Option informatique

Olivier Reynet

17-01-2023

TABLE DES MATIÈRES

I Introduction **3**

1 Information, concepts et ensembles **5**
A Les ensembles et la logique 5
B Ensembles inductifs 6
C Propriétés des ensembles inductifs 7
D Des fonctions pour calculer sur les termes d'un ensemble inductif 7
E Pourquoi OCaml? 8

2 OCaml, des fonctions et des types **9**
A Pratiquer OCaml 9
B Vocabulaire utile 9
C Description d'OCaml 10
D Expressions et inférence de type 11
E Expressions locales et globales 12
F Fonctions 13
G Effets 14
H Types algébriques 15
I Listes 18
J Filtrage de motif 19
K Références et programmation impérative en OCaml 21

II Logique **23**

3 De la logique avant toute chose **25**
A Logique et applications 25
B Constantes, variables propositionnelles et opérateurs logiques 27
C Opérateurs logique et notations 27
D Formules logiques : définition inductive et syntaxe 29
E Sémantique et valuation des formules logiques 32
F Lois de la logique 35
a Éléments neutres et absorbants, idempotence 35
b Commutativité, distributivité, associativité 35
c Lois de De Morgan 35

d Décomposition des opérateurs 35
e Démonstrations 36
G Principes et logique classique 36
H Formes normales 37
I Problème SAT 39
J Algorithme de Quine 39
a Principe 39
b Règles de simplification 41
c Algorithme de Quine 41

4 Déduction naturelle **43**

III Programmation récursive **45**

IV Structures de données **47**

V Exploration et graphes **49**

5 Retour sur trace **51**
A Exploration 51
B Principe du retour sur trace 53

6 Les mots des graphes **55**
A Typologie des graphes 55
B Implémentation des graphes 59
C Caractérisation structurelle des graphes 60
D Isomorphisme des graphes 61
E Chaînes, cycles et parcours 63
F Sous-graphes et connexité 67
G Coloration de graphes 67
H Distances 70
I Arbres 70

7 Propriétés des graphes **73**
A Des degrés et des plans 73
B Caractérisation des chaînes, des cycles et des graphes 74
C Graphes acycliques et connexes 75
D Coloration, graphes planaires et nombre chromatique 75
E Principe d'optimalité et plus court chemin dans un graphe 76

8 Algorithmes et graphes **77**
A Parcours d'un graphe . . . 77
a Parcours en largeur . . . 78
b Terminaison et correction du parcours en largeur . . . 79
c Complexité du parcours en largeur . . . 80
d Parcours en profondeur . . . 80
B Trouver un chemin dans un graphe . . . 81
C Plus courts chemins dans les graphes pondérés . . . 83
a Algorithme de Dijkstra . . . 83
b Algorithme de Bellman-Ford . . . 87
c Algorithme de Floyd-Warshall . . . 89
d A* . . . 91
D Arbres recouvrants . . . 92
a Algorithme de Prim . . . 93
b Algorithme de Kruskal . . . 93
E Tri topologique d'un graphe orienté . . . 94
a Ordre dans un graphe orienté acyclique . . . 94
b Tri topologique et détection de cycles dans un graphe orienté . . . 95
F Composantes fortement connexes d'un graphe orienté et 2-SAT . . . 95
G Graphes bipartis et couplage maximum . . . 96
a Couplage dans un graphe biparti . . . 96
b Chemin augmentant . . . 98

9 Des arbres aux tas **103**
A Des arbres . . . 103
B Arbres binaires . . . 105
C Induction et arbre binaire . . . 106
D Tas binaires . . . 107
a Définition . . . 107
b Implémentation . . . 107
c Opérations . . . 108
E Tri par tas binaire . . . 110
F File de priorités implémentée par un tas . . . 111

VI Langages et automates **113**

10 Introduction aux langages **115**
A Alphabets . . . 115
B Mots . . . 116
C Mots définis inductivement . . . 117
D Langages . . . 119
E Préfixes, suffixes, facteurs et sous-mots . . . 121
F Propriétés fondamentales . . . 123

11 Expressions régulières **125**
A Définition des expressions régulières . . . 126
B Définition des langages réguliers . . . 128
C Identités remarquables sur les expressions régulières . . . 128
D Arbre associé à une expression régulière . . . 129
E Expressions régulières dans les langages --→ HORS PROGRAMME . . . 130

12 Automates finis déterministes **133**
A Automate fini déterministe (AFD) . . . 133
B Représentation d'un automate . . . 134
C Acceptation d'un mot . . . 134
D Accessibilité et co-accessibilité . . . 135
E Complétion d'un AFD . . . 136
F Complémentaire d'un AFD . . . 137
G Produit de deux AFD - Automate produit . . . 138

13 Automates finis non déterministes **141**
A Automate fini non déterministe (AFND) . . . 141
B Représentation d'un AFND . . . 142
C Acceptation d'un mot . . . 142
D Déterminisé d'un AFND . . . 143
E ϵ-transitions . . . 146

14 Des expressions régulières aux automates **149**
A Théorème de Kleene . . . 149
B Algorithme de Thompson . . . 150
a Patron de conception d'un cas de base . . . 151
b Patron de conception de l'union . . . 151
c Patron de conception de la concaténation . . . 151
d Patron de conception de l'étoile de Kleene . . . 151
e Application . . . 152
f Élimination des transitions spontanées . . . 153
C Algorithme de Berry-Sethi et automate de Glushkov . . . 155
a Langages locaux . . . 155
b Expressions régulières linéaires . . . 158
c Automate locaux . . . 159
d Automate de Glushkov et algorithme de Berry-Sethi . . . 160
D Comparaison Thompson / Berry-Sethi . . . 163

15 Des automates aux expressions rationnelles **165**
A Automate généralisé . . . 165
B D'un automate généralisé à une expression régulière . . . 166

16 Au-delà des langages réguliers **171**
A Limites des expressions régulières 171
B Caractériser un langage régulier 171
C Les langages des puissances 173

VII Annexes **175**

Bibliographie **177**
Articles 177
Livres 178
Sites web 178

LISTE DES ALGORITHMES

1 Algorithme Quine (SAT) 41
2 Algorithme de recherche par force brute, problème de satisfaction de contraintes 51
3 Algorithme de retour sur trace 53
4 Parcours en largeur d'un graphe 78
5 Parcours en profondeur d'un graphe (version récursive) 81
6 Parcours en profondeur d'un graphe (version itérative) 81
7 Longueur d'une chaîne via un parcours en largeur d'un graphe pondéré 82
8 Algorithme de Dijkstra, plus courts chemins à partir d'un sommet donné 84
9 Algorithme de Bellman-Ford, plus courts chemins à partir d'un sommet donné . 88
10 Algorithme de Floyd-Warshall, plus courts chemins entre toutes les paires de sommet 91
11 A* 93
12 Algorithme de Prim, arbre recouvrant 93
13 Algorithme de Kruskal, arbre recouvrant 94
14 Algorithme de tri topologique 95
15 Recherche d'un couplage de cardinal maximum 101
16 Tri par tas, ascendant 111
17 Algorithme de déterminisation d'un AFND 144

Première partie

Introduction

1

INFORMATION, CONCEPTS ET ENSEMBLES

A Les ensembles et la logique

Comme mentionné dans l'introduction de l'informatique commune, l'informatique est la construction de l'information par le calcul. Mais comment peut-on calculer l'information, au-delà de l'information purement numérique ? Calculer sur les entiers est une chose, calculer un information en général en est une autre.

Le deux pierres angulaires aux fondements de l'informatique sont la théorie des ensembles et la logique. La fin du XIX^{e} siècle a marqué un tournant dans l'histoire des sciences et en particulier pour les mathématiques : la théorie des ensembles et ses paradoxes ont forcé les mathématiciens à mieux formaliser les raisonnements et les démonstrations, c'est-à-dire la logique.

Lorsque l'homme analyse l'information, il fait des regroupements : regrouper les informations d'un même type permet de trouver des propriétés à l'ensemble, de les manipuler par lot, de leur appliquer un même traitement : c'est une abstraction très efficace pour le calcul.

■ **Exemple 1 — Les ingrédients d'une recette de cuisine.** Les ingrédients d'une recette de cuisine forment un concept intéressant : on peut les classer (par texture, goût, couleur), les cuire, les mesurer, les combiner. Et même si un ingrédient est un concept qui n'a pas de réalité [a], les ingrédients forment un ensemble intéressant car on peut le construire : avec de la moutarde, un jaune d'œuf, du sel, du poivre et un mélange adapté, on construit une mayonnaise. La mayonnaise elle-même est un ingrédient : on peut l'utiliser pour élaborer d'autres ingérdients comme une salade de légumes par exemple. Cette salade pourra être incorporée dans un wrap... On en déduit qu'un ingrédient est lui-même un ingrédient, c'est un type récursif ! Ce type d'ensemble est nommé **ensemble inductif** et, si on prend la peine de réfléchir au monde qui nous entoure, il est très présent dans notre réalité.

a. un œuf n'existe pas, mais l'œuf qu'une poule a pondu sous vos yeux existe... De la même manière, une voiture est une abstraction : il n'existe que des modèles construits comme la 206 GTI ou la corolla break 2021 hybride.

B Ensembles inductifs

Les **ensembles inductifs** sont à la base du développement de l'informatique. C'est pourquoi ils irriguent toutes les parties du programme officiel de l'option informatique et donc tous les développement de ce cours. Ils ont été formalisés dans la théorie des types par Russel au début du XX^{e} siècle, ouvrant ainsi la voie aux systèmes formels. Pour ces ensembles inductifs, on dispose de **types de base** et de **constructeurs** qui permettent de créer des types à partir d'autres types. Tout objet d'un ensemble inductif est d'un certain type et il existe un ordre lié à la construction de l'ensemble. C'est ce qui différencie cette approche de la théorie des ensembles. Le programme de l'option informatique aborde des structures de données qui sont des ensembles inductifs en première et deuxième année dont les listes chaînées, les arbres, formules logiques, les expressions régulières.

■ **Exemple 2 — Définir l'ensemble des ingrédients de manière inductive.** Pour nos ingrédients, les termes de base pourraient être le sel, le poivre, l'eau, la farine, le beurre, le sucre, le jaune et le blanc d'œuf. Les règles de construction : prendre, mélanger, étaler, broyer, fondre, cuire.

Formellement, on pourrait définir l'ensemble des ingrédients $\mathcal{I}$ de manière inductive de la façon suivante :

Termes de base $\mathcal{B} = \{\text{sel, poivre, eau, farine, beurre, sucre, jaune d'œuf, blanc d'œuf}\}$

Constructeur (FONDRE) $\forall i \in \mathcal{I}$, FONDRE $(i) \in \mathcal{I}$

Constructeur (MÉLANGER) $\forall i_1, i_2 \in \mathcal{I}$, MÉLANGER $(i_1, i_2) \in \mathcal{I}$

En appliquant le constructeur FONDRE à BEURRE, on obtient FONDRE(BEURRE), du beurre fondu. En appliquant les constructeurs :
MÉLANGER(SUCRE, FONDRE(BEURRE)),
on obtient un nouvel ingrédient qui nous permettra de créer un Kouign-amann.

R Les éléments d'un ensemble inductif sont appelés **termes**.

R Un constructeur possède une certaine **arité** : il permet de construire un terme à partir de un ou plusieurs termes. Dans l'exemple des ingrédients, FONDRE est un constructeur d'arité 1 car il ne prend qu'un seul paramètre. MÉLANGER en prend deux, il est d'arité 2. On ne peut donc pas mélanger du beurre, mais on peut mélanger du beurre et du sucre.

R On observe qu'il y a un **ordre dans les termes** que l'on peut construire à partir de la définition inductive des ingrédients : par exemple, la mayonnaise est un ingrédient nécessaire à l'élaboration d'un sandwich au crabe et le sel est nécessaire à l'élaboration de la mayonnaise. On dit que le sel est un sous-terme de la mayonnaise et que la mayonnaise est un sous-terme du sandwich au crabe. D'une manière générale, $t = \text{CONSTRUCTEUR}(t_1, t_2, \ldots, t_n)$ signifie que l'on construit le terme t à partir des sous-termes $t_1, t_2, \ldots, t_n$. **Cet ordre est appelé ordre structurel.**

C Propriétés des ensembles inductifs

Les ensembles inductifs ont des propriétés qu'il est possible de démontrer grâce à l'**induction structurelle**.

■ **Définition 1 — Principe d'induction structurelle.** Soit $\mathcal{I}$ un ensemble inductif de termes de base $\mathcal{B}$ et de constructeurs $\mathcal{C}$. Soit $\mathcal{P}$ une propriété sur les termes de $\mathcal{I}$.

Si $\mathcal{P}$ est satisfaite pour chaque terme de base de $\mathcal{B}$ et pour chaque constructeur de $\mathcal{I}$, alors $\mathcal{P}$ est satisfaite pour tous les termes de $\mathcal{I}$.

Plus formellement,

$$\left.\begin{array}{ll} \forall b \in \mathcal{B}, & \mathcal{P}(b) \\ \forall c \in \mathcal{C}, \forall t_1, t_2, \dots t_n \in \mathcal{I}, & \mathcal{P}(c(t_1, t_2, \dots, t_n)) \end{array}\right\} \Longrightarrow \forall t \in \mathcal{I}, \mathcal{P}(t) \tag{1.1}$$

■ **Exemple 3 — Comestible.** Soit $\mathcal{I}$ l'ensemble inductif des ingrédients. Soit $\mathcal{P}$ la propriété *est comestible*. On cherche à montrer que tous les ingrédients sont comestibles par induction structurelle [a].

On sait que tous les ingrédients de base sont comestibles. Par ailleurs, faire fondre un ingrédient ne dégrade pas cette propriété, il est toujours comestible une fois fondu. Enfin, lorsqu'on mélange deux ingrédients comestibles, le résultat est comestible. C'est pourquoi, tous les ingrédients sont comestibles.

a. Attention, il ne s'agit que d'une modélisation des ingrédients limitée à des objets de base comestibles. Ne pas extrapoler le résultats ;-)

D Des fonctions pour calculer sur les termes d'un ensemble inductif

On peut facilement définir des fonctions sur les ensembles inductifs en s'appuyant sur leur définition. Cela permet de faire des calcul sur l'information qu'ils représentent.

■ **Exemple 4 — Masse d'un ingrédient.** On peut définir la masse d'un ingrédient en s'appuyant sur la définition inductive d'un ingrédient : la masse d'un ingrédient est la somme des masses des ingrédients qui la compose. Pour les ingrédients de base, on peut considérer que la masse d'un ingrédient $b \in \mathcal{B}$ est une constante connue m_b que l'on a mesurée.

On obtient alors une fonction formulée récursivement :

Cas de base $\forall b \in \mathcal{B}, \text{MASSE}(b) = m_b$

Construction $\forall i \in \mathcal{I}, \text{MASSE}(\text{FONDRE}(i)) = \text{MASSE}(i)$

Construction $\forall i_1, i_2 \in \mathcal{I}, \text{MASSE}(\text{MÉLANGER}(i_1, i_2)) = \text{MASSE}(i_1) + \text{MASSE}(i_2)$

En utilisant cette définition inductive, on peut calculer la masse de tous les ingrédients.

E Pourquoi OCaml ?

Comme on peut le voir ci-dessous et comme on le verra dans la section suivante, le langage OCaml est particulièrement adapté aux ensembles inductifs pour plusieurs raisons :

1. les types en OCaml sont nativement récursifs,
2. les types OCaml peuvent être des types somme `|` ou produit `*`. On les appelle des types algébriques.
3. OCaml procure la syntaxe dite du filtrage de motifs,
4. OCaml permet d'écrire des fonctions récursives.

Combiner ces quatre fonctionnalités permet de traduire directement la plupart des concepts mathématiques exprimés sous la forme d'un ensemble inductif. Les **fonctions** dont les paramètres peuvent être des types algébriques concrétisent les calculs sur les termes des ensembles inductifs.

Ces fonctionnalités de OCaml [1] expliquent en grande partie le choix du langage OCaml en CPGE pour l'option informatique.

Vocabulary 1 — Pattern matching ⇜⇝ Filtrage de motifs

```
type ingredient =
        | Sel | Poivre | Beurre | Sucre | Farine
        | Fondre of ingredient
        | Melanger of ingredient*ingredient;;

let rec masse = function
        | Sel -> 30
        | Poivre -> 10
        | Beurre -> 250
        | Sucre -> 250
        | Farine  -> 10
        | Fondre i ->  masse i
        | Melanger (i1,i2) -> (masse i1) + (masse i2);;

let kouignamann = Melanger(Beurre, Sucre);;
let m = masse(kouignamann);;
```

1. Tous les langages ne dispose pas de ces fonctionnalités : Python ne dispose pas de types récursifs et algébriques et son paradigme est impératif, pas fonctionnel.

2

OCaml, des fonctions et des types

A Pratiquer OCaml

Le plus simple Pour utiliser OCaml, il suffit de l'utiliser en ligne via les bacs à sable des sites OCaml ou TryOcaml.

Pour travailler localement Sur votre machine, le plus simple est d'utiliser l'interprète interactif OCaml. On peut facilement l'installer sur n'importe quel système d'exploitation en suivant ces instructions.

Pour travailler avec un éditeur de texte ăgrâce au mode Tuareg. Les commandes `M-x tuareg-mode` et `M-x run-ocaml` permettent d'activer ce mode. Le résumé (sic!) des commandes est accessible en ligne. Pour les puristes de la ligne de commande, ce mode est également disponible sous Vim.

Utiliser un IDE Enfin, il est possible d'utiliser OCaml avec la plupart des environnement de développement : Eclipse, Visual Studio ou IntelliJ (Jet Brains).

B Vocabulaire utile

■ **Définition 2 — Modèle de calcul.** Un modèle de calcul (MOC) est la description d'une manière de calculer une fonction mathématique étant donnée une entrée. Il existe des modèles de calcul séquentiel [a], fonctionnels [b] et concurrents [c].

a. les automates
b. Le lambda calcul
c. les circuits logiques, les réseaux de Petri ou Synchronous Data Flow (cf. simulinkl)

■ **Définition 3 — Paradigme de programmation.** Un paradigme de programmation est un ensemble de formes et de figures qui constitue un modèle propre à un langage.

■ **Définition 4 — Paradigme impératif.** Le paradigme impératif s'attache à décrire des sé-

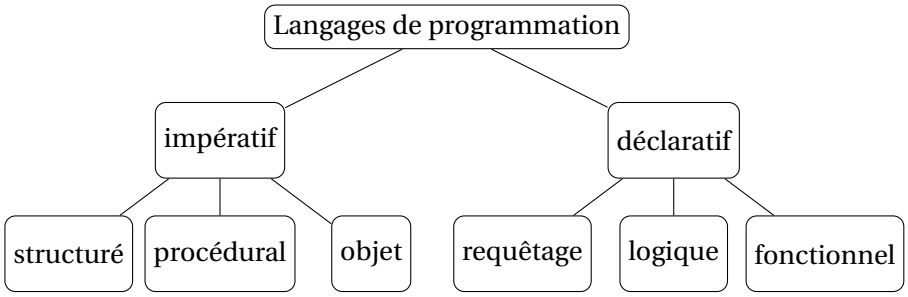


FIGURE 2.1 – Paradigmes des langages de programmation

quences d'instructions (ordres) qui agissent sur un état interne de la machine (contexte). L'impératif explicite le *comment procéder* pour exécuter un programme. Cette programmation se rapproche de la logique électronique des processeurs.

■ **Définition 5 — Paradigme procédural.** Ce paradigme est une déclinaison de l'impératif et propose de regrouper des éléments réutilisables de code dans des routines. Ces routines sont appelées procédures (si elles ne renvoient rien) ou fonctions (si elles renvoient un résultat).

■ **Définition 6 — Paradigme objet.** Ce paradigme est une déclinaison de l'impératif et propose de décrire un programme comme l'interaction entre des objets à définir. Une classe est un type d'objet qui possède des attributs et des comportements. Ces caractéristiques sont encapsulées et peuvent être masquées à l'utilisateur d'un objet : cela permet de protéger l'intégrité de l'objet et de garantir une cohérence dans la manipulation des données.

■ **Définition 7 — Paradigme déclaratif.** Le paradigme déclaratif est une syntaxe qui s'attache à décrire le *quoi*, c'est à dire *ce que le programme doit faire*, non pas comment il doit le faire. Un langage déclaratif ne dépend pas de l'état interne d'une machine (contexte). Cette programmation se rapproche de la logique mathématique et délègue au compilateur la délicate question du *comment procéder*.

■ **Définition 8 — Paradigme fonctionnel.** Le paradigme fonctionnel est une déclinaison du déclaratif qui considère qu'un programme n'est qu'un calcul et qu'un calcul est le résultat d'une fonction. Le mot fonction est ici à prendre au sens mathématique du terme (lambda calcul) : une fonction appelée avec les mêmes paramètres produit le même résultat en toute circonstance.

C Description d'OCaml

Le langage OCaml est un langage qui s'appuie sur le modèle de calcul Categorical Abstract Machine (CAM) qui lui confère une syntaxe fonctionnelle très proche du langage mathéma-

tique. OCaml est un langage interprété et compilé dont le typage est fort et statique. Les paradigmes de programmation d'OCaml sont les paradigmes fonctionnel, impératif et orienté objet.

Les points fots d'OCaml sont :

le typage fort et implicite OCaml est **fortement typé** : toute expression possède un type. Le typage est implicite : l'utilisateur n'a pas à le préciser car le compilateur OCaml utilise un algorithme d'**inférence de type** pour déterminer le type d'une expression. Le typage est statique et vérifié à la compilation : le couplage d'un typage fort et statique permet d'augmenter la performance du code et de rendre plus robuste le code face aux erreurs.

des structures de données muables et immuables OCaml propose des tableaux (Array), des structure de données mutables (tableaux, dictionnaires) mais aussi des structures immuables (listes). De nombreuses bibliothèques sont disponibles : files, tas, arbres…

un Garbage Collector un algorithme de gestion automatisée de la mémoire permet à OCaml de nettoyer les espaces mémoires qui ne sont plus utilisés par le programme en cours d'exécution. C'est important car les structures immuables engendrent en permanence la création et la destruction d'objets en mémoire.

la curryfication des fonctions OCaml permet de manipuler les fonctions comme des objets. Il permet également d'utiliser l'applicaion partielle des fonctions (curryfication).

Dans la suite de ce chapitre, on pourra tester en même temps les expressions et les évaluer sur machine ou en ligne avec le bac à sable OCaml.

D Expressions et inférence de type

En tant que langage fonctionnel (et donc déclaratif), OCaml considère le calcul comme l'évaluation de fonctions mathématiques : OCaml traite donc des expressions qu'il évalue lorsqu'on fait suivre l'expression par `;;`.

Les types simples disponibles en OCaml sont

- `int` les entiers,
- `float` les flottants,
- `string` les chaînes de caractères,
- `bool` les booléens.

Voici un exemple d'évaluation d'un expression :

```
1 >>> 3 + 2*5 ;;
2 - : int = 13
```

Cet exemple montre bien qu'OCaml a inféré le type du résultat : il a écrit `int = 13`, c'est à dire que le résultat de l'évaluation de l'expression est un entier. Il faut noter qu'OCaml ne fait pas de transtypage implicite : pour lui, `3` est un entier et le restera, tout comme `3.5` est un flottant et le restera également. C'est pourquoi les opérateurs `+`,`*` et `/` qui additionne, multiplie ou divise les entiers ne sont pas les mêmes que ceux qui opèrent sur les flottants `+.`, `*.` ou `/.`.

D'ailleurs, l'inférence de type ne s'y trompe pas :

```
>>> 3.0 + 2.0*.5.0 ;;
Error : This expression has type float but an expression was expected of
    type int
```

Dans cet exemple, on a oublié d'utiliser l'opérateur `+.` pour additionner des flottants. Le compilateur s'en aperçoit car pour lui l'opérateur `+` n'accepte que deux opérandes entières et son résultat est un entier. Or, les opérandes fournies sont des flottants. Il détecte donc l'incohérence, la signale et ne peut pas continuer l'exécution.

En OCaml, un chou est un chou et restera un chou. Parole de léonard.

E Expressions locales et globales

Le mot clef `let` permet de définir une variable globale. Cette définition est une déclaration-initialisation. Une variable est toujours initialisée : cela permet au mécanisme d'inférence de type de fonctionner. À droite du symbole `=` doit se trouver une expression (ci-dessous `21 * 2`). À gauche du symbole `=` doit se trouver un identifiant (ci-dessous `x`).

```
>>> let x = 21 * 2;;
val x : int = 42
>>> x / 2;;
- : int = 21
```

Les mots clefs `let ... in` permettent de définir des variables locales. La portée de la variable ainsi définie est l'expression qui suit le mot clef `in`.

```
>>> let recettes = 4000;;
val recettes : int = 4000
>>> let budget = let depenses = 3500 in recettes - depenses;;
val budget : int = 500
>>> depenses * 2;;
Error
: Unbound value depenses
>>> recettes
- : int = 4000
```

OCaml définit une expression conditionnelle à l'aide de l'opérateur ternaire `if t then e1 else e2`. À la différence de son homologue en programmation impérative, cette expression exige que les expressions `e1` et `e2` soient du même type.

```
>>> let recettes = 4000;;
val recettes : int = 4000
>>> let budget = let depenses = 3500 in recettes - depenses;;
val budget : int = 500
>>> let beneficiaire = if budget > 0 then true else false;;
val beneficiaire : bool = true
>>> let result = if beneficiaire then "Excellent !" else "Va falloir aller
    à la banque !";;
val result : string = "Excellent !"
```

Il faut bien noter que le symbole = pour déclarer une expression est utilisé conjointement à `let`. Il ne s'agit pas d'une affectation. D'ailleurs, un dernier exemple nous montre que l'opérateur = utilisé seul est un test d'égalité :

```
>>> let recettes = 4000;;
val recettes : int = 4000
>>> recettes = recettes + 400;;
- : bool = false
```

O En OCaml, les variables sont donc immuables... ce qui est déconcertant au premier abord, car le propre d'une variable n'est-il pas de varier? La réponse est qu'en OCaml, comme en mathématiques, les variables sont en fait des expressions. Si on prend l'équation $y = 2x + 3$, l'idée de modifier la variable x n'existe pas vraiment en tant que telle : x va pouvoir varier dans le sens où l'on peut l'initialiser à différentes valeurs et que l'équation reste valable.

F Fonctions

Une fonction OCaml est une fonction au sens mathématique du terme et une valeur comme une autre. C'est pourquoi OCaml permet de la déclarer via le mot clef `let`.

```
>>> let perimeter r = 2. *. 3.1415926 *. r;;
val perimeter : float -> float = <fun>
>>> perimeter 1.;;
- : float = 6.2831852
```

On observe que le résultat est bien une fonction marquée par le mot-clef `fun`. Lors de l'évaluation de la déclaration de la fonction, OCaml nous délivre la signature associée à la fonction : `perimeter : float -> float = <fun>`. Celle-ci signifie que la fonction `perimeter` prend un paramètre de type `float` et renvoie un paramètre de type `float`.

Si la fonction est récursive, il faut le mentionner pour qu'OCaml en tienne compte :

```
>>> let rec explode n = if n = 0 then "Boum !" else  ". " ^ (explode (n -
    1));;
val explode : int -> string = <fun>
>>> explode 3;;
- : string = ". . . Boum !"
```

Enfin, OCaml permet également de curryfier les fonctions : l'application partielle d'une fonction est une fonction.

```
>>> let ajoute a b = a + b;;
val ajoute : int -> int -> int = <fun>
>>> let ajoute_deux = ajoute 2;;
val ajoute_deux : int -> int = <fun>
>>> ajoute_deux 4;;
- : int = 6
```

G Effets

Vous aurez remarqué que cette introduction à OCaml n'a pas commencé par le traditionnel programme `"Hello Word!"` ! et il y a une bonne raison à cela : les langages fonctionnels n'aiment pas les effets de bords.

Vocabulary 2 — Side effect ⇝ Effet secondaire mal traduit en français par effet de bord.

■ **Définition 9 — Effet (de bord) d'une expression.** Un effet de bord d'une expression est une action de celle-ci qui modifie l'état d'une variable en dehors de l'environnement local à la fonction : l'effet est donc quelque chose d'observable en dehors du fonctionnement standard d'une fonction, c'est-à-dire en dehors de la valeur retournée par la fonction.

■ **Exemple 5 — Effets de bord.** Quelques exemples classiques d'effets de bord :

- la modification d'une variable définie en dehors de la fonction.
- la modification d'une variable muable passée en paramètre à la fonction.
- l'appel d'une fonction qui produit des effets,
- toute entrée/sortie : l'impression d'une chaîne de caractère sur la console d'un écran d'ordinateur, la réception d'un message sur un socket réseau, la lecture ou l'écriture dans un fichier, l'interaction avec un système d'exploitation.

■ **Définition 10 — Expression pure et impure.** On dit qu'une expression est pure si elle n'engendre aucun effet de bord. Dans le cas contraire, on la dit impure.

Dans l'esprit des concepteurs d'OCaml et des langages fonctionnels en général et pour des raisons de cohérence, une fonction doit toujours renvoyer exactement le même résultat si elle est invoquée avec les mêmes paramètres en entrée [1]. Or, si une fonction possède des effets de bords, il est possible que ce ne soit pas le cas.

■ **Exemple 6 — Fonction impure en Python.** Dans l'exemple ci-dessous, la fonction `setn` est appelée deux fois avec le même paramètre 3 mais produit deux résultats différents. C'est donc une fonction impure.

```
1 x = 0
2
3 def setn(n):
4     global x
5     n = n + x
6     return n
7
```

1. Dans le cadre de la programmation des systèmes parallèles (microprocesseurs à architectures multicœurs), cette vision fonctionnelle est très importante car elle permet d'éviter de nombreux problèmes de synchronisation mémoire.

```
n = setn(3)
assert n == 3
x = 4
n = setn(3)
assert n == 3
```

Le programme `hello` ci-dessous est un exemple de fonction impure en OCaml. Écrire un message sur la console, c'est agir sur l'environnement d'exécution en dehors de la fonction. Le risque est, par exemple, que la console ne réponde pas aux ordres du système d'exploitation. Dans ce cas, le programme ne s'exécute pas correctement à cause de l'effet de bord.

```
>>> let hello name = print_string ("Hello " ^ name ^"\n");;
val hello : string -> unit = <fun>
>>> hello "Olivier";;
Hello Olivier
- : unit = ()
```

La signature d'`hello` indique que la fonction renvoie un type `unit`.

■ **Définition 11 — Type unit.** Le type `unit = ()` représente le rien, le vide. Il est utilisé pour bien signifier qu'une fonction ne prend pas de paramètre ou ne renvoie rien.

H Types algébriques

En plus des types simples, OCaml propose des mécanismes pour construire d'autres types éventuellement récursifs à partir des types simples :

- les types algébriques qui sont des composés
  - de types sommes qui sont des alternatives ou des énumérations,
  - et de types produits, des produits cartésiens de types,
- les record, ou enregistrements, qui permettent d'enregistrer une collection de types dans un même objet,
- les type optionnels qui par nature n'existent pas nécessairement.

■ **Définition 12 — Types sommes ou énumérations.** Un type somme est une alternative de types. Il est défini par l'alternance de ses constructeurs.

■ **Exemple 7 — Type somme en OCaml.** Le type utilise l'alternative `|` et des constructeurs qui commencent par une majuscule.

```
>>>     type chess_piece = Pawn | Knight | Bishop | Rook | Queen | King;;
type chess_piece = Pawn | Knight | Bishop | Rook | Queen | King
>>> let p = Pawn;;
val p : chess_piece = Pawn
```

■ **Définition 13 — Type produit.** Un type produit est un produit cartésien de types. Il engendre des tuples. Généralement, on ne nomme pas un type produit.

■ **Exemple 8 — Type produit en OCaml.** Le produit de type est réalisé par l'opérateur $*$.

```
>>> type point3d = float * float * float;;
type point3d = float * float * float
>>> let zero = 0.0, 0.0, 0.0;;
val zero : float * float * float = (0., 0., 0.)
```

■ **Définition 14 — Type enregistrement.** Un enregistrement est une collection de types nommés et enregistrés dans une même structure.

■ **Exemple 9 — Type enregistrement en OCaml.** Dans le domaine des services réseaux, on a souvent besoin d'identifier un service par son socket réseau qui est la combinaison de son adresse IP ou nom d'hôte, d'un numéro de port et d'un protocole. Cela peut se faire via un type enregistrement. Attention à la syntaxe, aux points virgules, deux points et symbole égal.

```
>>> type service_info =
.. {    service_name : string;
    ..      port         : int;
    ..      protocol     : string;
    .. };;
type service_info = { service_name : string; port : int; protocol :
    string; }
>>> let http_service = {service_name = "www"; port = 80; protocol = "http
    " };;
val http_service : service_info =
{service_name = "www"; port = 80; protocol = "http"}
>>> http_service.port;;
- : int = 80
```

■ **Définition 15 — Type optionnel.** Un type optionnel est un type qui peut être typé et posséder une valeur ou ne pas être typé ni posséder de valeur.

■ **Exemple 10 — Type option en OCaml.** En OCaml, les mots-clefs pour le type `option` sont `None` et `Some`.

```
>>> type zip_code = None | Some of int;;
type zip_code = None | Some of int
>>> let brest_code = Some 29200;;
val brest_code : zip_code = Some 29200
>>> let lost_city_code = None;;
val lost_city_code : zip_code = None
```

■ **Définition 16 — Type algébrique.** Un type algébrique est un type éventuellement récursif qui est une alternative de types éventuellement produit.

■ **Exemple 11 — Type algébriques.** Dans cet exemple, on construit un jeu de carte. Une première fonction permet d'obtenir la liste des figures d'une couleur donnée. Une autre la liste des cartes numéros pour une couleur donnée (à la belote!). Le mot-clef `of` permet de préciser un type de donnée associé au constructeur.

```
>>> type couleur = Coeur | Carreaux | Pique | Trefle;;
type couleur = Coeur | Carreau | Pique | Trefle
>>> type carte = As of couleur | Roi of couleur | Dame of couleur |
    Valet of couleur | Numero of int * couleur;;
type carte =
As of couleur
| Roi of couleur
| Dame of couleur
| Valet of couleur
| Numero of int * couleur
>>> let figures_de c = [As c; Roi c; Dame c; Valet c];;
val figures_de : couleur -> carte list = <fun>
>>> figures_de Pique;;
- : carte list = [As Pique; Roi Pique; Dame Pique; Valet Pique]
>>> let numero_de c = [10,c;9,c;8,c;7,c];;
val numero_de : 'a -> (int * 'a) list = <fun>
>>> numero_de Coeur;;
- : (int * couleur) list = [(10, Coeur); (9, Coeur); (8, Coeur); (7,
    Coeur)]
```

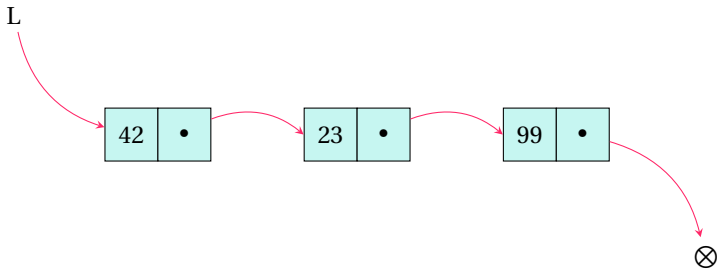


FIGURE 2.2 – Représentation d'une liste d'entiers L. L pointe sur le premier élément de la liste. Le dernier pointeur ne pointe sur rien.

I Listes

Le type liste OCaml est un type récursif immuable.

R **Immuable** signifie qu'on ne peut pas le modifier. Pour faire évoluer une liste, c'est à dire pour ajouter ou retirer des éléments, il est donc nécessaire de créer une autre liste.

O Une liste en OCaml ne contient qu'un seul type de données.

■ **Définition 17 — Définition inductive des listes.** Une liste est soit une liste vide soit un élément suivi d'une liste.

L'ensemble des listes $\mathcal{L}$ à valeur dans $\mathcal{E}$ est donc définie par :

Base la liste vide `[]` est une liste

Constructeur : : $\forall L \in \mathcal{L}, \forall e \in \mathcal{E}, e :: l \in \mathcal{L}$

Ce qui en OCaml donne :

```
1 type 'a list =
2     | []
3     | (::) of 'a * 'a list
```

Un élément d'une liste est une cellule constituée de deux parties :

- la première contient une donnée, par exemple un entier pour une liste d'entiers,
- la seconde contient le reste de la liste.

O Quelques remarques sur les opérations sur les listes en OCaml :

- pour supprimer un élément d'une liste, il faut en construire une autre.
- pour ajouter un élément à une liste, il faut en construire une autre.

Opération	Complexité	Raison
Accès à un élément au début	$O(1)$	L pointe sur le premier élément
Accès à un élément à la fin	$O(n)$	accès séquentiel
Accès à un élément au milieu	$O(n)$	accès séquentiel
Ajout d'un élément au début	$O(1)$	L pointe sur le premier élément
Ajout d'un élément à la fin	$O(n)$	accès séquentiel
Suppression d'un élément au début	$O(1)$	L pointe sur le premier élément
Suppression d'un élément à la fin	$O(n)$	accès séquentiel

TABLE 2.1 – Complexité des opérations associées à l'utilisation d'une liste simplement chaînée.

Ces opérations ont donc un coût comme l'illustre le tableau 2.1.

■ **Exemple 12 — Liste en OCaml.** On peut construire une liste OCaml directement en initialisant les valeurs des éléments :

```
>>> let l = [1;2;3;4;5];;
val l : int list = [1; 2; 3; 4; 5]
```

Le constructeur `::` permet d'ajouter un élément en tête de liste. Par exemple :

```
>>> let l = 42::21::14::7::l;;
val l : int list = [42; 21; 14; 7; 1; 2; 3; 4; 5]
>>> let wrong = 0.2::l;;
Line 1, characters 17-18:
|1 | let wrong = 0.2::l;;
                      ^
Error: This expression has type int list but an expression was expected
     of type float list. Type int is not compatible with type float
```

Le type d'objet contenu dans une liste est nécessairement toujours le même : dans l'exemple ci-dessus, le type de la liste est `int list`, une liste d'entiers.

On ne peut accéder directement [a] qu'au premier élément. L'accès aux autres éléments nécessite de balayer la liste. Généralement, comme c'est une structure inductive, on procède par déconstruction.

```
>>>let head = List.hd l;;
val head : int = 42
```

a. en temps constant O(1)

J Filtrage de motif

Le filtrage de motif est une technique pour gérer les types algébriques et les ensembles inductifs.

Vocabulary 3 — Pattern matching ⇜⇝ Filtrage de motif

■ **Définition 18 — Filtrage de motif.** Le filtrage de motif est l'action de tester une expression pour détecter sa constitution dans le but de faire un calcule en fonction du motif détecté.

■ **Exemple 13 — Filtrage de motif en OCaml.** Le code ci-dessous définit un type algébrique `shape` puis une fonction qui calcule l'aire en fonction du type `shape` passé en paramètre. Il faut noter que l'aire du triangle est calculée avec la formule de Héron.

```
1 type shape = Circle of float | Square of float | Triangle of (float *
      float * float);;
2
3 let area s =
4 match s with
5     | Circle r -> Float.pi *. (r ** 2.0)
6     | Square s -> s ** 2.0
7     | Triangle (a, b, c) -> let s = (a +. b +. c) /. 2.0 in sqrt (s *. (s
          -. a) *. (s -. b) *. (s -. c));;
8
9 area (Triangle (3.0, 4.0, 5.0));;
```

■ **Exemple 14 — Filtrage de motif et liste en OCaml.** La fonction ci-dessous est récursive et calcule la somme des éléments d'une liste. Elle déconstruit au fur et à mesure la liste pour additionner ses éléments.

```
1 let rec sum l =
2     match l with
3         | [] -> 0
4         | head :: tail -> head + sum tail;;
5 let l = 42::21::14::7::l;;
6 sum l;;
```

Le motif `[]` représente une liste vide. Si ce motif est détecté, alors la fonction renvoie 0, car la somme des éléments d'une liste vide vaut 0.

Sinon, il y a au moins un élément en tête de liste `head` suivi `(::)` par une sous-liste `tail` éventuellement vide. Alors on renvoie la valeur `head` ajoutée à la somme du reste de la liste.

Dans cette exemple emblématique, la liste `l` n'est pas détruite en mémoire, mais on la déconstruit à chaque appel récursif en créant une autre liste `t` qui est une partie de `l`. L'opération est de complexité linéaire.

O Les objets immuables comme les listes nécessitent donc un espace mémoire adapté : il faut être capable de créer des listes à volonté. Le Garbage Collector, en cours d'exécution du programme, collecte les objets (comme les listes) non utilisés puis libère l'espace mémoire associé. Il garantit ainsi un exécution correcte, sans encombrement mémoire dû à la création multiple d'objets par les fonctions et les constructeurs.

K Références et programmation impérative en OCaml

OCaml est un langage mutliparadigme et on peut programmer de manière impérative avec effet. On peut regretter une certaine lourdeur de la syntaxe par rapport à Python mais les exigences des deux langages n'ont rien à voir non plus : Python est un langage laxiste [2], OCaml ne l'est pas [3].

O Le mécanisme pour pouvoir modifier une variable est nommé **référence**. La référence en OCaml est ce qui permet de manipuler une variable comme on l'entend dans les langages impératifs, c'est à dire une variable **muable**.

Pour référencer une variable, il suffit de le déclarer avec le mot-clef `ref` suivi de la valeur d'initialisation. Pour utiliser la valeur d'une référence, il faut la précéder d'un point d'exclamation !. Enfin, pour affecter une nouvelle valeur à une variable, il faut utiliser l'opérateur `:=`.

■ **Exemple 15 — Déclaration, initialisation et utilisation d'une référence en OCaml.** Lorsqu'on affecte une nouvelle valeur à une référence en OCaml, l'opération renvoie `unit`. **Une affectation est un effet de bord**.

```
>>> let x = ref 4;;
val x : int ref = {contents = 4}
>>> let y = 3 + !x;
val y : int = 7
>>> x := 0;;
- : unit = ()
>>> x
- : int ref = {contents = 0}
>>> !x
-: int = 0
>>> y
- : int = 7
```

OCaml propose également les structures itératives `for` et `while`. Les tableaux (Array) sont des structures impératives, tout comme les dictionnaires.

■ **Exemple 16 — Boucle et tableau en OCaml.** Dans cet exemple, un tableau initialisé à zéro est créé. Puis ses valeurs sont modifiées en utilisant une boucle `for`. On notera que l'accès à un élément su tableau se fait par `.(i)` et l'affectation de la nouvelle valeur par `<-`.

```
>>> a
- : int array = [|0; 1; 2; 3; 4|]
>>> let a = Array.make 5 0;
val a : int array = [|0; 0; 0; 0; 0|]
>>> for i = 0 to (Array.length a - 1) do a.(i) <- a.(i) + i done;;
- : unit = ()
```

2. ce qui est pratique pour le prototypage d'applications et le calcul numérique
3. ce qui est important pour le développement d'applications fiables, robustes et performantes.

```
>>> a
- : int array = [|0; 1; 2; 3; 4|]
```

O En OCaml, chaque type possède ses propres opérateurs et fonctions. Le compilateur peut générer facilement des messages compréhensibles. L'approche est rigoureuse.

Deuxième partie

Logique

3

DE LA LOGIQUE AVANT TOUTE CHOSE

À la fin de ce chapitre, je sais :

- ☞ formuler des propositions logiques à partir du langage naturel
- ☞ utiliser les connecteurs logiques pour relier des variables
- ☞ établir une table de vérité
- ☞ utiliser les lois de Morgan, le tiers exclu et la décomposition de l'implication
- ☞ mettre une formule propositionnelle sous une forme normale
- ☞ étudier la satisfaisabilité d'une formule propositionnelle

A Logique et applications

Quelque soit le sujet, les êtres humains ont tendance à chercher s'assurer qu'ils ont raison ou que les autres ont tord, ne serait-ce que pour des raisons d'ego [1] ou de commerce [2]. Dans le cadre de la science et de l'ingénierie, il est fondamental de pouvoir apporter la preuve que le raisonnement tenu est correct, car c'est ainsi que la science progresse collectivement et ainsi que l'ingénierie garantit le bon fonctionnement de l'objet produit. Aujourd'hui,tous les domaines de l'industrie sont dépendants de la logique. On peut citer par exemple : la planification (logistique, organisation des tâches), la satisfaction de contraintes multiples, l'élaboration de diagnostiques, la vérification formelle de modèles de systèmes complexes ou l'élaboration et vérification des circuits électroniques.

Une démarche scientifique exige un raisonnement formalisé : c'est l'objet de la logique et du calcul propositionnel qui est présenté dans ce chapitre. Ce formalisme est né au milieu du XIX^{e} siècle grâce au travaux de Boole et a été finalisé au cours de la première moitié du XX^{e} siècle par Frege, Russel et Gödel.

1. D'abord, j'ai toujours raison!
2. Mon collègue est-il en train de m'arnaquer?

■ **Définition 19 — Logique.** Du grec *logos*, la raison. La logique en tant que discipline scientifique fournit les outils nécessaires à la construction d'un raisonnement : elle permet de manipuler des concepts et d'enchaîner les déductions. Les vérités logiques ne concernent aucun domaine de connaissance en particulier : elles sont valides en amont de toute vérité scientifique particulière empirique [a].

a. c'est-à-dire issue de l'expérience du monde réel

■ **Exemple 17 — Structure d'un raisonnement.** Considérons l'énoncé suivant :

> Si un professeur est un super-héros et que je suis un professeur, alors je suis un super-héros.

Cet énoncé est vrai et inspire une observation fondamentale : la notion de vérité que l'on peut lui associer ne repose que sur sa structure, sa forme, c'est-à-dire Si ... et ..., alors .... On pourrait en effet remplacer le terme *professeur* par *élève*, le terme *super-héros* par *star* et le terme *je* par *tu* et malgré tout, l'énoncé serait vrai.

> Si un élève est une star et que tu es un élève, alors tu es une star.

On peut donc **construire** des énoncés logiques formels à l'aide de connecteurs : Si, et, ou, alors, ne ... pas,donc.... Par exemple, on peut combiner :

p_1 le ciel est bleu

et

p_2 je me promène au bord de l'océan

ainsi :

p_3 Si le ciel est bleu, alors je me promène au bord de l'océan.

Formellement, on peut modéliser cet énoncé logique ainsi : $p_3 = (p_1 \longrightarrow p_2)$.

■ **Définition 20 — Proposition simple ou atomique.** Une proposition simple est une expression vraie ou fausse.

La logique des propositions est limitée : elle ne peut pas prendre en compte des énoncés quantifiés [3] comme dans l'exemple 18.

■ **Exemple 18 — Prédicat et raisonnement quantifié** ⇢ **Hors Programme** . Considérons l'énoncé suivant :

> Si tous les professeurs sont des super-héros et que je suis un professeur, alors je suis un super-héros.

La logique des propositions ne peut pas modéliser cet énoncé à cause du quantificateur universel *tous les*. En effet, la valeur de vérité de cette proposition dépend de ce quantificateur. Que se passe-t-il si un professeur n'est pas un super-héros ? Ce n'est plus la même

3. On parle alors de logique du premier ordre, logique des prédicats ⇢ HORS PROGRAMME

proposition. Une proposition simple ne contient donc pas de quantificateur. Pour modéliser ce type d'énoncé, on utilise les prédicats, ceux que vous manipulez en mathématiques : supposons que S est le prédicat unaire *super-héros*, P le prédicat unaire *professeur*. On peut écrire le prédicat suivant :

$$(\forall x, P(x) \longrightarrow S(x)) \wedge P(x) \longrightarrow S(x)$$

La suite de ce chapitre expose donc la logique des propositions. Elle débute par la définition des briques de bases que sont les constantes, les variables propositionnelles et les opérateurs logiques. Puis elle énonce la syntaxe de la logique propositionnelle en donnant la définition inductive des formules logiques. La définition de la valuation d'une formule permet de donner un sens à la logique propositionnelle et de définir une sémantique. Enfin, le chapitre aborde les problèmes de satisfaisabilité d'une formule logique.

B Constantes, variables propositionnelles et opérateurs logiques

Dans cette section est détaillé les éléments de l'ensemble qui forme la base des formules logiques.

■ **Définition 21 — Constante universelle $\top$.** La constante $\top$ désigne le vrai. On peut la voir comme un opérateur d'arité nulle.

■ **Définition 22 — Constante vide $\bot$.** La constante $\bot$ désigne le faux, l'absurde. On peut la voir comme un opérateur d'arité nulle.

Vocabulary 4 — Top et Bottom ⇜⇝ En anglais, la constante universelle $\top$ se dit *Top* et la constante vide $\bot$ Bottom.

■ **Définition 23 — Variable propositionnelle.** Une variable propositionnelle est une proposition atomique (cf. définition 20).

C Opérateurs logique et notations

L'étude des structures des propositions permet de définir opérateurs qui relient les formules logiques : la négation *non*, le *et*, le *ou*, l'*implication* et l'*équivalence*.

■ **Définition 24 — Opérateur.** Un opérateur est une fonction qui réalise une opération primitive dans un langage.

■ **Exemple 19 — Opérateurs.** Les symboles mathématiques $+, -, \times$ sont des opérateurs pour l'arithmétique des entiers naturels : $2+3\times5$. On utilise également ces symboles dans le cas de l'arithmétique des polynômes : $P+Q$. Cependant, il faut bien observer que ce ne sont pas les mêmes opérateurs.

De la même manière, en informatique, les symboles `+`,`-`,`*` représentent les opérateurs arithmétiques sur les entiers en Python. Même si on utilise les mêmes symboles pour faire les opérations de base sur les flottants, il faut bien remarquer que le langage offre ici ce qu'on appelle du sucre syntaxique, une facilité. Car ce sont en fait des opérateurs différents. D'ailleurs, en OCaml, les opérateurs arithmétiques sur les entiers `+`,`-`,`*` ne sont pas les opérateurs sur les flottants `+.`, `-.`, `*.`.

■ **Définition 25 — Arité d'un opérateur.** L'arité d'un opérateur est le nombre de paramètres que prend sa fonction sous-jacente. On distingue les opérateurs unaires (une seule opérande) et les opérateurs binaires (deux opérandes).

■ **Exemple 20 — Arités des opérateurs arithmétiques.** On dit que l'addition ou la multiplication sont des opérateurs binaires car ils prennent deux opérandes en entrée.

En informatique, on distingue l'opérateur binaire - de l'opérateur unaire -. Le premier réalise la soustraction des deux opérandes `a - b`, le second change le signe de l'opérande `-a`.

■ **Définition 26 — Notation infixe des opérateurs (connecteurs).** Dans le cadre d'une notation infixe des opérateurs binaires, l'opérateur est placé entre ses opérandes.

■ **Définition 27 — Notation préfixe des opérateurs (constructeurs).** Dans le cadre d'une notation préfixe des opérateurs binaires, l'opérateur est placé devant ses opérandes.

■ **Définition 28 — Négation (NON).** L'opérateur négation consiste à calculer la valeur opposée de son opérande.

- Connecteur : $\neg$
- Constructeur : `not`
- Notation informatique fréquente : not ou !

■ **Définition 29 — Conjonction (ET).** L'opérateur conjonction a pour résultat vrai si ses deux opérandes sont vraies.

- Connecteur : $\wedge$
- Constructeur : `and`
- Notation informatique fréquente : `&&` ou `and`

■ **Définition 30 — Disjonction logique (OU).** L'opérateur disjonction a pour résultat vrai si une des deux opérandes est vraies.

- Connecteur : $\vee$
- Constructeur : `or`
- Notation informatique fréquente : `||` ou `or`

R Les opérateurs négation, conjonction et disjonction sont les opérateurs premiers : ils permettent d'exprimer tous les autres.

■ **Définition 31 — Implication matérielle ($\Longrightarrow$).** L'opérateur implication logique a pour résultat faux seulement si la seconde opérande est fausse.

- Connecteur : $\Longrightarrow$
- Constructeur : `imp`

R L'implication matérielle est le seul opérateur de base en logique minimale.

■ **Définition 32 — Équivalence matérielle ($\Longleftrightarrow$).** L'opérateur équivalence matérielle est équivalent à une implication dans les deux sens.

- Connecteur : $\Longleftrightarrow$
- Constructeur : `eq`

■ **Exemple 21 — Notation infixe et préfixe d'une même formule logique.** Voici une même formule logique décrite à l'aide de variables propositionnelles et d'opérateurs infixes ou préfixes.

- avec les connecteurs : $(c \Longrightarrow b) \vee (\neg c \wedge b)$
- avec les constructeurs : or(imp(c,b),et(not(c), b))

Même s'il est possible de programmer des connecteurs, les informaticiens préfèrent les constructeurs pour plusieurs raisons :

- ils sont plus faciles à implémenter,
- ils donnent à l'ensemble des formules une structure arborescente facile à analyser.

R Les opérateurs possèdent un ordre de priorité, de la plus forte à la plus faible :

$$\neg > \wedge > \vee$$

Cela signifie qu'en cas d'ambiguïté sur une formule sans parenthèses, cet ordre permet trancher l'interprétation : on effectue d'abord la négation puis la conjonction et enfin la disjonction.

D Formules logiques : définition inductive et syntaxe

On considère un ensemble des variables propositionnelles $\mathcal{V}$ utilisé pour écrire un ensemble de formules $\mathcal{F}$ en logique propositionnelle.

■ **Définition 33 — Ensemble des formules propositionnelles $\mathcal{F}$ (défini inductivement).** L'ensemble $\mathcal{F}$ des formules propositionnelles sur $\mathcal{V}$ est défini inductivement comme suit :

$$\bot \in \mathcal{F} \quad \text{(Base)} \tag{3.1}$$
$$\top \in \mathcal{F} \quad \text{(Base)} \tag{3.2}$$
$$\forall v \in \mathcal{V}, v \in \mathcal{F} \quad \text{(Base)} \tag{3.3}$$
$$\forall \phi \in \mathcal{F}, \text{not}(\phi) \in \mathcal{F} \quad \text{(Constructeur négation)} \tag{3.4}$$
$$\forall \phi \in \mathcal{F}, \forall \psi \in \mathcal{F}, \text{and}(\phi, \psi) \in \mathcal{F} \quad \text{(Constructeur conjonction)} \tag{3.5}$$
$$\forall \phi \in \mathcal{F}, \forall \psi \in \mathcal{F}, \text{or}(\phi, \psi) \in \mathcal{F} \quad \text{(Constructeur disjonction)} \tag{3.6}$$

Cela signifie qu'une formule logique est soit :

- une constante universelle ou vide,
- une variable propositionnelle,
- une négation d'une formule logique,
- une conjonction ou une disjonction de formules logiques.

R L'intérêt principal de la définition inductive est qu'elle permet de **construire des formules logiques qui sont des objets informatiques avec lesquels on peut calculer**.

■ **Définition 34 — Formule atomique.** Une formule ϕ est atomique si ϕ est $\bot$, $\top$ ou une variable propositionnelle.

R Une formule logique de $\mathcal{F}$ peut être représentée par un arbre comme l'illustre la figure 3.1 : cette forme est très importante pour les compilateurs et les outils d'analyse de code en général. On la désigne par le terme arbre syntaxique. En parcourant un tel arbre, on peut calculer une formule logique. C'est pourquoi, la structure d'arbre est au programme et sera détaillée dans les prochains chapitres.

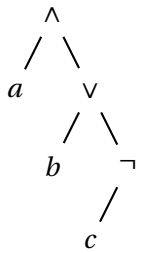


FIGURE 3.1 – Arbre représentant la formule logique $a \wedge (b \vee \neg c)$

■ **Définition 35 — Ensemble des sous-formules.** Soit ϕ une formule logique et $\mathcal{V}$ l'ensemble des ses variables propositionnelles. On définit l'ensemble des sous-formules de ϕ par la fonction sf :

$$\forall \phi \in \{\bot, \top, \} \cup \mathcal{V}, sf(\phi) = \phi \quad \text{(Base)} \tag{3.7}$$

$$\forall \phi \in \mathcal{F}, \exists \psi \in \mathcal{F}, \phi = \neg\psi \Longrightarrow sf(\phi) = \{\phi\} \cup sf(\psi) \quad \text{(Constructeur négation)} \tag{3.8}$$

$$\forall \phi \in \mathcal{F}, \exists \psi, \xi \in \mathcal{F}, \phi = \psi \wedge \xi \Longrightarrow sf(\phi) = \{\phi\} \cup sf(\psi) \cup sf(\xi) \quad \text{(Constructeur conjonction)} \tag{3.9}$$

$$\forall \phi \in \mathcal{F}, \exists \psi, \xi \in \mathcal{F}, \phi = \psi \vee \xi \Longrightarrow sf(\phi) = \{\phi\} \cup sf(\psi) \cup sf(\xi) \quad \text{(Constructeur disjonction)} \tag{3.10}$$

Une constante ou une variable propositionnelle est une sous-formule. Toutes les formules qui permettent de construire une formule logique sont des sous-formules.

■ **Définition 36 — Taille d'une formule.** La taille d'une formule logique est le nombre d'opérateurs qui construisent la formule. On la note $|\phi|$ et on la définit inductivement par :

$$|\bot| = 0 \quad \text{(Base)} \tag{3.11}$$

$$|\top| = 0 \quad \text{(Base)} \tag{3.12}$$

$$\forall x \in \mathcal{V}, |x| = 0 \quad \text{(Base)} \tag{3.13}$$

$$|\neg\phi| = 1 + |\phi| \quad \text{(Constructeur négation)} \tag{3.14}$$

$$|\phi \diamond \psi| = 1 + |\phi| + |\psi| \quad \text{(Constructeur conjonction ou disjonction } \diamond) \tag{3.15}$$

■ **Définition 37 — Hauteur d'une formule.** La hauteur d'une formule logique est la hauteur de son arbre syntaxique. On la note $h(\phi)$ et elle est définie inductivement :

$$h(\bot) = 0 \quad \text{(Base)} \tag{3.16}$$

$$h(\top) = 0 \quad \text{(Base)} \tag{3.17}$$

$$\forall x \in \mathcal{V}, h(x) = 0 \quad \text{(Base)} \tag{3.18}$$

$$h(\neg\phi) = 1 + h(\phi) \quad \text{(Constructeur négation)} \tag{3.19}$$

$$h(\phi \diamond \psi) = 1 + \max(h(\phi, h(\psi)) \quad \text{(Constructeur conjonction ou disjonction } \diamond) \tag{3.20}$$

E Sémantique et valuation des formules logiques

■ **Définition 38 — Ensemble des valeurs de vérité.** L'ensemble des valeurs de vérité est un ensemble à deux éléments que l'on peut noter de différentes manières :

$$\mathbb{B} = \{0,1\} = \{V,F\} = \{\text{Faux},\text{Vrai}\} \tag{3.21}$$

■ **Définition 39 — Valuation ou interprétation.** Une valuation de $\mathcal{V}$ est une distribution des valeurs de vérité sur l'ensemble des variables propositionnelles $\mathcal{V}$, soit une fonction ν : $\mathcal{V} \longrightarrow \mathbb{B}$.

■ **Définition 40 — Évaluation d'une formule logique (définie inductivement).** Soit ν une valuation de $\mathcal{V}$. L'évaluation d'une formule logique d'après ν est notée $[\![\phi]\!]_\nu$. Elle est définie inductivement par :

$$[\![\bot]\!]_\nu = \text{F} \qquad \text{(Base)} \tag{3.22}$$
$$[\![\top]\!]_\nu = \text{V} \qquad \text{(Base)} \tag{3.23}$$
$$\forall x \in \mathcal{V}, [\![x]\!]_\nu = \nu(x) \qquad \text{(Base)} \tag{3.24}$$
$$\forall \phi \in \mathcal{F}, [\![\neg\phi]\!]_\nu = \neg[\![\phi]\!]_\nu \qquad \text{(Constructeur négation)} \tag{3.25}$$
$$\forall \phi \in \mathcal{F}, \forall \psi \in \mathcal{F}, [\![\phi \wedge \psi]\!]_\nu = [\![\phi]\!]_\nu \wedge [\![\psi]\!]_\nu \qquad \text{(Constructeur conjonction)} \tag{3.26}$$
$$\forall \phi \in \mathcal{F}, \forall \psi \in \mathcal{F}, [\![\phi \vee \psi]\!]_\nu = [\![\phi]\!]_\nu \vee [\![\psi]\!]_\nu \qquad \text{(Constructeur disjonction)} \tag{3.27}$$
$$\tag{3.28}$$

Pour être capable d'évaluer une formule logique, il faut donc pouvoir évaluer des négations, des conjonctions et des disjonctions. Les tables de vérités des opérateurs premiers sont données sur les tableaux 3.1, 3.2, 3.3, 3.4 et 3.5.

a	$\neg a$
T	F
F	T

TABLE 3.1 – Table de vérité de l'opérateur négation

a	b	$a \wedge b$
T	T	T
T	F	F
F	T	F
F	F	F

TABLE 3.2 – Table de vérité de l'opérateur conjonction

a	b	$a \vee b$
T	T	F
T	F	T
F	T	T
F	F	T

TABLE 3.3 – Table de vérité de l'opérateur disjonction

a	b	$a \Longrightarrow b$
T	T	T
T	F	F
F	T	T
F	F	T

TABLE 3.4 – Table de vérité de l'opérateur implication matérielle

a	b	$a \Longleftrightarrow b$
T	T	T
T	F	F
F	T	F
F	F	T

TABLE 3.5 – Table de vérité de l'opérateur équivalence matérielle

■ **Définition 41 — Modèle.** Un modèle pour une formule logique ϕ est une valuation v telle que :

$$[\![\phi]\!]_v = \mathrm{V} \tag{3.29}$$

■ **Définition 42 — Conséquence sémantique $\models$.** Soit ϕ et ψ deux formules de $\mathcal{F}$. On dit que ψ est une conséquence sémantique de ϕ si tout modèle de ϕ est un modèle de ψ. On note alors : $\phi \models \phi$

Une formule ψ peut également être la conséquence sémantique d'un ensemble de formules Γ. On note alors : $\Gamma \models \psi$.

■ **Définition 43 — Équivalence sémantique $\equiv$.** Deux formules logiques ϕ et ψ sont équivalentes sémantiquement si quelle que soit la valuation choisie, l'évaluation des deux formules produit le même résultat. On note cette équivalence $\phi \equiv \psi$.

Plus formellement,

$$\phi \equiv \psi \Longleftrightarrow \forall v : \mathcal{V} \longrightarrow \mathbb{B}, [\![\phi]\!]_v = [\![\psi]\!]_v \tag{3.30}$$

■ **Définition 44 — Tautologie.** Une formule ϕ toujours vraie quel que soit le modèle d'interprétation est une tautologie. On la note $\top$ ou $\models \phi$.

R Deux formules sont donc équivalentes d'un point de vue sémantique si et seulement si leur équivalence matérielle est une tautologie : $\models (\phi \Longleftrightarrow \psi)$.

■ **Définition 45 — Antilogie.** Une formule a toujours fausse quel que soit le modèle d'interprétation est une antilogie. On la note $\bot$.

■ **Définition 46 — Formule satisfaisable.** Soit ϕ une formule logique de $\mathcal{F}$. S'il existe une valuation v de $\mathcal{V}$ qui satisfait ϕ, alors ϕ est dite satisfaisable.

Plus formellement :

$$\phi \text{ est une formule satisfaisable} \Longleftrightarrow \exists v : \mathcal{V} \longrightarrow \mathbb{B}, [\![\phi]\!]_v = \top \tag{3.31}$$

F Lois de la logique

Les lois de la logique sont des équivalences sémantiques.

a Éléments neutres et absorbants, idempotence

$$a \wedge \top \equiv a \tag{3.32}$$
$$a \wedge \bot \equiv \bot \tag{3.33}$$
$$a \vee \top \equiv \top \tag{3.34}$$
$$a \vee \bot \equiv a \tag{3.35}$$
$$a \wedge a \equiv a \tag{3.36}$$
$$a \vee a \equiv a \tag{3.37}$$

b Commutativité, distributivité, associativité

$$a \wedge b \equiv b \wedge a \tag{3.38}$$
$$a \vee b \equiv b \vee a \tag{3.39}$$
$$a \wedge (b \vee c) \equiv (a \wedge b) \vee (a \wedge c) \tag{3.40}$$
$$a \vee (b \wedge c) \equiv (a \vee b) \wedge (a \vee c) \tag{3.41}$$
$$a \wedge (b \wedge c) \equiv (a \wedge b) \wedge c \tag{3.42}$$
$$a \vee (b \vee c) \equiv (a \vee b) \vee c \tag{3.43}$$

c Lois de De Morgan

$$\neg(a \wedge b) \equiv \neg a \vee \neg b \tag{3.44}$$
$$\neg(a \vee b) \equiv \neg a \wedge \neg b \tag{3.45}$$

d Décomposition des opérateurs

$a \Longrightarrow b \equiv \neg a \vee b$	Implication et opérateurs premiers	(3.46)
$a \Longrightarrow b \equiv \neg b \Longrightarrow \neg a$	Contraposition	(3.47)
$(a \wedge b) \Longrightarrow c \equiv a \Longrightarrow (b \Longrightarrow c)$	Curryfication	(3.48)

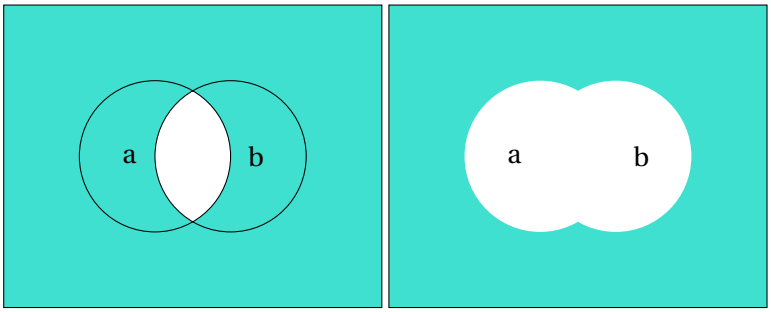


FIGURE 3.2 – Illustration des lois de De Morgan

e Démonstrations

Pour démontrer les lois précédentes, on peut produire les tables de vérités correspondantes puis utiliser les lois déjà prouvées pour démontrer les autres. Ces démonstrations constituent d'excellents exercices.

G Principes et logique classique

$a \vee \neg a \equiv \top$	Principe du tiers-exclus	(3.49)
$a \wedge \neg a \equiv \bot$	Principe de non-contradiction	(3.50)
$\bot \Longrightarrow a$	Principe d'explosion	(3.51)
$\neg\neg a \equiv a$	Double négation	(3.52)

R Il existe plusieurs logiques qui se différencient principalement par la manière de gérer les déductions de l'absurde :

- La logique minimale n'utilise qu'un seul connecteur : l'implication. Elle a pour caractéristique de ne rien déduire de $\bot$. Cela signifie qu'elle n'inclut ni le principe du tiers-exclus, ni le principe d'explosion.
- La logique classique utilise les opérateurs définis dans ce cours. Elle a pour caractéristique d'inclure les principes ci-dessus et permet donc de conduire des raisonnements par l'absurde : $(\neg\phi \Longrightarrow \bot) \Longrightarrow \phi$. La critique faite à ce type de raisonnement, c'est qu'il permet d'accepter l'existence d'un concept sans pouvoir le construire explicitement.
- La logique intuitionniste est constructive : la notion de preuve constructive remplace la notion de vérité. Constuire un concept, c'est exhiber sa preuve d'existence. Si un concept ne peut pas être établi par une preuve constructive, cela signifie qu'il n'existe pas. La logique intuitionniste distingue le *être vrai* du *ne pas être faux*. Aussi n'inclut-elle pas le

principe du tiers exclus ni la double négation.

H Formes normales

■ **Définition 47 — Littéral.** Un littéral est une variable propositionnelle ou sa négation.

■ **Définition 48 — Clause conjonctive.** Une clause conjonctive est une conjonction de littéraux.

■ **Définition 49 — Forme normale disjonctive(FND).** Une forme normale disjonctive d'une formule logique est une disjonction de clauses conjonctives.

Théorème 1 — Toute formule logique est équivalente à une forme normale disjonctive.

Démonstration. En fait, il suffit d'écrire que cette formule logique est la disjonction de toutes ses valuations vraies. Plus formellement :

$$\phi \equiv \bigvee_{\substack{\nu, \\ [\![\phi]\!]_\nu = \top}} \bigwedge_{\substack{x \in \mathcal{V}, \\ \nu(x) = \top}} x \tag{3.53}$$

■

■ **Exemple 22 — Lien entre la table de vérité et la forme disjonctive complète.** On considère la formule logique $\phi = (a \vee b) \wedge ((c \Longrightarrow b) \vee a)$. Sa table de vérité contient l'ensemble possible de ses valuations vraies et fausses :

a	b	c	$a \vee b$	$c \Longrightarrow b$	$(c \Longrightarrow b) \vee a$	ϕ
T	T	T	T	T	T	V
T	T	F	T	T	T	V
T	F	T	T	F	T	V
T	F	F	T	F	T	V
F	T	T	T	T	T	V
F	T	F	T	T	T	V
F	F	T	F	F	F	F
F	F	F	F	F	F	F

Par conséquence, en prenant la disjonction de toutes les valuations vraies, on obtient la FND :

$$\phi \equiv (a \wedge b \wedge c) \vee (a \wedge b \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge \bar{b} \wedge \bar{c}) \vee (\bar{a} \wedge b \wedge c) \vee (\bar{a} \wedge b \wedge \bar{c})$$

■ **Définition 50 — Clause disjonctive.** Une clause disjonctive est une disjonction de littéraux.

■ **Définition 51 — Forme normale conjonctive (FNC).** Une forme normale conjonctive d'une formule logique est une conjonction de clauses disjonctives.

Théorème 2 — Toute formule logique est équivalente à une forme normale conjonctive.

Démonstration. Soit ϕ une formule logique. On considère sa négation $\neg\phi$. D'après la question précédente, on peut mettre $\neg\phi$ sous une forme normale disjonctive, c'est à dire

$$\neg\phi \equiv c_1 \vee c_2 \ldots \vee c_n \tag{3.54}$$

où les $c_i = l_1 \wedge l_2 \wedge \ldots \wedge l_m$ sont des conjonctions de littéraux. En appliquant la loi de Morgan, on trouve que :

$$\begin{aligned}
\neg\neg\phi &\equiv (\neg c_1) \wedge (\neg c_2) \wedge \ldots \wedge (\neg c_n) && (3.55)\\
&\equiv (l_1 \vee l_2 \ldots \vee l_m) \wedge (\neg c_2) \wedge \ldots \wedge (\neg c_n) && (3.56)\\
&\equiv d_1 \wedge d_2 \wedge \ldots \wedge d_n && (3.57)\\
&\equiv \phi && (3.58)
\end{aligned}$$

où les d_i sont des disjonctions. Donc ϕ peut s'écrire sous une forme normale conjonctive. ■

■ **Exemple 23 — FNC équivalente.** Soit la formule logique $\phi = (a \vee \neg b) \wedge \neg(c \wedge \neg(d \wedge e))$. Une forme normale équivalente est $(a \vee \neg b) \wedge (\neg c \vee d) \wedge (\neg c \vee e)$. Celle-ci est obtenue en utilisant les lois logiques pour changer la forme de la formule.

■ **Exemple 24 — Table de vérité et forme normale conjonctive.** On considère la formule logique $\phi = (a \vee b) \wedge ((c \Longrightarrow b) \vee a)$. Sa table de vérité est toujours :

a	b	c	$a \vee b$	$c \Longrightarrow b$	$(c \Longrightarrow b) \vee a$	ϕ
T	T	T	T	T	T	V
T	T	F	T	T	T	V
T	F	T	T	F	T	V
T	F	F	T	F	T	V
F	T	T	T	T	T	V
F	T	F	T	T	T	V
F	F	T	F	F	F	F
F	F	F	F	F	F	F

Pour trouver la forme normale conjonctive, il faut sélectionner les lignes pour lesquelles la valuation est fausse, puis interpréter ces valuations comme des contre-modèles. Par conséquence, on a donc la FNC :

$$\phi \equiv (a \vee b \vee \neg c) \wedge (a \vee b \vee c)$$

En utilisant la distributivité, on obtient :

$$\phi \equiv (a \vee b) \vee (\neg c \vee c) \equiv a \vee b$$

I Problème SAT

■ **Définition 52 — Problème de décision.** Un problème de décision est un problème dont la réponse est binaire : soit le on peut le décider, soit on ne peut pas.

■ **Définition 53 — Problème SAT.** Le problème de satisfaisabilité booléenne (SAT) est un problème de décision lié à une formule de logique propositionnelle et dont l'objectif est de déterminer s'il existe une valuation qui rend la formule vraie.

On note SAT(ψ) = V si ψ est satisfaisable, et F sinon.

■ **Exemple 25 — Exemples de problème SAT.** Pour définir la date d'une réunion, on considère les contraintes suivantes :

- Johann est obligé d'assister à ses cours lundi, mercredi ou jeudi.
- Cécile ne peut pas se libérer mercredi,
- Annaïg est prise le vendredi
- Prosper n'est là ni le mardi ni le jeudi.

Est-il possible de trouver un jour pour fixer la réunion ? Il s'agit d'un problème de satisfaisabilité de la formule logique :

$$(\neg L \vee \neg Me \vee \neg J) \wedge (\neg Me) \wedge (\neg V) \wedge (\neg Ma \wedge \neg J)$$

La méthode de résolution d'un problème SAT par la force brute (cf. figure 3.3) est de complexité exponentielle : on explore toutes les valuations possibles. Si la formule possède n variables, alors la complexité est en $\Theta(2^n)$. L'algorithme de Quine (cf. algorithme 1) permet d'éviter de tester des valuations qui ne sont pas solution.

J Algorithme de Quine

a Principe

L'algorithme de Quine explore l'ensemble des valeurs possibles pour chaque variable propositionnelle. Cette exploration se fait de manière arborescente [4].

La racine de l'arbre d'exploration est la formule. À chaque niveau de l'arbre, l'hypothèse est faite qu'une variable est vraie ou fausse et l'algorithme simplifie la formule en conséquence en remplaçant la variable par sa valuation. Des règles de simplifications permettent de propager la valeur et de conclure sur la possibilité de satisfaire la formule sous cette hypothèse ou non. Si c'est le cas, on étiquette la feuille de l'arbre avec un $\top$.

Si la formule simplifiée est non satisfaisable, alors on ne poursuit pas l'exploration des solutions dans cette branche de l'arbre : on la marque $\bot$. Lorsque l'arbre ne contient que des feuilles étiquetées $\top$ ou $\bot$, l'algorithme est terminé.

4. L'algorithme de Quine, c'est le retour sur trace ou backtracking appliqué à au problème SAT, cf. le cours de deuxième année, chapitre 5).

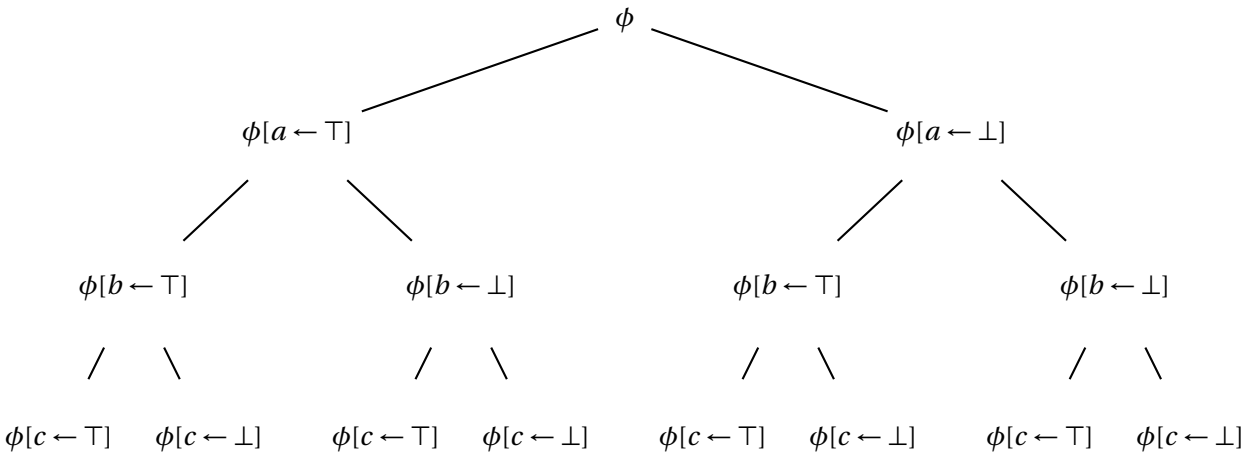


FIGURE 3.3 – Exemple d'arbre d'exploration de toutes les valuations possibles pour une formule logique simple $\phi = (a \wedge b) \vee c$. Selon la valuation des feuilles de l'arbre, on conclue sur la satisfaisabilité de la formule. Si au moins une feuille est évaluée $\top$, alors la formule est satisfaisable.

■ **Exemple 26 — Quine appliqué.** On considère la formule $\phi = (a \wedge b) \vee c$. La figure 3.4 détaille les étapes de l'algorithme de Quine sur l'arbre d'exploration. Les quatre branches de droites ont été raccourcies par les simplifications.

On en déduit la FND équivalente :

$$\phi \equiv (a \wedge b \wedge c) \vee (a \wedge \bar{b} \wedge c) \vee (\bar{a} \wedge c)$$

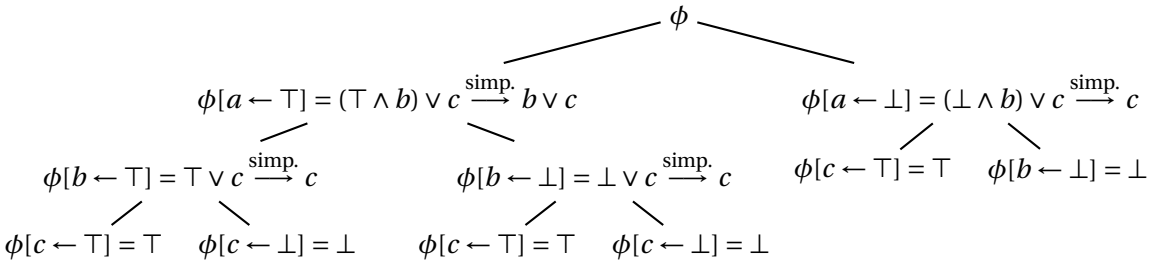


FIGURE 3.4 – Exemple d'arbre d'exploration structurant l'algorithme de Quine.

R Dans le pire des cas, l'algorithme de Quine nécessite d'explorer toutes les branches de l'arbre. Sa complexité est donc exponentielle en fonction du nombre de variables de la formule.

b Règles de simplification

Chaque nœud de l'arbre d'exploration est une formule créé en **remplaçant** une variable par $\top$ ou $\bot$ dans chaque clause de la formule FNC. Il est alors possible de simplifier cette formule,c'est-à-dire de réduire sa taille, c'est-à-dire son nombre d'opérateurs, en suivant les règles suivantes :

$$\neg\bot \equiv \top \tag{3.59}$$
$$\neg\top \equiv \bot \tag{3.60}$$
$$(\top \vee a) \equiv \top \tag{3.61}$$
$$(\bot \vee a) \equiv a \tag{3.62}$$
$$(\top \wedge a) \equiv a \tag{3.63}$$
$$(\bot \wedge a) \equiv \bot \tag{3.64}$$
$$(\top \Longrightarrow a) \equiv a \tag{3.65}$$
$$(a \Longrightarrow \top) \equiv \top \tag{3.66}$$
$$(\bot \Longrightarrow a) \equiv \top \tag{3.67}$$
$$(a \Longrightarrow \bot) \equiv \neg a \tag{3.68}$$
$$(\top \Longleftrightarrow a) \equiv a \tag{3.69}$$
$$(\bot \Longleftrightarrow a) \equiv \neg a \tag{3.70}$$
$$\tag{3.71}$$

De plus, si la formule est simplifiée en $\top$ alors l'algorithme de Quine renvoie Vrai. Sinon, il continue l'exploration de l'arbre.

c Algorithme de Quine

Algorithme 1 Algorithme Quine (SAT)

```
1 : Fonction QUINE_SAT(f)                                   ▷ f est une formule logique
2 :     SIMPLIFIER(f)
3 :     si f ≡ ⊤ alors
4 :         renvoyer Vrai
5 :     sinon si f ≡ ⊥ alors
6 :         renvoyer Faux
7 :     sinon
8 :         Choisir une variable x parmi les variables propositionnelles restantes de f
9 :         renvoyer QUINE(f[x ← ⊤]) || QUINE(f[x ← ⊥])
```

4

DÉDUCTION NATURELLE

À la fin de ce chapitre, je sais :

☞

2.3 Déduction naturelle pour la logique propositionnelle Il sagit de présenter les preuves comme permettant de pallier deux problèmes de la présentation du calcul propositionnel faite en première année : nature exponentielle de la vérification dune tautologie, faible lien avec les preuves mathématiques. Il ne sagit, en revanche, que dintroduire la notion darbre de preuve. La déduction naturelle est présentée comme un jeu de règles dinférence simple permettant de faire un calcul plus efficace que létude de la table de vérité. Toute technicité dans les preuves dans ce système est à proscrire. On sabstient dimplémenter ces règles. Lambition est dêtre capable décrire de petites preuves dans ce sys- tème. l' Ministère de lenseignement supérieur, de la recherche et de linnovation, 2021 Option informatique Filière MP https://www.enseignementsup-recherche.gouv.fr 7/10

Troisième partie

Programmation récursive

Quatrième partie

Structures de données

Cinquième partie

Exploration et graphes

5

RETOUR SUR TRACE

À la fin de ce chapitre, je sais :

- ☞ expliquer le principe du retour sur trace
- ☞ donner des exemples d'utilisation
- ☞ coder un algorithme de retour sur trace en OCaml

A Exploration

Soit un problème $\mathcal{P}$ de satisfaction de contraintes tel que les solutions $\mathcal{S}$ se trouvent dans une ensemble fini $\mathcal{E}$ de candidats. On cherche à trouver les solutions de $\mathcal{P}$ en explorant l'ensemble $\mathcal{E}$ tout en respectant les contraintes.

Il est imaginable aujourd'hui d'envisager l'usage de la force brute pour résoudre des problèmes dont la dimension est pourtant élevée.

■ **Définition 54 — Recherche par force brute.** Énumérer tous les éléments candidats de $\mathcal{E}$ et tester s'ils sont solution de $\mathcal{P}$.

Il s'agit donc d'une approche simple à énoncer et à implémenter comme le montre l'algorithme 2.

Algorithme 2 Algorithme de recherche par force brute, problème de satisfaction de contraintes

```
1 : Fonction FORCE_BRUTE(ℰ)
2 :     𝒮 ← ∅
3 :     pour e ∈ ℰ répéter
4 :         si e est un solution de 𝒫 alors
5 :             𝒮 ← 𝒮 ∪ {e}
6 :     renvoyer 𝒮
```

■ **Exemple 27 — Exemple d'algorithmes de recherche par force brute.** Parmi les algorithmes de recherche par force brute utilisés, on note :

- la recherche d'un code secret à quelques chiffres : on teste toutes les permutations possibles jusqu'à trouver la bonne,
- la recherche d'un élément dans un tableau : on teste tous les éléments les uns après les autres jusqu'à trouver le bon,
- le tri bulle : on essaie de placer un élément dans une case, puis on essaie la case du dessus.

■ **Exemple 28 — Problème des huit reines.** On cherche à placer sur un échiquier de 8x8 cases huit reines sans que celles-ci s'attaquent les unes les autres. Une solution est présentée sur la figure 5.1. On connaît les solutions de ce problème [1] et on peut même les formuler simplement.

En choisissant la recherche par force brute, il est nécessaire de tester $8^8 = 16777216$ configurations. Si le test de la validité de l'échiquier est effectué en moins d'une microseconde [a], l'intégralité des configurations sera examinée en un temps proche de la seconde.

a. ce qui est très réaliste avec une machine standard et un programme non optimisé

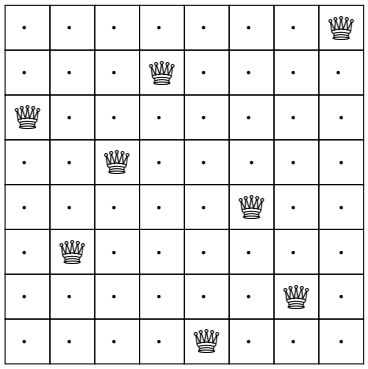


FIGURE 5.1 – Exemple d'échiquier 8x8 solution au problème des huit reines.

Même si elle est simple à énoncer et à implémenter, la recherche par force brute présente un inconvénient majeur : elle ne supporte pas le passage à l'échelle, c'est à dire qu'elle devient rapidement inutilisable à cause de l'explosion du cardinal de l'ensemble $\mathcal{E}$ à explorer qui induit un temps de calcul nécessaire rédhibitoire.

■ **Exemple 29 — Problème des n reines.** Le problème des n reines est la généralisation du problème des huit reines sur un échiquier de taille $n \times n$ et avec n reines à placer. L'ensemble des candidats est maintenant de taille n^n. Pour $n = 16$, le temps de calcul dépasse déjà la dizaine de milliers d'années. En effet, admettons que le test de validité de l'échiquier

soit toujours de l'ordre de la microseconde, on a : $(16^{16} \times 1^{-6})/(60 \times 60 \times 24 \times 365) \approx 584942$ années...

B Principe du retrour sur trace

Le retour sur trace est une technique exploratoire utilisée afin guider l'exploration et de ne pas tester toutes les configurations possibles.

■ **Définition 55 — Retour sur trace.** Le retour sur trace construit des ensembles de solutions partielles au problème $\mathcal{P}$. Ces solutions partielles peuvent être complétées de différentes manières pour former une solution au problème. La complétion des solutions se fait de manière incrémentielle.

Le retour sur trace utilise une représentation de l'espace des candidats $\mathcal{E}$ sous la forme d'un arbre de recherche. Chaque solution partielle est un nœud de cet arbre. Chaque solution complète forme un chemin descendant de la racine à une feuille de l'arbre. Au niveau de la feuille, on ne peut plus compléter la solution par quoi que ce soit.

L'algorithme de retour sur trace est un algorithme récursif qui parcours en profondeur l'arbre de recherche en vérifiant qu'il peut compléter la solution partielle par le nouveau nœud trouvé : si c'est le cas, alors il continue l'exploration de cette branche. Si ce n'est pas le cas, cette branche est écartée, car elle ne peut pas donne de solutions.

Vocabulary 5 — Backtracking ⇜⇝ Retour sur trace

Algorithme 3 Algorithme de retour sur trace

```
1 : Fonction RETOUR_SUR_TRACE(v)                    ▷ v est un nœud de l'arbre de recherche
2 :     si v est une feuille alors
3 :         renvoyer Vrai
4 :     sinon
5 :         pour chaque fils u de v répéter
6 :             si u peut compléter une solution partielle au problème P alors
7 :                 RETOUR_SUR_TRACE(u)
8 :     renvoyer Faux
```

■ **Exemple 30 — Retour sur trace sur le problème des quatre reines.** L'arbre de recherche nécessaire à l'exécution de l'algorithme de retour sur trace pour le problème des quatre reines est représenté sur la figure 5.2. La racine de l'arbre est le début de l'algorithme : on n'a pas encore placé de reines. Le première étape est le placement d'une reine sur l'échiquier : sur une même ligne on peut la placer sur quatre colonnes différentes qu'il va falloir tester. Chaque colonne représente donc une solution partielle différente et est un nœud fils de la racine. On peut placer la première reine sur n'importe quelle colonne.

La seconde étape est le placement d'une deuxième reine. Comme on parcours l'arbre

en profondeur, on teste la première configuration en premier, c'est à dire une reine sur la première case de la première ligne. On teste alors toutes les solutions partielles possibles. Le premier nœud ne satisfait pas les conditions de validité : on ne peut pas placer une reine dans la première colonne car il y en a déjà une sur la première ligne. Tout le sous-arbre lié à cette solution partiel est élagué. De même pour le second nœud pour lequel la première reine peut attaquer en diagonale.

On revient donc là où on en était et on continue avec les autres fils valides.

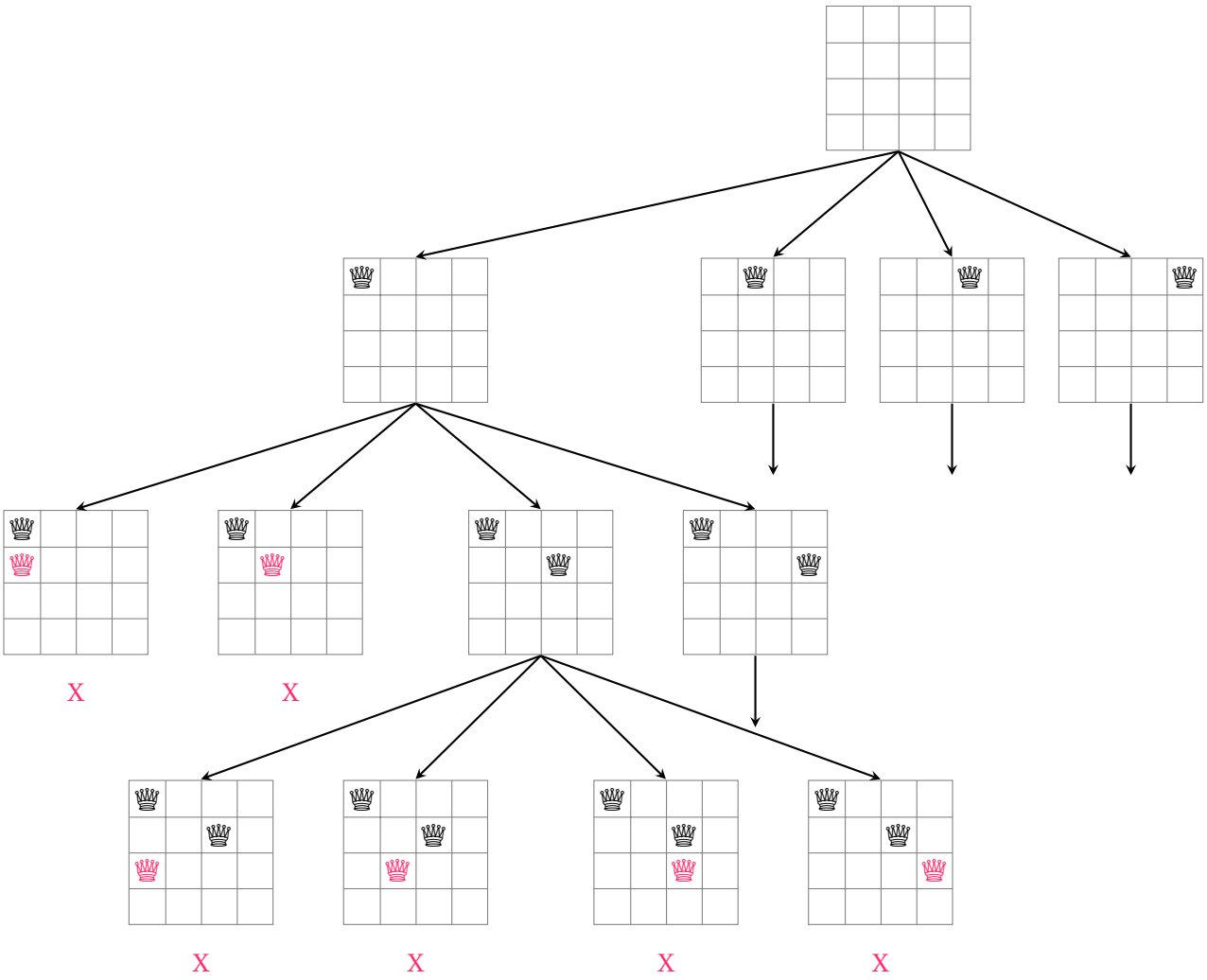


FIGURE 5.2 – Exemple d'arbre de recherche structurant l'algorithme de retour sur trace. Application au problème de quatre reines.

6

LES MOTS DES GRAPHES

À la fin de ce chapitre, je sais :

- ☞ utiliser des mots pour décrire les graphes
- ☞ énumérer quelques graphes remarquables
- ☞ distinguer un parcours d'une chaîne et d'un cycle
- ☞ distinguer un graphe orienté, non orienté, pondéré et un arbre

La théorie des graphes en mathématiques discrètes étudie les graphes comme objet mathématique. En informatique, en plus de les étudier, on a la chance de pouvoir les programmer, de jouer avec pour résoudre une infinité de problèmes. Les domaines d'application des graphes sont innombrables : les jeux, la planification, l'organisation, la production, l'optimisation, les programmes et modèles informatiques, les trajets dans le domaine des transports, le tourisme, la logistique ou tout simplement la géométrie... **Les graphes sont des objets simples que tout le monde peut dessiner.** Même s'il ne vous apparaît pas immédiatement que résoudre un sudoku est équivalent à la coloration d'un graphe, la pratique de ces derniers vous amènera à regarder le monde différemment.

A Typologie des graphes

■ **Définition 56 — Graphe.** Un graphe G est un couple $G = (V, E)$ où V est un ensemble fini et non vide d'éléments appelés sommets et E un ensemble de paires d'éléments de V appelées arêtes.

Vocabulary 6 — Graph ⇜⇝ Graphe

Vocabulary 7 — Vertex (plural : vertices) ⇜⇝ Sommet

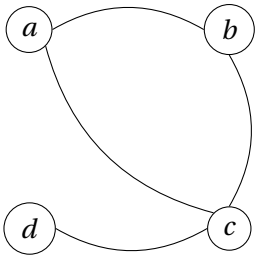


FIGURE 6.1 – Graphe simple

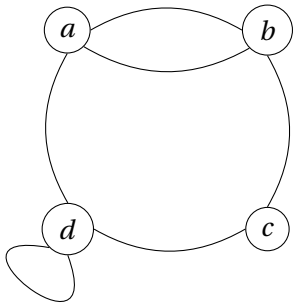


FIGURE 6.2 – Multigraphe à une boucle est deux arêtes parallèles

Vocabulary 8 — Edge ⇜⇝ Arête

La notation $G = (V, E)$ dérive donc directement des premières lettres des mots anglais. La définition 56 est en fait celle des **graphes simples et non orientés : ce sont eux que l'on considèrera la plupart du temps.**

■ **Définition 57 — Boucle.** Une boucle est une arête reliant un sommet à lui-même.

■ **Définition 58 — Arêtes parallèles.** Deux arêtes sont parallèles si elles relient les mêmes sommets.

■ **Définition 59 — Graphe simple.** Un graphe simple est un graphe sans arêtes parallèles et sans boucles.

■ **Définition 60 — Multigraphe.** Un multigraphe est un graphe avec des boucles et des arêtes parallèles.

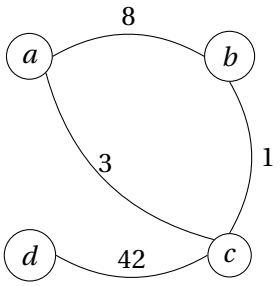


FIGURE 6.3 – Graphe pondéré

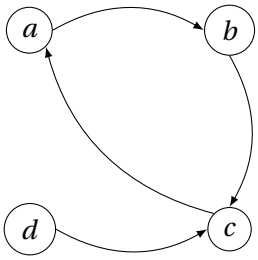


FIGURE 6.4 – Graphe orienté

■ **Définition 61 — Graphe pondéré.** Un graphe $G = (V, E)$ est pondéré s'il existe une application $w : E \longrightarrow \mathbb{R}$. Le poids de l'arête ab vaut $w(ab)$.

■ **Définition 62 — Graphe orienté.** Un graphe $G = (V, E)$ est orienté si ses arêtes sont orientées selon une direction. Les arêtes sont alors désignées par le mot arc.

■ **Définition 63 — Graphe complet.** Un graphe $G = (V, E)$ est complet si et seulement si une arête existe entre chaque sommet, c'est-à-dire si tous les sommets sont voisins.

En hommage à Kuratowski, on désigne les graphes complets par la lettre K_o indicée par l'ordre du graphe (cf. définition 67). La figure 6.5 représente le graphe complet d'ordre cinq K_5. Kuratowski a notamment démontré [10] que K_5 n'est pas planaire : quelle que soit la manière de représenter ce graphe sur un plan, des arêtes se croiseront. Le graphe de la figure 6.1 est planaire.

■ **Définition 64 — Graphe planaire.** Une graphe planaire est un graphe que l'on peut représenter sur un plan sans qu'aucune arête ne se croise.

■ **Définition 65 — Graphe biparti.** un graphe $G = (V, E)$ est biparti si l'ensemble V de ses sommets peut être divisé en deux sous-ensembles disjoints U et W tels que chaque arête de E ait une extrémité dans U et l'autre dans W.

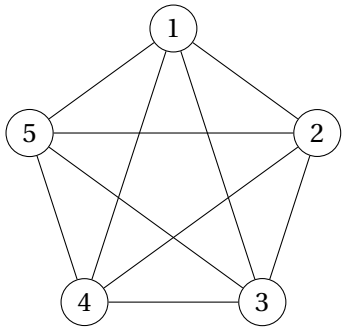


FIGURE 6.5 – Graphe complet K_5

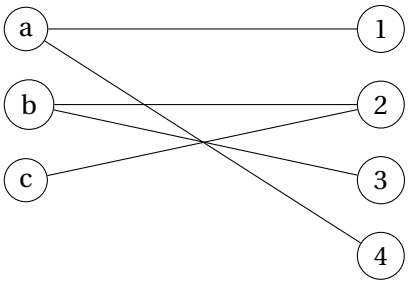


FIGURE 6.6 – Graphe biparti

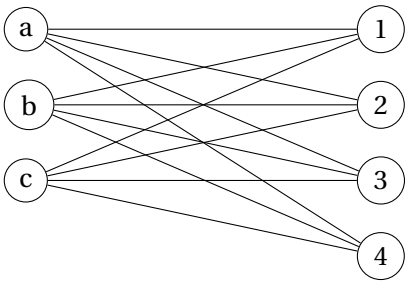


FIGURE 6.7 – Graphe biparti complet K_{34}

B Implémentation des graphes

On peut représenter graphiquement un graphe comme sur les figures précédentes 6.1, 6.3 ou 6.4. On peut également chercher à les implémenter sous la forme d'ensembles, de matrices ou de listes.

■ **Exemple 31 — Graphe et ensembles.** Le graphe de la figure 6.1 est un graphe simple que l'on peut noter $G = \{V = \{a,b,c,d\}, E = \{\{a,b\},\{a,c\},\{b,c\},\{c,d\}\}\}$ ou plus simplement $G = \{V = \{a,b,c,d\}, E = \{ab,ac,bc,cd\}\}$.

■ **Définition 66 — Adjacent ou voisins.** Deux sommets a et b sont adjacents ou voisins ă si le graphe contient une arête ab. Deux arêtes sont adjacentes ou voisines s'il existe un sommet commun à ces deux arêtes.

■ **Exemple 32 — Graphe et matrice d'adjacence.** Grâce au concept d'adjacence, on peut représenter un graphe par une matrice d'adjacence. Pour construire une telle matrice, il faut d'abord ordonner arbitrairement les sommets du graphes. Par exemple, pour le graphe de la figure 6.1, on choisit l'ordre (a,b,c,d). Les coefficients m_{ij} de la matrice d'adjacence sont calculés selon la règle suivante :

$$m_{ij} = \begin{cases} 1 & \text{s'il existe une arête entre le sommet } i \text{ et le sommet } j \\ 0 & \text{sinon} \end{cases} \tag{6.1}$$

Pour le graphe de la figure 6.1, on obtient :

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{6.2}$$

La matrice d'adjacence d'un graphe simple non orienté est de diagonale nulle (pas de boucles) et symétrique.

La matrice d'un graphe orienté n'est pas forcément symétrique.

Dans le cas d'un graphe pondéré, on peut remplacer le coefficient de la matrice par le poids de l'arête considérée.

En Python, on pourra utiliser une liste de liste ou un tableau numpy pour implémenter une matrice d'adjacence.

■ **Exemple 33 — Graphe et liste d'adjacence.** On peut représenter un graphe par la liste des voisins de chaque sommet. Par exemple, si on dénote les sommets a, b, c et d par les indices 1, 2, 3, et 4, alors le graphe de la figure 6.1 peut être décrit par la liste `[[2,3], [1,3], [1,2,4], [3]]`. Si on choisit d'utiliser un dictionnaire, alors on peut l'écrire : `{1:[2,3], 2:[1,3], 3:[1,2,4], 4:[3]}`.

Pour résumé, en Python, on a le choix de la représentation :

```
G = [[2,3], [1,3], [1,2,4], [3]]
G = {1:[2,3], 2: [1,3], 3:[1,2,4], 4:[3]}
n = len(G)   # graph order
```

Par ailleurs, si le graphe est pondéré, la liste d'adjacence est une liste de listes de tuples, chaque tuple étant un couple (sommet, poids). Par exemple, en OCaml :

```
type adj_list_graph = (int * int) list list;;

let lcg = [[(2,1); (1,7)];
           [(3,4); (4,2); (2,5); (0,7)];
           [(5,7); (4,2); (1,5); (0,1)];
           [(4,5); (1,4)];
           [(5,3); (3,5); (2,2); (1,2)];
           [(4,3); (2,7)]];;
```

R Ni la représentation graphique de la figure 6.1, ni la matrice d'adjacence M de l'équation 6.2, ni les listes d'adjacence ne sont des représentations uniques. On peut tracer différemment le graphe ou choisir un autre ordre pour les sommets et obtenir une autre matrice ou une autre liste d'adjacence. Cela traduit l'isomorphisme des graphes.

R Le choix d'une implémentation ou d'une autre est avant tout lié aux choix des algorithmes que l'on va utiliser. La structure de donnée utilisée est souvent le facteur clef qui permet d'améliorer ou de détériorer les performances d'un algorithme.

C Caractérisation structurelle des graphes

■ **Définition 67 — Ordre d'un graphe.** L'ordre d'un graphe est le nombre de ses sommets. Pour $G = (V, E)$, l'ordre du graphe vaut donc le cardinal de l'ensemble V que l'on note généralement $|V|$. On note parfois l'ordre d'un graphe $|G|$.

R Si $|V| = n$, alors une matrice d'adjacence de $G = (V, E)$ est de dimension $n \times n$. Une liste d'adjacence de G a pour taille n.

■ **Définition 68 — Taille d'un graphe.** La taille d'un graphe désigne le nombre de ses arêtes. On le note $|E|$ et parfois $||G||$

■ **Définition 69 — Voisinage d'un sommet.** L'ensemble de voisins d'un sommet a d'un graphe $G = (V, E)$ est le voisinage de ce sommet. On le note généralement $\mathcal{N}_G(a)$.

■ **Définition 70 — Incidence.** Une arête est dite incidente à un sommet si ce sommet est une des extrémités de cette arête

■ **Définition 71 — Degré d'un sommet.** Le degré $d(a)$ d'un sommet a d'un graphe G est le nombre d'arêtes incidentes au sommet a. C'est aussi $|\mathcal{N}_G(a)|$.

■ **Définition 72 — Degrés d'un graphe orienté.** Dans un graphe orienté G et pour un sommet a de ce graphe, on distingue :

- le degré entrant $d_+(a)$: le nombre d'arêtes incidentes à a et dirigées sur a,
- le degré sortant $d_-(a)$: le nombre d'arêtes incidentes qui sortent de a et qui sont dirigées vers un autre sommet.

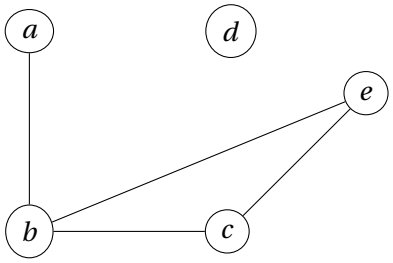


FIGURE 6.8 – Graphe d'ordre cinq, de taille quatre et de séquence `[0,1,2,2,3]`. Le sommet d est isolé. Ce graphe n'est ni complet ni connexe.

■ **Définition 73 — Sommet isolé.** Un sommet isolé est un sommet dont le degré vaut zéro.

■ **Définition 74 — Égalité de deux graphes.** Deux graphes $G = (V, E)$ et $G' = (V', E')$ sont égaux si et seulement si $V = V'$ et $E = E'$.

■ **Définition 75 — Séquence des degrés.** La séquence des degrés d'un graphe G est la liste ordonnée par ordre croissant des degrés des sommets de G.

Sur la figure 6.8, on a représenté un graphe d'ordre cinq avec un sommet isolé. Ce graphe n'est pas connexe ni complet et sa séquence des degrés est `[0,1,2,2,3]`.

■ **Définition 76 — Graphe complémentaire.** Soit $G = (V, E)$ un graphe. On dit que $\overline{G} = (V, \overline{E})$ est le complémentaire de G si les arêtes de $\overline{G}$ sont les arêtes possibles qui ne figurent pas dans G. On note ces arêtes $\overline{E}$.

Par exemple, le complémentaire du graphe 6.8 est représenté sur la figure 6.9.

D Isomorphisme des graphes

Considérons les graphes des figures 6.10 et 6.11. Ils ne diffèrent que par les noms des sommets et la signification des arêtes. Si on ne prête pas attention aux noms des sommets ni à la signification des arêtes, ces deux graphes sont identiques, leurs caractéristiques sont les mêmes :

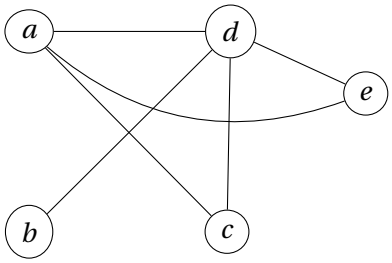


FIGURE 6.9 – Graphe complémentaire du graphe de la figure 6.8

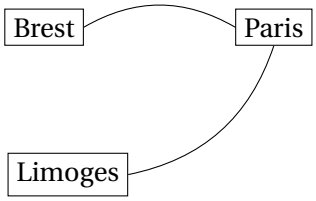


FIGURE 6.10 – Graphe d'ordre trois, de taille deux et de séquence `[1,1,2]`

ordre, degré, taille. On dit qu'ils sont isomorphes ou qu'ils sont identiques à un isomorphisme près. Finalement, c'est la structure du graphe telle qu'on peut la caractériser qui importe, pas son apparence.

■ **Définition 77 — Graphes isomorphes.** Deux graphes $G = (V, E)$ et $G' = (V', E')$ sont isomorphes si et seulement s'il existe une bijection σ de V vers V' pour laquelle $\sigma(E) = E'$, c'est à dire qu'à chaque arête ab de E correspond une seule arête de E' notée $\sigma(a)\sigma(b)$.

■ **Exemple 34 — Isomorphes et bijection.** Considérons les deux graphes G et G' représentés sur les figures 6.12 et 6.13.

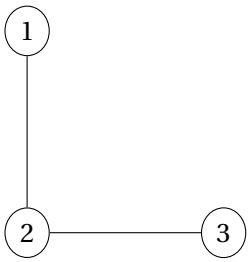


FIGURE 6.11 – Graphe d'ordre trois, de taille deux et de séquence `[1,1,2]`

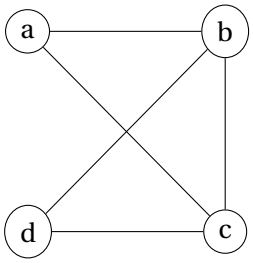


FIGURE 6.12 – Graphe d'exemple $G = (V = \{a, b, c, d\}, E = \{ab, ac, bc, bd, dc\})$

On peut les définir par des ensembles de la manière suivante :

$$G = (V = \{a, b, c, d\}, E = \{ab, ac, bc, bd, dc\}) \tag{6.3}$$
$$G' = (V' = \{1, 2, 3, 4\}, E' = \{12, 14, 24, 23, 43\}) \tag{6.4}$$

Formulé de la sorte, on pourrait croire que ces deux graphes ne sont pas isomorphes. Pourtant, c'est le cas. Comment le montrer? En exhibant une bijection ad-hoc!

On cherche donc une bijection entre les deux graphes en comparant les degrés des sommets et en observant leurs arêtes.

On peut proposer la bijection $\sigma : V \longrightarrow V'$ telle que :

$$\sigma(a) = 1 \tag{6.5}$$
$$\sigma(b) = 2 \tag{6.6}$$
$$\sigma(c) = 4 \tag{6.7}$$
$$\sigma(d) = 3 \tag{6.8}$$

On a également la correspondance des arêtes :

$$\sigma(a)\sigma(b) = 12 \tag{6.9}$$
$$\sigma(a)\sigma(c) = 14 \tag{6.10}$$
$$\sigma(b)\sigma(c) = 24 \tag{6.11}$$
$$\sigma(b)\sigma(d) = 23 \tag{6.12}$$
$$\sigma(c)\sigma(d) = 43 \tag{6.13}$$

R On peut compter le nombre de graphes isomorphes pour un ordre donné. Par exemple, il y a deux graphes isomorphes d'ordre 2 et 8 d'ordre 3.

E Chaînes, cycles et parcours

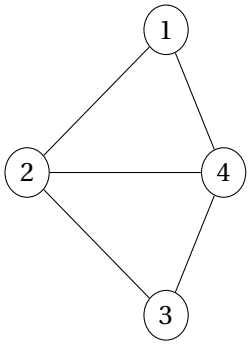


FIGURE 6.13 – Graphe d'exemple $G' = (V' = \{1,2,3,4\}, E' = \{12,14,24,23,43\})$

■ **Définition 78 — Chaîne.** Une chaîne reliant deux sommets a et b d'un graphe non orienté est une suite finie d'arêtes consécutives reliant a à b. Dans le cas d'un graphe orienté on parle de chemin.

■ **Définition 79 — Chaîne élémentaire.** Une chaîne élémentaire ne passe pas deux fois par un même sommet : tous ses sommets sont distincts.

■ **Définition 80 — Chaîne simple.** Une chaîne simple ne passe pas deux fois par une même arête : toutes ses arêtes sont distinctes.

■ **Définition 81 — Longueur d'une chaîne.** La longueur d'une chaîne $\mathcal{C}$ est :

- le nombre d'arêtes que comporte la chaîne dans un graphe simple non pondéré,
- la somme des poids des arêtes de la chaîne, c'est à dire $\sum_{e \in \mathcal{C}} w(e)$, dans le cas d'un graphe simple pondéré dont la fonction de valuation est w.

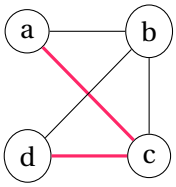


FIGURE 6.14 – Exemple de chaîne simple reliant a à d en rouge

■ **Définition 82 — Cycle.** Un cycle est une chaîne simple dont les deux sommets extrémités sont identiques.

La longueur d'un cycle est le nombre d'arêtes qu'il contient. Dans le cas des graphes orientés on parle de circuit.

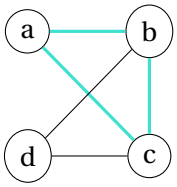


FIGURE 6.15 – Exemple de cycle en turquoise

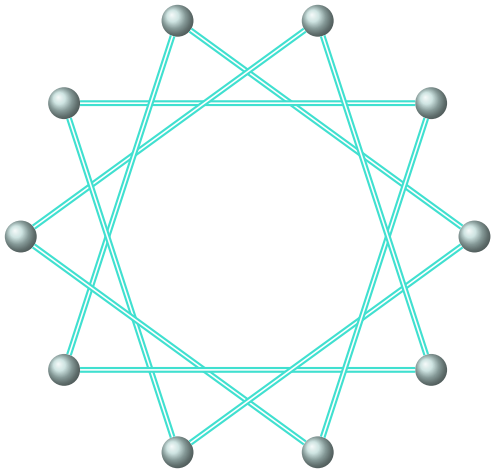


FIGURE 6.16 – Saurez-vous trouver le cycle eulérien de ce graphe ?

■ **Définition 83 — Chaîne eulérienne.** Une chaîne eulérienne est une chaîne simple qui passe par toutes les arêtes d'un graphe.

■ **Définition 84 — Cycle eulérien.** Un cycle eulérien est un cycle passant exactement une fois par chaque arête d'un graphe.

■ **Définition 85 — Cycle hamiltonien.** Un cycle hamiltonien est un sous-graphe couvrant qui est un cycle. Autrement dit, c'est un cycle qui passe par tous les sommets d'un graphe.

R Rowan Hamilton était un astronome irlandais qui a inventé le jeu icosien [a] en 1857.

a. The icosian game, jeu équivalent à l'icosagonal d'Édouard Lucas [11]

R Les graphes complets K_n sont eulériens et hamiltoniens : ils possèdent à la fois un cycle eulérien et un cycle hamiltonien.

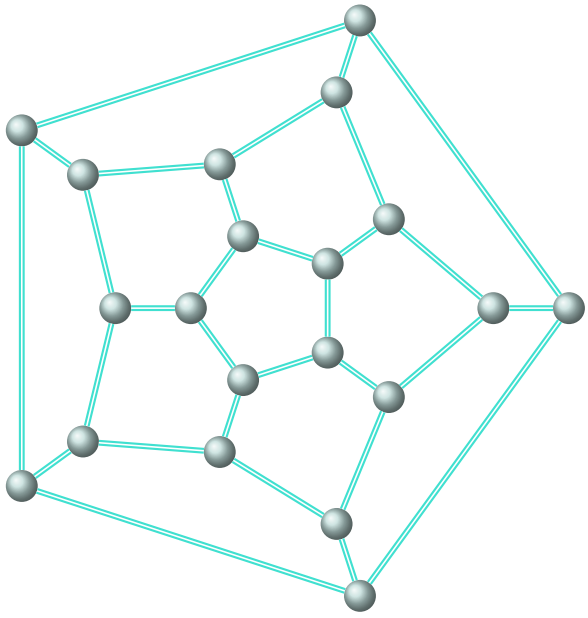


FIGURE 6.17 – Graphe du jeu icosien et du dodécahèdre (solide régulier à 12 faces pentagonales). C'est un graphe cubique car chaque sommet possède trois voisins. Ce graphe possède un cycle hamiltonien. Saurez-vous le trouver ?

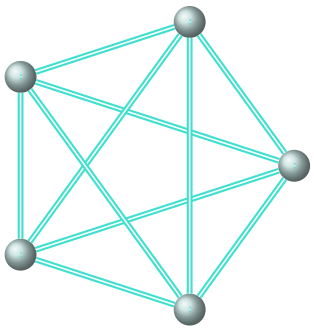


FIGURE 6.18 – Graphe K_5 : saurez-vous trouver des cycles hamiltonien et eulérien de ce graphe ?

R On peut également définir une chaîne comme le graphe d'ordre n isomorphe au graphe $P_n = \{V_n = \{1,2,\ldots,n\}, E_n = \{\{1,2\},\{2,3\},\ldots\{n-1,n\}\}\}$. Par convention, on pose $P_1 = \{V_1 = \{1\}\}$ et $E_1 = \emptyset$. Les extrémités de la chaînes sont les deux sommets de degré 1.

Avec la même approche et les mêmes notations, un cycle devient alors un graphe isomorphe au graphe $C_n = \{V_n, E'_n = E_n \cup \{n,1\}\}$, c'est à dire que la chaîne finit là où elle a commencé. L'ordre de C_n est supérieur ou égal à trois.

■ **Définition 86 — Parcours.** Un parcours d'un graphe G est une liste non vide et ordonnées de sommets de G telles que deux sommets consécutifs sont adjacents dans G. Il peut y avoir des répétitions de sommets dans un parcours, mais il n'y a pas de répétitions d'arêtes dans un cycle ou une chaîne simple.

Par exemple, $\pi = \{a, b, c, d, b\}$ est un parcours sur le graphe de la figure 6.15.

F Sous-graphes et connexité

■ **Définition 87 — Graphe connexe.** Un graphe $G = (V, E)$ est connexe si et seulement si pour tout couple de sommets (a, b) de G, il existe une chaîne d'extrémités a et b.

Par exemple, le graphe de la figure 6.1 est connexe, mais pas celui figure 6.8.

■ **Définition 88 — Sous-graphe.** Soit $G = (V, E)$ un graphe, alors $G' = \{V', E'\}$ est un sous-graphe de G si et seulement si $V' \subseteq V$ et $E' \subseteq E$.

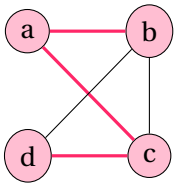


FIGURE 6.19 – Exemple de sous-graphe couvrant G en rouge : $G = (V = \{a, b, c, d\}, E = \{ab, ac, cd\})$

■ **Définition 89 — Sous-graphe couvrant.** G' est un sous-graphe couvrant de G si et seulement si G' est un sous-graphe de G et $V' = V$.

■ **Définition 90 — Sous-graphe induit.** Soit $S \subset V$ non vide. G' est un sous-graphe de G induit par S et on note $G[S]$ si et seulement si G' admet pour arêtes celles de G dont les deux extrémités sont dans S.

■ **Définition 91 — Clique.** Une clique est d'un graphe $G = (V, E)$ un sous-ensemble $C \subseteq V$ des sommets dont le sous-graphe induit $G[C]$ est complet.

Sur la figure 6.20, l'ensemble $S = \{b, c, d\}$ est un clique.

G Coloration de graphes

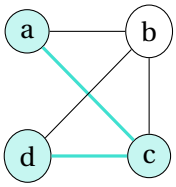


FIGURE 6.20 – Exemple de sous-graphe induit par les sommets $S = \{a, c, d\}$ en turquoise. $G[S] = (V = \{a, c, d\}, E = \{ac, cd\})$

■ **Définition 92 — Coloration.** Une coloration d'un graphe simple est l'attribution d'une couleur aux sommets de ce graphe.

■ **Définition 93 — Coloration valide.** Une coloration est valide lorsque deux sommets adjacents n'ont jamais la même couleur.

■ **Définition 94 — Nombre chromatique.** Le nombre chromatique d'un graphe G est le plus petit nombre de couleurs nécessaires pour obtenir une coloration valide de ce graphe. On le note généralement $\chi(G)$.

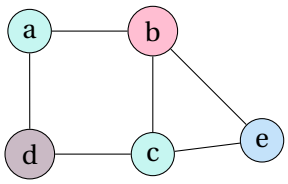


FIGURE 6.21 – Exemple de 4-coloration valide d'un graphe. Cette coloration n'est pas optimale.

■ **Définition 95 — k-coloration.** Lorsqu'une coloration de graphe utilise k couleurs, on dit d'elle que c'est une k-coloration.

■ **Définition 96 — Coloration optimale.** Une $\chi(G)$-coloration valide est une coloration optimale d'un graphe G.

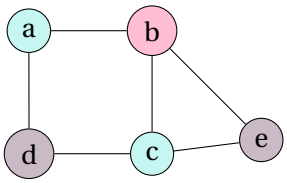


FIGURE 6.22 – Exemple de 3-coloration valide d'un graphe. Cette coloration est optimale.

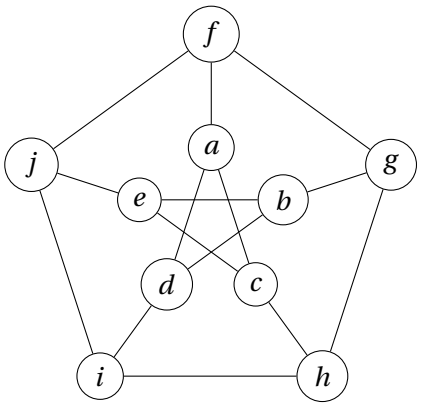


FIGURE 6.23 – Graphe de Petersen : saurez-vous proposer une coloration optimale de ce graphe sachant que son nombre chromatique vaut trois ?

H Distances

■ **Définition 97 — Distance dans un graphe simple non pondéré.** La distance d'un sommet a à un sommet b dans un graphe simple non pondéré G est la plus courte chaîne d'extrémités a et b. On la note $d_G(a,b)$.

(R) Cette définition coïncide avec la notion de distance en mathématiques. Pour les sommets a, b et c de G

- $d_G(a,b) = 0 \Leftrightarrow a = b$
- $d_G(a,b) = d_G(b,a)$
- $d_G(a,b) \leqslant d_G(a,c) + d_G(c,b)$

■ **Définition 98 — Valuation d'une chaîne dans un graphe pondéré.** La valuation d'une chaîne dans un graphe pondéré est la somme des poids de chacune de ses arêtes. Pour une chaîne P, on la note $v(P)$.

■ **Définition 99 — Distance dans un graphe pondéré.** La distance d'un sommet a à un sommet b dans un graphe pondéré G est la valuation minimum des chaînes d'extrémités a et b. On la note $d_{G,v}(a,b)$.

I Arbres

■ **Définition 100 — Arbre.** Un arbre est un graphe connexe et acyclique.

■ **Définition 101 — Feuilles.** Dans un arbre, les sommets de degré un sont appelés les feuilles.

■ **Définition 102 — Arbre recouvrant.** Un arbre recouvrant d'un graphe G est un sous-graphe couvrant de G qui est un arbre.

■ **Définition 103 — Arbre enraciné.** Parfois, on distingue un sommet particulier dans un arbre A : la racine r. Le couple (A,r) est un nommé arbre enraciné. On le représente un tel arbre verticalement avec la racine placée tout en haut comme sur la figure 9.1.

■ **Définition 104 — Arbre binaire.** Un arbre binaire est un graphe connexe acyclique pour lequel le degré de chaque sommet vaut au maximum trois. Le degré de la racine vaut au maximum deux.

■ **Définition 105 — Arbre binaire parfait.** Un arbre binaire parfait est un arbre dans lequel

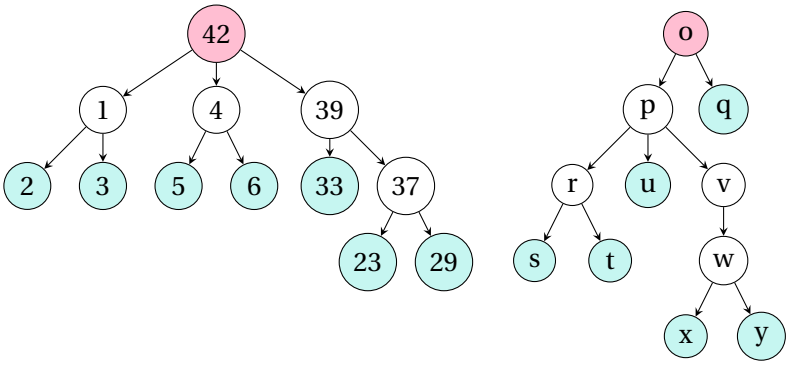


FIGURE 6.24 – Exemples d'arbres enracinés. Les racines des arbres sont en rouge, les feuilles en turquoise. Le tout forme une forêt.

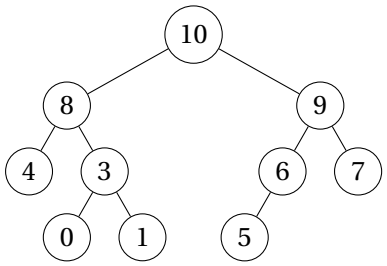


FIGURE 6.25 – Arbre binaire

tous les niveaux sauf le dernier doivent être totalement remplis. Si le dernier n'est pas rempli totalement alors il doit être rempli de gauche à droite.

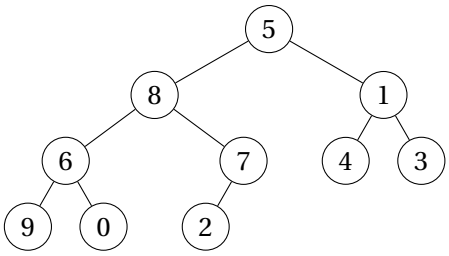


FIGURE 6.26 – Arbre binaire parfait

Ces arbres sont illustrés sur les figure 9.2 et 9.3.

7

PROPRIÉTÉS DES GRAPHES

À la fin de ce chapitre, je sais :

- ☞ expliquer le lemme des poignées de mains
- ☞ caractériser un cycle eulérien
- ☞ caractériser un graphe connexe, acyclique ou un arbre

A Des degrés et des plans

Théorème 3 — Somme des degrés d'un graphe. Le nombre d'arêtes d'un graphe simple est égale à la moitié de la somme des degrés de sommets de ce graphe.

Plus formellement, soit $G = (V, E)$ un graphe simple alors on a :

$$2|E| = \sum_{v \in V} d(v) \tag{7.1}$$

R On appelle souvent ce théorème le lemme des poignées de mains car il peut se traduire par le fait que dans un graphe il y a toujours un nombre pair de sommets de degré impair.

Théorème 4 — Formule d'Euler pour les graphes planaires. Soit $G = (V, E)$ un graphe simple. G est planaire si le nombre de régions du plan qu'il délimite R vaut :

$$R = 2 + |E| - |V| \tag{7.2}$$

R Pour vérifier cette formule, il ne faut pas oublier la région extérieur au graphe qui compte également.

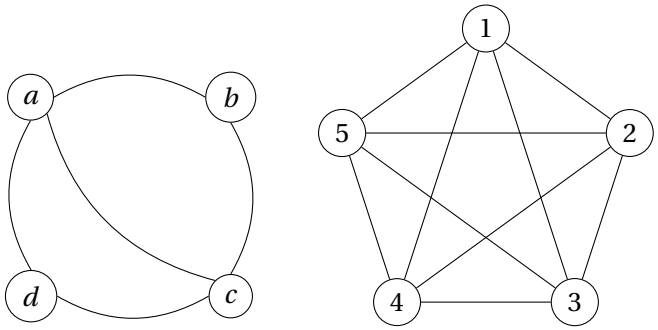


FIGURE 7.1 – Sur ces graphes, on peut vérifier les théorèmes de caractérisation des chaînes eulériennes et des cycles eulériens

B Caractérisation des chaînes, des cycles et des graphes

Théorème 5 — Caractérisation d'une chaîne eulérienne. Il existe une chaîne eulérienne dans un graphe lorsque seuls les sommets de départ et d'arrivée sont de degré impair.

Théorème 6 — Caractérisation d'un cycle eulérien. Il existe un cycle eulérien dans un graphe si tous les sommets sont de degré pair.

Pour bien visualiser ces caractérisations, on peut s'entraîner sur les graphes de la figure 7.1. Le graphe d'ordre quatre possède une chaîne eulérienne mais pas de cycle eulérien. Le graphe complet K_5 possède les deux.

Théorème 7 — Chaînes extraites et existence de chaînes. S'il existe un parcours d'un sommet a vers un sommet b dans un graphe G alors il existe une chaîne de a vers b dont les arêtes sont des arêtes du parcours.

Par transitivité, s'il existe une chaîne de a à b et une de a à c alors il existe une chaîne de b à c.

On appelle graphe hamiltonien un graphe qui possède un cycle hamiltonien (cf. définition 85). Un graphe hamiltonien :

- est connexe,
- d'ordre supérieur ou égal à trois,
- n'a pas de sommets de degré un.

Théorème 8 — Condition nécessaire pour un graphe non hamiltonien. Soit $G = (V, E)$ un graphe. Soit $U \subseteq V$ un ensemble de sommets de G. Si le nombre de composantes connexes de $G = (V \setminus U, E)$ est strictement supérieur au nombre de sommets de U, alors G n'est pas hamiltonien.

Théorème 9 — Condition nécessaire pour un graphe hamiltonien. Soit $G = (V, E)$ un graphe d'ordre supérieur ou égal à deux. Si pour toute paire de sommets a et b de G on a :

$$d(a) + d(b) \geqslant |V| \tag{7.3}$$

alors G est hamiltonien.

C Graphes acycliques et connexes

Théorème 10 — Condition nécessaire d'acyclicité d'un graphe. Soit un graphe $G = (V, E)$ possédant au moins une arête et acyclique alors G possède au moins deux sommets de degré un et on a :

$$|E| \leqslant |V| - 1 \tag{7.4}$$

Théorème 11 — Condition nécessaire de connexité d'un graphe. Si un graphe $G = (V, E)$ est connexe alors on a :

$$|E| \geqslant |V| - 1 \tag{7.5}$$

R On déduit des deux théorèmes précédents qu'un arbre (cf. définition 100) possède exactement $|V| - 1$ arêtes.

D Coloration, graphes planaires et nombre chromatique

Théorème 12 — Trois couleurs. Si tous les degrés des sommets d'un graphe planaire sont pairs, alors trois couleurs suffisent pour obtenir une coloration valide.

Théorème 13 — Quatre couleurs. Le nombre chromatique d'un graphe planaire ne dépasse jamais quatre.

On peut chercher à encadrer le nombre chromatique d'un graphe. Dans une premier temps, on peut remarquer que :

- $\chi(G) \leqslant |V|$, autrement dit, l'ordre d'un graphe est supérieur ou égal au nombre chromatique. L'égalité est atteinte pour les graphes complets : tous les sommets étant reliés les uns aux autres, on ne peut qu'utiliser des couleurs différentes pour chaque sommet.
- Pour un sous-graphe G' de G, on a $\chi(G') \leqslant \chi(G)$.

■ **Définition 106 — Degré maximum des sommets d'un graphe.** On note $\Delta(G)$ le degré maximum des sommets d'un graphe G.

■ **Définition 107 — Ordre du plus grand sous-graphe complet d'un graphe.** On note $\omega(G)$ l'ordre du plus grand sous-graphe **complet** d'un graphe G.

Théorème 14 — Encadrement du nombre chromatique. Pour un graphe G, on a :

$$\omega(G) \leqslant \chi(G) \leqslant \Delta(G) + 1 \tag{7.6}$$

E Principe d'optimalité et plus court chemin dans un graphe

Théorème 15 — Optimalité et plus court chemin dans graphe. Si $a \rightsquigarrow b$ est le plus court chemin passant par un sommet c, alors les sous-chemins $a \rightsquigarrow c$ et $c \rightsquigarrow b$ sont des plus courts chemins.

Démonstration. Soit $a \rightsquigarrow b$ le plus court chemin passant par un sommet c dans un graphe G. Si $a \rightsquigarrow c$ n'est pas le plus court chemin, alors il suffit de prendre le plus court chemin entre a et c et de le joindre à $c \rightsquigarrow b$ pour obtenir un chemin plus court de a vers b. Ce qui est en contradiction avec notre hypothèse que $a \rightsquigarrow b$ est le plus court chemin. ■

8

ALGORITHMES ET GRAPHES

À la fin de ce chapitre, je sais :

- ☞ parcourir un graphe en largeur et en profondeur
- ☞ utiliser une file ou un pile pour parcourir un graphe
- ☞ énoncer le principe de l'algorithme de Dijkstra (plus court chemin)
- ☞ expliquer l'intérêt d'un tri topologique
- ☞ schématiser le concept de forte connexité
- ☞ expliquer ce qu'est un arbre recouvrant
- ☞ expliquer l'intérêt d'un graphe biparti

A Parcours d'un graphe

Le parcours d'un graphe est une opération fondamentale et utilisée par de nombreux algorithmes, notamment Dijkstra et A*. On peut facilement mémoriser les différentes stratégies en observant les types d'ensemble qui sont utilisés pour stocker les sommets à parcourir au cours de l'algorithme :

1. Le parcours en **largeur** passe par tous les voisins d'un sommet avant de parcourir les descendants de ces voisins. Les sommets passent dans une **file** de type First In First Out.
2. Le parcours en **profondeur** passe par tous les descendants d'un voisin d'un sommet avant de parcourir tous les autres voisins de ce sommet. L'implémentation la plus simple et la plus commune est récursive. Mais on peut aussi l'implémenter de manière itérative en utilisant une *pile* de sommets de type Last In First Out.
3. L'algorithme de **Dijkstra** s'applique à un graphe pondéré : il passe par le voisin le plus proche d'un sommet avant de parcourir les autres voisins de ce sommet. C'est un parcours en largeur qui utilise une **file de priorités** : lorsqu'on insère un nouvel élément

dans cette file, celui-ci est placé d'après son niveau de priorité, le plus prioritaire en premier. Dans notre cas, la priorité est la distance la plus faible : la plus petite distance se situe en tête de la file.

a Parcours en largeur

Parcourir en largeur un graphe signifie qu'on cherche à visiter tous les voisins situé à une même distance d'un sommet (sur un même niveau) avant de parcourir le reste du graphe.

Vocabulary 9 — Breadth First Search ⇜⇝ Parcours en largeur

Le parcours en largeur d'un graphe (cf. algorithme 4) est un algorithme à la base de nombreux développement comme l'algorithme de Dijkstra et de Prim (cf. algorithmes 8 et 12). Il utilise une file FIFO [1] afin de gérer la découverte des voisins dans l'ordre de la largeur du graphe.

Pour matérialiser le parcours en largeur, on opère en repérant les sommets à visiter. Lorsqu'un sommet est découvert, il intègre l'ensemble des éléments à visiter, c'est-à-dire la file F. Lorsque le sommet a été traité, il quitte la file. Il est donc également nécessaire de garder la trace du passage sur un sommet afin de ne pas traiter plusieurs fois un même sommet : si un sommet a été visité alors il intègre l'ensemble des éléments visités.

Au fur et à mesure de sa progression, cet algorithme construit un arbre de parcours en largeur dans le graphe. La racine de cet arbre est l'origine du parcours. Comme un sommet de cet arbre n'est découvert qu'une fois, il a au plus un parent. L'algorithme 4 peut ne rien renvoyer et servir pour un traitement particulier sur chaque nœud. Il peut aussi renvoyer une trace du parcours dans l'arbre dans une structure de type liste (P) qui enregistre le parcours.

Algorithme 4 Parcours en largeur d'un graphe

```
1 : Fonction PARCOURS_EN_LARGEUR(G, s)                    ▷ s est un sommet de G
2 :     F ← une file FIFO vide                            ▷ F comme file FIFO
3 :     V ← ∅                                             ▷ V ensemble des sommets visités
4 :     P ← une liste vide                                ▷ P comme parcours
5 :     ENFILER(F,s)
6 :     AJOUTER(V,s)
7 :     tant que F n'est pas vide répéter
8 :         v ← DÉFILER(F)
9 :         AJOUTER(P, v)                                 ▷ ou traiter le sommet en place
10 :        pour chaque voisin x de v dans G répéter
11 :            si x ∉ V alors                            ▷ x n'a pas encore été découvert
12 :                AJOUTER(V, x)
13 :                ENFILER(F, x)
14 :    renvoyer P          ▷ Facultatif, on pourrait traiter chaque sommet v en place
```

1. First In Fisrt Out

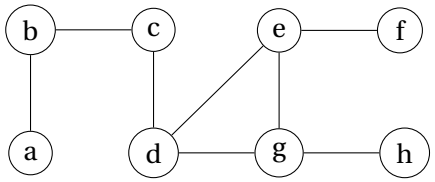


FIGURE 8.1 – Exemple de parcours en largeur au départ de a : a ⟶ b ⟶ c ⟶ d ⟶ e ⟶ g ⟶ f ⟶ h.

b Terminaison et correction du parcours en largeur

La terminaison du parcours en largeur peut être prouvée en considérant le variant de boucle $|F| + |\overline{V}|$, c'est à dire la somme des éléments présents dans la file et du nombre de nœuds non visités. En effet, au début de la boucle, si n est l'ordre du graphe, on a $|F| + |\overline{V}| = 1 + n - 1 = n$. Puis, à chaque tour de boucle on retire un élément de la file et on ajoute ses p voisins en même temps qu'on marque les p voisins comme visités. L'évolution du variant s'écrit :

$$|F| - 1 + p + |\overline{V}| - p = |F| + |\overline{V}| - 1 \tag{8.1}$$

À chaque tour de boucle, le variant décroit donc strictement de un et atteint nécessairement zéro au bout d'un certain nombre de tours. Lorsque le variant vaut zéro $|F| + |\overline{V}| = 0$, on a donc $|F| = 0$ et $|\overline{V}| = 0$. La file est nécessairement vide et tous les nœuds ont été visités. L'algorithme se termine alors. La structure de donnée file permet de garantir la correction du parcours en largeur d'abord.

Parcourir un graphe, à partir d'un sommet de départ s, cela veut dire trouver un chemin partant de s vers tous les sommets du graphe[2]. On remarque que tous les sommets du graphe sont à un moment ou un autre de l'algorithme **visités** et admis dans l'ensemble V : ceci vient du fait qu'on procède de proche en proche en ajoutant tous les voisins d'un sommet, sans distinction.

La correction peut se prouver en utilisant l'invariant de boucle $\mathcal{I}$: «Pour chaque sommet v ajouté à V et enfilé dans F, il existe un chemin de s à v.»

- Initialisation : à l'entrée de la boucle, s est ajouté à V et est présent dans la file F. Le chemin de s à s existe trivialement.
- Conservation : on suppose que l'invariant est vérifié jusqu'à une certaine itération. On cherche à montrer qu'il l'est toujours à la fin de l'itération suivante. Lors de l'exécution de cette itération, un sommet v est défilé. Ce sommet faisait déjà parti de V et par hypothèse, comme l'invariant était vérifié jusqu'à présent, il existe un chemin de s à v. Puis, les voisins de v sont ajoutés à V et enfilés. Comme ils sont voisins, il existe donc un chemin de v à ces sommets et donc il existe un chemin de s à ces sommets. À la fin de l'itération, l'invariant est donc vérifié.

2. On fait l'hypothèse que le graphe est connexe. S'il ne l'est pas, il suffit de recommencer la procédure avec un des sommets n'ayant pas été parcouru.

- Terminaison : à la fin de l'algorithme, il existe un chemin de s vers tous les sommets du graphe visités (ajoutés à V). Tous les sommets ont été parcourus.

c Complexité du parcours en largeur

La complexité de cet algorithme est lié, comme toujours, aux structures de données utilisées. Soit G un graphe d'ordre n et de taille m implémenté par une liste d'adjacence. On a choisi une file FIFO pour laquelle les opérations ENFILER et DÉFILER sont en $O(1)$. On parcours tous les sommets et chaque liste d'adjacence est parcourue une fois. Ces opérations sont donc en $O(n+m)$.

R Utiliser un liste d'adjacence pour implémenter le graphe est très important dans ce cas car cela permet d'accéder rapidement aux voisins : le coût de cette opération est l'accès à un élément de la liste. Si on avait utilisé une matrice, on aurait été obligé de rechercher les voisins à chaque étape.

R Si le graphe est complet, on note que la complexité $O(n+m)$ est en fait une complexité en $O(n^2)$ car $2|E| = n(n-1)$ d'après le lemme des poignées de main (cf t3).

R Si l'on avait choisi un type tableau dynamique (typiquement le type `list` en Python) au lieu d'une file FIFO pour implémenter F, alors l'opération DÉFILER ferait perdre du temps : en effet, le tableau serait réécrit dans sa totalité à chaque fois qu'une opération DÉFILER aurait lieu car on retirerait alors le premier élément du tableau et il faudrait donc allouer un autre espace mémoire à ce nouveau tableau. Une fois encore, le choix de la structure de données est important pour que l'algorithme soit efficace.

R L'ensemble V n'est pas indispensable dans l'algorithme 4. On pourrait se servir de la liste qui enregistre le parcours. Néanmoins, son utilisation permet de bien découpler la sortie de l'algorithme (le chemin C) de son fonctionnement interne et ainsi de prouver la terminaison.

d Parcours en profondeur

Parcourir en profondeur un graphe signifie qu'on cherche à emprunter d'abord les arêtes du premier sommet trouvé avant de parcourir les voisins de ce sommet et le reste du graphe.

Vocabulary 10 — Depth First Search ⇜⇝ Parcours en profondeur

Le parcours en profondeur d'un graphe s'exprime naturellement récursivement (cf. algorithme 5). Il peut également s'exprimer de manière itérative (cf. algorithme 6) en utilisant une pile P afin gérer la découverte des voisins dans l'ordre de la profondeur du graphe.

La complexité de cet algorithme est lié, comme toujours, aux structures de données utilisées. Soit G un graphe d'ordre n et de taille m implémenté par une liste d'adjacence. On a choisi

Algorithme 5 Parcours en profondeur d'un graphe (version récursive)

```
1 : Fonction REC_PARCOURS_EN_PROFONDEUR(G, s, V)        ▷ s est un sommet de G
2 :     AJOUTER(V, s)                                    ▷ s est marqué visité
3 :     pour chaque voisin x de s dans G répéter
4 :         si x ∉ V alors                               ▷ x n'a pas encore été découvert
5 :             REC_PARCOURS_EN_PROFONDEUR(G, x, V)
```

Algorithme 6 Parcours en profondeur d'un graphe (version itérative)

```
1 : Fonction PARCOURS_EN_PROFONDEUR(G, s)                ▷ s est un sommet de G
2 :     P ← une file vide                                ▷ P comme pile
3 :     V ← un ensemble vide                             ▷ V comme visités
4 :     C ← un liste vide                                ▷ C pour le parcours
5 :     EMPILER(P,s)
6 :     tant que P n'est pas vide répéter
7 :         v ← DÉPILER(P)
8 :         AJOUTER(C, v)
9 :         si v ∉ V alors                               ▷ v n'a pas encore été découvert
10 :            AJOUTER(V, x)
11 :            pour chaque voisin x de v dans G répéter
12 :                EMPILER(P, x)
13 :    renvoyer C          ▷ Facultatif, on pourrait traiter chaque sommet v en place
```

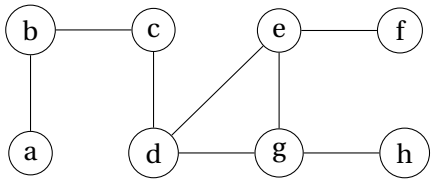


FIGURE 8.2 – Exemple de parcours en profondeur au départ de a : a ⟶ b ⟶ c ⟶ d ⟶ g ⟶ h ⟶ e ⟶ f

une pile LIFO pour laquelle les opérations EMPILER et DÉPILER sont en $O(1)$. On parcours tous les sommets et chaque liste d'adjacence est parcourue une fois. Ces opérations sont donc en $O(n+m)$.

B Trouver un chemin dans un graphe

On peut modifier l'algorithme 4 de parcours en largeur d'un graphe pour trouver un chemin reliant un sommet à un autre et connaître la longueur de la chaîne qui relie ces deux sommets. Il suffit pour cela de :

- garder la trace du prédécesseur (parent) du sommet visité sur le chemin,

- sortir du parcours dès qu'on a trouvé le sommet cherché (early exit),
- calculer le coût du chemin associé.

Le résultat est l'algorithme 7. Opérer cette recherche dans un graphe ainsi revient à chercher dans toutes les directions, c'est à dire sans tenir compte des distances déjà parcourues.

Algorithme 7 Longueur d'une chaîne via un parcours en largeur d'un graphe pondéré

```
1 : Fonction CD_PEL(G, a, b)                ▷ Trouver un chemin de a à b et la distance associée
2 :     F ← une file FIFO vide                                          ▷ F comme file FIFO
3 :     V ← un ensemble vide                                            ▷ V comme visités
4 :     P ← un dictionnaire vide                                        ▷ P comme parent
5 :     ENFILER(F,s)
6 :     AJOUTER(V,s)
7 :     tant que F n'est pas vide répéter
8 :         v ← DÉFILER(F)
9 :         si v est le sommet b alors                                  ▷ Objectif atteint, early exit
10 :            sortir de la boucle
11 :        pour chaque voisin x de v dans G répéter
12 :            si x ∉ V alors                                          ▷ x n'a pas encore été découvert
13 :                AJOUTER(V, x)
14 :                ENFILER(F, x)
15 :                P[x] ← v                                            ▷ Garder la trace du parent
16 :    c ← 0                                                           ▷ Coût du chemin
17 :    s ← b
18 :    tant que s est différent de a répéter
19 :        c ← c + w(s, P[s])                  ▷ w est la fonction de valuation du graphe G
20 :        sommet ← P[s]                                               ▷ On remonte à l'origine du chemin
21 :    renvoyer c
```

R Il faut noter néanmoins que le chemin trouvé et la distance associée issue de l'algorithme 7 n'est pas nécessairement la meilleure, notamment car on ne tient pas compte de la distance parcourue jusqu'au sommet recherché.

R Si le graphe n'est pas pondéré, cet algorithme fonctionne néanmoins, il suffit de compter le nombre de sauts pour évaluer la distance (la fonction de valuation vaut toujours 1).

C Plus courts chemins dans les graphes pondérés

Ⓡ Un graphe non pondéré peut-être vu comme un graphe pondéré dont la fonction de valuation vaut toujours 1. La distance entre deux sommets peut alors être interprétée comme le nombre de sauts nécessaires pour atteindre un sommet.

Théorème 16 — Existence d'un plus court chemin. Dans un graphe pondéré **sans pondérations négatives**, il existe toujours un plus court chemin.

Démonstration. Un graphe pondéré possède un nombre fini de sommets et d'arêtes (cf. définition 56). Il existe donc un nombre fini de chaînes entre les sommets du graphe. Comme les valuations du graphe ne sont pas négatives, c'est à dire que $\forall e \in E, w(e) \geqslant 0$, l'ensemble des longueurs de ces chaînes est une partie non vide de $\mathbb{N}$: elle possède donc un minimum. Parmi ces chaînes, il en existe donc nécessairement une dont la longueur est la plus petite, le plus court chemin. ■

■ **Définition 108 — Plus court chemin entre deux sommets d'un graphe.** Le plus court chemin entre deux sommets a et b d'un graphe G est une chaîne $\mathcal{C}_{ab}$ qui relie les deux sommets a et b et :

- qui comporte un minimum d'arêtes si G est un graphe non pondéré,
- dont le poids cumulé est le plus faible, c'est à dire $\min\limits_{\mathcal{C}_{ab} \in G} \left(\sum\limits_{e \in \mathcal{C}_{ab}} w(e) \right)$, dans le cas d'un graphe pondéré de fonction de valuation w.

■ **Définition 109 — Distance entre deux sommets.** La distance entre deux sommets d'un graphe est la longueur d'un plus court chemin entre ces deux sommets. Pour deux sommets a et b, on la note δ_{ab}. On a enfin :

$$\delta_{ab} = \min_{\mathcal{C}_{ab} \in G} \left(\sum_{e \in \mathcal{C}_{ab}} w(e) \right) \tag{8.2}$$

On se propose maintenant d'étudier les algorithmes les plus célèbres qui illustrent, dans différentes configurations, le concept de plus court chemin dans un graphe. La majorité de ces algorithmes reposent sur le principe d'optimalité de Bellman et la programmation dynamique et qui a déjà été énoncé dans le théorème 15. On peut formuler ce principe ainsi : **toute sous-chaîne entre p et q d'un plus court chemin entre a et b est un plus court chemin entre p et q.**

a Algorithme de Dijkstra

L'algorithme de Dijkstra[3] [4] s'applique à des **graphes pondérés $G = (V, E, w)$ dont la valuation est positive**, c'est à dire que $\forall e \in E, w(e) \geqslant 0$. C'est un algorithme glouton optimal (cf.

3. à prononcer "Daillekstra"

informatique commune) qui trouve les plus courts chemins entre un sommet particulier $a \in V$ et tous les autres sommets d'un graphe. Pour cela, l'algorithme classe les différents sommets par ordre croissant de leur distance minimale au sommet de départ. Dans ce but, il **parcourt en largeur le graphe en choisissant les voisins les plus proches en premier**.

Algorithme 8 Algorithme de Dijkstra, plus courts chemins à partir d'un sommet donné

```
1 : Fonction DIJKSTRA(G = (V, E, w), a)        ▷ Trouver les plus courts chemins à partir de a ∈ V
2 :     Δ ← a                                  ▷ Δ est l'ensemble des sommets dont on connaît la distance à a
3 :     Π ← ∅                                  ▷ Π[s] est le parent de s dans le plus court chemin de a à s
4 :     d ← ∅                                  ▷ l'ensemble des distances au sommet a
5 :     ∀ s ∈ V, d[s] ← w(a, s)                ▷ w(a, s) = +∞ si s n'est pas voisin de a, 0 si s = a
6 :     tant que Δ̄ n'est pas vide répéter      ▷ Δ̄ : sommets dont la distance n'est pas connue
7 :         Choisir u dans Δ̄ tel que d[u] = min(d[v], v ∈ Δ̄)
8 :         Δ = Δ ∪ {u}                        ▷ On prend la plus courte distance à a dans Δ̄
9 :         pour x ∈ Δ̄ répéter                 ▷ Ou bien x ∈ N_G(u) ∩ Δ̄, pour tous les voisins de u dans Δ̄
10 :            si d[x] > d[u] + w(u, x) alors
11 :                d[x] ← d[u] + w(u, x)      ▷ Mises à jour des distances des voisins
12 :                Π[x] ← u                   ▷ Pour garder la tracer du chemin le plus court
13 :    renvoyer d, Π
```

R Il faut remarquer que les boucles imbriquées de cet algorithme peuvent être comprises comme deux étapes successives de la manière suivante :

1. On choisit un nouveau sommet u de G à chaque tour de boucle *tant que* qui est tel que $d[u]$ est la plus petite des valeurs accessibles dans $\bar{\Delta}$. **C'est le voisin d'un sommet de $\bar{\Delta}$ le plus proche de a.** Ce sommet u est alors inséré dans l'ensemble Δ : c'est la **phase de transfert** de u de $\bar{\Delta}$ à Δ.
2. Lors de la boucle *pour*, on met à jour les distances des voisins de u qui n'ont pas encore été découverts. En effet, si x n'est pas un voisin de u, alors il n'existe par d'arête entre u et x et $w(u,x) = +\infty$. La mise à jour de la distance n'a donc pas lieu, on n'a pas trouvé une meilleure distance à x. C'est la **phase de mise à jour des distances** des voisins de u.

L'algoritme de Dijkstra procède donc de proche en proche.

■ **Exemple 35 — Application de l'algorithme de Dijkstra.** On se propose d'appliquer l'algorithme 8 au graphe représenté sur la figure 8.3. Le tableau 8.1 représente les distances successivement trouvées à chaque tour de boucle *tant que* de l'algorithme. En rouge figurent les distances les plus courtes à a à chaque tour. On observe également que certaines distances sont mises à jour sans pour autant que le sommet soit sélectionné au tour suivant.

À la fin de l'algorithme, on note donc que les distances les plus courtes de a à b, c, d, e, f sont $[5, 1, 8, 3, 6]$. Le chemin le plus court de a à b est donc $a \to c \to e \to b$. Le plus court de a à f est $a \to c \to e \to f$. C'est la structure de données Π qui garde en mémoire le prédécesseur (parent) d'un sommet sur le chemin le plus court qui permettra de reconstituer les chemins.

Δ	a	b	c	d	e	f	$\bar{\Delta}$
{}	0	7	1	$+\infty$	$+\infty$	$+\infty$	$\{a,b,c,d,e,f\}$
$\{a\}$	.	7	1	$+\infty$	$+\infty$	$+\infty$	$\{b,c,d,e,f\}$
$\{a,c\}$	.	6	.	$+\infty$	3	8	$\{b,d,e,f\}$
$\{a,c,e\}$	.	5	.	8	.	6	$\{b,d,f\}$
$\{a,c,e,b\}$	.	.	.	8	.	6	$\{d,f\}$
$\{a,c,e,b,f\}$	.	.	.	8	.	.	$\{d\}$
$\{a,c,e,b,f,d\}$	.	.	.	.	.	.	{}

TABLE 8.1 – Tableau d des distances au sommet a successivement trouvées au cours de l'algorithme de Dijkstra appliqué au graphe de la figure 8.3

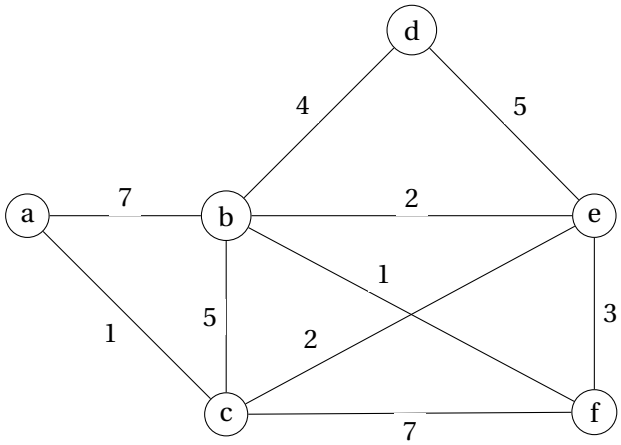


FIGURE 8.3 – Graphe pondéré à valeurs positives pour l'application de l'algorithme de Dijsktra.

Théorème 17 — L'algorithme de Dijkstra se termine et est correct.

Démonstration. Correction de l'algorithme : à chaque étape de cet algorithme, on peut distinguer deux ensembles de sommets : l'ensemble Δ est constitué des éléments dont on connaît la distance la plus courte à a et l'ensemble complémentaire $\bar{\Delta}$ qui contient les autres sommets.

D'après le principe d'optimalité, tout chemin plus court vers un sommet de $\bar{\Delta}$ passera nécessairement par un sommet de Δ. Ceci s'écrit :

$$\forall u \in \bar{\Delta}, d[u] = \min\left(d[v] + w[v,u], v \in \Delta\right) \tag{8.3}$$

On souhaite montrer qu'à la fin de chaque tour de boucle tant que (lignes 6-12), d contient les distances les plus courtes vers tous les sommets de Δ. On peut formuler cet invariant de boucle.

$\mathcal{I}$: à chaque fin de tour de boucle on a

$$\forall u \in \Delta, d[u] = \delta_{au} \tag{8.4}$$
$$\forall u \in \bar{\Delta}, d[u] = \min(d[v] + w[v,u], v \in \Delta) \tag{8.5}$$

À l'entrée de la boucle, l'ensemble Δ ne contient que le sommet de départ a. On a $d[a] = 0$, ce qui est la distance minimale. Pour les autres sommets de $\bar{\Delta}$, d contient :

- une valeur infinie si ce sommet n'est pas un voisin de a, ce qui, à cette étape de l'algorithme est le mieux qu'on puisse trouver,
- le poids de l'arête venant de a s'il s'agit d'un voisin, ce qui, à cette étape de l'algorithme est le mieux que l'on puisse trouver également.

On peut donc affirmer que d contient les distances entre a et tous les sommets de Δ. L'invariant est vérifié à l'entrée de la boucle.

On se place maintenant à une étape quelconque de la boucle. Notre hypothèse $\mathcal{H}$ est que toutes les itérations précédentes sont correctes. À l'entrée de la boucle on sélectionne un sommet u, le premier de la file de priorités. Il nous alors montrer que $d[u] = \delta_{au}$.

u entre dans Δ, c'est à dire que $u \in \bar{\Delta}$ et $\forall v \in \bar{\Delta}, d[u] \leqslant d[v]$. Considérons un autre chemin de a à u passant par un sommet v de $\bar{\Delta}$. Comme on a $d[u] \leqslant d[v]$, cet autre chemin sera au moins aussi long que $d[u]$, sauf s'il existe des arêtes de poids négatif (ce qui n'est pas le cas).

Formellement, on peut écrire cela ainsi

$$\delta_{au} = \delta_{av} + \delta_{vu} \tag{8.6}$$
$$\delta_{au} \geqslant \delta_{av} \tag{8.7}$$

Par ailleurs, comme v appartient à $\bar{\Delta}$, il vérifie l'hypothèse d'induction. On a donc :

$$d[v] = \min(d[x] + w[x,v], x \in \Delta) \tag{8.8}$$
$$= \min(\delta_{ax} + w[x,v], x \in \Delta) \tag{8.9}$$
$$= \delta_{av} \tag{8.10}$$

la deuxième ligne étant obtenue grâce à l'hypothèse d'induction également.

$$d[u] \leqslant d[v] = \delta_{av} \tag{8.11}$$
$$\leqslant \delta_{av} \tag{8.12}$$
$$\leqslant \delta_{au} \tag{8.13}$$

Or, $d[u]$ ne peut pas être plus petit que la distance de a à u. On a donc finalement $d[u] = \delta_{au}$.

d contient donc les distances vers tous les sommets à la fin de l'exécution de l'algorithme.

Terminaison de l'algorithme : avant la boucle *tant que*, $\bar{\Delta}$ possède $n-1$ éléments, si $n \in \mathbb{N}^\star$ est l'ordre du graphe. À chaque tour de boucle *tant que*, l'ensemble $\bar{\Delta}$ décroît strictement d'un élément et atteint donc nécessairement zéro. Le cardinal de $\bar{\Delta}$ est donc un variant de boucle. L'algorithme se termine lorsque le cardinal de $\bar{\Delta}$ atteint zéro. ■

La complexité de l'algorithme de Dijsktra dépend de l'ordre n du graphe considéré et de sa taille m. La boucle *tant que* effectue exactement $n-1$ tours. La boucle *pour* effectue à chaque fois un nombre de tour égal au nombre d'arêtes non découvertes qui partent du sommet u considéré et vont vers un sommet voisin de $\bar{\Delta}$. On ne découvre une arête qu'une seule fois, puisque le sommet u est transféré dans Δ au début de la boucle. Au final, on exécute donc la mise à jour des distances un nombre de fois égal à la taille m du graphe, c'est à dire son nombre d'arêtes. En notant la complexité du transfert c_t et la complexité de la mise à jour des distances c_d et en déroulant la boucle *tant que*, on peut écrire :

$$C(n,m) = (n-1)c_t + mc_d \tag{8.14}$$

Les complexités c_d et c_t dépendent naturellement des structures de données utilisées pour implémenter l'algorithme.

Si on choisit une implémentation de d par un tableau, alors on a besoin de rechercher le minimum des distances pour effectuer le transfert : cela s'effectue au prix d'un tri du tableau au minimum en $c_t = O(n\log n)$. Un accès aux éléments du tableau pour la mise à jour est en $c_d = O(1)$. On a donc $C(n) = (n-1)O(n\log n) + mO(1) = O(n^2\log n)$.

Si d est implémentée par une file à priorités (un tas) comme le propose Johnson [8], alors on a $c_t = O(\log n)$ et $c_d = O(\log n)$. La complexité est alors en $C(n) = (n+m)\log n$. Cependant, pour que le tas soit une implémentation pertinente, il est nécessaire que $m = O(\frac{n^2}{\log n})$, c'est à dire que le graphe ne soit pas complet, voire un peu creux!

■ **Exemple 36 — Usage de l'algorithme de Dijkstra .** Le protocole de routage OSPF implémente l'algorithme de Dijsktra. C'est un protocole qui permet d'automatiser le routage sur les réseaux internes des opérateurs de télécommunication. Les routeurs sont les sommets du graphe et les liaisons réseaux les arêtes. La pondération associée à une liaison entre deux routeurs est calculée à partir des performances en termes de débit de la liaison. Plus liaison possède un débit élevé, plus la distance diminue.

OSPF est capable de relier des centaines de routeurs entre eux, chaque routeur relayant les paquets IP de proche en proche en utilisant le plus court chemin de son point de vue [a]. Le protocole garantit le routage des paquets par les plus courts chemins en temps réel. Chaque routeur calcule ses propres routes vers toutes les destinations, périodiquement. Si une liaison réseau s'effondre, le routeurs en sont informés et recalculent d'autres routes immédiatement. La puissance de calcul nécessaire pour exécuter l'algorithme sur un routeur, même dans le cas d'un réseau d'une centaine de routeur, est relativement faible car la plupart des réseaux de télécommunications sont des graphes relativement creux. Ce n'est pas rentable de créer des graphes de télécommunications complets, même si ce serait intéressant pour le consommateur et très robuste!

a. Cela fonctionne grâce au principe d'optimalité de Bellman!

b Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford calcule les plus courts chemins depuis un sommet de départ, comme l'algorithme de Dijkstra. Cependant, il s'applique à des **graphes pondérés et orientés dont les pondérations peuvent être négatives mais sans cycles de longueur négative**[2, 6, 12].

R Soit un graphe pondéré non orienté qui possède des poids négatifs. Comme les arêtes d'un graphe non orienté sont des cycles, on ne peut pas appliquer l'algorithme de Bellman-Ford. On lui préfèrera alors Dijkstra.

R Les poids négatifs peuvent représenter des transferts de flux (chaleur en chimie, argent en économie) et sont donc très courants.

Algorithme 9 Algorithme de Bellman-Ford, plus courts chemins à partir d'un sommet donné

```
1 : Fonction BELLMAN_FORD(G = (V, E, w), a)
2 :     Π ← un dictionnaire vide        ▷ Π[s] est le parent de s dans le plus court chemin de a à s
3 :     d ← ensemble des distances au sommet a
4 :     d[s] ← w(a, s)                  ▷ w(a, s) = +∞ si s n'est pas voisin de a, 0 si s = a
5 :     pour _ de 1 à |V| − 1 répéter                                  ▷ Répéter n − 1 fois
6 :         pour (u, v) = e ∈ E répéter                                ▷ Pour toutes les arêtes du graphe
7 :             si d[v] > d[u] + w(u, v) alors                         ▷ Si le chemin est plus court par là...
8 :                 d[v] ← d[u] + w(u, v)                              ▷ Mises à jour des distances des voisins
9 :                 Π[v] ← u                                           ▷ Pour garder la tracer du chemin
10 :    renvoyer d, Π
```

■ **Exemple 37 — Application de l'algorithme de Bellman-Ford.** On se propose d'appliquer l'algorithme 9 au graphe pondéré et orienté représenté sur la figure 8.4. On note qu'il contient une pondération négative de b à f mais pas de cycle à pondération négative. Le tableau 8.2 représente les distances successivement trouvées à chaque itération.

On observe que le chemin de a à f emprunte bien l'arc de pondération négative.

Il faut noter que l'algorithme a convergé avant la fin de l'itération dans cas. C'est un des axes d'amélioration de cet algorithme.

N° d'itération	a	b	c	d	e	f
1	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
2	0	5	1	8	3	6
3	0	5	1	8	3	4
4	0	5	1	8	3	4
5	0	5	1	8	3	4

TABLE 8.2 – Tableau d des distances au sommet a successivement trouvées au cours de l'algorithme de Bellman-Ford appliqué au graphe de la figure 8.4

La complexité de l'algorithme 9 est en $O(nm)$ si n est l'ordre du graphe et m sa taille.

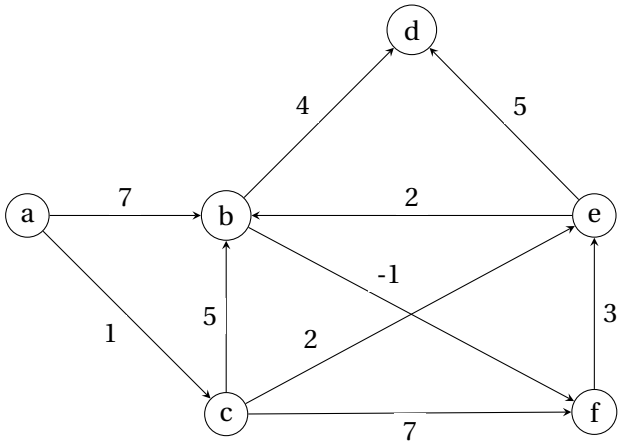


FIGURE 8.4 – Graphe pondéré et orienté à valeurs positives et négatives pour l'application de l'algorithme de Bellman-Ford.

■ **Exemple 38 — Protocole de routage RIP.** Le protocole de routage RIP utilise l'algorithme de Bellman-Ford pour trouver les plus cours chemins dans un réseau de routeur. Il est moins adapté que OSPF pour les grands réseaux.

c Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall [5, 14, 16] est l'application de la programmation dynamique à la recherche de **l'existence d'un chemin entre toutes les paires de sommets d'un graphe orienté et pondéré**. Les distances trouvées sont les plus courtes. Les pondérations du graphe peuvent être négatives mais on exclue tout circuit de poids strictement négatif.

Soit un graphe orienté et pondéré $G = (V, E, w)$. G peut être modélisé par une matrice d'adjacence M

$$\forall i, j \in [\![0, |V|-1]\!], M = \begin{cases} w(v_i, v_j) & \text{si } (v_i, v_j) \in E \\ +\infty & \text{si } (v_i, v_j) \notin E \\ 0 & \text{si } i = j \end{cases} \tag{8.15}$$

Un exemple de graphe associé à la matrice d'adjacence :

$$M_{\text{init}} = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & +\infty & 0 & +\infty \\ +\infty & 2 & 9 & 0 \end{pmatrix} \tag{8.16}$$

est donné sur la figure 8.5. Sur cet exemple, le chemin le plus court de v_4 à v_3 vaut 3 et passe par v_2.

Pour trouver le plus court chemin entre deux sommets, on essaye tous les chemins de toutes les longueurs possibles et on ne garde que les plus courts. Chaque étape p de l'algo-

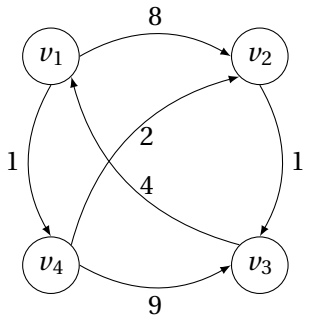


FIGURE 8.5 – Exemple de graphe orienté et pondéré pour expliquer le concept de matrice d'adjacence.

rithme de Floyd-Warshall est donc constitué d'un allongement **éventuel** du chemin par le sommet v_p. À l'étape p, on associe une matrice M_p qui contient la longueur des chemins les plus courts d'un sommet à un autre passant par des sommets de l'ensemble $\{v_0, v_1, \ldots, v_p\}$. On construit ainsi une suite de matrice finie $\left(M_p\right)_{p\in[\![0,n-1]\!]}$ et on initialise la matrice M avec avec M_{init}.

Supposons qu'on dispose de M_{p-1}. Considérons un chemin $\mathcal{C}$ entre v_i et v_j dont la longueur est minimale et dont les sommets intermédiaires sont dans $\{v_0, v_1, \ldots, v_{p-1}\}$, $p \leqslant n$. Pour un tel chemin :

- soit $\mathcal{C}$ passe par v_p. Dans ce cas, $\mathcal{C}$ est la réunion de deux chemins dont les sommets sont dans $\{v_0, v_1, \ldots, v_p\}$: celui de v_i à v_p et celui de v_p à v_j.
- soit $\mathcal{C}$ ne passe pas par v_p.

Entre ces deux chemins, on choisira le chemin le plus court.

Disposer d'une formule de récurrence entre M_p et M_{p-1} permettrait de montrer que le problème du plus court chemin entre deux sommets d'un graphe orienté et pondéré est à sous-structure optimale. On pourrait alors utiliser la programmation dynamique pour résoudre le problème. Or, on peut traduire notre explication ci-dessus par la relation de récurrence suivante :

$$\forall p \in [\![0, n-1]\!], \forall i, j \in [\![0, n-1]\!], M_p(i,j) = \min\left(M_{p-1}(i,j), M_{p-1}(i,p) + M_{p-1}(p,j)\right) \tag{8.17}$$

Pour $p = 0$, on pose $M_{-1} = M_{\text{init}}$.

L'algorithme de Floyd-Warshall 10 n'est que le calcul de la suite de ces matrices. C'est un bel exemple de programmation dynamique.

R Cet algorithme effectue le même raisonnement que Bellman-Ford mais avec un vision globale, à l'échelle du graphe tout entier, pas uniquement par rapport à un sommet de départ.

Algorithme 10 Algorithme de Floyd-Warshall, plus courts chemins entre toutes les paires de sommet

```
1 : Fonction FLOYD_WARSHALL(G = (V, E, w))
2 :     M ← la matrice d'adjacence de G
3 :     pour p de 0 à |V| − 1 répéter
4 :         pour i de 0 à |V| − 1 répéter
5 :             pour i de 0 à |V| − 1 répéter
6 :                 M(i, j) = min(M(i, j), M(i, p) + M(p, j))
7 :     renvoyer M
```

■ **Exemple 39 — Application de l'algorithme de Floyd-Warshall.** Si on applique l'algorithme au graphe de la figure 8.5, alors on obtient la série de matrices suivantes :

$$M_0 = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & +\infty & 0 & +\infty \\ +\infty & 2 & 9 & 0 \end{pmatrix} \tag{8.18}$$

$$M_1 = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & 12 & 0 & 5 \\ +\infty & 2 & 9 & 0 \end{pmatrix} \tag{8.19}$$

$$M_2 = \begin{pmatrix} 0 & 8 & 9 & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & 12 & 0 & 5 \\ +\infty & 2 & 3 & 0 \end{pmatrix} \tag{8.20}$$

$$M_3 = \begin{pmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{pmatrix} \tag{8.21}$$

$$M_4 = \begin{pmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{pmatrix} \tag{8.22}$$

d A*

L'algorithme A* [4] est un algorithme couteau suisse qui peut être considéré comme un algorithme de Dijkstra muni d'une heuristique : là où Dijkstra ne tient compte que du coût du chemin déjà parcouru, A* considère ce coût et une heuristique qui l'informe sur le reste du chemin à parcourir. Il faut bien remarquer que le chemin qu'il reste à parcourir n'est pas nécessai-

4. prononcer *A étoile* ou *A star*

rement déjà exploré : parfois il est même impossible d'explorer tout le graphe. Si l'heuristique pour évaluer le reste du chemin à parcourir est bien choisie, alors A* converge aussi vite voire plus vite que Dijkstra[15].

■ **Définition 110 — Heuristique admissible.** Une heuristique $\mathcal{H}$ est admissible si pour tout sommet du graphe, $\mathcal{H}(s)$ est une borne inférieure de la plus courte distance séparant le sommet de départ du sommet d'arrivée.

■ **Définition 111 — Heuristique cohérente.** Une heuristique $\mathcal{H}$ est cohérente si pour tout arête (s,p) du graphe $G=(V,E,w)$, $\mathcal{H}(s) \leqslant \mathcal{H}(p) + w(s,p)$.

■ **Définition 112 — Heuristique monotone.** Une heuristique $\mathcal{H}$ est monotone si l'estimation du coût **total** du chemin ne décroît pas lors du passage d'un sommet à ses successeurs. Pour un chemin $(s_0, s_1, \ldots, s_n)$, on $\forall 0 \leqslant i < j \leqslant n, c(s_j) \geqslant c(s_i)$.

Soit $G=(V,E,w)$ un graphe orienté. Soit d la fonction de distance utilisée par l'algorithme de Dijkstra (cf. algorithme 8). A*, muni d'une fonction h permettant d'évaluer l'heuristique, calcule alors le coût total pour aller jusqu'à un sommet p comme suit :

$$c(p) = d(p) + h(p) \tag{8.23}$$

Le coût obtenu n'est pas nécessairement optimal, il dépend de l'heuristique.

Supposons que l'on cherche le chemin le plus court entre les sommets s_0 et p. Supposons que l'on connaisse un chemin optimal entre s_0 et un sommet s. Alors on peut écrire que le coût total vers le sommet p vaut :

$$\begin{aligned} c(p) =& d(p) + h(p) && (8.24)\\ =& d(s) + w(s,p) + h(p) && (8.25)\\ =& d(s) + h(s) + w(s,p) - h(s) + h(p) && (8.26)\\ =& c(s) + w(s,p) - h(s) + h(p) && (8.27) \end{aligned}$$

Ainsi, on peut voir l'algorithme A* comme un algorithme de Dijkstra muni :

- de la distance $\tilde{d} = c$,
- et de la pondération $\tilde{w}(s,p) = w(s,p) - h(s) + h(p)$.

L'algorithme 11 donne le détail de la procédure à suivre.

D Arbres recouvrants

Les arbres recouvrants sont des éléments essentiels de l'industrie, notamment du point de vue de l'efficacité, de la robustesse et de l'optimisation. En ce sens, ils sont également indispensables à toute vision durable de notre développement. L'idée d'un arbre recouvrant est de ne sélectionner que certaines arêtes dans un graphe pondéré afin de garantir un service optimale (en fonction des pondérations). Les arbres recouvrants sont donc utilisés dans les domaines

Algorithme 11 A*

```
1 : Fonction ASTAR(G = (V, E, w), a)                          ▷ Sommet de départ a
2 :     Δ ← a
3 :     Π ←
4 :     d̃ ← l'ensemble des distances au sommet a
5 :     ∀s ∈ V, d̃[s] ← w̃(a, s)          ▷ Le graphe est partiel, l'heuristique fait le reste
6 :     tant que Δ̄ n'est pas vide répéter      ▷ Δ̄ : sommets dont la distance n'est pas connue
7 :         Choisir u dans Δ̄ tel que d̃[u] = min(d̃[v], v ∈ Δ̄)
8 :         Δ = Δ ∪ {u}
9 :         pour x ∈ Δ̄ répéter      ▷ Ou bien x ∈ N_G(u) ∩ Δ̄, pour tous les voisins de u dans Δ̄
10 :            si d̃[x] > d̃[u] + w̃(u, x) alors
11 :                d̃[x] ← d̃[u] + w̃(u, x)
12 :                Π[x] ← u                    ▷ Pour garder la tracer du chemin le plus court
13 :    renvoyer d̃, Π
```

des réseaux d'énergie, des télécommunications, des réseaux de fluides mais également en intelligence artificielle et en électronique. Les deux algorithmes phares pour construire des arbres recouvrants sont l'algorithme de Kruskal [9] et l'algorithme de Prim [13].

a Algorithme de Prim

L'algorithme de Prim est un algorithme glouton optimal qui s'applique aux graphes pondérés connexes. Pour construire l'arbre, l'algorithme part d'un sommet et fait croitre l'arbre en choisissant un sommet dont la distance est la plus faible et n'appartenant pas à l'arbre, garantissant ainsi l'absence de cycle.

Algorithme 12 Algorithme de Prim, arbre recouvrant

```
1 : Fonction PRIM(G = (V, E, w))
2 :     T ← ∅                          ▷ la sortie : l'ensemble des arêtes de l'arbre recouvrant
3 :     S ← s un sommet quelconque de V
4 :     tant que S ≠ V répéter
5 :         (u, v) ← min(w(u, v), u ∈ S, v ∈ E)                          ▷ Choix glouton !
6 :         S ← S ∪ {v}
7 :         T ← T ∪ {(u, v)}
8 :     renvoyer T
```

La complexité de cet algorithme, si l'on utilise un tas binaire, est en $O(m \log n)$.

b Algorithme de Kruskal

L'algorithme de Kruskal (cf. algorithme 13) est un algorithme glouton optimal qui s'applique aux graphes pondérés. Le graphe peut ne pas être connexe et dans ce cas on obtient

un forêt d'arbres recouvrants. Pour construire la forêt et effectuer un choix d'arête, il ordonne les arêtes du graphe par ordre de pondération croissante.

Algorithme 13 Algorithme de Kruskal, arbre recouvrant

```
1 : Fonction KRUSKAL(G = (V, E, w))
2 :     T ← ∅                                  ▷ la sortie : l'ensemble des arêtes de l'arbre recouvrant
3 :     pour k de 1 à |E| répéter
4 :         e ← l'arête de pondération la plus faible de E          ▷ Choix glouton !
5 :         si (S, T ∪ {e}, w) est un graphe acyclique alors
6 :             T ← T ∪ {e}
7 :     renvoyer T
```

Si on utilise une structure de tas ainsi La complexité de cet algorithme est en $O(m \log m)$ si $m = |E|$ est le nombre d'arêtes du graphe.

E Tri topologique d'un graphe orienté

a Ordre dans un graphe orienté acyclique

Dans un graphe orienté (cf. définition 62) acyclique, les arcs définissent un **ordre partiel**, le sommet à l'origine de l'arc pouvant être considéré comme le prédécesseur du sommet à l'extrémité de l'arc. Par exemple, sur la figure 8.6, a et b sont des prédécesseurs de d et e est un prédécesseur de g. Mais ces arcs ne disent rien de l'ordre entre e et h, l'ordre n'est pas total.

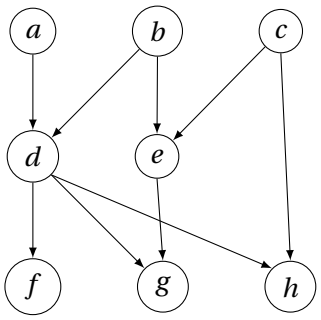


FIGURE 8.6 – Exemple de graphe orienté acyclique

L'algorithme de tri topologique permet de créer un ordre total $\preceq$ sur un graphe orienté acyclique. Formulé mathématiquement :

$$\forall (v,u) \in V^2, (v,u) \in E \Longrightarrow v \preceq u \tag{8.28}$$

Sur l'exemple de la figure 6.4, plusieurs ordre topologiques sont possibles. Par exemple :

- a,b,c,d,e,f,g,h
- a,b,d,f,c,h,e,g

b Tri topologique et détection de cycles dans un graphe orienté

L'algorithme de tri topologique permet de construire un ordre dans un graphe orienté acyclique. C'est en fait un parcours en profondeur du graphe qui construit une pile en ajoutant le concept de date à chaque sommet : un date de début qui correspond au début du traitement du sommets et une date de fin qui correspond à la fin du traitement du sommet par l'algorithme. La pile contient à la fin les sommets dans un ordre topologique, les sommets par ordre de date de fin de traitement.

Au cours du parcours en profondeur, un sommet passe tout d'abord de l'ensemble des sommets non traités à l'ensemble des sommets en cours de traitement (date de début). Puis, lorsque la descente est finie (plus aucun arc ne sort du sommet courant), le sommet passe de l'ensemble en cours de traitement à l'ensemble des sommets traités (date de fin).

Algorithme 14 Algorithme de tri topologique

```
1 : Fonction TOPO_SORT(G = (V, E))
2 :     pile ← une pile vide                          ▷ Contiendra les sommets dans l'ordre topologique
3 :     états ← un tableau des états des sommets   ▷ pas traité, en cours de traitement ou traité
4 :     dates ← un tableau des dates associées aux sommets
5 :     Les cases du tableau états sont initialisées à «pas traité»
6 :     Les cases du tableau dates sont initialisées à max_int  ▷ Date inconnue représentée par
    max_int
7 :     pour chaque sommet v de G répéter
8 :         si v n'est pas traité alors
9 :             TOPO_DFS(G = (V, E), pile, états, dates, v, 0)
10 :    renvoyer (pile, dates)
11 :
12 : Fonction TOPO_DFS(G = (V, E), pile, états, dates, v, date)
13 :    états[v] ← «en cours de traitement»
14 :    dates[v] ← date                          ▷ Au début de l'exploration la date de v vaut date
15 :    pour chaque voisin u de v répéter
16 :        si u n'est pas traité alors TOPO_DFS(G = (V, E), pile, états, dates, u, (date + 1))
17 :    états[v] ← «traité»
18 :    dates[v]+ = 1                          ▷ Pour distinguer le début de la fin de l'exploration
19 :    EMPILER(v, pile)
```

F Composantes fortement connexes d'un graphe orienté et 2-SAT

■ **Définition 113 — Composante fortement connexe d'un graphe orienté** $G = (V, E)$**.** Une composante fortement connexe d'un graphe orienté G est un sous-ensemble S de ses sommets, maximal au sens de l'inclusion, tel que pour tout couple de sommets $(s, t) \in S$ il existe un chemin de s à t dans G.

Le calcul des composantes connexes d'un graphe est par exemple utilisé pour résoudre le problème 2-SAT. Dans le cadre de ce problème, on dispose d'une formule logique sous la forme conjonctive normale et chaque clause comporte deux variables. L'idée est de construire un graphe à partir de la formule de cette formule F. Supposons qu'elle soit constituée de m clauses et n variables $(v_1, v_2, \ldots, v_n)$. On élabore alors un graphe $G = (V, E)$ à $2n$ sommets et $2m$ arêtes. Les sommets représentent les n variables v_i ainsi que leur négation $\neg v_i$. Les arêtes sont construites de la manière suivante : on transforme chaque clause de F de la forme $v_i \vee v_j$ en deux implications $\neg v_1 \implies v_2$ ou $\neg v_2 \implies v_1$. Cette transformation utilise le fait que la formule $a \implies b$ est équivalent à $\neg a \vee b$.

Théorème 18 F n'est pas satisfaisable si et seulement s'il existe une composante fortement connexe contenant une variable v_i et sa négation $\neg v_i$.

Démonstration. ($\Longleftarrow$) S'il existe une composante fortement connexe contenant a et $\neg a$, alors cela signifie $F : (a \implies \neg a) \wedge (\neg a \implies a)$. Or cette formule n'est pas satisfaisable. En effet, si a est vrai alors $(a \implies \neg a)$ est faux, car du vrai on ne peut pas conclure le faux d'après la définition sémantique de l'implication. De même, si a est faux alors $(\neg a \implies a)$ est faux, pour la même raison. Dans tous les cas, la formule est fausse. F n'est pas satisfaisable.

($\Longrightarrow$), par contraposée. Supposons qu'il n'existe pas de composante fortement connexe contenant a et $\neg a$. Cela peut se traduire en la formule $F : (a \implies \neg a) \vee (\neg a \implies a)$: soit il n'existe aucun chemin de a à $\neg a$, soit il existe un chemin dans un seul sens, mais pas dans les deux. Cette formule F est toujours satisfaisable. En effet, si a est vrai, alors $\neg a \implies a$ est vraie, car *ex falso quodlibet*, et donc F est vraie. Si a est faux, alors $a \implies \neg a$ est vraie pour la même raison. Dans tous les cas, F est vraie. On peut également le montrer en remarquant que F s'écrit $(a \vee \neg a) \vee (\neg a \vee a) = a \vee \neg a$. F est donc satisfaisable. ■

On peut montrer que SAT-2 est un problème de décision polynomial. C'est une restriction à des clauses de deux variables du problème général SAT qui lui est NP-complet.

G Graphes bipartis et couplage maximum

Théorème 19 — Caractérisation des graphes bipartis. Un graphe est biparti si et seulement s'il ne possède aucun cycle de longueur impair.

Pour savoir si un graphe est biparti on pourrait donc rechercher les cycles et évaluer leurs longueurs. Il existe cependant une autre approche. Un graphe biparti est un graphe bicolorable comme le montre la figure 6.6.

a Couplage dans un graphe biparti

■ **Définition 114 — Couplage.** Un couplage Γ dans un graphe non orienté $G = (V, E)$ est un

ensemble d'arêtes deux à deux non adjacentes. Formellement :

$$\forall (e_1, e_2) \in E^2, e_1 \neq e_2 \Longrightarrow e_1 \cap e_2 = \emptyset \tag{8.29}$$

c'est à dire que les sommets de e_1 et e_2 ne sont pas les mêmes.

■ **Définition 115 — Sommets couplés, sommets exposés.** Un sommet est couplé s'il fait parti d'une arête de Γ. Un sommet est exposé s'il ne fait pas parti des arêtes de Γ, c'est-à-dire il n'est pas couplé.

■ **Définition 116 — Couplage maximal.** Un couplage maximal est tel que si on lui ajoute une arête, ce n'est plus un couplage. Il est donc maximal au sens de l'inclusion.

■ **Définition 117 — Couplage de cardinal maximum.** Un couplage de cardinal maximum est un couplage contenant le plus grand nombre d'arêtes possible.

■ **Exemple 40 — Affectation des cadeaux sous le sapin .** Au pied du sapin de Noël, un papa a disposé six cadeaux dont les paquets sont tous différents et numérotés de 0 à 5 [a]. Il a décidé que les cadeaux seraient répartis en fonction des paquets que les enfants préfèrent.

Ses cinq enfants expriment donc leurs préférences. Le papa pourra-t-il affecter un cadeau à chaque enfant et faire en sorte que ce cadeau soit un de leurs préférés ?

Évidemment la réponse à cette question dépend des préférences émises par les enfants. Supposons qu'ils se soient exprimés ainsi :

Alix 0,2

Brieuc 1,3,4,5

Céline 1,2

Dimitri 0,1,2

Enora 2

On peut représenter par un graphe biparti cette situation comme sur la figure 8.7. Dans ce cas précis, comme il y a quatre enfants qui ne veulent que trois les trois premiers cadeaux, il n'y a pas de solution. Mais si Enora avait choisit 4 et 5 ???

a. Ce papa est informaticien !

R Si les arêtes du graphe sont valuées (graphe pondéré), alors la recherche d'un couplage de cardinal maximum et de poids minimum dans un graphe biparti est en fait le problème de l'**affectation des ressources**. Ceci n'est pas au programme.

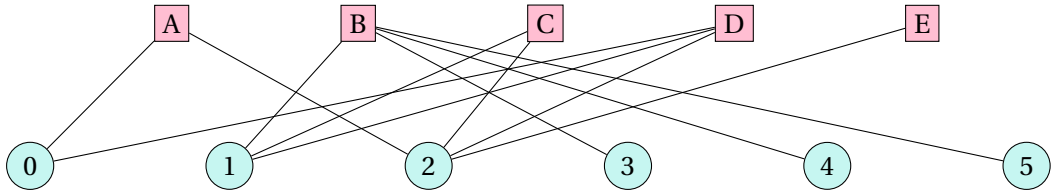


FIGURE 8.7 – Exemple de graphe biparti pour un problème d'affectation sans solution.

b Chemin augmentant

Pour résoudre le problème de trouver un couplage de cardinal maximum, on peut procéder en suivant l'algorithme 15. Il s'agit de construire un chemin augmentant pour atteindre un couplage de cardinal maximum.

■ **Définition 118 — Chemin alternant.** Une chemin alternant dans un graphe non orienté G et pour un couplage Γ est tel que les arêtes appartiennent successivement à Γ et $E \setminus \Gamma$.

■ **Définition 119 — Chemin augmentant.** Un chemin augmentant est un chemin alternant dont les extrémités sont des sommets exposés, c'est-à-dire qui n'appartiennent pas au couplage Γ.

La stratégie de l'algorithme de recherche d'un couplage de cardinal maximum est la suivante : à partir d'un couplage Γ, on construit un nouveau couplage de cardinal supérieur à l'aide d'un chemin augmentant comme le montre la figure 8.8.

Dans un graphe **biparti**, il est facile d'augmenter la taille d'un couplage jusqu'au cardinal maximum :

1. s'il existe deux sommets exposés reliés par une arête, il suffit d'ajouter cette arête au couplage. Puis, on appelle récursivement l'algorithme sur ce nouveau couplage.
2. sinon il faut trouver un chemin augmentant π dans le graphe.
   (a) s'il n'y en a pas, l'algorithme est terminé.
   (b) sinon on effectue la différence symétrique entre le couplage Γ et l'ensemble des arêtes du chemin augmentant π pour obtenir le nouveau couplage : $\Gamma \oplus \pi = \{e \in E, e \in \Gamma \setminus \pi \cup e \in \pi \setminus \Gamma\}$. Puis, on appelle récursivement l'algorithme avec ce nouveau couplage.

Il faut noter que le cardinal du couplage n'augmente pas nécessairement lorsqu'on effectue la différence symétrique mais il ne diminue pas.

Pour trouver un chemin augmentant dans un graphe biparti $G = ((U, D), E)$, on cherche le plus court chemin entre deux sommets exposés dans un graphe orienté auxiliaire G_o construit de la manière suivante :

1. toutes les arêtes de E qui n'appartiennent pas au couplage Γ sont orientés de U vers D.
2. toutes les arêtes de Γ sont orientées de D vers U.

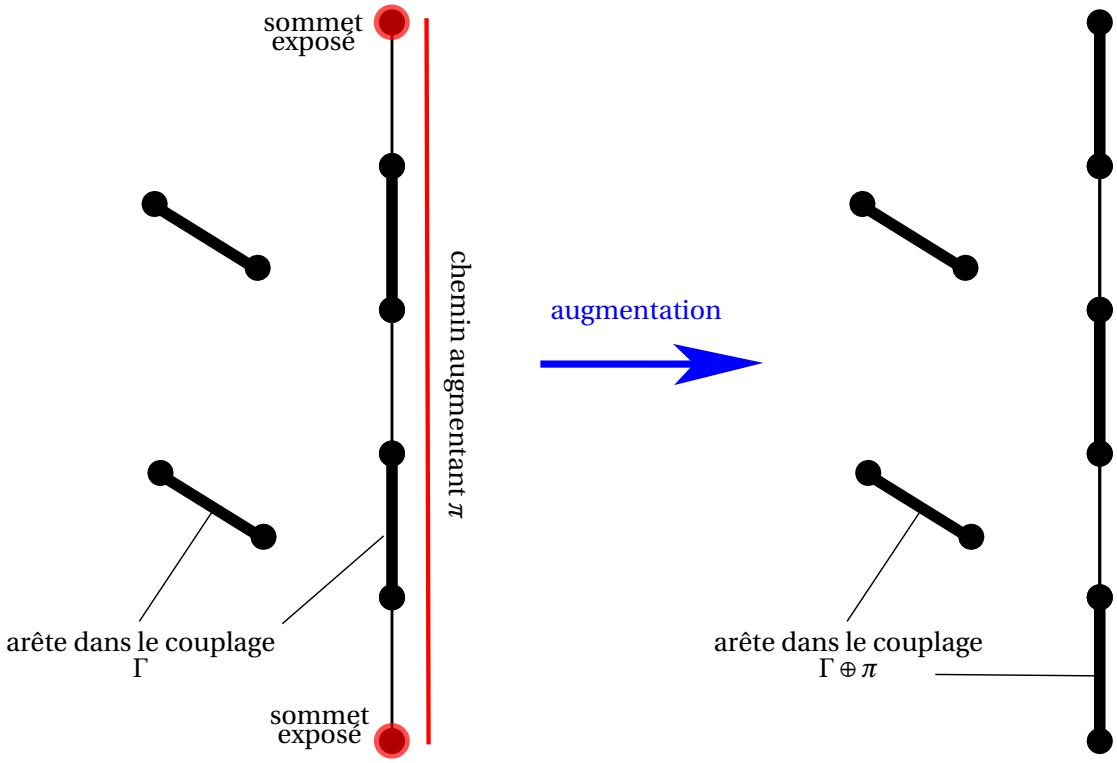
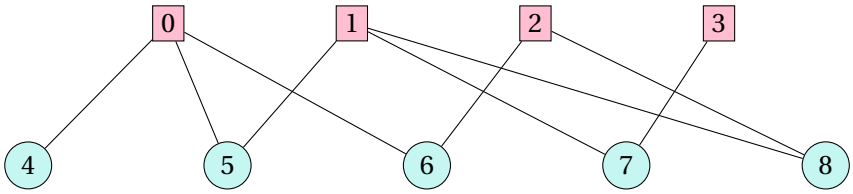


FIGURE 8.8 – Illustration de la construction d'un couplage de cardinal supérieur grâce à un chemin augmentant. (Source originale : Wikimedia Commons)

Le plus court chemin entre deux sommets exposés de G_o est nécessairement un chemin augmentant, son caractère alternant vient du fait que le graphe est biparti.

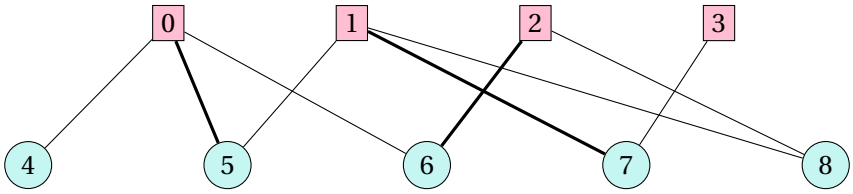
La figure 8.9 illustre les différentes étapes de l'algorithme. On vérifie bien que le résultat est correct : chaque enfant aura bien un cadeau qu'il avait mis sur sa liste.

Le graphe de départ de l'algorithme est le suivant :



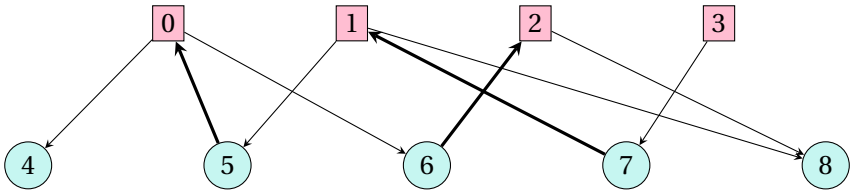
$\Gamma = \emptyset$

On effectue **trois appels récursifs** et, à chaque fois, on a trouvé une arête dont les sommets sont tous les deux exposés.



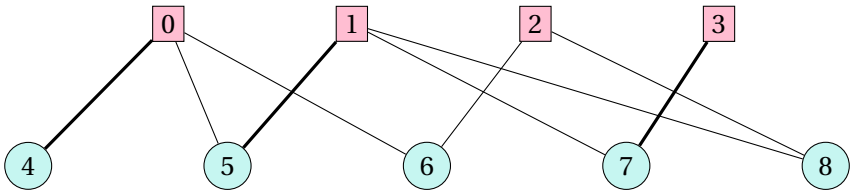
$\Gamma = \{(0,5),(1,7),(2,6)\}$

À ce stade de l'algorithme, aucun sommet exposé n'est relié par une arête à un autre sommet exposé. Donc, on construit le graphe G_o d'après le couplage Γ. On trouve le chemin le plus court entre les deux premiers sommets exposés 3 et 4 : π.



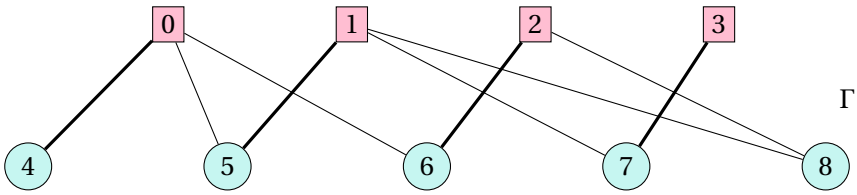
$\Gamma = \{(0,5),(1,7),(2,6)\}$
$\pi = \{3,7,1,5,0,4\}$

On en déduit un nouveau couplage $\Gamma = \Gamma \oplus \pi$:



$\Gamma = \{(0,4),(1,5),(3,7)\}$

On effectue **un appel récursif** et on trouve une arête dont les sommets sont tous les deux exposés : $(2,6)$. On effectue un dernier appel récursif et l'algorithme se termine car un seul sommet est non couplé.



$\Gamma = \{(0,4),(1,5),(3,7),(2,6)\}$

FIGURE 8.9 – Étapes de l'algorithme de recherche d'un couplage de cardinal maximum

Algorithme 15 Recherche d'un couplage de cardinal maximum

Entrée : un graphe biparti $G = ((U, D), E)$
Entrée : un couplage Γ initialement vide
Entrée : F_U, l'ensemble de sommets exposés de U initialement U
Entrée : F_D, l'ensemble de sommets exposés de D initialement D

```
1 : Fonction CHEMIN_AUGMENTANT(G = (V = (U, D), E), Γ, F_U, F_D))  ▷ M est le couplage, vide initialement
2 :     si une arête e = (u, v) entre un sommet de F_U et un sommet de F_D existe alors
3 :         CHEMIN_AUGMENTANT(G, Γ ∪ {e}, F_U \ {u}, F_D \ {v})
4 :     sinon
5 :         Créer le graphe orienté G_o                ▷ ∀e ∈ E, e de U vers D si e ∉ Γ, l'inverse sinon
6 :         Calculer le plus court chemin π entre un sommet de F_U et un de F_D dans G_o
7 :         si un tel chemin π n'existe pas alors
8 :             renvoyer M
9 :         sinon
10 :            CHEMIN_AUGMENTANT(G, Γ ⊕ π, F_U \ {π_start}, F_D \ {π_end})
11 :                                ▷ π_start début du chemin π, π_end fin du chemin π et Γ ⊕ π = {e ∈ E, e ∈ Γ \ π ∪ e ∈ π \ Γ}
```

9

DES ARBRES AUX TAS

À la fin de ce chapitre, je sais :

- ☞ définir un tas-min et un tas-max
- ☞ expliquer l'algorithme du tri par tas
- ☞ utiliser un tas pour créer une file de priorité
- ☞ appliquer les files de priorités à l'algorithme de Dijsktra

A Des arbres

■ **Définition 120 — Arbre.** Un arbre est un graphe connexe, acyclique et enraciné.

Ⓡ La racine d'un arbre $\mathcal{A}$ est un sommet r particulier que l'on distingue : le couple (A, r) est un nommé arbre enraciné. On le représente un tel arbre verticalement avec la racine placée tout en haut comme sur la figure 9.1. Dans le cas d'un graphe orienté, la représentation verticale permet d'omettre les flèches.

Ⓡ On confondra par la suite les arbres enracinés et les arbres.

■ **Définition 121 — Nœuds.** Les nœuds d'un arbre sont les sommets du graphe associé. Un nœud qui n'a pas de fils est une feuille (ou nœud externe). S'il possède des descendants, on parle alors de nœud interne.

■ **Définition 122 — Descendants, père et fils.** Si une arête mène du nœud i au nœud j, on dit que i est le **père** de j et que j est le **fils** de i. On représente l'arbre de telle sorte que le père soit toujours au-dessus de ses fils.

■ **Définition 123 — Arité d'un nœud.** L'arité d'un nœud est le nombre de ses fils.

■ **Définition 124 — Feuille.** Un nœud d'arité nulle est appelé une feuille.

■ **Définition 125 — Profondeur d'un nœud.** La profondeur d'un nœud est le nombre d'arêtes qui le sépare de la racine.

■ **Définition 126 — Hauteur d'un arbre.** La hauteur d'un arbre est la plus grande profondeur d'une feuille de l'arbre.

■ **Définition 127 — Taille d'un arbre.** La taille d'un arbre est le nombre de ses nœuds.

R Attention, la taille d'un graphe est le nombre de ses arêtes... Un arbre possède toujours $n-1$ arêtes si sa taille est n.

■ **Définition 128 — Sous-arbre.** Chaque nœud d'un arbre $\mathcal{A}$ est la racine d'un arbre constitué de lui-même et de ses descendants : cette structure est appelée sous-arbre de l'arbre $\mathcal{A}$.

R La notion de sous-arbre montre qu'un arbre est une structure intrinsèquement récursive ce qui sera largement utilisé par la suite !

■ **Définition 129 — Arbre recouvrant.** Un arbre recouvrant d'un graphe G est un sous-graphe couvrant de G qui est un arbre.

ă

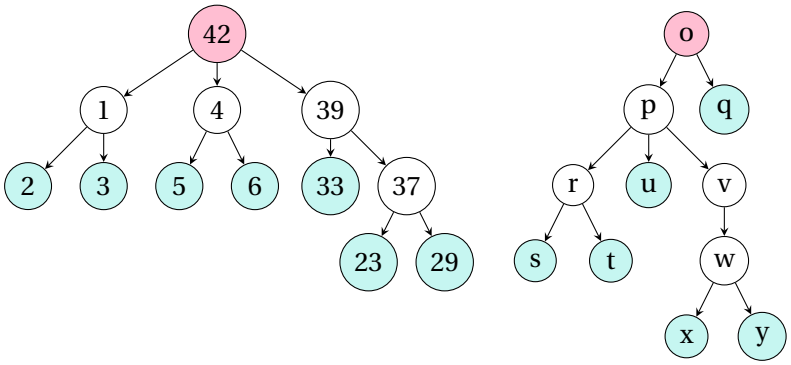


FIGURE 9.1 – Exemples d'arbres enracinés. Les racines des arbres sont en rouge, les feuilles en turquoise. Le tout forme une forêt.

B Arbres binaires

■ **Définition 130 — Arbre binaire.** Un arbre binaire est un arbre tels que tous les nœuds ont une arité inférieure ou égale à deux : chaque nœud possède au plus deux fils.

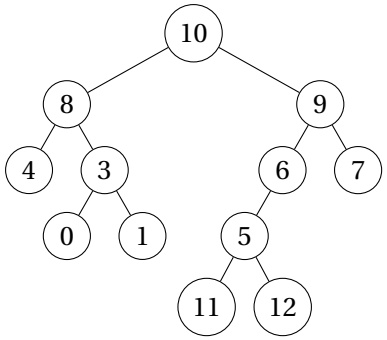


FIGURE 9.2 – Arbre binaire

■ **Définition 131 — Arbre binaire strict.** Un arbre binaire strict est un arbre dont tous les nœuds possèdent zéro ou deux fils.

■ **Définition 132 — Arbre binaire parfait.** Un arbre binaire parfait est un arbre dans lequel tous les niveaux sauf le dernier doivent être totalement remplis. Si le dernier n'est pas rempli totalement alors il doit être rempli de gauche à droite.

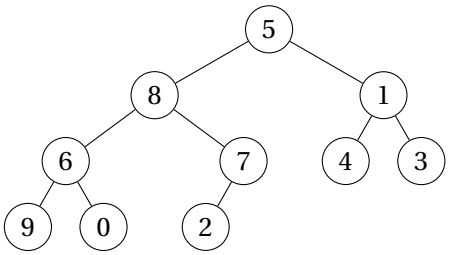


FIGURE 9.3 – Arbre binaire parfait

■ **Définition 133 — Arbre binaire équilibré.** Un arbre binaire est équilibré si sa hauteur est minimale, c'est à dire $h(a) = O(\log|a|)$.

R Un arbre parfait est un arbre équilibré.

Théorème 20 — La hauteur d'un arbre parfait de taille n vaut $\lfloor \log n \rfloor$. Soit a un arbre binaire parfait de taille n. Alors on a :

$$h(a) = \lfloor \log n \rfloor \tag{9.1}$$

Démonstration. Soit a un arbre binaire parfait de taille n. Comme a est parfait, on sait que tous les niveaux sauf le dernier sont remplis. Ainsi, il existe deux niveaux de profondeur $h(a)-1$ et $h(a)$. On peut encadrer le nombre de nœuds de a en remarquant que chaque niveau k possède 2^k nœuds, sauf le dernier. On a donc :

$$1+2+\ldots+2^{h(a)-1} < |a| \leqslant 1+2+\ldots+2^{h(a)} \tag{9.2}$$

$$\sum_{k=0}^{h(a)-1} 2^k < |a| \leqslant \sum_{k=0}^{h(a)} 2^k \tag{9.3}$$

$$2^{h(a)} - 1 < |a| \leqslant 2^{h(a)+1} - 1 \tag{9.4}$$

$$2^{h(a)} \leqslant |a| < 2^{h(a)+1} \tag{9.5}$$

On en conclut que $\lfloor \log_2 |a| \rfloor - 1 < h(a) \leqslant \lfloor \log_2 |a| \rfloor$ et donc que $h(a) = \lfloor \log_2(n) \rfloor$. ■

C Induction et arbre binaire

La plupart des caractéristiques et des résultats importants liés aux arbres binaires peuvent se démontrer par induction structurelle. Cette méthode est une généralisation des démonstrations par récurrences sur $\mathbb{N}$ pour un ensemble défini par induction.

■ **Définition 134 — Étiquette d'un nœud.** Une étiquette d'un nœud est une information portée au niveau d'un nœud.

■ **Définition 135 — Définition inductive d'un arbre binaire.** Soit E un ensemble d'étiquettes. L'ensemble $\mathcal{A}_E$ des arbres binaires étiquetés par E est défini inductivement par :

1. NIL est un arbre binaire appelé arbre vide (parfois noté $\circ$),
2. Si $e \in E$, $f_g \in \mathcal{A}$ et $f_d \in \mathcal{A}$ sont deux arbres binaires, alors $(f_g, x, f_d) \in \mathcal{A}_E$, c'est à dire que le triplet (f_g, x, f_d) est un arbre binaire étiqueté par E.

f_g et f_d sont respectivement appelés fils gauche et fils droit.

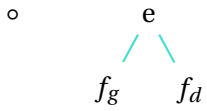


FIGURE 9.4 – Arbre binaire et définition inductive : arbre vide à gauche, arbre induit par e, f_g et f_d à droite

■ **Définition 136 — Démonstration par induction structurelle sur un arbre binaire.** Soit $\mathcal{P}(a)$ un prédicat exprimant une propriété sur un arbre a de $\mathcal{A}_E$, l'ensemble des arbres binaires étiquetés sur un ensemble E. On souhaite démonter cette propriété.

La démonstration par induction structurelle procède comme suit :

1. (**CAS DE BASE**) Montrer que $\mathcal{P}(\text{NIL})$ est vraie, c'est-à-dire que la propriété est vraie pour l'arbre vide,
2. (**PAS D'INDUCTION**) Soit $e \in E$ une étiquette et $f_g \in \mathcal{A}$ et $f_d \in \mathcal{A}$ deux arbres binaires pour lesquels $\mathcal{P}(f_g)$ et $\mathcal{P}(f_d)$ sont vraies. Montrer que $\mathcal{P}\big((f_g, e, f_d)\big)$ est vraie.
3. (**CONCLUSION**) Conclure que quelque soit $a \in \mathcal{A}$, $\mathcal{P}(a)$ est vraie.

■ **Définition 137 — Définition inductive d'une fonction à valeur dans $\mathcal{A}_E$.** On définit une fonction ϕ de $\mathcal{A}_E$ à valeur dans un ensemble $\mathcal{Y}$ par :

1. la donnée de la valeur de $\phi(\text{NIL})$,
2. en supposant connaître $e \in E$, $\phi(f_g)$ et $\phi(f_d)$ pour f_g et f_d dans $\mathcal{A}_E$, la définition de $\phi((f_g, e, f_d))$.

■ **Exemple 41 — Définition inductive de la hauteur d'un arbre.** Soit $a \in \mathcal{A}$ un arbre binaire. La hauteur $h(a)$ de a est donnée par :

1. $h(\text{NIL}) = 0$,
2. $h\big((f_g, e, f_d)\big) = 1 + \max\big(h(f_g), h(f_d)\big)$.

■ **Exemple 42 — Définition inductive de la taille d'un arbre.** Soit $a \in \mathcal{A}$ un arbre binaire. La taille $|a|$ de a est donnée par :

1. $|\text{NIL}| = 0$,
2. $|(f_g, e, f_d)| = 1 + |f_g| + |f_d|$.

D Tas binaires

a Définition

■ **Définition 138 — Tas max et tas min.** On appelle tas max (resp. tas min) un arbre binaire parfait étiqueté par un ensemble ordonné E tel que l'étiquette de chaque nœud soit inférieure (resp. supérieure) ou égale à l'étiquette de son père. La racine est ainsi la valeur maximale (resp. minimale) du tas.

b Implémentation

On peut naturellement implémenter un tas par un type `a' arbre` mais également par un tableau `Array` en numérotant les nœuds selon la numérotation Sosa-Stradonitz (cf. figure 9.7).

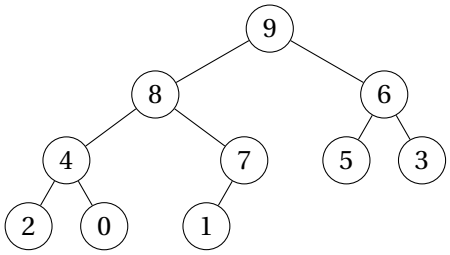


FIGURE 9.5 – Tas max

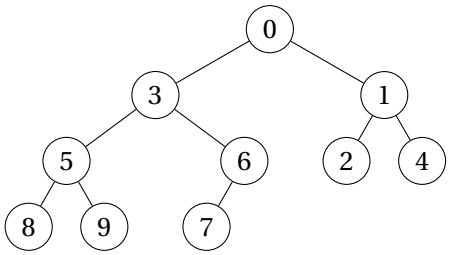


FIGURE 9.6 – Tas min

R Un tas implémenté par un tableau est une structure de taille donnée, fixée dès la construction du tas : on ne pourra donc pas représenter tous les tas, uniquement ceux qui pourront s'inscrire dans le tableau. Par ailleurs, pour construire cette structure et la préserver lors de l'exécution d'algorithmes, il est important que l'on puisse faire évoluer les éléments à l'intérieur du tas. C'est pourquoi cette structure de donnée doit être muable.

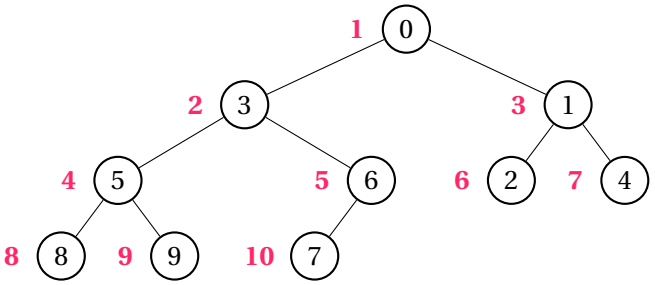
■ **Définition 139 — Numérotation Sosa-Stradonitz d'un arbre binaire.** Cette numérotation utilise les puissances de deux pour identifier les nœuds d'un arbre binaire. La racine se voit attribuer la puissance 0. Le premier élément de chaque niveau k de la hiérarchie possède l'indice 2^k. Ainsi, sur le troisième niveau d'un arbre binaire, on trouvera les numéros $8, 9, 10, 11, 12, 13, 14$ et 15. Cette numérotation est utilisée dans le domaine de la généalogie.

R Comme les langages de programmation comptent à partir de 0, on choisit souvent la convention de positionner la raciner dans la case d'indice 0 et décaler ensuite tous les indices de 1.

c Opérations

On s'intéresse à la construction et à l'évolution d'un tas au cours du temps : comment préserver la structure de tas lorsqu'on ajoute ou retire un élément ?

On définit des opérations descendre et faire monter un élément dans un tas qui préservent la structure du tas. Ce sont des opérations dans un tas sont des opérations dont la complexité



Indices	1	2	3	4	5	6	7	8	9	10
Valeurs	0	3	1	5	6	2	4	8	9	7

FIGURE 9.7 – Implémentation d'un tas min par un tableau selon les indices de la numérotation Sosa-Stradonitz. On vérifie que les fils du nœud à l'indice k se trouvent à l'indice $2k$ et $2k+1$

est $O(h(a)) = O(\log n)$.

Faire monter un élément dans le tas

Cette opération est expliquée sur la figure 9.8.

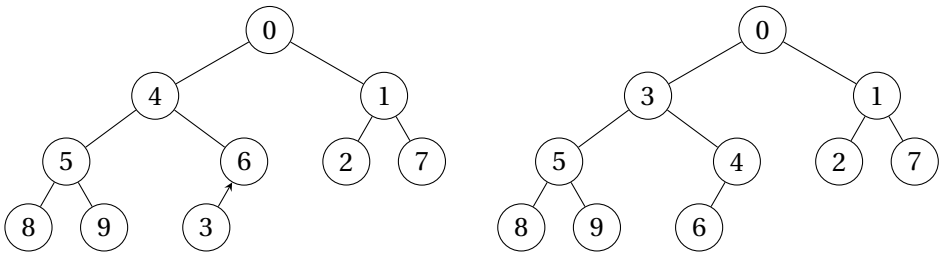


FIGURE 9.8 – Tas min : à gauche, l'élément 3 doit monter dans le tas min. À droite, on a échangé les places de 3 avec les pères jusqu'à ce que la structure soit conforme à un tas min

Faire descendre un élément dans le tas

Faire descendre dans le tas se dit aussi tamiser le tas et est expliqué sur la figure 9.9.

Construire un tas

On peut imaginer construire un tas en faisant monter les éléments ou en faisant descendre les éléments au fur et à mesure. Ces deux méthodes ne présentent pas la même complexité .

Si l'on procède en faisant monter les éléments, on considère que la racine est un tas à un élément et on intègre les éléments restant du tableau dans ce tas en les faisant monter. Dans

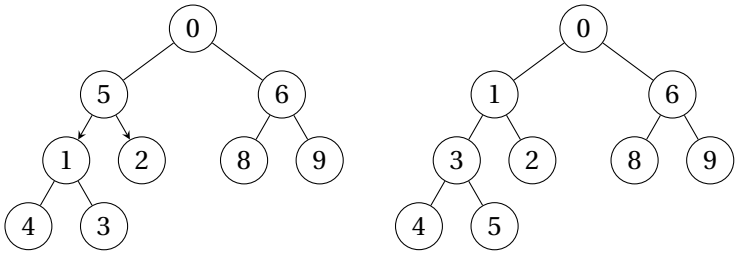


FIGURE 9.9 – Tas min : à gauche, l'élément 3 doit descendre dans le tas min. À droite, on a échangé la place de 3 avec le fils le plus petit pour que la structure soit conforme à un tas min

le pire des cas, on doit faire monter les $n-1$ nœuds depuis le niveau de profondeur maximale (hauteur de l'arbre) jusqu'à la racine et donc répéter l'opération monter (de complexité $\log n$). C'est pourquoi cette méthode présente une complexité en $O(n \log n)$.

La seconde méthode considère que chaque feuille est un tas à un élément. On fait descendre les $\lfloor n/2 \rfloor$ premiers éléments à partir du $\lfloor n/2 \rfloor$ [e] dans les tas. Au fur et à mesure, on réunit ces tas en un plus gros tas. Dans le pire des cas, on peut montrer que cette méthode est linéaire en $O(n)$. En effet, un élément i est à la hauteur $\lfloor \log_2 i \rfloor$ et il descend au maximum de $h - \lfloor \log_2 i \rfloor$ pour trouver sa place. Il est donc plus efficace de faire descendre les éléments pour créer un tas.

R Dans un tas de taille n, la première feuille se situe à l'indice $n/2$.

E Tri par tas binaire

■ **Définition 140 — Tri par tas.** Le tri par tas procède en formant d'un tas à partir du tableau à trier. Pour un tri ascendant, on utilise un tas-max et pour un tri descendant un tas-min.

On peut considérer que cette méthode est une amélioration du tri par sélection : le structure de tas permet d'éviter la recherche de l'élément à sélectionner.

Le tri par tas est un tri comparatif en place et non stable. Sa complexité dans le pire des cas est en $O(n \log n)$ car il nécessite au pire de descendre les n éléments au niveaux des feuilles dans le tas.

Vocabulary 11 — Heap sort ⇝ Tri par tas

L'algorithme 16 fait appel à un tas-max et son implémentation est radicalement simple, tout le cœur du mécanisme de tri reposant sur la structure de tas.

Algorithme 16 Tri par tas, ascendant

```
1 : Fonction TRI_PAR_TAS(t)
2 :     n ← nombre d'éléments de t
3 :     Faire un tas-max de t
4 :     pour k de n − 1 à 0 répéter
5 :         Échanger t[0] et t[k]                    ▷ Le plus grand va à la fin
6 :         Faire descendre t[0] dans le tas t[: k]   ▷ Le nouvel élément descend
7 :     renvoyer t
```

F File de priorités implémentée par un tas

Une autre application des tas binaires est l'implémentation d'une file à priorités.

■ **Définition 141 — TAD File de priorités.** Une file de priorités est une extension du TAD file dont les éléments sont à valeur dans un ensemble $E = (V, P)$ où P est un ensemble totalement ordonné. Les éléments de la file sont donc des couples valeur - priorité.

Les opérations sur une file de priorités sont :

1. créer une file vide,
2. insérer dans la file une valeur associée à une priorité (ENFILER),
3. sortir de la file la valeur associée la priorité maximale (ou minimale) (DÉFILER).

L'utilisation d'un tas permet d'obtenir une complexité en $O(\log n)$ pour les opérations DÉFILER et ENFILER d'une file de priorités. Pour des algorithmes comme celui du plus court chemin de Dijkstra, c'est une solution intéressante pour améliorer les performances de l'algorithme.

Sixième partie

Langages et automates

10

INTRODUCTION AUX LANGAGES

À la fin de ce chapitre, je sais :

- ☞ définir les concepts d'alphabet, de mot, de mot vide et de langage
- ☞ expliquer les concepts de suffixe, de préfixe, de facteur et de sous-mot
- ☞ expliquer le résultat du lemme de Levi

L'informatique est la construction de l'information par le calcul. Force est de constater que le seul outil conceptuel, universel et pratique pour construire et manipuler l'information est le langage : un langage est un moyen de communiquer, stocker et transformer de l'information. Le calcul de l'information par un ordinateur au moyen d'un ou plusieurs langages peut créer un sens ou pas, tout comme l'interprétation d'un texte par un humain.

C'est pourquoi la théorie des langages est un des principaux fondements de l'informatique. Qui dit langage dit alphabet, mots, préfixes, suffixes mais aussi ensembles de mots, agrégation de mots... Comment définir clairement ces concepts afin de pouvoir les calculer ? C'est la question qui guide ce chapitre.

A Alphabets

■ **Définition 142 — Ensemble.** Un ensemble est une collection de concepts qu'on appelle éléments. L'ensemble vide est noté $\emptyset$.

■ **Définition 143 — Cardinal d'un ensemble fini.** Le cardinal d'un ensemble fini E est son nombre d'éléments. On le note $|E|$.

■ **Définition 144 — Alphabet.** Un alphabet est un ensemble Σ de lettres (ou symboles) non vide.

■ **Définition 145 — Longueur d'un alphabet.** La longueur d'un alphabet est le nombre de lettres de celui-ci, c'est-à-dire $|\Sigma|$.

■ **Exemple 43 — Alphabet latin commun.** L'alphabet latin commun

$$\Sigma = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, X, Y, Z\}$$

a une longueur de 26.

■ **Exemple 44 — Héxadécimal.** L'alphabet héxadécimal

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$$

a une longueur de 16.

■ **Exemple 45 — ASCII.** L'alphabet ASCII est constitué des nombres entiers de 0 à 127 et représente les caractères nécessaire à l'écriture de l'américain, y compris les caractères de contrôle nécessaires à la pagination. Il possède une longueur de 128.

B Mots

Un mot d'un alphabet Σ est une séquence de lettres de Σ. Un mot peut être une séquence vide notée ϵ, car $\emptyset \subset \Sigma$ par définition d'un ensemble. On peut donner plusieurs définition d'un mot.

■ **Définition 146 — Mot (comme application).** Un mot de longueur $n \in \mathbb{N}^\star$ est une application de $[\![1, n]\!] \longrightarrow \Sigma$. À chaque position dans le mot correspond une lettre de l'alphabet.

■ **Définition 147 — Mot vide ϵ.** Le mot vide ϵ est l'application de l'ensemble vide dans Σ.

R Le mot vide ϵ est un mot ne comportant aucun symbole. Dans le contexte des mots, le mot vide est l'élément neutre de la concaténation de mots. On peut comparer son rôle au 1 pour la multiplication des entiers naturels $\mathbb{N}$.

■ **Définition 148 — Longueur d'un mot.** La longueur d'un mot w est le nombre de lettres qui composent sa séquence. On note souvent cette longueur $|\ w\ |$.

■ **Définition 149 — Ensemble de mots possibles.** On note Σ^* l'ensemble des mots possibles crées à partir d'un alphabet Σ.

■ **Définition 150 — Ensemble de mots de longueur n.** On note Σ^n l'ensemble de tous les mots de longueur n crées à partir d'un alphabet Σ.

■ **Définition 151 — Concaténation de mots.** Soit $v, w \in \Sigma^*$. On appelle concaténation de v et w l'opération $\circ$ notée $vw = v \circ w$ qui est obtenue par agrégation du mot w à la suite du mot v.

R Dans les notations suivantes, on omettra d'écrire le symbole $\circ$ entre les éléments, la concaténation étant juste une agrégation de symboles.

Théorème 21 — La concaténation est une loi de composition interne sur un ensemble Σ^* et ϵ en est l'élément neutre. .

Démonstration. Soit $v, w \in \Sigma^*$. On observe que $vw \in \Sigma^*$. Donc c'est une application de $\Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$. De plus, on peut observer que $v\epsilon = \epsilon v = v$. Donc ϵ est l'élément neutre de cette loi. ■

■ **Définition 152 — Monoïde.** Un ensemble E muni d'une loi de composition interne associative et d'un élément neutre e est nommée monoïde $(E, \star)$.

R On peut facilement montrer que la concaténation de mots est une loi de composition interne associative. C'est pourquoi, $(\Sigma^*, \circ)$ est un **monoïde**.

Il faut bien remarquer cependant qu'il n'existe pas a priori de concept d'inverse dans un monoïde, c'est-à-dire il **n'existe pas** de mots v et w tels que $vw = \epsilon$.

Toutefois, on peut simplifier par la gauche ou la droite :

$$vw = vx \Longrightarrow w = x \tag{10.1}$$
$$wv = xv \Longrightarrow w = x \tag{10.2}$$

R La longueur d'un mot est un morphisme de monoïde car $|vw| = |v| + |w|$.

■ **Définition 153 — Ensemble de mots non vides.** On note Σ^+ l'ensemble des mots non vides crées à partir d'un alphabet Σ. C'est le plus petit ensemble tel que :

$$\forall a \in \Sigma, a \in \Sigma^+ \tag{10.3}$$
$$\forall w \in \Sigma^+, \forall a \in \Sigma, wa \in \Sigma^+ \tag{10.4}$$

On a $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$.

C Mots définis inductivement

■ **Définition 154 — Mot (inductivement par la droite).** Soit Σ un alphabet. Alors on définit un mot de manière inductive par la droite ainsi :

Base ϵ est un mot sur Σ,

Règle de construction si w est un mot sur Σ et a une lettre de Σ, alors $w.a$ est un mot sur Σ.

où l'opération . est l'ajout d'une lettre à droite à un mot.

■ **Définition 155 — Mot (inductivement par la gauche)).** Soit Σ un alphabet. Alors on définit un mot de manière inductive par la droite ainsi :

Base ϵ est un mot sur Σ,

Règle de construction si w est un mot sur Σ et a une lettre de Σ, alors $a.w$ est un mot sur Σ.

où l'opération . est l'ajout d'une lettre à gauche à un mot.

■ **Définition 156 — Concaténation de mots (définie inductivement sur la première opérande).** On définit l'opérateur concaténation de mots $\circ$ par

$$\forall w \in \Sigma^*, \epsilon \circ w = w \text{ (Base)} \tag{10.5}$$

$$\forall v, w \in \Sigma^*, \forall a \in \Sigma, (a.v) \circ w = a.(v \circ w) \text{ (Règle de construction)} \tag{10.6}$$

où l'opération . est l'ajout d'une lettre à gauche à un mot.

■ **Définition 157 — Concaténation de mots (définie inductivement sur la seconde opérande).** On définit l'opérateur concaténation de mots $\circ$ par

$$\forall w \in \Sigma^*, w \circ \epsilon = w \tag{10.7}$$

$$\forall v, w \in \Sigma^*, \forall a \in \Sigma, v \circ (w.a) = (v \circ w).a \tag{10.8}$$

où l'opération . est l'ajout d'une lettre à droite à un mot.

Théorème 22 — ϵ est l'élément neutre de la concaténation.

Démonstration. on procède par induction sur la première opérande.

- Cas de base : pour $w = \epsilon$, on a $\epsilon \circ \epsilon = \epsilon$.
- Pas d'induction : soit $w \in \Sigma^*$. On suppose que ϵ est l'élément neutre pour ce mot : $w \circ \epsilon = \epsilon \circ w = w$. Considérons maintenant un élément a de l'alphabet Σ pour créer un mot plus long à partir de w. Par construction on a :

  $$(a.w) \circ \epsilon = a.(w \circ \epsilon)$$

  En utilisant l'hypothèse d'induction, on en déduit que $(a.w) \circ \epsilon = a.w$.
  ϵ est donc toujours l'élément neutre. On procède de même avec la définition sur la deuxième opérande.

■

R Une conséquence de ces définitions est qu'on peut confondre les opérateurs . et ∘ dans les notations. C'est ce qui est fait dans la suite de ce cours. On omettra également souvent l'opérateur lorsqu'il n'y a pas d'ambiguïtés.

■ **Définition 158 — Puissances d'un mot.** Les puissances d'un mot sont définies inductivement :

$$w^0 = \epsilon \tag{10.9}$$
$$w^n = w w^{n-1} \text{ pour } n \in \mathbb{N}^\star \tag{10.10}$$

D Langages

■ **Définition 159 — Langage.** Un langage sur un alphabet Σ est un ensemble de mots sur Σ.

R Un langage peut être vide, on le note alors $\mathcal{L} = \emptyset$, son cardinal est nul. C'est l'élément neutre de l'union des langages et l'élément absorbant de la concaténation de langages [a]. Il ne faut pas confondre ce langage vide avec le langage qui ne contient que le mot vide $\mathcal{L} = \{\epsilon\}$ dont le cardinal vaut un et qui est l'élément neutre de la concaténation des langages.

a. comme le zéro pour l'addition et la multiplication des entiers

R Σ^*, l'ensemble de tous les mots sur Σ, est également appelé langage universel.

■ **Exemple 46 — Langages courants et concrets.** Voici quelques exemples de langages concrets utilisés couramment :

- le langage des dates : une expression est-elle une date ? Par exemple, les dates 21/11/1943 et 11/21/43 sont-elles admissibles ?
- le langage des emails : utilisé pour détecter la conformité ou les erreurs dans les adresses emails,
- les protocoles réseaux : par exemple le protocole DHCP.

■ **Exemple 47 — Langage des mots de longueur paire.** Soit l'ensemble E de mots sur l'alphabet Σ de longueur paire. On peut définir ce langage en compréhension comme suit :

$$E = \{w \in \Sigma^*, |w| = 0 \bmod 2\}$$

■ **Exemple 48 — Langage des puissances n d'un alphabet.** Soit l'ensemble E de mots sur l'alphabet $\Sigma = \{a, b\}$ qui comportent autant de a que de b. On peut définir ce langage en

compréhension comme suit :

$$E = \{w \in \Sigma^*, \exists n \in \mathbb{N}, w \text{ est une permutation de } (a^n b^n)\}$$

Un langage est un ensemble. On peut donc définir les opérations ensemblistes sur les langages.

Soit deux langages $\mathcal{L}_1$ sur Σ_1 et $\mathcal{L}_2$ sur Σ_2.

■ **Définition 160 — Union de deux langages.** L'union de $\mathcal{L}_1$ et $\mathcal{L}_2$ est le langage défini sur $\Sigma_1 \cup \Sigma_2$ contenant tous les mots de $\mathcal{L}_1$ et de $\mathcal{L}_2$.

$$\mathcal{L}_1 \cup \mathcal{L}_2 = \{w, w \in \mathcal{L}_1 \text{ ou } w \in \mathcal{L}_2\} \tag{10.11}$$

■ **Définition 161 — Intersection de deux langages.** L'intersection de $\mathcal{L}_1$ et $\mathcal{L}_2$ est le langage défini sur $\Sigma_1 \cap \Sigma_2$ contenant tous les mots à la fois présents dans $\mathcal{L}_1$ et dans $\mathcal{L}_2$.

$$\mathcal{L}_1 \cap \mathcal{L}_2 = \{w, w \in \mathcal{L}_1 \text{ et } w \in \mathcal{L}_2\} \tag{10.12}$$

■ **Définition 162 — Complémentaire d'un langage.** Le complémentaire d'un langage $\mathcal{L}$ est le langage défini sur Σ qui contient tous les mots non qui ne sont pas dans $\mathcal{L}$.

$$C(\mathcal{L}) = \overline{\mathcal{L}} = \{w, w \in \Sigma^* \text{ et } w \notin \mathcal{L}\} \tag{10.13}$$

■ **Définition 163 — Différence de deux langages .** La différence de $\mathcal{L}_1$ et $\mathcal{L}_2$ est le langage défini sur Σ_1 contenant tous les mots présents dans $\mathcal{L}_1$ qui ne sont pas dans $\mathcal{L}_2$.

$$\mathcal{L}_1 \setminus \mathcal{L}_2 = \{w, w \in \mathcal{L}_1 \text{ et } w \notin \mathcal{L}_2\} \tag{10.14}$$

■ **Définition 164 — Produit de deux langages ou concaténation .** Le produit de $\mathcal{L}_1$ et $\mathcal{L}_2$ est le langage défini sur $\Sigma_1 \cup \Sigma_2$ contenant tous les mots formés par une mot de $\mathcal{L}_1$ suivi d'un mot de $\mathcal{L}_2$.

$$\mathcal{L}_1.\mathcal{L}_2 = \{vw, v \in \mathcal{L}_1 \text{ et } w \in \mathcal{L}_2\} \tag{10.15}$$

■ **Définition 165 — Puissances d'un langage.** Les puissances d'un langage $\mathcal{L}$ sont définies par induction :

$$\mathcal{L}^0 = \{\epsilon\} \tag{10.16}$$

$$\mathcal{L}^n = \mathcal{L}.\mathcal{L}^{n-1} \text{ pour } n \in \mathbb{N}^\star \tag{10.17}$$

■ **Définition 166 — Fermeture de Kleene d'un langage.** La fermeture de Kleene d'un langage $\mathcal{L}$ ou étoile de Kleene notée $\mathcal{L}^*$ est l'ensemble des mots formés par un nombre fini de

concaténation de mots de $\mathcal{L}$. Formellement :

$$\mathcal{L}^* = \bigcup_{n \geqslant 0} \mathcal{L}^n \tag{10.18}$$

La fermeture d'un langage peut également être définie inductivement par :

$$\epsilon \in \mathcal{L}^* \tag{10.19}$$
$$v \in \mathcal{L}, w \in \mathcal{L}^* \Longrightarrow vw \in \mathcal{L}^* \tag{10.20}$$
$$v \in \mathcal{L}^*, w \in \mathcal{L} \Longrightarrow vw \in \mathcal{L}^* \tag{10.21}$$

R Il existe un nombre dénombrable de mots sur un alphabet Σ, c'est à dire qu'on peut les mettre en bijection avec $\mathbb{N}$, il y en a une infinité mais on peut les compter. Néanmoins, le nombre de langages sur Σ n'est pas dénombrable puisqu'il s'agit des parties d'un ensemble dénombrable.

E Préfixes, suffixes, facteurs et sous-mots

■ **Définition 167 — Préfixe.** Soit v et w deux mots sur Σ. v est un préfixe de w et on le note $v \preceq w$ si et seulement s'il existe un mot u sur Σ tel que :

$$vu = w \tag{10.22}$$

■ **Définition 168 — Suffixe.** Soit v et w deux mots sur Σ. w est un suffixe de v si et seulement s'il existe un mot u sur Σ tel que :

$$uw = v \tag{10.23}$$

■ **Définition 169 — Facteur.** Soit v et w deux mots sur Σ. v est un facteur de w si et seulement s'il existe deux mots t et u sur Σ tel que :

$$tvu = w \tag{10.24}$$

Vocabulary 12 — Subword ⇜⇝ Facteur. Attention l'imbroglio n'est pas loin…

■ **Définition 170 — Sous-mot.** Soit $w = a_1 a_2 \dots a_n$ un mot sur $\Sigma = \{a_1, a_2, \dots, a_n\}$ de longueur n. Alors $v = a_{\psi(1)} a_{\psi(2)} \dots a_{\psi(p)}$ est un sous-mot de w de longueur p si et seulement s'il $\psi : [\![1, p]\!] \longrightarrow [\![1, n]\!]$ est une application strictement croissante.

R Cette définition implique que l'ordre d'apparition des lettres dans un sous-mot est préservé par rapport à l'ordre de lettres du mot.

Vocabulary 13 — Scattered Subword ⇜⇝ Sous-mot...

■ **Exemple 49 — Illustrations des concepts précédents.** Prenons par exemple le mot le plus long de la langue française, *anticonstitutionnellement*. Alors

- *anti* est un préfixe, tout comme antico mais uniquement pour les informaticiens, pas les linguistes...
- *ment* est un suffixe,
- *constitution* est un facteur,
- *colle* est un sous-mot.

Théorème 23 — Relations d'ordre partielles. Les relations «être préfixe de», «être suffixe de» et «être facteur de » sont des relations d'ordre partiel.

Démonstration. Il suffit de montrer que ces relations sont réflexives, transitives et antisymétriques. C'est un exercice à faire. ■

■ **Définition 171 — Ordre lexicographique.** Soit v et w deux mots sur un alphabet Σ sur lequel on dispose d'un ordre total $\leqslant_\Sigma$. Alors on peut définir[a] l'ordre lexicographique $\leqslant_l$ entre deux mots $v \leqslant_l w$ par :

- v est un préfixe de w
- ou bien $\exists t, v', w' \in \Sigma^*, v = tv', w = tw'$ et la première lettre de v' précède celle de w' dans l'alphabet, c'est-à-dire au sens de $\leqslant_\Sigma$.

a. Il était temps après 18 ans d'école!

■ **Définition 172 — Distance entre deux mots.** Supposons que l'on dispose d'une fonction λ capable de calculer le plus long préfixe, le plus long suffixe ou le plus long facteur commun entre deux mots. Alors on peut définir une distance entre deux mots v et w par :

$$d(v,w) = |vw| - 2\lambda(v,w) \tag{10.25}$$

■ **Définition 173 — Fermeture d'un langage par préfixe.** La fermeture par préfixe d'un langage $\mathcal{L}$ notée $\mathrm{Pref}(\mathcal{L})$ est le langage formé par l'ensemble des préfixes des mots de $\mathcal{L}$.

$$\mathrm{Pref}(\mathcal{L}) = \left\{w \in \Sigma^*, \exists v \in \Sigma^*, wv \in \mathcal{L}\right\} \tag{10.26}$$

■ **Définition 174 — Fermeture d'un langage par suffixe.** La fermeture par suffixe d'un langage $\mathcal{L}$ notée $\mathrm{Suff}(\mathcal{L})$ est le langage formé par l'ensemble des suffixes des mots de $\mathcal{L}$.

$$\mathrm{Suff}(\mathcal{L}) = \left\{w \in \Sigma^*, \exists v \in \Sigma^*, vw \in \mathcal{L}\right\} \tag{10.27}$$

■ **Définition 175 — Fermeture d'un langage par facteur.** La fermeture par facteur d'un langage $\mathcal{L}$ notée Fact($\mathcal{L}$) est le langage formé par l'ensemble des facteurs des mots de $\mathcal{L}$.

$$\text{Fact}(\mathcal{L}) = \left\{w \in \Sigma^*, \exists u, v \in \Sigma^*, uwv \in \mathcal{L}\right\} \tag{10.28}$$

F Propriétés fondamentales

On peut dors et déjà observer plusieurs faits :

- Soit v et w deux mots de Σ^*. A priori $vw \neq wv$, la concaténation n'est pas commutative.
- La décomposition d'un mot de Σ^* est unique en élément de Σ. Ceci est dû au fait qu'il n'y a pas d'inverse dans un monoïde. On peut le démontrer en raisonnant par l'absurde, en supposant qu'il existe deux décompositions différentes d'un même mot. Comme les lettres ne peuvent pas disparaître par inversion et qu'elles sont atomiques, c'est à dire non décomposables, on aboutit à une contradiction.

Ces deux observations engendrent de nombreux développements dans la théorie des langages.

Théorème 24 — Lemme de Levi. Soient t,u,v et w quatre mots de Σ^*. Si $tu = vw$ alors il existe un unique mot $z \in \Sigma^*$ tel que :

- soit $t = vz$ et $zu = w$,
- soit $v = tz$ et $zw = u$.

Démonstration. Supposons que $|t| \geqslant |v|$. Alors v est un préfixe de t et il existe un mot z tel que $t = vz$. Or, on a $tu = vw = vzu$. Par simplification à gauche, on obtient $w = zu$. On procède de même pour la seconde égalité. ■

Le lemme de Levi est illustré sur la figure 10.1.

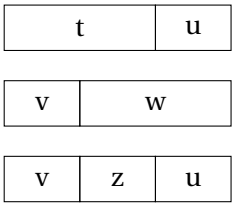


FIGURE 10.1 – Illustration du lemme de Levi

11

EXPRESSIONS RÉGULIÈRES

À la fin de ce chapitre, je sais :

- ☞ expliquer les définitions inductives des expressions régulières et des langages réguliers
- ☞ utiliser la sémantique des langages réguliers
- ☞ utiliser les identités remarquables sur les expressions régulières pour simplifier une expression
- ☞ construire un arbre représentant une expressions régulière

On peut décrire un langage de différentes manières :

- par compréhension : $\mathcal{L} = \{u \in \Sigma^*, |u| = 0 \bmod 2\}$, c'est-à-dire en utilisant une propriété spécifique au langage. Néanmoins, ceci n'est pas toujours évident et, de plus, cela n'implique pas de méthode concrète pour construire des mots de ce langage.
- pour les langages réguliers, par une expression régulière ou une grammaire régulière.

Les expressions régulières sont un moyen de caractériser de manière inductive certains langages et offre des **règles pour construire les mots** de ces langages. Tout comme en couture, on dit qu'elles constituent un **patron de conception**. À chaque expression régulière est associé un langage, c'est à dire l'ensemble des mots qu'elle permet d'élaborer. L'avantage principal des expressions régulières est qu'elles fournissent en plus une vision algébrique des langages réguliers et permettent donc le calcul.

R Régulier ou rationnel? Ces deux adjectifs sont employés de manière équivalente en français dans le cadre de la théorie des langages. Il existe des arguments en faveur de l'utilisation de chacun :

- rationnel : c'est le mot historique utilisé en France. Sa racine latine évoque le calcul et il s'agit donc des langages que l'on peut calculer.

- régulier : c'est un anglicisme mais dont la racine latine[a] évoque la conformation à une norme ou un règle. Il s'agit donc des langages que l'on peut décrire par une règle.

Ces deux adjectifs sont donc cohérents et utilisables en français pour décrire les langages et les expressions qui font l'objet de ce chapitre. L'un est plus pragmatique que l'autre !

a. via les anglo-normands et Guillaume le conquérant

A Définition des expressions régulières

■ **Définition 176 — Syntaxe des expressions régulières.** L'ensemble des expressions régulières $\mathcal{E}_R$ sur un alphabet Σ est défini inductivement par :

(Base) $\{\emptyset, \epsilon\} \cup \Sigma \in \mathcal{E}_R$,

(Règle de construction (union)) $\forall e_1, e_2 \in \mathcal{E}_R, e_1 \mid e_2 \in \mathcal{E}_R$

(Règle de construction (concaténation)) $\forall e_1, e_2 \in \mathcal{E}_R, e_1 e_2 \in \mathcal{E}_R$,

(Règle de construction (fermeture de Kleene)) $\forall e \in \mathcal{E}_R, e^* \in \mathcal{E}_R$.

R Cette définition peut s'exprimer ainsi : les expressions régulières sont constituées à la base de l'ensemble vide, du mot vide et des lettres de l'alphabet. On peut construire d'autres expressions régulières à partir de ces éléments de base en appliquant un nombre fini de fois les opérateurs d'union, de concaténation et de fermeture de Kleene.

R Le symbole de la concaténation est omis pour plus de lisibilité.

R L'utilisation des parenthèses est possible et souhaitable pour réduire les ambiguïtés. La priorité des opérateurs est l'étoile de Kleene, la concaténation puis l'union. Par défaut, l'associativité des opérateurs est choisie à gauche.

Par exemple, pour une alphabet $\{a, b, c\}$, on peut écrire :

- $a \mid b \mid c$ à la place de $a \mid (b \mid c)$
- $a \mid cb^*$ à la place de $a \mid (c(b^*))$

■ **Exemple 50 — Quelques expressions régulières.** Voici quelques expressions régulières pratiques dans la vie de tous jours :

$(P|MP|PC)SI$ sur l'alphabet latin désigne l'ensemble $\{PSI, MPSI, PCSI\}$

$(19|20)00$ sur l'alphabet $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$ désigne l'année 1900 ou l'année 2000.

1010101^* sur l'alphabet $\Sigma = \{0,1\}$ désigne l'ensemble des mots binaires préfixés par 101010 et se terminant éventuellement par des 1.

$\Sigma^* k \Sigma^+ q \Sigma^*$ il y a un mot de sept lettres engendré par cette expression régulière en français. Comme quoi, il y a toujours de l'espoir au Scrabble.

■ **Exemple 51 — D'autres expressions régulières.** Ces exemples sont typiques des expressions utilisées lors des épreuves de concours.

- Σ^* tous les mots
- $a\Sigma^*$ les mots commençant par a
- $\Sigma^* a$ les mots finissant par a
- $a^*|b^*$ les mots ne comportant que des a, que des b ou le mot vide
- $(ab^*a|b)^*$ les mots comportant un nombre pair de a
- $(aa|b)^*$ les mots comportant des blocs de a de longueur paire
- $(b|ab)^*(a|\epsilon)$ les mots ne comportant pas deux fois a consécutivement

R Les expressions régulières sont très puissantes et il est illusoire d'imaginer qu'on puisse les exprimer simplement avec des mots. C'est très rarement possible, uniquement sur des cas simples comme ci-dessus.

■ **Définition 177 — Langage dénoté par une expression régulière.** Pour toute expression régulière e, on note $\mathcal{L}_{ER}(e)$ le langage unique qui dénote cette expression. C'est en fait le résultat de l'application $\mathcal{L}_{ER} : \mathcal{E}_R(\Sigma^*) \longrightarrow \mathcal{P}(\Sigma^*)$ qui, à une expression régulière, fait correspondre l'ensemble des mots (le langage) engendré par cette expression régulière.

■ **Définition 178 — Sémantique des expressions régulières.** L'interprétation des expressions régulières en termes de langages, la sémantique d'une expression régulière, permet de définir inductivement l'ensemble des langages dénotés par une expression régulière ainsi :

(Base (i)) $\mathcal{L}_{ER}(\emptyset) = \{\} = \emptyset$,

(Base (ii)) $\mathcal{L}_{ER}(\epsilon) = \{\epsilon\}$,

(Base (iii)) $\forall a \in \Sigma, \mathcal{L}_{ER}(a) = \{a\}$,

(Règle de construction (i union)) $\forall e_1, e_2 \in \mathcal{E}_R, \mathcal{L}_{ER}(e_1 \mid e_2) = \mathcal{L}_{ER}(e_1) \cup \mathcal{L}_{ER}(e_2)$,

(Règle de construction (ii concaténation)) $\forall e_1, e_2 \in \mathcal{E}_R, \mathcal{L}_{ER}(e_1 e_2) = \mathcal{L}_{ER}(e_1).\mathcal{L}_{ER}(e_2)$,

(Règle de construction (iii fermeture de Kleene)) $\forall e \in \mathcal{E}_R, \mathcal{L}_{ER}(e^*) = \mathcal{L}_{ER}(e)^*$.

Théorème 25 — Un langage $\mathcal{L}$ est régulier si et seulement s'il existe une expression régulière e telle que $\mathcal{L}_{ER}(e) = \mathcal{L}$.

R Les apparences sont parfois trompeuses. Soit $\Sigma = \{a, b\}$ un alphabet et $\mathcal{L}$ le langage défini par $\mathcal{L} = \{a^n b^n, n \in [\![0, n]\!]\} = \{\epsilon, ab, aabb, aaabbb, \ldots\}$. $\mathcal{L}$ n'est pas un langage régulier. Si la formule entre crochets dénote bien un langage, un ensemble de mots, on ne peut cependant pas exprimer l'ensemble de ces mots par une expression régulière. Par exemple, $a^* b^*$ est

une expression régulière dont le langage associé dénote l'ensemble $\mathcal{L}$ ainsi que d'autres mots comme $\{a, b, aaa, bbb, aab, abb, abbb, \ldots\}$. Le lemme de l'étoile (cf. chapitre suivant) permet de démontrer que $\mathcal{L}$ n'est pas un langage régulier.

B Définition des langages réguliers

Il est également possible de donner une définition inductive directe des langages réguliers.

■ **Définition 179 — Ensemble des langages réguliers.** L'ensemble des langages réguliers $\mathcal{L}_{ER}$ sur un alphabet Σ est défini inductivement par :

(Base (i)) $\emptyset \in \mathcal{L}_{ER}$,

(Base (ii)) $\{\epsilon\} \in \mathcal{L}_{ER}$,

(Base (iii)) $\forall a \in \Sigma, \{a\} \in \mathcal{L}_{ER}$,

(Règle de construction (i)) $\forall \mathcal{L}_1, \mathcal{L}_2 \in \mathcal{L}_{ER}, \mathcal{L}_1 \cup \mathcal{L}_2 = \mathcal{L}_{ER}$

(Règle de construction (ii)) $\forall \mathcal{L}_1, \mathcal{L}_2 \in \mathcal{L}_{ER}, \mathcal{L}_1.\mathcal{L}_2 = \mathcal{L}_{ER}$,

(Règle de construction (iii)) $\forall \mathcal{L} \in \mathcal{L}_{ER}, \mathcal{L}^* = \mathcal{L}_{ER}$.

R Cette définition peut s'exprimer ainsi : les langages réguliers sont constitués à la base de l'ensemble vide, du langage ne contenant que le mot vide et des lettres de l'alphabet. On peut construire d'autres langages réguliers à partir de ces éléments de base en appliquant un nombre fini de fois les opérateurs d'union, de concaténation et de fermeture de Kleene.

■ **Définition 180 — Opérations régulières sur les langages.** L'union, la concaténation et la fermeture de Kleene sont les trois opérations régulières sur les langages.

R On note que l'intersection et la complémentation ne sont pas des opérations régulières.

Théorème 26 — Stabilité des langages réguliers. Les langages réguliers sont stables pour les opérations d'**union**, de **concaténation** et de **fermeture de Kleene**. Cela signifie que l'union, l'intersection ou la fermeture de Kleene de langages réguliers est un langage régulier.

Démonstration. Conséquence directe de la définition inductive. ■

C Identités remarquables sur les expressions régulières

Le tableau 11.1 recense quelques identités remarquables à connaître sur les expressions régulières. Leurs démonstrations s'appuient sur la sémantique des expressions régulières et les opérations sur les langages. Elles constituent un excellent exercice.

Démonstration. Par exemple, démontrons que $e|\emptyset = e$. D'après la sémantique des expressions régulières, on a :

$$\begin{aligned}\mathcal{L}_{ER}(e|\emptyset) =& \mathcal{L}_{ER}(e) \cup \mathcal{L}_{ER}(\emptyset) && (11.1)\\ =& \mathcal{L}_{ER}(e) \cup \emptyset && (11.2)\\ =& \mathcal{L}_{ER}(e) && (11.3)\end{aligned}$$

car l'ensemble vide est l'élément neutre de l'union ensembliste. ■

Expression régulière	Équivalent	Raison
$e\|\emptyset$	e	$\emptyset$ est l'élément neutre de l'union ensembliste
$e\emptyset$	$\emptyset$	Déf. de la concaténation, $\emptyset$ seul élément commun à $\mathcal{L}_{ER}(e)$ et $\{\emptyset\}$
$e\epsilon$	e	Déf. de la concaténation, ϵ élément neutre de la concaténation
$(e\|f)\|g$	$e\|f\|g$	Associativité de l'union ensembliste
$(e.f).g$	$e.f.g$	Associativité de la concaténation sur les langages
$e(f\|g)$	$ef\|eg$	Distributivité de la concaténation sur l'union
$(e\|f).g$	$eg\|fg$	Idem
$e\|f$	$f\|e$	Commutativité de l'union ensembliste
e^*	$\epsilon\|ee^*$	Définition de l'étoile de Kleene
e^*	$\epsilon\|e^*e$	Idem
$(\emptyset)^*$	ϵ	Définition de l'étoile de Kleene et des puissances d'un langage
$e\|e$	e	Idempotence
$(e^*)^*$	e^*	Idempotence

TABLE 11.1 – Identités remarquables des expressions régulières.

■ **Exemple 52 — Exemple de calcul sur les expressions régulières.** À l'aide des identités remarquables du tableau 11.1, il est possible de transformer des expressions régulières, de les simplifier par calcul. Voici un exemple sur $\Sigma = \{a, b\}$:

$$bb^*(a^*b^*|\epsilon)b = bb^*a^*b^*b \tag{11.4}$$

Cette expression est généralement exprimée ainsi $b^+a^*b^+$ dans les langages informatiques. Elle désigne l'ensemble des mots comportant au moins deux lettres qui commencent et terminent par un b. S'ils contiennent des a, ceux-ci sont consécutifs et encadrés par les b. On notera que le langage associé $\mathcal{L}_{ER}(b^+a^*b^+)$ ne contient pas le mot vide.

D Arbre associé à une expression régulière

De part sa nature inductive, une expression régulière peut naturellement être représentée sous la forme d'un arbre dont les feuilles sont les éléments de bases et les nœuds les opéra-

teurs réguliers. Ces arbres permettent de prouver par induction de nombreuses propriétés des expressions régulières.

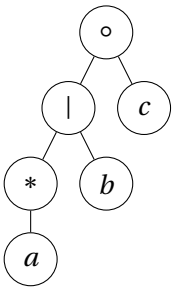


FIGURE 11.1 – Arbre associé à l'expression régulière $(a^*|b)c$

E Expressions régulières dans les langages --→ HORS PROGRAMME

Les expressions régulières sont intensivement utilisées par les informaticiens en pratique pour le test, le filtrage ou la transformation de l'information. Tous les langages évolués proposent une interface qui permet d'utiliser les expressions régulières.

Concrètement, les expressions rationnelles s'appuient sur un ensemble de caractères et de métacaractères qui permettent d'abstraire un motif de chaîne de caractères.

■ **Exemple 53 — Ensemble des lignes d'un fichier qui commencent au moins par un chiffre et qui se termine par Z.** On peut utiliser l'expression régulière suivante : `"^[0-9]+.*Z$"` où :

- `^` désigne le début d'une ligne
- `[0-9]` désigne l'ensemble des caractères 0,1,2,3,4,5,6,7,8,9
- `+` signifie au moins une occurrence du caractère précédent
- `.` symbolise n'importe quel caractère
- `*` signifie 0 ou un nombre quelconque d'occurrences du caractère précédent
- `Z` est le caractère Z
- `$` signifie la fin d'une ligne

■ **Définition 181 — Métacaractère.** Un métacaractère est un caractère qui a une signification abstraite, autre que son interprétation littérale.

Selon le langage utilisé, on pourra utiliser les classes ou les séquences spéciales (mode Perl).

■ **Exemple 54 — Les expressions célèbres.** Voici une liste non exhaustive d'expressions régulières couramment utilisées :

- `\d+` les entiers naturels

Métacaractère	Signification
`^`	début de la chaîne
`$`	fin de la chaîne
`.`	n'importe quel caractère sauf retour à la ligne
`*`	0 ou plusieurs fois le caractère précédent
`+`	au moins 1 fois le caractère précédent
`?`	0 ou 1 fois le caractère précédent
`|`	alternative (union)
`()`	groupement, motif
`[]`	ensemble de caractères
`{}`	nombre de répétition du motif

TABLE 11.2 – Liste des métacaractères

Expression	Signification
`e*`	zéro ou plusieurs e
`e+`	un ou plusieurs e
`e?`	0 ou 1 fois e
`e{m}`	exactement m fois e
`e{m,}`	au moins m fois e
`e{m,n}`	au minimum m fois a et au maximum n fois e

TABLE 11.3 – Répétition des motifs

Séquence	Signification
`\t`	tabulation
`\n`	nouvelle ligne
`\r`	retour chariot
`\b`	bordure de mot
`\B`	pas en bordure de mot
`\d`	correspond à n'importe quel chiffre [0-9]
`\D`	correpond à n'importe quel caractère sauf un chiffre
`\w`	correspond à n'importe quel mot [0-9a-zA-Z_]
`\W`	correspond à n'importe quelle séquence qui n'est pas un mot
`\s`	correspond à n'importe quel espace (espace, tabulation, nouvelle ligne)
`\S`	correspond à n'importe quel caractère qui n'est pas un espace

TABLE 11.4 – Séquences spéciales (mode Perl disponible en Python)

Classe	Signification
`[:alnum:]`	correspond aux caractères alphabétiques et numériques. [A-Za-z0-9]
`[:alpha:]`	correspond aux caractères alphabétiques. [A-Za-z]
`[:blank:]`	correspond à un espace ou à une tabulation
`[:cntrl:]`	correspond aux caractères de contrôle
`[:digit:]`	correspond aux chiffres [0-9]
`[:graph:]`	caractères graphiques affichables
`[:lower:]`	correspond aux caractères alphabétiques minuscules. [a-z]
`[:print:]`	caractères imprimables
`[:space:]`	correspond à tout espace blanc (espace, tabulation, nouvelle ligne)
`[:upper:]`	correspond à tout caractère alphabétique majuscule. [A-Z]
`[:xdigit:]`	correspond aux chiffres hexadécimaux. [0-9A-Fa-f]

TABLE 11.5 – Classes (mode expressions régulières étendues de grep)

- `-?\d+` les entiers
- `(\d{2}|\s)]{5}` un numéro de téléphone avec des espaces
- `[A-Za-z0-9_-]{3,16}` les noms des utilisateurs d'un système (login),
- `[[:alnum:]_-^@#!$% ]{8,42}` les mots de passe,
- `([a-z0-9_.-]+)@([0-9a-z.-]+).([a-z.]{2,24})` les adresses email

12

AUTOMATES FINIS DÉTERMINISTES

À la fin de ce chapitre, je sais :

- ☞ définir un automate fini déterministe
- ☞ représenter un automate fini déterministe
- ☞ qualifier les états d'un automates (accessibilité)
- ☞ compléter un AFD
- ☞ complémenter un AFD
- ☞ faire le produit de deux AFD

Les automates sont des machines simples dont les entrées sont des lettres et la sortie un résultat. Ces machines sont constituées par des états qui sont interconnectés par des liens permettant le passage d'un état à un autre selon un entrée et une direction. Selon les entrées reçues, l'automate réagit et se positionne donc dans un certain état. Les automates sont des éléments essentiels de l'informatique : ils établissent un lien fort entre la théorie des graphes et la théorie des langages.

Les chapitres de ce cours se focalisent sur les automates dont le résultat est l'acceptation ou non d'un mot d'un langage. Plus particulièrement, ce chapitre traite des automates finis déterministes (AFD), le déterminisme étant la capacité de l'automate à se positionner dans un seul état possible après la réception d'un lettre.

A Automate fini déterministe (AFD)

■ **Définition 182 — Automate fini déterministe (AFD).** Un automate fini déterministe est un quintuplet $(Q, \Sigma, q_i, \delta, F)$ tel que :

1. Q est un ensemble non vide et fini dont les éléments sont les états,

2. Σ est l'alphabet,
3. $q_i \in Q$ est l'état initial,
4. $\delta : Q \times \Sigma \longrightarrow Q$ est la **fonction** de transition de l'automate,
5. $F \subseteq Q$ est l'ensemble des états accepteurs ou terminaux.

R Le déterminisme d'un AFD est dû aux faits que :

- l'état initial est un singleton,
- δ est une fonction : à un couple (état,lettre) (q,a), δ associe au plus un état q'.

■ **Définition 183 — Fonction de transition partielle.** On dit que la fonction de transition δ est partielle s'il existe au moins un couple (état,lettre) pour lequel elle n'est pas définie.

■ **Définition 184 — Automate complet.** Un AFD $\mathcal{A} = (Q, \Sigma, q_i, \delta, F)$ est dit complet si δ est une application, c'est-à-dire δ n'est pas partielle, il existe une transition pour chaque lettre de Σ pour tous les états.

■ **Définition 185 — Automate normalisé.** Un automate est normalisé s'il ne possède pas de transition entrante sur son état initial et s'il possède un seul état final sans transition sortante.

B Représentation d'un automate

Les automates peuvent être représentés sous la forme de tableaux ou de graphes comme le montre les figures 12.1 et 12.1. Les algorithmes sur les graphes pourront être d'une aide précieuse pour l'étude des automates.

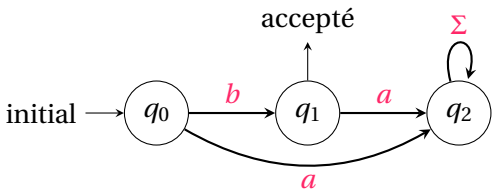


FIGURE 12.1 – Exemple d'automate fini déterministe représenté sous la forme d'un graphe

C Acceptation d'un mot

■ **Définition 186 — Fonction de transition étendue aux mots.** La fonction de transition peut être étendue aux mots par passages successifs d'un état à un autre en lisant les lettres

	$\downarrow q_0$	$\uparrow q_1$	q_2
a	q_2	q_2	q_2
b	q_1		q_2

	a	b
$\downarrow q_0$	q_2	q_1
$\uparrow q_1$	q_2	
q_2	q_2	q_2

TABLE 12.1 – Exemple d'automate fini déterministe représenté sous la forme de tableaux : on peut choisir de représenter les états en ligne ou en colonne.

d'un mot.

On définit inductivement cette fonction étendue noté δ^* :

$$\forall q \in Q, \delta^*(q, \epsilon) = q \tag{12.1}$$

$$\forall q \in Q, \forall w \in \Sigma^*, \forall a \in \Sigma, \delta^*(q, w.a) = \delta(\delta^*(q, w), a) \tag{12.2}$$

■ **Définition 187 — Acceptation d'un mot par un automate.** Un mot $w \in \Sigma^*$ est accepté par un automate $\mathcal{A}$ si et seulement si $\delta^*(q_i, w) \in F$, c'est-à-dire la lecture du mot w par l'automate conduit à un état accepteur.

■ **Définition 188 — Langage reconnu par un AFD.** Le langage $\mathcal{L}_{auto}(\mathcal{A})$ reconnu par un automate fini déterministe $\mathcal{A}$ est l'ensemble des mots reconnus par $\mathcal{A}$:

$$\mathcal{L}_{auto}(\mathcal{A}) = \{w \in \Sigma^*, w \text{ est accepté par } \mathcal{A}\} \tag{12.3}$$

■ **Définition 189 — Langage reconnaissable.** Un langage $\mathcal{L}$ sur un alphabet Σ est reconnaissable s'il existe un automate fini déterministe $\mathcal{A}$ d'alphabet Σ tel que $\mathcal{L} = \mathcal{L}_{auto}(\mathcal{A})$.

D Accessibilité et co-accessibilité

■ **Définition 190 — Accessibilité d'un état.** Un état q d'un automate est dit accessible s'il existe un mot $w \in \Sigma^*$ tel que $\delta^*(q_i, u) = q$, c'est-à-dire il est possible de l'atteindre depuis l'état initial.

■ **Définition 191 — Co-accessibilité d'un état.** Un état q d'un automate est dit co-accessible s'il existe un mot $w \in \Sigma^*$ tel que $\delta^*(q, u) \in F$, c'est-à-dire à partir de cet état, il est possible d'atteindre un état accepteur.

■ **Définition 192 — Automate émondé.** Un automate est dit émondé tous ses états sont à la fois accessibles et co-acessibles.

Théorème 27 — Automate d'un langage reconnaissable. Si un langage est reconnaissable alors il existe un automate fini déterministe :

- normalisé qui le reconnaît.
- émondé qui le reconnaît.
- complet qui le reconnaît.

E Complétion d'un AFD

M **Méthode 1 — Complété d'un AFD** Le complété d'un automate fini déterministe $\mathcal{A} = (Q, \Sigma, q_i, \delta, F)$ noté $C(\mathcal{A})$ est l'automate

$$C(\mathcal{A}) = (Q \cup \{q_p\}, \Sigma, q_i, C(\delta), F) \tag{12.4}$$

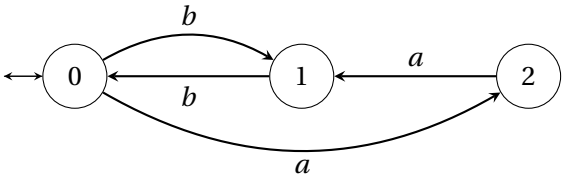
tel que :

- $q_p \notin Q$ est appelé l'état puits,
- $C(\delta)$ est l'application de $Q \cup \{q_p\} \times \Sigma$ dans $Q \cup \{q_p\}$ telle que :

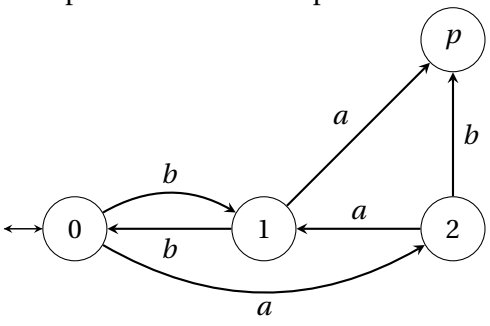
$$C(\delta)(q,a) = \begin{cases} \delta(q,a) \text{ si } \delta \text{ est définie pour } (q,a) \\ q_p \text{ sinon} \end{cases} \tag{12.5}$$

Cette méthode consiste donc à ajouter un état puits qui n'est pas co-accessible et à y faire converger toutes les transitions manquantes.

■ **Exemple 55 — Complétion d'un automate.** On considère l'automate fini déterministe suivant :



On observe que la fonction de transition n'est pas défini pour b en partant de l'état 2 ni pour a en partant de 1. La complétion de l'automate selon la méthode 1 donne :



Théorème 28 — Langage reconnus par un automate et son complété. Un automate $\mathcal{A}$ et son complété $C(\mathcal{A})$ reconnaissent le même langage :

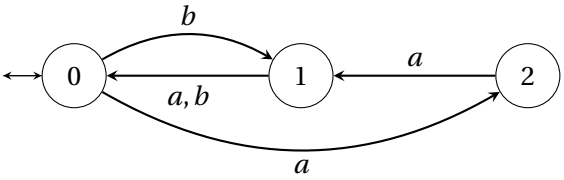
$$\mathcal{L}_{auto}(\mathcal{A}) = \mathcal{L}_{auto}(C(\mathcal{A})) \tag{12.6}$$

Démonstration. La fonction de transition pour un mot reconnu est la même sur les automates $\mathcal{A}$ et $C(\mathcal{A})$. Pour un mot non reconnu, elle diffère mais dans ce cas le mot n'appartient pas au langage. Donc les mots reconnus sont les mêmes. ■

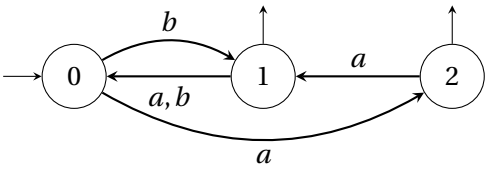
F Complémentaire d'un AFD

M **Méthode 2 — Complémentaire d'un AFD** Le complémentaire d'un automate fini déterministe $\mathcal{A} = (Q, \Sigma, q_i, \delta, F)$ noté $\mathcal{A}^c$ est l'automate complet $\mathcal{A}^c = (Q, \Sigma, q_i, \delta, Q \setminus F)$.

■ **Exemple 56 — Complémentaire d'AFD.** On considère l'automate fini déterministe suivant :



On observe que seul l'état 0 est un état accepteur. Le complémentaire de l'automate selon la méthode 2 donne :



Théorème 29 — Langage reconnus par un automate et son complémentaire. Le langage reconnu par l'automate complémentaire d'un automate est le complémentaire du langage reconnu par cet automate :

$$\mathcal{L}_{auto}(\mathcal{A}^c) = \Sigma^* \setminus \mathcal{L}_{auto}(\mathcal{A}) \tag{12.7}$$

Démonstration. On procède par double inclusion.

(⊂) Si w est un mot reconnu par l'automate complémentaire. Alors l'état accepteur de w n'appartient pas F, $w \notin Q \setminus F$. Donc w n'appartient pas à $\mathcal{L}_{auto}(\mathcal{A})$.

(⊃) Soit w un mot de l'ensemble $\Sigma^* \setminus \mathcal{L}_{auto}(\mathcal{A})$. En utilisant l'automate $\mathcal{A}$, le chemin emprunté en suivant les lettres de w ne mène donc pas à un état accepteur de $\mathcal{A}$. Comme tous les autres états de $\mathcal{A}$ sont des états accepteurs de $\mathcal{A}^c$, w est donc un mot de $\mathcal{L}_{auto}(\mathcal{A}^c)$.

■

Théorème 30 — Stabilité des langages reconnaissables par complémentation. Les langages reconnaissables sont stable par complémentation : s'il existe un langage reconnaissable sur Σ par un automate $\mathcal{A}$, alors le complémentaire de ce langage est reconnaissable par $\mathcal{A}^c$.

Démonstration. Ce théorème est une conséquence du théorème précédent et de la définition de l'automate complémentaire : si un langage est reconnaissable, alors son complémentaire l'est aussi puisqu'il existe un automate fini qui le reconnaît. ■

G Produit de deux AFD - Automate produit

M **Méthode 3 — Produit de deux automates** Le produit de deux automates sur Σ, $\mathcal{A}^a = (Q^a, \Sigma, q_i^a, \delta^a, F^a)$ et $\mathcal{A}^b = (Q^b, \Sigma, q_i^b, \delta^b, F^b)$, noté $\mathcal{A}^a \times \mathcal{A}^b$ est l'automate $\mathcal{A} = (Q, \Sigma, q_i, \delta, F)$ tel que :

- $Q = Q^a \times Q^b$
- $q_i = (q_i^a, q_i^b)$
- $\delta : (Q^a \times Q^b) \times \Sigma \longrightarrow (Q^a \times Q^b)$ est définie par : $\delta((q^a, q^b), s) = (\delta^a(q^a, s), \delta^b(q^a, s))$
- $F = F^a \times F^b$

Théorème 31 — Langage reconnus par un produit d'automates. Le langage reconnu par un produit d'automates est l'intersection des langages reconnus par ces automates :

$$\mathcal{L}_{auto}(\mathcal{A}^a \times \mathcal{A}^b) = \mathcal{L}_{auto}(\mathcal{A}^a) \cap \mathcal{L}_{auto}(\mathcal{A}^b) \tag{12.8}$$

Démonstration. On procède par double inclusion et on utilise la fonction de transition étendue aux mots.

($\subset$) Soit w un mot reconnu par l'automate produit. Alors, par définition de l'automate produit : $\delta^*((q^a, q^b), w) = (\delta^{a^*}(q^a, w), \delta^{b^*}(q^a, w))$ et $\delta^{a^*}(q^a, w) \in F^a$ et $\delta^{b^*}(q^a, w) \in F^b$. Donc $w \in \mathcal{L}_{auto}(\mathcal{A}^a) \cap \mathcal{L}_{auto}(\mathcal{A}^b)$.

($\supset$) soit w un mot reconnu par $\mathcal{A}^a$ et par $\mathcal{A}^b$. Alors un existe un chemin dans $\mathcal{A}^a$ qui, d'après le mot w, mène à un état accepteur de F^a. De même pour F^b. Donc $(\delta^a(q^a, w), \delta^b(q^a, w)) \in F^a \times F^b$. w appartient à $\mathcal{L}_{auto}(\mathcal{A}^a \times \mathcal{A}^b)$.

■

Théorème 32 — Stabilité des langages reconnaissables par l'intersection. Les langages reconnaissables sont stables par l'intersection : l'intersection de deux langages reconnaissables est un langage reconnaissable.

Démonstration. Ce théorème est un corolaire du théorème précédent. ■

R La stabilité des langages reconnaissables est importante car nous montrerons par la suite que les langages reconnaissables sont les langages réguliers. Or, les langages réguliers ne sont pas stables par définition pour les opérations non régulières, tout comme les langages reconnaissables ne sont pas stables par définition pour les opérations régulières (union, concaténation et fermeture de Kleene). Le théorème de Kleene va nous permettre d'étendre ces résultats d'une représentation à une autre.

13

AUTOMATES FINIS NON DÉTERMINISTES

À la fin de ce chapitre, je sais :

- ☞ reconnaître un automate fini non déterministe (AFND)
- ☞ déterminiser un AFND
- ☞ expliquer comment éliminer les transitions spontanées

A Automate fini non déterministe (AFND)

■ **Définition 193 — Automate fini non déterministe (AFND).** Un automate fini non déterministe est un quintuplet $(Q, \Sigma, Q_i, \Delta, F)$ tel que :

1. Q est un ensemble non vide et fini dont les éléments sont les états,
2. Σ est l'alphabet,
3. $Q_i \subseteq Q$ les états initiaux,
4. $\Delta \subseteq Q \times \Sigma \times Q$ est la **relation** de transition de l'automate,
5. $F \subseteq Q$ est l'ensemble des états accepteurs ou terminaux.

R Un AFND est par essence asynchrone. Son exécution nécessite l'exécution de tous les états possibles lors d'une transition.

R Le non déterminisme d'un AFND est dû au fait que Δ n'est pas une fonction mais une relation : une même lettre peut faire transiter l'AFND vers des états différents. Lequel choisir ? Là est le non déterminisme.

R Un AFND peut posséder plusieurs états initiaux, ce qui n'est pas le cas d'un AFD (q_i devient Q_i). On peut facilement se ramener à un seul état initial en utilisant des transitions spontanées. C'est pourquoi on considèrera souvent ce cas.

R Qui peut le plus peut le moins : un AFD est un cas particulier d'AFND pour lequel $\Delta = \{(q, a, q'), \delta(q, a) = q'\}$.

B Représentation d'un AFND

Les AFND peuvent être représentés de la même manière que les AFD sous la forme de tableaux ou de graphes comme le montre les figures 13.1 et 13.1.

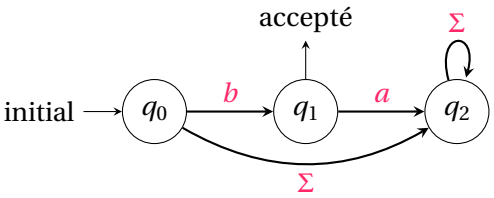


FIGURE 13.1 – Exemple d'automate fini non déterministe représenté sous la forme d'un graphe

	$\downarrow q_0$	$\uparrow q_1$	q_2
a	q_2	q_2	q_2
b	q_1, q_2		q_2

	a	b
$\downarrow q_0$	q_2	q_1, q_2
$\uparrow q_1$	q_2	
q_2	q_2	q_2

TABLE 13.1 – Exemple d'automate fini non déterministe représenté sous la forme de tableaux : on peut choisir de représenter les états en ligne ou en colonne.

R Pour un même mot, il peut donc exister plusieurs exécution possibles sur un AFND. La programmation des AFND n'est donc pas aussi simple que celles de AFD. Il faut être en mesure de tester tous les chemins possibles!

C Acceptation d'un mot

■ **Définition 194 — Relation de transition étendue aux mots.** La relation de transition peut être étendue aux mots par passages successifs d'un état à un autre en lisant les lettres d'un mot. On la note Δ^* .

■ **Définition 195 — Langage reconnu par un AFND.** Le langage $\mathcal{L}_{auto}(\mathcal{A})$ reconnu par un automate fini non déterministe $\mathcal{A}$ est l'ensemble des mots reconnus par $\mathcal{A}$:

$$\mathcal{L}_{auto}(\mathcal{A}) = \{w \in \Sigma^*, w \text{ est accepté par } \mathcal{A}\} \tag{13.1}$$

■ **Définition 196 — Langage reconnaissable par un AFND.** Un langage $\mathcal{L}$ sur un alphabet Σ est reconnaissable s'il existe un automate fini non déterministe $\mathcal{A}$ d'alphabet Σ tel que $\mathcal{L} = \mathcal{L}_{auto}(\mathcal{A})$.

D Déterminisé d'un AFND

M **Méthode 4 — Déterminisé d'un AFND** Le déterminisé d'un automate fini non déterministe $\mathcal{A} = (Q, \Sigma, Q_i, \Delta, F)$ est l'automate $\mathcal{A}_d = (\mathcal{P}(Q), \Sigma, q_i, \delta, \mathcal{F})$ défini par :

- $\mathcal{P}(Q)$ est l'ensemble des parties de Q,
- q_i est l'ensemble des états initiaux,
- $\forall \pi \in \mathcal{P}(Q), \forall a \in \Sigma, \delta(\pi, a) = \{q', (q, a, q') \in \Delta \text{ et } q \in \pi\}$,
- $\mathcal{F} = \{\pi \in \mathcal{P}(Q), \pi \cap F \neq \emptyset\}$.

Ce qui signifie que :

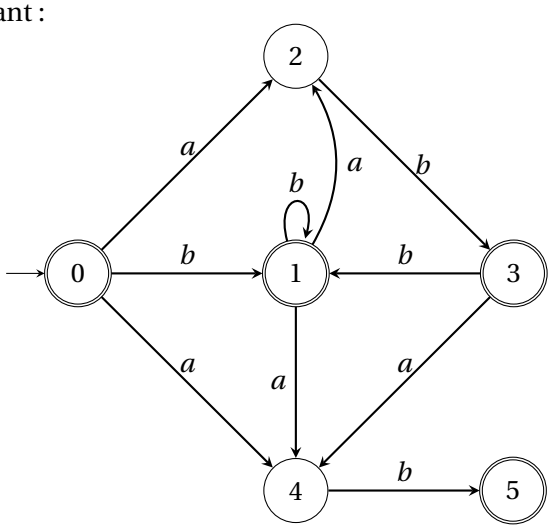
- l'état initial du déterminisé est l'état initial de l'AFND, ou bien, s'il possède plusieurs états initiaux, l'état initial constitué par la partition de tous les états initiaux de l'AFND.
- toute partie de Q est susceptible d'être un état. En pratique dans les exercices, vous construirez les états au fur et à mesure à partir de l'état initial comme dans l'exemple 57. L'ensemble des parties de Q n'a pas souvent besoin d'être explicité.
- les états accepteurs sont les parties qui contiennent un état accepteur de l'AFND.

L'algorithme 17 décrit cette méthode d'un point de vue opérationnel. Il s'agit de balayer un graphe en largeur et de mettre à jour les états et les transitions trouvées.

Algorithme 17 Algorithme de déterminisation d'un AFND

```
1 : Fonction DÉTERMINISER(A = (Q, Σ, Q_i, Δ, F))
2 :     transitions ← ∅
3 :     q_i ← la partition des états initiaux                    ▷ l'état initial
4 :     états ← q_i
5 :     file ← q_i
6 :     tant que file n'est pas vide répéter
7 :         q ← DÉFILER(file)
8 :         pour chaque lettre λ de l'alphabet Σ répéter
9 :             q' ← la partition des états possibles depuis (q, λ) d'après Δ
10 :            si q' n'est pas encore dans états alors
11 :                AJOUTER(états, q')
12 :                ENFILER(file,q')
13 :            AJOUTER(transitions, (q, λ, q'))
14 :    accepteurs ← ∅
15 :    pour chaque état e de états répéter
16 :        si e ∈F alors
17 :            AJOUTER(accepteurs, e)
18 :    renvoyer (états, Σ, q_i, transitions, accepteurs)
```

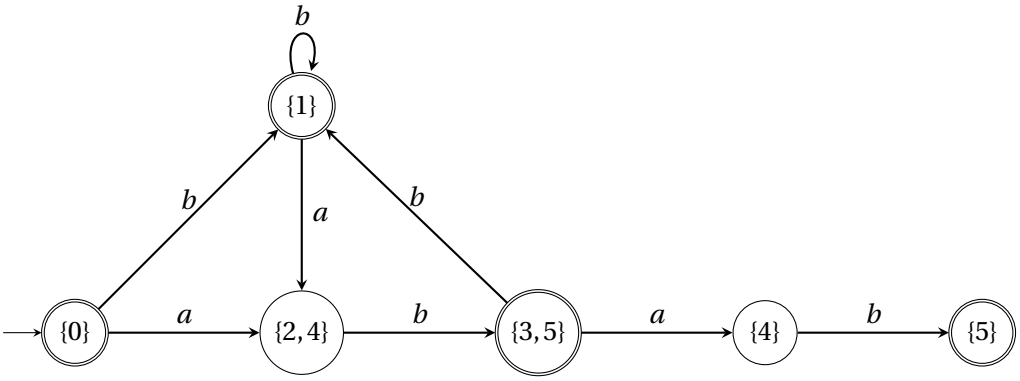
■ **Exemple 57 — Déterminer un AFND.** On considère l'automate fini non déterministe suivant :



En construisant les parties $\mathcal{P}(Q)$ au fur et mesure à partir de l'état initial, on obtient la fonction de transition :

	$\downarrow\{0\}$	$\uparrow\{1\}$	$\{2,4\}$	$\uparrow\{3,5\}$	$\{4\}$	$\uparrow\{5\}$
a	$\{2,4\}$	$\{2,4\}$	$\emptyset$	$\{4\}$	$\emptyset$	$\emptyset$
b	$\{1\}$	$\{1\}$	$\{3,5\}$	$\{1\}$	$\{5\}$	$\emptyset$

On en déduit l'AFD suivant :



Il est possible de renommer les états arbitrairement.

Théorème 33 Un AFND $\mathcal{A}$ et son déterminisé $\mathcal{A}_d$ reconnaissent le même langage.

$$\mathcal{L}_{auto}(\mathcal{A}) = \mathcal{L}_{auto}(\mathcal{A}_d) \tag{13.2}$$

Démonstration. $\mathcal{A}_d$ est bien un automate fini déterministe car :

- il possède au plus $2^{|Q|}$ états et $|Q|$ est fini puisque $\mathcal{A}$ est fini.
- son état de départ est unique,

- d'après la définition de δ, s'il existait deux états tels que $\delta(\pi, a) = \pi_1$ et $\delta(\pi, a) = \pi_2$, alors on aurait : $\pi_1 = \pi_2 = \{q', (q, a, q') \in \Delta \text{ et } q \in \pi\}$, c'est-à-dire que ces deux états seraient égaux. Donc δ est bien une fonction de transition et $\mathcal{A}_d$ est déterministe.
- comme $\mathcal{A}$ possède au moins un état final, $\mathcal{F}$ n'est pas vide.

Ensuite, il nous faut montrer que $\mathcal{L}_{auto}(\mathcal{A}_d) = \mathcal{L}_{auto}(\mathcal{A})$.

Tout d'abord, dire que le mot vide ϵ est dans le langage $\mathcal{L}_{auto}(\mathcal{A})$ est équivalent à dire que $q_i \cap F \neq \emptyset$. Sur le déterminé, cela se traduit par $q_i \in \mathcal{F}$. Donc, si le mot vide appartient à l'un, il appartient à l'autre.

($\Longrightarrow$) Soit w un mot non vide sur Σ. Si $w = a_1 \dots a_n$ est accepté par $\mathcal{A}$, alors cela signifie qu'il existe une succession d'états $(q_i, q_1 \dots, q_n)$ de $\mathcal{A}$ telle que $q_n \in F$. Mais alors, comme les transitions sont traduites dans l'automate déterminisé, pour cette succession d'état de $\mathcal{A}$, on peut trouver une succession d'états $(q_i, \pi_1, \dots, \pi_m)$ de $\mathcal{A}_d$ telle que $\pi_m \in \mathcal{F}$ et que chaque q_j de la succession d'états de $\mathcal{A}$ fasse partie d'une partition π_k de la succession d'état de $\mathcal{A}_d$. Donc w est accepté par l'automate déterminisé.

($\Longleftarrow$) On procède de même dans l'autre sens. ■

R Le déterminisé d'un AFND $\mathcal{A}$ comporte donc plus d'états que $\mathcal{A}$. Dans le pire des cas, l'algorithme de détermination a une complexité exponentielle.

E ϵ-transitions

■ **Définition 197 — ϵ-transition.** Une ϵ-transition est une transition dans un automate non déterministe dont l'étiquette est le mot vide ϵ. C'est une transition spontanée d'un état à un autre.

R Une transition spontanée fait qu'un automate peut être considéré dans deux états simultanément, celui qui précède la transition et le suivant. C'est pourquoi un automate déterministe ne comporte pas de transitions spontanées.

R Les automates à transition spontanée ou automates asynchrones sont utilisés notamment par l'algorithme de Thompson pour passer d'une expression régulière à un automate. Ces transitions peuvent aussi servir à normaliser un automate. La plupart du temps on les insère dans un automate pour les éliminer par la suite.

Un exemple d'un tel automate est donné sur la figure 13.2. Cet automate reconnaît les mots commençant par un nombre de b quelconque suivi d'un a ou bien les mots commençant par un nombre quelconque de a suivi par b. En fait, on va voir qu'on peut très bien exprimer un tel automate de manière non déterministe sans transitions spontanées et même de manière déterministe. Les $\epsilon-$transitions n'apportent donc pas d'expressivité en plus en terme de langage.

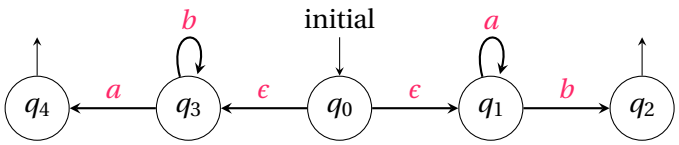


FIGURE 13.2 – Exemple d'automate fini non déterministe avec transition spontanée représenté sous la forme d'un graphe. On peut facilement les éliminer dans ce cas en créant plusieurs états initiaux.

14

DES EXPRESSIONS RÉGULIÈRES AUX AUTOMATES

À la fin de ce chapitre, je sais :

- ☞ expliquer le théorème de Kleene et ses conséquences
- ☞ transformer une expression régulière en automate
- ☞ montrer qu'un langage est local
- ☞ appliquer l'algorithme de Thompson et de Berry-Sethi
- ☞ décrire l'automate de Glushkov

A Théorème de Kleene

Les chapitres précédents ont permis de construire deux ensembles de langages :

1. l'ensemble des langages réguliers, c'est-à-dire dénotés par une expression régulière,
2. et l'ensemble des langages reconnaissables, c'est-à-dire reconnus par un automate fini.

Il s'agit maintenant d'établir une correspondance entre ces deux ensembles de langages.

Théorème 34 — Kleene. Un langage $\mathcal{L}$ sur un alphabet Σ est un langage régulier si et seulement s'il est reconnaissable.

Démonstration. ($\Longrightarrow$) Soit $\mathcal{L}$ un langage régulier. On utilise la définition inductive des langages réguliers pour montrer que ce langage est reconnaissable.

(Cas de base)
- l'ensemble vide est reconnu par un automate dont l'ensemble des états accepteurs F est vide. $\mathcal{L}_{ER}(\emptyset)$ est un langage reconnaissable.
- le mot vide est reconnu par un automate à un seul état dont l'état initial est accepteur. $\mathcal{L}_{ER}(\epsilon)$ est un langage reconnaissable.
- les lettres de l'alphabet sont des langages reconnaissables de la même manière.

(Pas d'induction) • (union) : soient $\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages réguliers reconnus par deux automates $\mathcal{A}_1$ et $\mathcal{A}_2$. Alors on a :

$$\begin{aligned}\mathcal{L}_1 \cup \mathcal{L}_2 =& \{w, w \in \mathcal{L}_1 \text{ ou } w \in \mathcal{L}_2\} \\ =& \{w, w \in \mathcal{L}_{auto}(\mathcal{A}_1) \text{ ou } w \in \mathcal{L}_{auto}(\mathcal{A}_2)\} \\ =& \mathcal{L}_{auto}(\mathcal{A}_1) \cup \mathcal{L}_{auto}(\mathcal{A}_2)\end{aligned}$$

D'après la loi de Morgan, on a $\mathcal{L}_{auto}(\mathcal{A}_1) \cup \mathcal{L}_{auto}(\mathcal{A}_2) = \overline{\overline{\mathcal{L}_{auto}(\mathcal{A}_1)} \cap \overline{\mathcal{L}_{auto}(\mathcal{A}_2)}}$. Or, les langages reconnaissables sont stables par intersection et passage au complémentaire. Ils sont donc stable pour l'union et $\mathcal{L}_{auto}(\mathcal{A}_1) \cup \mathcal{L}_{auto}(\mathcal{A}_2)$ est donc un langage reconnaissable.

- (concéténation) : soient $\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages réguliers reconnus par deux automates $\mathcal{A}_1$ et $\mathcal{A}_2$. Alors construisons l'automate $\mathcal{A}$ en reliant les états accepteurs de $\mathcal{A}_1$ à l'état initial de $\mathcal{A}_2$ par une transition spontanée (cf. figure 14.3). Cet automate $\mathcal{A}$ reconnaît alors le langage $\mathcal{L}_1.\mathcal{L}_2$.
- (fermeture de Kleene) : soit $\mathcal{L}$ un langages régulier reconnus par un automate $\mathcal{A}$. Construisons l'automate $\mathcal{A}$ associé comme sur la figure 14.4. Alors $\mathcal{A}$ reconnaît le langage $\mathcal{L}^*$.

(Conclusion) Un langage régulier $\mathcal{L}$ est un langage reconnaissable. Pour ce sens de la démonstration, on s'est appuyé sur l'algorithme de Thompson. Mais on peut aussi utiliser l'automate de Glushkov et l'algorithme de Berry-Sethi.

($\Longleftarrow$) L'algorithme de Mac Naughton-Yamada permet de calculer l'expression régulière associée à un automate fini (⇢ HORS PROGRAMME). On le montrera via l'élimination des états. ■

R Le théorème de Kleene permet d'affirmer que les langages réguliers sont stables par intersection, complémentation ce qui n'était pas évident d'après la définition des expressions régulières. Inversement, les langages reconnaissables sont stables par union, concaténation et passage à l'étoile de Kleene. Tout résultat sur un type de langage (reconnaissable ou régulier) peut se transposer à l'autre type grâce au théorème de Kleene.

Les sections qui suivent présentent les algorithmes au programme qui permettent de passer du formalisme d'une expression régulière à celui d'un automate et inversement.

B Algorithme de Thompson

L'algorithme de Thompson permet de construire un automate reconnaissant le langage dénoté par une expression régulière en utilisant des patrons de conception d'automate normalisé pour chaque cas de base et chaque opération (union, concaténation, étoile de Kleene). Cet algorithme porte également le nom de méthode compositionnelle car on compose des automates correspondants à des expressions simples.

a Patron de conception d'un cas de base

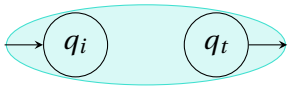
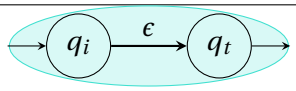
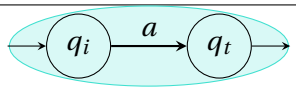
Expression régulière	Automate associé
$\emptyset$	
ϵ	
$a \in \Sigma$	

TABLE 14.1 – Automates associés aux cas de bases des expressions régulières.

b Patron de conception de l'union

On associe à l'union de deux expressions régulières $e_1|e_2$ l'automate décrit sur la figure 14.2. Au démarrage de la procédure, les deux expressions possèdent un automate équivalent comme le montre la figure 14.1.

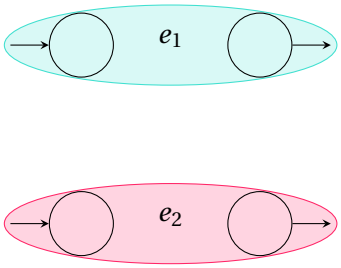


FIGURE 14.1 – Automates équivalents à e_1 et e_2 avant l'opération

c Patron de conception de la concaténation

On associe à la concaténation de deux expressions régulières l'automate décrit sur la figure 14.3.

d Patron de conception de l'étoile de Kleene

On associe à la fermeture de Kleene d'une expression régulière l'automate décrit sur la figure 14.4.

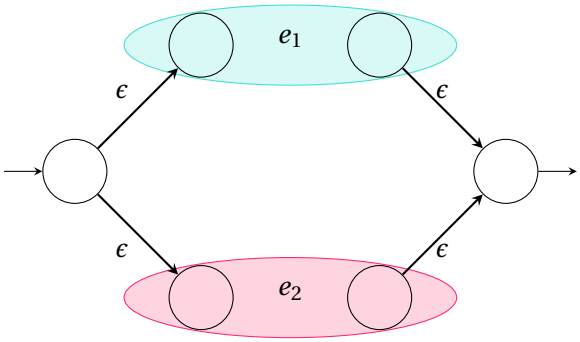


FIGURE 14.2 – Automate associé à l'union de deux expressions régulières $e_1|e_2$

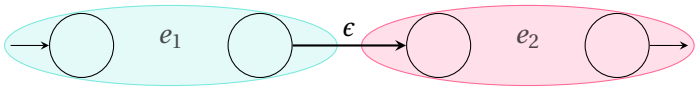


FIGURE 14.3 – Automate associé à la concaténation de deux expressions régulières $e_1 e_2$

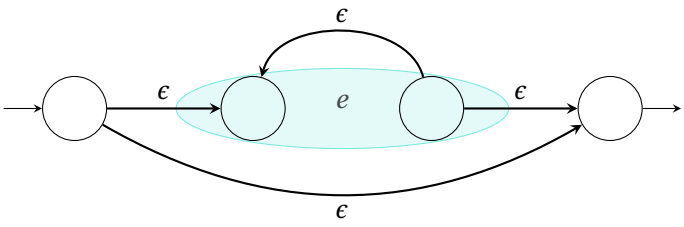
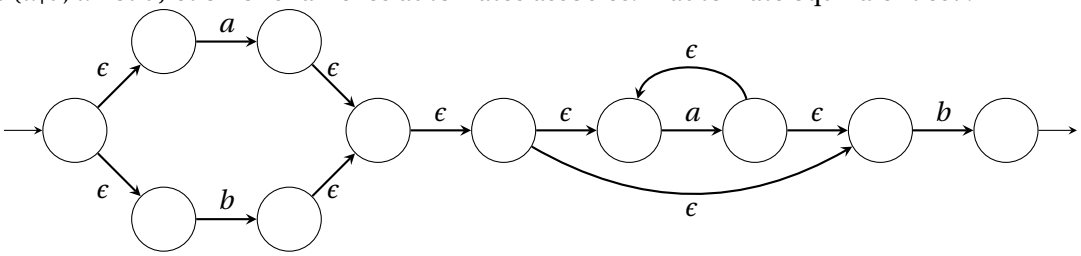


FIGURE 14.4 – Automate associé à la fermeture de Kleene de e

e Application

■ **Exemple 58** — $(a|b)a^*b$. On décompose l'expression régulière en éléments simples concaténés ($a|b$, a^* et b) et on enchaîne les automates associés. L' automate équivalent est :



f Élimination des transitions spontanées

L'automate de la figure 58 comporte un certain nombre de transitions spontanées. Si, parfois, ces transitions permettent de rendre plus lisible l'automate, elles multiplie cependant les états ce qui n'est pas souhaitable, surtout dans l'optique de programmer cet automate en le déterminisant... Il faut donc trouver une méthode pour éliminer ces transitions spontanées.

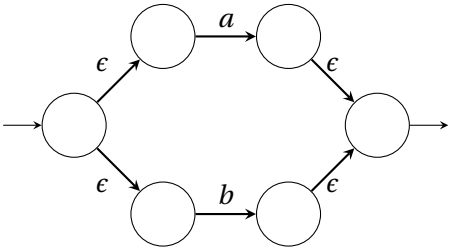
R Même s'il est possible d'éliminer les transitions spontanées une fois qu'on a construit tout l'automate associé à une expression régulière, il est souvent souhaitable de le faire à la volée afin de ne pas aboutir à un automate illisible.

M **Méthode 5 — Élimination des transitions spontanées** Il existe deux procédures similaires, une par l'avant, une par l'arrière.

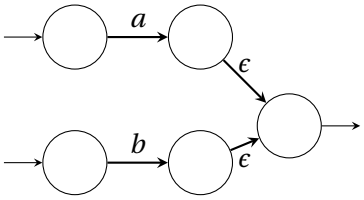
1. Fermeture par l'avant :
   - $\left(p \xrightarrow{a} q \xrightarrow{\epsilon} r\right) \rightsquigarrow \left(p \xrightarrow{a} r \text{ et } p \xrightarrow{a} q\right)$ Pour chaque transition d'un état p à un état q portant une lettre a et pour chaque transition spontanée de q à un état r, ajouter une transition de q à r portant la lettre a. Éliminer la transition spontanée.
   - $\left(\longrightarrow p \xrightarrow{\epsilon} q\right) \rightsquigarrow q \in Q_{init}$ Pour chaque transition spontanée d'un état p initial à un état q, ajouter q à l'ensemble des états initiaux. Éliminer la transition spontanée.
2. Fermeture par l'arrière :
   - $\left(p \xrightarrow{\epsilon} q \xrightarrow{a} r\right) \rightsquigarrow \left(p \xrightarrow{a} r \text{ et } q \xrightarrow{a} r\right)$ Pour chaque transition spontanée d'un état p à un état q et pour chaque transition de q à un état r portant la lettre a, ajouter une transition de p à r portant la lettre a. Éliminer la transition spontanée.
   - $\left(p \xrightarrow{\epsilon} q \longrightarrow\right) \rightsquigarrow p \in F$ Pour chaque transition spontanée d'un état p initial à un état q, ajouter q à l'ensemble des états initiaux. Éliminer la transition spontanée.

■ **Exemple 59 — Élimination des transitions spontanées de l'automate de l'exemple 58.** Pour chaque automate associé, on peut déjà appliquer la fermeture avant ou arrière.

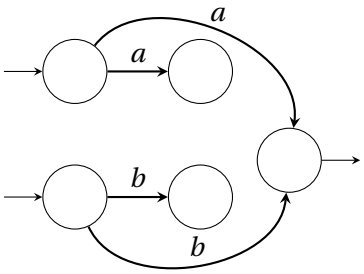
Pour l'union $a|b$, on trouve l'automate sans transitions spontanée en plusieurs étapes que voici :



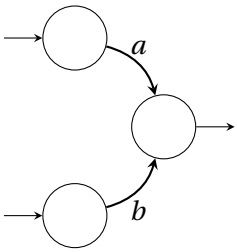
(i) état de départ



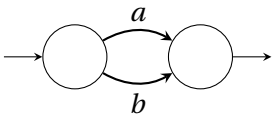
(ii) élmination par l'avant et l'état initial



(iii) élimination avant

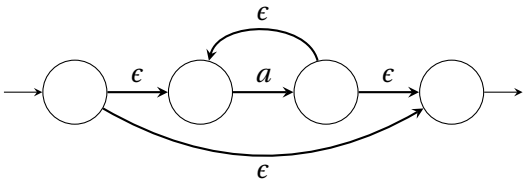


(iv) émondage

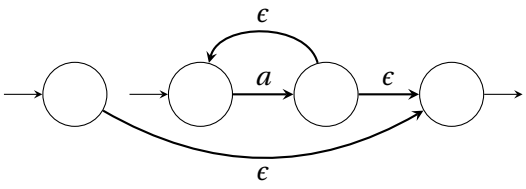


(iv) un seul état initial

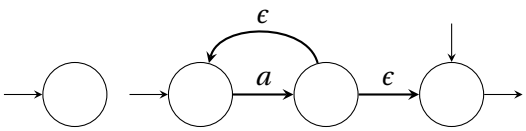
Pour la fermeture de Kleene, on procède de la même manière :



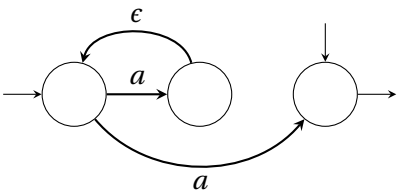
(i) état de départ



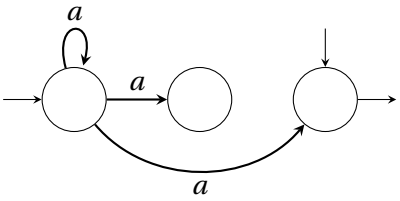
(ii) élimination par l'avant et l'état initial



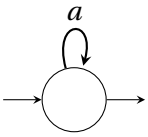
(iii) élimination par l'avant et l'état initial



(iv) émondage et élimination par l'avant

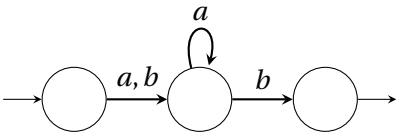


(v) élimination par l'avant



(vi) émondage et simplification

Finalement, on peut maintenant représenter l'automate normalisé (et déterministe) après élimination des transitions spontanées :



C Algorithme de Berry-Sethi et automate de Glushkov

Les notions de langage local et d'expression régulière linéaire sont introduites dans la seule perspective de construire l'automate de Glushkov[7] associé à une expression régulière linéaire par l'algorithme de Berry-Sethi[3], conformément au programme. C'est un long développement pour une procédure finale relativement simple.

a Langages locaux

■ **Définition 198 — Ensembles .** Soit $\mathcal{L}$ un langage sur Σ. On définit quatre ensembles de la manière suivante :

- les premières lettres des mots de $\mathcal{L}$

$$P(\mathcal{L}) = \{a \in \Sigma, \exists w \in \Sigma^*, aw \in \mathcal{L}\} \tag{14.1}$$

- les dernières lettres des mots de $\mathcal{L}$:

$$S(\mathcal{L}) = \{a \in \Sigma, \exists w \in \Sigma^*, wa \in \mathcal{L}\} \tag{14.2}$$

- les facteurs de longueur 2 des mots de $\mathcal{L}$:

$$F(\mathcal{L}) = \left\{v \in \Sigma^*, |v| = 2, \exists u, w \in \Sigma^*, uvw \in \mathcal{L}\right\} \tag{14.3}$$

- les facteurs de longueur 2 impossibles :

$$N(L) = \Sigma^2 \setminus F(L) \tag{14.4}$$

■ **Définition 199 — Langage local.** Un langage $\mathcal{L}$ sur Σ est local s'il existe deux parties P et S de Σ et une partie N de Σ^2 tels que :

$$\mathcal{L} \setminus \{\epsilon\} = (P\Sigma^* \cap \Sigma^* S) \setminus (\Sigma^* N \Sigma^*) \tag{14.5}$$

Dans ce cas, on a nécessairement $P = P(\mathcal{L})$, $S = S(\mathcal{L})$, $N = N(\mathcal{L})$.

R Cette définition signifie que l'appartenance d'un mot à un langage local peut être établie uniquement en regardant la première lettre, la dernière lettre et tous les blocs de deux lettres de ce mot. On peut imaginer que, pour vérifier, on fait glisser pour comparer tous les blocs de deux lettres non autorisés (N) sur le mot. D'où le nom local : on n'a pas besoin d'examiner dans sa globalité le mot, mais uniquement chaque lettre et sa voisine. P, S et N suffisent donc pour définir un langage local.

■ **Exemple 60 — Langages locaux .** Sur l'alphabet $\Sigma = \{a, b\}$, on peut déterminer pour chacun des langages suivants les langages P, S et N et dire s'ils sont locaux :

1. $\mathcal{L}_{ER}(a^*)$: $P = \{a\}$, $S = \{a\}$ et $N = \{ab, ba, bb\}$.
2. $\mathcal{L}_{ER}((ab)^*)$: $P = \{a\}$, $S = \{b\}$ et $N = \{aa, bb\}$.

■ **Exemple 61 — Langages non locaux.** Sur l'alphabet $\Sigma = \{a, b\}$, on peut déterminer pour chacun des langages suivants les langages P, S et N et trouver un contre-exemple pour montrer qu'ils ne sont pas locaux :

1. $\mathcal{L}_{ER}(a^*(ab)^*)$: $P = \{a\}$, $S = \{a, b\}$ et $N = \{bb\}$. Mais on observe que le mot aba est dans $(P\Sigma^* \cap \Sigma^* S) \setminus (\Sigma^* N \Sigma^*)$ mais pas dans $\mathcal{L}$.
2. $\mathcal{L}_{ER}(a^*|(ab)^*)$: idem

R Le dernier exemple montre que les langages locaux ne sont pas stables par union et concaténation.

Théorème 35 — L'ensemble des langages locaux est stable par intersection.

Démonstration. Soient $\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages locaux, et $\mathcal{L} = \mathcal{L}_1 \cap \mathcal{L}_2$. On pose $P = P(\mathcal{L})$, $S =$

$S(\mathcal{L})$ et $N = N(\mathcal{L})$, les ensembles définis comme en 198. Alors on a :

$$\begin{aligned}\mathcal{L} \setminus \{\epsilon\} &= (\mathcal{L}_1 \setminus \{\epsilon\}) \cap (\mathcal{L}_2 \setminus \{\epsilon\}) \\ &= ((P(\mathcal{L}_1)\Sigma^* \cap \Sigma^* S(\mathcal{L}_1)) \setminus (\Sigma^* N(\mathcal{L}_1)\Sigma^*)) \cap ((P(\mathcal{L}_2)\Sigma^* \cap \Sigma^* S(\mathcal{L}_2)) \setminus (\Sigma^* N(\mathcal{L}_2)\Sigma^*)) \\ &= (P(\mathcal{L}_1)\Sigma^* \cap \Sigma^* S(\mathcal{L}_1) \cap P(\mathcal{L}_2)\Sigma^* \cap \Sigma^* S(\mathcal{L}_2)) \setminus (\Sigma^* N(\mathcal{L}_1)\Sigma^* \cup \Sigma^* N(\mathcal{L}_2)\Sigma^*) \\ &= (P\Sigma^* \cap \Sigma^* S) \setminus (\Sigma^* N\Sigma^*).\end{aligned}$$

Donc $\mathcal{L}$ est bien local. ■

Théorème 36 — L'union de deux langages locaux définis sur deux alphabets disjoints est un langage local.

Démonstration. Soit Σ_1 et Σ_2 les alphabets **disjoints** sur lesquels sont définis $\mathcal{L}_1$ et $\mathcal{L}_2$, deux langages locaux et $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$ défini sur $\Sigma = \Sigma_1 \cup \Sigma_2$. On pose $P = P(\mathcal{L})$, $S = S(\mathcal{L})$ et $N = N(\mathcal{L})$, les ensembles définis comme en 198.

On doit montrer l'inclusion $(P(\mathcal{L})\Sigma^* \cap \Sigma^* S(\mathcal{L})) \setminus (\Sigma^* N(\mathcal{L})\Sigma^*) \subset \mathcal{L}$ puisque l'inclusion réciproque est toujours vraie.

Considérons donc un mot $w \in (P(\mathcal{L})\Sigma^* \cap \Sigma^* S(\mathcal{L})) \setminus (\Sigma^* N(\mathcal{L})\Sigma^*)$ que l'on décompose en lettres $w = a_1 \dots a_n$. Montrons que $w \in \mathcal{L}$

- Soit $a_1 \in P(\mathcal{L}) = P(\mathcal{L}_1) \cup P(\mathcal{L}_2)$, on peut supposer sans perte de généralité que $a_1 \in P(\mathcal{L}_1)$, alors $a_1 \in \Sigma_1$.
- $a_1.a_2 \in \Sigma^2 \setminus N(\mathcal{L}) = F(\mathcal{L}_1) \cup F(\mathcal{L}_2)$, or $a_1 \in \Sigma_1$ et les alphabets Σ_1 et Σ_2 sont disjoints, donc nécessairement $a_1 a_2 \in F(\mathcal{L}_1)$ et $a_2 \in \Sigma_1$.
- De proche en proche, on montre que $a_i a_{i+1} \in F(\mathcal{L}_1)$ et $a_i \in \Sigma_1$ pour tout i.
- Enfin, $a_n \in S(\mathcal{L}) = S(\mathcal{L}_1) \cup S(\mathcal{L}_2)$ et $a_n \in \Sigma_1$ donc $a_n \in S(\mathcal{L}_1)$.
- Finalement $w \in (P(\mathcal{L}_1)\Sigma^* \cap \Sigma^* S(\mathcal{L}_1)) \setminus (\Sigma^* N(\mathcal{L}_1)\Sigma^*) = \mathcal{L}_1$ car $\mathcal{L}_1$ est local.

En partant de l'hypothèse que $a_1 \in \Sigma_2$, on en aurait conclu que $w \in \mathcal{L}_2$. On en déduit que $w \in \mathcal{L}_1 \cup \mathcal{L}_2$ et, donc, $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$ est un langage local. ■

Théorème 37 — La concaténation de deux langages locaux définis sur deux alphabets disjoints est un langage local.

Démonstration. Soit Σ_1 et Σ_2 les alphabets **disjoints** sur lesquels sont définis $\mathcal{L}_1$ et $\mathcal{L}_2$, deux langages locaux et $\mathcal{L} = \mathcal{L}_1\mathcal{L}_2$ défini sur $\Sigma = \Sigma_1\Sigma_2$. On pose $P = P(\mathcal{L})$, $S = S(\mathcal{L})$ et $N = N(\mathcal{L})$, les ensembles définis comme en 198.

On doit montrer l'inclusion $(P(\mathcal{L})\Sigma^* \cap \Sigma^* S(\mathcal{L})) \setminus (\Sigma^* N(\mathcal{L})\Sigma^*) \subset \mathcal{L}$ puisque l'inclusion réciproque est toujours vraie.

$$P(\mathcal{L}_1\mathcal{L}_2) = \begin{cases} \{a \in \Sigma_2, \exists w \in \Sigma_2^*, aw \in \mathcal{L}_2\} & \text{si } \epsilon \in \mathcal{L}_1 \\ \{a \in \Sigma_1, \exists w \in \Sigma_1^*\Sigma_2^*, aw \in \mathcal{L}_1\mathcal{L}_2\} & \text{sinon} \end{cases}$$

$$S(\mathcal{L}_1\mathcal{L}_2) = \begin{cases} \{a \in \Sigma_1, \exists w \in \Sigma_1^*, wa \in \mathcal{L}_1\} & \text{si } \epsilon \in \mathcal{L}_2^* \\ \{a \in \Sigma_2, \exists w \in \Sigma_1^*\Sigma_2^*, wa \in \mathcal{L}_1\mathcal{L}_2\} & \text{sinon} \end{cases}$$

$$F(\mathcal{L}_1\mathcal{L}_2) = \{v \in \Sigma_1^*\Sigma_2^*, |v| = 2, \exists u, w \in \Sigma_1^*\Sigma_2^*, uvw \in \mathcal{L}_1\mathcal{L}_2\}$$

On considère un mot $w \in (P(\mathcal{L})\Sigma^* \cap \Sigma^* S(\mathcal{L})) \setminus (\Sigma^* N(\mathcal{L})\Sigma^*)$ et on le décompose en lettres $w = a_0 \dots a_n$. On traite différents cas :

- Si $a_0 \in \Sigma_2$, alors $\epsilon \in \mathcal{L}_1$ et $a_2 \in P(\mathcal{L}_2)$. De proche en proche on montre que $a_i a_{i+1} \in F(\mathcal{L}_2)$, $a_i \in \Sigma_2$ pour tout i et $a_n \in S(\mathcal{L}_2)$, donc $w \in \mathcal{L}_2$ car $\mathcal{L}_2$ est local.
- Si $a_0 \in \Sigma_1$, alors $a_0 \in P(\mathcal{L}_1)$. Notons $a_0 \dots a_k$ le plus long préfixe de w qui soit dans Σ_1^*. On montre de proche en proche que $a_i a_{i+1} \in F(\mathcal{L}_1)$ pour $i < k$. Puis deux cas se présentent :
  - Si $k = n$ alors $a_n \in S(\mathcal{L}_1)$, donc $w \in \mathcal{L}_1$ car $\mathcal{L}_1$ est local.
  - Si $k < n$, on a $a_k a_{k+1} \in S(\mathcal{L}_1)P(\mathcal{L}_2)$ donc $a_k \in S(\mathcal{L}_1)$ et $a_{k+1} \in P(\mathcal{L}_2)$. On prouve alors que $a_0 \dots a_k \in \mathcal{L}_1$ et $a_{k+1} \dots a_n \in \mathcal{L}_2$ avec les mêmes arguments.

  Finalement, $w \in \mathcal{L}_1\mathcal{L}_2$ et $\mathcal{L}$ est local.

■

Théorème 38 — La fermeture de Kleene d'un langage local est un langage local.

Démonstration. On pose $P = P(\mathcal{L})$, $S = S(\mathcal{L})$ et $N = N(\mathcal{L})$, les ensembles définis comme en 198.

On a :

$$P(\mathcal{L}^*) = \{a \in \Sigma, \exists w \in \Sigma^*, aw \in \mathcal{L}^*\}$$

$$S(\mathcal{L}^*) = \{a \in \Sigma, \exists w \in \Sigma^*, wa \in \mathcal{L}^*\}$$

$$F(\mathcal{L}^*) = \{v \in \Sigma^*, |v| = 2, \exists u, w \in \Sigma^*, uvw \in \mathcal{L}^*\}$$

On considère un mot $u = a_0 \dots a_n \in (P(\mathcal{L}^*)\Sigma^* \cap \Sigma^* S(\mathcal{L}^*)) \setminus (\Sigma^* N(\mathcal{L}^*)\Sigma^*)$ et on cherche à montrer qu'il appartient à $\mathcal{L}^*$

De la même manière que précédemment, si $a_0 \in \Sigma$ alors $a_0 \in P(\mathcal{L})$. Les facteurs de longueur 2 de w sont dans $F(\mathcal{L})$ et $a_n \in \mathcal{L}$. On en déduit une décomposition de w en mots dans $(P(\mathcal{L})\Sigma^* \cap \Sigma^* S(\mathcal{L})) \setminus (\Sigma^* N(\mathcal{L})\Sigma^*)$ donc dans $\mathcal{L}$ car $\mathcal{L}$ est local. Finalement, $w \in \mathcal{L}^*$, car qui peut le plus peut le moins et $\mathcal{L}^*$ est local. ■

b Expressions régulières linéaires

Les langages définis par des expressions régulières ne sont donc pas toujours locaux. En revanche les langages définis par une expression régulière linéaire le sont.

■ **Définition 200 — Expression régulière linéaire.** Une expression régulière e sur Σ est linéaire si toute lettre de Σ apparaît **au plus une fois** dans e.

■ **Exemple 62 — Expression régulière linéaire.** L'expression $(ab)^*$ est linéaire mais pas $(ab)^*a^*$.

Théorème 39 — Toute expression régulière linéaire dénote un langage local.

Démonstration. On procède par induction structurelle et en utilisant les propriétés des langages locaux.

Cas de base : $\emptyset$, ϵ et $a \in \Sigma$ sont des expressions régulières linéaires.

- $\mathcal{L}_{ER}(\emptyset) = \emptyset$ et on a bien $(P\Sigma^* \cap \Sigma^* S) \setminus (\Sigma^* N \Sigma^*) = \emptyset \setminus (\Sigma^* \Sigma^2 \Sigma^*) = \emptyset$
- $\mathcal{L}_{ER}(\epsilon) = \epsilon$ et on a bien $(P\Sigma^* \cap \Sigma^* S) \setminus (\Sigma^* N \Sigma^*) = \epsilon \setminus (\Sigma^* (\Sigma^2 \setminus \{\epsilon\}) \Sigma^*) = \epsilon$
- $\mathcal{L}_{ER}(a) = \{a\}$ et on a bien $(P\Sigma^* \cap \Sigma^* S) \setminus (\Sigma^* N \Sigma^*) = \{a\} \setminus (\Sigma^* (\Sigma^2 \setminus \{a\}) \Sigma^*) = \{a\}$

Pas d'induction :

(union) Soit Σ_1 et Σ_2 des alphabets **disjoints**. Soient e_1 et e_2 deux expressions régulières linéaires définies respectivement sur Σ_1 et Σ_2. Alors $e_1|e_2$ est régulière et linéaire et la sémantique des expressions régulières permet d'affirmer que $\mathcal{L}_{ER}(e_1|e_2) = \mathcal{L}_{ER}(e_1) \cup \mathcal{L}_{ER}(e_2)$. Or, l'union de deux langages locaux dont les alphabets sont disjoints est un langage local. Donc $\mathcal{L}_{ER}(e_1|e_2)$ est un langage local.

(concaténation) on procède de même en utilisant la concaténation de deux langages locaux.

(union) on procède de même en utilisant la fermeture de Kleene de deux langages locaux.

Finalement, les expressions régulières linéaires dénotent des langages locaux. ■

R La réciproque de ce théorème est fausse : par exemple, le langage $L(aa^*)$ est local mais aa^* n'est pas une expression régulière linéaire.

c Automate locaux

Automates locaux

■ **Définition 201 — .** Un automate fini déterministe $\mathcal{A} = (Q, \Sigma, q_i, \delta, F)$ est local si pour toute lettre $a \in \Sigma$, il existe un état $q \in Q$ tel que toutes les transitions étiquetées par a arrivent dans q.

Théorème 40 — Tout langage local $\mathcal{L}$ est reconnaissable par un automate local. De plus, si $\mathcal{L}$ ne contient pas le mot vide ϵ, alors l'automate est normalisé.

Démonstration. Soit $\mathcal{L}$ un langage local. On pose $P = P(\mathcal{L})$, $S = S(\mathcal{L})$ et $N = N(\mathcal{L})$, les ensembles définis comme en 198.

On considère l'automate $\mathcal{A} = (Q, \Sigma \cup \{\epsilon\}, q_0, \delta, S \cup \{\epsilon\})$ et la fonction δ définie par :

$$\forall a \in P, \delta(q_0, a) = q_a \tag{14.6}$$

$$\forall a_1 a_2 \in F, \delta(q_{a_1}, a_2) = q_{a_2} \tag{14.7}$$

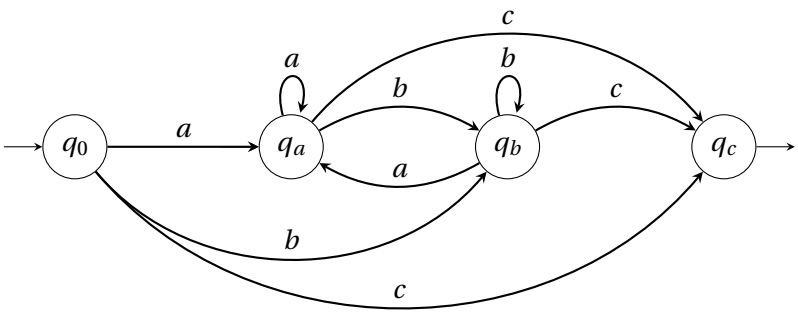
$$\tag{14.8}$$

Par construction de cet automate, un mot $w = a_0 a_1 \dots a_n$ est reconnu si et seulement si :

- $a_0 \in P$,
- $\forall i \in [\![0, n-1]\!], a_i a_{i+1} \in F$
- et $a_n \in S$.

Comme $\mathcal{L}$ est local, on a bien $\mathcal{L}_{auto}(\mathcal{A}) = \mathcal{L}$. Si $\mathcal{L}$ ne contient pas le mot vide, il suffit de l'exclure des états accepteurs et on obtient un automate normalisé. ■

■ **Exemple 63 — Automate associé à l'expression régulière linéaire** $(a|b)^*c$ **.** Le langage dénoté par cette expression régulière est local (le montrer !). On construit l'automate défini lors de la démonstration du théorème 40. Comme ce langage ne comporte pas le mot vide, l'automate est normalisé.



d Automate de Glushkov et algorithme de Berry-Sethi

On cherche maintenant un algorithme pour transformer une expression régulière (pas nécessairement linéaire) en un automate fini.

M **Méthode 6 — Algorithme de Berry-Sethi** Pour obtenir un automate fini local reconnaissant le langage $\mathcal{L}_{ER}(e)$ à partie d'une expressions régulière e sur un alphabet Σ :

1. Linéariser l'expression e : cela consiste à numéroter toutes les lettres qui apparaissent afin de créer une expression rationnelle linéaire e'.
2. Déterminer les ensembles P, S et F associés au langage local $\mathcal{L}(e')$.
3. Déterminer un automate **local** $\mathcal{A}$ reconnaissant $\mathcal{L}_{ER}(e')$ à partir de P, S et F. On peut associer ses états aux les lettres de l'alphabet de e'. L'état initial est relatif au mot vide : s'il appartient au langage, on fait de cet état un état accepteur. Toutes les transitions qui partent de l'état initial conduisent à un état associé à une lettre de P. Les états accepteurs sont associés aux éléments de S. Les facteurs de deux lettres déterminent les autres transitions.
4. Supprimer les numéros sur les transitions et faire réapparaître l'alphabet initial Σ.

L'automate obtenu est nommé automate de Glushkov[7]. C'est un automate **local et sans tran-**

sitions spontanées. Il n'est pas nécessairement déterministe mais on peut le déterminiser facilement en utilisant la procédure de déterminisation d'un AFND (cf méthode 4). Il possède $|\Sigma_e|$ états où Σ_e est l'alphabet étendu obtenu en faisant le marquage. $|\Sigma_e|$ correspond donc au nombre total de lettres dans e en comptant les répétitions. Dans le pire des cas, le nombre de transitions de l'automate est en $O(|\Sigma_e^2|)$.

Méthode 7 — Linéarisation de l'expression régulière À partir de l'expression régulière de départ, on numérote à partir de 1 chaque lettre de l'expression dans l'ordre de lecture.

Si une lettre apparaît plusieurs fois, elle est numérotée autant de fois qu'elle apparaît avec un numéro différent.

Par exemple, la linéarisation de $ab|(ac)^*$ est $a_1b_1|(a_2c_1)$.

La linéarisation d'une expression est en fait un marquage par une fonction

$$m : \{a_1, a_2, \ldots, b_1, \ldots,\} \longrightarrow \Sigma \tag{14.9}$$

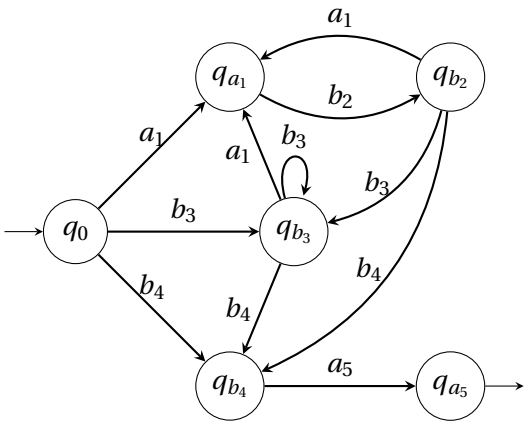
que l'on utilise également à la fin de l'algorithme de Berry-Sethi 6 pour supprimer les numéros.

La complexité de l'algorithme de Berry-Sethi est quadratique dans le pire des cas. Si n est le nombre de lettres recontrées dans l'expression linéarisée, on a :

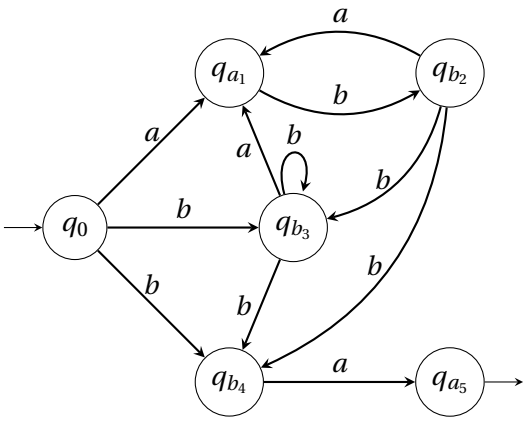
- la linéarisation est en $O(n)$,
- la construction des ensemble P, S, et F est linéaire en $O(n)$,
- la construction de l'automate avec les ensembles précédents est au pire quadratique en $O(n^2)$ (on construit n chemins qui peuvent être de longueur n),
- la suppression des marquages est linéaire en $O(n)$.

■ **Exemple 64 — Construire l'automate de Glushkov associé à l'expression régulière** $(ab|b)^*ba$**.**
On applique l'algorithme de Berry-Sethi :

1. Linéarisation : $(ab|b)^*ba \longrightarrow (a_1b_2|b_3)^*b_4a_5$,
2. Construction des ensembles associés au langage local de l'expression linéarisée :
   - $P = \{a_1, b_3, b_4\}$
   - $S = \{a_5\}$
   - $F = \{a_1b_2, b_2a_1, b_2b_4, b_3b_3, b_3b_4, b_3a_1, b_4a_5, b_2b_3\}$
3. Construction de l'automate local associé :



Suppression des marquages des transitions (numéros) :

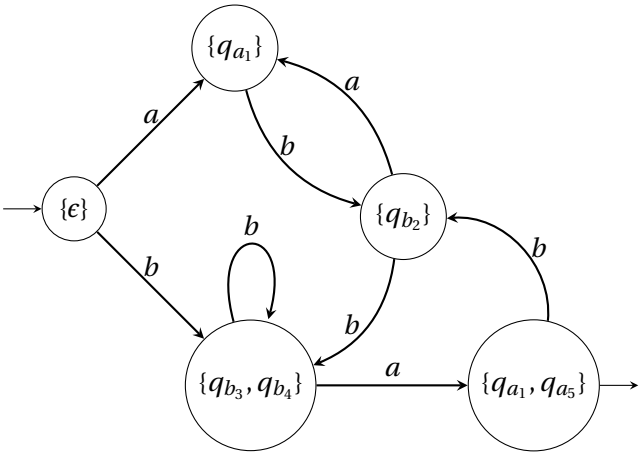


Déterminisation :

On construit la fonction de transition de proche en proche à partir de l'état initial :

	$\downarrow q_0$	$\{q_{a_1}\}$	$\{q_{b_2}\}$	$\{q_{b_3}, q_{b_4}\}$	$\uparrow\{q_{a_1}, q_{a_5}\}$
a	$\{q_{a_1}\}$		$\{q_{a_1}\}$	$\{q_{a_1}, q_{a_5}\}$	
b	$\{q_{b_3}, q_{b_4}\}$	$\{q_{b_2}\}$	$\{q_{b_3}, q_{b_4}\}$	$\{q_{b_3}, q_{b_4}\}$	$\{q_{b_2}\}$

ce qui se traduit par l'AFD :



D Comparaison Thompson / Berry-Sethi

Étape	Thompson	Berry-Sethi
Préparation	Aucune	Linéariser l'expression Construire les ensembles P, S et F
Automate	Construction directe en $O(n)$	Construction en $O(n^2)$
Finition	Supprimer les transitions spontanées	Aucun

TABLE 14.2 – Comparaison des algorithmes de Thompson et Berry-Sethi.

15

DES AUTOMATES AUX EXPRESSIONS RATIONNELLES

À la fin de ce chapitre, je sais :

- ☞ expliquer ce qu'est un automate généralisé
- ☞ fusionner des transitions multiples
- ☞ éliminer des états
- ☞ passer d'un automate à une expression régulière

Ce chapitre permet d'apporter une démonstration à la réciproque du théorème de Kleene, à savoir qu'un langage reconnaissable est un langage régulier. On s'appuie pour cela sur l'algorithme d'élimination des états qui nécessite le concept d'automate généralisé.

A Automate généralisé

■ **Définition 202 — Automate généralisé.** Soit Σ un alphabet et $\mathcal{E}_R$ l'ensemble des expressions régulières sur un Σ. Un automate généralisé est un automate tel que $\mathcal{A} = (Q, \mathcal{E}_R, Q_i, \Delta, F)$, c'est-à-dire un automate dont les étiquettes des arcs sont des expressions régulières.

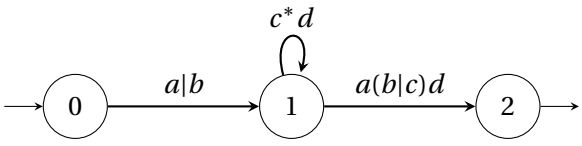


FIGURE 15.1 – Exemple d'automate généralisé sur l'ensemble des expressions régulières sur $\Sigma = \{a, b, c, d\}$

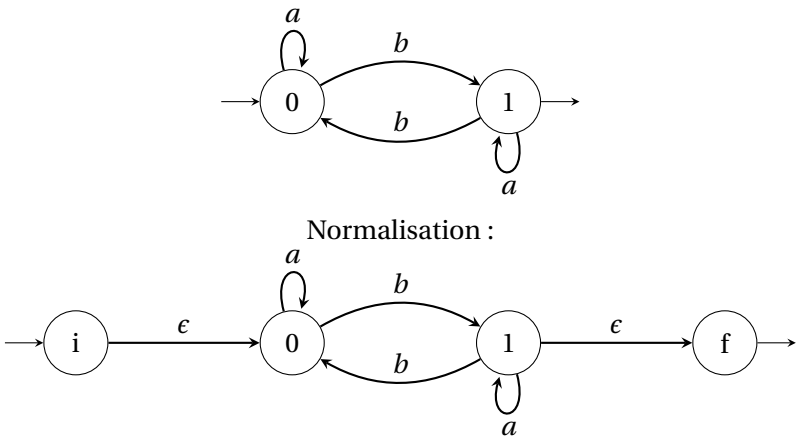


FIGURE 15.2 – Normalisation d'un automate

B D'un automate généralisé à une expression régulière

Soit un automate $\mathcal{A}$ **normalisé**. On souhaite trouver une expression régulière e telle que $\mathcal{L}(\mathcal{A}) = \mathcal{L}_{ER}(e)$, c'est-à-dire le langage reconnu par l'automate $\mathcal{A}$ est le même que celui dénoté par e.

■ **Définition 203 — Automate normalisé.** Un automate est normalisé s'il ne possède pas de transition entrante sur son état initial et s'il possède un seul état final sans transition sortante.

R Si l'automate n'est pas normalisé, on ajoute un état initial avec une transition spontanée vers l'état initial et un état final relié par des transitions spontanées depuis les états accepteurs.

■ **Exemple 65 — Normalisation d'un automate.** La figure 15.2 montre un automate et sa version normalisée. Il faut noter que si l'automate possède plusieurs états accepteurs, il faut relier tous ces états accepteurs au nouvel état final.

La construction de Brzozowski et McCluskey est intuitive et facile à programmer. Il s'agit d'éliminer un à un les états de l'automate généralisé associé à $\mathcal{A}$. À la fin de la procédure, il ne reste plus que deux états reliés par un seul arc étiqueté par une seule expression régulière e. Le langage dénoté par cette dernière est le langage reconnu par l'automate.

M **Méthode 8 — Construire l'expression régulière équivalent à un automate normalisé**
Deux grandes étapes sont nécessaires pour construire l'expression régulière équivalent à un automate. **Pour chaque état `q` à éliminer**, c'est-à-dire les états autres que l'état initial ou l'état

final,

1. **fusionner** les expressions régulières des transitions au départ de q_s et à destination du même état q_n comme illustré sur la figure 15.3. Formellement, si on a les transitions (q_s, e_1, q_n) et (q_s, e_2, q_n), alors on fusionne les deux expressions en faisant leur somme : $(q_s, e_1|e_2, q_n)$. On ne conserve ainsi qu'une seule expression par destination au départ de q_s.
2. **éliminer l'état** q_s en mettant à jour les transitions au départ des états précédents comme l'illustre la figure 15.4. Considérons chaque transition de type (q_p, e_1, q_s) et (q_s, e_2, q_n), c'est-à-dire les transitions pour lesquelles q_s intervient. Si on souhaite éliminer q_s, il faut considérer à chaque fois deux cas :
   (a) une transition boucle (q_s, e_b, q_s) existe : alors il est nécessaire d'ajouter la transition $(q_p, e_1 e_b^* e_2, q_n)$,
   (b) dans le cas contraire, il suffit d'ajouter la transition $(q_p, e_1 e_2, q_n)$.

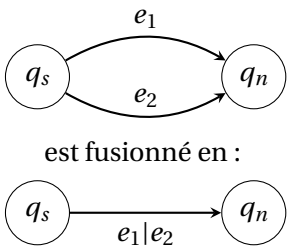


FIGURE 15.3 – Fusion de deux arcs au départ d'un état q_s et à destination du même état q_n

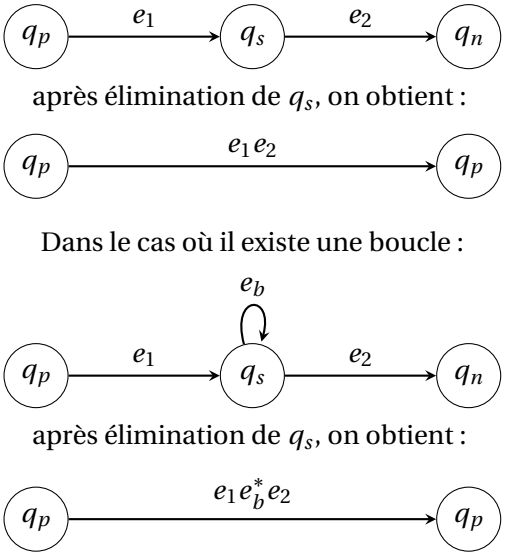
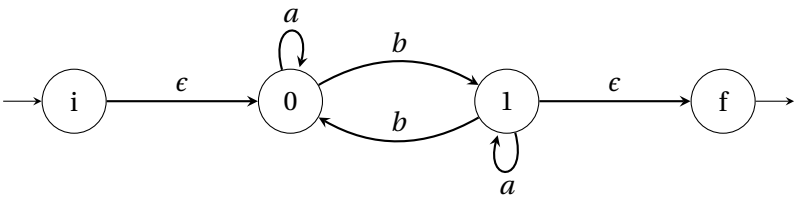
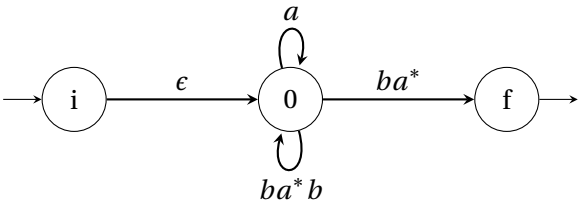


FIGURE 15.4 – Élimination d'un état q_s.

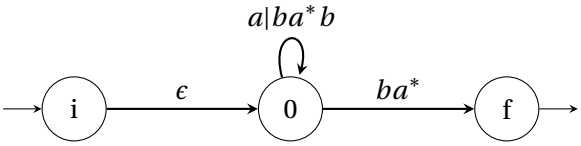
■ **Exemple 66 — Élimination des états d'un automate généralisé et normalisé .** Considérons l'automate normalisé de la figure 15.2.



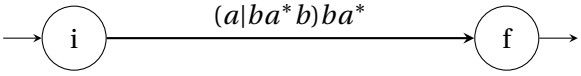
Élimination de l'état 1 :



Fusion des arcs partant de 0 à destination de 0 :



Élimination de l'état 0 :



L'expression régulière équivalente à l'automate est donc $(a|ba^*b)ba^*$.

16

AU-DELÀ DES LANGAGES RÉGULIERS

À la fin de ce chapitre, je sais :

- ☞ expliquer les limites des langages réguliers
- ☞ montrer qu'un langage n'est pas régulier

A Limites des expressions régulières

Les langages réguliers permettent de reconnaître un motif dans un texte. Néanmoins, ils ne permettent pas de mettre un sens sur le motif reconnu : celui-ci est reconnu par l'automate mais en quoi est-il différent d'un autre mot reconnu par cet automate ? Par exemple, on peut reconnaître les mots qui se terminent par *tion* mais on ne saura pas faire la différence sémantique entre *révolution* et *abstention*.

Un autre exemple classique est l'interprétation des expressions arithmétiques : comment comprendre que $a \times b - c$ se calcule $(a \times b) - c$ et pas $a \times (b - c)$. Les deux motifs sont des expressions arithmétiques valides mais elle ne s'interprètent pas de la même manière. C'est là une des limites des langages réguliers : une fois motif reconnu, on ne peut pas l'interpréter. Pour la dépasser, il faut utiliser les notions de grammaires --→ HORS PROGRAMME .

Une autre question se pose : comment savoir si un langage est régulier sans pour autant exhiber un automate ? Comment caractériser formellement un langage régulier ?

B Caractériser un langage régulier

Théorème 41 — Lemme de létoile. Soit $\mathcal{L}$ un langage sur un alphabet Σ reconnu par un automate $\mathcal{A}$ à n états. Alors on a :

$$\forall w \in \mathcal{L}, |w| \geqslant n \Longrightarrow \exists x, y, z \in \Sigma^*, w = xyz, |xy| \leqslant n, y \neq \epsilon \text{ et } xy^*z \subseteq \mathcal{L} \tag{16.1}$$

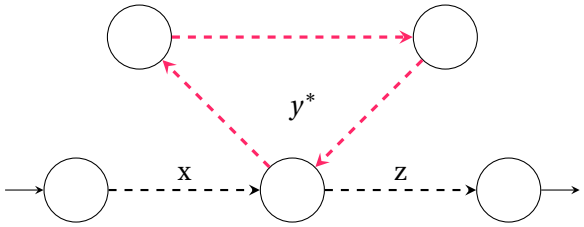


FIGURE 16.1 – Illustration du lemme de l'étoile : si le nombre de lettres d'un mot reconnu w est plus grand que le nombre d'états de l'automate n, alors il existe une boucle sur laquelle on peut itérer.

Démonstration. Soit w un mot reconnu par l'automate $\mathcal{A}$ à n états de longueur m. Il existe un chemin dans $\mathcal{A}$ qui part de l'état initial q_0 et s'achève sur un état accepteur q_m.

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_m} q_m$$

En numérotant de manière incrémentale les états de 0 à m, on a nécessairement $m \geqslant n$. D'après le principe des tiroirs, comme l'automate ne possède que n états, ce chemin repasse par certains états. Prenons le premier état par lequel le chemin repasse et notons le i. Il existe donc deux entiers i et j tels que $0 < i < j \leqslant n < m$ et $q_i = q_j$, c'est-à-dire il existe un cycle de longueur $j - i$ sur le chemin. Comme il s'agit du premier état par lequel on repasse, les état q_0 jusqu'à q_{j-1} sont tous distincts.

On choisit alors de poser $x = a_1 \dots a_{i-1}$, $y = a_i \dots a_{j-1}$ et $z = a_j \dots a_m$. On remarque que $w = xyz$ et que x et xy vérifient les propriétés du lemme de l'étoile car y n'est pas vide et $|xy| \leqslant n$. Il reste à montrer que $xy^*z \subseteq \mathcal{L}$. Comme le chemin reconnaissant y est un cycle, on peut le parcourir autant de fois que l'on veut, 0 ou k fois, le mot sera toujours reconnu par l'automate. ■

Théorème 42 — Principes des tiroirs. Si $n+1$ éléments doivent être placés dans n ensembles, alors il existe au moins un ensemble qui contient au moins 2 éléments. Autrement dit, si E et F sont deux ensembles finis tels que $|E| > |F|$, alors il n'existe aucune application injective de E dans F.

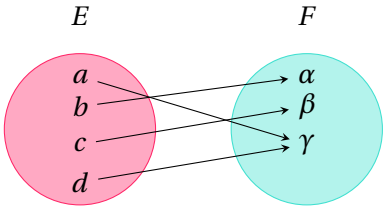


FIGURE 16.2 – Illustration du principe des tiroirs : on ne peut pas ranger les éléments de E dans les tiroirs de F sans en mettre deux dans un tiroir.

R Le lemme de l'étoile est parfois appelé le lemme de l'itération car on peut itérer autant de fois que l'on veut y.

Vocabulary 14 — Pumping lemma ⇜⇝ Lemme de l'étoile

■ **Définition 204 — Constante d'itération et facteur itérant.** D'après le lemme de l'itération, il existe un entier naturel N tel que chaque mot w de $\mathcal{L}$ tel que $|w| \geqslant N$ possède au moins un facteur non vide y pouvant être itéré.

On dit alors que N est une constante d'itération pour le langage $\mathcal{L}$ et que y est un facteur itérant.

■ **Définition 205 — Constante d'itération minimale.** Il s'agit de la plus petite constante d'itération d'un langage $\mathcal{L}$.

■ **Exemple 67 — Exemples de constantes minimales d'itération.** On considère les langages dénotés par des expressions régulières et on calcule la constante minimale d'itération du langage :

ab ⟶ 3, car le seul mot reconnu par le langage est ab. On ne peut pas itérer ce mot. De plus, il n'y a pas de mots de longueur supérieure à 2. C'est pourquoi, tous les mots (qui n'existent pas) de longueur supérieure ou égale à 3 peuvent être itérés.

aab^* ⟶ 3, car le plus petit mot du langage est aa mais il ne peut pas être itéré. Soit un mot w de longueur 3. On peut le décomposer comme suit : $w = xyz$, $x = aa$, $y = b$ et $z = \epsilon$. Cette décomposition satisfait le lemme de l'étoile.

$(a|b)^*$ ⟶ 1. On observe que $\epsilon \in \mathcal{L}_{ER}((a|b)^*)$, cependant il ne peut pas être itéré. Donc, la constante minimale ne peut pas être égale à zéro. Soit w un mot de longueur 1. Il vaut a ou b. Dans le premier cas, on peut choisir la décomposition $w = xyz$ avec $x = \epsilon$, $y = a$, $z = \epsilon$, dans le deuxième $x = \epsilon$, $y = b$, $z = \epsilon$. Dans les deux cas, la décomposition satisfait le lemme de l'étoile.

R Il faut remarquer que le lemme de l'étoile peut être vérifier par un langage non régulier. C'est pourquoi, la plupart du temps, on utilise le lemme de l'étoile pour montrer qu'un langage n'est pas régulier : s'il ne le vérifie pas, il n'est pas régulier.

C Les langages des puissances

■ **Définition 206 — Langage des puissances.** On appelle langage des puissances le langage défini par :

$$\mathcal{L}_p = \{a^n b^n, n \in \mathbb{N}\} \tag{16.2}$$

Théorème 43 — Le langage des puissances n'est pas régulier.

Démonstration. Par l'absurde en utilisant le lemme de l'étoile.

Supposons que $\mathcal{L}_p$ soit régulier. Alors il vérifie le lemme de l'étoile. Soit n un entier naturel, une constante d'itération de ce langage. Considérons $w = a^n b^n \in \mathcal{L}_p$. On a bien $|w| = 2n \geqslant n$. On peut donc appliquer le lemme de l'étoile à w.

Soient x, y et z, les mots formant la décomposition de $w = xyz$. D'après le lemme de l'étoile, $|xy| \leqslant n$. Il existe donc des entiers naturels i et $j > 0$ tels que $x = a^i$, $y = a^j \neq \epsilon$, $xy = a^{i+j}$ et $i + j \leqslant n$. On peut réécrire la décomposition comme suit : $w = xyz = a^i a^j a^{n-i-j} b^n$, c'est-à-dire que $z = a^{n-i-j} b^n$.

Le lemme de l'étoile affirme qu'on peut itérer sur y et appartenir toujours au langage. Donc le mot $xy^2z = a^i a^{2j} z = a^i a^{2j} a^{n-i-j} b^n$ devrait appartenir à $\mathcal{L}_p$. Or ce n'est manifestement pas le cas car $i + 2j + n - i - j = n + j > n$ car $j > 0$. C'est pourquoi $\mathcal{L}_p$ n'est pas un langage régulier.

NB : on aurait pu également étudier le mot xy^0z et aboutir à la même conclusion. ■

R C'est un résultat à connaître car on peut s'en servir pour démontrer la non régularité d'autres langages. La démonstration est également typique de l'utilisation du lemme de l'étoile.

Septième partie

Annexes

BIBLIOGRAPHIE

Articles

[1] Walter William Rouse BALL. "The eight queens problem". In : *Mathematical recreations and essays* (1960), pages 97-102 (cf. page 52).

[2] Richard BELLMAN. "On a routing problem". In : *Quarterly of applied mathematics* 16.1 (1958), pages 87-90 (cf. page 87).

[3] Gerard BERRY et Ravi SETHI. "From regular expressions to deterministic automata". In : *Theoretical computer science* 48 (1986). Publisher : Elsevier, pages 117-126 (cf. page 155).

[4] E. W. DIJKSTRA. "A note on two problems in connexion with graphs". In : *Numerische Mathematik* 1.1 (1er déc. 1959), pages 269-271. ISSN : 0945-3245. DOI : 10.1007/BF01386390. URL : https://doi.org/10.1007/BF01386390 (visité le 27/07/2022) (cf. page 83).

[5] Robert W FLOYD. "Algorithm 97 : shortest path". In : *Communications of the ACM* 5.6 (1962). Publisher : ACM New York, NY, USA, page 345 (cf. page 89).

[7] Victor Mikhaylovich GLUSHKOV. "The abstract theory of automata". In : *Russian Mathematical Surveys* 16.5 (1961). Publisher : IOP Publishing, page 1 (cf. pages 155, 160).

[8] Donald B JOHNSON. "Efficient algorithms for shortest paths in sparse networks". In : *Journal of the ACM (JACM)* 24.1 (1977). Publisher : ACM New York, NY, USA, pages 1-13 (cf. page 87).

[9] Joseph B KRUSKAL. "On the shortest spanning subtree of a graph and the traveling salesman problem". In : *Proceedings of the American Mathematical society* 7.1 (1956). Publisher : JSTOR, pages 48-50 (cf. page 93).

[10] Casimir KURATOWSKI. "Sur le probleme des courbes gauches en topologie". In : *Fundamenta mathematicae* 15.1 (1930), pages 271-283 (cf. page 57).

[13] Robert Clay PRIM. "Shortest connection networks and some generalizations". In : *The Bell System Technical Journal* 36.6 (1957). Publisher : Nokia Bell Labs, pages 1389-1401 (cf. page 93).

[14] Bernard ROY. "Transitivité et connexité". In : *Comptes Rendus Hebdomadaires Des Seances De L Academie Des Sciences* 249.2 (1959). Publisher : GAUTHIER-VILLARS/EDITIONS ELSEVIER 23 RUE LINOIS, 75015 PARIS, FRANCE, pages 216-218 (cf. page 89).

[16] Stephen WARSHALL. "A theorem on boolean matrices". In : *Journal of the ACM (JACM)* 9.1 (1962). Publisher : ACM New York, NY, USA, pages 11-12 (cf. page 89).

Livres

[11] Édouard LUCAS. *Récréations mathématiques*. Tome 2. Gauthier-Villars et fils, 1883 (cf. page 65).

Sites web