Une file de priorité permet de stocker les éléments de  $E$  de sorte que le minimum soit trouvable en  $O(1)$  et les coûts de mise à jour (insertion ou suppression) en  $O(\log n)$ . Les implantations les plus courantes utilisent un *tas*.

Deuxième partie

Graphes



## Chapitre 4

# Propriétés élémentaires des graphes

### 4.1 Introduction

Une partie importante de tout système d'informations consiste à modéliser les données ou les propriétés du système. Si les données sont indépendantes, on peut les représenter par un ensemble, qui sera implanté par exemple dans un tableau ou une liste. Si un lien hiérarchique simple est mis en évidence, on peut stocker les informations sous forme d'arbre (hiérarchie par exemple). Que faire quand les liens sont plus complexes, comme par exemple quand on doit représenter la carte d'un réseau routier, un circuit électronique ? On utilise alors souvent des graphes, qui permettent de coder de telles informations. Par exemple, on peut imaginer gérer une carte du réseau ferré en stockant les gares de France et les différentes distances entre ces gares.

Une fois que l'on dispose d'une telle représentation, on travaille généralement sur son abstraction en tant que graphe, ce qui permet d'utiliser de nombreux algorithmes résolvant des problèmes précis, comme par exemple trouver un chemin minimal en temps (ou en ressources) entre deux gares.

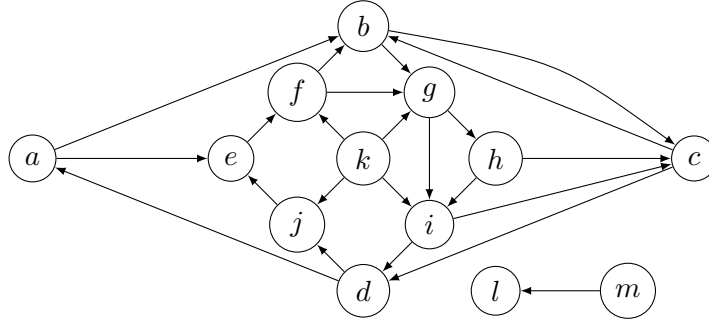
Citons encore comme exemples les graphes de dépendance entre fichiers sources de façon à optimiser la compilation, le diagramme d'héritage des classes dans un langage typé, mais aussi la représentation de tâches à accomplir dans une usine avec contraintes de précédence, etc.

### 4.2 Définitions

Un *graphe*  $\mathcal{G} = (\mathcal{S}, \mathcal{A})$  est donné par un ensemble  $\mathcal{S}$  de *sommets* et un ensemble  $\mathcal{A}$  d'*arcs*,  $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$ .

On visualise en général un graphe par un dessin dans le plan. Les sommets sont représentés par des points, les arcs par des flèches. Le graphe  $\mathcal{G}_1$  de la figure 4.1 a pour ensemble de sommets  $\mathcal{S} = \{a, b, c, d, e, f, g, h, i, j, k, l, m\}$ , et ensemble d'arêtes

$\mathcal{A} = \{(a, b), (b, c), (c, b), (c, d), (d, a), (a, e), (e, f), (f, b), (f, g), (b, g), (g, h), (h, c), (h, i), (g, i), (i, c), (i, d), (d, j), (j, e), (k, f), (k, g), (k, i), (k, j), (m, l)\}$ .

FIG. 4.1 – Le graphe  $\mathcal{G}_1$ .

Il est important de noter qu'il n'existe pas de dessin canonique d'un graphe. Par exemple, les deux dessins de la figure 4.2 correspondent au même graphe.

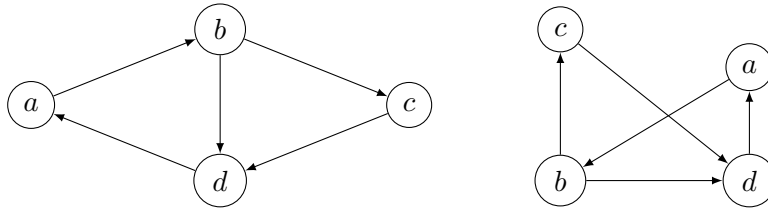
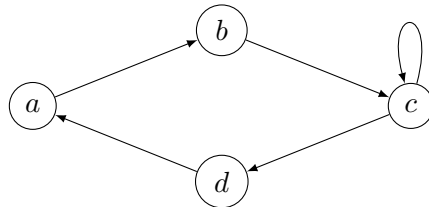


FIG. 4.2 – Deux dessins d'un même graphe.

Un arc  $\alpha = (a, b)$  est *orienté* de  $a$  vers  $b$ ;  $\alpha$  a pour *origine*  $a$  et *destination*  $b$ ; l'arc  $\alpha$  est *incident* à  $a$  ainsi qu'à  $b$ . On dira également que  $a$  est un *prédécesseur* de  $b$ , et  $b$  un *successeur* de  $a$ ;  $a$  et  $b$  seront également dits *adjacents*. L'arc  $(x, x)$  est une *boucle*.

Un graphe est *simple* s'il ne comporte pas de boucles. Le graphe de la figure 4.1 est simple. Le graphe  $\mathcal{G}_2$  de la figure 4.3 ne l'est pas.

FIG. 4.3 – Le graphe  $\mathcal{G}_2$ .

On peut enrichir un graphe en considérant des quantités associées à chaque arc, ou à chaque sommet. Penser par exemple à la carte de circulation d'une ville, avec ses sens

interdits et les longueurs des rues. On parlera d'*arcs valués*, c'est-à-dire qu'on utilisera une fonction  $\nu : \mathcal{A} \rightarrow \mathbb{N}$  qui donnera le poids ou la valeur associée à un arc. C'est le cas du graphe  $\mathcal{G}_3$  de la figure 4.4 dans lequel on a ajouté des poids sur chaque arc.

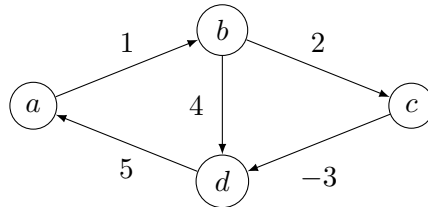


FIG. 4.4 – Le graphe valué  $\mathcal{G}_3$ .

#### 4.2.1 Graphes non orientés

Les graphes les plus généraux que nous considérerons sont *a priori orientés*, ce qui veut dire qu'un arc indique un sens de lecture ou de dessin. Autrement dit, si l'arc  $(a, b)$  existe, cela n'implique pas l'existence de l'arc  $(b, a)$  dans le graphe. Il peut arriver que pour tout arc  $(a, b)$ , son inverse  $(b, a)$  existe également. On dit que le graphe  $\mathcal{G}$  est *non orienté*. Dans ce cas, on utilise le terme *arête* au lieu d'arc et on note parfois  $\{a, b\}$  l'arête reliant  $a$  à  $b$ <sup>1</sup>. On dessine alors des traits au lieu de flèches, comme dans l'exemple du graphe  $\mathcal{G}_4$  de la figure 4.5.

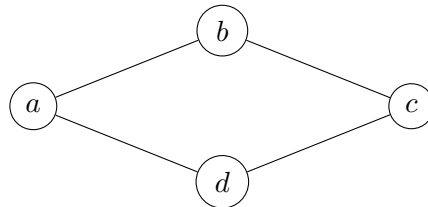


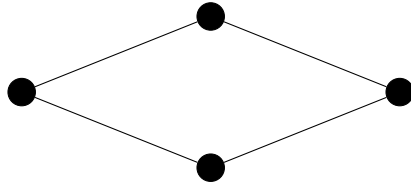
FIG. 4.5 – Le graphe  $\mathcal{G}_4$ .

#### 4.2.2 Sous-graphes, etc.

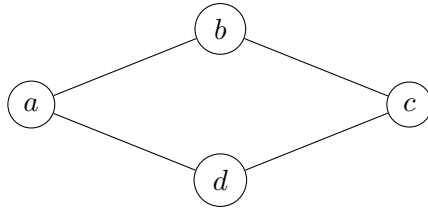
Il peut aussi arriver que seule la forme du graphe importe. On remplace alors les sommets par des cercles. Le graphe de la figure 4.5 devient alors simplement celui de la figure 4.6. De nombreux graphes deviennent isomorphes si on anonymise leurs nœuds.

De façon similaire, on peut avoir à considérer le graphe  $\mathcal{G}$  "sans ses flèches", c'est-à-dire considérer sa forme non orientée. On enlève alors les boucles et les flèches. La

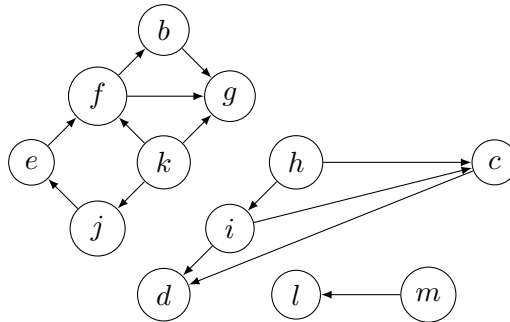
<sup>1</sup>Nous considérerons des paires au sens propre, c'est-à-dire qu'un graphe non orienté ne possède pas de boucles.

FIG. 4.6 – Le graphe abstrait correspondant à  $\mathcal{G}_4$ .

version non orientée (ou graphe non orienté *sous-jacent*) du graphe  $\mathcal{G}_2$  est donnée à la figure 4.7.

FIG. 4.7 – Le graphe non-orienté sous-jacent au graphe  $\mathcal{G}_2$ .

Si  $\mathcal{G} = (\mathcal{S}, \mathcal{A})$  est un graphe et  $\mathcal{S}'$  un sous-ensemble de  $\mathcal{S}$ , on note  $\mathcal{A}|\mathcal{S}'$  l'ensemble des arêtes de  $\mathcal{A}$  qui ont leurs deux extrémités dans  $\mathcal{S}'$ . Le graphe  $\mathcal{G}' = \mathcal{G}|\mathcal{S}' = (\mathcal{S}', \mathcal{A}|\mathcal{S}')$  est appelé le *graphe induit* de  $\mathcal{G}$  par  $\mathcal{S}'$ . Tout graphe  $(\mathcal{S}', \mathcal{A}')$  avec  $\mathcal{S}' \subset \mathcal{S}$  et  $\mathcal{A}' \subset \mathcal{A}$  est appelé *sous-graphe* de  $\mathcal{G}$ ; si  $\mathcal{S}' = \mathcal{S}$ , on parle de *graphe couvrant*. Pour le graphe de la figure 4.1, on trouve dans les figures 4.8, 4.9, 4.10 des exemples de chaque catégorie.

FIG. 4.8 – Un sous-graphe du graphe  $\mathcal{G}_1$ .

### 4.2.3 Degré

**Définition 4.2.1** Le degré d'un sommet  $s$  dans un graphe non orienté est le nombre d'arêtes incidentes à  $s$ .

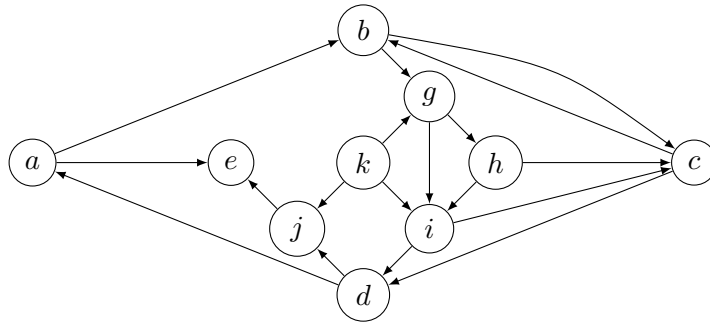
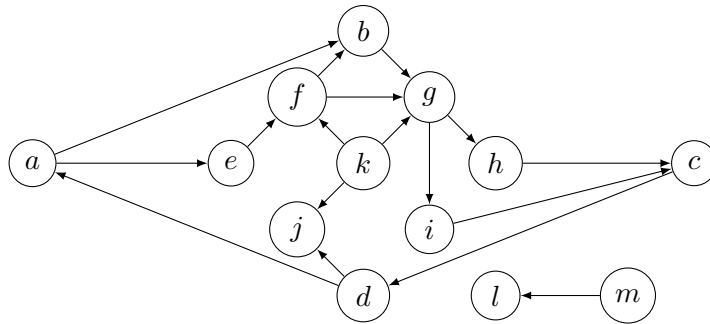
FIG. 4.9 – Un graphe induit de  $\mathcal{G}_1$ .

FIG. 4.10 – Un graphe couvrant.

**Proposition 4.2.1 (hand-shaking lemma)** Dans tout graphe  $\mathcal{G}$  non orienté, on a  $\sum_s \deg(s) = 2|\mathcal{A}|$ .

*Démonstration.* Chaque arête contribue pour deux sommets dans la somme.  $\square$

**Corollaire 4.2.1** Soit  $\mathcal{G}$  un graphe non orienté. Le nombre de sommets de degré impair est pair.

*Démonstration.* On peut récrire la somme précédente comme :

$$\sum_s \deg(s) = \sum_{s, \deg(s) \text{ impair}} \deg(s) + \sum_{s, \deg(s) \text{ pair}} \deg(s).$$

Les deux sommes extrêmes sont paires, donc celle du milieu également.  $\square$

Dans le cas des graphes orientés, on peut définir de la même façon le *degré entrant* (le nombre d'arcs arrivant sur un sommet), et le *degré sortant* (le nombre d'arcs partant d'un sommet).

#### 4.2.4 Quelques graphes non orientés classiques

Remarquons qu'une liste ou un arbre forment des exemples de graphes.

Le graphe *complet* à  $n$  sommets est le graphe dans lequel chaque sommet a  $n - 1$  voisins, voir la figure 4.11. Le graphe *biparti complet*  $K_{i,j}$  contient  $i + j$  sommets répartis en deux sous-ensembles, l'un contenant  $i$  sommets, l'autre  $j$  sommets. On relie ensuite chacun des  $i$  sommets à tous les  $j$  autres. On a dessiné dans la figure 4.11 le graphe  $K_{3,3}$ .

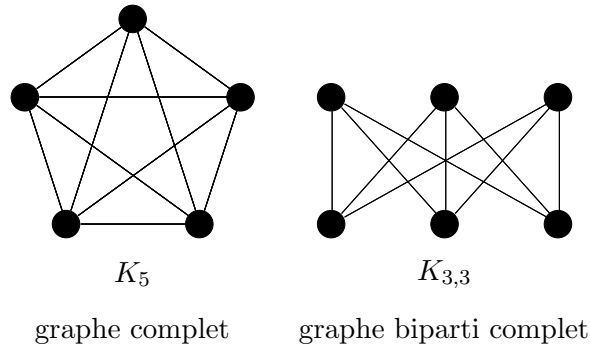


FIG. 4.11 – Graphes classiques.

### 4.3 Représentations en machine

Les dessins que nous avons faits dans la section précédente sont commodes pour faire des exemples et des raisonnements à la main, mais ils ne sont pas utilisables en machine. Dessiner dans le plan un graphe donné est en soi une tâche algorithmique et pratique relativement ardue. Donc nous devons trouver une autre représentation plus commode.

Le problème se réduit à représenter  $\mathcal{S}$  et  $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$  en machine. Sans perte de généralité, on peut se ramener au cas où  $\mathcal{S} = \{0, 1, \dots, n - 1\}$  si  $n = |\mathcal{S}|$ . Dans la pratique, il sera toujours possible de coder les éléments du graphe par un tel ensemble d'entiers, quitte à utiliser du hachage par exemple (voir les polys des cours précédents ou [6]). Passons au problème de la représentation de  $\mathcal{A}$ . Avec notre définition de graphe, il est clair que le cardinal de  $\mathcal{A}$  est borné par  $n^2$ . Si  $m = |\mathcal{A}|$  est proche de  $n^2$ , on parlera de graphe *dense*. Si  $m = o(n^2)$ , on parlera de graphe *creux*. Il est très fréquent en pratique d'avoir à traiter des graphes creux (éléments finis, carte routière, graphe d'Internet, etc.). Les algorithmes que nous étudierons auront une complexité qui pourra bien souvent être décrite en terme de  $n$  et  $m$ , ce qui est toujours intéressant quand  $m$

est loin de sa valeur maximale théorique  $O(n^2)$ .

### 4.3.1 Représentation par une matrice

C'est la façon la plus simple de représenter l'ensemble  $\mathcal{A}$ . On se donne donc une matrice  $\mathcal{M}$  de taille  $n \times n$ . Une telle matrice est appelée *matrice d'adjacence* du graphe. Il existe essentiellement deux choix pour le type de la matrice.

Le premier choix consiste à prendre une matrice de booléens tels que  $\mathcal{M}_{i,j}$  est vrai si et seulement si  $(i, j) \in \mathcal{A}$ . Pour le graphe  $\mathcal{G}_2$  de la figure 4.3, on numérote les sommets  $a, b, c, d$  selon la correspondance

$$\begin{array}{c|c|c|c} a & b & c & d \\ \hline 0 & 1 & 2 & 3 \end{array}$$

puis on construit la matrice booléenne (où nous avons mis  $T$  pour vrai et omis les valeurs fausses) :

$i, j$	0	1	2	3
0		$T$		
1			$T$	
2			$T$	$T$
3	$T$			

Le deuxième choix conduit à utiliser un tableau d'entiers (ou de flottants) quand les arcs de  $\mathcal{G}$  sont valués. On pose alors  $\mathcal{M}_{i,j} = \nu((i, j))$  et on utilise une valeur par défaut quand l'arc  $(i, j)$  n'existe pas (par exemple une valeur plus grande que toutes les valeurs de  $\nu$ ). Pour le graphe  $\mathcal{G}_3$  de la figure 4.4, on trouve :

$i, j$	0	1	2	3
0		1		
1			2	4
2				-3
3	5			

Dans le cas où  $\mathcal{G}$  est non orienté, la matrice  $\mathcal{M}$  est une matrice symétrique.

### 4.3.2 Représentation par un tableau de listes

Le stockage à l'aide de matrices prend une place proportionnelle à  $n^2$ . Quand  $m = |\mathcal{A}|$  est petit par rapport à  $n^2$ , cela représente une perte de place parfois importante (ainsi qu'une perte de temps). D'où l'idée d'utiliser une représentation *creuse*, par exemple sous la forme d'un tableau de listes  $L$ . La liste  $L_i$  pour  $i \in \mathcal{S}$ , contiendra la liste des successeurs de  $i$  dans le graphe, c'est-à-dire

$$L_i = (j \in \mathcal{S}, (i, j) \in \mathcal{A}).$$

En utilisant une représentation "graphique" des listes, l'exemple du graphe  $\mathcal{G}_2$  pourra conduire à :

$L[0] = (1)$

$L[1] = (2)$

$L[2] = (2, 3)$

$L[3] = (0)$

Il est important de noter, encore une fois, qu'une telle représentation n'est nullement canonique, car on peut décider de ranger les voisins dans n'importe quel ordre. De manière plus générale, on peut tout à fait représenter notre graphe sous forme creuse comme un tableau d'ensembles, les ensembles pouvant avoir une représentation et des propriétés variées.

La place mémoire nécessitée par une telle représentation est de  $|S|$  ensembles. Dans le cas de listes chaînées, chaque ensemble demande une place  $O(|L[i]|)$  (par exemple le contenu d'une cellule et d'un pointeur sur la suivante). En tout, on aura un stockage en  $O(n + m)$ , donc linéaire. En fonction des problèmes, on pourra trouver des représentations parfois plus adaptées.

# Chapitre 5

## Accessibilité

### 5.1 Problématique

Soit  $\mathcal{G} = (\mathcal{S}, \mathcal{A})$  un graphe orienté. L'un des problèmes élémentaires que l'on peut résoudre est celui de l'accessibilité d'un sommet à partir d'un autre. Étant donné une carte ferroviaire, peut-on aller de Paris à Strasbourg? Combien de sommets intermédiaires devons-nous visiter au minimum pour se rendre de l'un à l'autre? La difficulté de ce genre d'exercice réside en ce que l'on dispose de l'information sous forme indirecte, c'est-à-dire qu'à partir de renseignements locaux (qui est voisin de qui), on doit déterminer un résultat global : existe-t-il un chemin qui m'amène d'un sommet à un autre?

### 5.2 Définitions ; propriétés fondamentales

Un *chemin*  $(s_0, s_1, \dots, s_p)$  dans  $\mathcal{G}$  est une suite finie non vide de sommets telle que  $(s_k, s_{k+1}) \in \mathcal{A}$ . La *longueur* du chemin est  $p$ , c'est le nombre d'arcs empruntés par le chemin. Notons que les arcs ne sont pas nécessairement distincts. Par exemple, la suite  $(c, a, b, c, e)$  est un chemin dans le graphe  $\mathcal{G}_1$  de la figure 5.1.

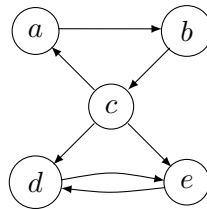


FIG. 5.1 – Le graphe  $\mathcal{G}_1$ .

Un sommet  $y$  est un *descendant* (resp. *ascendant*) de  $x$  s'il existe un chemin de  $x$  vers  $y$  (resp. de  $y$  vers  $x$ ). C'est un descendant (resp. ascendant) *propre* s'il existe un

chemin de longueur non nulle. Dans le graphe  $\mathcal{G}_1$ ,  $b$  est descendant propre de  $a$ , mais également de lui-même (par le chemin  $(b, c, a, b)$ ).

Un chemin est *simple* si les arcs  $(s_{i-1}, s_i)$  pour  $i = 1, \dots, p$  sont deux à deux distincts; il est *élémentaire* si les sommets sont deux à deux distincts. Le chemin est un *circuit* si  $p \geq 1$  et  $s_p = s_0$ ; il est *élémentaire* si  $(s_0, \dots, s_{p-1})$  est élémentaire.

Dans le graphe  $\mathcal{G}_1$ , le chemin  $(a, b, c)$  est élémentaire; le chemin  $(a, b, c, a)$  est un circuit élémentaire, mais pas un chemin élémentaire.

Dans le cas où  $\mathcal{G}$  n'est pas orienté, on définit une *chaîne* comme étant une suite finie de sommets  $(s_0, s_1, \dots, s_p)$  telle que deux sommets consécutifs soient extrémités d'une arête de  $\mathcal{A}$ , les arêtes étant distinctes deux à deux. Une chaîne est *élémentaire* si les sommets sont deux à deux distincts; c'est un *cycle* si  $s_0 = s_p$ . Un cycle est *élémentaire* si  $p \geq 1$  et  $(s_0, \dots, s_{p-1})$  est une chaîne élémentaire.

Le résultat suivant est facile, mais mérite d'être cité.

**Lemme 5.2.1** *L'ensemble des chemins d'un graphe est infini si et seulement si le graphe possède des circuits.*

Le lemme de König permet de se restreindre dans tous les cas aux chemins élémentaires. On dit que  $c'$  est un *chemin extrait* de  $c$  si  $c'$  est chemin, que  $c$  et  $c'$  ont même origine et extrémité et la suite des arcs de  $c'$  est une sous-suite de la suite des arcs de  $c$ . Pour le graphe  $\mathcal{G}_1$ , le chemin  $(a, b, c)$  est un chemin extrait de  $(a, b, c, a, b, c)$ .

**Lemme 5.2.2** (König) *De tout chemin, on peut extraire un chemin élémentaire.*

*Démonstration.* On effectue une récurrence sur la longueur  $p$  d'un chemin. Si  $p = 0$ , le chemin est élémentaire. Soit  $c = (s_0, s_1, \dots, s_p)$  un chemin de longueur  $p > 0$ . S'il n'est pas élémentaire, il existe deux sommets égaux  $s_i = s_j$  avec  $0 \leq i < j \leq p$ . Le chemin  $c' = (s_0, \dots, s_i, s_{j+1}, \dots, s_p)$  est strictement plus petit que  $c$  et on peut en extraire un chemin élémentaire par hypothèse de récurrence, qui sera lui-même extrait de  $c$ .  $\square$

### 5.3 Fermeture transitive

Cette section répond au problème déjà indiqué : existe-il un chemin entre deux sommets donnés de  $\mathcal{G}$ ? On obtient la réponse simultanément pour *toutes* les paires simultanément. Si la réponse est demandé pour un petit nombre de couples, un parcours sera sans doute plus efficace (voir chapitre 6). Par convention, un sommet sera toujours supposé accessible de lui-même.

**Définition 5.3.1** *On appelle fermeture transitive du graphe  $\mathcal{G}$  le graphe  $\mathcal{G}^* = (\mathcal{S}, \mathcal{A}^*)$  avec  $(s, t) \in \mathcal{A}^*$  si et seulement s'il existe un chemin de  $s$  à  $t$  dans  $\mathcal{G}$ .*

Pour ce genre de problème où nous devons calculer un résultat global sur le graphe, il est logique de supposer que l'on va obtenir un graphe  $\mathcal{G}^*$  dense, donc plutôt favoriser des algorithmes basés sur la matrice d'adjacence (booléenne ou entière) associée au graphe  $\mathcal{G}$ . Pour simplifier l'exposition, nous allons supposer (dans cette section) que  $\mathcal{S} = \{0, 1, 2, \dots, n-1\}$ .

### 5.3.1 Première solution par produit de matrice

On va construire de proche en proche une suite de graphes  $\mathcal{G}^{(r)}$ ,  $1 \leq r \leq n$  de la façon suivante :  $\mathcal{G}^{(1)}$  est le graphe initial ; pour  $r \geq 2$ ,  $\mathcal{G}^{(r)} = (\mathcal{S}, \mathcal{A}^{(r)})$  sera le graphe défini par  $(i, j) \in \mathcal{A}^{(r)}$  si et seulement s'il existe un chemin de  $i$  à  $j$  de longueur exactement  $r$ . On notera  $\mathcal{M}^{(r)}$  la matrice d'adjacence associée à  $\mathcal{G}^{(r)}$ .

S'il existe un chemin de longueur  $r > 1$  entre  $i$  et  $j$ , il s'écrit  $(i, k, \dots, j)$  avec  $k$  voisin de  $i$  et  $(k, \dots, j)$  un chemin de longueur  $r - 1$ . La formule booléenne s'écrit alors :

$$\mathcal{M}_{i,j}^{(r)} = \bigvee_{k=0}^{n-1} \mathcal{M}_{i,k}^{(1)} \wedge \mathcal{M}_{k,j}^{(r-1)}.$$

On reconnaît là la formule du produit des matrices  $\mathcal{M}^{(1)}$  et  $\mathcal{M}^{(r-1)}$  où on a remplacé l'addition par le OU logique, et la multiplication par le ET logique. Plus généralement, si  $(i, j) \in \mathcal{G}^*$ , c'est qu'il existe un chemin de longueur  $< n$  entre les deux (par le lemme de König).

Par la suite, on utilisera la syntaxe de l'addition et de la multiplication usuelles au lieu de OU et ET. Cela permet d'écrire :

$$\mathcal{M}^* = I_n + \mathcal{M}^{(1)} + \mathcal{M}^{(2)} + \dots + \mathcal{M}^{(n-1)}. \quad (5.1)$$

La matrice identité d'ordre  $n$ , notée  $I_n$ , a été ajoutée pour tenir compte de l'accessibilité de tout sommet avec lui-même.

Si le produit de deux matrices se fait en temps  $O(n^\omega)$  (où  $\omega$  est un exposant compris entre 2 et 3 – voir section 3.1.2), on voit que le calcul naïf nécessite  $O(n^{1+\omega})$  opérations, soit généralement  $O(n^4)$  en pratique.

Remarquons qu'on peut récrire la formule (5.1) sous la forme (en booléens) :

$$\mathcal{M}^* = (I + \mathcal{M})^{n-1}$$

ce qui fait qu'on peut calculer la puissance  $n$ -ième de la matrice en  $O((\log n)n^\omega)$  par une méthode d'exponentiation binaire.

Reprenons l'exemple du graphe  $\mathcal{G}_1$ , dont la matrice d'adjacence est :

$$\mathcal{M} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

On trouve :

$$\mathcal{M}^{(2)} = \mathcal{M}^{(1)} \cdot \mathcal{M}^{(1)} = \begin{pmatrix} & & & & \\ 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{pmatrix}, \quad \mathcal{M}^{(3)} = \begin{pmatrix} & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \\ & & & & & 1 \end{pmatrix},$$

$$\mathcal{M}^{(4)} = \begin{pmatrix} & 1 & & 1 & 1 \\ & & 1 & 1 & 1 \\ 1 & & & 1 & 1 \\ & & & 1 & \\ & & & & 1 \end{pmatrix}.$$

Faisant la somme booléenne, on trouve :

$$\mathcal{M}^* = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ & & & 1 & 1 \\ & & & 1 & 1 \end{pmatrix}.$$

On peut également compter le nombre de chemins entre deux sommets  $x$  et  $y$ , en considérant la matrice d'adjacence entière.

**Proposition 5.3.1** *Le nombre de chemins de longueur  $\ell$  entre  $i$  et  $j$  est  $\mathcal{M}_{i,j}^{(\ell)}$ ,  $\mathcal{M}$  étant considérée comme une matrice entière, les opérations étant faites avec l'addition et la multiplication des entiers.*

### 5.3.2 L'algorithme de Roy et Warshall

Dans l'algorithme précédent, nous avons raisonné sur la longueur des chemins. Une autre idée consiste à faire augmenter le nombre de sommets possibles dans un chemin. On va construire là aussi une suite de graphes auxiliaires  $\mathcal{G}^{(r)} = (\mathcal{S}, \mathcal{A}^{(r)})$ , mais de la façon suivante :  $(i, j)$  est dans  $\mathcal{A}^{(-1)}$  s'il existe un chemin direct de  $i$  à  $j$  et donc ne passant par aucun sommet intermédiaire, c'est-à-dire le graphe de matrice  $I_n + \mathcal{M}$  (chaque sommet étant accessible depuis lui-même). Un arc  $(i, j)$  est dans  $\mathcal{A}^{(0)}$  s'il existe un chemin de  $i$  à  $j$  passant au plus par le sommet 0. Plus généralement,  $(i, j) \in \mathcal{A}^{(r)}$  s'il existe un chemin de  $i$  à  $j$  ne passant par aucun sommet intermédiaire d'indice  $> r$ . Un tel chemin existe dans deux cas : ou bien il existait déjà un chemin de  $i$  à  $j$  ne passant pas par un sommet d'indice  $> r - 1$ , ou bien on peut aller de  $i$  à  $r$  dans les mêmes conditions, puis de  $r$  à  $j$ . En booléens :

$$\mathcal{M}_{i,j}^{(r)} = \mathcal{M}_{i,j}^{(r-1)} + \mathcal{M}_{i,r}^{(r-1)} \mathcal{M}_{r,j}^{(r-1)}.$$

Donnons le pseudocode pour l'algorithme :

```
RoyWarshall(M)

// M est la matrice d'adjacence booléenne n x n d'un graphe G

1. N := M; P := identite(n, n);
```

```

2. pour r <- 0 à n-1 faire
    pour i <- 0 à n-1 faire
        pour j <-0 à n-1 faire
            P[i][j] <- N[i][j] OU (N[i][r] ET N[r][j]);
        N := P; // copie de P dans N
3. retourner N; // matrice d'adjacence de G*

```

La complexité de cet algorithme est clairement en  $O(n^3)$ . On peut remarquer que la matrice P est inutile et qu'on peut effectuer les calculs *en place*, c'est-à-dire directement dans N. Le code correspondant est alors :

```

RoyWarshall(M)

// M est la matrice d'adjacence booléenne n x n d'un graphe G

1. N := M; pour i <- 0 à n-1 faire N[i][i] <- true;

2. pour r <- 0 à n-1 faire
    pour i <- 0 à n-1 faire
        pour j <-0 à n-1 faire
            N[i][j] <- N[i][j] OU (N[i][r] ET N[r][j]);
3. retourner N; // matrice d'adjacence de G*

```

La justification de la validité de la variante précédente dépend de la proposition suivante.

**Proposition 5.3.2** *Pout tout  $r, i, j$  dans  $S$ , les deux propriétés suivantes sont vraies :*

- (1)  $(i, r) \in \mathcal{A}^{(r-1)} \iff (i, r) \in \mathcal{A}^{(r)}$  ;
- (2)  $(r, j) \in \mathcal{A}^{(r-1)} \iff (r, j) \in \mathcal{A}^{(r)}$ .

*Démonstration.* Traitons le point (1), le point (2) se traitant de façon similaire.

Dire que  $(i, r) \in \mathcal{A}^{(r-1)}$  veut dire qu'il existe un chemin de  $i$  à  $r$  ne passant par aucun sommet intermédiaire d'indice  $> r - 1$  ; *a fortiori*, il ne passe pas par un sommet d'indice  $> r$ , d'où le sens direct.

Si  $(i, r) \in \mathcal{A}^{(r)}$ , cela veut dire qu'il existe un chemin de  $i$  à  $r$  ne passant par aucun sommet intermédiaire d'indice  $> r$  ; en considérant un chemin élémentaire entre  $i$  et  $r$ , on voit qu'il ne peut passer par aucun sommet d'indice  $> r - 1$ , à part le sommet  $r$  lui-même, donc  $(i, r) \in \mathcal{A}^{(r-1)}$ .  $\square$

Donnons maintenant une implantation dans notre classe Graphe (voir l'annexe 15 pour la définition de la classe abstraite et des implantations) :

```

static Graphe RoyWarshall(Graphe G){
    Graphe F = G.copie();

    for(Sommet s : G.sommets())
        F.ajouterArc(s, s);
    for(Sommet r : G.sommets())
        for(Sommet s : G.sommets())
            for(Sommet t : G.sommets())
                if(!F.existeArc(s, t)
                    && (F.existeArc(s, r)
                        && F.existeArc(r, t)))
                    F.ajouterArc(s, t);

    return F;
}

```

## 5.4 Plus court chemin

Cette fois, on cherche à calculer pour chaque paire de sommets  $(i, j)$  le plus court chemin qui les joint, quand un chemin existe. Nous verrons plus loin comment répondre à la même question quand on cherche la distance entre un sommet donné et plusieurs sommets.

De façon générale, on peut se donner une fonction distance  $d : \mathcal{A} \rightarrow \mathbb{R}_+$  vérifiant :

$$d((i, j)) = \begin{cases} > 0 & \text{si } (i, j) \in \mathcal{A}, \\ 0 & \text{si } i = j, \\ \infty & \text{sinon.} \end{cases}$$

De façon analogue au premier algorithme de recherche d'accessibilité, on peut établir une récurrence sur le nombre de voisins visités. On définit  $D_{i,j}^{(k)}$  comme étant la longueur du plus court chemin de  $i$  à  $j$  ayant au plus  $k$  arcs. On a pour  $k \geq 1$  :

$$D_{i,j}^{(k)} = \min \left( D_{i,j}^{(k-1)}, \min_{0 \leq \ell < n} (D_{i,\ell}^{(k-1)} + D_{\ell,j}^{(k-1)}) \right),$$

avec  $D_{i,j}^{(0)} = d((i, j))$ . On en déduit un algorithme en  $O(n^4)$ , ainsi qu'un algorithme en  $O((\log n)n^\omega)$  en utilisant de la multiplication rapide de matrices (dans une algèbre particulière).

L'algorithme de Floyd fonctionne comme l'algorithme d'accessibilité de Roy-Warshall. On considère  $F_{i,j}^{(r)}$  comme étant la longueur du plus court chemin de  $i$  à  $j$  ne passant par aucun sommet intermédiaire d'indice  $> r$ . On a :

$$F_{i,j}^{(r)} = \min \left( F_{i,j}^{(r-1)}, F_{i,r}^{(r-1)} + F_{r,j}^{(r-1)} \right).$$

Le coût de l'algorithme est en  $O(n^3)$ .

**Exercice 5.4.1** Écrire un programme qui implante cette méthode et qui stocke le plus court chemin, en mémorisant l'indice  $r$  fournissant le chemin minimum de  $i$  à  $j$ .

## 5.5 Arbres

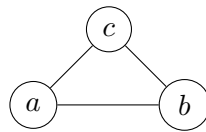
Nous allons revoir les propriétés des arbres, déjà introduits dans les cours précédents (INF321 ou INF421), comme cas particulier de graphes. On se place ici encore dans le cas des graphes simples (sans boucles), non orientés.

**Lemme 5.5.1** Soit  $\mathcal{G}$  un graphe simple non orienté. Si  $\mathcal{G}$  a un cycle, alors ce cycle est de longueur  $\geq 3$ .

*Démonstration.* Si  $\mathcal{G}$  a un cycle, c'est une chaîne, donc toutes les arêtes doivent être distinctes, ce qui interdit :



Le plus petit cycle possible est ainsi :



**Définition 5.5.1** Un graphe non orienté  $\mathcal{G}$  est connexe s'il existe toujours une chaîne entre deux sommets de  $\mathcal{G}$ .

Si un graphe n'est pas connexe, on peut le décomposer comme réunion de *composantes connexes*, c'est-à-dire des classes d'équivalence de la relation  $s \equiv t$  si et seulement si il existe une chaîne de  $s$  à  $t$ .

Le but de cette section est de donner une description géométrique des arbres :

**Proposition 5.5.1** Soit  $\mathcal{G}$  un graphe simple à  $n$  sommets. Les propriétés suivantes sont équivalentes et caractérisent le fait que  $\mathcal{G}$  est un arbre.

- (i)  $\mathcal{G}$  est un graphe connexe sans cycle ;
- (ii)  $\mathcal{G}$  est un graphe connexe ayant  $n - 1$  arêtes ;
- (iii)  $\mathcal{G}$  est un graphe sans cycle possédant  $n - 1$  arêtes ;
- (iv)  $\mathcal{G}$  est un graphe dans lequel deux sommets quelconques sont liés par une seule chaîne ;
- (v)  $\mathcal{G}$  est un graphe connexe dont la connexité disparaît dès qu'on enlève une arête quelconque ;
- (vi)  $\mathcal{G}$  est un graphe sans cycle tel que l'ajout d'une arête quelconque crée un cycle et un seul.

**Lemme 5.5.2** *Un graphe connexe à  $n$  sommets possède au moins  $n - 1$  arêtes.*

*Démonstration.* On utilise une récurrence sur  $n$ . La propriété est vraie pour  $n = 1$ . Supposons  $n \geq 2$  et soit  $\mathcal{G} = (\mathcal{S}, \mathcal{A})$  un graphe connexe à  $n$  sommets. Soit  $u$  un sommet de  $\mathcal{G}$  et considérons le graphe  $\mathcal{G}_u$  induit de  $\mathcal{G}$  par  $\mathcal{S} - \{u\}$ . Soient  $C_1, C_2, \dots, C_p$  les composantes connexes de  $\mathcal{G}_u$ . Pour tout  $k \in \{1, \dots, p\}$ , il existe au moins une arête liant  $u$  à un sommet de  $C_k$  (sinon  $\mathcal{G}$  ne serait pas connexe). Soit  $\mathcal{G}^{(k)}$  le sous-graphe induit par  $C_k$ . Par hypothèse de récurrence, les nombre d'arêtes et de sommets de  $\mathcal{G}^{(k)}$  satisfont  $m_k \geq n_k - 1$ . On en déduit :

$$m \geq p + \left( \sum_{k=1}^p m_k \right) \geq \sum_{k=1}^p n_k = n - 1.$$

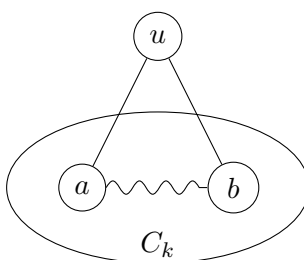
d'où le résultat.  $\square$

**Lemme 5.5.3** *Un graphe à  $n$  sommets ayant au moins  $n$  arêtes possède un cycle.*

*Démonstration.* On utilise là encore une récurrence. La propriété est vraie pour  $n \leq 3$  (on ne suppose pas  $\mathcal{G}$  simple). Soit  $n \geq 4$  et  $\mathcal{G} = (\mathcal{S}, \mathcal{A})$  un graphe à  $n$  sommets et  $m$  arêtes, avec  $m \geq n$ . On raisonne par l'absurde en supposant  $\mathcal{G}$  sans cycle. Soit  $u \in \mathcal{S}$  et  $\mathcal{G}_u$  le graphe induit par  $\mathcal{S} - \{u\}$  comme dans le lemme précédent, et  $C_1, C_2, \dots, C_p$  les composantes connexes de  $\mathcal{G}_u$ , ainsi que  $\mathcal{G}^{(k)}$  le sous-graphe induit par  $C_k$ . Chaque  $\mathcal{G}^{(k)}$  est sans cycle, donc  $m_k \leq n_k - 1$ . Le nombre d'arêtes incidentes à  $u$  est

$$m - \sum_{k=1}^p m_k \geq m + p - \sum_{k=1}^p n_k \geq p + 1.$$

Il existe donc un  $k$  tel que  $u$  soit adjacent à deux sommets distincts de  $a$  et  $b$  de  $C_k$ .



Comme  $\mathcal{G}^{(k)}$  est connexe, il existe une chaîne de  $a$  à  $b$  dans  $\mathcal{G}^{(k)}$  et donc un cycle dans  $\mathcal{G}$  passant par  $u, a$  et  $b$ . Contradiction.  $\square$

*Démonstration de la proposition :* on démontre (i)  $\Rightarrow$  (ii)  $\Rightarrow$  (iii)  $\Rightarrow$  (iv)  $\Rightarrow$  (v)  $\Rightarrow$  (vi)  $\Rightarrow$  (i).

(i)  $\Rightarrow$  (ii) : soit  $\mathcal{G}$  un graphe connexe sans cycle. D'après le lemme 5.5.2, on a  $m \geq n - 1$ . D'après le lemme 5.5.3,  $m \leq n - 1$ , d'où l'égalité.

(ii)  $\Rightarrow$  (iii) : si  $\mathcal{G}$  connexe et  $n - 1$  arêtes avait un cycle, alors la suppression d'une arête du cycle entraînerait l'existence d'un graphe connexe à  $n$  sommets et  $n - 2$  arêtes.

(iii)  $\Rightarrow$  (iv) : si  $\mathcal{G}$  est sans cycle et non connexe, chaque composante connexe  $C_k$  pour  $k = 1..p$  est telle que  $m_k = n_k - 1$ . D'où  $m = n - p$  et  $p \geq 2$ , ce qui est absurde. S'il existe deux chaînes entre deux sommets, il existe un cycle, contradiction.

(iv)  $\Rightarrow$  (v) : un graphe vérifiant (iv) est forcément connexe. Si  $\mathcal{G}$  restait connexe après disparition de l'arête  $\{a, b\}$ , il existerait une chaîne reliant  $a$  à  $b$ , ce qui contredirait l'unicité.

(v)  $\Rightarrow$  (vi) : un tel graphe est nécessairement sans cycle, sinon la disparition d'une arête ne suffirait pas à le déconnecter. Si on ajoute une arête  $\{a, b\}$ , on crée un cycle, puisqu'il existe déjà une chaîne entre  $a$  et  $b$ ; et ce cycle passerait par  $\{a, b\}$ . Si l'ajout créait deux cycles, c'est qu'il existait déjà deux chaînes.

(vi)  $\Rightarrow$  (i) : si l'ajout de  $\{a, b\}$  crée un cycle,  $G$  est connexe, car sinon un pont entre deux composantes connexes ne créerait pas de cycle.  $\square$

## Exercices

1. [Centre d'un graphe orienté] Soit  $s$  un sommet du graphe  $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ . L'*excentricité* de  $s$  est définie par :

$$\max_{t \in \mathcal{S}} \{\text{longueur du plus court chemin entre } s \text{ et } t\}.$$

Un *centre* du graphe est un sommet d'excentricité minimale. Donner un algorithme de recherche du centre d'un graphe.

# Chapitre 6

## Parcours

Dans ce chapitre, nous présentons des algorithmes fondamentaux permettant de découvrir des propriétés des graphes, orientés ou non. Une connaissance globale d'un graphe est généralement difficile (par exemple une représentation dense prendrait trop de place en mémoire), mais les algorithmes de parcours permettent de visiter les structures et d'en déduire des informations comme la connexité et la forte connexité, que nous verrons au chapitre 7. Les parcours permettent de résoudre de multiples problèmes, comme l'accessibilité de deux sommets, la recherche de cycles ou de circuits, ou encore trouver des solutions à des problèmes d'ordonnement.

### 6.1 Le cas non orienté

#### 6.1.1 Définitions et propriétés générales

Nous commençons par supposer que le graphe  $\mathcal{G}$  est non orienté et connexe.

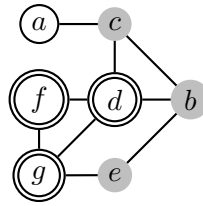
**Définition 6.1.1** *Un parcours de  $\mathcal{G} = (\mathcal{S}, \mathcal{A})$  à partir du sommet  $s$  est une liste de sommets  $L$  telle que :*

- le premier sommet de  $L$  est  $s$  ;
- chaque sommet de  $\mathcal{S}$  apparaît une fois et une seule dans  $L$  ;
- tout sommet de la liste (sauf le premier) est adjacent dans  $\mathcal{G}$  à au moins un sommet placé avant lui dans la liste.

À partir du sommet  $s$ , on explore une partie des sommets du graphe, les voisins de  $s$ . De proche en proche, nous sommes conduits à choisir les sommets suivants dans la bordure du graphe.

**Définition 6.1.2** *La bordure  $\mathcal{B}(T)$  de  $T \subset \mathcal{S}$  est l'ensemble des sommets de  $\mathcal{S} - T$  adjacents à  $T$ .*

Considérons par exemple le graphe  $\mathcal{G}_1$  de la figure 6.1. La bordure de  $\{d, f, g\}$  est  $\{b, c, e\}$ .

FIG. 6.1 – Le graphe  $\mathcal{G}_1$ .

**Définition 6.1.3** Le support  $\sigma(L)$  de  $L$  est l'ensemble des sommets contenus dans  $L$ .

Par abus de notation, on notera souvent  $\mathcal{B}(L) = \mathcal{B}(\sigma(L))$ .

Une fois ces notations introduites, on peut esquisser un algorithme générique (non déterministe) de parcours d'un graphe  $\mathcal{G}$ .

```
parcoursGenerique(G, s)
1. L <- (s);
2. tantque B(L) est non vide faire
   2.1 choisir v dans B(L);
   2.2 L <- L # v;
```

À l'étape 2.1, on choisit un sommet, le choix dépendant des algorithmes. On parle souvent d'*exploration* du sommet.

Tout parcours induit une (re)numérotation des sommets de  $\mathcal{G}$ , par ordre d'apparition dans le parcours. On peut modifier le programme de façon à numéroter les sommets :

```
parcoursGenerique(G, s)
1. numéro <- tableau de taille n = |S|; num <- 0;
2. L <- (s);
   numéro[s] <- num; num++;
3. tantque B(L) est non vide faire
   3.1 choisir v dans B(L);
   3.2 L <- L # v;
   3.3 numéro[v] <- num; num++;
```

Insistons sur le fait qu'il n'existe pas de parcours canonique d'un graphe, tout comme il n'y a pas de représentation canonique.

Dans les programmes précédents, la bordure de  $L$  évolue de manière dynamique.

**Proposition 6.1.1** Chaque sommet de  $\mathcal{G}$  est choisi une fois et une seule dans un parcours.

Il est difficile de parler de complexité générique, car nous n'avons pas spécifié comment gérer la bordure. Disons qu'au mieux, la complexité sera  $O(n + m)$ .

### 6.1.2 Arbre couvrant

**Proposition 6.1.2** Soit  $L = (x_0, \dots, x_{n-1})$  un parcours de  $\mathcal{G}$ . Pour tout  $k$ ,  $1 \leq k < n$ , soit  $x'_k$  un sommet de  $(x_0, \dots, x_{k-1})$  adjacent à  $x_k$ . Le sous-graphe induit par les arêtes  $(x_k, x'_k)$ ,  $1 \leq k < n$ , est un arbre, appelé arbre couvrant de  $G$  (relatif à  $L$ ).

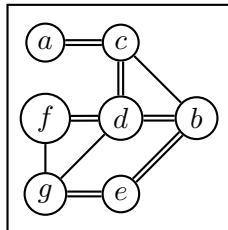
**Définition 6.1.4** Les arêtes  $(x_k, x'_k)$  sont appelées arêtes de liaison.

*Démonstration.* On note  $G_p$  le graphe induit par  $k \in \{0, \dots, p\}$  et on raisonne par récurrence sur  $p$ .

$p = 1$  : seule arête  $(x_0, x_1)$ , donc  $x'_1 = x_0$ .

$p > 1$  :  $G_{p-1}$  connexe  $\Rightarrow G_p$  connexe car on a rajouté une arête avec un sommet dans  $G_{p-1}$ ;  $G_p$  a  $p - 1$  arêtes et  $p$  sommets, n'a pas de cycle, donc est un arbre (en utilisant la proposition 5.5.1).  $\square$

Ex. Un arbre couvrant du graphe  $\mathcal{G}_1$  relatif à  $L = (b, d, e, g, f, c, a)$  est dessiné avec des traits doubles ci-dessous :

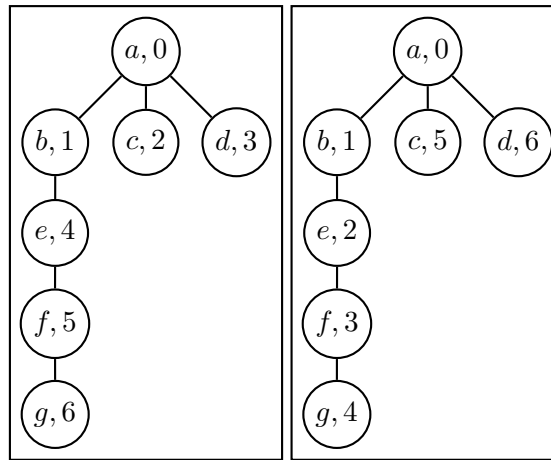


Que faire quand  $\mathcal{G}$  n'est pas connexe ? On généralise à une *forêt* de recouvrement, en construisant un arbre couvrant par composante connexe.

### 6.1.3 Les deux parcours les plus fréquents

On peut programmer un parcours quelconque, en tirant au sort un sommet dans la bordure. Parmi tous les parcours possibles, deux sont très souvent employés, le *parcours en largeur d'abord*, et le *parcours en profondeur d'abord*.

Comparons sur un graphe très simple ces deux types de parcours. À partir d'un sommet donné, le parcours en largeur va au plus près (il trouve d'abord ses voisins), alors que dans le parcours en profondeur, on va au plus loin d'abord :



### 6.1.4 Parcours en largeur d'abord

L'idée de base du parcours en largeur d'abord (*breadth-first search*) consiste à parcourir les sommets d'un graphe à partir d'un sommet  $s$  donné en procédant par cercles concentriques.

Sur l'arbre binaire de la figure 6.2, en partant de la racine de l'arbre, cela revient à numéroter les sommets comme indiqué. On explore  $a$  et on stocke ses deux voisins  $b$ ,  $c$ . On repart des voisins de  $b$ , à savoir  $d$  et  $e$ ; puis on prend les voisins de  $c$ , à savoir  $f$  et  $g$ .

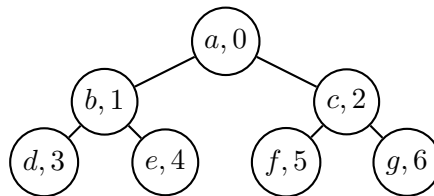


FIG. 6.2 – Parcours en largeur d'abord d'un arbre binaire.

Esquissons l'algorithme :

```
BFS( $G, s$ )
1. explorer les voisins de  $s$ ;
2. explorer les voisins des voisins, etc.
```

Le point 1 est facile à réaliser, il suffit de parcourir la liste des voisins de  $s$ , qui sont les plus proches du sommet de départ. Pour aller plus loin, on va mettre les sommets dans une file d'attente, et les explorer un par un. Cela donne :

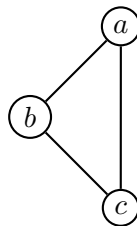
```
bfs( $G, s$ )
1.  $F \leftarrow (s)$ ;
```

```

2. L <- (s);
3. tantque F n'est pas vide faire
   t <- tête(F);
   L <- L # t;
   pour u voisin de t faire
     F <- F # u;

```

Il est facile de voir que l'algorithme esquissé ci-dessus va boucler indéfiniment sur le triangle suivant :



Pour ne pas boucler, il faut être capable de repérer un sommet déjà vu, ce qui donne le pseudocode :

```

bfs(G, s)
1. F <- (s);
2. L <- (s);
3. tantque F n'est pas vide faire
   t <- tête(F);
   L <- L # t;
   pour u voisin de t faire
     si u n'a pas déjà été vu alors
       F <- F # u;

```

Pour coder cette information plus précisément, on va gérer l'état d'un sommet. Au début de l'algorithme, tous les sommets sont dans l'état *inexploré*. Cet état va évoluer au cours du temps. À chaque fois que l'on défile un sommet, son état devient *exploré*. À chaque fois qu'on ajoute un sommet à la file, son état devient *encours*, de sorte qu'un même sommet ne soit pas rajouté de nombreuses fois lors de l'exploration. Le programme devient alors :

```

bfs(G, s)
0. pourtout sommet t faire etat[t] <- inexploré;
1. F <- (s); etat[s] <- encours;
2. tantque F n'est pas vide faire
   t <- tête(F);
   pour u voisin de t faire
     si etat[u] == inexploré alors
       etat[u] <- encours;
       F <- F # u;

```

```
etat[t] <- exploré;
```

Dans un parcours en largeur d'abord, le prochain sommet exploré est le plus ancien sommet de type `encours`.

**Proposition 6.1.3** *La complexité de bfs est  $O(|\mathcal{S}| + |\mathcal{A}|)$ .*

*Démonstration* : notons d'abord que chaque sommet est ajouté dans la file au plus une fois, s'il n'a pas déjà été vu. Le nombre de passage dans la boucle 2. est donc au plus  $n$ . Récupérer la tête de  $F$  coûte  $O(1)$  opérations. Pour chacun des  $n_t$  voisins de  $t$ , on teste l'état de la marque (coût en  $O(1)$ ), et si le test est bon, on modifie l'état du sommet (coût encore en  $O(1)$ ), puis on ajoute  $u$  à la file (encore  $O(1)$ ). Par suite, le coût est borné par :

$$\sum_{t \in \mathcal{S}} \left( O(1) + \sum_{u \text{ voisin de } t} O(1) \right) = O(n) + O\left(\sum_{t \in \mathcal{S}} n_t\right).$$

En utilisant le fait que  $\sum_{t \in \mathcal{S}} n_t = |\mathcal{A}| = m$ , on en déduit le résultat.  $\square$

Arrivés à ce point, la traduction en Java ne pose pas de problème majeur. L'état d'un sommet est stocké dans une table de hachage. On implante une file à l'aide de `LinkedList` et on utilise les méthodes idoines.

```
final static int inexploré = 0, exploré = 1, encours = 2;
public void bfs(Sommet s){
    Hashtable<Sommet,Integer> etat
        = new Hashtable<Sommet,Integer>();
    LinkedList<Sommet> f = new LinkedList<Sommet>();

    for(Sommet t : sommets()){
        etat.put(t, inexploré);
    etat.put(s, encours);
    f.addLast(s);
    while(! f.isEmpty()){
        Sommet t = f.removeFirst();
        for(Arc<Sommet> a : voisins(t)){
            Sommet u = a.destination();
            if(etat.get(u) == inexploré){
                etat.put(u, encours);
                f.addLast(u);
            }
        }
        etat.put(t, exploré);
    }
}
```

On peut également se passer de la valeur `encours` dans beaucoup de cas, mais cela simplifie l'exposition.

**Exercice 6.1.1** Montrer qu'on peut se contenter d'utiliser deux états pour un sommet, au lieu de trois.

Montrons sur un exemple le fonctionnement de l'algorithme pour le graphe dont les listes de voisins sont :

a: (g, e, b, f)

b: (c, a, d)

c: (b, e)

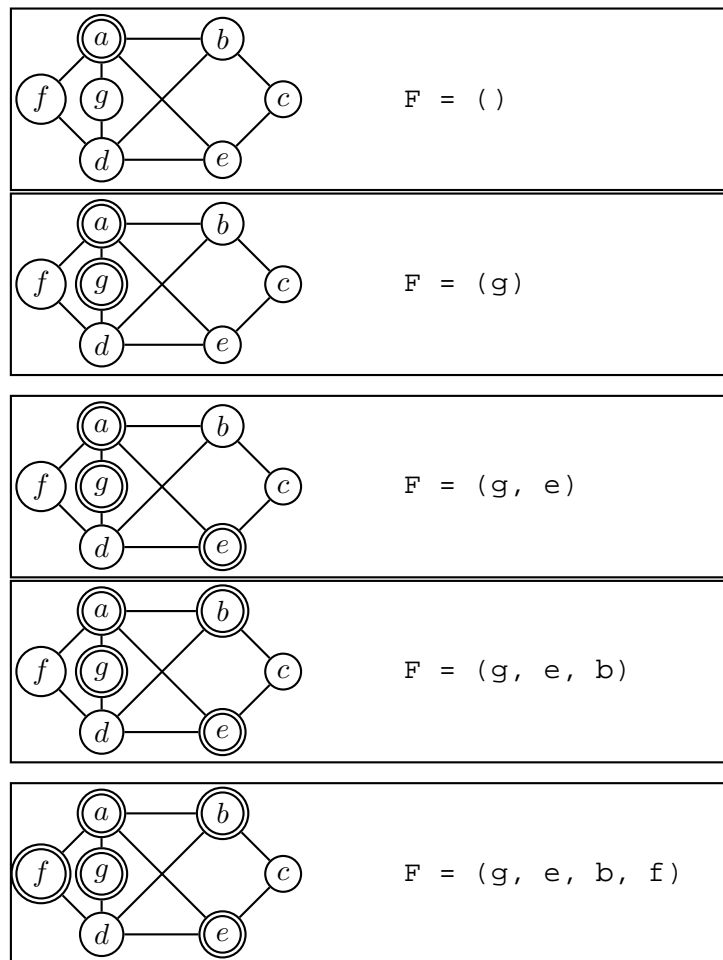
d: (b, f, e)

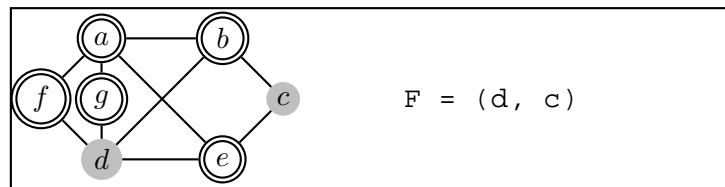
e: (c, d, a)

f: (a, d)

g: (a, d)

On indique dans les figures qui suivent l'état du graphe et l'état de la file au même moment :





En modifiant le programme pour numéroter les sommets dans l'ordre du parcours, on aurait, en utilisant une table de hachage pour les numéros, et en affichant les numéros au moment où ils sont calculés :

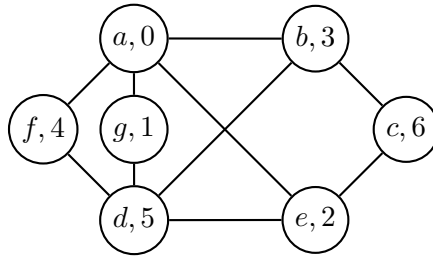
```

public void bfsNum(Sommet s){
    Hashtable<Sommet,Integer> etat =
        new Hashtable<Sommet,Integer>();
    Hashtable<Sommet,Integer> numero =
        new Hashtable<Sommet,Integer>();
    LinkedList<Sommet> f = new LinkedList<Sommet>();
    int num = 0;

    for(Sommet t : sommets()){
        etat.put(t, inexplorer);
        numero.put(t, 0);
    }
    etat.put(s, encours);
    f.addLast(s);
    numero.put(s, num++);
    while(! f.isEmpty()){
        Sommet t = f.removeFirst();
        for(Arc<Sommet> a : voisins(t)){
            Sommet u = a.destination();
            if(etat.get(u) == inexplorer){
                etat.put(u, encours);
                f.addLast(u);
                numero.put(u, num++);
            }
        }
        etat.put(t, explore);
    }
}

```

On trouve alors les numéros suivant :



Donnons un exemple d'application, celui où on cherche les distances entre  $s$  et les autres sommets. On va modifier le programme de sorte qu'il retourne un tableau des distances, ici une table de hachage indexée par les sommets.

```

public Hashtable<Sommet,Integer> bfsDistance(Sommet s){
    Hashtable<Sommet,Integer> etat =
        new Hashtable<Sommet,Integer>();
    LinkedList<Sommet> f = new LinkedList<Sommet>();
    Hashtable<Sommet,Integer> distance =
        new Hashtable<Sommet,Integer>();

    for(Sommet t : sommets()){
        etat.put(t, inexplorer);
        distance.put(t, 0);
    }
    etat.put(s, encours);
    f.addLast(s);
    while(! f.isEmpty()){
        Sommet t = f.removeFirst();
        System.out.println("J'explorer "+t);
        for(Arc<Sommet> a : voisins(t)){
            Sommet u = a.destination();
            if(etat.get(u) == inexplorer){
                etat.put(u, encours);
                f.addLast(u);
                distance.put(u, distance.get(t) + 1);
            }
        }
        etat.put(t, explore);
    }
    return distance;
}

```

Pour le graphe présenté ci-dessus, on trouve :

$a$	$b$	$c$	$d$	$e$	$f$	$g$
0	1	2	2	1	1	1

**Proposition 6.1.4** *L'algorithme `bfsDistance` calcule les distances minimales entre le sommet  $s$  et les autres sommets du graphe (qui sont dans sa composante connexe).*

*Démonstration.* Raisonnons par récurrence. À chaque entrée dans la boucle [1], les distances des sommets contenus dans la file  $F$  à  $s$  sont connues et exactes. C'est vrai à l'initialisation, puis quand on vient de visiter les voisins de  $s$ . À l'étape  $k$ , on ne rajoute dans la file que des sommets non encore visités, et qui sont donc les suivants à considérer, qui sont donc à une distance 1 de plus que le sommet  $t$  en cours d'exploration.  $\square$

Nous verrons au chapitre 9 un autre algorithme pour résoudre le même problème, avec cette fois des distances positives quelconques pour chaque arc (algorithme de Dijkstra).

**Exercice 6.1.2** Modifier l'algorithme bfs d'exploration pour détecter la présence de cycles dans un graphe.

### 6.1.5 Parcours en profondeur d'abord (*depth-first search*)

Dans ce cas-là, le prochain sommet visité est le plus récent sommet dont l'état a été déclaré *encours*. La présentation de l'algorithme est plus facile de manière récursive :

```
dfs(G, s)
0. pourtout sommet t faire etat[t] <- inexploré;
1. dfsRec(G, s, etat);

dfsRec(G, s, etat)
si etat[s] == inexploré alors
    etat[s] <- encours;
    pourtout t voisin de s
        dfsRec(G, t, etat);
    etat[s] <- exploré;
```

que l'on traduit en Java par :

```
public void dfsRec(Hashtable<Sommet,Integer> etat, Sommet s){
    if(etat.get(s) == inexplorable){
        etat.put(s, encours);
        for(Arc<Sommet> a : voisins(s))
            dfsRec(etat, a.destination());
        etat.put(s, explore);
    }
}
```

Dans cette écriture, le rôle des états *exploré* et *en cours* est plus clairement identifiable. Un sommet est *en cours* tant que tous ses voisins et leurs descendants ne sont pas explorés.

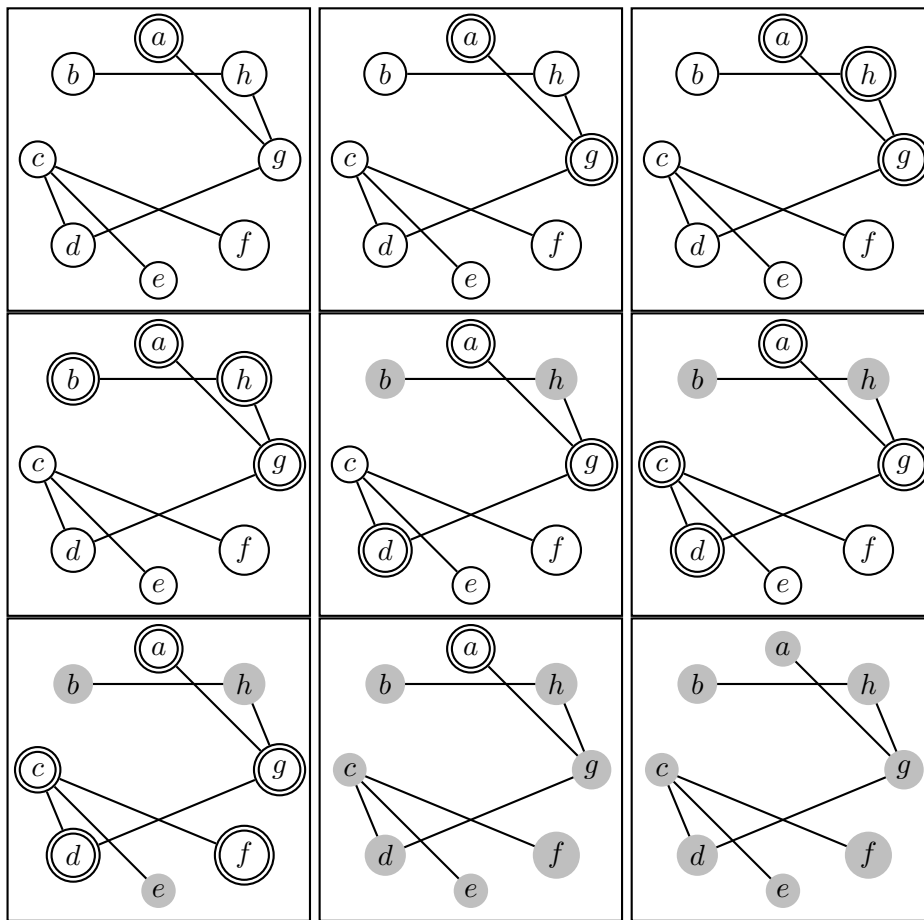


FIG. 6.3 – Exemple de déroulement de la dfs.

Regardons sur un exemple (voir figure 6.3) le comportement de l'algorithme. Un sommet encours est doublement cerclé, un sommet exploré est grisé.

Il est possible de dérécursiver l'algorithme sous la forme suivante (en incorporant également le calcul du rang), en donnant directement le code Java. Nous allons utiliser une pile `p` pour simuler la récursivité.

```

public void dfs(Hashtable<Sommet,Integer> etat, Sommet s){
    LinkedList<Sommet> p = new LinkedList<Sommet>();
    int rg = -1;

    p.addFirst(s);
    while(! p.isEmpty()){
        Sommet t = p.removeFirst();

        if(etat.get(t) != explore){

```

```

        System.out.println("J'explore "+t);
        etat.put(t, encours);
        ++rg;
        System.out.println("rang["+t+"]="+rg);
        for(Arc<Sommet> a : voisins(t)){
            Sommet u = a.destination();
            p.addFirst(u);
        }
    }
    etat.put(t, explore);
}
}

```

Le résultat suivant n'est que la copie du résultat sur bfs.

**Proposition 6.1.5** *L'algorithme dfs a complexité  $O(|S| + |A|)$ .*

*Démonstration.* les opérations de pile ont le même coût que les opérations de file.  $\square$

**Exercice 6.1.3** À quoi correspond une dfs sur un arbre ?

### 6.1.6 Construction de l'arbre couvrant

Nous allons plutôt construire la forêt de recouvrement, de façon à traiter le cas des graphes non connexes. Une forêt sera un ensemble d'arbres  $n$ -aires correspondant au parcours. Nous avons des arbres pour lesquels chaque nœud a *a priori* un nombre quelconque d'enfants. La classe proposée restera simple, chaque nœud de l'arbre étant une liste chaînée d'arbres :

```

import java.io.*;
import java.util.*;

class Arbre{
    Sommet racine;
    LinkedList<Arbre> fils;

    Arbre(Sommet r){
        racine = r;
        fils = new LinkedList<Arbre>();
    }

    void ajouterFils(Arbre A){
        fils.addLast(A);
    }
}

```

La fonction ci-dessous fabrique un arbre couvrant à partir du sommet  $s$ , en ajoutant un sous-arbre par fils de  $s$  (on a modifié le programme pour ne pas retourner des arbres **null**) :

```
// on retourne un arbre couvrant de racine s
public Arbre dfsRec(Hashtable<Sommet,Integer> etat,
                   Sommet s){
    etat.put(s, encours);
    Arbre A = new Arbre(s);
    System.out.println("J'explore "+s);
    for(Arc<Sommet> a : voisins(s)){
        Sommet t = a.destination();
        if(etat.get(t) == inexplorer)
            A.ajouterFils(dfsRec(etat, t));
    }
    etat.put(s, explore);
    return A;
}
```

Pour stocker la forêt, on convient que l'arbre possède une racine spéciale, que nous appellerons ALPHA.

```
public void dfsCouvrant(){
    Sommet ALPHA = new Sommet("ALPHA", 0);
    Arbre F = new Arbre(ALPHA);
    Hashtable<Sommet,Integer> etat =
        new Hashtable<Sommet,Integer>();

    for(Sommet s : sommets())
        etat.put(s, inexplorer);
    for(Sommet s : sommets())
        if(etat.get(s) == inexplorer)
            F.ajouterFils(dfsRec(s));
}
```

On trouve à la figure 6.4 l'arbre couvrant correspondant au parcours de la figure 6.3.

**Définition 6.1.5** Une arborescence est un arbre dont on a distingué un sommet, la racine.

**Définition 6.1.6** Nous appellerons arborescence de Trémaux un arbre couvrant obtenu lors d'une dfs.

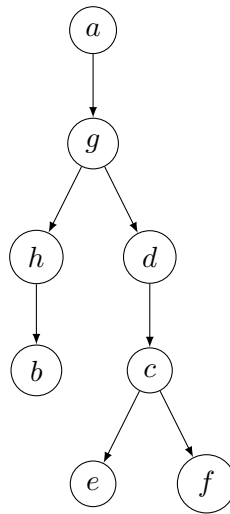


FIG. 6.4 – Arbre couvrant pour le parcours de la figure 6.3.

## 6.2 Parcours dans le cas orienté

### 6.2.1 Propriétés générales

On considère un graphe  $\mathcal{G}$  orienté sans boucle pour simplifier les définitions.

Les notions de bordure et parcours se généralisent sans problème au cas orienté.

**Définition 6.2.1** La bordure  $\mathcal{B}(T)$  d'une partie  $T \subset \mathcal{S}$  est le sous-ensemble des sommets de  $\mathcal{S} - T$  qui sont les extrémités d'un arc dont l'origine est dans  $T$ .

**Définition 6.2.2** Un parcours de  $\mathcal{G}$  est une liste  $L$  de sommets de  $G$  telle que :

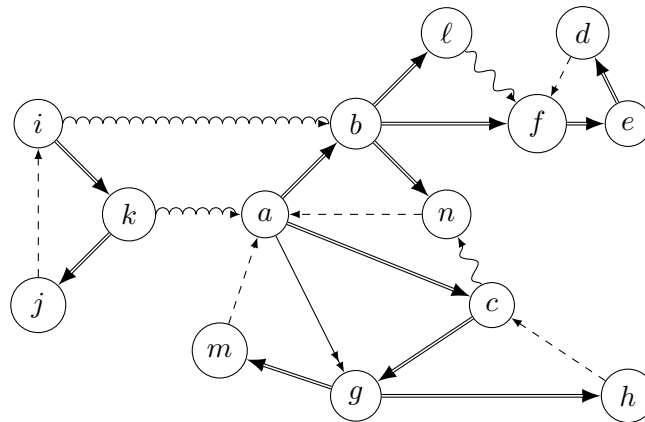
- chaque sommet de  $\mathcal{S}$  apparaît une fois et une seule dans  $L$  ;
- chaque sommet de  $L$  (sauf le premier) appartient à la bordure du sous-ensemble des sommets placés avant lui dans la liste, si toutefois celle-ci est non vide.

Les propriétés des parcours dans le cas non orienté se généralisent au cas orienté. On remplace les arêtes de liaison par les *arcs de liaison* (voir définition 6.1.4). On vérifie facilement :

**Proposition 6.2.1** Les arcs de liaison d'un parcours constituent une forêt couvrante de  $\mathcal{G}$ .

Pour le parcours  $L_3 = (a, b, f, e, d, \ell, n, c, g, h, m, i, k, j)$  du graphe  $\mathcal{G}_3$  de la figure 6.5, on trouve deux arborescences indiquées en traits doubles. Ces arborescences sont encore appelées arborescences de Trémaux.

Nous reviendrons plus loin sur la classification générale des arcs.

FIG. 6.5 – Le graphe  $\mathcal{G}_3$ .

### 6.2.2 Propriétés du parcours en profondeur d’abord

La dfs se programme comme dans le cas non orienté sans difficulté.

**Exercice 6.2.1** Quelle structure de données peut-on utiliser pour répondre facilement à la question :  $y$  est-il un descendant du sommet  $s$ , dans le cas où  $s$  est fixé ?

#### Le rang

On peut lire de nombreuses propriétés du graphe sur un parcours  $L$  en profondeur d’abord, comme le parcours  $L_3$  associé au graphe  $\mathcal{G}_3$  de la figure 6.7. Notons  $\text{rang}(x)$  le rang d’un sommet  $x$  dans le parcours  $L$ . Dans l’exemple,  $\text{rang}(a) = 0$ ,  $\text{rang}(b) = 1$ , etc.

Il est facile de calculer le rang des sommets, au moment où on effectue le parcours en profondeur d’abord :

```

dfsRang(G)
0. etat <- tableau de taille n;
1. rang <- tableau de taille n;
2. pourtout sommet s faire etat[s] <- inexploré;
3. rg <- 0;
4. tantqu’il reste un sommet s non exploré faire
5.     rg <- dfsRec(etat, rang, rg, s);

dfsRec(etat, rang, rg, s)
// s est en cours
10. etat[s] <- encours;
    rang[s] <- rg;
    rg <- rg + 1;
    pour t voisin de s faire

```

```

11.     si etat[t] == inexploré alors
           rg <- dfsRec(etat, rang, rg, t);
       etat[s] <- exploré;
12. retourner rg.

```

On gère le tableau rang qui se remplit progressivement avec les rangs des sommets. On retourne systématiquement la prochaine valeur de rang à utiliser (variable rg).

En Java, le tableau rang pourra être une Hashtable :

```

public void dfsRec(Hashtable<Sommet,Integer> etat,
                   Hashtable<Sommet,Integer> rang,
                   int rg, Sommet s){
    etat.put(s, encours);
    rang.put(s, rg++);
    System.out.println("J'explore "+s);
    for(Arc<Sommet> a : voisins(s)){
        Sommet t = a.destination();
        if(etat.get(t) == inexploré)
            rg = dfsRec(etat, rang, rg, t);
    }
}

void dfsRang(){
    Hashtable<Sommet,Integer> etat =
        new Hashtable<Sommet,Integer>();
    Hashtable<Sommet,Integer> rang =
        new Hashtable<Sommet,Integer>();
    for(Sommet s : sommets())
        etat.put(s, inexploré);
    int rg = 0;
    for(Sommet s : sommets())
        if(etat.get(s) == inexploré)
            rg = dfsRec(etat, rang, rg, s);
    for(Sommet s : sommets())
        System.out.print("rang[ "+s+" ]="+rang.get(s));
}

```

**Ex.** On a indiqué dans la figure 6.7 les rangs des sommets pour le parcours  $L_3$ .

### Arborescence préfixe

Chaque passage dans la ligne 5 de l'algorithme 6.2.2 crée une nouvelle arborescence dont le sommet  $s$  sera la racine. On peut modifier les fonctions précédentes de façon à fabriquer de la même façon l'arborescence correspondante.

Les rangs des nœuds d'une arborescence de Trémaux forment un intervalle des entiers  $[n, m]$ . L'arborescence est *préfixe*, c'est-à-dire que pour tout sommet  $s$ , les rangs des descendants de  $s$  est un sous-intervalle de  $[n, m]$  dont le plus petit élément est  $\text{rang}(s)$ .

Pour le parcours de la figure 6.7, on trouve que la première arborescence de Trémaux est celle de la figure 6.6.

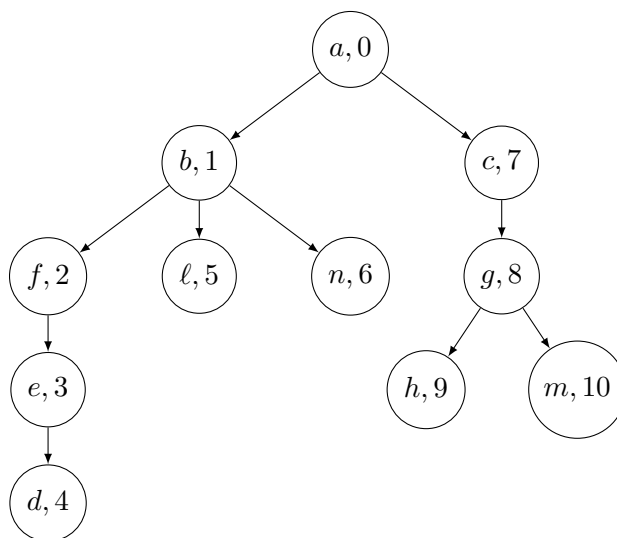


FIG. 6.6 – Première arborescence de Trémaux pour le parcours de la figure 6.7.

L'intervalle correspondant est  $[0..10]$ . Le premier sous-arbre de racine  $b$  correspond à  $[1..6]$ , subdivisé lui-même en  $[2..4]$  (suivi de  $[3..4]$  et  $[4]$ ),  $[5]$  et  $[6]$ ; le second à  $[7..10]$ , puis  $[8..10]$ , puis  $[9]$  et  $[10]$ .

### Les différents types d'arc

Soit  $\mathcal{F}$  la forêt couvrante induite de  $L$ . Rappelons que les arcs de  $\mathcal{F}$  sont les arcs de liaison. On peut distinguer trois autres types d'arcs dans  $\mathcal{G}$ .

**Définition 6.2.3** *Un arc  $(s, t)$  est :*

- avant si  $s$  est un ascendant de  $t$  dans  $\mathcal{F}$  ;
- arrière si  $t$  est un ascendant de  $s$  dans  $\mathcal{F}$  ;
- transverse inter-arbre si  $s$  et  $t$  appartiennent à deux arborescences différentes, ou transverse intra-arbre si  $s$  et  $t$  ont un ancêtre commun  $z$  dans  $\mathcal{F}$  distinct de  $s$  et  $t$ . Si on n'a pas besoin de distinguer ces deux cas, on parlera d'arc transverse.

Dans l'exemple de la figure 6.7, l'arc  $(a, g)$  est avant ;  $(j, i)$ ,  $(d, f)$ ,  $(h, g)$  sont arrière ;  $(i, b)$  et  $(k, a)$  sont transverses inter-arbre (deux arborescences différentes) ;  $(l, f)$  est transverse intra-arbre (ancêtre commun).

La proposition suivante rassemble les propriétés de la fonction rang.

**Proposition 6.2.2** (i) Si  $t$  est un descendant (resp. ascendant) strict de  $s$  dans l'arborescence, alors  $\text{rang}(s) < \text{rang}(t)$  (resp.  $\text{rang}(s) > \text{rang}(t)$ ).

(ii) Si  $(s, t)$  est un arc arrière, alors  $\text{rang}(t) < \text{rang}(s)$ .

(iii) Si  $(s, t)$  est un arc avant, alors  $\text{rang}(s) < \text{rang}(t)$ .

(iv) Le sommet  $t$  appartient à l'arborescence de racine  $s$  si et seulement si  $\text{rang}(s) \leq \text{rang}(t) \leq \text{rang}(s) + \mathcal{D}(s)$  où  $\mathcal{D}(s)$  est le nombre de descendants de  $s$ .

(v) Si  $(s, t)$  est un arc transverse, alors  $\text{rang}(s) > \text{rang}(t)$ . Autrement dit,  $s$  est visité après  $t$ .

*Démonstration.* Soient  $A_1, \dots, A_n$  les arborescences correspondant à  $L$ . Par définition des arborescences, il n'existe pas d'arc  $(u, v)$  avec  $u \in A_i$  et  $v \in A_j$  pour  $i < j$ .

(i-iv) On ne fait là que traduire les propriétés élémentaires de la fonction rang.

(v) Si  $(s, t)$  est tel que  $s \in A_i$  et  $t \in A_j$  avec  $i \neq j$ , on a nécessairement  $i > j$  et donc  $\text{rang}(s) > \text{rang}(t)$ .

Dans le second cas,  $s$  et  $t$  sont dans la même arborescence et ont un ancêtre commun  $z \notin \{s, t\}$ . Si  $\text{rang}(s) < \text{rang}(t)$ , quand  $s$  est visité,  $t$  ne l'est pas encore et donc  $s$  sera un ascendant de  $t$ , ce qui est absurde, l'arc  $(s, t)$  serait soit de liaison, soit avant.  $\square$

Montrons comment déduire de ces résultats un algorithme qui donne les types des arcs lors de la dfs. À l'étape 11, on déduit que  $(s, t)$  est un arc de liaison.

Si le sommet  $t$  a déjà été visité, il peut se passer différentes choses. Si  $t$  est dans l'état *encours*, cela veut dire que  $t$  est un ascendant de  $s$ , donc que  $(s, t)$  est un arc arrière.

Si  $t$  a déjà été exploré, son rang a été calculé. Soit  $r$  la racine de l'arborescence courante. Si  $\text{rang}(t) < \text{rang}(r)$ , c'est que  $t$  est dans une arborescence dont l'exploration est terminée, donc que  $(s, t)$  est transverse inter-arbre. Dans le cas contraire,  $t$  appartient à l'arborescence de racine  $r$ , contenant  $s$ ;  $t$  ne peut être un ascendant de  $s$ , sinon son état ne serait pas exploré, mais *encours*. Si  $\text{rang}(s) < \text{rang}(t)$ ,  $t$  a été visité (et exploré) après  $s$ ; l'arc  $(s, t)$  est donc avant; si  $\text{rang}(s) > \text{rang}(t)$ ,  $s$  est exploré après  $t$ , donc  $(s, t)$  est transverse intra-arbre.

Il suffit donc de garder la racine de l'arborescence courante pour avoir toutes les informations dont nous avons besoin. Nous donnons à la figure 6.1 un programme qui écrit à l'écran les arcs visités ainsi que leur type.

Listing 6.1 – Trouver les types des arcs

```
dfsRec(etat, rang, rg, racine, s)
// s est non exploré,
// racine est la racine de l'arborescence courante
10. etat[s] <- encours;
    rang[s] <- rg;
    rg <- rg + 1;
    pour t voisin de s faire
11.     si etat[t] == inexploré alors
        rg <- dfsRec(etat, rang, rg, racine, t);
```

```

12.   sinon si etat[t] == encours alors
      écrire "(s, t) arrière";
      sinon // t est déjà exploré
        si rang[t] < rang[racine] alors
          écrire "(s, t) transverse inter-arbre";
        sinon si rang[s] < rang[t] alors
          écrire "(s, t) avant";
        sinon // rang[s] > rang[t]
          écrire "(s, t) transverse intra-arbre";
      etat[s] <- exploré;
13. retourner rg.

```

Nous complétons la classification des arcs pour notre graphe  $\mathcal{G}_3$  dans la figure 6.7.

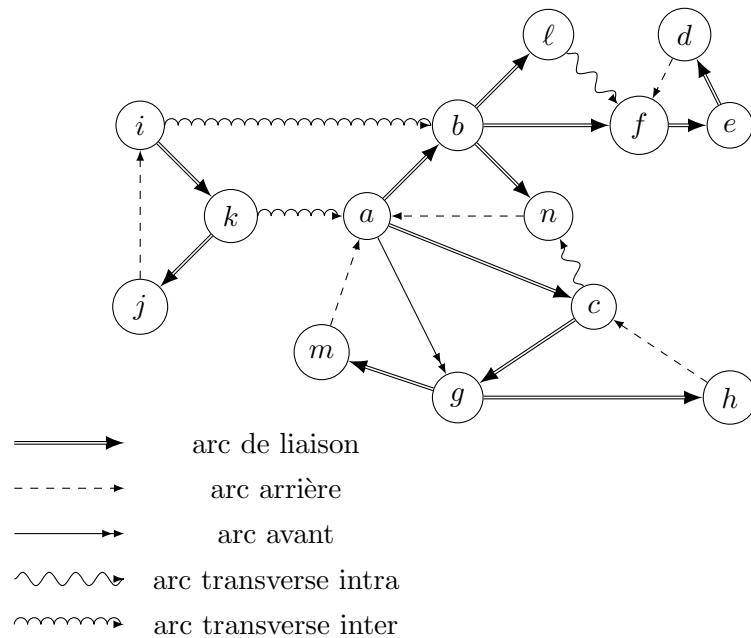


FIG. 6.7 – Le graphe  $\mathcal{G}_3$ .

**Exercice 6.2.2** Modifier l'algorithme 6.1 pour assigner à chaque sommet le numéro de l'arborescence dans lequel il est découvert.

Les propriétés que nous venons de démontrer, ainsi que les algorithmes décrits, formeront la base du calcul des composantes fortement connexes présenté au chapitre 7. Nous allons donner deux autres applications pour clore ce chapitre.

### 6.2.3 Graphes sans cycles

**Proposition 6.2.3** *Soit  $L$  un parcours en profondeur d'abord d'un graphe orienté  $\mathcal{G}$ . Le graphe  $\mathcal{G}$  est sans circuit si et seulement s'il n'existe pas d'arc arrière.*

*Démonstration.* S'il existe un arc arrière, il referme un circuit, donc la condition est nécessaire.

Supposons que  $\mathcal{G}$  contienne un circuit  $C$ . Après avoir effectué la dfs correspondant à  $L$ , soit  $s$  le sommet de  $C$  de rang minimum. Soit  $t$  le prédécesseur de  $s$  sur le circuit ;  $t$  doit être un descendant de  $s$  dans une arborescence d'exploration ; donc  $(t, s)$  n'est pas transverse. Puisque  $\text{rang}(t) > \text{rang}(s)$ , ce ne peut être un arc de liaison. Donc  $(t, s)$  est un arc arrière.  $\square$

**Exercice 6.2.3** Montrer comment modifier l'algorithme précédent pour détecter si un graphe contient un circuit.

### 6.2.4 Tri topologique

**Définition 6.2.4** *Une liste topologique des sommets de  $\mathcal{G} = (\mathcal{S}, \mathcal{A})$  est une permutation  $(s_0, s_1, \dots, s_{n-1})$  des sommets de  $\mathcal{S}$  telle que pour tout arc  $(s_i, s_j)$ , on a  $i < j$ .*

Si l'on imagine que les arcs désignent l'ordre de tâches à effectuer, autrement dit que  $(s, t) \in \mathcal{A}$  si  $s$  doit être exécutée avant  $t$ , une liste topologique induit un ordre dans lequel quand chaque tâche est effectuée, alors toutes les précédentes dans le graphe l'ont été avant. La proposition qui suit montre qu'une telle liste existe s'il n'existe pas de tâches mutuellement dépendantes.

**Proposition 6.2.4** *Un graphe  $\mathcal{G} = (\mathcal{S}, \mathcal{A})$  est sans circuit si et seulement s'il existe une liste topologique des sommets de  $\mathcal{G}$ .*

*Démonstration.* Si  $\mathcal{G}$  possède une liste topologique, c'est qu'il existe une permutation  $(s_0, s_1, \dots, s_{n-1})$  des sommets de  $\mathcal{S}$  telle que pour tout arc  $(s_i, s_j)$ , on a  $i < j$ . S'il existait un circuit  $(s_{i_0}, s_{i_1}, \dots, s_{i_r}, s_{i_0})$ , on aurait  $i_0 < i_1 < \dots < i_r < i_0$ , d'où une contradiction.

Démontrons la condition nécessaire par récurrence sur  $n = |\mathcal{S}|$ . La propriété est vraie pour  $n = 1$ . Supposons la vraie pour  $n - 1$  avec  $n \geq 2$ . Soit  $\mathcal{G}$  un graphe sans circuit avec  $n$  sommets. Il existe au moins un sommet  $s$  sans descendants propres, car sinon nous aurions un circuit. Le sous-graphe  $\mathcal{G}'$  de  $\mathcal{G}$  induit par  $\mathcal{S} - \{s\}$  a  $n - 1$  sommets et ne possède pas de circuit. Appliquant l'hypothèse de récurrence, soit  $L'$  une liste topologique pour les sommets de  $\mathcal{G}'$ . La liste  $L' \#(s)$  est une liste topologique pour les sommets de  $\mathcal{G}$ .  $\square$

On peut modifier l'algorithme de dfs pour stocker l'ordre topologique des sommets d'un graphe sans circuit dans une liste  $L$ . Il suffit de rajouter à la liste  $L$  le sommet en fin d'exploration. Dans cet algorithme, nous n'avons en fait pas besoin du rang, ni de la racine :

```

triTopologique(G)
1. pourtout sommet s faire etat[s] <- inexploré;
2. L <- NIL;
3. tantqu'il reste un sommet s non exploré faire
   dfsTopo(L, etat, s);

dfsTopo(L, etat, s)
// s est non exploré
  etat[s] <- encours;
10. pour t voisin de s faire
    si etat[t] == inexploré alors
      rg <- dfsTopo(L, etat, t);
    sinon si etat[t] == encours alors erreur;
  etat[s] <- exploré;
  L <- s # L;

```

Pour le graphe  $\mathcal{G}_4$  dessiné à la figure 6.8, on trouve que  $L = (d, e, g, f, a, b, c)$ .

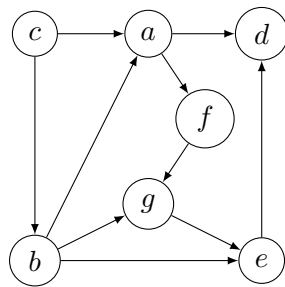


FIG. 6.8 – Le graphe  $\mathcal{G}_4$ .



# Chapitre 7

## Topologie

Nous présentons dans ce chapitre des algorithmes élaborés pour répondre à des questions liées à la topologie des graphes, comme la connexité, la forte connexité et la planarité. Décomposer un graphe en composantes connexes ou fortement connexes permet de réduire un problème sur un graphe à l'étude du problème sur les différents morceaux, ce qui permet souvent de gagner du temps. On en verra un exemple avec les tests de planarité. Les références utilisées pour ce chapitre sont [5, 3, 10, 2].

### 7.1 Forte connexité

On s'intéresse ici à un graphe  $\mathcal{G}$  orienté. Le concept de connexité n'est pas très pertinent dans ce cas-là, à moins qu'on ne s'intéresse aux propriétés du graphe sous-jacent.

#### 7.1.1 Définition, exemples

**Définition 7.1.1** *On dira qu'un graphe  $\mathcal{G} = (\mathcal{S}, \mathcal{A})$  est fortement connexe si et seulement si pour tout couple  $(s, t) \in \mathcal{A}$ , il existe un chemin de  $s$  à  $t$  et un chemin de  $t$  à  $s$ .*

Le graphe  $\mathcal{G}_1$  de la figure 7.1 est fortement connexe, le graphe  $\mathcal{G}_2$  ne l'est pas.

Trouver les composantes fortement connexes d'un graphe est un problème important dans plusieurs domaines de l'informatique. Nous allons montrer comment un parcours en profondeur d'abord peut nous apporter des informations cruciales sur ces composantes.

Nous nous appuyerons sur l'exemple déjà donné au chapitre 6. Sur la figure 7.2, nous indiquons les composantes fortement connexes trouvées à la main.

Dans tout ce qui suit, on suppose qu'on a fixé un parcours  $L$  et que les sommets ont un rang calculé lors du parcours.

**Notation :** on notera  $C(u)$  la composante fortement connexe contenant  $u$ .

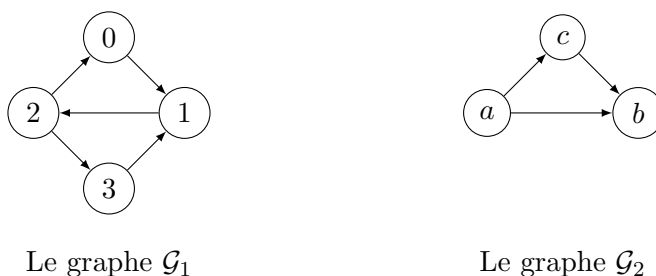
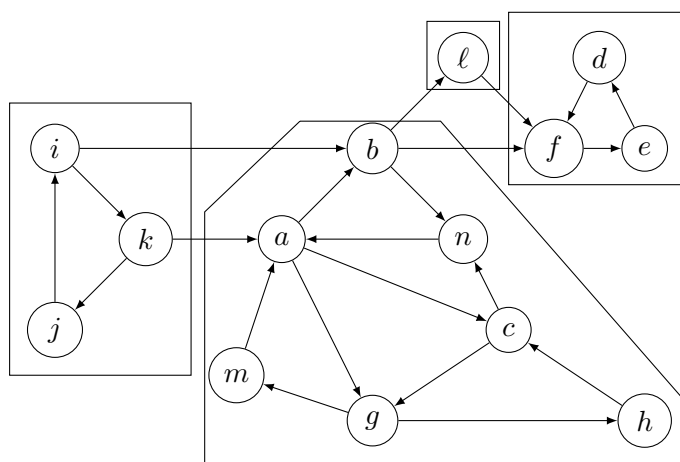


FIG. 7.1 – Exemples de graphe.

FIG. 7.2 – Le graphe  $\mathcal{G}_3$  et ses composantes fortement connexes.

Un algorithme naïf consisterait à partir d'un sommet  $x$  et à établir la liste de ses descendants. Pour un tel sommet  $y$ , il suffirait de faire une dfs dans le graphe *renversé* de  $\mathcal{G}$  (c'est-à-dire inverser les flèches), à partir de  $x$  pour voir s'il existe un chemin vers  $y$ . Cet algorithme serait assez coûteux, et nous allons donner un algorithme plus rapide, mais plus complexe et dû à Tarjan.

### 7.1.2 Le retour de Trémaux

Les sommets qui ne sont pas dans une arborescence de racine  $r$  ne sont pas dans  $C(r)$ , mais le contraire n'est pas vrai.

**Définition 7.1.2** Une sous-arborescence  $(Y', T')$  de racine  $r'$  d'une arborescence  $(Y, T)$  de racine  $r$  est constituée par  $Y' \subset Y$ ,  $T' \subset T$ , tel que  $(Y', T')$  forme une arborescence de racine  $r'$ .

On va montrer que la composante fortement connexe  $C(u)$  est une sous-arborescence de  $(Y, T)$ . Plus précisément :

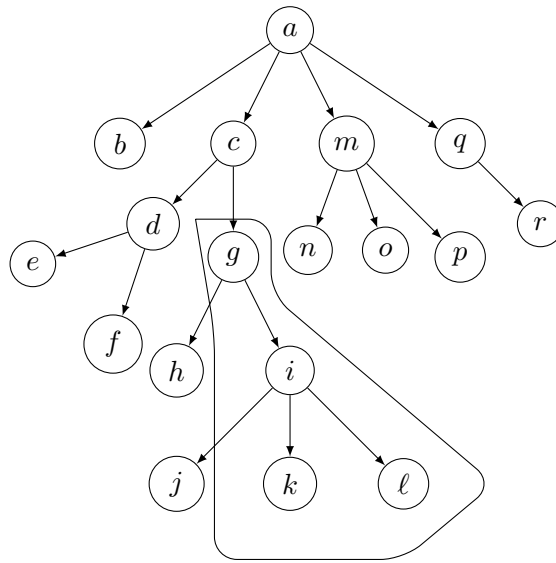


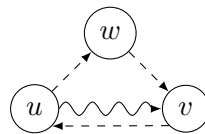
FIG. 7.3 – Exemple de sous-arborescence.

**Proposition 7.1.1** Soit  $(Y, T)$  une arborescence de Trémaux de racine  $x$ . Soit  $u$  un sommet de  $Y$  et  $u_0$  le sommet de plus petit rang dans  $C(u)$ . Alors :

- (i) pour tout  $v \in C(u)$ , tous les sommets du chemin de  $(Y, T)$  joignant  $u_0$  à  $v$  sont dans  $C(u)$  ;
- (ii) tous les éléments de  $C(u)$  sont des descendants de  $u_0$  dans  $(Y, T)$ .

*Démonstration.*

(i) Si  $u$  et  $v$  sont dans la même composante fortement connexe et  $w$  est sur un chemin de  $u$  à  $v$ , alors il existe un chemin entre  $u$  et  $w$  ainsi qu'entre  $w$  et  $u$ , donc  $w \in C(u)$ .



(ii) On va montrer que pour tout  $v$  dans  $C(u)$ ,  $v$  est un descendant de  $u_0$  dans  $(Y, T)$ . Supposons que ce ne soit pas vrai. Il existe un chemin  $c$  d'origine  $u_0$  et extrémité  $v$ . Soit  $w$  le premier sommet du chemin qui n'est pas un descendant de  $u_0$ , et  $w'$  le sommet précédent :

$$c = (u_0, \dots, w', w, \dots, v).$$

L'arc  $(w', w)$  n'est pas un arc de liaison (il n'est pas dans  $T$ ), et n'est pas un arc avant (sinon on pourrait le remplacer par une suite d'arcs de  $T$ ). Il est donc arrière ou transverse, et donc

$$\text{rang}(w) < \text{rang}(w').$$

D'après la propriété (i),  $w$  est dans  $C(u)$  et par minimalité, on a  $\text{rang}(u_0) < \text{rang}(w)$ . Mais comme  $T$  est préfixe, cela veut justement dire que  $w$  est descendant de  $u_0$ , contradiction.  $\square$

### 7.1.3 Points d'attache

**Définition 7.1.3** Soit  $x \in \mathcal{G}$  racine de l'arborescence de Trémaux  $(Y, T)$ . Le point d'attache  $at(y)$  de  $y \in Y$  est le sommet de plus petit rang extrémité d'un chemin de  $\mathcal{G}$ , d'origine  $y$  et contenant au plus un arc  $(u, v)$  tel que  $\text{rang}(u) > \text{rang}(v)$  (i.e., un arc de retour ou un arc transverse intra arbre). On suppose que le chemin vide d'origine et extrémité  $y$  est un tel chemin et donc  $\text{rang}(at(y)) \leq \text{rang}(y)$ . Le rang du point d'attache est appelé rang d'attache.

Un chemin de  $y$  à son point d'attache dans  $\mathcal{G}$  est soit vide (et  $at(y) = y$ ), ou bien une suite d'arcs **dans**  $T$  suivis par un arc arrière ou un arc transverse. Les arcs avant ne servent à rien dans la recherche, ils peuvent être remplacés par les arcs de liaison (donc dans  $T$ ). Les arcs de  $T$  partant de  $y$  conduisent à des sommets de rang supérieur à  $\text{rang}(y)$ .

**Proposition 7.1.2** Le point d'attache  $at(y)$  est le sommet de plus petit rang parmi les sommets suivants :

- $y$  lui-même;
- les points d'attache des fils de  $y$  dans  $(Y, T)$ ;
- les extrémités des arcs transverse (intra arbre) ou arrière d'origine  $y$ .

Nous donnons dans la figure 7.4 les rangs des sommets de  $\mathcal{G}_3$  ainsi que leur rang d'attache, pour le parcours  $L_3 = (a, b, f, e, d, l, n, c, g, h, m, i, k, j)$ .

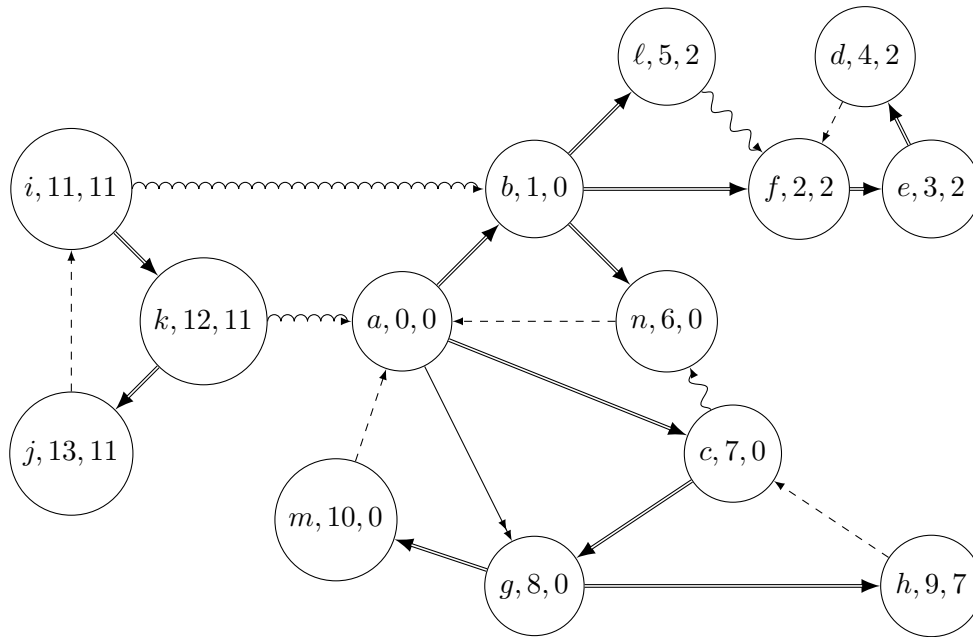
La recherche des points d'attache se fait facilement par modification de l'algorithme de classification des arcs donnés à la figure 6.1, ce que nous donnons à la figure 7.1.

Listing 7.1 – Calcul des rangs d'attache

```

dfsRec(etat, rang, rg, rat, racine, s)
// s est non exploré,
// racine est la racine de l'arborescence courante
10. etat[s] <- encours;
    rang[s] <- rg;
    rat[s] <- rang[s];
    rg <- rg + 1;
    pour t voisin de s faire
11.     si etat[t] == inexploré alors
        rg <- dfsRec(etat, rang, rg, racine, t);
        rat[s] <- min(rat[s], rat[t]);
12.     sinon si etat[t] == encours alors
        // (s, t) est un arc arrière

```

FIG. 7.4 – Les rangs d'attache pour les sommets de  $\mathcal{G}_3$ .

```

    rat[s] <- min(rat[s], rang[t]);
sinon // t est déjà exploré
    si rang[t] < rang[racine] alors
        // (s, t) transverse inter-arbre,
        // on ne fait rien
    sinon si rang[s] < rang[t] alors
        // (s, t) avant, on ne fait rien
    sinon // rang[s] > rang[t]
        // (s, t) transverse intra-arbre
        rat[s] <- min(rat[s], rang[t]);
    etat[s] <- exploré;
13. retourner rg.

```

#### 7.1.4 Application à la forte connexité

**Théorème 7.1.1** *Si  $u \in Y$  satisfait :*

(i)  $u = at(u)$  ;

(ii) *Pour tout descendant  $v$  de  $u$  dans  $(Y, T)$ , on a  $\text{rang}(at(v)) < \text{rang}(v)$ .*

*Alors l'ensemble des descendants de  $u$  dans  $(Y, T)$  forme une composante fortement connexe de  $\mathcal{G}$ .*

*Le sommet  $u$  est appelé point d'entrée du parcours dans la composante fortement connexe.*

*Démonstration.*

a) Soit  $v$  un descendant de  $u$  et montrons que  $v \in C(u)$ . Il est donc extrémité d'un chemin d'origine  $u$ . On veut montrer que  $u$  est extrémité d'un chemin d'origine  $v$ . Si ce n'est pas le cas, on peut toujours supposer que  $v$  est le sommet de plus petit rang pour lequel on ne puisse pas joindre  $u$ . Soit  $c_1$  le chemin joignant  $v$  à  $at(v)$ . Soit  $c_2$  un chemin de  $(Y, T)$  joignant  $u$  à  $v$  et  $c_3 = c_1 \cup c_2 = (u, \dots, v, \dots, at(v))$ . Par définition de  $at(v)$ , ce chemin contient au plus un arc de retour ou transverse :

$$\text{rang}(u) = \text{rang}(at(u)) \leq \text{rang}(at(v)) < v.$$

Comme  $(Y, T)$  est préfixe, on en déduit que  $at(v)$  est un descendant de  $u$ . En utilisant l'hypothèse de minimalité sur  $v$ , on voit qu'il existe un chemin  $c_4$  de  $at(v)$  à  $u$  dans  $(Y, T)$  et par suite  $c_2 \cup c_4$  est un chemin de  $v$  à  $u$ , contradiction.

b) Soit  $w \in C(u)$ . Montrons qu'il est un descendant de  $u$ , ce qui se fera à l'aide du résultat auxiliaire suivant :

**Lemme 7.1.1** *Tout arc dont l'origine est dans  $\text{desc}(u)$  a aussi son extrémité dans  $\text{desc}(u)$ .*

*Démonstration du lemme :*

Soit  $(v_1, v_2) \in \mathcal{A}$  tel que  $v_1 \in \text{desc}(u)$ . Si  $\text{rang}(v_2) > \text{rang}(v_1)$ ,  $v_2$  est un descendant de  $v_1$  qui est descendant de  $u$ , donc  $v_2$  aussi. Si  $\text{rang}(v_2) < \text{rang}(v_1)$ , le chemin de  $u$  à  $v_2$  contient exactement un arc arrière ou transverse, donc

$$\text{rang}(u) = \text{rang}(at(u)) \leq \text{rang}(v_2) < \text{rang}(v_1),$$

donc  $v_2 \in \text{desc}(u)$  en utilisant une fois encore la préfixité de  $(Y, T)$ .  $\square$

### 7.1.5 L'algorithme

L'algorithme consiste à faire un parcours en profondeur d'abord (dfs) de façon à calculer le rang ainsi qu'une fonction qui s'apparente au rang d'attache dans le même temps, et d'en déduire les composantes fortement connexes à la volée. On descend une arborescence jusqu'à trouver un point d'entrée. Une fois celui-ci trouvé, on en déduit la composante fortement connexe, qu'on retire du graphe, et on continue.

Pour y voir plus clair (et bien qu'on n'en ait pas vraiment besoin), on peut gérer deux types d'état pour un sommet. On a toujours les états classiques (inexploré, encours, exploré), mais également un état indiquant si le sommet est dans une composante en construction ou pas. L'état correspondant sera *libre*, *empilé*, ou *exclus* quand on a trouvé la composante fortement connexe du sommet. On a les propriétés :

$$\text{libre} \Leftrightarrow \text{inexplore}, \text{encours} \Rightarrow \text{empile}, \text{exclus} \Rightarrow \text{explore}.$$

Au départ, tous les sommets sont dans l'état *libre*. Quand on n'a pas encore trouvé la composante fortement connexe contenant le sommet  $s$ , son état est *empilé*, car nous allons stocker tous ces sommets dans la pile `pile`, qui est différente de la pile d'appels

de la dfs. Quand on aura trouvé une composante fortement connexe et dépilé tous les sommets la constituant, ceux-ci seront dits exclus.

La fonction d'appel est :

```
cfc(G)
1. etat <- tableau de taille n;
   etat_cfc <- tableau de taille n;
   rang <- tableau de taille n;
   prat <- tableau de taille n; // pseudo numéro d'attache
2. pourtout sommet s faire
   etat[s] <- inexploré; etat_cfc[s] <- libre;
3. rg <- 0;
4. pile <- NIL;
5. tantqu'il reste un sommet s libre faire
   rg <- dfsCfc(pile, etat, etat_cfc, rang, rg, prat, s);
```

On notera qu'on remplit ces pseudo numéros d'attaches, qui correspondent aux rangs d'attache du graphe initial lors de la localisation de la première composante fortement connexe.

Tout se passe dans la fonction :

```
dfsCfc(pile, etat, etat_cfc, rang, rg, prat, s)
  etat[s] <- encours;
  rang[s] <- rg; rg <- rg+1;
  prat[s] <- rang[s]; // initialisation
  pile <- s # pile; etat_cfc[s] <- empilé;
  pour t voisin de s faire
11.   si etat[t] == inexploré alors
      // t est un descendant de s
      rg <- dfsCfc(pile, etat, etat_cfc, rang, rg, prat, t);
      prat[s] <- min(prat[s], prat[t]);
   sinon
      si etat_cfc[t] == empilé alors
         si rang[t] > rang[s] alors
            // (s, t) arc avant, on ne fait rien
         sinon // (s, t) arriere ou intra-arbre
            si etat[t] == encours alors
               // (s, t) est arriere
               prat[s] <- min(prat[s], rang[t]);
            sinon
               // (s, t) est intra-arbre
               // t est dans C(s) car t ∈ D(s)
               // et t est successeur de s
               prat[s] <- min(prat[s], rang[t]); // [1]
         sinon // t est exclus, on ne fait rien
```

```

si prat[s] == rang[s] alors
    // s est point d'entrée, C(s) est
    // dans la pile jusqu'à s
    dépiler(etat_cfc, pile, s);
    etat[s] <- exploré;
12. retourner rg.

dépiler(etat_cfc, pile, s)
    répéter
        t <- tête(pile);
        etat_cfc[t] <- exclus;
        écrire t;
    jusqu'à ce que t == s;

```

Rappelons que faire `tête(pile)` modifie le premier élément de la pile. L'implantation en Java ne pose pas de problème. Voir la section 7.4.1.

### 7.1.6 L'exemple détaillé

On reprend l'exemple du graphe  $\mathcal{G}_3$ , et on donne le déroulement de l'algorithme en détail, ce qui permet de mieux s'imprégner de son fonctionnement.

```

rang[a] <- 0;
prat[a] <- 0;
pile <- (a);
etat[a] <- encours; etat_cfc[a] <- empilé;
b est voisin de a
    rang[b] <- 1;
    prat[b] <- 1;
    pile = (b, a);
    etat[b] <- encours; etat_cfc[b] <- empilé;
f est voisin de b
    rang[f] <- 2;
    prat[f] <- 2;
    pile = (f, b, a);
    etat[f] <- encours; etat_cfc[f] <- empilé;
e est voisin de f
    rang[e] <- 3;
    prat[e] <- 3;
    pile = (e, f, b, a);
    etat[e] <- encours; etat_cfc[e] <- empilé;
d est voisin de e
    rang[d] <- 4;
    prat[d] <- 4;
pile = (d, e, f, b, a);

```

```

        etat[d] <- encours; etat_cfc[d] <- empilé;
        f est voisin de d, mais il est dans la pile et
            rang[f] < rang[d], donc prat[d] <- 2;
        etat[d] <- exploré;
        prat[e] <- min(3, 2) = 2;
        etat[e] <- exploré;
    prat[f] n'est pas modifié;
    prat[f] == rang[f] donc f est point d'entrée
    on dépile C(f) = {d, e, f};
        etat_cfc[d] <- exclus;
        etat_cfc[e] <- exclus;
        etat_cfc[f] <- exclus;
    etat[f] <- exploré;
    prat[b] n'est pas modifié;
    l est voisin de b
        rang[l] <- 5;
        prat[l] <- 5;
        pile = (l, b, a);
        etat[l] <- encours; etat_cfc[l] <- empilé;
        f est voisin de l, et il n'est plus dans la pile;
        prat[l] == rang[l] donc l est point d'entrée
            on dépile C(l) = {l}; etat[l] <- exclus;
        etat[l] <- exploré;
    prat[b] n'est pas modifié;
    n est voisin de b
        rang[n] <- 6;
        prat[n] <- 6;
        pile = (n, b, a);
        etat[n] <- encours; etat_cfc[n] <- empilé;
        a est voisin de n, qui est empilé
            prat[n] <- min(prat[n], rang[a]) = 0;
        etat[n] <- exploré;
    prat[b] <- min(prat[b], prat[n]) = 0;
    etat[b] <- exploré;
    c est voisin de a
        rang[c] <- 7;
        prat[c] <- 7;
        pile = (c, n, b, a);
        etat[c] <- encours; etat_cfc[c] <- empilé;
        n est voisin de c, mais il est dans la pile et
            rang[n] < rang[c], donc prat[c] <- prat[n] = 0;
        g est voisin de c
            rang[g] <- 8;
            prat[g] <- 8;

```

```

pile = (g, c, n, b, a);
etat[g] <- encours1; etat_cfc[g] <- empilé;
h est voisin de g
  rang[h] <- 9;
  prat[h] <- 9;
  pile = (h, g, c, n, b, a);
  etat[h] <- encours; etat_cfc[h] <- empilé;
  c est voisin de h, mais il est dans la pile et
    rang[c] < rang[h], donc prat[h] <- prat[c] = 0;
  etat[h] <- exploré;
prat[g] <- 0;
m est voisin de g
  rang[m] <- 10;
  prat[m] <- 10;
  pile = (m, h, g, c, n, b, a);
  etat[m] <- encours; etat_cfc[m] <- empilé;
  m est voisin de a, qui est dans la pile et
    rang[a] < rang[m], donc prat[m] <- prat[a] = 0;
  etat[m] <- exploré;
prat[c] n'est pas modifié;
etat[c] <- exploré;
prat[a] n'est pas modifié;
g est voisin de a, mais il est déjà exploré;
prat[a] == rang[a] donc a est point d'entrée
  on dépile C(a) = {m, h, g, c, n, b, a};
  etat_cfc[m] <- exclus; etc.
etat[a] <- exploré;
// deuxième arborescence
rang[i] <- 11;
prat[i] <- 11;
pile = (i);
etat[i] <- encours; etat_cfc[i] <- empilé;
b est voisin de i mais il est exclus;
k est voisin de i
  rang[k] <- 12;
  prat[k] <- 12;
  pile = (k, i);
  etat[k] <- encours; etat_cfc[k] <- empilé;
a est voisin de k mais il est exclus;
j est voisin de k
  rang[j] <- 13;
  prat[j] <- 13;
  pile = (j, k, i);
  etat[j] <- encours; etat_cfc[j] <- empilé;

```

```

    i est voisin de j dans la pile
      prat[j] <- 9;
      etat[j] <- exploré;
    prat[k] <- min(10, 9) = 9;
    etat[k] <- exploré;
  prat[i] n'est pas modifié;
  prat[i] == rang[i] donc i est point d'entrée
  on dépile C(i) = {j, k, i};
  etat_cfc[i] <- exclus; etc.

```

**Exercice 7.1.1** Annoter la trace précédente avec les différents types d'arcs rencontrés.

### 7.1.7 Esquisse de la preuve

Pour prouver que cet algorithme fonctionne correctement, on raisonne par récurrence sur le nombre de composantes fortement connexes.

Qu'est-ce qu'un graphe ayant une seule composante fortement connexe ? C'est par exemple un cercle comme celui de la figure 7.5.

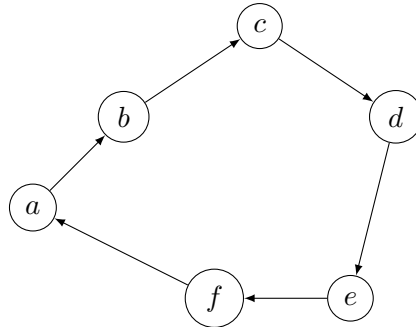


FIG. 7.5 – Un cercle.

On explore suivant l'ordre  $(a, b, c, d, e, f)$  ; quand on arrive sur le sommet  $f$ , on voit que son point d'attache est  $a$ , et en propageant en arrière, c'est le cas de tous les sommets, donc la composante fortement connexe est le cycle tout entier.

**Lemme 7.1.2** *Quand le graphe ne contient qu'une composante, le point d'entrée est le sommet de rang minimum  $r$ .*

*Démonstration.* Si on passe dans [1], alors  $rat(s) < rang(s)$  : ou bien c'était déjà le cas, ou de toute façon  $rang(t) < rang(s)$ . Et donc  $s$  ne sera pas point d'entrée.  $\square$

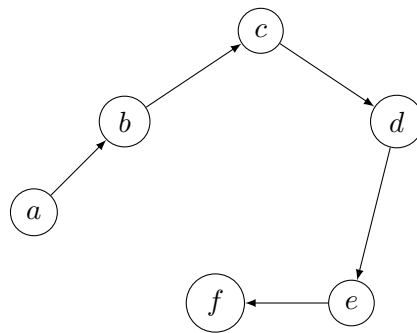
Plus généralement :

**Lemme 7.1.3** *Dans un graphe avec une seule composante fortement connexe, il existe un chemin élémentaire  $(s_1 = r, s_2, \dots, s_k = r)$  tel que  $rat(s_i) = rat(r)$ .*

De façon imagée, on peut voyager de cercles en cercles sans jamais repasser deux fois par le même arc.

*Cas général* : quand on trouve la première composante, on a calculé les rangs d'attache du graphe de départ et on applique le théorème fondamental. On enlève cette composante et il reste à se convaincre que la fonction *prat* a les mêmes propriétés que *rat* elle-même. □

**Exercice 7.1.2** Exécutez l'algorithme sur le graphe :



## 7.2 Connexité dans les graphes non orientés

### 7.2.1 Énumération des composantes connexes

Soit  $\mathcal{G}$  un graphe non orienté. Montrons comment utiliser le parcours en largeur d'abord pour énumérer les composantes connexes du graphe. L'idée est de partir d'un sommet inexploré, puis de chercher tous ses voisins, et de proche en proche tous les sommets qui sont liés. Pour arriver au même but, un parcours en profondeur d'abord conviendrait également.

On peut par exemple afficher tous les membres d'une même composante. Le programme s'écrira :

```

composantesConnexes(G)
0. pourtout sommet t faire
    etat[t] <- inexploré;
    nc <- 0;
1. pour s dans S
    si etat[s] = inexploré alors
        nc <- nc+1; // numéro de composante
        afficher("Composante " + nc);
        bfs(G, s);

bfs(G, s)
1. F <- (s);
  
```

```

2. tantque F n'est pas vide faire
  t <- tête(F);
  etat[t] <- exploré;
  afficher(t);
  pour u voisin de t faire
    si etat[u] = inexploré alors
      etat[u] <- encours;
      F <- F # u;

```

Il serait facile de stocker les sommets appartenant à la même composante connexe.

### 7.2.2 Points d'articulation

#### Théorie

On se donne un graphe connexe orienté  $\mathcal{G}$ . Dans certains problèmes, n'avoir que la connexité ne suffit pas.

**Définition 7.2.1** *Un point d'articulation est un sommet dont la disparition supprime la connexité de  $\mathcal{G}$ .*

**Définition 7.2.2** *Un graphe qui ne possède pas de point d'articulation est dit biconnexe.*

**Définition 7.2.3** *Un sous-graphe maximal biconnexe est appelé bloc.*

On peut vouloir savoir s'il existe un sommet qui permet de séparer le graphe en deux (ou plusieurs) morceaux connexes. Sur une carte routière, un tel point sera névralgique pour le réseau et on devra donc surveiller qu'aucun accident ne le bloque. On verra à la section 7.3 que c'est la première opération à effectuer sur un graphe dont on veut tester la planarité, ce qui permettra de tester la planarité bloc par bloc.

On trouve à la figure 7.6 l'exemple du graphe  $\mathcal{G}_5$  et ses points d'articulation indiqués par des cercles doubles. Les blocs sont également indiqués à droite du graphe.

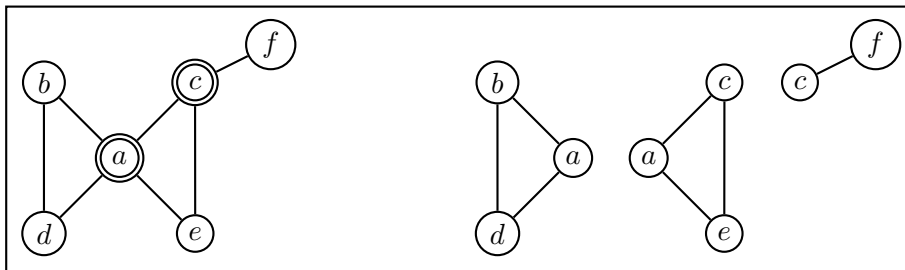


FIG. 7.6 – Le graphe  $\mathcal{G}_5$ , ses points d'articulation et les blocs.

Une solution brutale consiste à enlever chaque sommet  $s$  à tour de rôle et à faire une bfs (ou dfs) pour tester la connexité du graphe restant. En déduire les blocs est un sous-produit de l'algorithme.

Quand le graphe est gros, c'est inefficace, et nous allons montrer qu'un algorithme très proche de celui cherchant les composantes fortement connexes permet de résoudre le problème avec une bonne complexité.

### L'algorithme

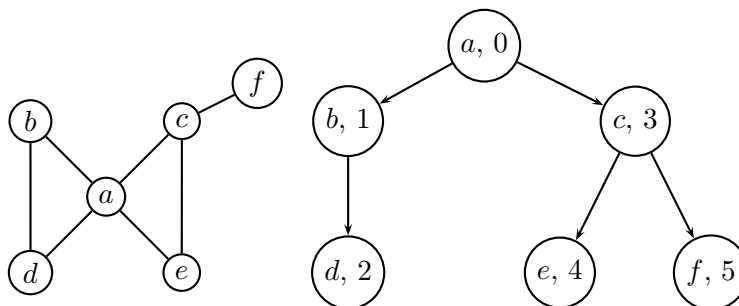
Le sommet  $v$  est un point d'articulation si et seulement s'il existe deux sommets  $x$  et  $y$  tel que tout chemin de  $x$  à  $y$  passe par  $v$ . Dans ce cas là et seulement dans ce cas là, la disparition de  $v$  déconnecte  $\mathcal{G}$ .

On va là encore utiliser le rang d'attache d'un sommet, comme déjà introduit dans le cas orienté.

Nous allons nous servir de l'exemple du graphe  $\mathcal{G}_5$  pour illustrer notre propos. Conformément à l'ordre suivant pour les voisins des sommets :

a: (b, d, c, e)  
 b: (a, d)  
 c: (a, e, f)  
 d: (a, b)  
 e: (a, c)  
 f: (c)

on trouve :



**Proposition 7.2.1** Soit  $T$  une arborescence de Trémaux associée à  $\mathcal{G}$ . Le sommet  $u$  est un point d'articulation si et seulement s'il existe deux sommets  $v$  et  $w$  tels que  $(u, v) \in T$ ,  $w \in T$ ,  $w \neq u$ ,  $w$  n'est pas un descendant de  $v$  dans  $T$ , et  $\text{rang}(at(v)) \geq \text{rang}(u)$ .

**Exemple.** (suite) On voit que dans notre exemple,  $a$  est point d'articulation (en utilisant  $v = b$ ,  $w = c$ ) ;  $c$  l'est également ( $v = e$ ,  $w = f$ ).

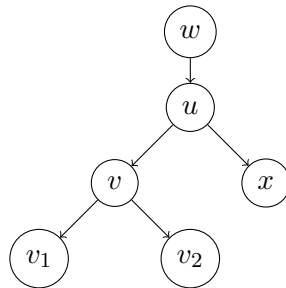
*Démonstration.* Supposons dans un premier temps que  $v$  et  $w$  existent. Comme  $(u, v) \in T$  et  $\text{rang}(at(v)) \geq \text{rang}(u)$ , tout chemin d'origine  $v$  ne passant par  $u$  doit rester dans la sous-arborescence de racine  $v$ . Comme  $w$  n'est pas descendant de  $v$  dans  $T$ , un tel

chemin ne peut contenir  $w$ . Les seuls chemins éventuels entre  $v$  et  $w$  doivent passer par  $u$ , donc  $u$  est point d'articulation.

Supposons maintenant que  $u$  est point d'articulation. Si  $u$  est la racine de  $T$ , il y a au moins deux arêtes qui partent de  $u$ ; s'il n'y en avait qu'une, alors il y aurait un chemin entre chaque paire de sommets de  $\mathcal{S} - \{u\}$  ne contenant pas  $u$ . Soient  $(u, v)$  et  $(u, w)$  ces deux arêtes;  $v$  et  $w$  conviennent.

Dans notre exemple, c'est le cas de  $a$  et des deux arêtes  $(a, b)$  et  $(a, c)$  dans  $T$ .

Si  $u$  n'est pas la racine de  $T$ , il a un ancêtre  $w$ . L'une des composantes biconnexes contenant  $u$  a tous ses sommets parmi les descendants de  $u$  dans  $T$ . Ces sommets sont tous des descendants d'un sommet  $v$ , avec  $(u, v)$  dans  $T$ . Ce seront les  $v$  et  $w$  demandés. Pourquoi est-ce vrai? Tout d'abord,  $u$  a forcément un descendant, sinon ce ne serait pas un point d'articulation. Si la propriété était fausse, on aurait une situation comme :



Si  $x$  et  $v$  étaient dans la même composante bi-connexe, alors il existerait un chemin entre  $x$  et  $v$  ne passant par  $u$ , et donc  $x$  serait un descendant de  $v$  dans  $T$ . Contradiction.  $\square$

**Exemple.** (suite) Dans notre exemple, pour le sommet  $c$ , on peut prendre  $w = a$  et  $v = f$ .

L'algorithme que nous allons décrire ressemble beaucoup à celui qui cherche les composantes fortement connexes. Une différence importante est liée au traitement de la racine de l'arborescence, car nous devons compter le nombre de fils de la racine dans celle-ci et que donc on ne saura que la racine est point d'articulation que quand on aura terminé l'exploration.

```

pointsDArticulation(G)
1. etat <- tableau de taille n;
   rang <- tableau de taille n;
   rat <- tableau de taille n; // numéro d'attache
2. pour tout sommet s faire etat[s] <- inexploré;
3. rg <- 0;
4. tantqu'il reste un sommet s inexploré faire
   // exploration composante connexe par composante connexe
   rg <- dfsPDARacine(etat, rang, rg, rat, s);
  
```

Le code traitant la racine est relativement simple :

```

dfsPDARacine(etat, rang, rg, rat, s)
  etat[s] <- encours;
  rang[s] <- rg; rg <- rg+1;
  rat[s] <- rang[s]; // initialisation
  nfils <- 0;
  pour t voisin de s faire
11.   si etat[t] == inexploré alors
        // t est un descendant de s
        rg <- dfsPda(etat, rang, rg, rat, s, t);
        nfils++;
    si nfils > 1 alors
        écrire s; // point d'articulation
    etat[s] <- exploré;
12. retourner rg.

```

Le code pour un sommet non racine utilise le père du sommet courant. Un arc  $(s, p)$  avec  $p$  père de  $s$  n'est pas un arc intéressant dans le calcul du rang d'attache.

```

dfsPda(etat, rang, rg, rat, pere, s)
  // pere est le père de s dans l'arborescence
  etat[s] <- encours;
  rang[s] <- rg; rg <- rg+1;
  rat[s] <- rang[s]; // initialisation
  pour t voisin de s faire
11.   si etat[t] == inexploré alors
        // t est un descendant de s
        rg <- dfsPda(etat, rang, rg, rat, s, t);
        rat[s] <- min(rat[s], rat[t]);
        si rat[t] >= rang[s] alors
            écrire s; // point d'articulation
        sinon si (rang[t] < rang[s]) et (t ≠ pere) alors
            // (s, t) est arrière
            rat[s] <- min(rat[s], rang[t]);
    etat[s] <- exploré;
12. retourner rg.

```

La recherche des blocs est une chose un peu plus délicate, car il faut définir la notion correcte de bloc (avec partage des sommets points d'articulation), et trouver une structure de données adéquate. Une variante particulièrement astucieuse du programme que nous venons de donner se trouve dans [7] et permet de trouver ces blocs.

**Exercice 7.2.1** Exécuter l'algorithme précédent sur l'exemple de la figure 7.6.

## 7.3 Planarité

Tester la planarité d'un graphe est important dans plusieurs applications, comme le dessin de circuits imprimés, la réalisation de câblages. Nous allons donner quelques idées sur la façon dont on peut caractériser les graphes planaires et esquisser un algorithme de test. L'algorithme linéaire de Tarjan dépasse le cadre de ce cours et nous renvoyons à la littérature pour sa description. De nombreux résultats seront cités sans démonstration.

Dans cette section,  $\mathcal{G}$  sera un graphe *simple* (c'est-à-dire sans arête multiple et sans boucle), et connexe.

**Définition 7.3.1** *Un graphe est planaire si on peut le dessiner sans que les arcs se croisent.*

Même sans formaliser outrancièrement, il faut être conscient que cette définition peut cacher beaucoup de choses, surtout si on veut la rendre très formelle (elle est en fait très topologique). Disons pour simplifier qu'on peut réaliser un dessin de  $\mathcal{G}$  en associant aux sommets du graphe des points du plan et représenter les arêtes par des segments. Il est possible de montrer que tout graphe planaire peut être dessiné ainsi, de façon que les arêtes ne se croisent pas.

Les graphes  $\mathcal{G}_5$  et  $K_4$  de la figure 7.7 sont planaires (exercice),  $K_{3,3}$  ne l'est pas. Dans ce domaine là également, il ne saurait y avoir de dessin canonique d'un graphe planaire, comme le montre l'exemple de  $K_4$ .

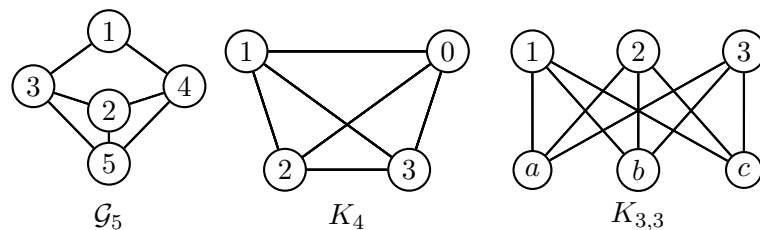


FIG. 7.7 – Des exemples de graphes planaires ou non.

### 7.3.1 Faces

Nous appellerons *graphe plan*  $\mathbf{G}$  un graphe planaire équipé d'un dessin dans le plan, où aucune arête n'en coupe une autre. Les arêtes de  $\mathbf{G}$  découpent l'espace en *régions* ou *faces*; il existe une face spéciale, la *face infinie*.

**Définition 7.3.2** *Le degré d'une face est le nombre d'arêtes rencontrées dans un chemin qui suit la frontière de  $f$ .*

**Exemple.** Le graphe de la figure 7.8 découpe l'espace en 4 faces, dont voici les degrés :  $\deg(f_1) = 3$ ,  $\deg(f_2) = 9$ ,  $\deg(f_3) = 4$ ,  $\deg(f_4) = 6$ .

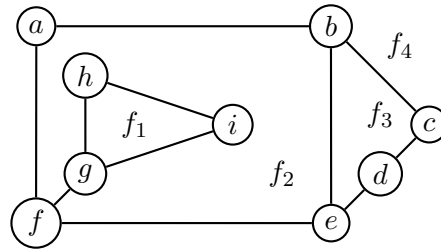


FIG. 7.8 – Un graphe planaire et les faces associées.

**Exercice 7.3.1** Montrer que si  $\mathbf{G}$  est un arbre à  $n$  sommets, alors il n'y a qu'une face de degré  $2n - 2$ .

### 7.3.2 Graphe dual

**Définition 7.3.3** Le graphe dual  $\mathbf{G}^*$  d'un graphe plan  $\mathbf{G}$  est obtenu à partir de  $\mathbf{G}$  en créant pour toute face de  $\mathbf{G}$ , un sommet de  $\mathbf{G}^*$ ; si deux faces sont contiguës, on crée une arête entre les deux sommets de  $\mathbf{G}^*$ . Par construction, le graphe  $\mathbf{G}^*$  est un graphe plan.

On donne à la figure 7.9 deux exemples de graphe avec son dual.

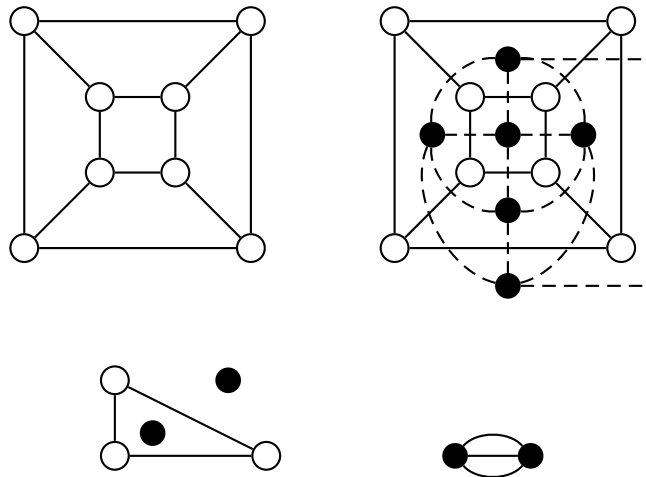


FIG. 7.9 – Graphes et leurs duals.

Le résultat suivant permet de relier les paramètres du graphe plan et de son dual.

**Proposition 7.3.1** Si  $\mathbf{G}$  a  $n$  sommets,  $m$  arêtes,  $f$  faces, alors  $\mathbf{G}^*$  a  $f$  sommets,  $m$  arêtes,  $n$  faces.

**Définition 7.3.4** On dit que deux graphes  $G$  et  $H$  sont isomorphes si et seulement si on passe de  $G$  à  $H$  par renumérotation des sommets.

**Théorème 7.3.1** Le dual  $(\mathbf{G}^*)^*$  est isomorphe à  $\mathbf{G}$ .

**Proposition 7.3.2** Si  $\mathcal{G}$  est planaire, alors pour tout dessin planaire de  $\mathcal{G}$ , on a  $\sum_f \deg(f) = 2m$ .

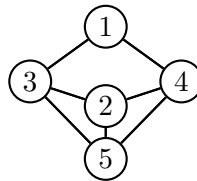
*Démonstration.* on applique le hand-shaking lemma (Lemme 4.2.1) au graphe dual.  $\square$

### 7.3.3 Relation d'Euler

**Théorème 7.3.2** On note  $f$  le nombre de régions de  $\mathbf{G}$ . Alors

$$f = m - n + 2.$$

**Exemple.** Pour le graphe



on trouve  $n = 5$ ,  $m = 7$ ,  $f = 4$  et on a bien  $4 = 7 - 5 + 2$ .

**Exercice 7.3.2** Montrer que le résultat est vrai pour les arbres.

*Démonstration.* On va raisonner par récurrence sur le nombre d'arêtes  $m$  :

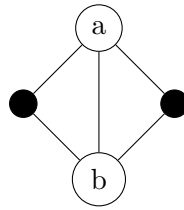
$H_m$  : la relation d'Euler est vraie pour tous les graphes plans avec au plus  $m$  arêtes.

Les propriétés  $H_2$  et  $H_3$  sont vraies comme on peut s'en convaincre avec les dessins :



Intéressons-nous à  $H_m$  pour  $m > 3$ . On part donc d'un graphe plan  $\mathbf{G}$  avec  $m + 1$  arêtes. Nous allons enlever une arête pour nous ramener au cas de  $H_m$ .

Supposons d'abord que l'arête enlevée appartienne à deux faces distinctes, comme l'arête  $(a, b)$  de la figure suivante :



Quand on enlève cette arête, le graphe résultant a  $m - 1$  arêtes, reste connexe, a le même nombre de sommets, mais le nombre de face a diminué de 1 par fusion des deux faces. On écrit alors :

$$f - m + n = (f - 1) - (m - 1) + n = 2$$

en utilisant  $H_{m-1}$ .

Il y a un deuxième cas, celui où l'arête enlevée appartenait à la même face. Commençons par le cas facile :

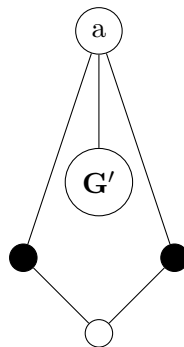


En coupant l'arête, le graphe perd sa connexité, mais les deux morceaux restent connexes et planaires de paramètres  $f_i$ ,  $m_i$  et  $n_i$  vérifiant :  $f = f_1 + f_2 - 1$  (il ne faut pas compter deux fois la face infinie),  $n = n_1 + n_2$ ,  $m = m_1 + m_2 + 1$ , ce qui fait que

$$f - m + n = (f_1 - m_1 + n_1) + (f_2 - m_2 + n_2) - 2 = 2,$$

par application de  $H_{m_i}$ .

Le même raisonnement marche pour le cas plus élaboré :



où  $\mathbf{G}'$  est un sous-graphe de  $\mathbf{G}$ .  $\square$

Donnons quelques applications de la formule d'Euler.

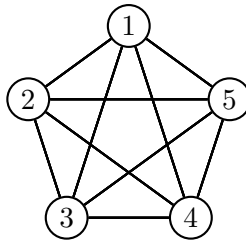
**Corollaire 7.3.1** *Si  $\mathbf{G}$  est un graphe plan avec  $n \geq 3$ , alors  $m \leq 3n - 6$ .*

*Démonstration.* Le degré de chaque face est au moins 3, d'où  $3f \leq 2m$  et on utilise Euler.  $\square$

Ce résultat est très important. Il montre en particulier qu'un graphe planaire ne peut pas avoir trop d'arêtes, et donc qu'en particulier  $|\mathcal{A}| = O(|\mathcal{S}|)$ , ce qui montre que le paramètre pertinent est  $n$  plutôt que  $m$  et laisse la place pour des algorithmes linéaires en le nombre de sommets. En outre, cela permet de faire des raisonnements évitant d'avoir à introduire le nombre de faces, donc de faire des dessins qui peuvent se révéler faux.

**Corollaire 7.3.2** *Le graphe complet  $K_5$  n'est pas planaire.*

*Démonstration.*



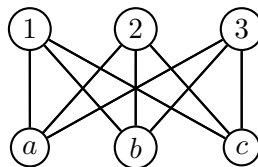
On trouve que  $n = 5$ ,  $m = 10$ , mais  $10 > 3 \times 5 - 6$ .  $\square$

**Corollaire 7.3.3** *Si  $G$  est un graphe plan avec  $n \geq 3$  et n'a pas de triangle, alors  $m \leq 2n - 4$ .*

*Démonstration.* chaque face a degré au moins 4, donc  $4f \leq 2m$  d'où le résultat.  $\square$

**Corollaire 7.3.4** *Le graphe  $K_{3,3}$  n'est pas planaire.*

*Démonstration.*



On calcule  $n = 6$ ,  $m = 9$ ; le graphe n'a pas de triangle mais pourtant :  $9 > (2 \times 6) - 4$ .  $\square$

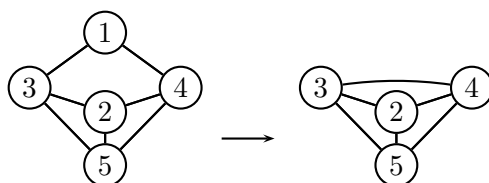


FIG. 7.10 – Deux graphes homéomorphes.

### 7.3.4 Un critère théorique de planarité

**Définition 7.3.5** Deux graphes sont homéomorphes ssi on peut passer de l'un à l'autre par fusion ou scission d'arcs passant par un nœud de degré 2.

**Exemple.** On part du graphe (a) de la figure 7.10 et supprime le sommet 1. On obtient le graphe (b) qui lui est homéomorphe.

L'un des théorèmes principaux du domaine est le suivant.

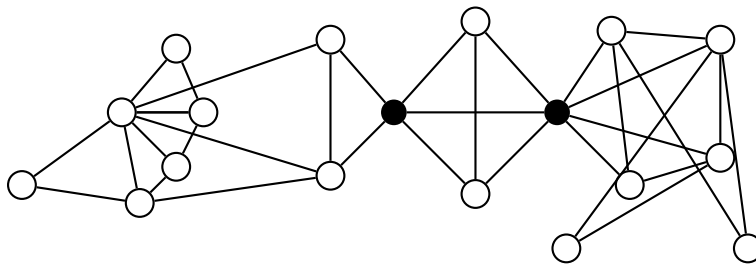
**Théorème 7.3.3 (Kuratowski, 1930)** Un graphe  $\mathcal{G}$  est planaire si et seulement s'il ne contient aucun sous-graphe homéomorphe à  $K_5$  ou  $K_{3,3}$ .

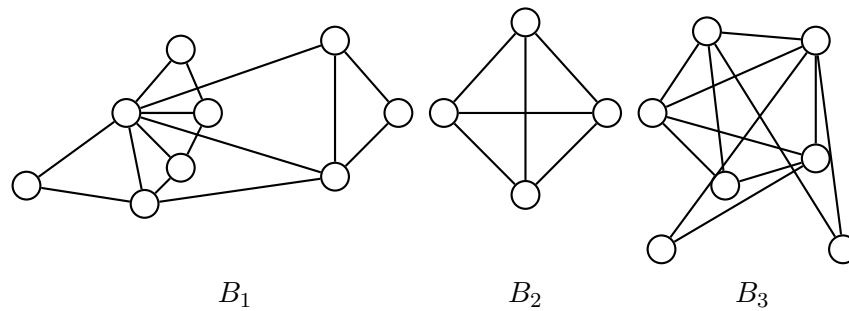
*Démonstration.* Le sens difficile est le sens direct. Le sens facile est la réciproque. En effet, on vient de montrer que ni  $K_5$ , ni  $K_{3,3}$  ne sont planaires. Un graphe contenant un sous-graphe homéomorphe à l'un de ses graphes ne peut être planaire.  $\square$

### 7.3.5 Vers un algorithme de test de planarité

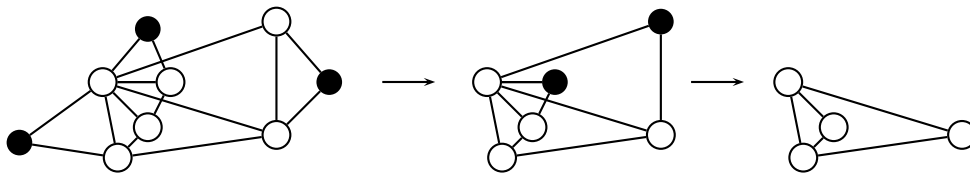
Expliquons sur un exemple la manière de procéder pour se ramener à des cas plus simples. La première chose à faire pour déterminer si un graphe est planaire est d'abord de travailler composante connexe par composante connexe. Ensuite, on remarque qu'un graphe connexe est planaire si et seulement si chacun de ses blocs l'est. On les calcule donc par l'algorithme de la section 7.2.2.

Considérons l'exemple du graphe suivant, que l'on coupe en trois blocs  $B_1$ ,  $B_2$ ,  $B_3$  à partir des points d'articulation :

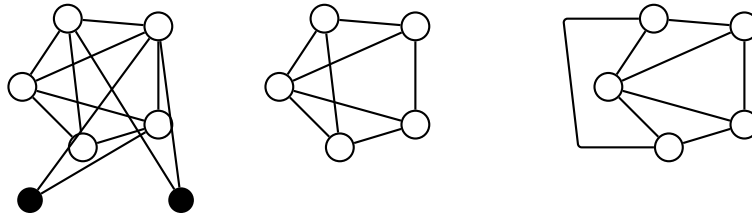




Le graphe  $B_2$  est planaire par inspection (ou en remarquant que tout graphe avec  $n < 5$  est planaire). Pour  $B_1$ , on agit par homéomorphisme, en enlevant les sommets de degré 2 (les sommets en noir) tour à tour comme indiqué sur le dessin.



Il reste à traiter  $B_3$ , pour lequel on procède par homéomorphisme, puis inspection :



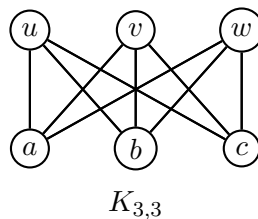
On peut aussi montrer que  $m < 9$  implique  $\mathcal{G}$  planaire.

### 7.3.6 Une idée d'algorithme

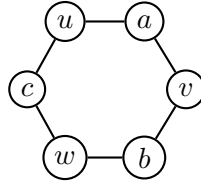
On suppose dans un premier temps que le graphe  $\mathcal{G}$  est hamiltonien : ou bien c'est un arbre (donc planaire), ou bien ses sommets peuvent être mis sur un polygone fermé (un cercle).

L'idée est de répartir les arêtes à l'extérieur ou à l'intérieur du cycle. Si on y arrive sans que les arêtes ne se coupent,  $\mathcal{G}$  est planaire, sinon, il ne l'est pas.

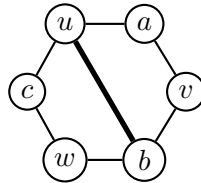
De nouveau, considérons l'exemple de  $K_{3,3}$  et appliquons cette démarche.



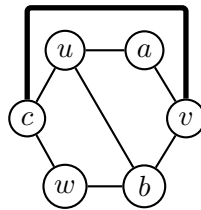
On sélectionne le circuit hamiltonien **uavbwc**. S'il existe un dessin planaire, alors ce circuit formera nécessairement un polygone :



Les arêtes non encore utilisées sont **ub**, **vc**, **wa**. On peut dessiner **ub** sans problème :

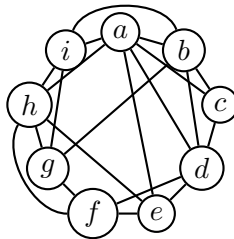


On peut dessiner l'arête **vc** en passant à l'extérieur du polygone :

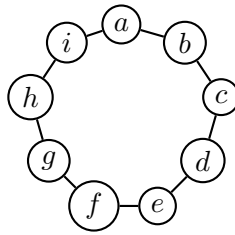


Il ne nous reste plus que l'arête **wa** à placer, mais on ne peut le faire sans couper les arêtes déjà présentes. On peut montrer que la conclusion reste valide en permutant ces trois arêtes entre elles, donc  $K_{3,3}$  n'est pas planaire.

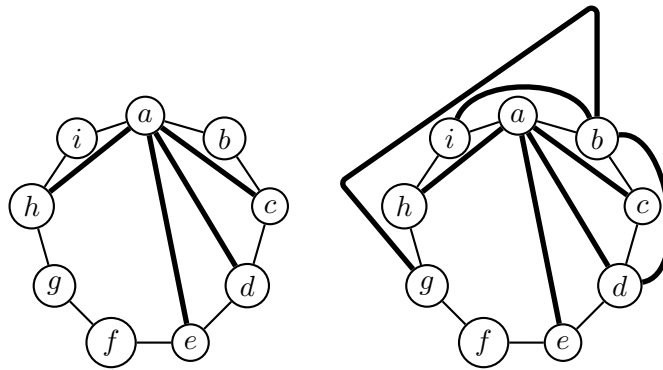
Donnons un deuxième exemple.



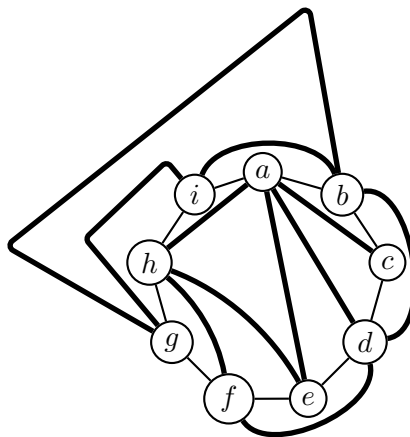
On choisit le cercle :



Il reste à placer les arêtes **ac, ad, ae, ah, bd, bg, bi, df, eh, fh, gi**. On place d'abord **ac, ad, ae, ah** à l'intérieur, puis **bd, bg, bi** à l'extérieur, conformément au dessin :



Finalement, on place **df, eh, fh, gi** :



et le graphe est donc bien planaire.

Il est clair que nous n'avons fait qu'esquisser l'algorithme. Nous renvoyons à l'algorithme de Tarjan qui fonctionne en temps linéaire en  $O(|S|)$ .

## 7.4 Les programmes en Java

### 7.4.1 Composantes fortement connexes

Nous ajoutons de nouvelles valeurs pour les marques.

```

final static int libre = 0, empile = 1, exclus = 2;

int dfsCfc(Hashtable<Sommet,Integer> etat,
           Hashtable<Sommet,Integer> etat_cfc,
           LinkedList<Sommet> pile,
           Hashtable<Sommet,Integer> rang, int rg,
           Hashtable<Sommet,Integer> prat, Sommet s){
    etat.put(s, encours);
    rang.put(s, rg++);
    int prats = rang.get(s);
    pile.addFirst(s);
    etat_cfc.put(s, empile);
    for(Arc<Sommet> a : voisins(s)){
        Sommet t = a.destination();
        if(etat.get(t) == inexplore){
            rg=dfsCfc(etat,etat_cfc,pile,rang,rg,prat,t);
            prats = Math.min(prats, prat.get(t));
        }
        else{
            if(etat_cfc.get(t) == empile){
                if(rang.get(t) <= rang.get(s)){
                    // (s, t) est arriere ou intra-arbre
                    // t est dans C(s) car t descendant
                    // de s et s -> t
                    if(rang.get(t) < prat.get(s))
                        prats=Math.min(prats, rang.get(t));
                }
            }
        }
    }
    prat.put(s, prats);
    if(prats == rang.get(s)){
        // s est un point d'entr ee
        Sommet t;
        System.out.print("CFC={");
        do{
            t = pile.removeFirst();
            etat_cfc.put(t, exclus);
            System.out.print(" "+t);

```

```

        } while(! t.equals(s));
        System.out.println(" }");
    }
    return rg;
}

void cfc(){
    Hashtable<Sommet,Integer> etat =
        new Hashtable<Sommet,Integer>();
    Hashtable<Sommet,Integer> etat_cfc =
        new Hashtable<Sommet,Integer>();
    Hashtable<Sommet,Integer> rang =
        new Hashtable<Sommet,Integer>();
    Hashtable<Sommet,Integer> prat =
        new Hashtable<Sommet,Integer>();

    for(Sommet s : sommets()){
        etat.put(s, inexplorer);
        etat_cfc.put(s, libre);
    }
    int rg = 0;
    LinkedList<Sommet> pile = new LinkedList<Sommet>();
    for(Sommet s : sommets())
        if(etat.get(s) == inexplorer)
            rg=dfsCfc(etat,etat_cfc,pile,rang,rg,prat,s);
}

```

#### 7.4.2 Composantes connexes

Ce qui peut donner en Java

```

public void composantesConnexes(){
    for(Sommet s : sommets())
        etat.put(s, inexplorer);
    int ncc = 0;
    for(Sommet s : sommets())
        if(etat.get(s) == inexplorer){
            ncc++;
            System.out.println("Composante "+ncc);
            bfs(etat, s);
        }
}

public void bfs(Hashtable<Sommet,Integer> etat, Sommet s){
    LinkedList<Sommet> f = new LinkedList<Sommet>();
}

```

```

etat.put(s, encours);
f.addLast(s);
while(! f.isEmpty()){
    Sommet t = f.removeFirst();
    System.out.println("J'explore "+t);
    for(Arc<Sommet> a : voisins(t)){
        Sommet u = a.destination();
        if(etat.get(u) == inexplorer){
            etat.put(u, encours);
            f.addLast(u);
        }
    }
    etat.put(t, explore);
}
}

```

### 7.4.3 Points d'articulation

```

// pere -> s
int dfsPda(Hashtable<Sommet,Integer> etat,
    LinkedList<Arc<Sommet>> pile,
    Hashtable<Sommet,Integer> rang, int rg,
    Hashtable<Sommet,Integer> rat,
    Sommet pere, Sommet s){
    etat.put(s, encours);
    System.out.println("J'explore "+s);
    rang.put(s, rg++);
    int rats = rang.get(s);
    for(Arc<Sommet> a : voisins(s)){
        Sommet t = a.destination();
        if(etat.get(t) == inexplorer){
            pile.addFirst(a);
            rg = dfsPda(etat, pile, rang, rg, rat, s, t);
            rats = Math.min(rats, rat.get(t));
            if(rat.get(t) >= rang.get(s)){
                System.out.print(s+" est point");
                System.out.println(" d'articulation");
            }
        }
        else if((rang.get(t) < rang.get(s))
            && !t.equals(pere)){
            System.out.print("(" +s+" , "+t+" ) arrière");
        }
    }
}

```

```

        System.out.println(" ou transverse");
        rats = Math.min(rats, rang.get(t));
    }
}
rat.put(s, rats);
etat.put(s, explore);
return rg;
}

// s est la racine de l'arborescence
int dfsPDARacine(Hashtable<Sommet,Integer> etat,
                LinkedList<Arc<Sommet>> pile,
                Hashtable<Sommet,Integer> rang, int rg,
                Hashtable<Sommet,Integer> rat, Sommet s){
    etat.put(s, encours);
    rang.put(s, rg++);
    int rats = rang.get(s);
    int nfiles = 0;
    for(Arc<Sommet> a : voisins(s)){
        Sommet t = a.destination();
        if(etat.get(t) == inexplorer){
            pile.addFirst(a);
            rg = dfsPda(etat, pile, rang, rg, rat, s, t);
            nfiles++;
        }
    }
    if(nfiles > 1){
        System.out.print(s+" est racine");
        System.out.println(" ET point d'articulation");
        depiler(pile);
    }
    rat.put(s, rats);
    System.out.print("rang["+s+"]="+rang.get(s));
    System.out.println(" rat["+s+"]="+rat.get(s));
    etat.put(s, explore);
    return rg;
}

void pointsDArticulation(){
    Hashtable<Sommet,Integer> etat =
        new Hashtable<Sommet,Integer>();
    Hashtable<Sommet,Integer> rang =
        new Hashtable<Sommet,Integer>();
    Hashtable<Sommet,Integer> rat =

```

```
        new Hashtable<Sommet,Integer>();

for(Sommet s : sommets())
    etat.put(s, inexplorer);
LinkedList<Arc<Sommet>> pile =
    new LinkedList<Arc<Sommet>>();
int rg = 0;
for(Sommet s : sommets())
    if(etat.get(s) == inexplorer)
        // nouvelle composante connexe
        rg=dfsPDARacine(etat, pile, rang, rg, rat, s);
}
```

## Chapitre 8

# Euler et Hamilton

Dans ce chapitre, nous introduisons deux concepts utilisés couramment, ceux de graphe eulérien et graphe hamiltonien, notions qui prédatent l'utilisation des graphes en informatique, mais qui apparaissent comme les actes fondateurs de la théorie.

Les graphes considérés seront *a priori* non orientés et connexes, même si les notions s'étendent facilement au cas non orienté.

### 8.1 Euler : fondateur de la théorie des graphes

Le problème considéré à l'époque est celui des ponts de Koenigsberg, un plan d'une ville dont on se demandait s'il était possible de parcourir tous les ponts une fois et une seule au cours d'une promenade :



FIG. 8.1 – Ponts de Königsberg (Kaliningrad)

**Définition 8.1.1** *On appelle cycle Eulérien un cycle qui passe par toutes les arêtes du graphe une fois et seule (mais peut-être plusieurs fois par le même sommet). Un graphe eulérien est un graphe possédant un cycle eulérien.*

**Ex.** Pour le graphe de la figure 8.2, **abcdefbgcegfa** ou **afgcedegbcefba** sont des cycles eulériens.

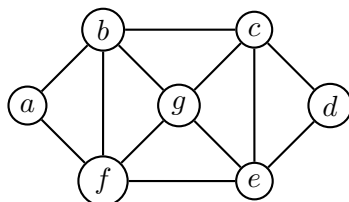


FIG. 8.2 – Le graphe  $\mathcal{G}$ .

Nous pouvons maintenant énoncer le résultat principal.

**Théorème 8.1.1 (Euler, 1736)** *Un graphe  $\mathcal{G}$  possède un cycle Eulérien si et seulement si chaque sommet est de degré pair.*

**Corollaire 8.1.1** *Il n'existe pas de chemin passant une fois et une seule par les 7 ponts de Königsberg.*

Nous commençons par le résultat suivant :

**Proposition 8.1.1** *Si tous les sommets sont de degré pair, alors  $\mathcal{G}$  peut être découpé en cycles n'ayant pas d'arête en commun.*

*Démonstration.* On part d'un sommet arbitraire  $u$ , et quand on arrive à un sommet, on sait qu'on peut ressortir par une autre arête. Comme  $\mathcal{G}$  est fini, on finit par rencontrer un sommet  $v$  qu'on a déjà rencontré, ce qui forme le cycle  $C_1$ . Le graphe  $H = \mathcal{G} - C_1$  possède la même propriété (composante connexe par composante connexe), etc.  $\square$

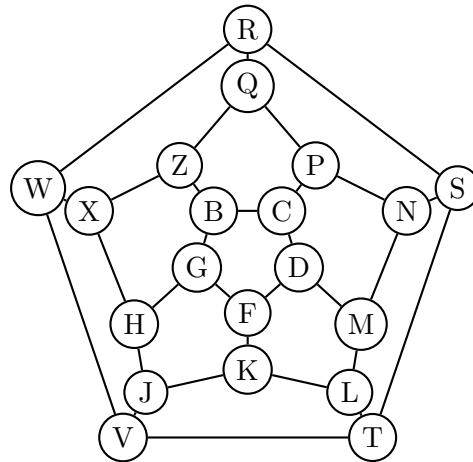
*Démonstration du théorème d'Euler.* (a) Si  $\mathcal{G}$  est eulérien, alors tout sommet est de degré pair. En effet, chaque sommet visité par le chemin apporte une contribution de 2 au degré du sommet.

(b) Montrons que si tout sommet est de degré pair, alors  $\mathcal{G}$  est eulérien. D'après la proposition, on peut découper  $\mathcal{G}$  en cycles disjoints  $C_1, C_2, \dots, C_k$ . On parcourt  $C_1$  jusqu'à trouver un sommet de  $C_2$ , que l'on parcourt. Si on n'a pas fini, on recommence le parcours de  $C_1 \cup C_2$  jusqu'à trouver un sommet d'un cycle  $C_3$ , etc.  $\square$

## 8.2 Hamilton (1805–1865)

**Définition 8.2.1** *Un cycle Hamiltonien est un cycle qui passe par tous les sommets une fois et seule. Un graphe hamiltonien est un graphe possédant un cycle hamiltonien.*

**Ex.**  $abcdegfa$  ou  $afedcgba$  sont des cycles hamiltoniens pour le graphe de la figure 8.2. Pour le graphe ci-dessous, on trouve que  $BCPNMDFKLT SRQZXWVJHGB$  et  $BCPNMDFGHXWVJKLTSRQZB$  sont deux cycles hamiltoniens.



Contrairement au cas eulérien, il n'y a pas de critère simple d'existence de cycle hamiltonien. On dispose de quelques résultats théoriques.

**Proposition 8.2.1** *Le graphe  $K_n$  est hamiltonien, puisque tous les sommets sont reliés entre eux.*

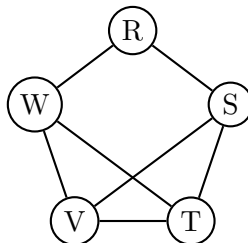
Intuitivement, il sera plus facile de trouver un chemin Hamiltonien si le graphe est dense, ce que tend à souligner le résultat qui suit.

**Théorème 8.2.1 (Ore, 1960)** *Si  $n \geq 3$ , et*

$$\deg(v) + \deg(w) \geq n$$

*pour tout  $(v, w) \notin \mathcal{A}$ , alors  $\mathcal{G}$  est Hamiltonien.*

**Ex.** Ce résultat permet de prouver que le graphe suivant est hamiltonien.



**Exercices complémentaires**

1. (Le graphe de de Bruijn) Le problème est le suivant : est-il possible de construire une suite circulaire de bits telle que tous les nombres de  $n$  chiffres binaires apparaissent par simple décalage d'un bit ?

a) On considère le cas  $n = 3$ . On construit le graphe dont les sommets sont les suites de 2 bits. Les arcs sortant du sommet  $s$  sont étiquetés par  $sb$  avec  $b \in \{0, 1\}$ . De même, les arcs entrant sont du type  $bs$  avec  $b \in \{0, 1\}$ . Dessiner le graphe correspondant.

Montrer qu'un circuit eulérien dans ce graphe donne la réponse, en sélectionnant la première lettre de chaque étiquette dans l'ordre du circuit, par exemple la suite circulaire 00010111.

b) Généraliser à  $n$  quelconque.

c) Généraliser à  $n$  chiffres sur un alphabet à  $m$  lettres.

## Chapitre 9

# Introduction à l'optimisation combinatoire

Les graphes permettent de modéliser de nombreux problèmes de nature combinatoire, comme trouver des chemins de coût minimaux, ou bien encore jouer à des jeux comme les échecs.

### 9.1 L'algorithme de Dijkstra

L'algorithme de Floyd permet de calculer les distances minimales entre tous les sommets avec une complexité en  $O(n^3)$ . Dans certains cas, on a besoin d'une information partielle, à savoir, la distance minimale entre deux sommets  $s$  et  $s'$ . Il se trouve qu'il est tout aussi facile de trouver les distances entre  $s$  et tous les autres sommets. C'est une approche dite *gloutonne*.

#### 9.1.1 Théorie

On se donne un graphe valué avec

$$d(u, v) = \begin{cases} 0 & \text{si } u = v; \\ d(u, v) > 0 & \text{si } u \neq v; \\ +\infty & \text{si } (u, v) \notin \mathcal{A}. \end{cases}$$

Les principes de l'algorithme sont les suivants :

- L'algorithme construit un ensemble  $T \subset S$  tel que pour tout  $t \in T$ , tout chemin minimal de  $s$  à  $t$  ne passe que par des éléments de  $T$ . L'ensemble  $T$  grossit incrémentalement de  $\emptyset$  à  $S$ .
- À chaque sommet  $t \in S$ , on associe une étiquette  $L(t)$ , initialisée à  $d(s, t)$ ; à la fin de l'algorithme, cette valeur contient la valeur du plus petit chemin de  $s$  à  $t$ .

Donnons l'algorithme en pseudocode :

```
1. [Initialisation] pour tout  $t \neq s$  faire  $L(t) \leftarrow d(s, t)$ ;  
2.  $L(s) \leftarrow 0$ ;  $T \leftarrow \{s\}$ ;
```

```

3. tantque  $T \neq S$ 
4.   trouver  $v \notin T$  telque  $L(v) = \min \{L(t), t \notin T\}$ ;
5.    $T \leftarrow T \cup \{v\}$ ;
6.   pourtout  $t \notin T$  faire
7.      $L(t) \leftarrow \min (L(t), L(v) + d(v, t))$ ;
    
```

La validité de l'algorithme provient de la proposition suivante.

**Proposition 9.1.1** À l'entrée dans la boucle 3, on a :

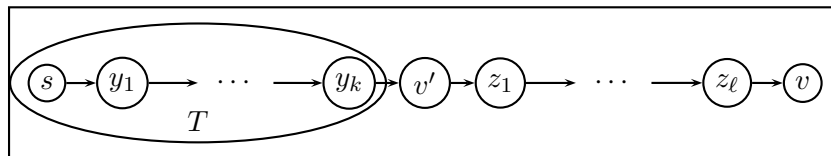
- (a) pour tout  $t \in T$ ,  $L(t)$  est la longueur d'un chemin minimal de  $s$  à  $t$  dans  $\mathcal{G}$ ;
- (b) pour tout  $v \notin T$ ,  $L(v)$  est la longueur d'un chemin minimal de  $s$  à  $v$  ne passant par aucun sommet de  $S - T$  (en dehors de  $v$ ).

*Preuve.* On procède par récurrence.

Pour  $|T| = 1$ , c'est-à-dire  $T = \{s\}$  : (a) et (b) sont vraies par construction (cf. ligne 2.).

Supposons donc  $|T| > 1$ . Juste avant la ligne 5, d'après l'hypothèse de récurrence,  $L(v)$  est la distance d'un chemin minimal de  $s$  à  $v$  ne passant que par des sommets de  $T$ .

Raisonnons par l'absurde et supposons que quand on ajoute  $v$ , ce n'est pas la longueur d'un chemin minimal de  $s$  à  $v$  dans  $\mathcal{G}$ . Si c'est le cas, un plus court chemin de  $s$  à  $v$  doit contenir au moins un autre sommet qui ne soit pas dans  $T$ . Soit  $v'$  le premier que l'on rencontre en partant de  $s$ .



Soit  $c$  ce plus court chemin  $c = (s, y_1, \dots, y_k, v', z_1, \dots, z_\ell, v)$ ; pour tout  $i$ ,  $y_i \in T$ . Soit  $c'$  le chemin extrait  $c' = (s, y_1, \dots, y_k, v')$ . C'est un plus court chemin entre  $s$  et  $v'$ , car sinon, on aurait un chemin encore plus court de  $s$  à  $v$ . Par construction, la longueur  $\delta$  du chemin  $c'$  vérifie  $\delta < L(v)$ .

Par hypothèse de récurrence sur (b),  $\delta = L(v')$ , et donc  $L(v') < L(v)$ , ce qui contredit la propriété à la ligne 5.

La propriété (a) est bien encore valide; (b) s'en déduit grâce aux lignes 6-7.  $\square$

**Ex.** Considérons le graphe  $\mathcal{G}_1$  de la figure 9.1. Nous allons calculer les distances minimales entre 0 et les autres sommets.

Au départ de l'algorithme, les valeurs de la fonction  $L$  sont données par :

$t$	0	1	2	3	4
$L(t)$	0	$+\infty$	3	1	6

L'ensemble  $T$  contient d'abord  $\{0\}$ .

On cherche alors le sommet  $v$  d'étiquette  $L(v)$  minimale parmi les sommets non encore dans  $T$ . Ici, c'est le sommet 3, ce qui conduit au dessin

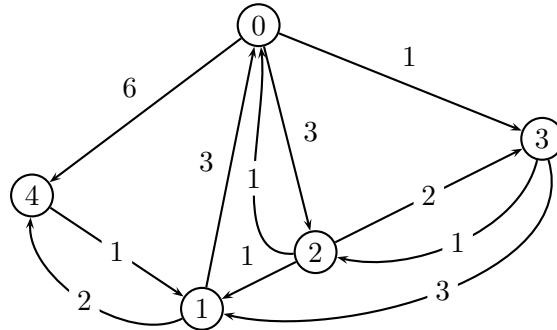


FIG. 9.1 – Le graphe  $\mathcal{G}_1$ .

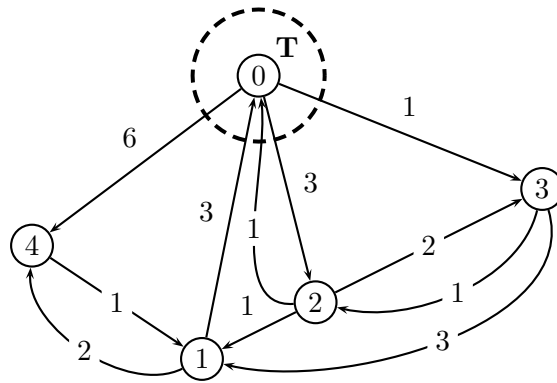


FIG. 9.2 – Étape 0.

La mise à jour de la fonction  $L$  donne :

$t$	1	2	3	4
$L(t)$	4	2	1	6

et on ajoute le sommet 2 dans  $T$  pour obtenir la figure 9.4.

La mise à jour de la fonction  $L$  donne :

$t$	1	2	3	4
$L(t)$	3	2	1	6

et on ajoute 1 à  $T$ . À l'étape 3, on trouve finalement

$t$	1	2	3	4
$L(t)$	3	2	1	5

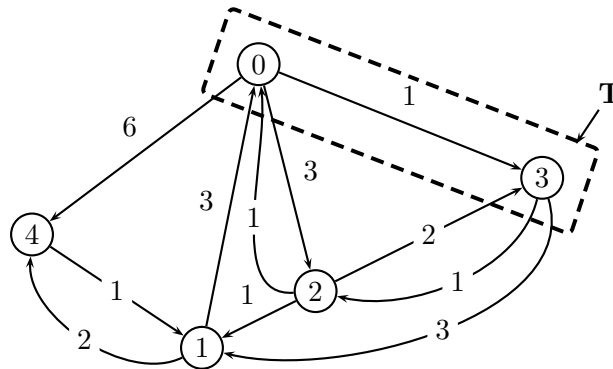


FIG. 9.3 – Étape 1.

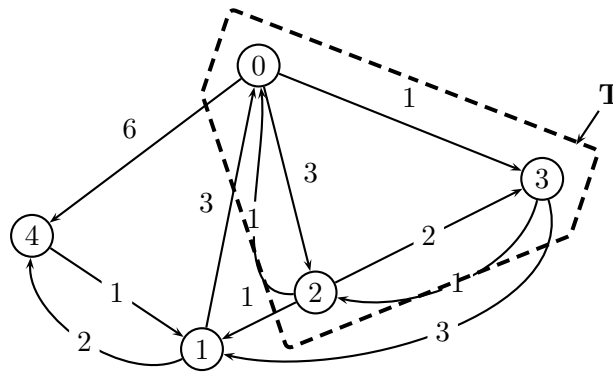


FIG. 9.4 – Étape 2.

### 9.1.2 Implantations

Cet algorithme est très intéressant à étudier et implanter, car le choix des structures de données a un impact crucial sur la complexité de l'algorithme. Notons tout de suite que l'algorithme va effectuer  $n = |\mathcal{S}|$  étapes.

#### Préliminaires

La sortie de l'algorithme sera un tableau indexé par les sommets, donc une table de hachage indexée par les sommets.

Notons qu'il est plus facile de gérer l'ensemble  $U = S - T$ . Cela nous conduit à récrire le pseudocode sous la forme :

```

1. [Initialisation] pourtout  $t \neq s$  faire  $L(t) \leftarrow d(s, t)$ ;
2.  $L(s) \leftarrow 0$ ;  $U \leftarrow S - \{s\}$ ;
    
```

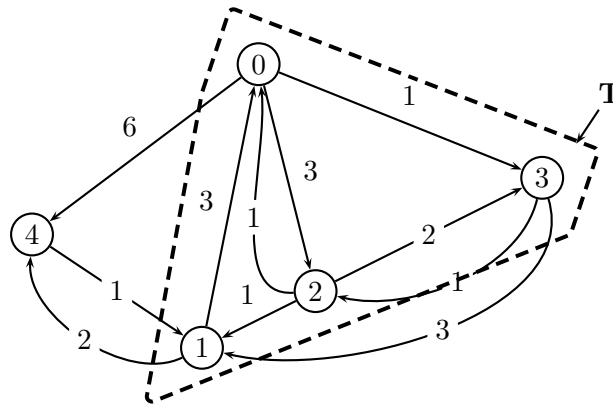


FIG. 9.5 – Étape 3.

```

3. tantque  $U \neq \emptyset$ 
4.   trouver  $v \in U$  telque  $L(v) = \min \{L(t), t \in U\}$ ;
5.    $U \leftarrow U - \{v\}$ ;
6.   pourtout  $t \in U$  faire
7.      $L(t) \leftarrow \min (L(t), L(v) + d(v, t))$ ;

```

### Utilisation d'un ensemble

La première solution envisageable est de représenter  $U$  par un ensemble. L'étape 4 est la simple recherche du minimum dans un ensemble.

Quelle performance obtenons-nous? Trouver le minimum des  $L(t)$  pour  $t \in U$  coûte  $O(|U|) = O(n)$ . La mise à jour coûte  $O(n_v)$  si  $n_v$  est le nombre de voisins de  $v$ , d'où un coût total :

$$\sum_{|T|=1}^n (O(n - |T|) + O(n_v)) = O(n^2) + O(|\mathcal{A}|) = O(n^2).$$

En Java, on peut par exemple utiliser un objet de `HashSet`, qui permet de récupérer un élément en temps constant. Le code correspondant est alors, en laissant les points de repère du pseudocode :

```

Hashtable<Sommet,Integer> Dijkstra(Sommet s){
    int INFINITY = 1000000; // [1]
    Hashtable<Sommet,Integer> L =
        new Hashtable<Sommet,Integer>();

    // 1. initialisation
    for(Sommet t : sommets())
        if(! t.equals(s))

```

```

        if(existeArc(s, t))
            L.put(t, valeurArc(s, t));
        else
            L.put(t, INFINITY);
// 2.
HashSet<Sommet> U
    = new HashSet<Sommet>(sommets()); // [2]
L.put(s, 0);
U.remove(s);
// 3.
while(! U.isEmpty()){
    // 4. recherche du minimum
    Sommet v = null;
    int Lv = INFINITY;
    for(Sommet t : U)
        if(L.get(t) < Lv){
            v = t;
            Lv = L.get(t);
        }
    // 5.
    U.remove(v);
    // 6.
    for(Arc<Sommet> a : voisins(v)){
        Sommet t = a.destination();
        int tmp = Lv + a.valeur();
        if(tmp < L.get(t)) // [3]
            L.put(t, tmp);
    }
}
return L;
}

```

Apportons quelques précisions :

- 1 On a utilisé un majorant sur les valeurs des arcs. Le cas échéant, on peut déterminer un vrai majorant à la construction du graphe, pour un coût  $O(|A|)$ .
- 2 Il faut être prudent, car nous sommes amenés à enlever les sommets de  $U$ . Une écriture du type

```
Collection<Sommet> U = sommets();
```

aurait eu pour conséquence d'enlever les sommets de la table des sommets du graphe de départ, ce qui n'est pas une bonne idée en général.

- 3 On ne doit pas s'inquiéter du fait qu'on ne teste pas directement l'appartenance de  $t$  à  $U$  comme l'algorithme l'indique. En effet, si  $t$  est dans  $T$ , il n'y a aucun risque d'améliorer la valeur  $L(t)$ .

### Utilisation d'une file de priorité

Nous décrivons dans cette section une amélioration de la complexité de la phase de recherche.

Quelle opération avons-nous à faire ? À chaque étape, on doit trouver l'élément  $v$  de  $U = S - T$  qui minimise  $L$ . On peut utiliser une file de priorité qui permet de trouver le minimum d'un ensemble en temps  $O(\log n)$ , avec un coût de mise à jour en  $O(\log n)$ .

Le temps de calcul de notre algorithme serait alors :

$$\sum_{|T|=1}^n (O(\log n) + O(n_v \log n)) = O((|\mathcal{A}| + n) \log n),$$

ce qui est meilleur que  $O(n^2)$  si  $|\mathcal{A}| \ll n^2 / \log n$ .

Donnons un programme Java implantant cette idée<sup>1</sup>. Cela va nous permettre de donner des exemples non-triviaux de syntaxe et d'utilisation de Java.

En Java, la classe `TreeSet` permet d'implanter une file de priorité. Elle prend en paramètre une classe avec un comparateur (**extends** `Comparable<T>`) ou bien directement un comparateur à la création. C'est cette dernière syntaxe que nous allons donner. Nous choisissons d'utiliser une file de priorité de `SommetValue` qui utilise une valeur pour un sommet :

```
package grapheX;

public class SommetValue {
    Sommet s;
    int valeur;

    SommetValue(Sommet ss, int vv) {
        s = ss;
        valeur = vv;
    }
}
```

Donnons la fonction d'initialisation de notre deuxième version de l'algorithme :

```
TreeSet<SommetValue> Dijkstra2Init(Hashtable<Sommet,Integer> L,
                                   Sommet s){
    int INFINITY = 1000000;
    TreeSet<SommetValue> fp =
        new TreeSet<SommetValue>(
            new Comparator<SommetValue>(){
                public int compare(SommetValue s1, SommetValue s2){
                    if(s1.valeur < s2.valeur)
```

<sup>1</sup>Le programme qui suit peut être omis dans une première phase de la lecture du poly.

```

        return -1;
    else if(s1.valeur > s2.valeur)
        return 1;
    if(s1.s.equals(s2.s))
        return 0;
    return -1;
    }
    }
    );
for(Sommet u : sommets()){
    if(u.equals(s))
        L.put(u, 0);
    else if(existeArc(s, u)){
        int val = valeurArc(s, u);
        fp.add(new SommetValue(u, val));
        L.put(u, val);
    }
    else{
        fp.add(new SommetValue(u, INFINITY));
        L.put(u, INFINITY);
    }
}
return fp;
}
}

```

La boucle sur les sommets du graphe permet l'initialisation de la file de priorité, ainsi que de la table L qui a la même utilisation que dans la première version. Notez le rôle joué par INFINITY comme précédemment. Nous stockons deux fois la valeur  $L(t)$ , une fois dans le résultat final, une fois dans la file de priorité. Nous en avons besoin pour pouvoir enlever des couples  $(t, L(t))$  de la file. L'initialisation telle qu'écrite a un coût  $O(|S| + n_s) = O(n)$ .

La fonction principale est alors :

```

Hashtable<Sommet,Integer> Dijkstra2(Sommet s){
    Hashtable<Sommet,Integer> L
        = new Hashtable<Sommet,Integer>();

    // 1. initialisation
    TreeSet<SommetValue> U = Dijkstra2Init(L, s);
    // 3.
    while(! U.isEmpty()){
        // 4.
        Sommet v = U.first();
        U.remove(v);
        int Lv = v.valeurMarque();
    }
}

```

```

        // 5.
        // 6.
        for(Arc<Sommet> a : voisins(v.s)){
            Sommet t = a.destination();
            int tmp = Lv + a.valeur();
            if(tmp < L.get(t)){
                U.remove(new SommetValue(t, L.get(t))); // [1]
                U.add(new SommetValue(t, tmp));
                L.put(t, tmp);
            }
        }
    }
    return L;
}

```

- 1 Pour mettre la file de priorité à jour, le plus simple est d'enlever le sommet, puis de mettre à la place un sommet avec sa nouvelle valeur.

On pourrait améliorer un peu les constantes d'implantation en étant moins brutal sur la mise à jour de  $U$ . Toutefois, ce n'est pas possible en restant générique. Il faudrait choisir une implantation par tas ou arbre et être capable d'aller chercher un nœud en plein milieu, pour accélérer la mise à jour. Nous laissons ce point comme sujet de méditation.

### Pour aller plus loin

En utilisant un tas de Fibonacci, on peut faire décroître la complexité à  $O(n \log n) + O(|A|)$ . Nous renvoyons à la littérature pour cet algorithme qui dépasse le cadre actuel du cours.

## 9.2 Arbre couvrant de poids minimal

Imaginons un réseau de canaux de capacité connue devant irriguer un nombre donné de villes. Le problème est de minimiser la somme totale des capacités. On peut modéliser le problème par un graphe (non orienté) dont les sommets sont les villes et les arêtes les canaux, dont les capacités sont données. On notera  $val(a)$  la valuation de l'arc  $a$ .

Considérons le graphe de la figure 9.6.

On note  $\varpi(\mathcal{G})$  le poids d'un graphe, défini comme la somme des valeurs de ses arcs.

Nous cherchons un sous-graphe connexe du graphe de départ, qui passe par tous les sommets du graphe, et de poids minimum. Si ce sous-graphe possédait un cycle, il suffirait de l'enlever pour diminuer son coût. Donc on cherche en fait un arbre couvrant de poids minimum.

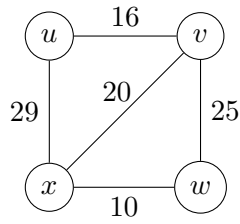
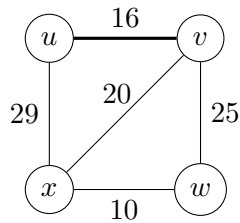


FIG. 9.6 –

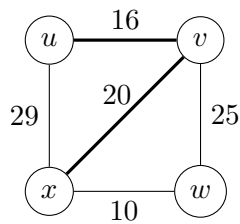
### 9.2.1 L'algorithme de Prim

1. Choisir un sommet initial  $s$ ;
2. **tantque** c'est possible
3. Trouver l'arête de poids minimal joignant un sommet déjà sélectionné **à** un sommet non encore sélectionné.

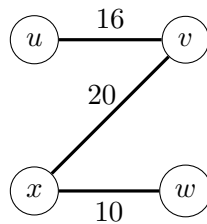
Reprenons encore le graphe donné en exemple, en commençant par exemple par le sommet  $u$ . On ajoute alors l'arête minimale joignant  $u$ , ici l'arête  $uv$  :



Puis nous passons en revue les arêtes contenant  $u$  et celles contenant  $v$ . L'arête de poids minimal est  $xv$  :



On cherche alors parmi les arêtes contenant  $\{u, v, x\}$ , ici  $wx$  :



Tous les sommets ayant été utilisés, l'algorithme s'arrête.

Comment peut-on implanter l'algorithme. Le plus efficace est de gérer une file de priorité d'arcs, classés par ordre croissant de valuation.

```

Prim(G = (S, A))
X <- S;
tantque X ≠ ∅ faire
  u <- élément suivant de X;
  X <- X - {u};
  T[u] <- ∅; // arbre couvrant de racine u
  F <- file de priorité avec les arêtes (u, x);
  Y <- {u}; // ensemble des sommets de T[u]
  tantque F ≠ ∅ faire
    a = (s, t) <- arête minimale dans F;
    si t ∈ Y alors
      // (s, t) ferme un cycle
    sinon
      Y <- Y ∪ {t};
      X <- X - {t};
      T[u] <- T[u] ∪ {(s, t)};
      pour v voisin de t
        ajouter (t, v) dans F;

```

**Exercice 9.2.1** Implanter l'algorithme de Prim en Java.

**Exercice 9.2.2** Montrer qu'il existe une variante de Prim fonctionnant en temps  $O(m \log n)$ .

## 9.2.2 L'algorithme de Kruskal

Il peut être décrit simplement comme suit :

```

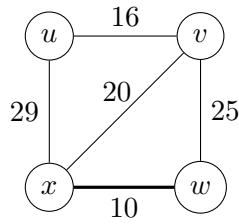
1. Trier les arêtes par ordre croissant pour obtenir
   B = (a1, ..., am);
2. T <- ∅;
3. pour i <- 1 à m faire
   si ai ne crée pas de cycle
     ajouter ai à T;

```

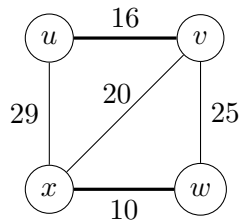
Revenons au cas du graphe de la figure 9.6. On trie les arêtes :

$$wx = 10, uv = 16, xv = 20, vw = 25, xu = 29.$$

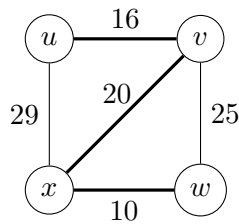
On commence par sélectionner l'arête  $wx$ , que nous dessinons en gras :



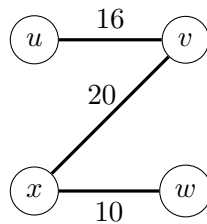
La deuxième arête est  $uv$  :



La troisième est  $xv$ , puisque nous n'avons toujours pas de cycle :



Nous ne pouvons pas utiliser  $vw$ , car cela créerait un cycle. Nous l'enlevons donc du graphe, ainsi que l'arête  $xu$ , qui elle aussi fermerait un cycle :



Le coût de l'étape 1 est bien sûr  $O(m \log m)$  avec  $m = |\mathcal{A}|$ .

Comment implanter la deuxième étape ? On va créer une collection d'ensemble  $C$  indexée par les sommets. Si  $t \in C[s]$ , cela veut dire qu'il existe un chemin entre  $t$  et  $s$  dans la forêt en construction. Détaillons le code permettant d'insérer éventuellement une arête  $a$ . On suppose qu'à l'initialisation, on a mis tous les  $C[s]$  à NIL.

```
insérer(C, a)
a = (s, t)
si C[s] == NIL et C[t] == NIL alors
  // l'arête (s, t) est nouvelle
```

```

C[s] <- {s, t};
C[t] <- C[s]; // partage
sinon si C[s] == NIL alors
  // t est déjà connu
  C[t] <- C[t] ∪ {s};
  C[s] <- C[t]; // partage
sinon si C[t] == NIL alors
  // s est déjà connu
  C[s] <- C[s] ∪ {t};
  C[t] <- C[s];
sinon // s et t sont déjà connus
  si C[s] == C[t] alors
    // s et t sont dans la même composante,
    // (s, t) forme un cycle, on ne fait rien
  sinon
    fusionner C[s] et C[t].

```

Reprenons notre exemple. On commence par

$$C[w] = C[x] = \{w, x\}.$$

La deuxième arête est  $uv$ , ce qui crée

$$C[u] = C[v] = \{u, v\}.$$

Quand on veut insérer  $xv$ , on assiste à la fusion des composantes :

$$C[w] = C[x] = C[u] = C[v] = \{w, x, u, v\}.$$

Les deux dernières arêtes ne sont pas insérées, puisque les sommets sont déjà dans  $C$ .

Le lecteur attentif aura reconnu le principe de l'algorithme Union-Find. On sait que la complexité de traitement est  $O(m\alpha(m))$  avec  $\alpha(m)$  l'inverse de la fonction d'Ackermann  $Ack(m, m)$ , ce qui le rend quasi-linéaire.

**Exercice 9.2.3** Implanter l'algorithme de Kruskal en Java.

**Exercice 9.2.4** Trouver un codage de l'arbre couvrant construit par l'un ou l'autre des algorithmes.

### 9.2.3 Ébauche de preuve

Cette esquisse est valable pour les deux algorithmes.

On construit  $T$  avec la liste d'arêtes  $(a_1, a_2, \dots, a_{n-1})$  telle que

$$val(a_1) \leq val(a_2) \leq \dots \leq val(a_{n-1}).$$

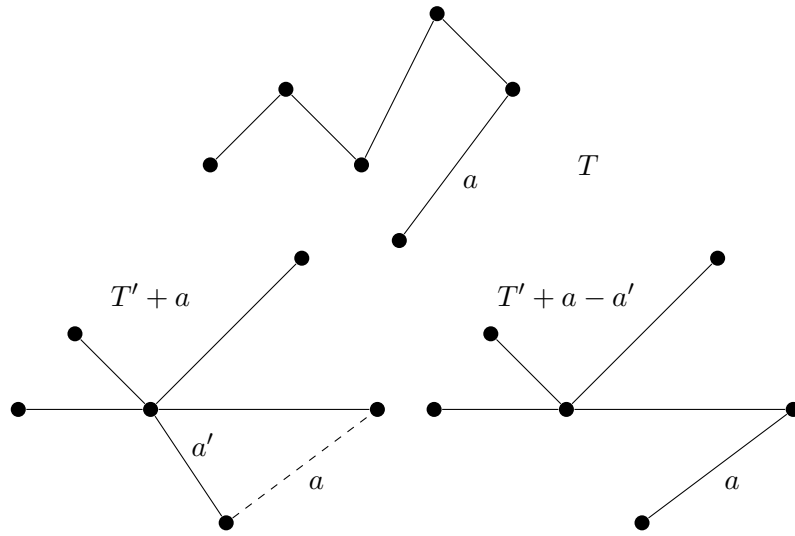


FIG. 9.7 – Les arbres  $T$  et  $T'$ .

Supposons qu'il existe un arbre couvrant  $T'$  tq  $\varpi(T') < \varpi(T)$ . Soit  $a$  une arête de poids minimum présente dans  $T$  mais pas dans  $T'$  :

$$(a_1, a_2, \dots, a_{r-1}, a_r = a, a_{r+1}, \dots, a_{n-1})$$

avec  $a_i \in T'$  pour  $i < r$ .

**Lemme 9.2.1**  $T' + a$  contient un cycle, qui contient une autre arête  $a'$  non présente dans  $T$ .

*Démonstration.* tous les sommets de  $\mathcal{S}$  sont déjà dans  $T'$ , donc dès qu'on rajoute une arête, on ferme un cycle (voir la figure 9.7).

Si toutes les arêtes du cycle étaient dans  $T$ ,  $a$  n'aurait pu être choisi par l'algorithme (car alors  $a$  aurait créé ce cycle).  $\square$

**Lemme 9.2.2** Le graphe  $T'' = T' + a - a'$  est encore un arbre couvrant.

*Démonstration.* Si  $val(a') < val(a)$  : comme  $a' \notin T$ , il n'était pas éligible par l'algorithme, donc il y aurait un cycle dans  $\{a_1, a_2, \dots, a', \dots, a_{r-1}\}$  et donc un cycle dans  $T'$ . D'où  $\varpi(T'') = \varpi(T') + val(a) - val(a') \leq \varpi(T') < \varpi(T)$ .

De proche en proche, on peut donc passer de  $T$  à  $T'$ , alors que le poids de  $T$  est plus grand que celui de  $T'$ , contradiction.  $\square$

### 9.2.4 Prim ou Kruskal ?

Dans sa variante rapide, Prim a coût  $O(m \log n)$ , Kruskal  $O(m \log m)$ . Comme  $\log m = O(\log n)$ , on est dans la même classe de complexité. Néanmoins, si on sait que  $n \ll m$ , alors Prim gagne un peu.

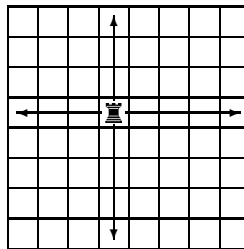
Si l'on regarde bien, Prim ne trouve l'arbre couvrant que pour la composante connexe du sommet de départ, alors que Kruskal construit tous les arbres en même temps.

### 9.3 Les $n$ reines

Nous allons encore voir un algorithme de backtrack pour résoudre un problème combinatoire. Dans la suite, nous supposons que nous utilisons un échiquier  $n \times n$ .

#### 9.3.1 Prélude : les $n$ tours

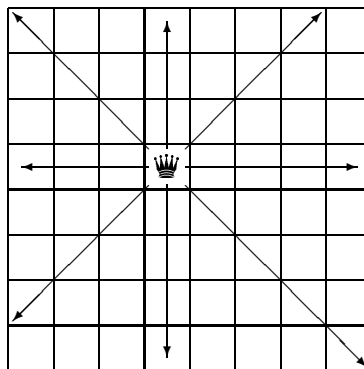
Rappelons quelques notions du jeu d'échecs. Une tour menace toute pièce adverse se trouvant dans la même ligne ou dans la même colonne.



On voit facilement qu'on peut mettre  $n$  tours sur l'échiquier sans que les tours ne s'attaquent. En fait, une solution correspond à une permutation de  $1..n$ , et on sait déjà faire. Le nombre de façons de placer  $n$  tours non attaquantes est donc  $T(n) = n!$ .

#### 9.3.2 Des reines sur un échiquier

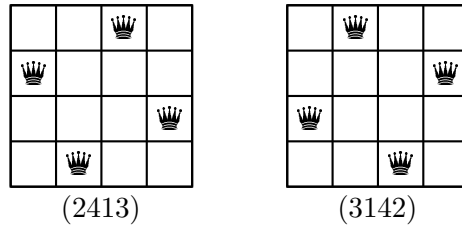
La reine se déplace dans toutes les directions et attaque toutes les pièces (adverses) se trouvant sur les même ligne ou colonne ou diagonales qu'elle.



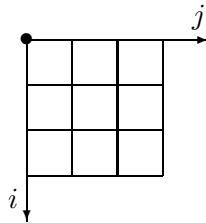
Une reine étant une tour avec un peu plus de pouvoir, il est clair que le nombre maximal de reines pouvant être sur l'échiquier sans s'attaquer est au plus  $n$ . On peut

montrer que ce nombre est  $n$  pour  $n = 1$  ou  $n \geq 4$ . Reste à calculer le nombre de solutions possibles, et c'est une tâche difficile, et non résolue.

Donnons les solutions pour  $n = 4$  :



Expliquons comment résoudre le problème de façon algorithmique. On commence par chercher un codage d'une configuration. Une configuration admissible sera codée par la suite des positions d'une reine dans chaque colonne. On oriente l'échiquier comme suit :



Avec ces notations, on démontre :

**Proposition 9.3.1** *La reine en position  $(i_1, j_1)$  attaque la reine en position  $(i_2, j_2)$  si et seulement si  $i_1 = i_2$  ou  $j_1 = j_2$  ou  $i_1 - j_1 = i_2 - j_2$  ou  $i_1 + j_1 = i_2 + j_2$ .*

*Démonstration* : si elle sont sur la même diagonale nord-ouest/sud-est,  $i_1 - j_1 = i_2 - j_2$  ; ou encore sur la même diagonale sud-ouest/nord-est,  $i_1 + j_1 = i_2 + j_2$ .  $\square$

On va procéder comme pour les permutations : on suppose avoir construit une solution approchée dans  $t[1..i_0]$  et on cherche à placer une reine dans la colonne  $i_0$ . Il faut s'assurer que la nouvelle reine n'attaque personne sur sa ligne (c'est le rôle du tableau `utilise` comme pour les permutations), et personne dans aucune de ses diagonales (fonction `pasDeConflit`). Le code est le suivant :

```

// t[1..i0] est déjà rempli
static void reinesAux(int[] t, int n,
                    boolean[] utilise, int i0){
    if(i0 > n)
        afficher(t);
    else{
```

<sup>2</sup>Les petits cas peuvent se faire à la main, une preuve générale est plus délicate et elle est due à Ahrens, en 1921.

```

        for(int v = 1; v <= n; v++)
            if(! utilise[v] && pasDeConflit(t, i0, v)){
                utilise[v] = true;
                t[i0] = v;
                reinesAux(t, n, utilise, i0+1);
                utilise[v] = false;
            }
    }
}

```

La programmation de la fonction pasDeConflit découle de la proposition 9.3.1 :

```

// t[1..i0[ est déjà rempli
static boolean pasDeConflit(int[] t, int i0, int j){
    int x1, y1, x2 = i0, y2 = j;

    for(int i = 1; i < i0; i++){
        // on récupère les positions
        x1 = i;
        y1 = t[i];
        if((x1 == x2) // même colonne
            || (y1 == y2) // même ligne
            || ((x1-y1) == (x2-y2))
            || ((x1+y1) == (x2+y2)))
            return false;
    }
    return true;
}

```

Notons qu'il est facile de modifier le code pour qu'il calcule le nombre de solutions. Terminons par un tableau des valeurs connues de  $R(n)$  :

$n$	$R(n)$	$n$	$R(n)$	$n$	$R(n)$	$n$	$R(n)$
4	2	9	352	14	365596	19	4968057848
5	10	10	724	15	2279184	20	39029188884
6	4	11	2680	16	14772512	21	314666222712
7	40	12	14200	17	95815104	22	2691008701644
8	92	13	73712	18	666090624	23	24233937684440

Vardi a conjecturé que  $\log R(n)/(n \log n) \rightarrow \alpha > 0$  et peut-être que  $\alpha = 1$ . Rivin & Zabih ont d'ailleurs mis au point un algorithme de meilleur complexité pour résoudre le problème, avec un temps de calcul de  $O(n^2 8^n)$ .

## 9.4 Les ordinateurs jouent aux échecs

Nous ne saurions terminer un chapitre sur la recherche exhaustive sans évoquer un cas très médiatique, celui des ordinateurs jouant aux échecs.

### 9.4.1 Principes des programmes de jeu

Deux approches ont été tentées pour battre les grands maîtres. La première, dans la lignée de Botvinnik, cherche à programmer l'ordinateur pour lui faire utiliser la démarche humaine. La seconde, et la plus fructueuse, c'est utiliser l'ordinateur dans ce qu'il sait faire le mieux, c'est-à-dire examiner de nombreuses données en un temps court.

Comment fonctionne un programme de jeu ? En règle général, à partir d'une position donnée, on énumère les coups valides et on crée la liste des nouvelles positions. On tente alors de déterminer quelle est la meilleure nouvelle position possible. On fait cela sur plusieurs tours, en parcourant un arbre de possibilités, et on cherche à garder le meilleur chemin obtenu.

Dans le meilleur des cas, l'ordinateur peut examiner tous les coups et il gagne à coup sûr. Dans le cas des échecs, le nombre de possibilités en début et milieu de partie est beaucoup trop grand. Aussi essaie-t-on de programmer la recherche la plus profonde possible.

### 9.4.2 Algorithme minimax

Dans cet algorithme, on imagine que le joueur A va prendre la meilleure décision possible compte tenu des coups qu'il peut jouer et des coups que son adversaire B peut jouer. À partir d'une position donnée au niveau 0, A joue, ouvrant un certain nombre de coups valides à B, ce qui conduit à d'autres coups, et ainsi de suite. Regardons ce qui se passe sur un exemple à la figure 9.8.

Au niveau le plus, A joue et cela conduit à des positions positives si la position est gagnante, négative sinon. Si A a le choix entre une position finale à +1 ou -2, elle va choisir +1, ce que l'on symbolise au cran précédent par +1 dans le nœud de l'arbre. A maximise ainsi l'évaluation de son coup. Au cran d'avant, B va au contraire minimiser le gain de A, puisqu'il veut gagner. C'est ainsi qu'entre un coup conduisant à +1 ou +3, il choisit +1. Parcourant ainsi toutes les branches de l'arbre, on voit que le meilleur coup pour A avec une profondeur de 4 conduit à une position +3.

Il est facile de voir que la complexité de cette approche est  $O(b^{\text{prof}})$ .

### 9.4.3 Principe de l'élagage $\alpha\beta$

Dans certains cas, on peut couper des branches de l'arbre, ce qui permet de gagner du temps dans l'exploration. Expliquons ce qui se passe sur l'exemple de la figure 9.9

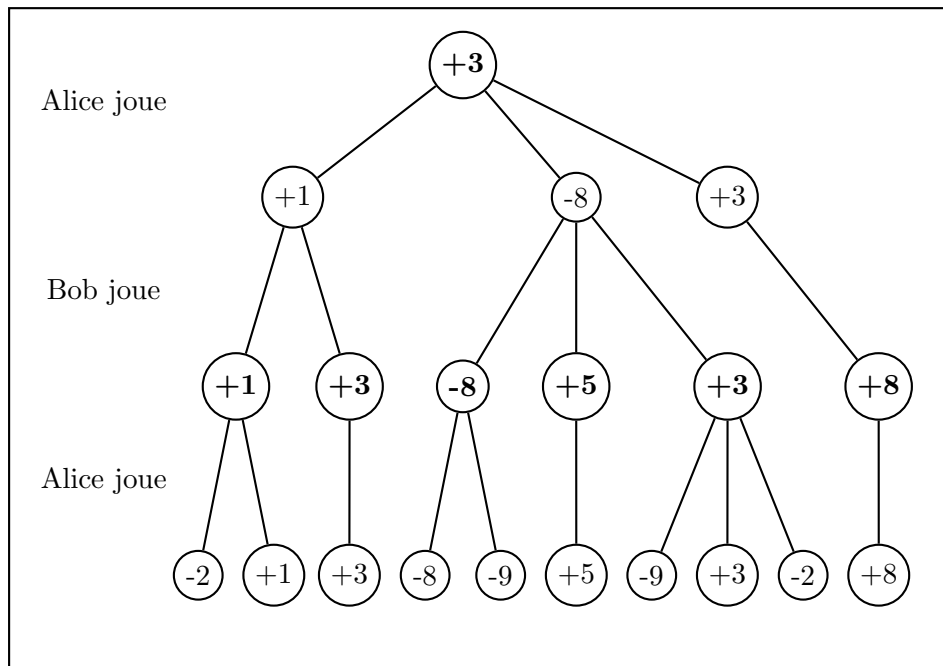


FIG. 9.8 – Exemple de minimax.

#### 9.4.4 Retour aux échecs

##### Codage d'une position

La première idée qui vient à l'esprit est d'utiliser une matrice  $8 \times 8$  pour représenter un échiquier. On l'implante généralement sous la forme d'un entier de type `long` qui a 64 bits, un bit par case. On gère alors un ensemble de tels entiers, un par type de pièce par exemple.

On trouve dans la thèse de J. C. Weill un codage astucieux :

- les cases sont numérotées de 0 à 63;
- les pièces sont numérotées de 0 à 11 : pion blanc = 0, cavalier blanc = 1, ..., pion noir = 6, ..., roi noir = 11.

On stocke la position dans le vecteur de bits

$$(c_1, c_2, \dots, c_{768})$$

tel que  $c_{64i+j+1} = 1$  ssi la pièce  $i$  est sur la case  $j$ .

Les positions sont stockées dans une table de hachage la plus grande possible qui permet de reconnaître une position déjà vue.

##### Fonction d'évaluation

C'est un des secrets de tout bon programme d'échecs. L'idée de base est d'évaluer la force d'une position par une combinaison linéaire mettant en œuvre le poids d'une

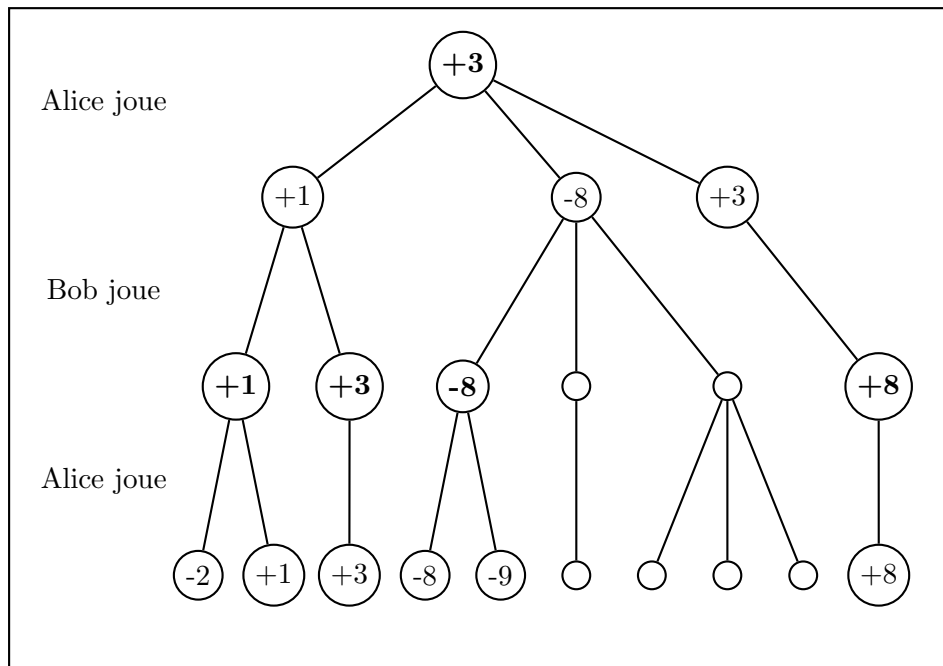


FIG. 9.9 – Exemple d'élagage alpha beta.

pièce (reine = 900, tour= 500, etc.). On complique alors généralement la fonction en fonction de stratégies (position forte du roi, etc.).

### Bibliothèques de début et fin

Une façon d'accélérer la recherche est d'utiliser des bibliothèques d'ouvertures pour les débuts de partie, ainsi que des bibliothèques de fins de partie.

Dans ce dernier cas, on peut tenter, quand il ne reste que peu de pièces d'énumérer toutes les positions et de classer les perdantes, les gagnantes et les nulles. L'algorithme est appelé analyse rétrograde et a été décrite par Ken Thompson (l'un des créateurs d'Unix).

À titre d'exemple, la figure 9.10 décrit une position à partir de laquelle il faut **243 coups** (contre la meilleure défense) à Blanc (qui joue) pour capturer une pièce sans danger, avant de gagner (Stiller, 1998).

### Deep blue contre Kasparov (1997)

Le projet a démarré en 1989 par une équipe de chercheurs et techniciens : C. J. Tan, Murray Campbell (fonction d'évaluation), Feng-hsiung Hsu, A. Joseph Hoane, Jr., Jerry Brody, Joel Benjamin. Une machine spéciale a été fabriquée : elle contenait 32 nœuds avec des RS/6000 SP (chip P2SC) ; chaque nœud contenait 8 processeurs spécialisés pour les échecs, avec un système AIX. Le programme était écrit en C pour le IBM SP

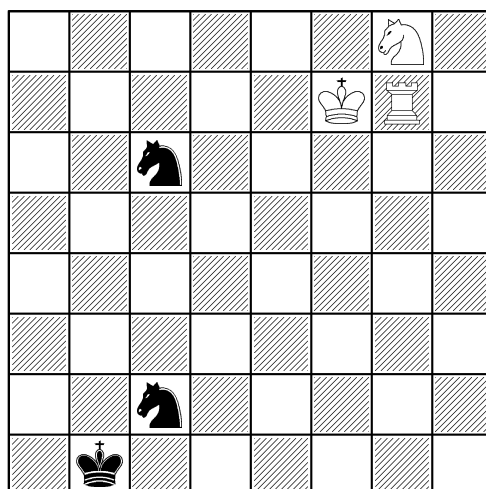


FIG. 9.10 – Une très longue partie.

Parallel System (MPI). La machine était capable d'engendrer 200,000,000 positions par seconde (ou  $60 \times 10^9$  en 3 minutes, le temps alloué). Deep blue a gagné 2 parties à 1 contre Kasparov<sup>3</sup>.

### Deep Fritz contre Kramnik (2002)

C'est cette fois un ordinateur plus raisonnable qui affronte un humain : 8 processeurs à 2.4 GHz et 256 Mo, qui peuvent calculer 3 millions de coups à la seconde. Le programmeur F. Morsch a soigné la partie algorithmique. Kramnik ne fait que match nul (deux victoires chacun, quatre nulles), sans doute épuisé par la tension du match.

### Conclusion

Peut-on déduire de ce qui précède que les ordinateurs sont plus intelligents que les humains ? Certes non, ils *calculent* plus rapidement sur certaines données, c'est tout. Pour la petite histoire, les joueurs d'échec peuvent s'adapter à l'ordinateur qui joue face à lui et trouver des positions qui le mettent en difficulté. Une manière de faire est de jouer systématiquement de façon à maintenir un grand nombre de possibilités à chaque étape.

<sup>3</sup>[www.research.ibm.com/deepblue](http://www.research.ibm.com/deepblue)



Troisième partie

Langages



## Chapitre 10

# Langages rationnels

Une des fonctions principales des ordinateurs dans le monde est de rechercher des informations dans des bases de connaissance : courrier, fichiers de documentation, articles, livres, pages Web, etc. Dans les moteurs de recherche du Web, dans tous les systèmes d'interrogation bibliographiques, dans les traducteurs et compilateurs on retrouve cette fonction essentielle qui permet de retrouver rapidement les pages, les ouvrages, les passages du texte contenant un mot, un groupe de mots avec ou sans variante. Les logiciels les plus élaborés peuvent tenir compte de la phonétique comme dans l'annuaire électronique de France-Télécom. De fait, la théorie sous-jacente a été développée dans les années 1950, sous l'impulsion initiale de linguistes, de logiciens et d'informaticiens théoriciens pour donner la théorie des automates finis et des langages rationnels (et expressions régulières).

Par ailleurs, ces mêmes automates sont à la base de très nombreux automatismes développés par les électroniciens et autres concepteurs d'appareils électro-mécaniques. Les réseaux téléphoniques, les circuits de commande et de contrôle sont en fait des automates finis munis de senseurs et d'effecteurs. On remonte en cela à la tradition de Vaucanson et Jacquard (et bien d'autres). Les montres (mécaniques ou électroniques) sont de beaux exemples, ainsi que tous les appareils-photos et autres caméras ou lecteurs de CD.

Comprendre la puissance et les limites de ce modèle simple est donc important. On verra aussi apparaître le phénomène du non-déterminisme.

### 10.1 Recherche de motifs dans un texte

Cette section est dévolue à l'étude et à l'analyse d'algorithmes qui recherchent des motifs ou des ensembles de motifs simples. On commencera donc par la recherche d'une suite particulière de lettres (un *mot*) dans un texte, puis on compliquera en passant à un mot parmi un ensemble fini.

### 10.1.1 Un algorithme naïf

Considérons un ensemble fini  $A$  de lettres appelé *alphabet*. On appelle *mot* toute suite finie de lettres de  $A$ .

**Définition 10.1.1** *L'ensemble des mots sur un alphabet  $A$  est noté  $A^*$ . Un mot  $w \in A^*$  est défini par la suite de ses lettres :  $w = w_1w_2\dots w_k$ , où  $k = |w|$  est la longueur, ou encore le nombre de lettres, de  $w$ .*

*L'opérateur de concaténation est une loi interne binaire de  $A^*$  qui à deux mots  $u$  et  $v$  associe le mot  $w = uv$ , de longueur  $|u| + |v|$ , défini par*

$$w_i = \text{si } i \leq |u| \text{ alors } u_i \text{ sinon } v_{i-|u|}.$$

*Il existe un mot unique de longueur 0, appelé mot vide et noté  $\varepsilon$ .*

Notons, de plus, que la loi de concaténation n'est pas commutative, qu'elle est associative et que le mot vide est élément neutre.

**Définition 10.1.2** *L'ensemble  $A^*$  muni de la concaténation a une structure de monoïde, dont on dit qu'il est libre car la décomposition d'un mot sur les lettres de l'alphabet est unique.*

Cette propriété de décomposition unique n'est pas vraie de tous les monoïdes. Elle est ici triviale, est caractéristique des ensembles de mots formant des codes (la décomposition est unique) et est à la base de la théorie des codes combinatoires.

L'algorithme le plus simple de recherche de motif dans un texte essaie toutes les positions possibles du motif  $m$  en dessous du texte  $t$ . Comment tester qu'il existe une occurrence en position  $i$ ? Il suffit d'utiliser un indice  $j$  qui va servir à comparer  $m[j]$  à  $t[i+j]$  de proche en proche :

```
static boolean occurrence(String t, String m, int i){
    for(int j = 0; j < m.length(); j++)
        if(t.charAt(i+j) != m.charAt(j))
            return false;
    return true;
}
```

Nous utilisons cette primitive dans la fonction suivante, qui teste toutes les occurrences possibles :

```
static void naif(String t, String m){
    System.out.print("Occurrences en position :");
    for(int i = 0; i < t.length()-m.length(); i++)
        if(occurrence(t, m, i))
            System.out.print(" "+i+",");
    System.out.println("");
}
```

Cette fonction opère en un nombre de comparaisons qui dépend du motif et du texte. En effet, considérons le motif  $m = aaaaaab$  et les textes  $t_1 = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaab$  et  $t_2 = bbbbbbbbbbbbbbbbbbbbbbbbbb$  ; le nombre de comparaisons entre lettres du motif et du texte dans pour  $t_1$  est équivalent à  $|m|.|t_1|$  essentiellement alors qu'il n'est que de  $|t_2|$  dans le deuxième cas. Même si on peut montrer qu'en moyenne le coût ne dépend pas de la taille du motif (pour un texte aléatoire), on aimerait avoir une méthode insensible à la nature du motif et du texte, indépendante de leurs régularités ou irrégularités.

### 10.1.2 L'algorithme KMP

On doit remarquer que le gros défaut de la méthode naïve est d'oublier ce qui a été lu et comparé du texte après chaque décalage. On pourrait pourtant économiser beaucoup de comparaisons si l'on pouvait mémoriser par exemple que  $t_1$  ne contient que des  $a$  et donc qu'en cas de lecture de nombreux  $a$  consécutifs, il suffit de chercher le premier  $b$ . Plus précisément on peut construire, sur un exemple moins caricatural le dispositif suivant bien plus efficace.

**Définition 10.1.3** Soit  $w \in A^*$  un mot non vide ; pour toute décomposition de  $w = f_1 f_2 \dots f_k$  telle que  $f_i \neq \varepsilon$ , on dira que  $f_i$  est un facteur de  $w$  ;  $f_1$  sera appelé préfixe ou facteur gauche et  $f_k$  suffixe ou facteur droit.

Soient le motif  $m = abaabbab$  et le texte  $t = ababbabaabaabb\dots$ . On peut alors commenter la recherche d'une occurrence de  $m$  dans  $t$  comme suit, les étoiles correspondant aux comparaisons couronnées de succès et les dièses aux comparaisons conclues par un échec :

```

t = a b a b b a b a a b a a b b . . .
m = a b a a b b a b
    * * * #                reprendre la comparaison après le
                           dernier a b
      a b a a b b a b
        #                reprendre la comparaison au début
          a b a a b b a b
            * * * * * #   reprendre après le dernier a b a
              a b a a b b a b
                * * * . . . etc.
    
```

- On aura compris :
- que lors de la première différence entre  $m$  et  $t$ , on a pu garder en mémoire le fait que leur dernières lettres lues du texte coïncident avec les deux premières du motif et donc,
  - qu'on peut poursuivre la lecture sur la troisième lettre du motif sans revenir en arrière sur le texte.

Il y a alors échec immédiat et comme aucun suffixe des 5 premières lettres du texte ne correspond à un préfixe du motif, on en déduit qu'on peut décaler entièrement le motif.

Pour le troisième échec on constate que "a b a" est le plus long suffixe de la portion de texte lue qui soit aussi préfixe du motif, et on décale donc le motif de 3 positions, etc.

On constate donc qu'il serait possible de ne pas revenir en arrière sur le texte. Le processus de reprise en cas de discordance entre le texte et le motif peut se représenter par le schéma suivant où il suffit de suivre les flèches en fonction de la lettre lue sur le texte. Un tel schéma est appelé *automate*. Chaque position numérotée est appelée *état* ; le dernier état (matérialisé par un  $\diamond$ ) est appelé final et correspond à une occurrence du motif.

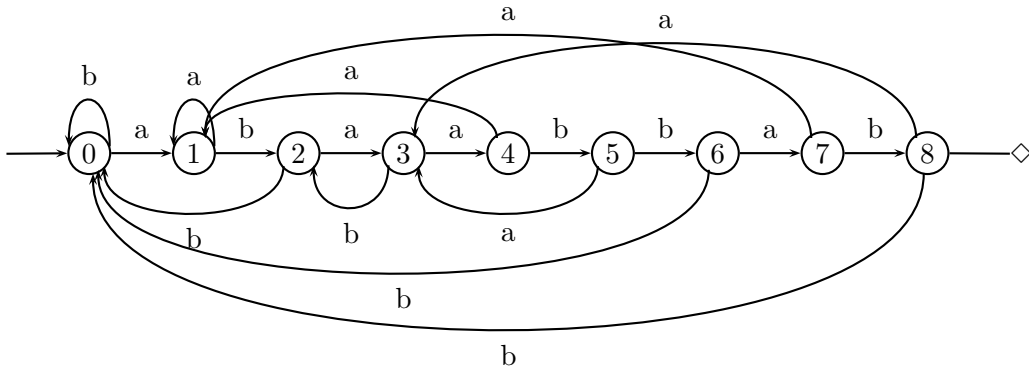


FIG. 10.1 – L'automate de reconnaissance du motif  $m = abaabbab$

L'automate associé à un motif  $m$ , possède donc  $|m| + 1$  états nommés  $0, 1, 2, \dots, |m|$  et les *transitions* de l'automate (ses flèches) sont définies par la fonction  $\tau : [0..|m|] \times A \rightarrow [0..|m|]$ , telle que  $\tau(i, \alpha)$  est le nombre de lettres du plus grand suffixe de  $m[1..i]\alpha$  qui soit aussi préfixe de  $m$ .

On a donc :

$$\tau(i, \alpha) = \begin{cases} i + 1 & \text{si } \alpha = m[i + 1], \\ \max\{j \leq i \mid m[1..j] = m[i - j + 2..i]\alpha\} & \text{sinon.} \end{cases}$$

La construction de cette table  $\tau$  prend un temps  $O(|m|^3)$  si l'on procède de manière directe, mais une fois ce pré-traitement (on dit souvent en jargon *preprocessing*) la recherche du motif se fait en temps linéaire  $O(|t|)$ , puisqu'il suffit de parcourir le graphe de l'automate en lisant à chaque étape une lettre de  $t$ .

Cependant, il est possible de construire la table de l'automate en temps linéaire, à condition d'introduire la fonction auxiliaire  $\delta : [0..|m|] \rightarrow [0..|m|]$ , telle que  $\delta(i)$  soit le nombre de lettres du plus grand préfixe de  $m[1..i - 1]$  qui soit aussi suffixe de  $m[2..i]$ .

On a donc :

$$\delta(i) = \begin{cases} 0 & \text{si } i = 0, \\ \max\{j < i \mid m[1..j] = m[i - j + 1..i]\} & \text{sinon.} \end{cases}$$

On remarquera les différences subtiles dans le domaine de variation de la variable  $j$ . On a alors la propriété remarquable que  $\tau$  et  $\delta$  se calculent ensemble de façon très efficace.

**Proposition 10.1.1** *Les fonctions  $\tau$  et  $\delta$  satisfont les récurrences croisées :*

$$\begin{aligned} \delta(i) &= \begin{array}{l} \text{si } i \leq 1 \text{ alors } 0 \\ \text{sinon } \tau(\delta(i-1), m[i]) \end{array} \\ \tau(i, \alpha) &= \begin{array}{l} \text{si } \alpha = m[i+1] \text{ alors } i+1 \\ \text{sinon si } i = 0 \text{ alors } 0 \\ \text{sinon } \tau(\delta(i), \alpha) \end{array} \end{aligned}$$

Preuve : (Esquisse)

Elle se fait par récurrence sur  $i$ .

Les choses sont évidentes pour  $i = 0, 1$ .

Dans le cas général, deux situations se présentent selon les valeurs respectives de  $m[i+1]$  et  $m[\delta(i)+1]$  : on garde en mémoire que, par hypothèse de récurrence,  $m[1..\delta(i)] = m[i-\delta(i)+1..i]$ .

Dans le cas où  $m[i+1] = m[\delta(i)+1] = \alpha$ , alors  $m$  a la structure suivante lorsque les facteurs appelés  $f$  ne sont pas chevauchants :  $m = f\alpha u f\alpha v f\alpha w$ , où  $f$  est de longueur  $\delta(i+1) - 1$ , le premier  $\alpha$  est donc en position  $\delta(i+1)$ , le second  $\alpha$  est en position  $\delta(i)+1$ , et le troisième  $\alpha$  est en position  $i+1$ . (Il est possible que les facteurs  $f\alpha$  se chevauchent, mais ceci n'affecte pas le raisonnement.)

Il faut justifier qu'il n'existe pas de facteur gauche  $g\alpha$  de  $m$  plus grand que  $f\alpha$  qui soit aussi facteur droit de  $m[1..i+1]$ . Si tel était le cas, comme  $m[1..\delta(i)]$  est aussi facteur droit de  $m[1..\delta]$ , il aurait aussi  $g$  comme facteur droit, ce qui est contraire à l'hypothèse de récurrence qui stipule que  $\delta(i) = |f|$ . En conséquence on a bien que  $\tau(\delta(i), m[i+1]) = \delta(i+1)$  et satisfait donc les définitions de  $\tau$  et  $\delta$ .

Dans le cas où  $m[i+1] \neq m[\delta(i)+1]$ ,  $m$  se décompose aussi selon un schéma semblable  $m = f u f v f w$ , les  $\alpha$  étant simplement omis. Le raisonnement est exactement identique et le résultat aussi.

On vérifie de la même façon (mais c'est un peu une évidence) que la fonction  $\delta$  se définit bien comme mentionné à partir de  $\tau$  et de  $m$ .

Le calcul des fonctions  $\tau$  et  $\delta$  devient alors beaucoup plus rapide, puisque de quadratique en  $|m|$ , il devient linéaire, le théorème précédent permettant un calcul par récurrence linéaire.

**Proposition 10.1.2** *Le calcul simultané des fonctions  $\tau$  et  $\delta$ , pour un motif  $m$  donné, se fait en temps et espace  $O(|m|)$ .*

On a donc le résultat (non trivial) de D. E. Knuth, J. H. Morris et V. R. Pratt, qui a des prolongements importants en matière de reconnaissance de mots à structure palindromique itérée.

**Théorème 10.1.1** *La recherche de toutes les occurrences d'un motif  $m$  dans un texte  $t$  se fait en temps linéaire  $O(|m| + |t|)$ .*

Preuve : On commence par construire les tables  $\tau$  et  $\delta$  des fonctions  $\tau$  et  $\delta$  associées au motif  $m$ , puis on parcourt séquentiellement le texte  $t$  selon le programme suivant, dans lequel on dénote par  $l_{\text{motif}}$  et  $l_{\text{texte}}$  les longueurs respectives de  $m$  et  $t$  :

```

int etat = 0; // la position de départ de l'automate
for(int i = 0; i < T.length(); i++){
    if(M.charAt(etat+1) == T.charAt(i+1)){
        etat++;
        if(etat == M.length()){
            System.out.print("occurrence en :");
            System.out.println((i+1-M.length()));
        }
    }
    else
        etat = tau[delta[etat]][(int)T.charAt(i+1)];
}

```

**Exercice 10.1.1** Compléter le programme précédent pour en faire un vrai programme Java.

Le coût de construction des tables étant  $O(|m|)$  et celui d'exécution de la boucle de recherche étant trivialement  $O(|t|)$ , on en déduit l'assertion.

**Remarques :**

1. On peut ne pas précalculer les deux tables dans leur totalité, mais se contenter de les remplir au fur et à mesure des besoins en faisant un appel récursif si le résultat n'a pas été stocké dans la table auparavant. On économise ainsi du pré-traitement. Une telle technique de programmation est appelée *mémo-fonction*.
2. Lorsque l'alphabet est de grande cardinalité, par exemple ASCII 7 bits ou pire Unicode, la table  $\tau$  sert à moins de 1%, voire moins de  $10^{-4}$ . L'usage de mémo-fonctions est bien sûr une des réponses à cette situation ; on peut aussi convertir le problème en binaire et appliquer les méthodes précédentes dans le cas binaire sur des motif et texte un peu plus longs. Le lecteur trouvera matière à réflexion dans les exercices.

### 10.1.3 Multimotifs

Une première généralisation du problème est de rechercher un motif parmi plusieurs possibles. Peut-on gagner du temps en factorisant, pour ainsi dire, les recherches des divers motifs sur le même texte ?

La solution proposée est de nature semblable en compliquant légèrement la structure de l'automate du paragraphe précédent. Prenons un exemple avec pour ensemble de motifs  $M = \{ababb, aba, baa\}$ .

On construit alors l'automate donné en figure 10.2, sur lequel on remarquera que le squelette a une forme arborescente fournie par l'arbre binaire (incomplet) des motifs

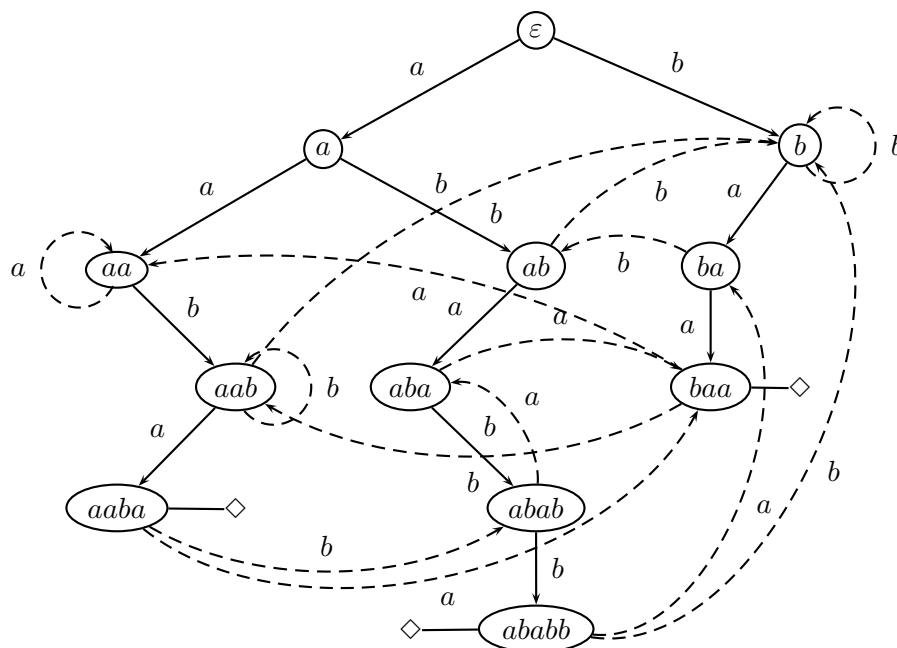


FIG. 10.2 – L’automate de reconnaissance du multimotif  $M = \{ababb, aaba, baa\}$  ; le squelette de l’automate est marqué en arcs pleins alors que les autres actions possibles le sont en arcs pointillés ; les  $\diamond$  marquent les états finaux.

de l’ensemble  $M$ . Cet arbre est incomplet car il ne possède que trois branches, correspondant chacune à l’un des motifs ; de façon générale, il est construit sur l’ensemble des facteurs gauches des mots de  $M$ , encore appelés *préfixes* de  $M$ , et noté  $FG(M)$ . Les sommets de l’arbre (qui sont aussi les états de l’automate) sont précisément les éléments de  $FG(M) = \{\varepsilon, a, b, aa, ab, ba, aab, aba, baa, aaba, abab, ababb\}$ , où  $\varepsilon$  désigne le mot vide, préfixe de tout mot.

Les trois états  $\{ababb, aaba, baa\}$  sont finaux car on n’y arrive que si l’on vient précisément de lire dans le texte un mot de  $M$ . Les autres arêtes sont calculées comme dans le cas précédent au moyen de la fonction  $\tau(f, \alpha)$  dont la valeur est le plus grand suffixe de  $f\alpha$  qui en soit aussi préfixe d’un mot de  $M$  :

$$\tau(f, \alpha) = \mu\{g \in FG(M) \mid \exists h, l \ f\alpha = gh = lg\},$$

définie sur  $FG(M) \times \{a, b\}$  et où  $\mu$  désigne l’opérateur *le plus long*. Remarquons que si  $f\alpha \in FG(M)$ , alors  $g = f\alpha$ .

Comme au paragraphe précédent on gagne du temps de calcul pour la fonction  $\tau$ , en utilisant la fonction auxiliaire  $\delta(f)$ , pour  $f \in FG(M)$ , qui calcule le plus grand suffixe strict de  $f$  qui soit aussi dans  $FG(M)$  et qui est définie par

$$\delta(f) = \mu\{g \in FG(M) \mid \exists h, l \neq \varepsilon \ f = gh = lg\}.$$

Cette fonction  $\delta$  permet de définir  $\tau$  et  $\delta$  par les récurrences conjointes, où  $f \in FG(M)$  et  $\alpha \in \{a, b\}$  :

$$\begin{aligned} \delta(\varepsilon) &= \varepsilon \\ \delta(f\alpha) &= \begin{array}{l} \text{si } |f| = \varepsilon \text{ alors } \varepsilon \\ \text{sinon } \tau(\delta(f), \alpha) \end{array} \\ \tau(f, \alpha) &= \begin{array}{l} \text{si } f\alpha \in FG(M) \text{ alors } f\alpha \\ \text{sinon si } f = \varepsilon \text{ alors } \varepsilon \\ \text{sinon } \tau(\delta(f), \alpha). \end{array} \end{aligned}$$

## 10.2 Automates finis

Nous présentons maintenant le formalisme général de cette classe de mécanisme de calcul simple. De fait, nous allons voir ces machines plus comme des machines à caractériser des familles de mots que comme des machines à calculer un résultat numérique ou symbolique. On dit que ces automates fonctionnent en *reconnaissance de langage*. Un des côtés les plus surprenants, et c'est un nouveau concept, est que ces machines sont *non-déterministes*, c'est-à-dire qu'elles sont susceptibles de plusieurs évolutions différentes dans le même environnement. Ce n'était pas le cas des deux exemples donnés dans la section précédente où le lecteur vérifiera qu'étant donné un texte le chemin suivi dans le graphe de l'automate est unique et pré-déterminé.

### 10.2.1 Définitions

Un automate fini agit sur les mots appartenant à l'ensemble  $A^*$  défini par l'alphabet fini  $A$  ; pour chaque mot, il permettra de déterminer s'il appartient ou non à l'ensemble caractérisé par l'automate, comme on le définit maintenant.

**Définition 10.2.1** *Un automate fini  $A$  sur un alphabet fini  $A$  est défini par un quintuplet*

$$\mathcal{A} = \langle A, Q, q_0, F, \Delta \rangle,$$

où  $Q$  est l'ensemble fini des états,  $q_0 \in Q$  est l'état initial, l'ensemble  $F \subset Q$  est celui des états finaux et  $\Delta$ , appelée table de transition est un ensemble fini

$$\Delta \subset Q \times (A \cup \{\varepsilon\}) \times Q.$$

*Si la table de transition est une relation fonctionnelle, on dit que l'automate fini est déterministe.*

Exemples :

1. L'automate de la figure 10.1 reconnaissant les mots contenant le motif *abaabbab* est défini comme suit :
  - alphabet :  $A = \{a, b\}$  ;
  - états :  $Q = \{0, 1, 2, 3, 4, 5, 6, 8\}$  ;

- état initial : 0 ;
- états finaux :  $F = \{8\}$  ;
- table :  $\Delta = \{(i, \alpha, j) \mid j = \tau(i, \alpha), 0 \leq i, j \leq 8, \alpha \in A\}$ .

2. L'automate de la figure 10.2 reconnaissant les mots contenant l'un des motifs de l'ensemble  $M = \{ababb, aaba, baa\}$  est défini comme suit :

- alphabet :  $A = \{a, b\}$  ;
- états :  $Q = FG(M) = \{\varepsilon, a, b, aa, ab, ba, aab, aba, baa, aaba, abab, ababb\}$  ;
- état initial :  $\varepsilon$  ;
- états finaux :  $F = \{aaba, baa, ababb\}$  ;
- table :  $\Delta = \{(f, \alpha, g) \mid g = \tau(f, \alpha), f, g \in Q, \alpha \in A\}$ .

Il faut noter pour ces deux automates que  $\Delta$  est en fait le graphe de la fonction  $\tau$  et donc qu'étant donnés  $f \in Q$  et  $\alpha \in A$ , il existe au plus un  $g$  tel que  $(f, \alpha, g) \in \Delta$ . L'automate est alors qualifié de *déterministe*, comme on le retrouvera plus bas.

3. Un automate pathologique : soit  $\mathcal{P}_7$  défini comme suit :

- alphabet :  $A = \{a, b\}$  ;
- états :  $Q = \{0, 1, 2, 3, 4, 5, 6, 7\}$  ;
- état initial : 0 ;
- états finaux :  $F = \{7\}$  ;
- table :  $\Delta = \{(0, a, 0), (0, b, 0), (0, a, 1)\} \cup \{(i, a, i + 1), (i, b, i + 1) \mid 1 \leq i < 7\}$ .

Graphiquement, il est sans doute plus facile de comprendre la structure de l'automate, comme on le voit sur la figure 10.3.

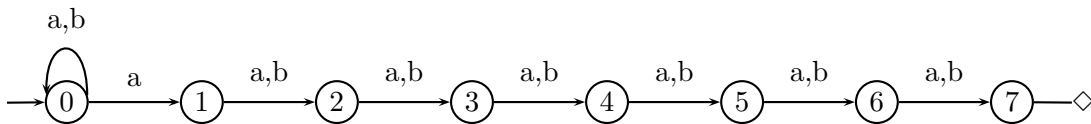


FIG. 10.3 – L'automate  $\mathcal{P}_7$

Cet automate est non-déterministe car il existe deux façons de réagir lorsqu'on est dans l'état 0 et qu'on lit la lettre  $a$  ; une flèche va vers l'état 1 tandis que l'autre va vers 0. Par ailleurs cet automate est incomplet, car quand on arrive en 7, il n'y a pas moyen de poursuivre, ce qui se traduit par le fait qu'il faut tomber juste pour pouvoir accepter un mot. Bien sûr, on n'est pas autorisé à revenir en arrière et il faut donc deviner le bon moment pour se lancer dans la suite de transitions menant de 0 à 7...

Avec un peu d'entraînement, il est facile de voir que cet automate reconnaît un mot si et seulement si il contient un  $a$  en 7<sup>e</sup> position avant sa fin...

### 10.2.2 Langage reconnu par un automate fini

On définit maintenant formellement l'ensemble des mots reconnus (ou acceptés) par un automate fini.

**Définition 10.2.2** On appelle configuration d'un automate  $\mathcal{A}$ , la donnée d'un mot  $w \in A^*$  sur son alphabet  $A$  et d'un état  $q \in Q$ . Une transition  $\delta$  de l'automate s'applique à une configuration  $(w, q)$  si et seulement si une des deux conditions est satisfaite :

- $\delta = (q, \varepsilon, q')$  et alors l'automate peut passer dans la configuration  $(w, q')$  ;
- $\delta = (q, \alpha, q')$  et  $w = \alpha v$  (commence donc par la lettre  $\alpha$ ) et alors l'automate peut passer dans la configuration  $(v, q')$  (en ayant donc lu la lettre  $\alpha$ ).

On notera  $(w, q) \xrightarrow{\delta} (w, q')$  ou  $(w, q) \xrightarrow{\delta} (v, q')$  l'application de la transition  $\delta$  à la configuration  $(w, q)$  et le lecteur remarquera bien qu'il s'agit d'évolution possible, mais non forcée. La notion de calcul se définit comme suit.

**Définition 10.2.3** On appelle calcul d'un automate  $\mathcal{A}$  toute suite de configurations  $c_0, c_1, \dots, c_k$ , telles que l'on peut passer de  $c_i$  à  $c_{i+1}$  par une transition de  $\mathcal{A}$ , pour  $0 \leq i < k$ .

Un calcul accepte un mot  $w$  si et seulement sa première configuration est  $c_0 = (w, q_0)$  et sa dernière est  $c_k = (\varepsilon, q)$ , pour quelque  $q \in F$  état final.

Remarquons donc qu'un calcul acceptant un mot de longueur  $n$  est de longueur au moins  $n$ , puisqu'il faut que l'automate lise toutes les lettres du mot. Sur l'exemple de la figure 10.3 le mot  $f = \text{abbababbaaab}$  admet plusieurs suites de calcul :

- $C_1 : (\text{abbababbaaab}, 0) \rightarrow (\text{bbababbaaab}, 1) \rightarrow (\text{bababbaaab}, 2) \rightarrow (\text{ababbaaab}, 3) \rightarrow (\text{babbaaab}, 4) \rightarrow (\text{abbaaab}, 5) \rightarrow (\text{bbaaab}, 6) \rightarrow (\text{baaab}, 7)$  qui se bloque car il n'y a plus de transition applicable dans l'état 7 et le mot d'entrée n'a pas été lu entièrement ;
- $C_2 : (\text{abbababbaaab}, 0) \rightarrow (\text{bbababbaaab}, 0) \rightarrow (\text{bababbaaab}, 0) \rightarrow (\text{ababbaaab}, 0) \rightarrow (\text{babbaaab}, 0) \rightarrow (\text{abbaaab}, 0) \rightarrow (\text{bbaaab}, 1) \rightarrow (\text{baaab}, 2) \rightarrow (\text{aaab}, 3) \rightarrow (\text{aab}, 4) \rightarrow (\text{ab}, 5) \rightarrow (\text{b}, 6) \rightarrow (\varepsilon, 7)$  qui accepte donc le mot  $f$  ;
- $C_3 : (\text{abbababbaaab}, 0) \rightarrow (\text{bbababbaaab}, 0) \rightarrow (\text{bababbaaab}, 0) \rightarrow (\text{ababbaaab}, 0) \rightarrow (\text{babbaaab}, 0) \rightarrow (\text{abbaaab}, 0) \rightarrow (\text{bbaaab}, 0) \rightarrow (\text{baaab}, 0) \rightarrow (\text{aaab}, 1) \rightarrow (\text{aab}, 2) \rightarrow (\text{ab}, 3) \rightarrow (\text{b}, 4) \rightarrow (\varepsilon, 5)$  qui s'arrête dans un état non final.

On dira que le mot  $f = \text{abbababbaaab}$  est reconnu par  $\mathcal{A}$ , car un des calculs au moins l'accepte. Ceci correspond donc au fait qu'il existe une manière correcte de l'analyser.

**Définition 10.2.4** Un mot  $w$  est reconnu par un automate  $\mathcal{A}$ , s'il existe au moins un calcul de  $\mathcal{A}$  sur  $w$  qui l'accepte. On appelle langage reconnu par un automate  $\mathcal{A}$ , et on le note  $\mathcal{L}(\mathcal{A})$ , l'ensemble des mots reconnus par l'automate.

Si l'automate  $\mathcal{A}$  est déterministe, il existe un seul calcul possible par mot et par conséquent on détermine l'appartenance d'un mot quelconque  $w$  au langage  $\mathcal{L}(\mathcal{A})$  en un temps  $O(|w|)$ . Si, au contraire, l'automate est non-déterministe, il faut trouver la

bonne analyse (si elle existe) qui permettra de décider positivement de l'appartenance de  $w$  au langage reconnu par  $\mathcal{A}$ .

Voici une autre façon de définir l'ensemble reconnu par un automate fini. Le lecteur pourra reconnaître des idées et concepts déjà vus en algorithmique des graphes.

Définissons pour chaque état  $q \in Q$  le langage  $L_q$  comme étant l'ensemble des mots permettant d'atteindre l'état  $q$  à partir de l'état initial  $q_0$ . Les transitions de l'automate du type  $(q, \alpha, q')$ , pour  $q, q' \in Q$  et  $\alpha \in A \cup \{\varepsilon\}$ , peuvent être interprétées sur le graphe de l'automate comme des arêtes entre états ( $q$  et  $q'$ ) étiquetées par la lettre  $\alpha$  de  $A$  ou le mot vide  $\varepsilon : q \xrightarrow{\alpha} q'$ . Alors

$$L_q = \{w \mid \exists n \exists q_1, \dots, q_{n+1} = q \in Q, w_0, w_1, \dots, w_n \in A \cup \{\varepsilon\}, \\ q_0 \xrightarrow{w_0} q_1 \xrightarrow{w_1} \dots q_n \xrightarrow{w_n} q_{n+1} = q \ \& \ w = w_0 w_1 \dots w_n\}$$

On doit noter ici le rôle que peuvent jouer les  $\varepsilon$ , la longueur de  $w$  n'étant pas forcément  $n + 1$ . Les  $L_q$  se définissent récursivement comme le montre la proposition suivante.

**Proposition 10.2.1** *Les langages  $L_q$ ,  $q \in Q$ , sont définis par*

$$L_q = K_q \cup \{w\alpha \mid \exists q' \in Q, w \in L_{q'} \ \& \ (q', \alpha, q) \in \Delta\},$$

où  $K_{q_0} = \{\varepsilon\}$  et  $K_q = \emptyset$  pour  $q \neq q_0$ .

Preuve : Si  $w$  est un mot de  $L_{q'}$  et si  $(q', \alpha, q) \in \Delta$  est une transition de l'automate alors, il est évident que  $w\alpha$  est un élément de  $L_q$ , d'après sa définition ; donc  $L_q \subset \{w\alpha \mid \exists q' \in Q, \exists w \in L_{q'}, (q', \alpha, q) \in \Delta\}$ , lorsque  $q \neq q_0$ , et  $L_{q_0} \subset \{\varepsilon\} \cup \{w\alpha \mid \exists q' \in Q, \exists w \in L_{q'}, (q', \alpha, q_0) \in \Delta\}$ , car  $\varepsilon \in L_{q_0}$  par construction (et pour amorcer le dispositif).

Réciproquement, si  $w$  est un mot de  $L_q$ , c'est que lui-même ou son facteur gauche  $v$  défini par  $w = v\alpha$ ,  $\alpha \in A$ , appartiennent à un  $L_{q'}$  et vérifient la propriété ci-dessus. D'où la proposition.

Remarquons de plus que ces ensembles sont parfaitement bien définis par récurrence sur la longueur des chemins dans le graphe sous-jacent de l'automate, chemins que l'on peut par exemple exprimer au moyen des puissances de la matrice d'adjacence du graphe, déduite de la table de transitions. Il s'agit des plus petits ensembles satisfaisant la récurrence ; dans certains cas des sur-ensembles de ces ensembles peuvent aussi fournir une solution, mais nous ne la considérerons pas.

Sur ces bases, on pose alors la définition du langage reconnu par l'automate (équivalente à celle donnée informellement plus haut).

**Définition 10.2.5** *Le langage reconnu par un automate  $\mathcal{A}$  est la réunion des langages*

$$\mathcal{L}(\mathcal{A}) = \cup_{q \in F} L_q.$$

Exemple : Pour l'automate des multi-motifs de la figure 10.2, l'ensemble  $\mathcal{L}(\mathcal{A}) = L_{baa} \cup L_{aaba} \cup L_{ababb}$  est l'ensemble des mots *se terminant* par l'un des motifs de  $M = \{baa, aaba, ababb\}$ .

Comment faudrait-il le modifier pour qu'il reconnaisse l'ensemble de mots *contenant* l'un de ces motifs ?

**Remarques :**

1. Un mot est donc reconnu par un automate si et seulement s'il étiquette un chemin menant de l'état initial à un état final.
2. On peut aussi voir la définition des  $L_q$  comme un système d'équations ensemblistes, dont les  $L_q$  forment la plus petite solution vis-à-vis de l'inclusion. Cette idée sera exploitée au chapitre suivant.
3. Les automates sont a priori non-déterministes. Si la relation  $\Delta$  est fonctionnelle, c'est-à-dire si  $(q, \alpha, q'), (q, \alpha, q'') \in \Delta \Rightarrow q' = q''$ , alors l'automate est appelé déterministe, comme noté plus haut : il n'y a alors pas d' $\varepsilon$ -transitions.

Pour terminer ce paragraphe donnons deux propriétés élémentaires des langages reconnaissables par automates finis.

**Propriétés élémentaires :**

1. Tout ensemble fini est reconnu par un automate fini : il suffit de construire l'arbre des préfixes des mots du langage.
2. Le sens de lecture n'importe pas : si  $L = \mathcal{L}(\mathcal{A})$  est reconnu par l'automate fini  $\mathcal{A}$ , alors il existe un automate  $\mathcal{B}$  qui reconnaît  $L^R = \{w^R \mid w \in L\}$ <sup>1</sup> La construction de  $\mathcal{B}$  consiste à renverser la table de transition de l'automate, comme on le verra en exercice.

### 10.2.3 Combinaisons d'automates

Appelons  $\mathcal{Aut}$  la famille des langages reconnus par des automates finis. Nous avons déjà vu deux des propriétés suivantes.

**Proposition 10.2.2** *La classe  $\mathcal{Aut}$  des langages reconnus par automate fini contient les langages finis  $\mathcal{Fin}$ , et est close par l'opérateur miroir  $L^R$ . De plus elle est dénombrable.*

Nous étudions maintenant les opérations ensemblistes classiques. La classe est close par toutes les opérations classiques, mais nous ne traitons pour le moment la complémentation que dans un cas particulier.

**Proposition 10.2.3** *La classe  $\mathcal{Aut}$  est close par union, concaténation, quasi-inverse, intersection, morphisme et morphisme inverse, complémentation.*

Preuve : Nous allons construire explicitement des automates réalisant ces opérations.

1. Union : Soient  $\mathcal{A} = \langle \Sigma, Q^A, q_0^A, F^A, \Delta^A \rangle$  et  $\mathcal{B} = \langle \Sigma, Q^B, q_0^B, F^B, \Delta^B \rangle$  deux automates reconnaissant les langages  $\mathcal{L}^A$  et  $\mathcal{L}^B$  respectivement. Pour simplifier, et sans aucune

<sup>1</sup>On rappelle que la notation  $w^R$  désigne le mot  $w$  lu de droite à gauche de telle sorte que si  $w = w_1w_2\dots w_{n-1}w_n$ , on a  $w^R = w_nw_{n-1}\dots w_2w_1$ .

restriction, on peut supposer que les ensembles d'états  $Q^A$  et  $Q^B$  sont disjoints, ce qu'on peut toujours réaliser par renommage.

On construit alors un automate  $\mathcal{C}$  qui reconnaît  $\mathcal{L}^C = \mathcal{L}^A \cup \mathcal{L}^B$  comme suit :

- alphabet  $\Sigma$  ;
- états  $Q^C = \{q_0^C\} \cup Q^A \cup Q^B$  ;
- état initial  $q_0^C$  ;
- états finaux  $F^C = F^A \cup F^B$  ;
- transitions  $\Delta^C = \{(q_0^C, \varepsilon, q_0^A), (q_0^C, \varepsilon, q_0^B)\} \cup \Delta^A \cup \Delta^B$ .

L'automate  $\mathcal{C}$  contient donc deux copies disjointes de  $\mathcal{A}$  et de  $\mathcal{B}$  ; de plus les deux  $\varepsilon$ -transitions supplémentaires permettent de se brancher à volonté sur l'une de ces copies et par conséquent de reconnaître précisément  $\mathcal{L}^A \cup \mathcal{L}^B$ .

2. Concaténation : On veut donc reconnaître le langage  $\mathcal{L}^C = \mathcal{L}^A \cdot \mathcal{L}^B = \{fg \mid f \in \mathcal{L}^A \ \& \ g \in \mathcal{L}^B\}$ , les automates  $\mathcal{A}$  et  $\mathcal{B}$  étant définis comme précédemment. L'automate  $\mathcal{C}$  qui reconnaît  $\mathcal{L}^C$  est défini comme suit :

- alphabet  $\Sigma$  ;
- états  $Q^C = Q^A \cup Q^B$  ;
- état initial  $q_0^A$  ;
- états finaux  $F^B$  ;
- transitions  $\Delta^C = \Delta^A \cup \Delta^B \cup \{(q_f, \varepsilon, q_0^B) \mid q_f \in F^A\}$ .

Ainsi, une suite de calculs acceptant un mot  $w$  pour  $\mathcal{C}$  commence par reconnaître un mot de  $\mathcal{L}^A$ , puisqu'elle commence par l'état  $q_0^A$  et passe nécessairement par un état final de  $\mathcal{A}$  — seule façon de sauter vers un état final en passant dans  $\mathcal{B}$  en ayant, ce faisant, lu un facteur gauche  $u$  de  $w$  — puis lit le facteur droit restant  $v$  qui est précisément reconnu par  $\mathcal{B}$  puisque la suite de calculs commence après l' $\varepsilon$ -transition  $(q_f, \varepsilon, q_0^B)$  par l'état initial de  $\mathcal{B}$  et se termine sur un de ses états finaux.

La réciproque est triviale par construction.

3. Quasi-inverse : Cette opération ensembliste est très importante en informatique ainsi qu'en combinatoire des mots. C'est l'opération de fermeture des monoïdes définie en début de chapitre, mais on l'applique désormais à tout ensemble de mots (fini ou infini) :  $\mathcal{X}^* = \{\varepsilon\} \cup \mathcal{X} \cup \mathcal{X}^2 \cup \dots \cup \mathcal{X}^k \cup \dots = \cup_{k \geq 0} \mathcal{X}^k$ .<sup>2</sup>

Toujours avec les mêmes notations, construisons l'automate  $\mathcal{C}$  qui doit reconnaître le langage  $\mathcal{L}^C = \mathcal{L}^{A^*}$  :

- alphabet  $\Sigma$  ;
- états  $Q^C = Q^A \cup \{q_0^C\}$  ;
- état initial  $q_0^C$  ;
- états finaux  $\{q_0^C\}$  ;
- transitions  $\Delta^C = \Delta^A \cup \{(q_0^C, \varepsilon, q_0^A)\} \cup \{(q_f, \varepsilon, q_0^C) \mid q_f \in F^A\}$ .

L'introduction de l'état  $q_0^C$ , à la fois initial et final, permet de prendre en compte le mot vide  $\varepsilon$  qui appartient au langage  $\mathcal{L}^C$  indépendamment de son appartenance à  $\mathcal{L}^A$  ;

<sup>2</sup>Une remarque importante est que le monoïde  $\mathcal{X}^*$  ainsi défini n'a plus de raisons d'être libre, ce qui traduit le fait que la décomposition d'un mot de  $\mathcal{X}^*$  en facteurs appartenant à  $\mathcal{X}$  n'est pas forcément unique, un même mot pouvant avoir plusieurs scansions.

de plus cet état sert de point relai pour séparer chacun des facteurs d'un mot  $w \in \mathcal{L}^{\mathcal{C}}$  qui doivent appartenir à  $\mathcal{L}^{\mathcal{A}}$ . Un mot est reconnu par l'automate  $\mathcal{C}$  si et seulement s'il est la concaténation de mots reconnus par  $\mathcal{A}$ .

4. Intersection : Soient donc  $\mathcal{L}^{\mathcal{A}}$  et  $\mathcal{L}^{\mathcal{B}}$  les langages reconnus par les automates  $\mathcal{A}$  et  $\mathcal{B}$ , définis comme précédemment. Construisons un automate  $\mathcal{C}$  qui va réaliser le produit de ces deux automates en permettant de rendre leur comportement synchrone et donc de les faire marcher en parallèle. Pour ce faire on introduit des  $\varepsilon$ -transitions qui permettent de boucler arbitrairement sur chaque état de  $\mathcal{A}$  et  $\mathcal{B}$  sans pour autant lire de lettre de la donnée, et même mieux, de pouvoir changer d'état sur l'un sans modifier l'état de l'autre.

Formellement, on ajoute donc respectivement aux ensembles  $\Delta^{\mathcal{A}}$  et  $\Delta^{\mathcal{B}}$  les transitions  $\{(q, \varepsilon, q) \mid \forall q \in Q^{\mathcal{A}}\}$  et  $\{(q, \varepsilon, q) \mid \forall q \in Q^{\mathcal{B}}\}$ . Ce faisant on ne change en rien les langages  $\mathcal{L}^{\mathcal{A}}$  et  $\mathcal{L}^{\mathcal{B}}$  reconnus par ces automates modifiés que nous continuons d'appeler  $\mathcal{A}$  et  $\mathcal{B}$ , pour ne pas alourdir les notations.

On a alors formellement pour l'automate  $\mathcal{C}$  :

- alphabet  $\Sigma$  ;
- états  $Q^{\mathcal{C}} = Q^{\mathcal{A}} \times Q^{\mathcal{B}}$  ;
- état initial  $(q_0^{\mathcal{A}}, q_0^{\mathcal{B}})$  ;
- états finaux  $F^{\mathcal{C}} = F^{\mathcal{A}} \times F^{\mathcal{B}}$  ;
- transitions  $\Delta^{\mathcal{C}} = \{((q, r), \alpha, (q', r')) \mid \forall q, q', \alpha, r, r' : (q, \alpha, q') \in \Delta^{\mathcal{A}} \& (r, \alpha, r') \in \Delta^{\mathcal{B}}\}$ .

On voit donc que si  $\mathcal{C}$  reconnaît un mot  $w$ , alors les deux projections du calcul sur les composantes  $\mathcal{A}$  et  $\mathcal{B}$  sont séparément des calculs valides de ces deux automates et donc  $w \in \mathcal{L}^{\mathcal{A}} \cap \mathcal{L}^{\mathcal{B}}$ .

Inversement, soit  $w$  appartenant à ces deux langages ; il est donc reconnu par deux séquences de calcul  $\sigma^{\mathcal{A}}$  et  $\sigma^{\mathcal{B}}$ . Supposons qu'aucune des deux ne contienne d' $\varepsilon$ -transition : elles ont donc la même longueur et lisent la même succession de lettres (celles de  $w$ ) et, en conséquence, on peut les superposer et en faire le produit, transition par transition. De  $q_0 \xrightarrow{w_0} q_1 \xrightarrow{w_1} \dots q_n \xrightarrow{w_n} q_{n+1}$  pour  $\mathcal{A}$  et  $r_0 \xrightarrow{w_0} r_1 \xrightarrow{w_1} \dots r_n \xrightarrow{w_n} r_{n+1}$  pour  $\mathcal{B}$ , on obtient ainsi une séquence de calcul pour  $\mathcal{C}$  :  $(q_0, r_0) \xrightarrow{w_0} (q_1, r_1) \xrightarrow{w_1} \dots (q_n, r_n) \xrightarrow{w_n} (q_{n+1}, r_{n+1})$ , qui accepte  $w$  si et seulement si les deux séquences le faisaient.

Si l'une ou l'autre (ou les deux) des séquences de  $\mathcal{A}$  et  $\mathcal{B}$  contiennent des  $\varepsilon$ -transitions, il suffit (après avoir mis en correspondance les transitions lisant les lettres homologues de  $w$ ) de placer en vis-à-vis dans l'autre séquence une  $\varepsilon$ -transition qui ne change pas l'état (ce sont précisément celles qui furent ajoutées en début de construction). Ainsi réalise-t-on l'automate produit qui reconnaît l'intersection des deux langages.

5. Morphisme et morphisme inverse : un morphisme  $h$  est une application d'un ensemble de mots  $\mathcal{A}^*$  dans un autre  $\mathcal{B}^*$  et qui est défini par sa restriction —  $h : \mathcal{A} \rightarrow \mathcal{B}^*$  — à l'alphabet  $\mathcal{A}$ , puis étendu à tout  $\mathcal{A}^*$  par la règle  $h(f.g) = h(f).h(g)$ .

Montrons que si  $\mathcal{L}^{\mathcal{A}}$  est reconnu par  $\mathcal{A}$ , il existe un automate  $\mathcal{C}$  qui reconnaît  $H = h(\mathcal{L}^{\mathcal{A}})$ . L'automate  $\mathcal{C}$  va reprendre la structure de l'automate  $\mathcal{A}$  en remplaçant chaque transition  $(q, \alpha, q')$ , telle que  $h(\alpha) = \beta_1\beta_2\dots\beta_k \neq \varepsilon$  par l'ensemble de transitions :  $\{(q, \beta_1, r_1^{(q, \alpha, q')}), (r_1^{(q, \alpha, q')}, \beta_2, r_2^{(q, \alpha, q')}), \dots, (r_{k-1}^{(q, \alpha, q')}, \beta_k, q')\}$ . Si le morphisme efface  $\alpha$  (c'est-est-

à-dire  $h(\alpha) = \varepsilon$ ), on remplace la transition par  $(q, \varepsilon, q')$ .

Il est facile de voir qu'en prenant des nouveaux états tous distincts, la structure de  $\mathcal{A}$  est maintenue dans  $\mathcal{C}$  et qu'un mot est accepté par  $\mathcal{C}$  si et seulement s'il est l'image d'un mot accepté par  $\mathcal{A}$ .

Pour le morphisme inverse (défini par  $\mathcal{K} = h^{-1}(\mathcal{L}) = \{w \mid h(w) \in \mathcal{L}\}$ ), il faut construire un automate *universel*, capable de simuler tout automate fini donné par sa table, ce qui est la démarche symétrique de ce qui vient d'être fait. Voir les exercices pour des éléments de solution.

6. Complémentation (cas simplifié) :

Considérons le langage  $\mathcal{L}^{\mathcal{A}}$  reconnu par  $\mathcal{A}$  et faisons sur cet automate les hypothèses supplémentaires suivantes (dont on montrera par la suite qu'elles peuvent toujours être satisfaites) :

- $\mathcal{A}$  est déterministe et sans  $\varepsilon$ -transition ;
- $\mathcal{A}$  est complet, en ce sens que, pour tout état  $q \in Q^{\mathcal{A}}$  et toute lettre de l'alphabet  $\alpha \in \Sigma$ , il existe un état  $q'$  unique (car  $\mathcal{A}$  est déterministe) et une transition  $(q, \alpha, q')$  correspondante dans la table  $\Delta^{\mathcal{A}}$ .

On a la propriété fondamentale suivante. Soit  $w$  un mot quelconque :

— puisque l'automate est complet, il existe au moins un chemin  $q_0 \xrightarrow{w} q$  étiqueté par  $w$  ;

– puisque l'automate est complet et sans  $\varepsilon$ -transition, ce chemin est unique.

En conséquence, pour chaque mot  $w$ , il y a existence et unicité de l'état produit par la lecture de  $w$  à partir de l'état initial  $q_0^{\mathcal{A}}$  et dont l'appartenance à  $F^{\mathcal{A}}$  détermine l'appartenance de  $w$  à  $\mathcal{L}^{\mathcal{A}}$ . Soit  $\mathcal{C}$  l'automate obtenu, à partir de  $\mathcal{A}$ , en prenant  $F^{\mathcal{C}} = Q^{\mathcal{A}} \setminus F^{\mathcal{A}}$  ; cet automate reconnaît exactement le complémentaire des mots acceptés par  $\mathcal{A}$  :  $\mathcal{L}(\mathcal{C}) = Q \setminus \mathcal{L}(\mathcal{A})$ .

On prouvera un peu plus tard la fermeture par complémentation en toute généralité.

## 10.3 Expressions régulières et langages rationnels

Les automates procurent un cadre effectif et calculatoire pour définir des classes de motifs, mais ils ne donnent pas nécessairement et directement une idée de la nature combinatoire de ces motifs. Une bien meilleure formulation est fournie par ce que l'on appelle expressions régulières dans la littérature anglo-saxonne (*regular expressions*) et langages rationnels dans la tradition française. Ces expressions régulières sont un des outils privilégiés de description de motifs utilisés dans les bases de données, les moteurs de recherche du Web et des systèmes documentaires. Ces concepts font donc partie de la culture de base de tout utilisateur éclairé et a fortiori de celle des concepteurs avancés.

### 10.3.1 Langages rationnels : définitions

La famille des langages rationnels se définit très simplement et de façon minimale comme le montre la définition ci-dessous. Cependant on verra qu'on peut étendre les

outils de description en augmentant ainsi la lisibilité de la définition sans pour autant sortir de la classe des langages rationnels.

**Définition 10.3.1** *Soit un alphabet fini  $A$ . On appelle famille des langages rationnels sur l'alphabet  $A$  la plus petite famille de sous-ensembles de  $A^*$  contenant les ensembles finis et close par union, produit de concaténation et quasi-inverse : on la note  $\mathcal{Rat}$ .*

Cette définition concise a été reprise dans le monde de l'informatique sous le nom d'expressions régulières (*regular expressions* en anglais) et figure dans tous les langages de programmation de systèmes : Unix, DOS, VMS, etc.

L'appendice reprend une partie du manuel Linux qui définit les expressions régulières de la commande GNU-grep.

Nous allons considérer maintenant la version minimale, noyau du concept, qui définit les expressions régulières comme des expressions algébriques sur le monoïde des mots construits sur un alphabet  $A$ .

**Définition 10.3.2** *On appelle expression régulière — et on note  $\mathcal{Reg}$  leur ensemble — toute formule composée aux moyens des opérations suivantes, à partir d'un ensemble fini  $A$  appelé alphabet, des opérateurs  $+$ ,  $\cdot$ ,  $*$  (les deux premiers sont binaires et le troisième unaire), ainsi que des parenthèses :*

- tout élément de  $A$  est dans  $\mathcal{Reg}$  ;
- pour toute paire  $e_1, e_2 \in \mathcal{Reg}$ , alors  $e_1 + e_2 \in \mathcal{Reg}$  et  $e_1.e_2 \in \mathcal{Reg}$  ;
- pour toute  $e \in \mathcal{Reg}$ , alors  $(e) \in \mathcal{Reg}$  et  $(e)^* \in \mathcal{Reg}$  ;
- l'ensemble vide est noté  $\varepsilon$ .

Le lecteur notera que les parenthèses permettent de lever les ambiguïtés ; on utilisera aussi le fait que l'opération de quasi-inverse  $*$  est prioritaire sur l'opération de concaténation  $\cdot$  qui est elle-même prioritaire sur l'union  $+$ . On supprimera généralement les parenthèses autour d'une lettre unique ainsi que les  $\cdot$  et on écrira par exemple  $a^*ba^*$  au lieu de  $((a)^*) \cdot (b)((a)^*)$ .

Chaque expression régulière  $e$  définit un sous-ensemble — noté  $\mathcal{L}(e)$  et appelé langage associé à  $e$  — de l'ensemble  $A^*$  des mots sur l'alphabet  $A$ , comme explicité ci-dessous :

- $\mathcal{L}(\varepsilon) = \{\varepsilon\}$  ;
- $\mathcal{L}(a) = \{a\}$ , pour toute lettre  $a \in A$  ;
- $\mathcal{L}(e_1 + e_2) = \{w \mid w \in \mathcal{L}(e_1) \vee w \in \mathcal{L}(e_2)\}$  ;
- $\mathcal{L}(e_1.e_2) = \{w_1.w_2 \mid w_1 \in \mathcal{L}(e_1) \ \& \ w_2 \in \mathcal{L}(e_2)\}$  ;
- $\mathcal{L}(e^*) = \{\varepsilon\} \cup \mathcal{L}(e) \cup (\mathcal{L}(e))^2 \cup (\mathcal{L}(e))^3 \cup \dots$

On a donc de toute évidence  $\mathcal{L}(\mathcal{Reg}) = \mathcal{Rat}$ , puisqu'il ne s'agit que d'une question de formulation.

On a de plus et de façon élémentaire le premier pas de caractérisation calculatoire de cette définition ensembliste.

**Proposition 10.3.1** *Les langages définis par des expressions régulières sont reconnaissables par automates finis :*

$$\mathcal{Rat} \subset \mathcal{Aut}.$$

Preuve : Elle reprend en grande partie les constructions de la proposition 2.5 et est basée sur une décomposition récursive de l'expression régulière.

1. Si la formule est une lettre, un mot ou encore un ensemble fini, on construit l'automate filiforme ou arborescent qui reconnaît la lettre, le mot ou l'ensemble fini, comme ceci fut fait dans la construction du squelette de l'automate KMP en première partie de chapitre.

2. Si la formule est de type  $(e_1 + e_2)$  ou  $(e_1.e_2)$  ou  $(e_1)^*$ , on utilise de façon récursive les automates  $\mathcal{A}_1$  et  $\mathcal{A}_2$  construits pour les expressions  $e_1$  et  $e_2$  et on leur applique les constructions de la proposition 2.5.

### 10.3.2 La réciproque : le théorème de Kleene

La propriété remarquable de cette section est que la réciproque est vraie, et donc que l'on dispose d'une double caractérisation pour ces ensembles de mots : ensembliste comme nous venons de le faire et calculatoire au moyen des automates finis. La preuve est due à S. C. Kleene et repose sur une méthode d'énumération nouvelle et très intéressante des chemins dans un graphe, que l'on peut exprimer comme suit : montrer que pour tout automate fini  $\mathcal{A}$  d'ensemble d'états  $Q$ , les langages<sup>3</sup>  $L_q$  des mots qui étiqùent les chemins dans le graphe de l'automate sont tous rationnels.

De plus on sait produire algorithmiquement une expression régulière pour ces langages comme le montre la proposition suivante.

**Proposition 10.3.2** *Étant donné un automate fini  $\mathcal{A}$  et un état  $q$  de  $\mathcal{A}$ , soit  $L_q$  l'ensemble des mots des mots permettant d'atteindre l'état  $q$  à partir de l'état initial  $q_0$ . Alors  $L_q \in \mathcal{Rat}$ .*

L'algorithme ci-dessous fournit une expression régulière pour le langage  $L_q$ . Preuve :

0. Soit  $\mathcal{A} = \langle A, Q, q_*, F, \Delta \rangle$  l'automate, dont on suppose seulement qu'il est sans  $\varepsilon$ -transitions ; il est donc non-déterministe a priori.

1. Ranger les états de  $Q$  dans un ordre quelconque :  $Q = \{q_1, q_2, \dots, q_n\}$ . L'état initial  $q_*$  est l'un de ces états mais son numéro n'a aucune importance.

Poser

$$\begin{aligned} Q_0 &= \emptyset \\ Q_1 &= \{q_1\} \\ Q_2 &= \{q_1, q_2\} \\ &\dots \\ Q_n &= \{q_1, q_2, \dots, q_n\} \end{aligned}$$

2. Initialisation :

Pour  $1 \leq i, j \leq n$  faire

$$X_{i,j}^{Q_0} = \{\alpha \in A \mid (q_i, \alpha, q_j) \in \Delta\}.$$

<sup>3</sup>Revoir la définition formelle au § 2.2.2.

On remarquera que les  $X_{i,j}^{Q_0}$  sont des ensembles finis que l'on représentera trivialement par l'expression régulière équivalente :  $\{\alpha_1, \alpha_2, \dots, \alpha_k\} \equiv (\alpha_1 + \alpha_2 + \dots + \alpha_k)$ .

3. Itération :

Pour  $k = 1, n, 1$  faire Pour  $1 \leq i, j \leq n$  faire

$$X_{i,j}^{Q_k} = X_{i,j}^{Q_{k-1}} + X_{i,k}^{Q_{k-1}} (X_{k,k}^{Q_{k-1}})^* X_{k,j}^{Q_{k-1}}.$$

Considérons les  $X_{i,j}^{Q_1} = X_{i,j}^{Q_0} + X_{i,1}^{Q_0} (X_{1,1}^{Q_0})^* X_{1,j}^{Q_0}$  ; comme  $X_{i,j}^{Q_0}$  est l'ensemble des lettres qui permettent de passer de  $q_i$  à  $q_j$  par une transition de l'automate, on en déduit que  $X_{i,j}^{Q_1}$  est l'ensemble des mots qui amènent de  $q_i$  à  $q_1$ , puis bouclent sur  $q_1$  et vont en un coup de  $q_1$  à  $q_j$ .

De même, on interprète  $X_{i,j}^{Q_2} = X_{i,j}^{Q_1} + X_{i,2}^{Q_1} (X_{2,2}^{Q_1})^* X_{2,j}^{Q_1}$  comme étant l'ensemble des étiquettes des chemins qui vont de  $q_i$  à  $q_2$  (premier passage) en passant par  $q_1$ , puis bouclent sur  $q_2$  en passant par  $q_1$  et enfin vont de  $q_2$  (dernier passage) à  $q_j$  en passant par  $q_1$ . Remarquons que tout chemin allant de  $q_i$  à  $q_j$  en passant uniquement par  $q_1$  et  $q_2$  peut être ainsi décomposée et que  $X_{i,j}^{Q_2}$  les contient donc tous au travers de l'expression régulière.

Et donc par récurrence,  $X_{i,j}^{Q_k}$  contient tous les mots étiquetant des chemins allant de  $q_i$  à  $q_j$  en passant par les sommets de l'ensemble  $Q_k$ .

4. En conclusion,  $X_{i,j}^{Q_n}$  contient tous les mots étiquetant des chemins allant de  $q_i$  à  $q_j$  en passant par les sommets de l'ensemble  $Q_n$  et par conséquent représente toutes les façons de passer de  $q_i$  à  $q_j$ . Comme cas particulier, on a donc pour tout état  $q \in Q$ ,

$$L_q = X_{q^*,q}^{Q_n}.$$

En conséquence tout automate fini produit uniquement des langages rationnels, expressibles donc par expression régulière. D'où le théorème annoncé.

**Théorème 10.3.1** [Kleene] *Les ensembles rationnels sont identiques aux ensembles reconnaissables par automate fini :*

$$\mathcal{A}ut = \mathcal{R}at.$$

Le nombre de phases de calcul de cet algorithme est  $n = |Q|$ , chaque phase comportant le calcul de  $n^2$  expressions ; on pourrait donc en déduire que le temps de calcul est polynomial. Or il importe de noter que la taille de ces expressions peut quadrupler à chaque phase, ce qui fait que la longueur de l'expression pour  $X_{i,j}^{Q_k}$  est  $O(4^k \cdot |A|)$  et donc que si l'on procède par recopie le temps de calcul sera  $O(n^2 4^n)$ ... De même, le simple fait d'écrire le résultat peut être exponentiel en  $|Q|$ .

## 10.4 Déterminisme contre Non-Déterminisme

La définition des automates finis comme reconnaisseurs non-déterministes est fondamentale du point de vue de leurs propriétés de composition et de pouvoir expressif.

Cependant, il faut bien reconnaître que du point de vue pratique de l'informaticien qui *calcule* le non-déterminisme est source de frustration et d'inefficacité. Nous allons développer plusieurs réponses à cette critique, visant à rendre déterministe le calcul du résultat attendu de l'automate non-déterministe.

### 10.4.1 Simulation du non-déterminisme

Considérons un automate fini  $\mathcal{A} = \langle A, Q, q_*, F, \Delta \rangle$ , dont on suppose seulement qu'il est sans  $\varepsilon$ -transitions, pour simplifier. Le mécanisme de l'automate fini est simulé au moyen d'une mémoire de taille  $Q$  qui contiendra à chaque pas de simulation le sous-ensemble des états accessibles depuis l'état initial. Plus précisément, si après avoir lu la  $k^{\text{e}}$  lettre de la donnée  $w$ , l'ensemble des états accessibles est

$$H_k = \{q \mid q_* \xrightarrow{w_1 \dots w_k} q\}$$

et si  $\alpha$  est la lettre suivante de  $w$ , l'ensemble  $H_{k+1}$  est défini par :

$$H_{k+1} = \{q' \mid \exists q \in H_k \ \& \ (q, \alpha, q') \in \Delta\}.$$

Le programme qui effectue la simulation sur un mot  $w$ , vu comme une liste de lettres, a donc la forme suivante :

```

H <- {q*}; // état initial
répéter
  l <- tête(w); // la première lettre de w
  w <- queue(w); // qui est supprimée
  K <- {};
  pour q dans H faire
    pour (q, l, qq) dans Delta faire
      K <- K ∪ {qq};
    H := K; // recopie
jusqu'à ce que w == NIL;
si H contient un état de F alors
  accepter(w);

```

Ce programme opère en  $O(|w| \cdot |\Delta|)$  opérations, et même un peu moins à condition de pouvoir faire une recherche rapide des transitions applicables à l'ensemble  $H$  courant.

**Proposition 10.4.1** *La simulation d'un automate non-déterministe  $\mathcal{A}$  sur un mot de longueur  $n$  se fait en temps  $O(n \cdot |\mathcal{A}|)$ .*

### 10.4.2 Suppression des $\varepsilon$ -transitions

Nous avons plusieurs fois fait l'hypothèse simplificatrice que l'automate ne comportait pas de transition du type  $(q, \varepsilon, q')$ . Nous montrons dans ce paragraphe comment on peut effectivement transformer tout automate fini  $\mathcal{A}$  en un automate fini équivalent et qui ne contient pas d' $\varepsilon$ -transitions.

Le principe de l'opération est simple : soit un état  $q$  tel qu'il existe deux transitions,  $(q', \alpha, q)$  pour un certain  $\alpha \in A$  en amont et  $(q, \varepsilon, q'')$  en aval. On voit que l'on peut ajouter la transition  $(q', \alpha, q'')$  à la table sans changer pour autant l'ensemble reconnu par  $\mathcal{A}$ . Si l'on répète cette opération pour toutes les suites d' $\varepsilon$ -transitions issues de  $q$  et toutes les transitions amenant à  $q$ , on peut alors retirer la transition  $(q, \varepsilon, q'')$ . L'automate ainsi obtenu est équivalent à  $\mathcal{A}$ . D'où l'algorithme.

Élimination des  $\varepsilon$ -transitions :

1. Calculer pour chaque état  $q$  de l'automate la liste des états qui en sont accessibles par une suite d' $\varepsilon$ -transitions :  $E_q$ .
2. Pour chaque transition  $(q', \alpha, q)$ ,  $\alpha \neq \varepsilon$ , ajouter les transitions  $(q', \alpha, q'')$  pour tout  $q'' \in E_q$ .
3. Supprimer toutes les  $\varepsilon$ -transitions.

Le nouvel automate  $\mathcal{B}$  ainsi obtenu est équivalent à  $\mathcal{A}$  car pour tout mot  $w$ , si  $\mathcal{A}$  le reconnaît :

- par une suite de calcul sans  $\varepsilon$ -transitions, cette suite existe aussi dans  $\mathcal{B}$  ;
- par une suite de calcul avec  $\varepsilon$ -transitions, une suite existe aussi dans  $\mathcal{B}$  qui est obtenue en remplaçant les séquences maximales du type  $q \xrightarrow{\alpha} q_1 \xrightarrow{\varepsilon} q_2 \xrightarrow{\varepsilon} \dots \xrightarrow{\varepsilon} q'$  dans  $\mathcal{A}$  par  $q \xrightarrow{\alpha} q'$  dans  $\mathcal{B}$ .

Inversement, toute suite dans  $\mathcal{B}$  possède par construction son homologue avec  $\varepsilon$ -transitions dans  $\mathcal{A}$ .

### 10.4.3 Suppression du non-déterminisme

Reprenons le § 2.4.1 et exprimons les choses un peu différemment : l'ensemble des valeurs possibles pour la liste  $H$  est fini, puisqu'il correspond à l'ensemble des parties  $\mathcal{P}(Q) = 2^Q$  de l'ensemble des états  $Q$ . L'algorithme consiste en une boucle qui lit séquentiellement les lettres de la donnée  $w$  et transforme la valeur de  $H$  en passant d'un élément de  $\mathcal{P}(Q)$  à un autre. Ceci n'est finalement rien d'autre que le comportement d'un automate fini que nous définissons maintenant formellement.

Soit  $\mathcal{A} = \langle A, Q, q_*, F, \Delta \rangle$  un automate fini sans  $\varepsilon$ -transitions. Soit  $\hat{\mathcal{A}} = \langle A, \mathcal{P}(Q), \{q_*\}, \hat{F}, \hat{\Delta} \rangle$  l'automate fini défini par  $\hat{F} = \{H \in \mathcal{P}(Q) \mid H \cap F \neq \emptyset\}$  pour ensemble d'états finaux et  $\hat{\Delta} = \{(H, \alpha, H') \mid H \in \mathcal{P}(Q) \ \& \ H' = \{q' \mid q \in H \ \& \ (q, \alpha, q') \in \Delta\}\}$  pour table de transitions. L'automate  $\hat{\mathcal{A}}$  est appelé *automate des parties* et il est évidemment déterministe par construction.

C'est alors une simple affaire de rephrasage de la preuve de correction de l'algorithme du § 2.4.1 que de prouver que les deux langages  $\mathcal{L}(\mathcal{A})$  et  $\mathcal{L}(\hat{\mathcal{A}})$  sont identiques.

**Théorème 10.4.1** *Tout automate fini est équivalent à un automate fini déterministe : l'automate des parties.*

Ce résultat est remarquable d'un point de vue théorique. Il permet en particulier de démontrer en toute généralité la propriété de clôture de la famille  $\mathcal{Rat}$  par complémentation. Il faut cependant être conscient du fait que la taille de l'ensemble des parties  $\mathcal{P}(Q)$  est  $2^{|Q|}$  et donc devient rapidement prohibitive. Les Unixiens confirmés auront à cœur de comprendre les limitations imposées à la commande `grep`.

## 10.5 Complexité et limites

### 10.5.1 Déterminisation

Nous donnons dans ce paragraphe un exemple où le saut exponentiel de taille de l'automate déterministe par rapport à celle du non-déterministe est effectivement atteinte.

Considérons sur l'alphabet  $A = \{a, b\}$  la famille des langages  $P_n = (a + b)^* a (a + b)^{n-1}$ , pour  $n \geq 1$ .

1. L'ensemble  $P_n$  est trivialement reconnu par un automate non-déterministe à  $n + 1$  états, comme le montre la figure 10.3 dans le cas  $n = 7$ .

L'automate  $\mathcal{P}_n$  associé est défini comme suit :

- alphabet :  $A = \{a, b\}$  ;
- états :  $Q = \{0, 1, \dots, n\}$  ;
- état initial : 0 ;
- états finaux :  $F = \{n\}$  ;
- table :  $\Delta = \{(0, a, 0), (0, b, 0), (0, a, 1)\} \cup \{(i, a, i + 1), (i, b, i + 1) \mid 1 \leq i < n\}$ .

2.1 Le langage  $P_1$  est reconnu par un automate déterministe  $\mathcal{P}_1$  à 2 états  $Q_1 = \{\varepsilon, a\}$ , d'état initial  $\varepsilon$ , d'état final  $\{a\}$  et de table  $\Delta_1 = \{(\varepsilon, a, a), (\varepsilon, b, \varepsilon), (a, a, a), (a, b, \varepsilon)\}$ . Il est facile de voir que ces 2 états sont nécessaires car seuls les mots se terminant par  $a$  sont acceptés.

2.2 Le langage  $P_2$  est reconnu par un automate déterministe  $\mathcal{P}_2$  à 4 états  $Q_2 = \{\varepsilon, a, aa, ab\}$ , d'état initial  $\varepsilon$ , d'états finaux  $\{aa, ab\}$  et de table  $\Delta_2 = \{(\varepsilon, a, a), (\varepsilon, b, \varepsilon), (a, a, aa), (a, b, ab), (aa, a, aa), (aa, b, ab), (ab, a, a), (ab, b, \varepsilon)\}$ . De nouveau ces 4 états sont nécessaires car chacun mémorise des situations distinctes pour les deux dernières lettres lues à savoir en régime permanent et dans l'ordre des états  $bb, ba, aa, ab$ , et qui ne peuvent pas être fusionnées sous peine de reconnaître des mots n'appartenant pas au langage.

2.3 Donnons comme ultime exemple l'automate déterministe  $\mathcal{P}_3$  à 8 états qui reconnaît  $P_3$  :

- $Q_3 = \{\varepsilon, a, aa, ab, aaa, aab, aba, abb\}$  est l'ensemble des états ;
- $\varepsilon$  est l'état initial ;
- $F = \{aaa, aab, aba, abb\}$  est l'ensemble des états finaux ;
- les transitions sont données par  $\Delta_3 = \{(\varepsilon, a, a), (\varepsilon, b, \varepsilon), (a, a, aa), (a, b, ab), (aa, a, aaa), (aa, b, aab), (ab, a, aba), (ab, b, abb), (aaa, a, aaa), (aaa, b, aab), (aab, a, aba), (aab, b, abb), (aba, a, aa), (aba, b, ab), (abb, a, a), (abb, b, \varepsilon)\}$ . Si l'on prend deux états quelconques, il est toujours possible de trouver un mot pour lesquels le résultat du calcul est différent et par conséquent tous les états sont nécessaires : par exemple avec les états  $aa$  et  $abb$ , le mot  $a$  conduit à un état final pour le premier et pas pour le second.

En règle générale, l'automate déterministe  $\mathcal{P}_n$  qui reconnaît  $P_n$  comporte  $2^n$  états que l'on dénotera par les mots de longueur au plus  $n$  commençant par  $a$  plus le mot vide :

- $Q_n = \{\varepsilon, a, aa, ab, aaa, aab, aba, abb, \dots, a^n, a^{n-1}b, a^{n-2}ba, a^{n-2}bb, \dots, ab^{n-2}a, ab^{n-1}\}$  ;

- $\varepsilon$  est l'état initial ;
- $F = \{a^n, a^{n-1}b, a^{n-2}ba, a^{n-2}bb, \dots, ab^{n-2}a, ab^{n-1}\}$  est l'ensemble des états finaux ;
- les transitions sont données par les règles suivantes exécutées dans cet ordre :  $(\varepsilon, a, a) \in \Delta$  et  $(\varepsilon, b, \varepsilon) \in \Delta$  ; si  $q$  est un état de longueur inférieure strictement à  $n$ ,  $(q, a, qa) \in \Delta$  et  $(q, b, qb) \in \Delta$  ; si  $q$  est de longueur  $n$  et  $q = aaf$  alors  $(q, a, afa) \in \Delta$  et  $(q, b, afb) \in \Delta$  ; si  $q$  est de longueur  $n$  et  $q = ab^k f$ ,  $f$  ne commençant pas par  $b$ , alors  $(q, a, fa) \in \Delta$  et  $(q, b, fb) \in \Delta$ .

Le lecteur reprendra les arguments ci-dessus pour se convaincre de la justesse de la proposition.

**Proposition 10.5.1** *L'automate déterministe minimal équivalent à un automate non-déterministe de taille  $n$  peut avoir  $\Omega(2^n)$  états.*

### 10.5.2 Automate minimal

Le dernier concept de cette section est qu'il existe une forme canonique des automates déterministes qui "contient" en un sens mathématiquement précis toutes les autres et qui est appelée "automate minimal". Dans toute la suite on suppose que les automates ont été préalablement nettoyés de leurs  $\varepsilon$ -transitions et que tous leurs états sont utiles, c'est-à-dire sont accessibles depuis l'état final et peuvent conduire à un état final.

#### Indiscernabilité

Oublions cependant les automates et considérons un langage  $L \subset A^*$  quelconque. Ce langage permet de définir une relation d'équivalence sur les mots de  $A^*$  que l'on appelle la *L-indiscernabilité*.

**Définition 10.5.1** *Soit  $L \subset A^*$  un langage. On dit que deux mots  $x$  et  $y$  de  $A^*$  sont L-indiscernables si et seulement si*

$$\forall w \in A^* \quad xw \in L \Leftrightarrow yw \in L.$$

On note  $x \equiv_L y$  la relation de L-indiscernabilité.

Il est évident que  $\equiv_L$  est une relation d'équivalence quel que soit le langage  $L$ .

#### Classes et automates finis

Considérons maintenant un automate fini *déterministe*  $\mathcal{A}$ , son ensemble d'états  $Q$  et notons  $L$  le langage qu'il reconnaît. Rappelons que  $L_q = \{x \mid q_* \xrightarrow{x} q\}$  est l'ensemble des mots qui amènent de l'état initial dans l'état  $q$ .

Deux mots  $x$  et  $y$  d'un même  $L_q$  sont L-indiscernables, puisque pour tout mot  $w$ , l'automate agit de la même façon sur  $xw$  et  $yw$ , le résultat étant dans les deux cas celui du calcul de  $\mathcal{A}$  sur  $w$  à partir de l'état  $q$ .

De plus, soient deux états  $q$  et  $q'$  distincts et donc tels que  $L_q \cap L_{q'} = \emptyset$  (puisque  $\mathcal{A}$

est déterministe) ; alors s'il existe  $x \in L_q$  et  $x' \in L_{q'}$  tels que  $x \equiv_L x'$ , alors les deux ensembles tout entiers le sont :  $L_q \equiv_L L_{q'}$ , ce que nous écrirons plus simplement au niveau des états,  $q \equiv_L q'$ . Les deux états  $q$  et  $q'$  sont  $L$ -indiscernables.

Si deux états d'un automate  $\mathcal{A}$  sont  $\mathcal{L}(\mathcal{A})$ -indiscernables, on peut donc simplifier cet automate en leur donnant le même nom et en modifiant la table de transitions en conséquence. Un critère simple d'indiscernabilité est le suivant, dont la preuve est évidente.

**Lemme 10.5.1** *Soient deux états  $q$  et  $q'$  d'un automate fini déterministe  $\mathcal{A}$  qui vérifient la propriété :  $\forall \alpha \in A, (q, \alpha, r) \in \Delta \Leftrightarrow (q', \alpha, r) \in \Delta$ . Alors  $q$  et  $q'$  sont  $\mathcal{L}(\mathcal{A})$ -indiscernables.*

On construit donc à partir de cette remarque un automate qui ne peut plus être réduit et qu'on appelle *automate minimal*. Cet algorithme construit l'automate minimal à partir des états finaux, puis par accréation des états indiscernables ; il sera détaillé en exercices. On montre maintenant que cet automate est unique car il ne dépend en fait que du langage et non de l'automate.

### Langages rationnels

Tout langage rationnel  $L \subset A^*$  étant reconnaissable par un automate fini détermine ce faisant une partition finie de  $A^*$  en classes de  $L$ -indiscernabilité : par exemple, celle déterminée par les états de l'automate fini.

Inversement, considérons un langage  $L$  dont les classes d'indiscernabilité

$$C_1, C_2, \dots, C_p \subset A^*$$

sont en nombre fini  $p$ . On a donc :

- $C_i \cap C_j = \emptyset$ , pour tous  $1 \leq i, j \leq p$  ;
- $\cup_{1 \leq i \leq p} C_i = A^*$  ;
- $\forall x, y \in C_i, x \equiv_L y$ .

Alors  $L$  est union de certaines des classes  $C_i$ , puisque chaque classe est soit contenue dans  $L$  soit disjointe de lui (en prenant  $w = \varepsilon$  dans la définition des classes). Par ailleurs pour chaque lettre  $\alpha \in A$  et chaque classe  $C_i$  il existe un  $j$  unique tel que  $C_i \alpha \subset C_j$ , car  $x \equiv_L y \Rightarrow x\alpha \equiv_L y\alpha$ .

Considérons alors l'automate fini  $\mathcal{A}$  dont :

- les états sont les  $C_i$ ,
- l'état initial celui des  $C_i$  qui contient le mot vide  $\varepsilon$ ,
- les états finaux sont les  $C_i \subset L$ ,
- la table est définie par les  $(i, \alpha, j)$  tels que  $C_i \alpha \subset C_j$ .

Cet automate reconnaît à l'évidence le langage  $L$  et est fini. D'où le théorème.

**Théorème 10.5.1** *Un langage  $L \subset A^*$  est rationnel si et seulement s'il existe une partition finie de  $A^*$  en classes  $L$ -indiscernables.*

### Automate minimal

Combinons maintenant les deux derniers paragraphes. Tout automate fini  $\mathcal{A}$  reconnaît un langage rationnel  $L$  dont l'ensemble des classes d'indiscernabilité est fini et correspond à la *partition la plus grossière* selon  $\equiv_L$  (celle qui possède le moins de classes). Les classes  $L_q$  de  $\mathcal{A}$  sont donc des sous-ensembles de ces classes d'après les raisonnements des paragraphes précédents et il est donc impossible de trouver un automate plus petit que celui construit sur les classes d'indiscernabilité. D'où le théorème.

**Théorème 10.5.2** [Myhill, Nerode]

*Tout langage rationnel admet un unique automate déterministe minimal : celui défini par les classes d'indiscernabilité du langage.*

## 10.6 Langages non rationnels : le lemme d'itération

Ce théorème donne une indication sur la structure répétitive inévitable des mots des langages rationnels : on l'appelle encore *Lemme de l'étoile* en raison de la forme de son énoncé.

**Théorème 10.6.1** *Soit  $L \in \mathcal{Rat}$  un langage rationnel infini. Il existe un entier  $\lambda$  tel que, pour tout mot  $w \in L$ ,  $|w| \geq \lambda$ ,  $w$  peut se factoriser en  $w = xyz$ ,  $y \neq \varepsilon$  et tel que*

$$xy^*z \subset L.$$

En d'autres termes, le langage infini des mots obtenus en itérant un nombre arbitraire de fois le facteur  $y$  est inclus dans  $L$  ; ou encore, les mots assez longs de tout langage rationnel contiennent des formes de périodicités.

Preuve : Soit  $\mathcal{A}$  un automate (déterministe) reconnaissant  $L$  ; soit  $\lambda$  son nombre d'états et considérons pour un mot  $w$  du langage, de longueur  $|w| \geq \lambda$ , le calcul qui le reconnaît :

$$q_* \xrightarrow{w_1} q_1 \xrightarrow{w_2} q_2 \xrightarrow{w_3} \dots \xrightarrow{w_n} q_f.$$

Il y a strictement plus de  $\lambda$  occurrences d'états dans cette chaîne et donc deux au moins sont identiques, par exemple à  $q$  ; soit  $y$  le facteur de  $w$  qui les sépare :

$$q \xrightarrow{y} q.$$

On peut répéter cette séquence et obtenir une nouvelle séquence de calcul qui valide un mot plus long, de même qu'on peut la couper sans altérer la validité du calcul, de sorte que toutes les séquences du type

$$q_* \xrightarrow{x} q \left( \xrightarrow{y} q \right)^* \xrightarrow{z} q_f,$$

sont valides et reconnaissent précisément les mots du type  $xy^*z$  définis dans l'énoncé.

Ce résultat est un outil puissant pour montrer que certains langages ne sont pas rationnels. Nous donnons à titre d'exemple un corollaire qui sera repris dans le chapitre suivant.

**Corollaire 10.6.1** *Le langage  $P = \{a^n b^n \mid n \in \mathbb{N}\}$  n'est pas rationnel.*

Preuve : Soient  $x, y, z \in \{a, b\}^*$ , avec  $y \neq \varepsilon$ , tels que  $xy^*z \subset P$ . Alors  $y$  doit contenir autant de  $a$  que de  $b$ , les  $a$  précédant les  $b$ ,  $y = a^l b^l$  ; comme ce nombre  $l$  n'est pas nul,  $yy = a^l b^l a^l b^l$ , et donc le mot  $xy^2z$  n'est pas dans  $P$ . Contradiction.

## 10.7 Appendice

Nous reproduisons ci-dessous la définition des expressions régulières pour la commande `grep` de Linux, telle qu'elle apparaît dans le manuel de référence. Cette commande construit un automate à partir de l'expression régulière puis teste si le fichier proposé en second argument contient des facteurs qui satisfont cette expression. Il s'agit d'expressions étendues, mais qui restent équivalentes à des expressions usuelles.

### REGULAR EXPRESSIONS

A regular expression is a pattern that describes a set of strings. Regular expressions are constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions. Grep understands two different versions of regular expression syntax: "basic" and "extended." In GNU grep, there is no difference in available functionality using either syntax. In other implementations, basic regular expressions are less powerful. The following description applies to extended regular expressions; differences for basic regular expressions are summarized afterwards.

The fundamental building blocks are the regular expressions that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. Any metacharacter with special meaning may be quoted by preceding it with a backslash.

A list of characters enclosed by [ and ] matches any single character in that list; if the first character of the list is the caret ^ then it matches any character not in the list. For example, the regular expression [0123456789] matches any single digit. A range of ASCII characters may be specified by giving the first and last characters, separated by a hyphen. Finally, certain named classes of characters are predefined. Their names are self explanatory, and they are [:alnum:], [:alpha:], [:cntrl:], [:digit:], [:graph:], [:lower:], [:print:], [:punct:], [:space:], [:upper:], and [:xdigit:]. For example, [[:alnum:]] means [0-9A-Za-z], except the latter form is dependent upon the ASCII character encoding, whereas the former is portable. (Note that the brackets in these class names are part of the symbolic names, and must be included in addition to the brackets delimiting the bracket list.) Most metacharacters lose their special meaning inside lists. To include a literal ] place it first in the list. Similarly, to include a literal ^ place it anywhere but first. Finally, to include a literal - place it last.

The period . matches any single character. The symbol \w is a synonym for [[:alnum:]] and \W is a synonym for [^[:alnum:]].

The caret `^` and the dollar sign `$` are metacharacters that respectively match the empty string at the beginning and end of a line. The symbols `\<` and `\>` respectively match the empty string at the beginning and end of a word. The symbol `\b` matches the empty string at the edge of a word, and `\B` matches the empty string provided it's not at the edge of a word.

A regular expression may be followed by one of several repetition operators:

- ?        The preceding item is optional and matched at most once.
- \*        The preceding item will be matched zero or more times.
- +        The preceding item will be matched one or more times.
- {n}      The preceding item is matched exactly n times.
- {n,}     The preceding item is matched n or more times.
- {n,m}    The preceding item is matched at least n times, but not more than m times.

Two regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated subexpressions.

Two regular expressions may be joined by the infix operator `|`; the resulting regular expression matches any string matching either subexpression.

Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole subexpression may be enclosed in parentheses to override these precedence rules.

The backreference `\n`, where `n` is a single digit, matches the substring previously matched by the `n`th parenthesized subexpression of the regular expression.

In basic regular expressions the metacharacters `?`, `+`, `{`, `|`, `(`, and `)` lose their special meaning; instead use the backslashed versions `\?`, `\+`, `\{`, `\|`, `\(`, and `\)`.

## Exercices

1. Décrire les langages engendrés par les expressions régulières suivantes :
  - $0(0+1)^*0$  ;
  - $((\varepsilon+0)1^*)^*$  ;
  - $(0+1)^*0(0+1)(0+1)$  ;
  - $0^*10^*10^*10^*$  ;
  - $(00+11)^*((01+10)(00+11)^*(01+10)(00+11)^*)^*$ .
2. Écrire des expressions régulières pour les langages suivants :
  - tous les mots qui contiennent les 5 voyelles dans l'ordre ;
  - toutes les suites de chiffres sans qu'aucun ne soit répété ;
  - toutes les suites de chiffres sans qu'aucun ne soit répété consécutivement ;
  - les mots qui ne contiennent pas le facteur 011 ;
  - les mouvements du jeu d'échecs.
3. Construire les automates qui reconnaissent les langages :
  - $(a+b)^*$  ;
  - $(a^*+b^*)^*$  ;
  - $((\varepsilon+a)b^*)^*$  ;
  - $(a+b)^*abb(a+b)^*$ .
 Montrer leurs calculs possibles sur le mot *ababbab*.
4. Construire des automates déterministes équivalents.
5. Les faire aussi petits que possible.
6. Montrer que  $(a+b)^*$ ,  $(a^*+b^*)^*$  et  $((\varepsilon+a)b^*)^*$  engendrent le même langage.
7. Les langages suivants sont-ils rationnels ?
  - i)  $\{w \in (a+b)^* \mid |w|_a = |w|_b\}$ , en notant  $|w|_a$  le nombre de *a* dans *w* ;
  - ii)  $\{a^i b^j \mid i \neq j\}$  ;
  - iii)  $\{a^i b^j \mid i = j \bmod k\}$ , pour *k* fixé ;
  - iv)  $\{c^k a^i b^j \mid i = j \bmod k\}$ .
8. Dans un automate fini certains états peuvent être inutiles ; expliquer pourquoi. Peut-on les éliminer algorithmiquement ? Comment ?
9. Peut-on déterminer si le langage reconnu par un automate fini est non vide ? Estimer la complexité de mise en œuvre de la réponse.
10. Peut-on déterminer algorithmiquement si deux automates engendrent le même langage ?
11. Écrire complètement l'algorithme d'élimination des  $\varepsilon$ -transitions d'un automate.



## Chapitre 11

# Parenthésages, Langages algébriques

Le lemme de l'étoile montre clairement les limites des expressions régulières et des automates finis pour exprimer des propriétés structurelles à distance dans les mots. Un grand nombre de cas pratiques relèvent pourtant de ce type de contraintes : expressions arithmétiques et logiques, exécution des appels de fonctions dans les langages de programmation acceptant la récursivité, démontage et remontage d'assemblages complexes de pièces mécaniques, structures biologiques pour certains ARN et protéines, structures mathématiques en combinatoire et en physique statistique, etc. Développer un formalisme permettant de décrire et d'analyser ces structures emboîtées est donc une nécessité. N. Chomsky, le premier, a compris qu'il fallait dépasser le formalisme des expressions régulières pour exprimer les structures grammaticales des langues naturelles, en particulier celles des langues indo-européennes ; il a ainsi construit le premier modèle de "grammaires hors-contexte" (*contextfree grammars*) et en a étudié les propriétés mathématiques et informatiques avec M.-P. Schützenberger, à qui l'on doit de nombreux développements. La théorie s'est aussi imposée, peu après, dans la conception des compilateurs, et donc dans celle des langages de programmation. Elle est devenue un point de passage inévitable et est intégrée, sous des formes particulières, dans les langages et systèmes, avec par exemple le concept de *stream* et de *parser* de Caml et le module *Yacc* d'Unix. Ce chapitre est consacré aux bases conceptuelles ; le suivant sera une brève introduction à la construction d'analyseurs syntaxiques, comme on en trouve dans les compilateurs.

### 11.1 Exemples de structures parenthésées

#### 11.1.1 Notations

La structure la plus évidente est celle des expressions arithmétiques, qu'elle soit classique, comme dans

$$(a + b)(a^2 - 2ab + b^2),$$

ou informatique comme dans

$$(a+b) * (a*a-2*a*b+b*b),$$

où le lecteur remarquera que l'on a déjà utilisé des conventions sur la " priorité " des opérateurs pour évaluer sans ambiguïté les sommes de produits.

Dans cette notation dite " infixe ", car les opérateurs sont placés entre les opérandes, on doit préciser (à défaut de tout parenthéser) que les multiplications et divisions sont exécutées *avant* les additions et soustractions elles-mêmes prioritaires sur l'opposé (– unaire), et que les opérations de même priorité sont exécutées de gauche à droite. Dans ces conditions le système est interprétable de façon unique. Dès les années 20, les logiciens, en particulier ceux de l'école polonaise avaient mis au point des systèmes de notation non ambigus et sans parenthèses : les formes polonaises préfixée ou postfixée encore appelées respectivement notations préfixe et postfixe. Ainsi pour l'expression qui nous sert d'exemple a-t-on en forme préfixée

$$*+ab+-*aa*ab*bb,$$

et en forme postfixée

$$ab+aa*ab*-bb*+*.$$

Toutes ces représentations découlent en fait naturellement d'une seule et même notation qui voit et implante les expressions sous forme d'arbres.

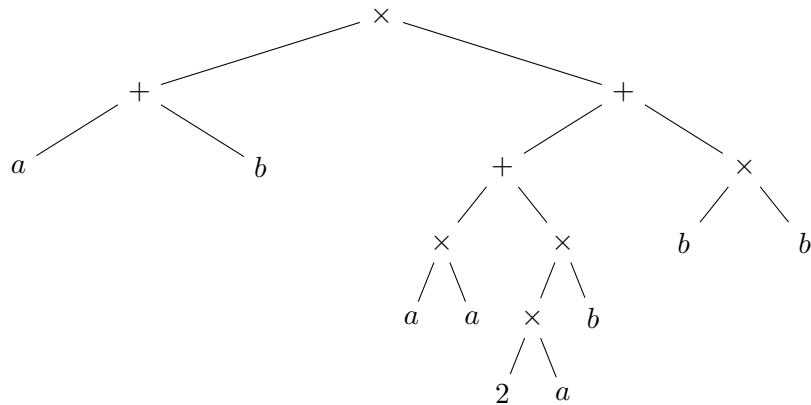


FIG. 11.1 – L'arbre pour l'expression  $(a+b)(a^2-2ab+b^2)$ .

## 11.2 Grammaires hors-contexte

Le problème est d'étendre le mécanisme des automates finis et des expressions régulières. On verra dans la section suivante comment on peut procéder au moyen

d'automates. Dans cette section on revient d'abord sur les langages rationnels associés à un automate et on dégage le concept de grammaire de réécriture qui va être étendu pour définir une classe plus importante de langages.

**11.2.1 Automates, langages et grammaires**

À partir d'un automate fini  $\mathcal{A} = \langle A, Q, q_0, Q_f, \Delta \rangle$ , on a vu au chapitre précédent qu'on définissait les langages  $L_q$ , pour chaque état  $q \in Q$ , constitués des mots qui étiquettent un chemin dans le graphe de l'automate amenant de l'état initial  $q_0$  à l'état  $q$ . On a vu également que ces langages se définissent récursivement par un système de relations du type

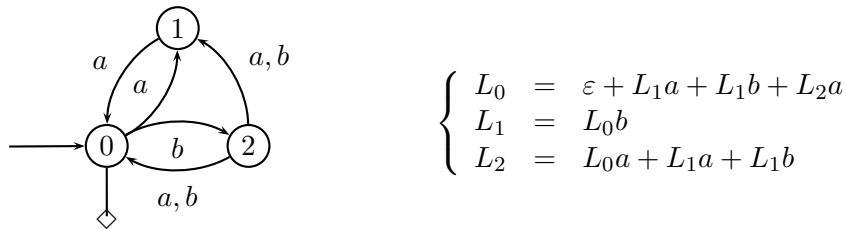
$$L_q = \{wa \mid a \in A \cup \{\varepsilon\} \ \& \ \exists q' \ w \in L_{q'} \ \& \ \langle q', a, q \rangle \in \Delta\},$$

ce qui s'écrit encore, en utilisant la concaténation ensembliste,

$$L_q = \bigcup_{\langle q', a, q \rangle \in \Delta} L_{q'}a,$$

en ajoutant pour l'état initial que  $\varepsilon \in L_{q_0}$ .

Par exemple, on aura le système suivant pour l'automate dessiné en vis-à-vis :



Comme on l'a fait dans les expressions régulières, on utilise le + pour noter l'union ensembliste, et la concaténation est notée par la simple juxtaposition.

Parcourir un chemin dans l'automate est équivalent à une réécriture depuis une "variable"  $L_q$  correspondant à un état final — 0 en l'occurrence. On notera que cette réécriture se fait en sens inverse du parcours du graphe de l'automate :

$$L_0 \rightarrow L_1a \rightarrow L_0ba \rightarrow L_2aba \rightarrow L_0aaba \rightarrow aaba.$$

Considérons maintenant le système plus compliqué suivant :

$$L = aLb + \varepsilon.$$

Une suite de réécritures conservant l'ordre des facteurs dans la concaténation donne typiquement :

$$L \rightarrow aLb \rightarrow aaLbb \rightarrow aaaLbbb \rightarrow aaabbb,$$

ce qui par récurrence immédiate nous permet de caractériser et d'engendrer le langage  $\{a^n b^n \mid n \geq 0\}$  qui demeurerait inaccessible dans le cadre des expressions régulières.

De la même façon, le système de règles  $\mathcal{D}$ , défini par

$$S = (S)S + x,$$

permet de définir les mots correctement parenthésés.

Comme précédemment, il est judicieux de construire un arbre qui explicite les récritures successives, comme le montre la figure 11.2.

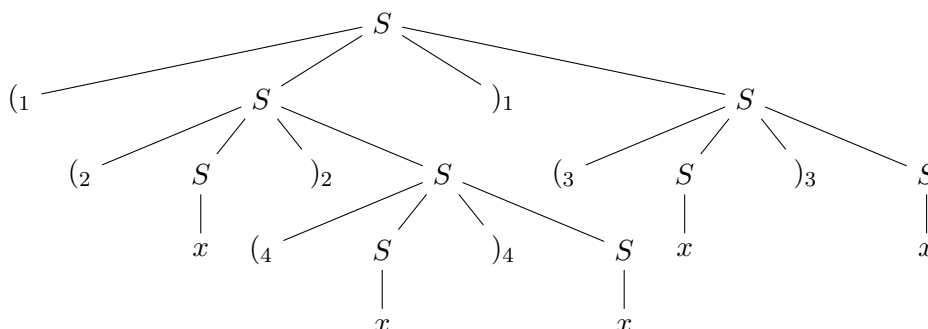


FIG. 11.2 – Un arbre de dérivation pour  $(1(2x))_2(3x)_3x)_1(4x)_4x$  : on a numéroté les parenthèses pour mettre en évidence les paires ouvrante-fermante homologues.

### 11.2.2 Terminologie

**Définition 11.2.1** Une grammaire hors-contexte (context-free dans la terminologie de Chomsky) est un système de récriture  $\mathcal{G}$  défini par :

- $V$ , un ensemble fini de variables (ou non-terminaux) ;
- $T$ , un ensemble fini de terminaux (alphabet du langage) ;
- $\mathcal{R} \subset V \times (V \cup T)^*$ , un ensemble fini de règles de récriture ;
- $S \in V$ , un symbole distingué appelé axiome.

Dans l'exemple des expressions arithmétiques de la section 1, les variables sont l'ensemble  $\{\text{Expression}, \text{Terme}, \text{Facteur}\}$ , les terminaux sont les chiffres et les lettres, les règles sont données dans la figure 3.4, et l'axiome est la variable `Expression`. On prendra soin de noter que les variables d'une formule ne sont pas celles de la grammaire mais au contraire ses terminaux.

Définissons maintenant le langage engendré par une grammaire, en formalisant le concept de dérivation : on omettra de préciser dans toute la suite du chapitre que les grammaires sont hors-contexte.

**Définition 11.2.2** Étant donné une grammaire  $\mathcal{G}$  et deux mots  $w, w' \in (V \cup T)^*$ , on dit que  $w$  se récrit en  $w'$  (ou que  $w'$  dérive en un coup de  $w$ ), si et seulement si :

- $w$  se décompose en  $w = xvy$ ,  $x, y \in (V \cup T)^*$  et  $v \in V$  ;
- $w'$  se décompose en  $w' = xfy$ ,  $x, y \in (V \cup T)^*$  (les mêmes) et  $f \in (V \cup T)^*$  ;
- $(v, f)$  est une règle de  $\mathcal{G}$ .

On note une réécriture  $w \rightarrow \mathcal{G}w'$  ou encore  $w \rightarrow w'$  lorsque la grammaire est implicite.

Une suite de réécritures

$$w \rightarrow w_1 \rightarrow w_2 \rightarrow \dots \rightarrow w_n = w'$$

est appelée dérivation et est notée  $w \xrightarrow{*} w'$ .

Si, de plus, le mot  $w'$  est tout entier sur l'alphabet terminal ( $w' \in T^*$ ), on dit que la dérivation est terminale.

Un même mot peut posséder plusieurs dérivations ; ainsi le mot

$$(1(2x))_2(3x)_3x_1(4x)_4x$$

est-il dérivable des deux façons suivantes où l'on a souligné la variable réécrite à chaque étape :

$$\begin{aligned} S \rightarrow (S)\underline{S} \rightarrow (S)(\underline{S})S \rightarrow (\underline{S})(x)S \rightarrow ((S)\underline{S})(x)S \rightarrow ((S)(\underline{S})S)(x)S \rightarrow ((S)(x)\underline{S})(x)S \\ \rightarrow ((S)(x)x)(x)\underline{S} \rightarrow ((\underline{S})(x)x)(x)x \rightarrow ((x)(x)x)(x)x, \text{ et} \end{aligned}$$

$$\begin{aligned} S \rightarrow (\underline{S})S \rightarrow ((\underline{S})S)S \rightarrow ((x)\underline{S})S \rightarrow ((x)(\underline{S})S)S \rightarrow ((x)(x)\underline{S})S \rightarrow ((x)(x)x)\underline{S} \\ \rightarrow ((x)(x)x)(\underline{S})S \rightarrow ((x)(x)x)(x)\underline{S} \rightarrow ((x)(x)x)(x)x. \end{aligned}$$

Les arbres de dérivation associés sont identiques ; dans la seconde dérivation, on a toujours effectué la réécriture la plus à gauche dans l'arbre. Et il est évident que cette suite de dérivations est toujours possible et est caractéristique d'un arbre.

**Proposition 11.2.1** *Toute dérivation dans une grammaire hors-contexte est équivalente à une dérivation dans laquelle le symbole non-terminal le plus à gauche est systématiquement réécrit.*

On en “ déduit ” que les réécritures ne dépendent pas du contexte et que chaque sous-arbre de l'arbre de dérivation peut être calculé indépendamment des autres.

**Définition 11.2.3** *On appelle langage engendré par une grammaire  $\mathcal{G}$  l'ensemble des mots sur l'alphabet terminal dérivables de l'axiome de la grammaire :*

$$\mathcal{L}(\mathcal{G}) = \{w \in T^* \mid S \xrightarrow{*} w\}.$$

Remarquons encore que les mots obtenus à chaque étape de la dérivation sont les lisières des arbres de dérivation successifs et que les mots du langage sont les lisières écrites sur l'alphabet terminal. On appelle lisière le mot obtenu en lisant les feuilles de l'arbre en ordre préfixe.

### 11.2.3 Ambiguïté

On a déjà fait remarquer que l'analyse d'une formule pouvait être multiple comme dans le cas des expressions arithmétiques dans lesquelles on ne tiendrait pas compte des priorités. Ce souhait d'avoir une propriété de "décodage unique" est évident si l'on veut associer comme en linguistique un sens unique (incarné par l'arbre syntaxique) à la phrase (le mot engendré par la grammaire), souhait encore plus impératif en programmation, où il faut qu'un programme soit exécuté de la même façon par toutes les machines. On souhaite donc construire des grammaires possédant la propriété de n'engendrer chaque mot du langage qu'une fois et une seule.

**Définition 11.2.4** *Une grammaire  $G$  est dite non-ambiguë si et seulement si chaque mot du langage possède un arbre de dérivation unique.*

Par exemple, la grammaire déjà écrite pour le langage  $\{a^n b^n \mid n \geq 0\}$  est non-ambiguë. On pourra facilement l'étendre en des grammaires non-ambiguës pour les langages  $L_1 = \{a^n b^n c^m \mid n, m \geq 0\}$  et  $L_2 = \{a^n b^m c^m \mid n, m \geq 0\}$ . Il est cependant impossible de construire une grammaire non-ambiguë pour le langage union  $L_1 \cup L_2$  qui est donc inhéremment ambigu.

La théorie de l'ambiguïté est complexe et dépasse largement le cadre de ce cours.

## 11.3 Propriétés des langages algébriques

Les langages engendrés par des grammaires hors-contexte satisfont des systèmes d'équations qui sont le pendant des systèmes algébriques en variables commutatives, de la même façon que les langages rationnels, associés aux expressions régulières satisfont des équations linéaires et sont le pendant des fractions rationnelles en variables non-commutatives. Nous les baptiserons donc langages algébriques selon la terminologie de M.-P. Schützenberger et noterons  $\mathcal{Alg}$  leur classe. Ils vérifient un certain nombre de propriétés de clôture ensemblistes.

**Proposition 11.3.1** *La classe  $\mathcal{Alg}$  des langages algébriques est close par union, produit et quasi-inverse. Elle est close par homomorphisme et substitution algébrique.*

Preuve : Soient  $L_1 = \mathcal{L}(\mathcal{G}_1)$  et  $L_2 = \mathcal{L}(\mathcal{G}_2)$  les langages engendrés par les grammaires  $\mathcal{G}_1$  et  $\mathcal{G}_2$  d'axiomes respectifs  $S_1$  et  $S_2$ . Remarquons qu'on peut toujours supposer que les alphabets de variables  $V_1$  et  $V_2$  sont disjoints (en les renommant). On construit alors explicitement des grammaires pour  $L_1 \cup L_2$ ,  $L_1 L_2$  et  $L_1^*$ , à partir de  $\mathcal{G}_1 = \langle V_1, T_1, \mathcal{R}_1, S_1 \rangle$  et de  $\mathcal{G}_2 = \langle V_2, T_2, \mathcal{R}_2, S_2 \rangle$ .

La nouvelle grammaire s'écrit alors sous la forme générique  $\mathcal{G} = \langle V, T, \mathcal{R}, S \rangle$ , où les ensembles  $V$  et  $\mathcal{R}$  sont définis dans chaque cas comme suit.

Union :

- $T = T_1 \cup T_2$  ;
- $V = \{S\} \cup V_1 \cup V_2$  ;
- $\mathcal{R} = \{S = S_1 + S_2\} \cup \mathcal{R}_1 \cup \mathcal{R}_2$ .

Produit :

- $T = T_1 \cup T_2$  ;
- $V = \{S\} \cup V_1 \cup V_2$  ;
- $\mathcal{R} = \{S = S_1 S_2\} \cup \mathcal{R}_1 \cup \mathcal{R}_2$ .

Quasi-inverse :

- $T = T_1$  ;
- $V = \{S\} \cup V_1$  ;
- $\mathcal{R} = \{S = S_1 S + \varepsilon\} \cup \mathcal{R}_1$ .

Homomorphisme :

L'homomorphisme est une application  $h$  de  $T_1$  dans  $T_2^*$  ; on l'étend à  $V_1$  comme l'identité ( $h(v) = v$  sur  $V_1$ ) ;

- $T = T_2$  ;
- $V = V_1$  ;
- $\mathcal{R} = \{S = S_1\} \cup h(\mathcal{R}_1)$ , chaque règle étant transformée par  $h$  qui laisse les variables inchangées.

Substitution algébrique :

La substitution  $\sigma$  est une application  $h$  de  $T_1$  dans  $T_2^*$  (en supposant  $T_2$  disjoint de  $T_1$ ) telle que l'image de chaque lettre  $x \in T_1$  est un langage algébrique  $L_x$  inclus dans  $T_2^*$ , de grammaire  $\mathcal{G}_x$  et d'axiome  $S_x$  ; étant donné un mot  $w = x_1 x_2 \dots x_k$ , son image par la substitution est le langage  $\sigma(w) = \{w_1 w_2 \dots w_k \mid w_i \in L_{x_i}\}$  ; pour ce langage on a la grammaire avec :

- $T = T_2$  ;
- $V = V_1 \cup T_1 \cup_{x \in T_1} V_x$  ;
- $\mathcal{R} = \{S = S_1\} \cup \mathcal{R}_1 \cup_{x \in T_1} \mathcal{R}_x \cup_{x \in T_1} (x = S_x)$ .

De façon assez immédiate les grammaires s'imbriquent les unes dans les autres, conduisant à l'effet cherché.

Une autre opération d'importance est l'intersection avec un langage rationnel, autrement dit, la superposition à la structure parenthétique de régularités décrites par expressions régulières. Le prochain théorème montre comment l'on peut prendre en compte les deux systèmes de contraintes en croisant une grammaire pour un langage algébrique avec un automate fini qui reconnaît le langage rationnel. Ce théorème montre aussi l'importance des arbres de dérivation dans ces processus.

**Théorème 11.3.1** *La classe Alg des langages algébriques est close par intersection avec un langage rationnel.*

Preuve : Soit  $L$  un langage algébrique donné par sa grammaire  $\mathcal{G} = \langle V, T, \mathcal{R}, S \rangle$  et soit  $K$  un langage rationnel reconnu par l'automate déterministe  $\mathcal{A} = \langle T, Q, q_0, F, \Delta \rangle$ .

Le langage  $L \cap K \subset T^*$  est défini par la grammaire suivante  $\mathcal{G}'$  :

- $V' = \{S_0\} \cup \{(q, v, q') \mid q, q' \in Q, v \in V\}$  ;
- $S_0$  est l'axiome ;
- les règles sont de deux types  $\mathcal{R}_1$  et  $\mathcal{R}_2$  et expriment de façon récursive que l'ensemble des mots engendrés par la variable  $(q, v, q')$  est celui des mots  $w \in T^*$  qui, simultanément, font passer l'automate  $\mathcal{A}$  de l'état  $q$  dans l'état  $q'$ .

Les variables de  $\mathcal{G}'$  susceptibles de produire des mots de  $L \cap K$  doivent donc être du type  $(q_0, S, q_f)$ ,  $q_f \in F$  étant un état final de l'automate. Pour n'avoir qu'un seul axiome dans la grammaire nous utilisons  $S_0$  et  $\mathcal{R}_1$  est formé des règles

$$\mathcal{R}_1 = \{S_0 = (q_0, S, q) \mid q \in F\}.$$

L'ensemble  $\mathcal{R}_2$  décrit récursivement le double processus de  $L$  et  $K$ . Pour chaque règle  $v = f$  terminale, c.-à-d. où  $f \in T^*$ ,  $\mathcal{R}_2$  contient les règles du type

$$\{(q, v, q') = f \mid q \xrightarrow{f} q'\};$$

et pour chaque règle de la forme  $v = f_0 v_1 f_1 v_2 \dots v_k f_k$ , où  $v_i \in V$  et  $f_i \in T^*$ ,  $\mathcal{R}_2$  contient les règles du type

$$\begin{aligned} \{(q, v, q') = f_0(q_1, v_1, q'_1) f_1(q_2, v_2, q'_2) \dots (q_k, v_k, q'_k) f_k \\ \mid \forall i, 1 \leq i < k, q'_i \xrightarrow{f_i} q_{i+1} \ \& \ q \xrightarrow{f_0} q_1 \ \& \ q'_k \xrightarrow{f_k} q'\}. \end{aligned}$$

Par récurrence sur la structure des dérivations on voit qu'à chaque étape de la construction de l'arbre d'un mot du langage  $L$  on propage la contrainte que ce mot soit aussi reconnu par l'automate de  $K$ , puisque l'on impose — ce que la grammaire  $\mathcal{G}'$  vérifie — que le mot engendré par  $L$  amène aussi l'automate dans un état terminal.

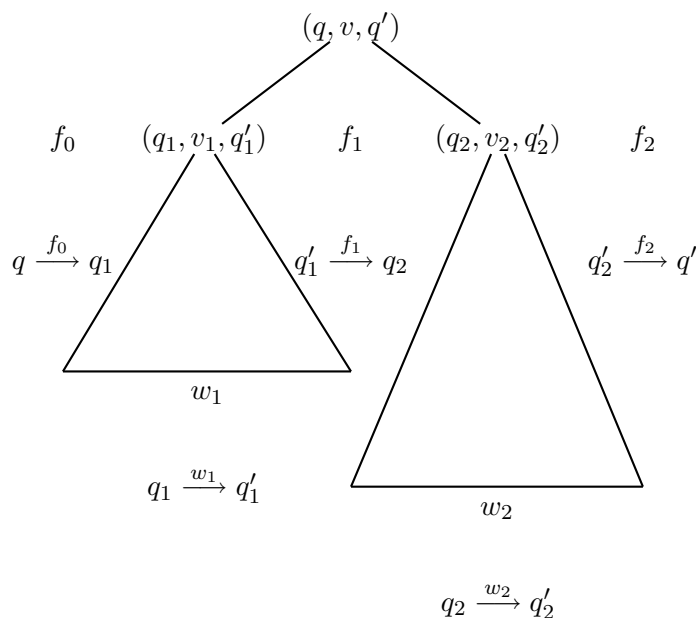


FIG. 11.3 – Illustration pour l'arbre de dérivation dans la grammaire avec intersection par un rationnel.

## 11.4 Grammaires de Greibach et automates à pile

On montre dans cette section que l'on peut mettre les grammaires dans une forme canonique qui permet, premièrement, d'éliminer les récritures récursives de variables dans le sous-arbre gauche, et deuxièmement de mettre en place un mécanisme de reconnaissance des mots du langage, un peu moins coûteux en mémoire et en temps que le processus d'exploration brut. Ce processus, appelé *automate à pile* reste cependant non-déterministe et il faudra bien comprendre qu'il est impossible de faire autrement. On s'éloigne ainsi de la situation rencontrée pour les langages rationnels et les automates finis.

### 11.4.1 Forme normale de Greibach

Les grammaires hors-contexte définies plus haut possèdent des règles de réécriture du type  $v = w$ , où  $w \in V^*$  ; on peut donc "tourner en rond" dans les dérivations avant de commencer à produire des lettres de l'alphabet terminal. Ceci rend l'analyse prédictive des mots assez compliquée et on souhaiterait pouvoir se passer de telles productions.

**Proposition 11.4.1** *Il existe un algorithme qui teste si le langage engendré par une grammaire hors-contexte est vide ou non.*

Preuve : Considérons un arbre de dérivation qui produise un mot  $w$  du langage  $L$  donc sur l'alphabet terminal, en supposant le langage non vide. Deux cas peuvent se produire : les branches issues de la racine (l'axiome) sont étiquetées par des variables toutes différentes, ou bien une variable, au moins, est répétée.

Considérons par exemple la grammaire  $\mathcal{G} = \langle \{A, B, C, S\}, \{a, b, c\}, S, \{S = AC, A = aB + \varepsilon, B = Ab, C = Cc + \varepsilon\} \rangle$ .

Un arbre de dérivation pour le mot  $aabbc$  est donné ci-dessous, et on voit que la variable  $A$  apparaît répétée sur une branche alors que  $C$  ne l'est pas. Il est cependant possible en tronquant cet arbre de produire un mot terminal de la grammaire qui soit sans redondance comme le montre l'exemple.

On en arrive donc à la situation suivante :

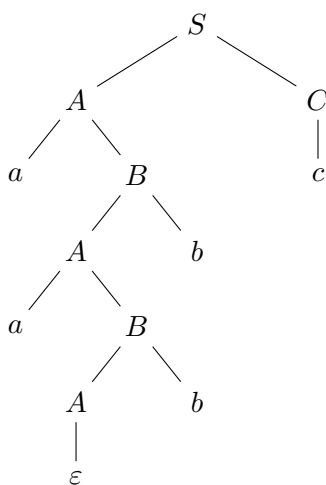
- si  $L$  est non vide, il existe un mot dont l'arbre de dérivation a une profondeur au plus  $m = |V|$ , nombre de variables de la grammaire puisque, sur chacune de ses branches, aucune feuille n'est répétée ;
- il suffit d'énumérer tous les arbres de dérivation possibles de profondeur au plus  $m$  pour décider de la vacuité de  $\mathcal{L}(\mathcal{G})$ .

Considérons maintenant les variables inutiles ; elles sont de deux types : celles qui ne produisent pas de mots sur  $T^*$  et celles qu'on ne peut pas atteindre depuis l'axiome.

**Proposition 11.4.2** *Toute grammaire peut être transformée en une grammaire équivalente où chaque variable produit un mot terminal :  $v \xrightarrow{*} f \in T^*$ .*

Preuve : On procède en deux temps :

- pour chaque  $v \in V$ , tester si les mots engendrés à partir de  $v$  (pris pour axiome)



dans la grammaire  $\mathcal{G}$  forment un langage vide ;

— si tel est le cas, supprimer la variable  $v$  de la grammaire ainsi que toutes les productions qui y font référence.

Le processus ne modifie pas les mots du langage puisque si un mot dérivé contient une variable  $v$  non productive, alors il ne peut pas se dériver en un mot terminal. Par ailleurs les dérivations terminales ne sont pas touchées.

**Proposition 11.4.3** *Pour tout langage algébrique  $L$ , on peut construire une grammaire hors-contexte telle que pour toute variable  $v$  il existe une dérivation terminale où elle intervient :  $S \xrightarrow{*} fvg \xrightarrow{*} fwg$ , où  $v \in V$  et  $f, g, w \in T^*$ .*

Preuve : Dans la grammaire précédente, toutes les variables produisent un mot terminal, donc si  $S \xrightarrow{*} fvg$ , on en déduit que  $v$  satisfait la propriété. Il suffit donc de rechercher systématiquement les variables accessibles à partir de l'axiome par une suite de dérivations, ce qui se fait en considérant le graphe  $G$  dont les sommets sont les variables de  $\mathcal{G}$  et les arêtes sont telles que  $(v, v')$  existe si et seulement s'il existe une règle dans  $\mathcal{G}$  de la forme  $v = fv'g$ . On cherche alors dans le graphe  $G$  les sommets accessibles de  $S$  l'axiome, par exemple en construisant un arbre de Trémaux. Les sommets non accessibles sont retirés de la grammaire ainsi que les productions qui les contiennent.

Enfin il est possible de supprimer les simples récritures de variables du type  $v = v'$  en les remplaçant par les membres droits des productions récrivant  $v'$ .

**Proposition 11.4.4** *Pour chaque grammaire, on peut construire une grammaire équivalente dans laquelle il n'y a pas de règle du type  $v = v'$ .*

Les transformations précédentes permettent de simplifier notablement les grammaires et d'en éliminer les variables scories. Elles ne permettent pas de trouver facilement un arbre de dérivation potentiel pour un mot si celui-ci appartient au langage

engendré. Une étape importante, que nous retrouverons en compilation, est d'éliminer les appels récursifs gauches, ce qui revient sur la grammaire à faire que tous les membres droits de règle commencent par un symbole terminal. Commençons par deux lemmes.

**Lemme 11.4.1** *Dans une grammaire hors-contexte où  $v' = \gamma_1 + \gamma_2 + \dots + \gamma_k$  est un ensemble de règles, on peut remplacer les règles du type  $v = \alpha v' \beta$  par  $v = \alpha \gamma_1 \beta + \alpha \gamma_2 \beta + \dots + \alpha \gamma_k \beta$  sans changer le langage engendré.*

Preuve : On note qu'en procédant ainsi on ne fait que compresser deux récritures en une seule, et donc que le résultat est inchangé.

Le second lemme est plus intéressant car il va permettre de transformer des récurrences droites en des récurrences gauches. Il repose sur une propriété élémentaire des langages rationnels et des grammaires qui les engendrent.

**Lemme 11.4.2** *Soit  $v$  une variable dont les récritures possibles sont données par les règles  $v = v\alpha_1 + v\alpha_2 + \dots + v\alpha_k + \beta_1 + \beta_2 + \dots + \beta_l$ , où les  $\alpha_i$  et  $\beta_j$  sont dans  $(V \cup T)^*$ . Alors, on ne modifie pas le langage engendré en les remplaçant par les deux jeux de règles :*

$$\begin{aligned} v &= \beta_1 + \beta_2 + \dots + \beta_l + \beta_1 \hat{v} + \beta_2 \hat{v} + \dots + \beta_l \hat{v} \\ \hat{v} &= \alpha_1 + \alpha_2 + \dots + \alpha_k + \alpha_1 \hat{v} + \alpha_2 \hat{v} + \dots + \alpha_k \hat{v}. \end{aligned}$$

Preuve : Il suffit de vérifier que les deux jeux de règles engendrent également l'ensemble

$$(\beta_1 + \beta_2 + \dots + \beta_l)(\alpha_1 + \alpha_2 + \dots + \alpha_k)^*.$$

Pour ce faire, on se restreint à l'étude des dérivations gauches de la grammaire qui produisent, comme on le sait, l'ensemble des mots du langage.

On est maintenant en position de conclure et de prouver le théorème de Greibach.

**Théorème 11.4.1** *Tout langage algébrique peut être engendré par une grammaire où toutes les règles sont du type*

$$v = aw, \text{ où } v \in V, a \in T, w \in V^*.$$

Preuve :

1. Commençons par mettre la grammaire sous la forme suivante

$$v = a_1 w_1 + a_2 w_2 + \dots + a_k w_k + w'_1 + w'_2 + \dots + w'_l, \text{ où } v \in V, a_i \in T, w_j, w'_j \in V^*.$$

Ceci est toujours possible en créant de nouvelles variables de telle sorte que si  $v = a_1 h_1 a_2 h_2 \dots a_k h_k$ , avec  $a_i \in T$  et  $h_j \in V^*$ , alors on remplace cette règle par :

$$v = a_1 h_1 u_1, u_1 = a_2 h_2 u_2, \dots, u_{k-1} = a_k h_k.$$

Si le membre droit ne commence pas par un terminal, la même démarche conduit à la forme annoncée.

2. Les règles du type  $v = aw$  sont maintenant conservées dans la grammaire finale.

3. Il faut maintenant traiter les autres pour éliminer les récurrences à gauche.

3.1 Soit  $V$  l'ensemble des variables. On va transformer les règles, au prix de l'ajout de quelques variables auxiliaires  $Z$ , pour faire en sorte que se dégage un ordre  $V' = \{v_1, v_2, \dots, v_k, \hat{v}_1, \hat{v}_2, \dots, \hat{v}_k\} = V \cup \hat{V}$  tel que dans les productions de type  $v = w$ ,  $w$  commence par une variable de rang supérieur strictement à celui de  $v$ .

Par un algorithme inspiré de celui du tri topologique, on peut trouver un début d'ordre,  $v_1 < v_2 < \dots < v_i$  satisfaisant cette propriété, et si  $i = k$  alors la première phase de la transformation est achevée.

Sinon, soit  $v_i$  la première variable telle qu'il existe  $v_i = v_j w$ , avec  $j \leq i$ . Alors comme  $v_j$  vérifie la propriété que toutes ses productions commencent par des  $v_{j'}$ , avec  $j' > j$  ou des terminaux, on peut faire la substitution de ces membres droits dans ceux des règles attachées à  $v_i$  et on répète le processus jusqu'à ce que les variables de tête des productions de  $v_i$  soient toutes d'indice au moins  $i$ . La grammaire est équivalente par le lemme 11.4.1.

3.2 On est maintenant en présence de récurrences gauches du type  $v = vw + v'w'$  ( $v'$  est d'indice supérieur à  $v$  ou est terminal) et qui sont donc passibles du lemme 11.4.2 et sont transformées en  $v = v'w'\hat{v} + v'w'$  et  $\hat{v} = w + w\hat{v}$ .

Considérons alors la dernière variable  $v_k$  : comme il n'y a pas de  $v$  d'indice supérieur toutes ses productions commencent par un terminal. Par récurrence, les productions de  $v_{k-1}$  commencent soit par un terminal soit par  $v_k$  et par substitution (lemme 11.4.1) on les fait toutes commencer par un terminal. En itérant, toutes les productions des variables originelles commencent donc par un terminal.

Enfin, on remarque que les productions des variables additionnelles commencent toujours par un terminal ou une variable originelle ; par substitution encore, on se ramène à avoir toujours un terminal en tête de règle.

Ce qui prouve le théorème.

### 11.4.2 Automates à pile

Le théorème de forme normale de Greibach permet alors un passage facile à une famille d'automates reconnaissant les langages algébriques. (On pourrait aussi le faire directement).

Commençons par deux exemples.

Pour reconnaître si un mot  $w$  est dans le langage  $M = \{fcf^R \mid f \in (a+b)^*\}$  et qui est évidemment algébrique, on peut procéder ainsi :

- lire le mot  $f$  jusqu'au  $c$  et le stocker dans une liste chaînée  $\pi(f)$  ;
- poursuivre la lecture après le  $c$  et comparer chaque lettre lue de la donnée avec la lettre précédente de la liste  $\pi(f)$  ;
- si une discordance apparaît dans le processus, le mot n'est pas dans  $M$  ; si, au contraire on épuise la donnée en même temps que l'on atteint le début de la liste  $\pi$ , le

mot est bien dans  $M$ .

On a donc un procédé déterministe de reconnaissance des mots de  $M$ .

Considérons le langage  $D$  engendré par la grammaire  $S = (S)S + \varepsilon$  et qui est donc constitué des mots bien parenthésés. On peut reconnaître qu'un mot  $f$  est dans  $D$ , par exemple, en recherchant les paires de parenthèses homologues consécutives  $()$  et en les supprimant. Il est facile de montrer, par récurrence, que ce processus doit réduire le mot  $f$  de départ au mode vide  $\varepsilon$  si et seulement si  $f \in D$ .

Cette méthode est cependant coûteuse à mettre en place algorithmiquement et on doit explorer d'autres voies qui permettent de ne pas revenir en arrière sur la donnée. Le principe est de mémoriser encore dans une liste doublement chaînée les informations qui seront exploitées un peu plus loin (voire même très loin) dans le mot, à savoir les parenthèses ouvrantes et à les mettre en correspondance avec la fermante correspondante, car on montre facilement qu'elle est unique. Le dessin ci-dessous indique simplement le niveau de parenthésage courant au fil de la lecture.

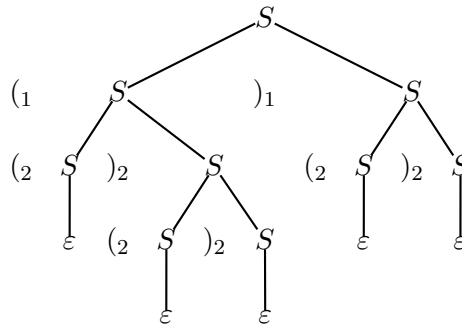


FIG. 11.4 – Un arbre de dérivation pour  $(1(2)2(2)2)1(1)1$  : on a étiqueté les parenthèses pour mettre en évidence les paires ouvrante-fermante homologues et leur hauteur dans la pile.

On observe donc que dans les deux cas, on peut reconnaître les mots du langage en les lisant séquentiellement de gauche à droite en gardant de l'information dans une liste doublement chaînée sur laquelle on travaille *toujours sur la même extrémité* aussi bien en adjonction qu'en suppression. Les décisions sont prises en fonction de la lettre courante du mot analysé et de ce qui est en tête de liste. On s'interdit de venir regarder dans la liste. Une telle structure de données est bien sûr une *pile* et c'est elle qui permet de caractériser les langages algébriques.

**Définition 11.4.1** On appelle automate à pile un dispositif défini par :

- un ensemble fini d'états  $Q$  ;
- $q_0 \in Q$ , l'état initial ;
- $T$  l'alphabet de lecture (fini) ;
- $P$  l'alphabet de pile (fini) ;
- $p_0 \in P$  est le symbole initial de pile ;
- la table de transitions  $\Delta : Q \times (T \cup \{\varepsilon\}) \times P \rightarrow \mathcal{P}(Q \times P^*)$ .

Le calcul d'un automate à pile est défini par une suite de *configurations* compatible avec la table de transitions. On notera bien que ces automates ne sont pas déterministes *a priori*.

Une configuration d'un automate est un triplet  $(f, q, \pi)$ , où  $f \in T^*$  est le suffixe de la donnée qui reste à lire,  $q \in Q$  est l'état courant et  $\pi \in P^*$  est le mot de pile : par convention l'extrémité accessible est la première lettre de  $\pi$ .

Pour en décrire le fonctionnement, il est commode d'introduire les fonctions `push` et `pop` sur la pile, et d'utiliser les notations Caml (`hd` et `tl`) pour la gestion des files :

— la fonction `pop` a pour effet de supprimer la lettre de sommet de pile (l'extrémité accessible de la pile) :  $\text{pop}(\pi) = (\text{tl } \pi)$ , mais on ne peut pas faire de `pop` sur une pile vide car il y a alors erreur :

— la fonction `push(w)` exécute d'abord un `pop`, puis empile le mot  $w$ , de telle sorte que  $\text{push}(w, \pi) = w @ \text{pop}(\pi)$ , à condition que la pile ne soit pas vide. Remarquons que  $\text{pop} \equiv \text{push}(\varepsilon)$ .

On passe de la configuration  $(f, q, \pi)$  à la configuration  $(f', q', \pi')$  à condition qu'il existe une règle dans la table  $\Delta$ ,  $(q, \alpha, p) \rightarrow (q', w)$ , où  $\alpha$  peut être  $\varepsilon$  est un mot (éventuellement vide) sur l'alphabet de pile et telle que :  $f = \alpha f'$  et  $\pi' = \text{push}(w, \pi)$ .

Un calcul sur le mot  $f$  commence par la configuration  $(f, q_0, p_0)$ .

*Si la pile est vide en même temps que le mot de donnée est lu, l'automate s'arrête avec succès. Si aucune règle n'est applicable le calcul n'est pas valide. Un mot est accepté par l'automate si et seulement s'il existe une suite de configurations qui l'amène à la pile vide. Il faut remarquer qu'il n'y a pas unicité de la suite de calculs, que pour un même mot, il peut exister une ou même des suites qui l'acceptent et d'autres qui ne le font pas. Il s'agit ici encore d'une conséquence irréductible du non-déterminisme.*

Voici un exemple d'automate à pile :

$T = \{a, b\}, Q = \{0, 1, 2\}, P = \{b, c\}, q_0 = 0, p_0 = c$

$\Delta = \{(0, a, c) \rightarrow (0, c), (0, b, c) \rightarrow (0, bc), (0, b, b) \rightarrow (0, bb), (0, b, b) \rightarrow (1, bb) \text{ (on empile les } b)\}$

$(0, b, b) \rightarrow (1, bb) \text{ (on change de mode)}$

$(1, a, b) \rightarrow (1, \varepsilon) \text{ (on compte autant de } a \text{ que de } b)$

$(1, b, c) \rightarrow (2, c) \text{ (on a trouvé le motif)}$

$(2, a, c) \rightarrow (2, c), (2, b, c) \rightarrow (2, c), (2, b, c) \rightarrow (2, \varepsilon)\}$  (pour terminer le mot).

Cet automate reconnaît les mots de la forme  $a^*ba^{i_1}ba^{i_2}b\dots ba^{i_k}b$  pour lesquels il existe un  $j$  tel que  $j = i_j$ . On voit bien que ce processus est fondamentalement non-déterministe, car il existe des mots acceptant plusieurs analyses correctes.

**Théorème 11.4.2** *La class Alg est identique à celle des langages reconnaissables par automate à pile.*

Preuve :

Considérons un langage  $L$  engendré par une grammaire en forme de Greibach  $\mathcal{G}$ . Les règles de cette grammaire sont donc de la forme  $v = av_1v_2\dots v_k$  ou  $v = a$ , pour  $a \in T$  lettre de l'alphabet terminal et pour des  $v$  et  $v_i \in V$  variables.

Construisons un automate à pile comme suit :  $T$  est son alphabet d'entrée,  $V$  est son alphabet de pile et  $S$  (l'axiome de  $\mathcal{G}$ ) est son symbole de début de pile,  $Q$  et réduit à

un seul état qu'on ne note donc pas. La table de transition est formée des deux types de règles trivialement déduites de celles de  $\mathcal{G}$  :

— pour  $v = a$  de  $\mathcal{G}$ ,  $\Delta$  contient  $(-, a, v) \rightarrow (-, \varepsilon)$ , ce qui revient à lire  $a$  tout en dépilant  $v$  ;

— pour  $v = av_1v_2\dots v_k$  de  $\mathcal{G}$ ,  $\Delta$  contient  $(-, a, v) \rightarrow (-, v_1v_2\dots v_k)$ , ce qui revient à lire  $a$  tout en remplaçant  $v$  par  $v_1v_2\dots v_k$  en sommet de pile,  $v_1$  devenant le nouveau sommet.

Considérons alors une dérivation gauche d'un mot  $f$  dans  $\mathcal{G}$ . Les mots produits par les récritures successives sont de la forme  $hw$ , où  $h \in T^*$  est un préfixe de  $f$  et  $w \in V^*$  fournira le suffixe  $g$  de  $f = hg$ . Un calcul valide de l'automate défini ci-dessus sur le mot  $f$  est celui où les mots de pile de sont précisément les  $w$  successifs de la dérivation gauche et chaque lettre de  $f$  est lue au moment où elle est produite par une récriture du non-terminal gauche. Donc le mot est accepté par l'automate. Réciproquement tout calcul de l'automate acceptant un mot est immédiatement interprétable en terme de dérivation gauche de ce mot, par construction même.

Inversement, on construit pour tout automate à pile un automate équivalent sans états, ceux-ci étant mémorisés dans la pile. L'automate résultant est alors semblable à celui construit dans la partie directe ; on construit par le même mécanisme (mais en sens inverse) une grammaire hors-contexte dont il est évident qu'elle engendre le même langage.

D'où le théorème.

## 11.5 Analyse par programmation dynamique

Dans cette section, on montre l'existence d'un algorithme déterministe en temps polynomial pour décider si un mot est dans le langage engendré par une grammaire ou non.

### 11.5.1 Forme normale de Chomsky

Une autre normalisation des productions d'une grammaire hors-contexte est d'une grande utilité. Elle permet de rendre plus réguliers les arbres de dérivations en rendant essentiellement leurs sommets internes binaires. Les productions d'une telle grammaire sont de l'une des deux formes suivantes :  $A = BC$  pour  $A, B, C, \in V$  ou  $A = a$  pour  $a \in T$ . On voit donc que les nœuds binaires de l'arbre correspondent à des récritures d'une variable en deux variables, alors que les nœuds unaires sont une récriture terminale produisant une lettre du mot engendré.

**Proposition 11.5.1** *Pour toute grammaire hors-contexte, il existe une grammaire équivalente sous forme normale de Chomsky.*

Preuve : On peut supposer, sans restriction, que la grammaire  $\mathcal{G}$  est propre, c'est-à-dire que toutes les variables sont utiles, qu'il n'y a pas de règle du type  $A = B$ , pour  $A, B \in V$ , et qu'à l'exception de l'axiome aucune ne produit le mot vide. Toutes ces constructions ont été vues précédemment. On transforme alors les règles de  $\mathcal{G}$  comme

suit, en deux temps, en introduisant autant de nouvelles variables que nécessaire :

1. chaque règle de la forme  $A = a$ ,  $A \in V$  et  $a \in T$ , est conservée ;
2. chaque règle de la forme  $A = w$ ,  $A \in V$  et  $w \in (V \cup T)^*$ , est remplacée par  $A = W_1W_2\dots W_{|w|}$ , où les  $W_i$  soit sont  $W_i = w_i$  si  $w_i \in V$ , soit sont de nouvelles variables si  $w_i \in T$  auquel cas on ajoute la règle  $W_i = w_i$  ;
3. les règles  $A = W_1W_2\dots W_k$  ainsi créées sont ensuite normalisées pour  $k > 2$  et transformées en l'ensemble de règles binaires :  $A = W_1W'_2; W'_2 = W_2W'_3; \dots; W'_{k-1} = W_{k-1}W_k$ .

Ainsi la nouvelle grammaire est en forme de Chomsky et il est évident qu'elle est strictement équivalente à  $\mathcal{G}$ . Remarquons de plus que sa taille est de l'ordre de 4 fois celle de  $\mathcal{G}$ .

### 11.5.2 Reconnaissance déterministe des langages algébriques

La mise en forme de Chomsky des grammaires hors-contexte permet de programmer très simplement l'analyse des mots du langage. En effet, définissons la fonction  $Drv : V \times T^* \rightarrow \{0, 1\}$ , où  $\{0, 1\}$  sont les valeurs de vérité faux et vrai comme suit, en utilisant pour convention que les variables sont représentées par des capitales ( $A, B, C, \dots \in V$ ) et que les mots sur l'alphabet terminal le sont par des minuscules ( $f, g, h, \dots \in T^* \setminus \{\varepsilon\}$ ). Alors :

$$Drv(A, f) = \begin{array}{ll} & \{\text{pour } f \neq \varepsilon\} \\ \text{si } f \in T & \text{alors } (A = f) \in \mathcal{R}? \\ \text{sinon} & \exists g, h \neq \varepsilon \exists A, B, C \\ & \{ f = gh \ \& \ A = BC \in \mathcal{R} \ \& \ Drv(B, g) \ \& \ Drv(C, h) \}. \end{array}$$

Une telle définition se traduit immédiatement dans tout langage de programmation puisqu'il suffit de programmer les deux quantificateurs existentiels par deux boucles :  
— la première sur les césures de  $f$  en  $f = gh$  ;  
— la seconde sur toutes les règles de  $\mathcal{R}$  dont le membre gauche est la variable  $A$ .  
Voici comment on peut l'exprimer en pseudocode, en utilisant les fonctions booléennes de test d'appartenance à l'alphabet (`estLettre`) et à l'ensemble  $\mathcal{R}$  des règles de la grammaire (`estRegle`) :

```
// R est une liste de listes [A B C] ou [A a]
derive(A, f)
// retourne true si et seulement si A produit f
si (longueur(f) == 1) ET estLettre(tête(f)) alors
    retourner estRegle([A, tête(f)]);
sinon
    retourner itererDecoupe(A, [tête(f)], queue(f));

itererDecoupe(A, g, f)
// recherche une césure correcte pour A->f
```

```

si f == [] alors
  retourner false;
sinon
  si itererRegle(A, R, g, f) alors
    retourner true;
  sinon
    retourner itererDecoupe(A, g # [tête(f)], queue(f));

itererRegle(A, R, g, h)
// recherche une règle correcte pour A -> gh
si R == [] alors
  retourner false;
sinon
  r <- tête(R); R1 <- queue(R);
  si A == r ET longueur(queue(r)) == 2 alors
    r = (r1, r2);
    si derive(r1, g) ET derive(r2, h) alors
      retourner true;
    sinon
      retourner itererRegle(A, R1, g, h);
  sinon
    retourner itererRegle(A, R1, g, h);

```

La programmation est donc simple en C, Caml ou Java, mais elle sera en général très inefficace car le nombre de façons de décomposer en facteurs analysables est essentiellement proportionnel au nombre d'arbres binaires à  $n$  sommets, ou encore de parenthésages à  $n$  paires de parenthèses, quantité appelée *nombre de Catalan*<sup>1</sup> et valant

$$C_n = \frac{1}{2n+1} \binom{2n}{n} \approx \frac{1}{\sqrt{\pi}} 4^n n^{-3/2}.$$

Le coût d'exécution de ce programme est donc exponentiel !

Il existe pourtant deux façons de faire tomber sa complexité dans le domaine du raisonnable polynomial. Elles reposent toutes deux sur la remarque que l'on répète un très grand nombre de fois les mêmes appels récursifs, et donc qu'il faut éviter de reprendre ces calculs soit en les mémorisant soit en les organisant mieux.

### Mémo-fonctions

Cette méthode est utilisée pour accélérer l'accès aux valeurs déjà calculées d'une fonction ; elle est particulièrement efficace lorsque la définition est hautement récursive et permet, moyennant très peu de modifications au code d'en accélérer prodigieusement l'exécution. Prenons par exemple le cas de la fonction qui calcule les nombres du triangle

<sup>1</sup>C'est un équivalent de ces nombres dans le cas ternaire que l'on a vu apparaître au chapitre 1 lors du dénombrement des fonctions booléennes.

de Pascal selon leur définition récursive standard :  $\binom{n}{0} = 1$ ,  $\binom{n}{p} = 0$  si  $p > n$ ,  $\binom{n+1}{p} = \binom{n}{p-1} + \binom{n}{p}$ .

Utilisons un tableau memo de longueur suffisante pour stocker les  $\frac{(N+1)(N+2)}{2}$  valeurs de la fonction jusqu'au rang  $N$ . On initialise ce tableau à 0 et on rangera la valeur de  $\binom{n}{p}$  en position  $\frac{(n+1)(n+2)}{2} + p$ , commençant ainsi à l'adresse 1..

Le programme initial pour la fonction comb,

```

comb(n, p)
si n == 0 alors
  retourner 1;
sinon
  si p > n alors
    retourner 0;
  sinon
    retourner comb(n-1, p-1) + comb(n-1, p);

```

est changé en

```

memo <- tableau de taille 10000;

mcomb(n, p)
si n == 0 alors
  retourner 1;
sinon
  si p > n alors
    retourner 0;
  sinon
    vm <- memo[(n+1)*(n+2)/2+p];
    si vm != 0 alors
      retourner vm;
    sinon
      vm <- mcomb(n-1, p-1)+mcomb(n-1, p);
      memo[(n+1)*(n+2)/2+p] <- vm;
    retourner vm;

```

Nous avons utilisé ici le fait qu'il existait une bijection simple entre les éléments du triangle de Pascal et le vecteur memo. Si tel n'est pas le cas, on peut avoir recours à une table de hachage qui permet de ranger un ensemble difficilement ordonnable et de retrouver facilement les éléments déjà rangés, la structure du nouveau programme restant la même.

Dans le cas de la reconnaissance d'un langage algébrique par la fonction `derive`, on doit donc construire une table à double entrée, fonction de l'ensemble des non-terminaux de la grammaire  $V$  et de l'ensemble des facteurs du mot  $f$  qui sont en nombre  $|f|^2$ . Chaque appel de la fonction `derive` sur un mot de longueur  $p$  demande donc une lecture de la table des productions de la grammaire et la recherche des valeurs

de la fonction pour les  $p$  décompositions possibles de la donnée. En conséquence le coût de cette implantation de l'algorithme est  $O(|\mathcal{G}| \cdot |f|^3)$ , ce qui est polynomial et donc acceptable.

### La programmation dynamique : Cocke, Kasami, Younger.

On peut aussi programmer cet algorithme de recherche d'un arbre de dérivation de façon entièrement tabulaire et complètement itérative, en utilisant un des paradigmes classiques de programmation, appelé *programmation dynamique*, très en vogue en mathématiques appliquées. Cela consiste essentiellement à démonter complètement le mécanisme des appels récursifs, puis à reconstruire le calcul à partir des appels terminaux (sur des mots de 1 lettre), puis en remontant aux facteurs de 2 lettres, puis 3 et ainsi de suite jusqu'à atteindre le mot  $f$  de départ de longueur  $n$ . On utilise alors un tableau  $T$  à deux dimensions de taille  $n \times n$ , dont les éléments sont des listes de non-terminaux de la grammaire  $\mathcal{G}$  : chaque liste est composée des non-terminaux qui peuvent engendrer un facteur bien déterminé du mot, selon le schéma suivant.

La première ligne contient les variables qui engendrent les lettres du mot  $f$ , en utilisant donc les règles du type  $A \rightarrow a$  :

$$T[1, i] = \{A \mid A \rightarrow f_i \in \mathcal{R}\}.$$

Dans la seconde ligne on assemble les informations de la première, pour déterminer comment on peut former les digrammes (deux lettres consécutives) de  $f$ , selon la règle :

$$\begin{aligned} T[2, i] &= \{A \mid A \rightarrow f_i f_{i+1} \in \mathcal{R}\} \\ &= \{A \mid \exists A \rightarrow BC \in \mathcal{R} \ \& \ B \in T[1, i] \ \& \ C \in T[1, i+1]\}. \end{aligned}$$

Avant de donner la règle générale, considérons la troisième ligne où se détermine la génération des trigrammes (trois lettres consécutives) de  $f$ , à partir des deux précédentes, selon la règle :

$$\begin{aligned} T[3, i] &= \{A \mid A \rightarrow f_i f_{i+1} f_{i+2} \in \mathcal{R}\} \\ &= \{A \mid \exists A \rightarrow BC \in \mathcal{R} \ \& \\ &\quad ((B \in T[1, i] \ \& \ C \in T[2, i+1]) \vee ((B \in T[2, i] \ \& \ C \in T[1, i+2])))\}. \end{aligned}$$

De façon générale, la  $k$ -ième ligne contient les non-terminaux possibles pour les facteurs de longueur  $k$  du mot  $f$  et se calcule comme suit à partir des lignes précédentes (on a évidemment  $k+i \leq n+1$ ) :

$$\begin{aligned} T[k, i] &= \{A \mid A \rightarrow f_i f_{i+1} \dots f_{i+k-1} \in \mathcal{R}\} \\ &= \{A \mid \exists A \rightarrow BC \in \mathcal{R}, \ \exists l < k, \ B \in T[l, i] \ \& \ C \in T[k-l, i+l-1]\}. \end{aligned}$$

Le processus est itéré jusqu'à la  $n$ -ième ligne du tableau dans laquelle on ne remplit que la case  $T[n, 1]$ . Par construction ce processus n'est que la mise en forme itérative de la méthode récursive et en conséquence, le mot  $f$  appartient au langage  $\mathcal{L}(\mathcal{G})$  si et seulement si l'axiome  $S$  de la grammaire appartient à l'ensemble  $T[n, 1]$ .

Le tableau  $T$  comporte  $\frac{n(n+1)}{2}$  cases effectivement utilisées, et le temps de calcul pour une cellule de la  $k$ -ième ligne est  $O(|\mathcal{G}|.k)$  ; en conséquence le temps total est cubique en  $|f|$ .

**Théorème 11.5.1** *Les deux algorithmes ci-dessus déterminent l'appartenance d'un mot de longueur  $n$  à un langage algébrique en temps  $O(n^3)$ .*

### Borne inférieure

Dans la pratique les algorithmes de coût cubique sont encore trop complexes ; on verra dans la suite du chapitre qu'en imposant des restrictions à la grammaire, on peut souvent faire l'analyse en temps linéaire. Dans le cas général, le meilleur résultat connu est dû à L. Valiant qui a montré que l'on pouvait ramener le problème à celui de la résolution de systèmes linéaires et donc à la multiplication matricielle, problème connu depuis V. Strassen pour être moins que cubique.

Voici un point récent sur la question par Steven Finch :

“ Define the exponent of matrix multiplication  $\omega$  as the infimum of all real numbers  $\tau$  such that multiplication of  $n \times n$  matrices may be achieved with  $O(n^\tau)$  multiplications. Clearly  $\omega \leq 3$  and it can be proved that  $\omega \geq 2$ .

Strassen discovered a surprising base algorithm to compute the product of  $2 \times 2$  matrices with only seven multiplications. The technique can be recursively extended to large matrices via a tensor product construction. In this case, the construction is very simple : large matrices are broken down recursively by partitioning the matrices into quarters, sixteenths, etc. This gives  $\omega \leq \ln(7)/\ln(2) < 2.808$ .

More sophisticated base algorithms and tensor product constructions permit further improvements. Many researchers have contributed to this problem, including Pan who found  $\omega < 2.781$  and Strassen who found  $\omega < 2.479$ ...

Coppersmith & Winograd presented a new method, based on a combinatorial theorem of Salem & Spencer, which gives dense sets of integers containing no three terms in arithmetic progression. They consequently obtained  $\omega < 2.376$ , which is the best known upper bound today.

Is  $\omega = 2$  ? Bürgisser called this the central problem of algebraic complexity theory. ”

## 11.6 Algèbricité

De la même façon que l'on a caractérisé les langages rationnels par leur propriété de finitude de leurs classes d'indiscernabilité, on caractérise les langages algébriques par leur système de parenthésages. Nous allons d'abord démontrer un analogue du lemme d'itération, puis nous établirons que le langage des parenthèses est le prototype de toute la famille  $\mathcal{Alg}$ .

### 11.6.1 Le lemme d'itération des langages algébriques

La proposition 3.3 traitait de la finitude des langages algébriques ; réinterprétons-la sous un autre angle. Supposons donc qu'une variable  $A$  se répète sur une branche

d'un arbre de dérivation, comme illustré par la figure 11.5, et que cette même variable produise également un mot  $y$  non vide sur l'alphabet terminal. Alors on peut couper le sous-arbre intermédiaire ou bien le recopier à volonté, obtenant de nouveaux arbres de dérivation valides comme le montre la figure 11.6.

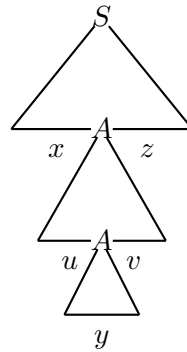


FIG. 11.5 – La répétition d'une variable sur une branche.

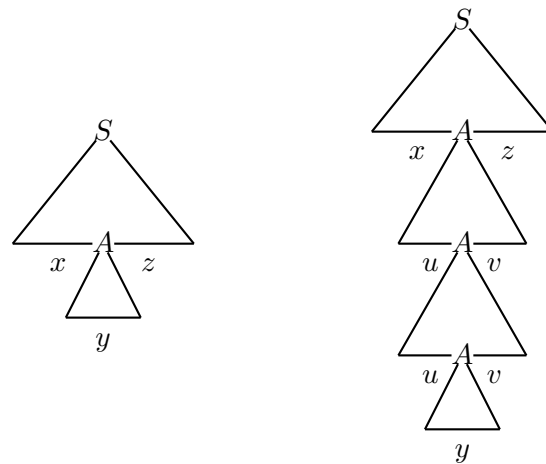


FIG. 11.6 – Itérations possibles d'une variable sur une branche : 0 fois à gauche et 2 fois à droite.

Le lemme d'itération pour les langages algébriques s'énonce donc comme suit.

**Théorème 11.6.1** [Bar-Hillel, Perles, Shamir]

Soit  $L$  un langage algébrique infini ; il existe un entier  $\lambda$  tel que pour tout mot  $w \in L$  assez grand ( $|w| \geq \lambda$ ), le mot  $w$  se décompose en

$$w = xuyvz, \quad |uv| \geq 1,$$

et le langage  $L$  admet pour sous-ensemble

$$\{xu^n yv^n z \mid n \geq 0\} \subset L.$$

Preuve : Si  $L$  est infini, les arbres de dérivation sont de profondeur non bornée (car le degré de chaque sommet est borné) ; les branches sont de longueur non bornée et quand celle-ci dépasse le nombre de variables, l'une d'entre elles au moins apparaît deux fois. De plus, on peut choisir une grammaire sans  $\varepsilon$ -productions et débarrassée des productions du type  $A \rightarrow B$ . La situation est alors celle de la figure 11.5 et le sous-arbre de racine  $A$  produisant le mot  $uAv$ ,  $uv \neq \varepsilon$ , peut être itéré arbitrairement :

$$A \xrightarrow{*} uAv \xrightarrow{*} uuAvv \xrightarrow{*} uuuAvvv \xrightarrow{*} u^n Av^n \xrightarrow{*} \dots,$$

laquelle dérivation combinée avec  $S \xrightarrow{*} xAy$  et  $A \xrightarrow{*} y$  conduit à l'énoncé.

On notera que seule la contrainte  $uv \neq \varepsilon$  est imposable ; en effet, le langage peut parfaitement être rationnel et alors l'un des deux  $u$  ou  $v$  peut être vide. Enfin la valeur de  $\lambda$  peut être fixée à  $\kappa^{|V|}$ , où  $\kappa$  est la longueur de la plus grande production.

On en déduit alors que certains langages ne peuvent pas être décrits par des grammaires hors-contexte. C'est le cas de tous les langages qui comportent des systèmes de parenthèses croisés ou encore dont les contraintes correspondent à des nœuds.

**Proposition 11.6.1** *Le langage  $\{a^n b^n c^n \mid n \geq 0\}$  n'est pas algébrique.*

Preuve : S'il l'était, pour  $n$  assez grand on aurait, par le lemme d'itération, une décomposition du type  $a^n b^n c^n = xyvz$  et une contradiction car en itérant  $u$  et  $v$ , on ne peut pas maintenir la forme globale des mots, comme on l'a vu pour les langages rationnels.

On en déduit donc en particulier que l'intersection de deux langages algébriques ne l'est pas forcément.

### 11.6.2 Langage générateur

Pour terminer, nous allons montrer que tout langage algébrique peut s'obtenir à partir d'un seul et même langage appelé *langage générateur* par application des deux opérations de morphisme, et intersection avec un langage rationnel.

**Définition 11.6.1** *Soit  $X_n = \{(, )_i \mid 1 \leq i \leq n\}$  un alphabet à  $2n$  symboles. On appelle langage de Dyck  $D_n^*$  l'ensemble des mots bien parenthésés sur cet alphabet et qui est défini par la grammaire d'axiome  $S$  :*

$$S = \sum_{1 \leq i \leq n} (, )_i S + \varepsilon.$$

Étant donné un langage  $L \subset T^*$  dont la grammaire  $\mathcal{G} = \langle V, T, S, \Delta \rangle$  est sous forme normale de Chomsky, nous allons montrer qu'il existe un homomorphisme  $h$  et un langage rationnel  $K$  tels que, pour un certain  $n$ ,

$$L = h(D_n \cap K).$$

En fait nous allons utiliser un ensemble de parenthèses  $P$  indicées de façon plus complexe (et surtout plus parlante), et nous considérerons le langage  $D_P$  des mots bien parenthésés sur  $P$  mais il sera toujours possible de les renuméroter sur  $\mathbb{N}$ . Il est défini par les trois règles suivantes.

1. Les lettres de l'alphabet terminal  $T$  seront représentées par la paire de parenthèses  $(a)_a$ , pour tout  $a \in T$ .
2. Chaque usage d'une règle terminale du type  $A \rightarrow a$  sera rendu par une paire de parenthèses  $\{^a_A\}_A^a$ .
3. Chaque usage d'une règle terminale du type  $A \rightarrow BC$  sera rendu par un mot de parenthèses  $\{^BC_A\}_A^{BC}$ .

Considérons le langage  $\hat{L}$  défini par la grammaire  $\hat{\mathcal{G}} = \langle V \cup T, P, S, \hat{\Delta} \rangle$  dont les productions sont définies par :

1. pour tout  $a \in T$ ,  $a \rightarrow (a)_a$  ;
2. pour tout  $A \rightarrow a \in \Delta$ ,  $A \rightarrow \{^a_A a\}_A^a$  ;
3. pour tout  $A \rightarrow BC \in \Delta$ ,  $A \rightarrow \{^BC_A\}_A^{BC}$ .

Associons alors à toute dérivation d'un mot  $w$  dans  $L$ , le mot de parenthèses obtenues en appliquant les règles homologues de  $\hat{\mathcal{G}}$  ; le mot obtenu  $\hat{w}$  contient à l'évidence toute la dérivation de  $w$  et permet de le retrouver en effaçant toutes les parenthèses autres que les  $(a)$ . Cette propriété se vérifie par récurrence sur les dérivations ; donc

$$L = h(\hat{L}),$$

où  $h$  est ce morphisme effaceur. De plus, on peut faire un certain nombre de déductions de proximité.

R1 : Tout mot de  $\hat{L}$  dérivé par exemple d'une règle  $A \rightarrow a \in \Delta$ , est forcément du type  $\{^a_A (a)_a\}_A^a$ , contraintes qui sont rationnelles car locales.

R2 : Tout mot de  $\hat{L}$  dérivé par exemple d'une règle  $A \rightarrow BC \in \Delta$ , commence par  $\{^BC_A\}_A^{BC}$ , se termine par  $\}_A^{BC}$  et doit contenir un digramme  $\}_A^{BC} \{^BC_A$  à l'exclusion de toute autre combinaison.

R3 : Dans tout mot du langage  $\hat{L}$  :

- une lettre de type  $\{^BC_A$  doit être suivie d'une lettre de type  $\{^x_B\}$  où  $x$  est une lettre de  $T$  ou un digramme de  $V^2$  ;
- une lettre de type  $\}_A^{BC}$  doit être suivie d'une lettre de type  $\}_C^x$  où  $x$  est une lettre de  $T$  ou un digramme de  $V^2$  ;
- une lettre de type  $\}_A^{BC}$  doit être précédée d'une lettre de type  $\}_B^x$  où  $x$  est une lettre de  $T$  ou un digramme de  $V^2$  ;
- une lettre de type  $\}_A^{BC}$  doit être précédée d'une lettre de type  $\}_C^x$  où  $x$  est une lettre de  $T$  ou un digramme de  $V^2$ .

R4 : Les mots du langage  $\hat{L}$  comencent par une lettre du type  $\{^BC_S$  ou  $\{^a_S$  et se terminent par une  $\}_S^{BC}$  ou  $\}_S^a$ .

Toutes ces contraintes sont exprimables par un automate fini qui définit donc sur l'alphabet  $P$  un langage  $K$  et l'on a  $\hat{L} \subset D_P \cap K$ .

Réciproquement, soit un mot  $\hat{w}$  de  $K_P = D_P \cap K$  ; il est donc mot de parenthèses, et de plus il vérifie les contraintes de  $K$ . Par récurrence sur ses parenthèses, on vérifie qu'il est analysable selon la grammaire  $\hat{\mathcal{G}}$ . En effet :

– tout mot de  $K_P$  contenant une des quatre lettres de la séquence  $\{ {}_A^a (a)_a \}_A^a$  contient forcément cette séquence qui dans  $\hat{\mathcal{G}}$  est produite par une règle  $A \rightarrow a \in \Delta$ , par les règles R1 et R2 ;

– tout mot contenant une des six lettres de la séquence  $\{ {}_A^{BC} ({}_A^{BC})_A^{BC} [{}_A^{BC}]_A^{BC} \}_A^{BC}$  contient toutes les autres et dans cet ordre du fait de son appartenance à  $D_P$  et à  $K$  par la règle R3, et si, par hypothèse de récurrence ses facteurs  $w_B$  et  $w_C$  placés entre les parenthèses homologues  $({}_A^{BC})_A^{BC}$  et  $[{}_A^{BC}]_A^{BC}$  respectivement sont engendrés (nécessairement à partir des variables  $B$  et  $C$ ) par la grammaire  $\hat{\mathcal{G}}$ , alors le mot  $w = \{ {}_A^{BC} ({}_A^{BC} w_B)_A^{BC} [{}_A^{BC} w_C]_A^{BC} \}_A^{BC}$  l'est également à partir de  $A$  en raison de la construction de la grammaire ;

– enfin la règle R4 impose que le mot soit issu de l'axiome de la grammaire.

En conséquence,

$$\hat{L} = D_P \cap K$$

et par application du morphisme effaceur  $h$ , on en déduit le théorème.

**Théorème 11.6.2** *Étant donné un langage algébrique  $L$ , il existe un langage de Dyck  $D$ , un langage rationnel  $K$  et un morphisme  $h$  tels que :*

$$L = h(D \cap K).$$

## Exercices

1. Décrire des automates à pile pour les langages suivants :

a)  $\{w c w^R \mid w \in \{a, b\}^*\}$  ;

b)  $\{w w^R \mid w \in \{a, b\}^*\}$  ;

c)  $\{w \mid |w|_a = |w|_b\}$  ;

d)  $\{a^n b^m \mid n \leq m \leq 2n\}$  ;

e) le langage engendré par la grammaire

$$\mathcal{G} = \langle \{S, A\}, \{a, b\}, S, \{S = aAA, A = bS + aS + a\} \rangle.$$

2. Donner une grammaire pour le langage reconnu par l'automate s'arrêtant sur pile vide et défini par

$$M = \langle \{q_0, q_1\}, \{0, 1\}, \{Z_0, X\}, q_0, Z_0, \delta \rangle,$$

où  $\delta$  consiste en les actions :

$$\begin{array}{ll} \delta(q_0, 1, Z_0) = \{(q_0, XZ_0)\} & \delta(q_0, \varepsilon, Z_0) = \{(q_0, \varepsilon)\} \\ \delta(q_0, 1, X) = \{(q_0, XX)\} & \delta(q_1, 1, X) = \{(q_1, \varepsilon)\} \\ \delta(q_0, 0, X) = \{(q_1, X)\} & \delta(q_1, 0, Z_0) = \{(q_0, Z_0)\} \end{array}$$

3. Montrer que l'on peut définir l'acceptation d'un automate à pile par un ensemble d'états finaux sans modifier la puissance de calcul du modèle.

4. Quels sont les automates déterministes dans l'exercice 1 ci-dessus ?

5. Montrer que, pour tout automate à pile, on peut lui trouver un équivalent à un seul état.

6. Appliquer l'algorithme d'élimination des récurrences gauches à la grammaire formée des règles

$$A_1 = A_2 A_2 + 0, A_2 = A_1 A_2 + 1.$$

7. Donner une méthode pour calculer le nombre de dérivations possibles pour les mots de longueur  $n$  dans une grammaire en forme de Chomsky. Peut-on en tirer une information en ce qui concerne l'ambiguïté de la grammaire ? Appliquer cette méthode au langage  $\{a^i b^j c^k \mid (i = j) \vee (j = k)\}$ .

8. Montrer que le langage  $K_n = \{a^{2^k} \mid k \leq n\}$  est rationnel et construire une expression régulière ou un automate. Exprimer cet ensemble au moyen d'une grammaire algébrique. Quel est le rapport des tailles de description ?

Montrer que le langage  $K_* = \{a^{2^k} \mid k \geq 0\}$  n'est pas algébrique.

9. Que peut-on dire des langages

$$L_{15} = \{ab^{i_1} ab^{i_2} a \dots ab^{i_{15}} \mid \exists l \ i_l = l\}$$

et

$$L_* = \{ab^{i_1} ab^{i_2} a \dots ab^{i_k} \mid k \geq 1 \ \& \ \exists l \leq k \ i_l = l\}?$$



## Chapitre 12

# Analyse syntaxique et autres langages

La théorie des langages de parenthèses est d'une grande importance par ses applications dans l'analyse des langues naturelles et dans celle des langages de programmation. Dans le cas des langues naturelles, elle a été à l'origine d'un changement radical dans la méthodologie, mais nous ne développerons pas ce point. Pour ce qui est de l'analyse syntaxique des programmes, premier point obligé de la compilation et de l'interprétation, cette théorie, construite en même temps que l'on concevait les premiers langages de programmation comme Fortran, Lisp et Cobol, a permis de se sortir de l'approche pragmatique du groupe Fortran pour autoriser le développement d'outils systématiques et puissants.

Nous aborderons ensuite quelques prolongements sur d'autres classes de langages aux caractéristiques bien différentes et dont l'utilité n'est pas moindre.

### 12.1 Éléments d'analyse syntaxique

L'analyse syntaxique vise à fournir pour une phrase, un texte, un programme une traduction abstraite qui permette de lui donner une forme dénuée de toute ambiguïté en vue de la traduire dans un autre langage ou un univers sémantique bien défini.

On a vu au chapitre précédent que la reconnaissance des mots d'un langage algébrique se fait en temps  $O(n^3)$  par une méthode de programmation dynamique. Les améliorations basées sur la multiplication matricielle rapide ne sont pas compétitives dans la pratique en raison de la trop grande complexité de mise en place de ces algorithmes récursifs.

On recherche des sous-classes de grammaires qui permettent de réaliser cette opération en temps linéaire et qui produisent également un arbre d'analyse syntaxique comme résultat : c'est cet arbre qui définira une première forme de sémantique pour le texte analysé, qu'il soit un programme ou un texte littéraire.

### 12.1.1 Analyse descendante

Considérons la grammaire  $\mathcal{G}_1$  d'axiome  $S$  dont les règles sont les suivantes :

$$\begin{aligned} S &= aB + bCc + c, \\ B &= bB + aCC, \\ C &= aSS + c. \end{aligned}$$

Le mot  $w = aaacc$  appartient au langage engendré et ne possède qu'un seul arbre de dérivation correspondant à la dérivation gauche :  $S \rightarrow aB \rightarrow aaCC \rightarrow aaaSSC \rightarrow aaacSC \rightarrow aaaccC \rightarrow aaacc$ .

En effet, on constate que puisque  $w$  commence par  $a$ , seule la règle  $S \rightarrow aB$  a pu être utilisée, puis que seule  $B \rightarrow aCC$  a pu fournir le second  $a$  de  $w$ , et ainsi de suite car la grammaire  $\mathcal{G}_1$  a la propriété que chaque paire  $(\alpha, X)$ , pour  $\alpha \in A$  et  $X \in V$ , détermine de façon unique la production à appliquer et donc conduit à une analyse *déterministe et linéaire* des mots du langage. En fait c'est aussi l'automate à pile trivialement associé à la grammaire qui est déterministe et que l'algorithme suggéré ci-dessus ne fait que simuler. Cette situation peut se généraliser comme le montrent les exemples suivants.

Soit la grammaire  $\mathcal{G}_2$  d'axiome  $S$  dont les règles sont les suivantes :

$$\begin{aligned} S &= aBBB + bC, \\ B &= bB + aC, \\ C &= SS + c. \end{aligned}$$

Cette grammaire n'a plus la bonne propriété de  $\mathcal{G}_1$ , cependant on constate que si l'on développe le premier  $S$  de la production  $C = SS$  en  $C = aBBBS + bCS$ , on se ramène au cas précédent et donc on sait analyser les mots du langage en temps linéaire.

La grammaire  $\mathcal{G}_3$  d'axiome  $S$  et règles

$$\begin{aligned} S &= aBB + bC, \\ B &= CB + aC, \\ C &= abS + c \end{aligned}$$

nous met dans une situation encore différente : la simple réécriture de  $B = CB$  en  $B = abSB + cB$  ne permet pas de lever l'indétermination en raison de l'existence de la production  $B = aC$ . Toutefois si l'on développe encore ce  $C$ , on arrive à remplacer les deux productions initiales de  $B$  par  $B = abSB + cB + aabS + ac$ , et alors on peut savoir quelle production appliquer quand on doit dériver un  $B$  produisant un  $a$  en regardant la lettre suivante ! Cette anticipation (*look-ahead* en anglais) conduit à un algorithme d'analyse du même style et linéaire.

En toute généralité cette méthode s'appuie sur une propriété des dérivations gauches que nous formalisons maintenant.

**Définition 12.1.1** Soit  $FG_k$  la fonction définie sur  $A^*$  pour  $k \in \mathbb{N}$  par

$$FG_k(f) = \begin{cases} f & \text{si } |f| \leq k, \\ g & \text{tel que } f = gh \text{ \& } |g| = k \text{ sinon.} \end{cases}$$

Une grammaire est  $LL(k)$  si et seulement si pour toute variable  $v$  admettant deux dérivations telles que  $v \xrightarrow{*} f_1w_1$  et  $v \xrightarrow{*} f_2w_2$ , avec  $f_1, f_2 \in T^k$  et  $w_1, w_2 \in (T \cup V)^*$  et de surcroît

$$FG_k(f_1) \neq FG_k(f_2),$$

alors nécessairement les deux dérivations divergent dès la première réécriture :

$$\begin{array}{l} v \rightarrow \hat{v}_1 \xrightarrow{*} f_1w_1 \\ v \rightarrow \hat{v}_2 \xrightarrow{*} f_2w_2, \text{ avec } \hat{v}_1 \neq \hat{v}_2. \end{array}$$

Cette définition est clairement la formalisation des remarques informelles faites en début de paragraphe. On en déduit immédiatement que la méthode de reconnaissance esquissée ci-dessus se généralise au cas des grammaires  $LL(k)$ . L'automate à pile va prendre de l'avance et stocker dans son mécanisme d'états finis les  $k$  derniers symboles lus, ce qui lui permet de prendre la bonne décision en matière de dérivation... Il faut bien sûr préparer ces décisions, ce qui se fait en construisant des tables pour l'automate à pile, qui du même coup est déterministe. On renvoie à l'une des éditions du Dragon [1] pour les astuces d'implantation.

**Proposition 12.1.1** *Tout langage engendré par une grammaire  $LL(k)$  est reconnaissable par un automate à pile déterministe en temps linéaire.*

Pour terminer ce paragraphe, notons que l'on peut tester la propriété d'être une grammaire  $LL(k)$  en temps  $O(|\mathcal{G}|^k)$ , ce qui est en gros le coût de construction de l'automate (ou des tables du programme). Enfin le fait pour un langage d'être engendré par une grammaire  $LL(k)$  entraîne sa non-ambiguïté.

### 12.1.2 Analyse ascendante

Tous les langages ne sont pas forcément susceptibles de posséder une grammaire  $LL(k)$  : ainsi les langages produisant des systèmes de parenthèses comme celui donné par la grammaire  $\mathcal{G}_4$  ci-dessous d'axiome  $S$

$$\begin{array}{l} S = [S] + R, \\ R = E \equiv E, \\ E = [E \otimes E] + a + b, \end{array}$$

pour lequel il est impossible de décider, quel que soit le  $k$ , si les parenthèses initiales relèvent de  $S$  ou de  $E$ ...

La levée des ambiguïtés ne peut se faire que si l'on reconstruit différemment l'arbre syntaxique sous-jacent, en l'occurrence à partir de la droite (et non plus de gauche) ce qui est possible si la grammaire a certaines caractéristiques.

Nous allons considérer dans ce paragraphe des dérivations *droites* uniquement. On fera aussi la supposition que toutes les variables sont utiles, en ce sens qu'elles contribuent à former des mots du langage : on sait qu'il est facile de purifier la grammaire pour se ramener à ce cas.

**Définition 12.1.2** *Une dérivation est appelée droite si et seulement si à chaque étape la réécriture porte sur le non-terminal le plus à droite.*

Dans ce paragraphe nous utiliserons les conventions typographiques suivantes : tout mot  $w \in (V \cup T)^*$  est scandé en  $w = \alpha A f$ , où  $f \in T^*$  est son plus long facteur droit terminal,  $A \in V$  est la variable la plus à droite sur laquelle portera la réécriture et  $\alpha \in (V \cup T)^*$  est le contexte gauche.

La réécriture (droite) de  $A$  selon la règle  $A = \beta$ , pour  $\beta \in (V \cup T)^*$  donnera alors :

$$\alpha A f = w \rightarrow_d w' = \alpha \beta f.$$

Toute la question est de déterminer dans quelles conditions la connaissance d'information sur  $\beta$  permet d'en déduire que c'est précisément la règle  $A = \beta$  qui avait été appliquée. Pour ce faire nous introduisons le concept fondamental de toute la reconnaissance montante (ou droite) : le contexte gauche d'une variable.

**Définition 12.1.3** *Soit  $\mathcal{G}$  une grammaire. Pour toute variable  $A \in V$  on appelle contexte gauche de la variable l'ensemble des facteurs gauches qu'elle définit en étant en position droite dans une dérivation droite :*

$$CG(A) = \{\alpha \mid S \xrightarrow{*}_d \alpha A f, \alpha \in (V \cup T)^*, f \in T^*\}.$$

Par exemple pour la grammaire de règles  $S = aAb$  et  $A = aAcAb + c$  et d'axiome  $S$ , on a :

- $CG(S) = \{\varepsilon\}$  ;
- $CG(A) = a(aAc)^*(a + \varepsilon)$ .

Le lecteur est invité à construire les contextes gauches des variables de la grammaire  $\mathcal{G}_4$  ci-dessus.

On a la propriété remarquable qui conditionne toutes les constructions à venir.

**Proposition 12.1.2** *Pour toute grammaire  $\mathcal{G}$  et toute variable  $A$  utile, le contexte gauche de  $A$  est un langage rationnel :  $CG(A) \in \mathcal{Rat}$ .*

Preuve : Les règles de  $\mathcal{G}$  où  $A$  apparaît s'écrivent de façon générique sous la forme  $B_i = \gamma_i A \delta_i$  avec  $\alpha_i, \delta_i \in (V \cup T)^*$  : on doit noter que cette scansion est différente de celle de la convention du début du paragraphe et qu'il y a autant de telles décompositions, pour une règle donnée, qu'il y a d'occurrences de la lettre  $A$  dans le membre droit de la règle. Les langages de contextes gauches vérifient alors le système d'équations

$$CG(A) = CG(B_1)\gamma_1 + CG(B_2)\gamma_2 + \dots + \varepsilon(A),$$

où  $\varepsilon(A) = \text{si } (A = S) \text{ alors } \{\varepsilon\} \text{ sinon } \emptyset$ .

Un tel système d'équations n'est autre qu'une grammaire linéaire droite, ou encore le système d'équations associé aux langages d'étiquettes des états d'un automate fini. Ces langages sont donc tous rationnels et nous avons construit implicitement un automate qui les reconnaît.

**Définition 12.1.4** Soit  $A = w$  une règle de la grammaire  $\mathcal{G}$  ; on appelle *contexte gauche de la règle  $A = w$*  le langage

$$CG(A = w) = CG(A)w.$$

Ces langages sont donc tous rationnels et effectivement calculables à partir de la grammaire.

Nous en arrivons à la notion cruciale qui va permettre de retrouver en fonction de ce qui a été lu sur le texte et de ce que la machine a mémorisé quelle production a été employée dans le processus de réécriture droite par lequel on tente d'analyser le texte. Pour ceci on étend la notation concernant les facteurs gauches.

**Notation 12.1.1** Soit  $L$  un langage. On note  $FG(L)$  l'ensemble des facteurs gauches de  $L$  :

$$FG(L) = \{f \mid \exists g \, fg \in L\}.$$

**Définition 12.1.5** Une grammaire est dite  $LR(0)$  si et seulement si pour toute paire de productions  $A = w$  et  $A' = w'$  on a la propriété :

$$CG(A = w) \cap FG(CG(A' = w')) = \emptyset.$$

En termes intuitifs, cette propriété permet de garantir que l'on sait caractériser les productions par la seule connaissance des facteurs gauches des mots de dérivation droite dans l'arbre syntaxique, et partant que l'on pourra le reconstruire du bas vers le haut et de droite à gauche. C'est le sens de la dénomination  $LR$  qui vient de l'anglais *Left-Right*.

**Proposition 12.1.3** Il existe un algorithme pour déterminer si une grammaire est  $LR(0)$ .

*Preuve* : Les deux langages de contextes gauches sont rationnels, ainsi que le langage des facteurs gauches. Les rationnels sont clos par intersection et l'on sait déterminer si un rationnel est vide ou non.

Passons maintenant à l'algorithme de reconnaissance des mots du langage engendré par la grammaire  $LR(0)$ .

Chaque langage de contexte gauche de règle  $CG(A = w)$  est reconnu par un automate  $\mathcal{A}_{A=w}$  : nous noterons, pour chaque règle  $R_i$ , l'automate correspondant par  $\mathcal{A}_{R_i}$ . On construit alors l'automate produit, pour toutes les règles de la grammaire,

$$\mathcal{A}_{\mathcal{G}} = \mathcal{A}_{R_1} \times \mathcal{A}_{R_2} \times \dots \times \mathcal{A}_{R_p}.$$

Cette construction est identique à celle de l'automate reconnaissant l'intersection de deux langages rationnels : les états de  $\mathcal{A}_G$  sont des  $p$ -uplets formés sur les états des automates  $\mathcal{A}_{R_i}$  à ceci près que l'on se soucie peu des états finaux de  $\mathcal{A}_G$ . Une autre façon de voir cet automate produit est de dire que l'on effectue les calculs séparés de chacun des  $p$  automates en parallèle.

L'alphabet de  $\mathcal{A}_G$  est  $V \cup T$ , son ensemble d'états  $Q_{R_1} \times Q_{R_2} \times \dots \times Q_{R_p}$  et son état initial le  $p$ -uplet des états initiaux de chaque automate. La table de transitions s'obtient comme le produit des tables de chaque automate.

**Proposition 12.1.4** *À chaque étape du calcul de  $\mathcal{A}_G$  sur un mot  $w$ , au plus une des composantes de l'état courant est état final de l'automate correspondant  $\mathcal{A}_{R_i}$ .*

Preuve : La propriété de la grammaire d'être  $LR(0)$  est précisément que  $CG(R_i) \cap FG(CG(R_j)) = \emptyset$ , pour  $i \neq j$ , ce qui entraîne l'impossibilité pour deux automates d'être simultanément dans un état final. Il est parfaitement possible qu'aucun automate ne soit dans un état final – et c'est même ce qui se passe le plus souvent.

Considérons donc une dérivation droite pour un mot  $f$  de  $\mathcal{L}(\mathcal{G})$ , dans laquelle nous sélectionnons un sous-arbre particulier correspondant à la dérivation (droite) d'un non-terminal particulier :

$$S \xrightarrow{*}_d \alpha Aw \rightarrow_d \alpha \beta w \xrightarrow{*}_d f,$$

où  $\alpha, \beta \in (V \cup T)^*$ ,  $w \in T^*$  et, de plus,  $f = w'w$  avec  $\alpha\beta \xrightarrow{*}_d w' \in T^*$ .

Si l'on donne le mot  $\alpha\beta$  à lire à l'automate  $\mathcal{A}_G$ , celui doit indiquer que  $\alpha\beta \in CG(A = \beta)$ , et donc que, en remontant la dérivation pour la reconstruire,  $\alpha\beta$  devrait être réduit à  $\alpha A$ .

Supposons, pour simplifier, que ce même  $A$  ait lui-même été produit par une règle de la forme  $B = \gamma Ax$  où  $x \in T^*$  ; alors  $w$  admet nécessairement  $x$  pour facteur gauche, et l'automate  $\mathcal{A}_G$  doit reconnaître que le mot  $\alpha Ax$  est dans l'ensemble  $CG(B = \gamma Ax)$  et que donc  $\delta By \rightarrow \alpha Aw$ , avec  $\alpha = \delta\gamma$  et  $w = xy$ . On peut donc réduire le mot courant  $\alpha Aw$  à  $\delta By$  et repartir avec les lettres de  $y$  pour poursuivre la réduction.

Dans le cas général, le  $x$  précédent contient des variables que le mécanisme aura su réduire en partant de la variable la plus à droite, selon un processus récursif que nous explicitons maintenant et qui se présente comme un automate à pile déterministe.

L'automate à pile :

- alphabet  $X = V \cup T$  ;
- alphabet de pile  $X \times Q_{\mathcal{A}}$ , soit un ensemble de  $(p + 1)$ -uplets composés d'une lettre de  $X$  et d'un état de  $\mathcal{A}_G$ .

Ses règles de fonctionnement sont données par le programme suivant qui s'appuie sur le fait qu'à tout instant la pile vérifie la propriété que si  $\pi \in X^*$  est le mot de pile lu sur sa première composante alors le  $p$ -uplet de sommet de pile  $q_\pi$  est celui des états des  $p$  automates  $\mathcal{A}_{R_i}$  de contextes gauches des règles après lecture de ce mot  $\pi$ . De plus ceci est vrai à tous les niveaux de la pile, la preuve se faisant par récurrence sur le fonctionnement.

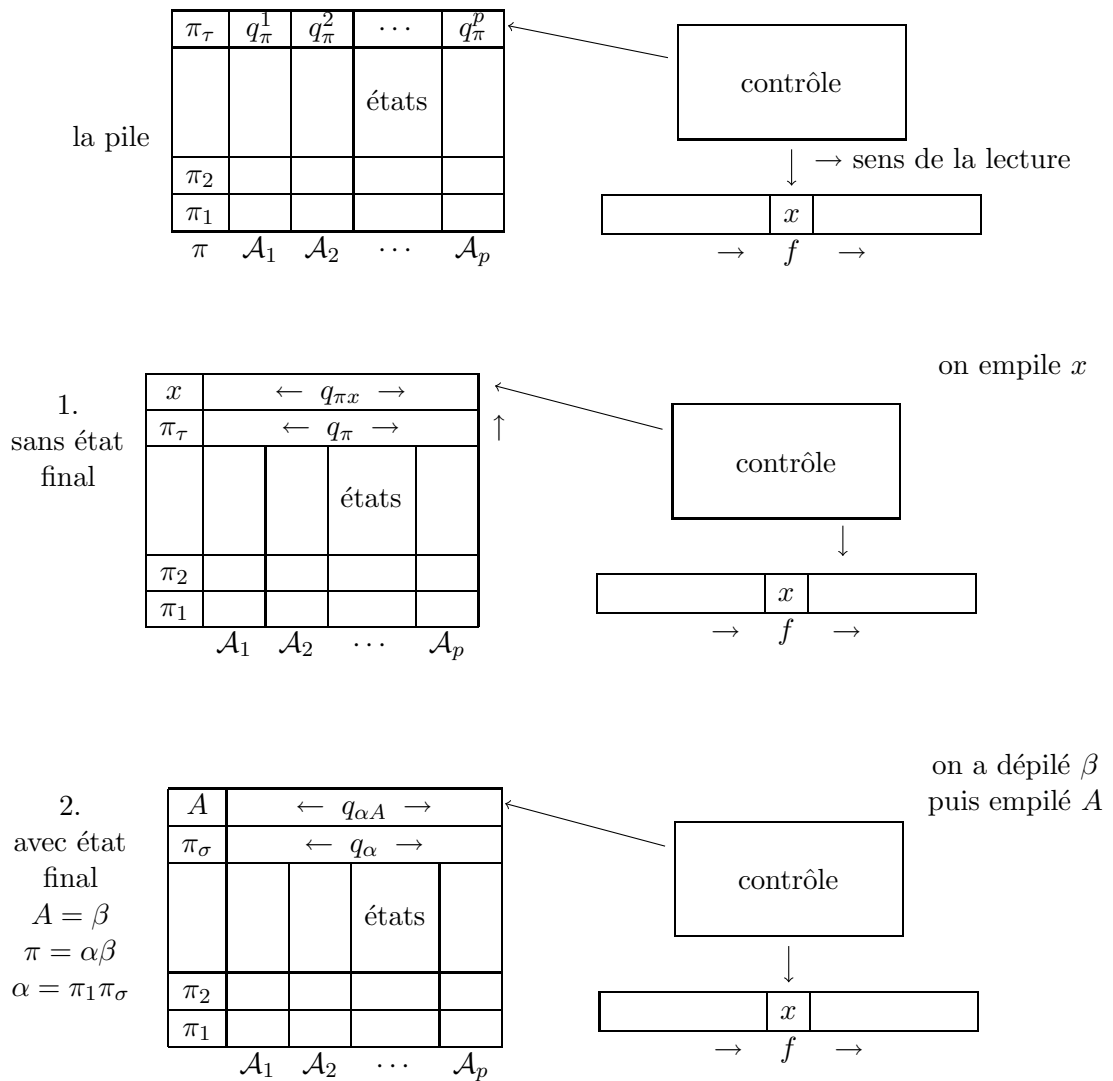


FIG. 12.1 – Une représentation de la pile et du fonctionnement de l'automate de reconnaissance  $LR(0)$

Le fonctionnement est régi par les deux cas :

- si  $q_\pi$  n'a pas de composante qui soit état final d'un des  $\mathcal{A}_{R_i}$ , alors lire la lettre suivante  $x$  de la donnée et empiler  $(x, q_{\pi x})$ , où  $q_{\pi x}$  est le  $p$ -uplet d'états obtenus en faisant lire à chaque automate la lettre  $x$  ;
- si  $q_\pi$  a une composante qui soit état final de  $\mathcal{A}_{R_i}$  (par exemple et on sait que ce  $i$  est unique), correspondant à la règle  $A = \beta$ , alors nécessairement  $\beta$  est suffixe de  $\pi = \alpha\beta$  et l'automate dépile toute la partie correspondant à  $\beta$  ne laissant que le fond de pile avec  $\alpha$  en première colonne ; puis il empile  $(A, q_{\pi A})$ .

Ainsi, à chaque instant où une composante du sommet de pile est un état final, ce mécanisme permet de garantir que le mot de pile est facteur gauche d'une dérivation droite légale et que toute dérivation droite légale conduira à avoir dans la pile le seul niveau  $(S, q_S)$  nécessaire et suffisant pour garantir l'appartenance du mot au langage. Toute autre situation conduit au blocage de l'automate et par conséquent au rejet de la donnée.

Ces grammaires sont une des catégories les plus utilisées en compilation ; on leur donne souvent un peu plus de souplesse en autorisant un certain *look-ahead* ce qui donne les grammaires *LALR* dont les tables sont beaucoup plus petites et dont le fleuron est le fameux YACC d'Unix (*Yet Another Compiler Compiler*) qui permet de fabriquer des compilateurs à partir d'une grammaire *LALR(1)*.

## 12.2 Au-delà des algébriques

On a montré que le langage  $\{a^n b^n c^n \mid n \geq 0\}$  n'est pas algébrique en utilisant le lemme d'itération pour ces langages. Or il est de nombreux cas où des phénomènes de croissance multiple et contrôlée s'observent et donc où le cadre des grammaires algébriques est trop restrictif. Le premier à avoir fait ce genre de remarque est Sir D'Arcy Wentworth Thompson (1860-1948) qui fit le premier travail de mathématicien sur la croissance des formes en biologie (1917). Cette approche plutôt analytique fut reprise par Aristid Lindenmeyer qui développa une théorie des L-systèmes pour simuler la croissance des structures biologiques (1968)<sup>1</sup>.

Nous donnons ici un bref aperçu de ces systèmes de réécriture.

### 12.2.1 L-systèmes

Dans les modèles hors-contextes des chapitres précédents, toutes les réécritures se font indépendamment les unes des autres. Lindenmeyer a eu l'idée d'imposer une réécriture de *toutes* les lettres à chaque étape de dérivation : ainsi avec la règle élémentaire  $A = AA$ , on fabrique comme ensemble de mots dérivés de l'axiome  $A$ , la famille  $\{A^{2^k} \mid k \geq 0\}$  dont on vérifie assez facilement qu'elle n'est pas algébrique. Nous montrons ici à titre d'exemples trois classes couramment utilisées.

---

<sup>1</sup>voir par exemple la page <http://www.xs4all.nl/~ljlapre/index.html>.

**Les 0L-systèmes**

Commençons par le plus simple des  $L$ -systèmes.

**Définition 12.2.1** *Un 0L-système est défini par la donnée d'une grammaire  $\mathcal{G} = \langle \Sigma, w, \mathcal{R} \rangle$ , où  $\Sigma$  est un alphabet fini,  $w \in \Sigma^+$  est l'axiome et  $\mathcal{R}$  est l'ensemble des règles — sous-ensemble fini de  $\Sigma \times \Sigma^*$ .*

À notre habitude chaque règle sera écrite sous la forme  $A = f$ . Notons que nous n'avons pas de distinction a priori entre lettres terminales et non-terminales. Un tel système peut être non-déterministe en ce sens qu'une même lettre peut se récrire de plusieurs manières différentes.

Un mot  $u \in \Sigma^*$  se dérive en  $v \in \Sigma^*$  — ce qu'on notera à l'usuel  $u \rightarrow v$  — si et seulement si  $u = u_1 u_2 \dots u_n$ ,  $u_i \in \Sigma$ , et il existe des règles  $u_i = v_i$  dans l'ensemble  $\mathcal{R}$  telles que  $v = v_1 v_2 \dots v_n$ .

Le langage engendré par un tel système est l'ensemble des mots dérivables en un nombre fini de coups de l'axiome :

$$\mathcal{L}(\mathcal{G}) = \{f \mid w \xrightarrow{*} f\}.$$

Un bon exemple de 0L-système — 0 signifie “sans contexte” et  $L$  est évidemment un rappel de Lindenmeyer — est donné par :

$$\mathcal{F} = \langle \{\diamond, \circ\}, \circ, \{\circ = \diamond, \diamond = \circ\circ\} \rangle .$$

Les premiers mots de  $\mathcal{L}(\mathcal{F})$  sont assez facilement vus pour être dans l'ordre :

$$\begin{aligned} f_0 & : \circ \\ f_1 & : \diamond \\ f_2 & : \circ \diamond \\ f_3 & : \diamond \circ \diamond \\ f_4 & : \circ \diamond \circ \diamond \\ f_5 & : \diamond \circ \diamond \circ \diamond \circ \diamond \\ & \text{etc.} \end{aligned}$$

Le lecteur montrera sans peine que  $f_{n+2} = f_n f_{n+1}$  et que les longueurs de ces mots sont les nombres de Fibonacci.

Les 0L-systèmes sont trop contraints par les mécanismes de croissance en parallèle pour jouir des propriétés de fermeture prouvées pour les langages rationnels ou algébriques. On peut trouver des contre-exemples pour prouver que la famille des langages engendrés par les 0L-systèmes *n'est close par aucune* des opérations d'union, concaténation, intersection avec les rationnels, homomorphisme, etc.

Cette famille ne contient même pas les ensembles finis, comme par exemple  $\{a, aaa\}$  !

### Les $E0L$ -systèmes

Pour étendre convenablement la famille des langages — d'où le  $E$  — il faut réintroduire un peu de souplesse dans la synchronisation et pour ce faire permettre des récritures non productives ultimement de façon à pouvoir mieux sélectionner. On introduit donc en plus des lettres de l'alphabet, des variables supplémentaires qui seront destinées à disparaître dans les mots sélectionnés : en d'autres termes on coupe l'ensemble produit par un  $0L$ -système par un langage rationnel du type  $\Sigma^*$ .

**Définition 12.2.2** *Un  $E0L$ -système est défini par la donnée d'une grammaire  $\mathcal{G} = \langle \Sigma \cup V, w, \mathcal{R} \rangle$ , où  $\Sigma$  et  $V$  sont des alphabets finis,  $w \in (V \cup \Sigma)^+$  est l'axiome et  $\mathcal{R}$  est l'ensemble des règles — sous-ensemble fini de  $(V \cup \Sigma) \times (V \cup \Sigma)^*$ . Les règles de production des mots dérivés de l'axiome  $w$  sont celles du paragraphe précédent, mais le langage engendré par ce système est restreint à  $\Sigma^*$ .*

Avec  $V = \{S, A, B, C, \bar{A}, \bar{B}, \bar{C}, N\}$  comme variables,  $\Sigma = \{a, b, c\}$  comme alphabet final,  $S$  comme axiome et

$$\begin{aligned} \mathcal{R} = \{ & S = ABC, A = A\bar{A} + a, B = B\bar{B} + b, C = C\bar{C} + c, \\ & \bar{A} = \bar{A} + a, \bar{B} = \bar{B} + b, \bar{C} = \bar{C} + c \\ & a = N, b = N, c = N, N = N\}, \end{aligned}$$

on construit comme seuls mots sur l'alphabet terminal  $\{a, b, c\}$  les mots de la forme

$$\{a^n b^n c^n \mid n \geq 1\}.$$

En effet, si les  $A, B, C$  et les  $\bar{A}, \bar{B}, \bar{C}$  ne se récrivent pas de manière homologue et synchrone, ils introduiront en passant dans l'état  $a, b, c$  respectivement des  $N$  ineffaçables...

On notera que ce langage n'est pas algébrique.

Contrairement à ce qui se passe pour les  $0L$ -systèmes, la classe des langages définis par des  $E0L$ -systèmes a des propriétés de clôture semblables à celles des langages algébriques qu'elle contient. Ce que nous énonçons sans démonstration, car elle est en tous points semblable à celle faite pour les langages algébriques.

**Proposition 12.2.1** *La classe des langages définis par des  $E0L$ -systèmes contient la famille des langages algébriques et est close par union, produit, quasi-inverse et intersection avec un rationnel.*

Le lecteur est invité à montrer que tout langage rationnel peut être engendré par un  $E0L$ -système.

### Les $ET0L$ -systèmes

La dernière généralisation présentée dans ce court exposé consiste à ranger les règles en plusieurs sous-ensembles,  $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \dots \cup \mathcal{R}_p$ , appelés *tables* (d'où le  $T$  de l'acronyme) et à utiliser dans chaque récriture parallèle les règles de la même table :

la table varie évidemment au cours du temps. Le cas des *EOL*-système est un cas particulier à une seule table.

Considérons le système suivant dont l'alphabet terminal est  $\Sigma = \{A, N, 1, (, )\}$ , l'alphabet non-terminal est  $V = \{S, R, T, F\}$ , l'axiome est  $w = S$  et les tables sont respectivement — en notant  $\alpha$  un symbole a priori quelconque de l'un des deux alphabets :

$$\begin{aligned} \mathcal{R}_1 &= \begin{cases} S = (xAyAz)S + (xAyAz), & \text{pour } x, y, z \in \{T, NT, F, NF\} \\ & \text{et au moins un des } x, y, z \in \{T, NF\}, \\ \alpha = \alpha, & \text{pour } \alpha \neq S; \end{cases} \\ \mathcal{R}_2 &= \begin{cases} S = R, F = 1F, T = 1 + 1T, \\ \alpha = \alpha, \text{ pour } \alpha \neq S, T, F; \end{cases} \\ \mathcal{R}_3 &= \begin{cases} S = R, T = 1T, F = 1 + 1F, \\ \alpha = \alpha, \text{ pour } \alpha \neq S, T, F. \end{cases} \end{aligned}$$

Ce système produit par exemple le mot

$$(11AN1AN111)(AN11AN1111A111)(1A11AN1111),$$

que l'on va interpréter comme un codage de la formule booléenne

$$(x_2 + \bar{x}_1 + \bar{x}_3)(\bar{x}_2 + \bar{x}_4 + x_3)(x_1 + x_2 + \bar{x}_4)$$

dont on vérifie qu'elle est satisfaisable.

Avec cette interprétation des choses — ou plus exactement des mots engendrés par le système — on constate et prouve que le système engendre précisément les formules booléennes en FNC qui sont satisfaisables. On a soupçonné au premier chapitre que cet ensemble est difficile à caractériser algorithmiquement et nous étayerons cette idée dans le dernier chapitre.

Cette famille des langages *ETOL* est donc plus large que la précédente et vérifie les mêmes propriétés de clôture.

### 12.2.2 Autres classes

De fait il est possible de définir bien d'autres classes. Il existe des hiérarchies infinies de classes de langages reconnaissables par diverses adaptations ou extensions de grammaires, systèmes de réécriture, automates ou autres machines. On pourrait parler aussi de machines non-déterministes et on pourrait aussi évoquer les machines probabilistes.



## Chapitre 13

# Analyses lexicale et syntaxique par l'exemple

Une bonne référence pour ce qui suit reste le livre surnommé le dragon [1] (également traduit en français). On pourra également consulter [8]. On procédera plus par exemples qu'en théorisant le sujet.

### 13.1 Motivations

Un des problèmes fondamentaux de l'informatique est l'écriture de compilateurs. Être capable de récupérer de l'information dans un gros fichier de données est une autre problématique fréquente (penser à des résultats d'expériences). On a souvent à écrire de petits langages adaptés à des problèmes spécifiques, et transformer le programme en objets plus utilisables est primordial.

Le schéma type d'un compilateur classique se trouve à la figure 13.1.

Le programme source subit de multiples transformations avant que du code exécutable soit produit. Notez également le rôle non négligeable des interactions avec le gestionnaire d'erreurs, dont la présence est indispensable pour l'utilisateur.

Nous allons surtout insister sur le début de la chaîne, laissant la suite pour le cours de Compilation de Majeure 2.

L'analyse lexicale sert à isoler les lexèmes dans une chaîne de caractères.

**Définition 13.1.1** *Un lexème (en anglais token) est une unité élémentaire importante d'une chaîne pour une phase ultérieure de calcul.*

**Ex.** De

```
public static void main(String[] args){
    int n = 2;
}
```

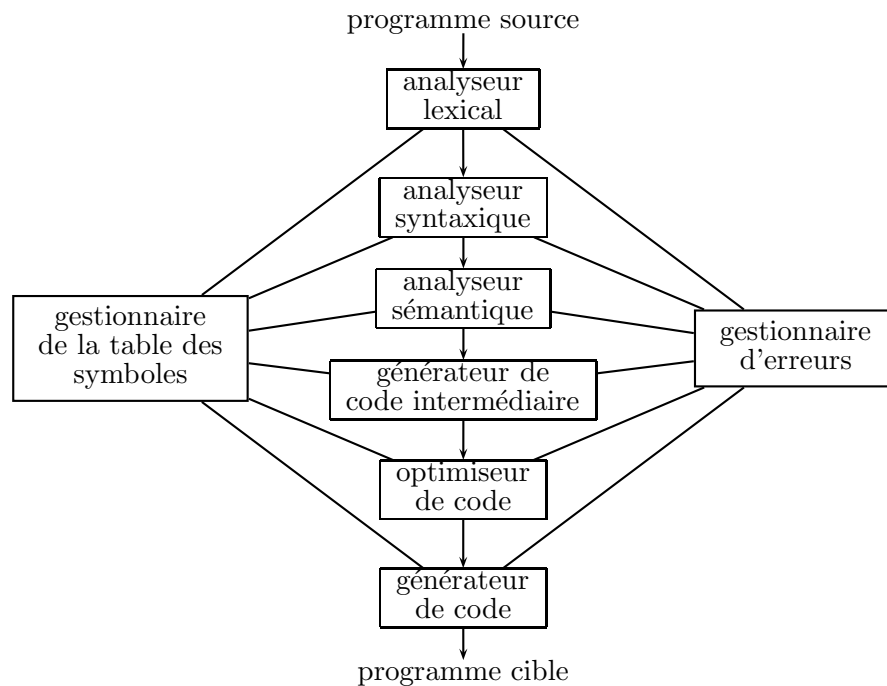


FIG. 13.1 – Schéma type d'un compilateur.

on récupère les mots clefs **public**, **static**, etc., des variables `args`, `n` et une constante `2`.

Plus généralement, on récupère un *identificateur*, une constante *numérique*, un *opérateur*, une *chaîne* de caractères, une constante *caractère*, etc. D'habitude, les espaces, tabulations, retours à la ligne, commentaires ne sont pas des lexèmes.

Donnons des exemples, avec comme source un morceau de programme Java, tout d'abord

```
" expression = 3 * x + 2 ; "
```

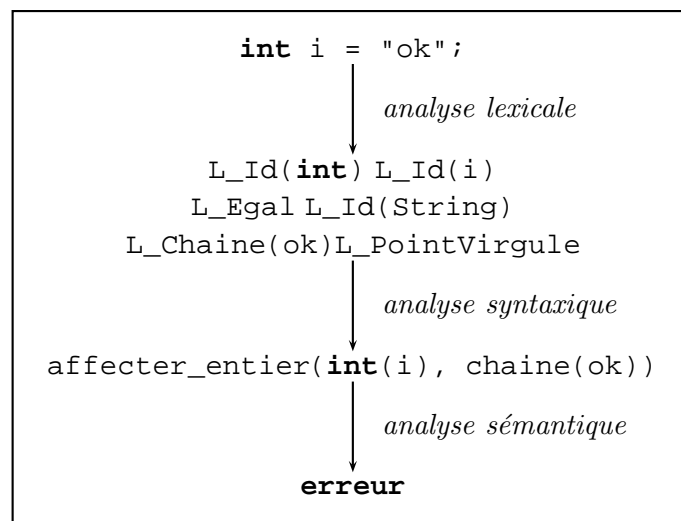
On obtient comme résultat la suite de lexèmes suivante

```
L_Id(expression) L_Egal L_Nombre(3) L_Mul L_Id(x)
L_Plus L_Nombre(2) L_PointVirgule L_EOF
```

On obtiendra le même résultat si la chaîne d'entrée est

```
" expression =
    3 * x
    + 2 ; "
```

Donnons un cas qui provoquera une erreur.



Intéressons-nous maintenant à un programme Java un peu plus vrai :

```

" class PremierProg{
  public static void main(String[] args){
    System.out.println("\Bonjour!\");
  }
} "
  
```

On obtiendra alors :

```

L_Id(class) L_Id(PremierProg) L_AccG L_Id(public)
L_Id(static) L_Id(void) L_Id(main)
L_ParG L_Id(String) L_CroG L_CroD L_Id(args) L_ParD
L_AccG L_Id(System) L_Point L_Id(out) L_Point
L_Id(println) L_ParG L_Chaine(Bonjour!) L_ParD
L_PointVirgule L_AccD L_AccD L_EOF
  
```

## 13.2 Construction de l'analyseur lexical

Nous allons donner les principes de construction d'un analyseur lexical pour une version simplifiée de Java. On se donne trois lexèmes définis par des expressions régulières :

Identificateur = Lettre (Lettre | Chiffre)\*

Entier = Chiffre Chiffre\*

Operateur = + | - | \* | /

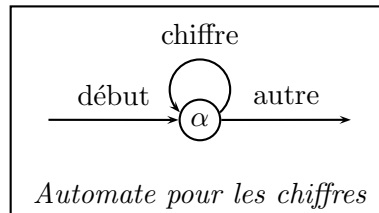
Lettre = a | b... | z | A | B... | Z

Chiffre = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

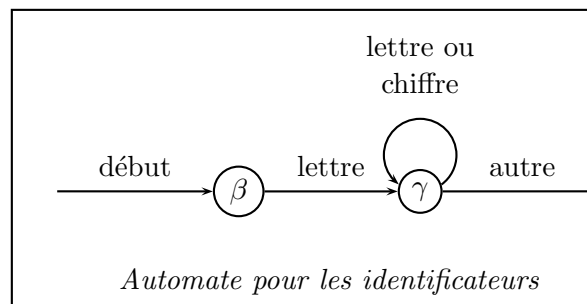
Blancs = ( ' ' | '\t' | '\r' | '\n' )\*

### 13.2.1 Utilisation des diagrammes de transition

Le principe de l'analyse est le suivant. On connecte entre eux des automates élémentaires qui savent reconnaître chaque lexème. Par exemple, un chiffre est reconnu par l'automate :

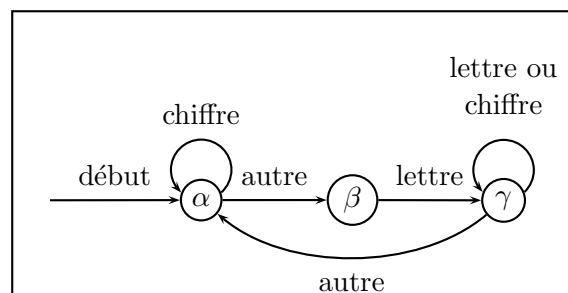


De même qu'un identificateur aura pour automate :



À chaque tour de boucle, on accumule les caractères lus. Si on sort avec une chaîne non nulle, on a trouvé un lexème et son type.

Il ne reste alors plus qu'à assembler les diagrammes de transition pour pouvoir reconnaître plus de lexèmes :



On a ainsi construit l'embryon du programme qui reconnaît un entier ou un identificateur. Une fois le schéma complété, il ne reste plus qu'à en déduire le programme correspondant.

### 13.2.2 Le programme en Java

On va représenter chacun des lexèmes par un objet contenant la nature du lexème (une constante entière), une valeur éventuelle (si on a affaire à une constante numérique), un nom (si nécessaire) :

```

class Lexeme{
    final static int L_Nombre = 0, L_Id = 1,
                  L_Plus = '+', L_Moins = '-',
                  L_Mul = '*', L_Div = '/',
                  L_ParG = '(', L_ParD = ')',
                  L_CroG = '[', L_CroD = ']',
                  L_EOF = -1;

    int nature;
    int valeur;
    String nom;

    Lexeme(int t, int i) { nature = t; valeur = i; }
    Lexeme(int t, String s) { nature = t; nom = s; }
    Lexeme(int t) { nature = t; }

    // aller chercher le lexème suivant
    static Lexeme suivant() { ... }

```

La principale difficulté d'un analyseur lexical tient à la gestion du caractère courant sur le tampon de lecture. Ici, on suppose que le caractère courant est une variable de classe. La méthode `avancer` est une des primitives les plus simples qui met à jour ce caractère courant. Dans la foulée, on écrit aussi une méthode qui "saute" les caractères inutiles (disons les espaces) :

```

static int c;    // caractère courant

static void avancer(){
    try{
        c = in.read();
    } catch (IOException e) { }
}

static void sauterLesBlancs(){
    while(Character.isWhitespace(c))
        avancer();
}

```

On peut alors écrire la méthode suivante :

```

static Lexeme suivant(){
    sauterLesBlancs();
    if(Character.isLetter(c))
        return new Lexeme(L_Id, identificateur());
    else if(Character.isDigit(c))
        return new Lexeme(L_Nombre, nombre());
}

```

```

else switch(c){
    case '+': case '-': case '*': case '/':
    case '(': case ')':
        char c1 = (char)c;
        avancer();
        return new Lexeme(c1);
    default:
        throw new Error ("Caractère illégal");
}
}

```

Le programme précédent n'est pas complet, car il ignore les problèmes liés à la fin de fichier. On laisse cela au lecteur.

Le traitement des deux automates pour les deux expressions régulières simples sont alors :

```

static String identificateur(){
    StringBuffer r;
    while(Character.isLetterOrDigit(c)){
        r = r.append(c);
        avancer();
    }
    return r;
}
static int nombre(){
    int r = 0;
    while(Character.isDigit(c)){
        r = r * 10 + c - '0';
        avancer();
    }
    return r;
}

```

### 13.2.3 Pour aller plus loin

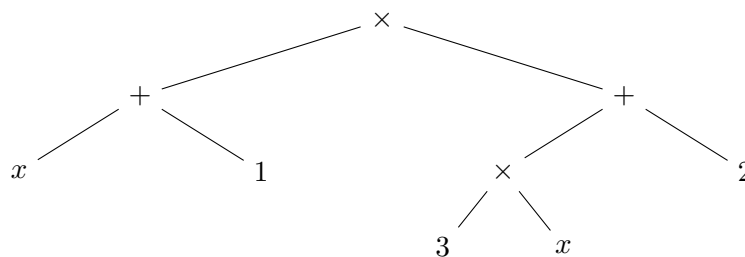
Quand il y a de nombreux mots-clé, on leur donne (comme pour les opérateurs) des numéros de lexème distincts. Chaque lexème identificateur est comparé à une *table des mots-clé*. En fonction de la taille, on peut se contenter d'une table séquentielle, ou bien d'une table de hachage, ou encore d'un arbre de recherche. Nous renvoyons au dragon pour plus de détails si besoin est.

Dans la vraie vie, il existe de nombreux outils permettant de simplifier l'analyse lexicale, comme `lex` dans le monde Unix. En Java, et après avoir écrit son premier lexeur à la main, il est recommandé d'utiliser la classe `StringTokenizer`.

## 13.3 Analyse syntaxique

### 13.3.1 Arbre de syntaxe abstraite

Le résultat de l'analyse syntaxique d'un texte (par exemple d'un programme ou d'une formule) sera souvent un *arbre de syntaxe abstraite* (ASA), c'est-à-dire une structure de données adaptée au traitement subséquent. Par exemple, à partir de l'expression arithmétique  $(x + 1) * (3 * x + 2)$ , on pourra construire l'arbre



puis se servir de cet arbre pour évaluer l'expression en  $x = 10$ , ou encore afficher la formule de façon agréable.

Donnons une implantation simple de ce concept, basé sur la grammaire simplifiée des expressions arithmétiques, et utilisant un type Terme qui est l'union des différents types possibles.

```

class Terme{

    final static int ADD = 0, SUB = 1, MUL = 2,
                  DIV = 3, MINUS = 4,
                  VAR = 5, CONST = 6;

    int nature;
    Terme a1, a2;
    String nom;
    int val;

    // des constructeurs pratiques
    Terme(int t, Terme a) {nature = t; a1 = a; }
    Terme(int t, Terme a, Terme b){
        nature = t; a1 = a; a2 = b;
    }
    Terme(String s) {nature = VAR; nom = s; }
    Terme(int v) {nature = CONST; val = v; }
}

```

Un terme correspondant à l'expression  $(x + 1) * (3 * x + 2)$  pourra être créé par :

```

Terme t = new Terme(MUL,
    new Terme(ADD, new Terme("x"), new Terme(1)),

```

```

new Terme(ADD,
    new Terme(MUL, new Terme(3), new Terme("x")),
    new Terme(2));

```

Une telle expression pourra être évaluée dans un *environnement*  $e$ , c'est-à-dire d'une table des valeurs des variables, typiquement une table de couples (nom, valeur). Le code correspondant pourra être :

```

static int evaluer(Terme t, Environnement e){
    switch(t.nature){
    case ADD: return evaluer(t.a1, e) + evaluer(t.a2, e);
    case SUB: return evaluer(t.a1, e) - evaluer(t.a2, e);
    case MUL: return evaluer(t.a1, e) * evaluer(t.a2, e);
    case DIV: return evaluer(t.a1, e) / evaluer(t.a2, e);
    case CONST: return t.val;
    case VAR: return valeurDe(t.nom, e);
    default: throw new Error ("Erreur dans evaluation");
    }
}

static int valeurDe(String s, Environnement e) {
    if(e == null) throw new Error("Variable non définie");
    if(e.nom.equals(s)) return e.val;
    else return valeurDe(s, e.suivant);
}

```

**Exercice 13.3.1** Écrire un code Java pour dériver formellement un Terme par rapport à une variable donnée.

Les arbres de syntaxe abstraite sont donc très utiles pour manipuler des programmes ou des formules. Il reste maintenant à savoir passer de l'expression proprement dite à l'ASA correspondant.

### 13.3.2 Construction des ASA

#### Le cas des structures parenthésées

C'est le cas le plus simple, comme déjà esquissé au chapitre 11. L'écriture  $(x + 1) * (3 * x + 2)$  est habituelle, mais on pourrait aussi l'écrire (formes polonaises) :

- en notation préfixée :  $* + x1 + *3x2$
- en notation postfixée :  $x1 + 3x * 2 + *$

Ces deux formules sont très faciles à analyser pour construire l'ASA. Esquisons un programme Java qui construirait un ASA à partir d'une expression préfixée :

```

static Lexeme lc; // lexème courant

```

```

static void avancer() {lc = Lexeme.suivant(); }

static Terme lireExpr(){
  if(lc.nature == Lexeme.L_Mul){
    avancer(); Terme g = lireExpr();
    avancer(); Terme d = lireExpr();
    return new Terme(MUL, g, d);
  }
  else if(lc.nature == Lexeme.L_Plus){ ... }
  else if(lc.nature == Lexeme.L_Nombre){
    return new Terme(lc.valeur);
  }
  else if(lc.nature == Lexeme.L_Id){
    return new Terme(lc.nom);
  }
  else{ ... }
}

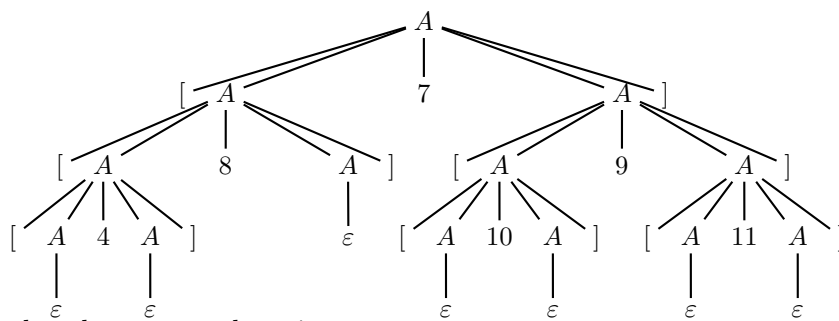
```

### Un exemple d'analyse descendante

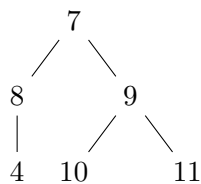
Contentons-nous de l'exemple d'une grammaire  $LL(1)$ , et considérons l'exemple de la représentation linéaire d'arbre binaire, dont la grammaire est définie par :

$$\begin{aligned} \Sigma &= \{[, ], nb\}, V = \{A\} \\ A &\rightarrow [ A nb A ] \\ A &\rightarrow \end{aligned}$$

Par exemple, le mot  $[[[4]8]7[[10]9[11]]]$  aura pour arbre de syntaxe concrète (ou *arbre syntaxique*) :



et arbre de syntaxe abstraite :

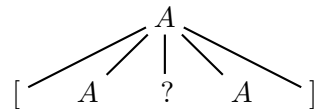


Le principe de la méthode d'analyse descendante est de partir de l'axiome  $A$  en choisissant l'une des deux règles de production en fonction du premier lexème (terminal) de la chaîne d'entrée.

On commence par lire un crochet gauche

$$\begin{array}{c} [ [ [ 4 ] 8 ] 7 [ [ 10 ] 9 [ 11 ] ] ] \\ \uparrow \end{array}$$

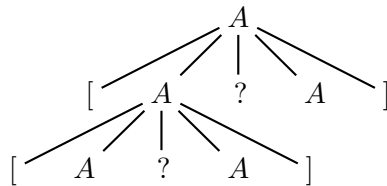
ce qui donne le début de l'arbre :



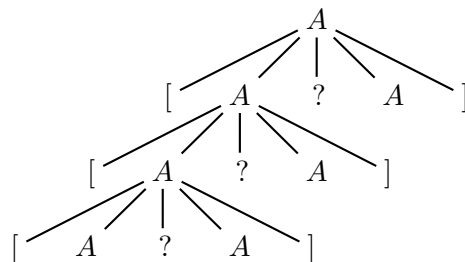
On lit alors le deuxième crochet gauche :

$$\begin{array}{c} [ [ [ 4 ] 8 ] 7 [ [ 10 ] 9 [ 11 ] ] ] \\ \uparrow \end{array}$$

ce qui déclenche :

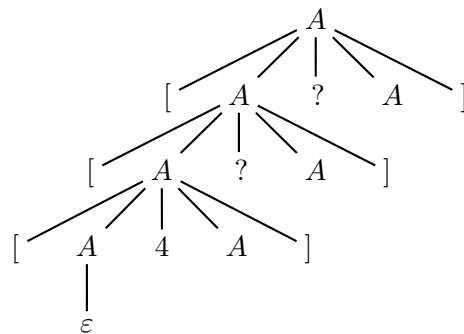


Le troisième crochet conduit à :

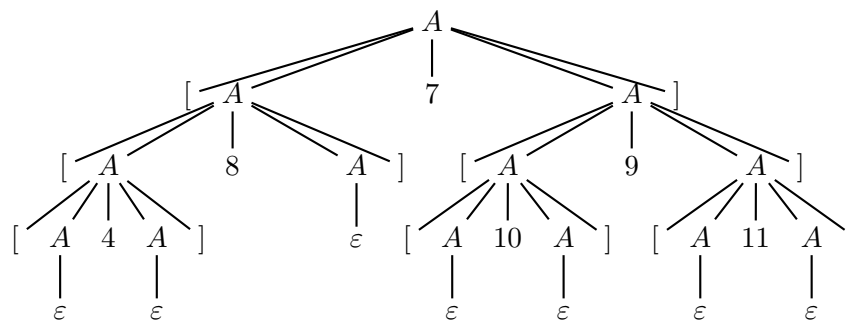
$$\begin{array}{c} [ [ [ 4 ] 8 ] 7 [ [ 10 ] 9 [ 11 ] ] ] \\ \uparrow \end{array}$$


La lecture du 4 conduit à :

$$\begin{array}{c} [ [ [ 4 ] 8 ] 7 [ [ 10 ] 9 [ 11 ] ] ] \\ \uparrow \end{array}$$



et finalement, de proche en proche, on trouve :



Un programme Java qui implanterait cette idée est le suivant :

```

static Lexeme lc; // lexème courant
static void avancer() {lc = Lexeme.suivant(); }

static Arbre lireArbre(){
  if(lc.nature == Lexeme.L_CroG){
    avancer(); Arbre a = lireArbre();
    if(lc.nature == Lexeme.L_Nombre){
      int x = lc.val;
      avancer(); Arbre b = lireArbre();
      if(lc.nature == Lexeme.L_CroD){
        avancer(); return new Arbre(x, a, b);
      } else
        throw new Error("Syntaxe: \"\"]\" manquant.");
    } else
      throw new Error("Syntaxe: nombre manquant.");
    }
  } else return null;
}

```

**Expressions générales**

Si le programme précédent est particulièrement facile, c'est que la forme préfixée se prête remarquablement bien à la traduction. Pour le faire à partir de la forme postfixée, il faut utiliser une pile qui permet de mémoriser les sous-expressions construites et de les assembler au fur et à mesure que la lecture d'opérateurs conduira à le faire. Cette construction est laissée à titre d'exercice.

Dans le cas général, il faut revenir à la description des expressions et à leur décomposition hiérarchique. Sans chercher à expliciter plus le formalisme rappelons que les formes valides d'expressions Caml, C ou Java sont usuellement définies par des équations dont nous nous inspirons pour exprimer ce qu'est une expression arithmétique valide ainsi que son interprétation induite des règles de priorité énoncées en début de chapitre. Cette description est appelée forme de Backus-Naur en hommage à deux grands concepteurs de compilateurs, pour Fortran et Algol respectivement.

## Quatrième partie

### Annexes



## Chapitre 14

# Quelques classes prédéfinies de Java

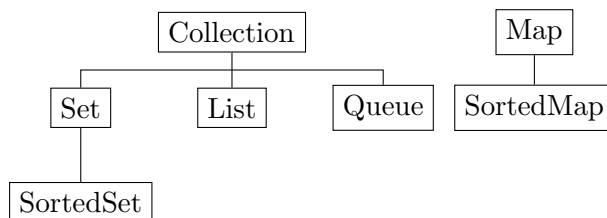
Nous présentons rapidement la classe des collections, ainsi que quelques-unes des classes que nous utiliserons au moment d'implanter les opérations et algorithmes sur les graphes.

Rappelons que le but de ces classes est de faciliter le partage et la réutilisation de code. Cela permet d'obtenir une certaine concision du code, ainsi qu'un prototypage plus rapide. Gardons cependant en tête que dans certains cas, il vaut mieux, après cela, récrire le code avec une structure de données ultra-spécifique si besoin est.

### 14.1 La classe Collection

En Java, la classe `Collection`<sup>1</sup> regroupe les structures de données les plus utilisées dans une forme standard. Elle incorpore plusieurs interfaces, associées à plusieurs implantations. De nombreux algorithmes de manipulation sont déjà implantés, ce qui facilite la réutilisation de code, donc simplifie l'écriture des programmes, et permet de les écrire dans une forme relativement standardisée.

Le cœur des collections est donné dans le diagramme suivant :



Remarquons que ces interfaces sont génériques, donc permettent de définir des Collections de type quelconque. Les collections permettent de gérer des ensembles d'éléments.

---

<sup>1</sup><http://java.sun.com/docs/books/tutorial/collections/index.html>

La classe `Set` s'intéresse aux ensembles dont les éléments ne sont pas dupliqués ; `SortedSet` stocke les éléments de manière ascendante. La `List` permet de stocker des éléments (éventuellements égaux) en ayant un certain contrôle sur l'endroit de stockage. Une `Queue` permet de modéliser une file ou une file de priorité.

Une `Map` associe une valeur à une clef. Ce sera la base des tables de hachage. `SortedMap` est la variante gardant un ordre sur les clefs.

Une collection peut être parcourue à l'aide de :

```
for(Object o : collection)
    System.out.println(o);
```

ou bien

```
for(Iterator<?> it = c.iterator(); it.hasNext(); )
    System.out.println(it.next());
```

Dans le cas où on veut pouvoir supprimer un élément, il faut utiliser l'itérateur explicite, comme dans :

```
static void filter(Collection<?> c) {
    for(Iterator<?> it = c.iterator(); it.hasNext(); )
        if(!cond(it.next()))
            it.remove();
}
```

Les autres opérations possibles sur les collections sont décrites dans la documentation de Java, ainsi que les fonctions sur les autres classes.

### 14.1.1 La classe `Vector`

Cette classe implante des tableaux de taille variable, c'est-à-dire que le tableau peut croître ou diminuer en fonction des accès. Il est conseillé dans certains cas d'utiliser `ArrayList` à la place de `Vector`, notamment dans le cas d'accès concurrents.

### 14.1.2 La classe `LinkedList`

Elle permet d'implanter les listes chaînées, les piles et les files, ainsi que les listes appelées *dequeue* (double-ended queues). Parmi les opérations intéressantes, notons `addFirst`, `removeFirst` (qui permettent de simuler une pile), ou bien de même `addLast`, `removeLast`, toutes opérations se faisant en temps constant.

### 14.1.3 La classe `Hashtable`

Cette classe permet d'implanter une table de hachage d'objets (non nuls), à condition que la classe des clefs implante les deux méthodes `equals` et `hashCode`. La table peut être paramétrée à la création en fonction d'une taille prévisible.

L'exemple typique d'utilisation est :

```
Hashtable nombres = new Hashtable();
nombres.put("un", new Integer(1));
nombres.put("deux", new Integer(2));
nombres.put("trois", new Integer(3));

Integer n = (Integer)nombres.get("deux");
if(n != null){
    System.out.println("deux = " + n);
}
```

La méthode `containsKey(Object key)` permet de tester si la clef existe dans la table. Toutes les opérations d'insertion, tests d'appartenance et suppression se font en  $O(1)$ .

#### 14.1.4 La classe TreeSet

C'est la classe qui permet d'implanter les files de priorité sur des objets. Il faut que la classe en question soit une extension de type `Comparable` ou utilise un comparateur explicite.

Donnons un premier exemple, qui implante une classe paire d'entiers :

```
import java.io.*;
import java.util.*;

public class Paire implements Comparable{
    int i, j;

    Paire(int ii, int jj){
        i = ii; j = jj;
    }

    public int compareTo(Object o){
        Paire p = (Paire)o;
        if(i < p.i) return -1;
        if(i > p.i) return 1;
        if(j < p.j) return -1;
        if(j > p.j) return 1;
        return 0;
    }

    public boolean equals(Object o){
        return compareTo(o) == 0;
    }
}
```

```

public static void main(String[] args){
    TreeSet<Paire> t = new TreeSet<Paire>();

    for(int i = 1; i >= 0; i--){
        for(int j = 1; j >= 0; j--){
            t.add(new Paire(i, j));
        }
    }
    while(! t.isEmpty()){
        Paire p = t.first();
        t.remove(p);
        System.out.println(p.i+" "+p.j);
    }
}

```

L'exécution du programme donnera :

```

0 0
0 1
1 0
1 1

```

On aurait pu écrire également le programme suivant :

```

public static void main(String[] args){
    TreeSet<Paire> t =
        new TreeSet<Paire>(
            new Comparator<Paire>(){
                public int compare(Paire p1, Paire p2){
                    if(p1.i < p2.i) return -1;
                    if(p1.i > p2.i) return 1;
                    if(p1.j < p2.j) return -1;
                    if(p1.j > p2.j) return 1;
                    return 0;
                }
            }
        );

    for(int i = 1; i >= 0; i--){
        .....
    }
}

```

pour le même résultat.

## Chapitre 15

# La classe Graphe

**Remerciements :** le package `grapheX` a été mis au point avec P. Chassignet, qui a également participé à l'écriture de la documentation qui suit. Merci à J. Cervelle pour de nombreuses discussions sur le sujet. Les classes ont été testées en même temps que le poly a été écrit. Le package est disponible sur la page web du cours.

### 15.1 Les choix

On fait souvent l'hypothèse que les sommets d'un graphe sont numérotés de 0 à  $n - 1$ , de façon à utiliser des implantations à base de matrices d'adjacence ou bien de pouvoir faire des calculs. Toutefois, de nombreux algorithmes sur les graphes sont écrits en manipulant des sommets abstraits. Il nous a paru plus proche d'une certaine réalité d'en faire autant.

Après réflexion, écrire une classe générique de graphe est une tâche quasi-impossible, car chaque algorithme qui opère sur les graphes utilise des structures de données auxiliaires et on ne peut envisager tous les cas possibles. Néanmoins, une certaine abstraction est possible. Cela a conduit à définir une classe `Sommet` qui code un sommet, dont le nom est une chaîne.

Deux sommets, différents au sens de l'objet Java, mais construits avec le même nom, seront considérés comme identiques pour les méthodes qui opèrent sur un graphe. Pour assurer cela, quelles que soient les structures de données que l'on utilise dans l'implantation d'un graphe, il convenait de fixer un certains nombres de propriétés d'un objet `Sommet`. Ces propriétés sont finalement très générales et cela justifie l'introduction d'une classe `Identifiable` dont `Sommet` hérite. En fonction des applications, il sera facile de définir de nouvelles classes de sommets sur le même modèle.

Pour aller au-delà, la classe générique `Arc` s'est imposée, un arc générique est formé de deux sommets (origine et extrémité) qui héritent de `Identifiable`, et il est doté d'une valeur (un entier). On peut alors réaliser des listes de sommets, des piles d'arcs, etc.

On peut ensuite définir une classe `GrapheGenerique` qui est à la fois abstraite et générique (par le type des sommets) et qui spécifie les fonctions de base sur un

graphe. La spécialisation de `GrapheGenerique` à la classe `Sommet` nous donne la classe abstraite `Graphe` qui nous permettra d'écrire de manière un peu plus lisible tous les algorithmes du poly. Cette classe `Graphe` est accompagnée de deux implantations : `GrapheMatrice` (pour utiliser des matrices d'adjacence) et `GrapheListe` qui utilise des tableaux de listes de voisin. La figure 15.1 donne un aperçu graphique du contenu du package et des liens entre les différentes classes.

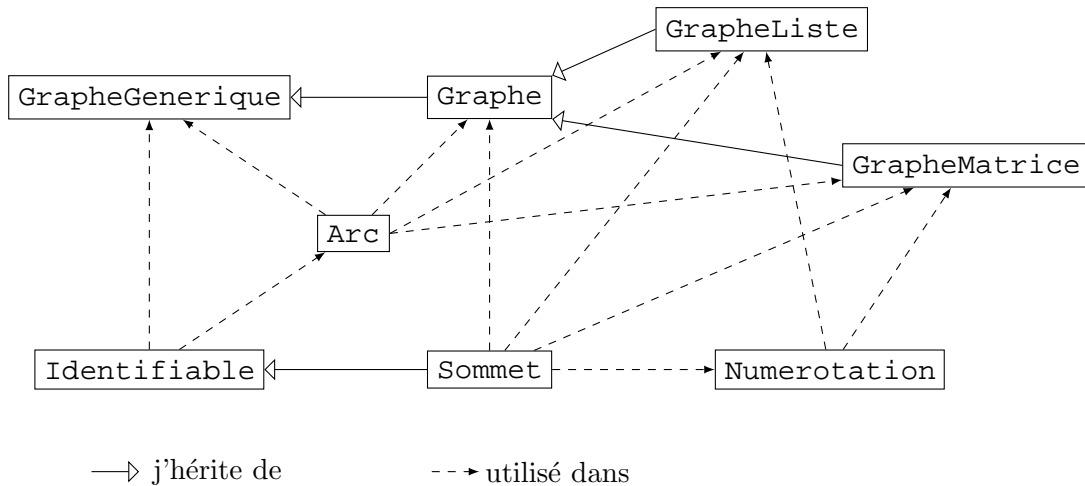


FIG. 15.1 – Panorama des classes du package `grapheX`.

Tout a été écrit dans `Graphe` de façon à privilégier les parcours sur les voisins d'un sommet, sans savoir quelle représentation sera finalement prise pour un problème donné. Pour faciliter le traitement des sommets du graphe, on introduit une classe numérotation qui permet d'assurer un ordre sur les sommets par exemple. Pour en comprendre l'intérêt, on regardera le code de la fonction `deFichier` de la classe `GrapheListe`.

## 15.2 `grapheX` comme exemple de package

Il est apparu naturel de regrouper tout cela dans un **package** au sens de Java, auquel le nom de **grapheX** a été donné. Cela permet de donner un exemple réel d'utilisation de package.

### 15.2.1 Comment ça marche ?

Après récupération sur le réseau<sup>1</sup>, on exécute les commandes :

```
tar zxvf grapheX.tgz
```

<sup>1</sup><http://www.enseignement.polytechnique.fr/informatique/INF431/>

Cela crée un répertoire `grapheX`, qui contient les fichiers sources :

```
Arbre.java           GrapheListe.java      Sommet.java
Arc.java            GrapheMatrice.java   SommetValue.java
Graphe.java         Identifiable.java    package-info.java
GrapheGenerique.java Numerotation.java
```

Notez la commande **package** `grapheX`; présente dans chacun des fichiers.

Au même niveau que `grapheX` se trouve un fichier de menu étendu qui montre comment utiliser les fonctions contenues dans le package (fichier `TestGraphe.java`). Ce fichier se compile par :

```
javac TestGraphe.java
```

et ira compiler tout seul dans le répertoire `grapheX` les différentes classes, les fichiers `.class` restant dans `grapheX`.

### 15.2.2 Faire un .jar

Si on le souhaite, on peut faire un `.jar`. Nous en profitons pour donner la syntaxe (en Unix), qui n'est pas si évidente que cela :

```
javac grapheX/*.java # il faut les .class!
jar -cf grapheX.jar grapheX
```

On peut alors compiler `TestGraphe.java` :

```
javac -cp grapheX.jar:. TestGraphe.java
```

Pour exécuter, il suffit de taper

```
java -cp grapheX.jar:. TestGraphe ../Data/prim1.in Prim
```

## 15.3 Sommets

Un sommet d'un graphe sera supposé contenir comme information une chaîne de caractères qui sert d'identifiant. On peut facilement afficher l'identifiant d'un sommet `s` à l'écran par :

```
System.out.println(s);
```

Nous aurons besoin d'utiliser les sommets dans diverses structures de `java.util`, notamment comme des clefs dans des tables, en sachant que nous voulons considérer comme identiques deux sommets qui sont construits avec le même identifiant. Il faut donc définir correctement la méthode `equals` pour quelle retourne **true** si et seulement si les deux objets qu'elle compare ont le même identifiant. Pour des tables de hachage, il faut aussi la méthode `hashCode`. Pour des structures ordonnées comme

les arbres, il faut la méthode `compareTo` et nos sommets doivent implanter l'interface `Comparable`. Enfin, les méthodes `hashCode` et `compareTo` doivent être consistantes avec `equals`. En complément, la fonction `compare` est un équivalent statique à `compareTo`.

```

package grapheX;

/**
 * Définit des objets identifiés par une String. Deux
 * objets Identifiable qui sont construits avec le
 * même identifiant seront considérés identiques pour
 * les méthodes equals, hashCode et
 * compareTo. Cette classe sert notamment de
 * super-classe pour les sommets d'un graphe.
 *
 * @author PChassignet (chassign@lix.polytechnique.fr)
 * @version 2007.03.21
 */

public class Identifiable implements Comparable<Identifiable> {
    private final String ident;

    /**
     * @param l'identifiant pour cet objet.
     */
    public Identifiable(String identifiant) {
        ident = identifiant;
    }

    /**
     * @return l'identifiant pour cet objet.
     */
    public final String identifiant() {
        return ident;
    }

    /**
     * @return equals sur les identifiants.
     */
    public final boolean equals(Object o) {
        if (!(o instanceof Identifiable))
            return false;
        String oid = ((Identifiable) o).ident;
        if (ident == null)

```

```
        return oid == null;
    else
        return ident.equals(oid);
}

/**
 * @return hashCode sur l'identifiant.
 */
public final int hashCode() {
    if (ident == null)
        return 0;
    else
        return ident.hashCode();
}

/**
 * @return compareTo sur les identifiants.
 */
public final int compareTo(Identifiable id) {
    if (ident == null)
        if (id.ident == null)
            return 0;
        else
            return -1;
    else if (id.ident == null)
        return 1;
    else
        return ident.compareTo(id.ident);
}

/**
 * @param id1 le premier objet à comparer,
 * @param id2 le second objet à comparer,
 * @return le résultat de id1.compareTo(id2).
 */
public static final int compare(Identifiable id1,
    Identifiable id2) {
    if (id1 == null)
        if (id2 == null)
            return 0;
        else
            return -1;
    else if (id2 == null)
        return 1;
```

```

    else
        return id1.compareTo(id2);
    }

    public String toString() {
        if (ident == null)
            return "[null]";
        else
            return "[\"" + ident + "\"]";
    }
}

```

Les principales méthodes de `Identifiable` sont déclarées **final** pour se protéger d'une redéfinition qui pourrait détruire les propriétés voulues. Notre classe `Sommet` a alors une définition basique :

```

package grapheX;

/**
 * Classe de sommets, toutes les propriétés sont héritées de
 * Identifiable.
 *
 * @author FMorain (morain@lix.polytechnique.fr)
 * @author PChassignet (chassignet@lix.polytechnique.fr)
 * @version 2007.03.21
 */

public class Sommet extends Identifiable {

    public Sommet(String nn) {
        super(nn);
    }

    public String toString() {
        return identifiant();
    }
}

```

on a simplement redéfini `toString` pour alléger les affichages. Les propriétés qui sont héritées de `Identifiable` peuvent être illustrées par l'exemple suivant :

```

import java.util.*;
import grapheX.*;
public class Test{
    public static void main(String args[]){
        HashSet<Sommet> HS = new HashSet<Sommet>();
    }
}

```

```

    HS.add(new Sommet("a"));
    System.out.println(HS.contains(new Sommet("a")));
}
}

```

qui affiche **true** car les deux objets construits par `new Sommet("a")` sont considérés comme identiques. La valeur de hachage dépend seulement du nom du sommet. On aurait le même résultat en remplaçant `HashSet` par `TreeSet` qui utilise `compareTo`.

## 15.4 Sommets valués

De nombreux algorithmes sur les graphes utilisent le fait d'attribuer des marques booléennes ou des valeurs numériques aux sommets d'un graphe. On pourrait être tenté de définir des sommets avec un champ `valeur`, comme dans l'exemple suivant :

```

package grapheX;

public class SommetValue {
    Sommet s;
    int valeur;

    SommetValue(Sommet ss, int vv){
        s = ss;
        valeur = vv;
    }
}

```

ou, ce qui change peu :

```

package grapheX;

public class SommetValueVariante extends Sommet {
    int valeur;

    SommetValueVariante(Sommet ss, int vv){
        super(ss.identifiant());
        valeur = vv;
    }
}

```

ou, encore, en étendant directement `Identifiable`. Dans tous ces cas, il y a le risque d'une incohérence qui consisterait à avoir deux objets, considérés comme identiques au sens défini par `Identifiable`, mais qui auraient des champs `valeur` différents. La classe `SommetValue` est utilisée dans un contexte très particulier où on sait contrôler de telles incohérences.

Dans un contexte plus général, il est préférable d'attribuer des valeurs aux sommets par l'intermédiaire d'une table, par exemple pour des entiers :

```
HashMap<Sommet,Integer> etat = new HashMap<Sommet,Integer>();
```

## 15.5 La classe Arc

La classe Arc est générique et elle est paramétrée par le type des sommets. Nous utilisons les mêmes principes que pour les sommets, en permettant la gestion facile de tables indexées par des arcs.

```
package grapheX;

/**
 * Classe d'arcs
 *
 * @author FMorain (morain@lix.polytechnique.fr)
 * @author PChassignet (chassignet@lix.polytechnique.fr)
 * @author JCervelle (julien.cervelle@univ-mlv.fr)
 * @version 2007.03.21
 */

// L'arc o -> d avec valeur val
public class Arc<S extends Identifiable>
    implements Comparable<Arc<S>> {
    private S o, d;
    private int val;

    public Arc(S o0, S d0, int val0) {
        this.o = o0;
        this.d = d0;
        this.val = val0;
    }

    public Arc(S o0, S d0) {
        this.o = o0;
        this.d = d0;
        this.val = 0;
    }

    public Arc(Arc<S> a) {
        this.o = a.o;
        this.d = a.d;
        this.val = a.val;
    }
}
```

```
}

public S destination() {
    return d;
}

public S origine() {
    return o;
}

public int valeur() {
    return val;
}

public void modifierValeur(int vv) {
    this.val = vv;
}

public String toString() {
    return "(" + this.o + ", " + this.d + ")";
}

public int hashCode() {
    int codeOri = (o == null ? 0 : o.hashCode());
    int codeDst = (d == null ? 0 : d.hashCode());
    return codeDst ^ (codeOri * 31);
}

public boolean equals(Object aa) {
    if (!(aa instanceof Arc))
        return false;
    Arc<?> arc = (Arc<?>) aa;
    boolean equalsO = o == null && arc.o == null || o != null
        && o.equals(arc.o);
    boolean equalsD = d == null && arc.d == null || d != null
        && d.equals(arc.d);
    return equalsO && equalsD && (val == arc.val);
}

public int compareTo(Arc<S> a) {
    int comp = Identifiable.compare(this.o, a.o);
    if (comp == 0)
        comp = Identifiable.compare(d, a.d);
    return comp;
}
```

```
}
}
```

On fera une remarque sur les méthodes `hashCode` et `compareTo`. On veut qu'un arc soit repérable par ses deux sommets, pas par sa valeur. On pourrait changer ce comportement si besoin est. Il faut pourtant respecter les contraintes de cohérence imposées par l'usage des tables de `java.util`, à savoir que si `equals` répond `true`, alors les deux objets ont même `hashCode` et `compareTo` répond 0.

## 15.6 La classe abstraite Graphe et deux implantations

Le but de la classe `Graphe` est de permettre de manipuler facilement des graphes, comme on peut le lire dans les chapitres du poly concernant les graphes. Notons en passant que nous utiliserons systématiquement des itérateurs sur les sommets d'un graphe ou les voisins d'un sommet.

Le parti pris est celui d'une collection d'arcs, donc d'un graphe *a priori* orienté. Dans de rares cas, l'utilisation d'une classe plus spécifique d'arête au lieu d'arc pourrait être envisagée.

### 15.6.1 Définitions génériques

Ces quelques méthodes suffisent à implanter tous les algorithmes décrits dans le cours.

```
package grapheX;
import java.util.Collection;

/**
 * Super-classe abstraite des graphes, les sommets doivent
 * être identifiables.
 *
 * @author FMorain (morain@lix.polytechnique.fr)
 * @author PChassignet (chassignet@lix.polytechnique.fr)
 * @version 2007.03.21
 */

public abstract class GrapheGenerique<S extends Identifiable> {

    /**
     * @return le nombre de sommets de ce graphe.
     */
    public abstract int taille();

    /**
     * @param s le sommet à ajouter à ce graphe.

```

```
*/
public abstract void ajouterSommet(S s);

/**
 * @return une <tt>Collection</tt> de tous les sommets de ce
 *         graphe.
 */
public abstract Collection<S> sommets();

/**
 * Teste l'existence de l'arc de <tt>s</tt> à <tt>t</tt>
 * dans ce graphe.
 * @param s l'origine de l'arc,
 * @param t l'extrémité de l'arc.
 */
public abstract boolean existeArc(S s, S t);

/**
 * @param s l'origine de l'arc,
 * @param t l'extrémité de l'arc,
 * @param val une valeur entière attachée à l'arc
 *           de <tt>s</tt> à <tt>t</tt> dans ce graphe.
 * Cela revient à modifier la valeur s'il y a déjà un arc
 * de <tt>s</tt> à <tt>t</tt> dans ce graphe.
 */
public abstract void ajouterArc(S s, S t, int val);

/**
 * @param s l'origine de l'arc,
 * @param t l'extrémité de l'arc.
 * @return la valeur entière attachée à l'arc
 *        de <tt>s</tt> à <tt>t</tt> dans ce graphe.
 */
public abstract int valeurArc(S s, S t);

/**
 * Supprime l'arc de <tt>s</tt> à <tt>t</tt> dans ce
 * graphe.
 * @param s l'origine de l'arc,
 * @param t l'extrémité de l'arc.
 */
public abstract void enleverArc(S s, S t);

/**
```

```

* @param s l'origine des arcs.
* @return une <tt>Collection</tt> de tous les arcs de ce
*         graphe ayant <tt>s</tt> pour origine. Ces arcs
*         sont de type <tt>Arc<S></tt>.
*/
public abstract Collection<Arc<S>> voisins(S s);

/**
* @return une copie de ce graphe, les arcs sont dupliqués.
*/
public abstract GrapheGenerique<S> copie();
}

```

### 15.6.2 La classe Graphe

La classe Graphe est maintenant définie ainsi :

```
public abstract class Graphe extends GrapheGenerique<Sommet>
```

Il s'agit d'une spécialisation de la classe générique pour des sommets de type Sommet, les arcs sont alors de type Arc<Sommet>. On donne ci-dessous un équivalent des définitions qui sont héritées (les méthodes restent abstraites) :

```

public abstract int taille();
public abstract void ajouterSommet(Sommet s);
public abstract Collection<Sommet> sommets();
public abstract boolean existeArc(Sommet s, Sommet t);
public abstract void ajouterArc(Sommet s, Sommet t, int val);
public abstract int valeurArc(Sommet s, Sommet t);
public abstract void enleverArc(Sommet s, Sommet t);
public abstract Collection<Arc<Sommet>> voisins(Sommet s);

```

et on doit redéclarer

```
public abstract Graphe copie();
```

pour préciser le type de retour.

### 15.6.3 Numérotation

Cette classe est utilisée dans les deux implantations de la classe abstraite, car nous avons souvent besoin (en interne) d'un ordre sur les sommets.

Il est important, en interne, d'avoir un moyen de numéroter les sommets lors de leur création. C'est le rôle de cette classe, qui sera utilisée dans les deux classes GrapheMatrice et GrapheListe. Pour que l'utilisateur ne soit pas tenté d'utiliser cette numérotation, tous les champs sont privés. On peut récupérer des itérateurs sur les sommets, au moyen de la méthode `elements()`.

```
package grapheX;

import java.io.*;
import java.util.*;

/**
 * Nume'rotation des graphes
 *
 * @author FMorain (morain@lix.polytechnique.fr)
 * @author PChassignet (chassignet@lix.polytechnique.fr)
 * @version 2006.11.22
 */
public class Numerotation{
    private int compteur;
    private Hashtable<Sommet,Integer> HSI;
    private Vector<Sommet> VS;

    public Numerotation(int n){
        compteur = -1;
        HSI = new Hashtable<Sommet,Integer>();
        VS = new Vector<Sommet>(n);
        VS.setSize(n);
    }

    public int taille(){
        return VS.size();
    }

    public boolean ajouterElement(Sommet s){
        if(!HSI.containsKey(s)){
            compteur++;
            HSI.put(s, compteur);
            VS.set(compteur, s);
            return true;
        }
        return false;
    }

    public int numero(Sommet s){
        return HSI.get(s);
    }
}
```

```

    public Sommet elementAt(int i){
        return VS.elementAt(i);
    }

    public Collection<Sommet> elements(){
        return VS;
    }
}

```

#### 15.6.4 Implantation par matrice

C'est là une implantation proche des matrices d'adjacence, même si on utilise des vecteurs de vecteurs.

```

package grapheX;

import java.io.*;
import java.util.*;

/**
 * Graphes implante's dans des "matrices"
 *
 * @author FMorain (morain@lix.polytechnique.fr)
 * @version 2007.01.12
 */
public class GrapheMatrice extends Graphe{
    private Vector<Vector<Arc>> M;
    private Numerotation numerotation;

    public int taille(){
        return M.size();
    }

    public GrapheMatrice(int n){
        numerotation = new Numerotation(n);
        M = new Vector<Vector<Arc>>(n);
        M.setSize(n);
    }

    public void ajouterSommet(Sommet s){
        if(numerotation.ajouterElement(s)){
            int n = taille();
            Vector<Arc> vs = new Vector<Arc>(n);

```

```
        vs.setSize(n);
        M.set(enumerotation.numero(s), vs);
    }
}

public boolean existeArc(Sommet s, Sommet t){
    int si = enumerotation.numero(s);
    int ti = enumerotation.numero(t);
    return M.get(si).get(ti) != null;
}

private boolean existeArc(int i, int j){
    return M.get(i).get(j) != null;
}

public void ajouterArc(Sommet s, Sommet t, int val){
    ajouterSommet(s);
    ajouterSommet(t);
    int si = enumerotation.numero(s);
    int ti = enumerotation.numero(t);
    M.get(si).set(ti, new Arc(s, t, val));
}

public int valeurArc(Sommet s, Sommet t){
    int si = enumerotation.numero(s);
    int ti = enumerotation.numero(t);
    return M.get(si).get(ti).valeur();
}

private int valeurArc(int i, int j){
    return M.get(i).get(j).valeur();
}

public void enleverArc(Sommet s, Sommet t){
    int si = enumerotation.numero(s);
    int ti = enumerotation.numero(t);
    M.get(si).remove(ti);
}

public void modifierValeur(Sommet s, Sommet t, int val){
    int si = enumerotation.numero(s);
    int ti = enumerotation.numero(t);
    M.get(si).get(ti).modifierValeur(val);
}
}
```

```
public LinkedList<Arc> voisins(Sommet s){
    LinkedList<Arc> l = new LinkedList<Arc>();
    int si = numerotation.numero(s);

    for(int j = 0; j < taille(); j++)
        if(existeArc(si, j))
            l.addLast(M.get(si).get(j));
    return l;
}

public Collection<Sommet> sommets(){
    return numerotation.elements();
}

public GrapheMatrice copie(){
    int n = taille();
    GrapheMatrice G = new GrapheMatrice(n);
    for(int i = 0; i < n; i++)
        G.ajouterSommet(numerotation.elementAt(i));
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            if(M.get(i).get(j) != null)
                G.ajouterArc(numerotation.elementAt(i),
                    numerotation.elementAt(j),
                    valeurArc(i, j));

    return G;
}

public static GrapheMatrice deMatrice(int[][] M){
    int n = M.length;
    GrapheMatrice G = new GrapheMatrice(n);

    for(int i = 0; i < n; i++)
        G.ajouterSommet(new Sommet(i+""));
    for(int i = 0; i < n; i++){
        for(int j = 0; j < M[i].length; j++)
            if(M[i][j] == 1)
                G.ajouterArc(G.numerotation.elementAt(i),
                    G.numerotation.elementAt(j),
                    1);
    }
    return G;
}
```

```
}
```

### 15.6.5 Implantation avec des listes

```
package grapheX;

import java.io.*;
import java.util.*;

/**
 * Graphes implantes dans des "listes"
 *
 * @author FMorain (morain@lix.polytechnique.fr)
 * @version 2007.01.30 [propagation modifs de Arc]
 */

public class GrapheListe extends Graphe{
    private Vector<LinkedList<Arc>> L;
    private Numerotation numerotation;

    public int taille(){
        return L.size();
    }

    private static void remplir(GrapheListe G, int n){
        G.numerotation = new Numerotation(n);
        G.L = new Vector<LinkedList<Arc>>(n);
        G.L.setSize(n);
        G.orienté = true; // default
    }

    // construction d un graphe vide a n sommets
    public GrapheListe(){
    }

    public GrapheListe(int n){
        remplir(this, n);
    }

    public GrapheListe(String str){
        remplirAvecScanner(this, new Scanner(str));
    }
}
```

```
public void ajouterSommet(Sommet s){
    if(enumerotation.ajouterElement(s))
        L.set(enumerotation.numero(s), new LinkedList<Arc>());
}

public boolean existeArc(Sommet s, Sommet t){
    for(Arc a : L.get(enumerotation.numero(s)))
        if((a.destination()).equals(t))
            return true;
    return false;
}

private boolean existeArc(int i, int j){
    Sommet t = enumerotation.elementAt(j);
    for(Arc a : L.get(i))
        if(a.destination().equals(t))
            return true;
    return false;
}

public void ajouterArc(Sommet s, Sommet t, int val){
    ajouterSommet(s);
    ajouterSommet(t);
    int si = enumerotation.numero(s);
    L.get(si).addLast(new Arc(s, t, val));
}

public void ajouterArc(int i, int j, int val){
    L.get(i).addLast(new Arc(enumerotation.elementAt(i),
        enumerotation.elementAt(j),
        val));
}

public int valeurArc(Sommet s, Sommet t){
    for(Arc a : L.get(enumerotation.numero(s)))
        if(a.destination().equals(t))
            return a.valeur();
    return -1; // convention
}

public int valeurArc(int i, int j){
    Sommet t = enumerotation.elementAt(j);
    for(Arc a : L.get(i))
```

```
        if(a.destination().equals(t))
            return a.valeur();
    return -1; // convention
}

public void enleverArc(Sommet s, Sommet t){
    int si = numerotation.numero(s);
    Arc a = null;
    for(Arc aa : L.get(numerotation.numero(s)))
        if(aa.destination().equals(t)){
            a = aa;
            break;
        }
    if(a != null)
        L.get(numerotation.numero(s)).remove(a);
}

public void modifierValeur(Sommet s, Sommet t, int val){
    for(Arc a : L.get(numerotation.numero(s)))
        if(a.destination().equals(t)){
            a.modifierValeur(val);
            return;
        }
}

public LinkedList<Arc> voisins(Sommet s){
    return L.get(numerotation.numero(s));
}

public Collection<Sommet> sommets(){
    return numerotation.elements();
}

public GrapheListe copie(){
    int n = taille();
    GrapheListe G = new GrapheListe(n);
    for(int i = 0; i < n; i++)
        G.ajouterSommet(numerotation.elementAt(i));
    for(int i = 0; i < n; i++){
        // recopie dans le meme ordre
        LinkedList<Arc> Li = G.L.get(i);
        for(Arc a : L.get(i))
            Li.addLast(a);
    }
}
```

```

    return G;
}

// retourne vrai si le caractere c est dans str
private static boolean option(String str, char c){
    for(int i = 0; i < str.length(); i++)
        if(str.charAt(i) == c)
            return true;
    return false;
}

public static void remplirAvecScanner(GrapheListe G, Scanner scan){
    System.out.println(scan.next());
    int n = scan.nextInt();
    remplir(G, n);
    String str = scan.next();
    boolean estValue = option(str, 'v');
    boolean estSym = option(str, 's');
    boolean avecCouples = option(str, 'c');

    if(estSym)
        G.oriente = false;
    System.out.println("n = "+n);
    for(int i = 0; i < n; i++){
        Sommet s = new Sommet(scan.next());
        G.ajouterSommet(s);
    }
    if(avecCouples){
        System.out.println("Avec couples");
        // on lit des lignes "i j dij" ou "i j"
        while(scan.hasNext()){
            Sommet s = new Sommet(scan.next());
            Sommet t = new Sommet(scan.next());
            int si = G.numerotation.numero(s);
            int ti = G.numerotation.numero(t);
            if(estValue)
                G.ajouterArc(si, ti, (int)scan.nextInt());
            else
                G.ajouterArc(si, ti, 1);
        }
    }
    else{
        // format "s ns t0 t1 ... t{ns-1}"
        // ou      "s ns t0 v0 t1 v1 ... t{ns-1} v{ns-1}"
    }
}

```

```

System.out.println("Avec listes, estvalue="+estValue);
for(int r = 0; r < n; r++){
    Sommet s = new Sommet(scan.next());
    int si = G.numerotation.numero(s);
    int nj = (int)scan.nextInt();
    for(int k = 0; k < nj; k++){
        Sommet t = new Sommet(scan.next());
        int ti = G.numerotation.numero(t);
        if(estValue)
            G.ajouterArc(si, ti,
                (int)scan.nextInt());
        else
            G.ajouterArc(si, ti, 1);
    }
}
}
System.out.println("G="+G);
if(estSym)
    // on doit symetriser G
    for(Sommet s : G.sommets())
        for(Sommet t : G.sommets())
            if(G.existeArc(s, t)
                && !G.existeArc(t, s))
                G.ajouterArc(t, s,
                    G.valeurArc(s, t));
}

public static GrapheListe deFichier(String nomfic){
    try{
        Scanner scan =
            new Scanner(
                new BufferedReader(new FileReader(nomfic)));
        GrapheListe G = new GrapheListe();
        remplirAvecScanner(G, scan);
        return G;
    }
    catch(Exception e) { }
    return null;
}
}

```

La méthode `deFichier` donne un exemple d'utilisation des différentes primitives. Donnons un exemple de fichier d'entrée. Le fichier :

```
#prim1
4 vs
u v w x
u 2 v 16 x 29
v 3 u 16 x 20 w 25
w 2 v 25 x 10
x 3 u 29 v 20 w 10
```

La première ligne est un commentaire qui n'est pas traité par le programme. La seconde contient le nombre de sommets du graphe. Suivent des codes comme `v` pour graphe valué, `s` pour graphe symétrique (non orienté). La ligne suivante contient les noms des 4 sommets du graphe. Ils seront stockés dans cet ordre dans une table, ce qui fait que

```
for(Sommet s : sommets())
    System.out.println(s);
```

affichera

```
u
v
w
x
```

Suivent alors 4 lignes. Chacune commence par un nom de sommet, puis par le nombre de voisins  $k$ , puis  $k$  paires nom de sommet suivi de la valeur de l'arc. Ce fichier correspond au graphe de la figure 9.6.

Un graphe non valué est stocké sous une forme plus simple :

```
#pda1
6 s
a b c d e f
a 4 b d c f
b 2 a d
c 3 a e f
d 2 a b
e 2 a c
f 1 c
```

C'est le graphe représenté à la figure 7.6.

# Bibliographie

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison–Wesley, 1986. A été traduit en français.
- [2] J. M. Aldous and R. J. Wilson. *Graphs and applications – an introductory approach*. Springer, 2000.
- [3] D. Beauquier, J. Berstel, and P. Chrétienne. *Éléments d’algorithmique*. Masson, Paris, 1992.
- [4] F. P. Brooks, Jr. *The mythical man-month – Essays on Software Engineering – Anniversary Edition*. Addison-Wesley, 1995.
- [5] R. Cori and J.-J. Lévy. *Algorithmes et Programmation*. Cours de l’École polytechnique, 1998.
- [6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.
- [7] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1988.
- [8] D. Kozen. *Automata and Computability*. Springer, 1997. <http://www.cs.cornell.edu/~kozen/>.
- [9] J.-J. Lévy. *Informatique fondamentale*. Cours de l’École polytechnique, 2004.
- [10] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial algorithms – theory and practice*. Prentice-Hall, 1977.