Cours d'Option Informatique

Jules Svartz
Lycée Masséna

Préambule

Ce polycopié est issu du cours d'option informatique dispensé aux élèves du lycée Masséna des classes de première année MPSI (831 et 832), et de deuxième année MP*.

Le polycopié se divise en deux parties, découpage induit par le programme officiel entre première et deuxième années. Le dernier chapitre (chapitre 15), présente les modules usuels en Ocaml et est donc transversal.

— Le programme de première année est assez ambitieux vu le temps imparti, car l'option informatique ne débute qu'au second semestre. Il faut, durant ce laps de temps, acquérir la connaissance du langage Caml(light) et les bases de l'informatique théorique. Le découpage choisi est le suivant.
  — Le chapitre 0 est un chapitre d'introduction au langage OCaml, où la comparaison avec le langage Python est souvent faite : on s'appuie sur les connaissances des élèves en Python pour progresser rapidement, c'est pourquoi les aspects les plus fonctionnels du langage (structure de liste chaînée et récursivité) sont traités ultérieurement.
  — Le chapitre 1 présente les structures de données usuelles au programme, qui seront un fil conducteur dans l'enseignement des deux années : piles, files, files de priorité et dictionnaires. Leurs implémentations concrètes seront vues tout au long du polycopié, on donne en toute fin de document les modules usuels en Ocaml, et une implémentation complète d'un module pour les files de priorité.
  — Le chapitre 2 traite de l'étude théorique des algorithmes (terminaison, correction et complexité), qui ne font pas usage de récursivité : on se concentre sur les aspects impératifs de Caml, le chapitre en lui-même étant très proche de ce que les élèves ont déja vu dans le cours commun d'informatique. Néanmoins, la fin du chapitre est dédiée à l'implémentation des structures de pile et de file dans un tableau.
  — Le chapitre 3 aborde la récursivité, et les listes chaînées.
  — Le chapitre 4 donne les outils utiles pour l'analyse des algorithmes récursifs. Il est plus mathématique que les précédents, car la notion de bon ordre est omniprésente pour traiter les questions de terminaison et de correction. Les récurrences usuelles dans l'étude de la complexité des algorithmes récursifs, en particulier « diviser pour régner », sont abordées ici.
  — Le chapitre 5 présente des exemples d'algorithmes « diviser pour régner », que l'on sait analyser depuis le chapitre précédent.
  — Le chapitre 6 est une introduction à la programmation dynamique, les méthodes gloutonnes sont également évoquées.
  — Enfin, le chapitre 7 est une introduction aux arbres, implémentés uniquement de manière persistante ici. Après un descriptif mathématique, on se concentre sur l'étude et l'implémentation des arbres binaires et arbres binaires entiers, plusieurs implémentations sont proposées.
— Le programme de deuxième année est lui aussi ambitieux, riche et très varié. Ce cours ayant été écrit pour une classe de MP*, on se cantonne rarement au programme officiel. En effet certaines questions émergent naturellement [1], et il aurait été dommage de ne pas les traiter. Néanmoins, on reste raisonnable : ce cours n'est pas une introduction complète à l'algorithmique et a pour but d'être intégralement traité dans l'année.
  — Le chapitre 8 est un prolongement du chapitre 7 : on réalise une structure (impérative) de file de priorité à l'aide d'un tas stocké dans un tableau. On réalise également une structure de dictionnaire à l'aide d'arbres binaires de recherche. Pour garantir une structure efficace, il faut que les arbres soient un minimum équilibrés, c'est pourquoi on s'écarte légèrement du programme officiel en présentant également les arbres AVL.
  — Le chapitre 9 est un court chapitre présentant les preuves par induction. Ce n'est pas au programme, mais ce chapitre a été initialement traité en cours parce qu'il a été réclamé par des élèves.

1. Par exemple, on définit les composantes fortement connexes d'un graphe orienté, mais dans le programme officiel ne figure aucun algorithme permettant de les calculer !

— Le chapitre 10 présente le cours de logique. Bien qu'assez étoffé, il reste dans les clous du programme officiel, en se restreignant aux expressions logiques sans quantificateurs. Sont évoquées les différences entre syntaxe et sémantique, les formes canoniques, et les notions de formules satisfiable ou tautologique. L'écriture d'un programme testant la satisfiabilité d'une expression logique est en général effectuée en travaux pratiques.

— Le chapitre 11 traite des graphes non pondérés. Ce chapitre aborde notamment les parcours en profondeur et en largeur, et le calcul des composantes connexes d'un graphe non orienté. On s'éloigne un peu du programme officiel pour étudier les graphes orientés sans circuit ainsi que le calcul des composantes fortement connexes via l'algorithme de Kosaraju. Une application de ce dernier algorithme en lien avec le chapitre précédent est la résolution du problème 2-SAT.

— Le chapitre 12 fait suite au précédent, et traite des graphes pondérés. Le chapitre se concentre, comme le programme officiel, sur des calculs de plus courts chemins. On s'éloigne un peu du programme officiel pour présenter d'autres algorithes que ceux de Dijkstra et de Floyd-Warshall, et on aborde également le problème de l'arbre couvrant minimal, qu'on résout par l'algorithme de Prim, facile à comprendre une fois l'algorithme de Dijkstra étudié.

— Le chapitre 13 traite des langages et expressions rationnelles. Après quelques considérations sur des langages particuliers (langage des mots de Dyck, notamment), on se concentre sur les langages rationnels. Dans la perspective du chapitre suivant, sont abordés les expressions rationnelles linéaires. On termine le chapitre par la résolution (hors programme) d'équations aux langages, via le lemme d'Arden.

— Le chapitre 14 aborde les automates. On reste très proche du programme officiel en étudiant l'algorithme de Berry-Sethi pour construire un automate reconnaissant un langage rationnel dénoté par une expression rationnelle donnée. Néanmoins on se sert du chapitre précédent pour montrer qu'un langage reconnaissable est rationnel, sens hors programme du théorème de Kleene énonçant l'équivalence. Enfin, on aborde le lemme de l'étoile, hors programme, mais très pratique pour montrer qu'un langage n'est pas rationnel.

— Enfin, le chapitre 15 présente brièvement les implémentations en Ocaml des structures usuelles (piles, files, dictionnaires), et donne une implémentation personnalisée d'un module pour les files de priorité, utilisant essentiellement les idées du chapitre 8.

Licence. Cette œuvre est mise à disposition sous licence Attribution - Partage dans les Mêmes Conditions 2.0 France. Pour voir une copie de cette licence, visitez `http://creativecommons.org/licenses/by-sa/2.0/fr/` ou écrivez à Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Table des matières

I Programme de première année **7**

0 Introduction au langage OCaml **9**
0.1 Caml : un langage fonctionnel (permettant la programmation impérative) . . . 9
0.2 Un langage fortement typé . . . 10
0.3 Types simples . . . 10
0.3.1 Type « rien » . . . 10
0.3.2 Entier . . . 10
0.3.3 Flottants . . . 11
0.3.4 Booléens . . . 11
0.4 Déclaration, déclaration locale, déclaration simultanée, références . . . 11
0.4.1 Déclaration . . . 11
0.4.2 Déclaration locale . . . 12
0.4.3 Déclaration simultanée . . . 12
0.4.4 Références . . . 12
0.5 Quelques types plus complexes . . . 13
0.5.1 Tuples . . . 13
0.5.2 Tableaux . . . 13
0.5.3 Chaînes de caractères . . . 14
0.6 Fonctions . . . 15
0.6.1 Quelques exemples . . . 15
0.6.2 Fonctions à un seul argument. Appel . . . 15
0.6.3 Curryfication des fonctions . . . 16
0.6.4 Création de fonctions curryfiées . . . 16
0.6.5 Création de fonctions non curryfiées . . . 17
0.7 Conditions et boucles en Caml . . . 17
0.7.1 Expressions conditionnelles . . . 17
0.7.2 Séquence d'instructions . . . 18
0.7.3 Boucles . . . 18
0.8 Filtrages . . . 19
0.8.1 Introduction . . . 19
0.8.2 Règles du filtrage sur motif . . . 19
0.9 Types enregistrement et types somme . . . 21
0.9.1 Types enregistrement . . . 21
0.9.2 Types somme . . . 23
0.10 Exceptions . . . 24

1 Structure de données usuelles **27**
1.1 Introduction . . . 27
1.2 Piles . . . 27
1.3 Files . . . 28
1.4 Files de priorité . . . 28
1.5 Dictionnaires . . . 29

2 Programmation impérative en Caml **31**
2.1 Analyse d'algorithmes : terminaison, correction, complexité . . . 31
2.1.1 Terminaison . . . 31
2.1.2 Correction . . . 31
2.1.3 Complexité . . . 32
2.1.4 Un exemple important : l'exponentiation . . . 33
2.2 Les tableaux en Caml . . . 34
2.2.1 Quelques exemples de fonctions basiques . . . 34
2.2.2 Recherche dichotomique dans un tableau trié . . . 35
2.2.3 Exemples : les algorithmes de tri . . . 36
2.3 Implémentation d'une pile et d'une file dans un tableau . . . 37
2.3.1 Piles . . . 38
2.3.2 Files . . . 39

3 Récursivité et listes **41**
3.1 Exemple introductif : la fonction factorielle . . . 41
3.2 Pratique de la récursivité . . . 41
3.2.1 Ce qui se passe « en interne » : la pile d'appels . . . 41
3.2.2 Récursivité terminale . . . 42
3.2.3 Deux exemples de fonctions récursives . . . 43
3.2.4 Récursivité croisée . . . 44
3.2.5 Attention aux appels qui se chevauchent ! . . . 45
3.3 Un exemple plus complet : les tours de Hanoï . . . 45
3.4 Structure de liste chaînée . . . 47
3.4.1 Définition . . . 47
3.4.2 Le type list en Caml . . . 48
3.4.3 Exemples de fonctions sur les listes . . . 49
3.4.4 Construction de listes . . . 50
3.4.5 Une implémentation personnalisée des listes . . . 50
3.4.6 Implémentation de structures de pile et de file à l'aide de listes chaînées . . . 50
3.5 Un exemple fondamental : le tri fusion . . . 52

4 Analyse des fonctions récursives **55**
4.1 Introduction . . . 55
4.2 Terminaison . . . 55
4.3 Correction . . . 58
4.4 Complexité des fonctions récursives . . . 58
4.4.1 Introduction . . . 58
4.4.2 Premiers résultats . . . 59
4.4.3 Récurrences « Diviser pour régner » . . . 60

5 Algorithmes « Diviser pour régner » **63**
5.1 Introduction . . . 63
5.2 Tri fusion : analyse . . . 63
5.2.1 Rappel : le code du tri . . . 63
5.2.2 Analyse des fonctions auxiliaires . . . 63
5.2.3 Analyse de la fonction de tri . . . 64
5.3 Algorithmes de multiplication rapide : polynômes et matrices . . . 64
5.3.1 Multiplication rapide de polynômes : algorithme de Karatsuba . . . 65
5.3.2 Algorithme de Strassen . . . 67
5.4 Calcul de la paire de points les plus proches dans un nuage de points . . . 68
5.4.1 Approche naïve . . . 68
5.4.2 Approche « diviser pour régner » . . . 68

6 Introduction aux arbres **71**
6.1 Les arbres comme objets mathématiques . . . 71
6.1.1 Définitions . . . 71
6.1.2 Un peu de dénombrement . . . 72
6.2 Les arbres en Caml . . . 73
6.2.1 Arbres généraux . . . 73
6.2.2 Arbres binaires entiers . . . 74
6.2.3 Arbres binaires . . . 74
6.3 Parcours d'arbres binaires entiers . . . 75
6.3.1 Parcours en profondeur . . . 75
6.3.2 Parcours en largeur . . . 77

7 Preuves par induction **79**
7.1 Introduction . . . 79
7.2 Définitions inductives . . . 79
7.2.1 Le théorème du point fixe . . . 79
7.2.2 Des exemples . . . 80
7.3 Preuves par induction . . . 81
7.4 Induction non ambiguë et définitions de fonctions sur un ensemble inductif . . . 81
7.4.1 Définition non ambiguë d'un ensemble inductif . . . 81
7.4.2 Ordre sur un ensemble inductif . . . 82
7.4.3 Fonctions sur un ensemble inductif . . . 82
7.4.4 Analyse d'une fonction récursive . . . 83

8 Implémentation des dictionnaires **85**
8.1 Tas et file de priorité . . . 85
8.1.1 Une file de priorité à l'aide d'une liste ? . . . 85
8.1.2 La structure de tas . . . 85
8.1.3 Opérations sur un tas . . . 86
8.1.4 Stockage d'un arbre binaire complet à gauche dans un tableau . . . 88
8.1.5 Implémentation de la structure de file de priorité max dans un tableau . . . 88
8.1.6 Complexité . . . 90
8.1.7 Intermède : le tri par tas . . . 90
8.1.8 Modification d'une clé . . . 90
8.2 Arbres binaires de recherche, Arbres AVL . . . 90
8.2.1 Implémentation d'une structure de dictionnaire avec une liste chaînée . . . 90
8.2.2 Structure d'arbre binaire de recherche . . . 91
8.2.3 Implémentation des ABR en Caml . . . 91
8.2.4 Implémentation de la structure de dictionnaire avec des ABR . . . 92
8.2.5 Arbres AVL . . . 94
8.2.6 Rotations et maintien de la structure d'arbre AVL . . . 94
8.2.7 Opérations sur les arbres AVL en Caml . . . 94

9 Logique **97**
9.1 Introduction . . . 97
9.2 Syntaxe des expressions logiques et représentation arborescente . . . 97
9.2.1 Définitions . . . 97
9.2.2 Représentation arborescente . . . 97
9.2.3 Simplification de l'écriture . . . 97
9.2.4 Implémentation . . . 98
9.3 Sémantique des expressions logiques . . . 98
9.3.1 Distribution de vérité . . . 98
9.3.2 Évaluation d'une expression logique. Équivalence sémantique . . . 99
9.3.3 Expressions logiques à 2 variables . . . 99
9.3.4 Retour au problème de l'introduction . . . 100
9.4 Tautologies, antilogies et formules satisfiables . . . 100
9.4.1 Définitions . . . 100
9.4.2 Les tautologies : la base du raisonnement mathématique . . . 101
9.5 Formes normales . . . 101

9.5.1 Formes normales conjonctives et disjonctives . . . 101
9.5.2 Mise sous forme canonique . . . 102
9.5.3 Formes normales conjonctives et disjonctives canoniques . . . 104
9.6 Le problème SAT . . . 106

II Programme de deuxième année 107

10 Implémentation des files de priorité 109
10.1 Une file de priorité à l'aide d'une liste ? . . . 109
10.2 La structure de tas . . . 109
10.3 Opérations sur un tas . . . 110
10.4 Stockage d'un arbre binaire complet à gauche dans un tableau . . . 112
10.5 Implémentation de la structure de file de priorité max dans un tableau . . . 112
10.6 Complexité . . . 114
10.7 Intermède : le tri par tas . . . 114
10.8 Modification d'une clé . . . 114

11 Introduction au Backtracking 115
11.1 Introduction . . . 115
11.2 Modélisation par un arbre et algorithme générique . . . 115
11.3 Sudoku : résolution pratique . . . 116

12 Graphes non pondérés : compléments et rappels 119
12.1 Introduction . . . 119
12.2 Vocabulaire des graphes et propriétés mathématiques . . . 120
12.2.1 Graphes non orientés . . . 120
12.2.2 Arbres au sens des graphes non orientés. . . . 121
12.2.3 Graphes orientés . . . 122
12.3 Implémentation des graphes . . . 125
12.3.1 Implémentation « générale » . . . 125
12.3.2 Implémentation « creuse » . . . 126
12.3.3 Représentation « dense » . . . 127
12.3.4 D'une représentation à une autre . . . 128
12.4 Parcours de graphes donnés par liste d'adjacence . . . 129
12.4.1 Parcours générique de graphe depuis un sommet source . . . 129
12.4.2 Parcours en largeur et plus courts chemins . . . 129
12.4.3 Parcours en profondeur . . . 131
12.5 Applications du parcours en profondeur . . . 132
12.5.1 Composantes connexes d'un graphe non orienté . . . 132
12.5.2 Tri topologique d'un graphe orienté sans circuit . . . 133
12.5.3 Calcul des composantes fortement connexes d'un graphe orienté (HP) . . . 134
12.6 Résolution du problème 2-SAT . . . 136
12.6.1 Construction d'un graphe à partir d'une instance de 2-SAT . . . 137
12.6.2 À la recherche d'une distribution de vérité . . . 137
12.6.3 Complexité . . . 138

13 Couplage de cardinal maximal dans un graphe biparti 139
13.1 Définitions . . . 139
13.2 Chemin augmentant et recherche d'un couplage de cardinal maximum . . . 140
13.3 Détection de chemins augmentants et complexité de l'algorithme . . . 142

14 Graphes pondérés : compléments et arbres couvrants 143
14.1 Introduction . . . 143
14.2 Définition et représentations des graphes pondérés . . . 143
14.2.1 Pondération . . . 143
14.2.2 Implémentation . . . 143
14.3 Définitions et premières propriétés sur les plus courts chemins . . . 144
14.3.1 Définition . . . 144

14.3.2 Différents problèmes de plus courts chemins . . . 145
14.3.3 Optimalité des solutions aux sous-problèmes . . . 145
14.4 Plus courts chemins à origine unique . . . 145
14.4.1 Relâchement d'arcs . . . 145
14.4.2 Algorithme de Bellman-Ford (HP) . . . 146
14.4.3 Algorithme de Dijkstra . . . 147
14.5 Plus courts chemins pour tous couples de sommets . . . 149
14.5.1 Multiplication de matrices (HP) . . . 150
14.5.2 Algorithme de Floyd-Warshall . . . 151
14.5.3 Résumé des algorithmes de plus courts chemins . . . 153
14.6 Arbre couvrant de poids minimal . . . 154
14.6.1 Algorithme de Prim (HP) . . . 155
14.6.2 Algorithme de Kruskall . . . 156

15 Preuves en logique **159**
15.1 Introduction . . . 159
15.2 Séquent . . . 159
15.3 Règles de la déduction naturelle pour les formules propositionnelles . . . 160
15.4 Correction de la déduction naturelle . . . 162
15.5 Complétude de la déduction naturelle . . . 162

16 Langages et expressions rationnelles **165**
16.1 Introduction . . . 165
16.2 Mots sur un alphabet . . . 166
16.2.1 Définition et structure mathématique . . . 166
16.2.2 Lemme de Lévi et application à la commutativité . . . 166
16.2.3 Mots de Dyck . . . 167
16.3 Langages . . . 168
16.3.1 Définition et cardinalité . . . 168
16.3.2 Opérations sur les langages . . . 169
16.3.3 Expressions rationnelles et langages rationnels . . . 169
16.3.4 Quelques réductions . . . 170
16.4 Langages locaux et expressions rationnelles linéaires . . . 171
16.4.1 Expressions rationnelles linéaires . . . 171
16.4.2 Langages locaux . . . 171
16.4.3 Propriétés de clôture des langages locaux . . . 172
16.5 Implémentation . . . 173

17 Automates **175**
17.1 Introduction . . . 175
17.2 Automates finis déterministes . . . 175
17.2.1 Définitions . . . 175
17.2.2 Équivalence d'automates . . . 176
17.2.3 Automates locaux . . . 178
17.3 Automates non déterministes . . . 179
17.3.1 Définitions . . . 179
17.3.2 Détérminisation d'un automate non déterministe . . . 180
17.3.3 Automate de Glushkov et algorithme de Berry-Sethi . . . 182
17.4 Automates généralisés et élimination des états . . . 184
17.4.1 Automates étiquetés par des expressions rationnelles . . . 184
17.4.2 Automate de Thomson associé à une expression rationnelle . . . 185
17.4.3 Algorithme d'élimination des états . . . 186
17.4.4 Théorème de Kleene . . . 188
17.5 Stabilité des langages rationnels . . . 188
17.5.1 Opérations ensemblistes . . . 188
17.5.2 Préfixes, suffixes, facteurs, sous-mots, miroir... . . . 189
17.6 Lemme de l'étoile . . . 189
17.7 Application à la reconnaissance de motifs. Expressions régulières étendues. . . . 190
17.7.1 Reconnaissance de motifs . . . 190

17.7.2 Algorithme KMP . . . 191
17.7.3 Autres problèmes de reconnaissance . . . 191

18 Structures usuelles en Ocaml **193**
18.1 Introduction . . . 193
18.2 Piles . . . 193
18.2.1 Fonctions usuelles . . . 193
18.2.2 Fonctions additionnelles . . . 193
18.3 Files . . . 193
18.3.1 Fonctions usuelles . . . 193
18.3.2 Fonctions additionnelles . . . 194
18.4 Dictionnaires . . . 194
18.5 Files de priorité . . . 194
18.6 Créer un module en Ocaml . . . 196

Première partie

Programme de première année

Chapitre 0

Introduction au langage OCaml

Ce chapitre présente de manière succincte les bases de la programmation en Caml. On commence principalement par les aspects impératifs en s'appuyant sur ce qui est déja connu en Python. On y introduit donc la déclaration de variables, de fonctions, les types simples et les types impératifs que sont les tableaux et les chaînes de caractères. On poursuit par les instructions conditionnelles et boucles. Viennent ensuite des aspects plus fonctionnels : filtrages, types somme et enregistrement, et exceptions.

0.1 Caml : un langage fonctionnel (permettant la programmation impérative)

On distingue deux principaux styles de langages de programmation : les langages impératifs, et les langages fonctionnels.

Programmation impérative. La programmation impérative est un paradigme de programmation qui décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'*état* du programme. Un état représente l'ensemble des variables d'un programme. L'exécution d'un programme consiste, à partir d'un état initial, à exécuter une séquence finie de commandes d'affectation modifiant l'état courant. Les boucles (`for` et `while`) sont à disposition pour permettre la répétition d'instructions et les structures conditionnelles (`if`, `then`, `else`) pour permettre l'exécution conditionnelle d'instructions.

La quasi-totalité des processeurs qui équipent les ordinateurs sont de nature impérative : ils sont faits pour exécuter du code écrit sous forme d'*opcodes* (pour operation codes), qui sont des instructions élémentaires exécutables par le processeur. L'ensemble des opcodes disponibles forme le langage machine spécifique au processeur et à son architecture. L'état du programme à un instant donné est défini par le contenu de la mémoire centrale à cet instant, et le programme lui-même est écrit en style impératif en langage machine, ou le plus souvent dans une traduction lisible par les humains du langage machine, dénommée assembleur.

Les langages impératifs suivent cette nature, tout en permettant des opérations plus complexes (il n'y a pas de boucles en assembleur) : c'est pour cela qu'ils sont les plus répandus. Python est un langage de programmation impératif.

Programmation fonctionnelle. La programmation fonctionnelle est un paradigme de programmation qui considère le calcul comme l'évaluation de fonctions mathématiques. Ainsi, un programme est une fonction au sens mathématique, l'exécution d'un programme étant l'évaluation d'une fonction. Le programmeur écrivant un programme dans le paradigme fonctionnel va écrire des fonctions, simples à la base, puis de plus en plus complexes : une fonction déja écrite sert de « boite noire » dans une autre. Dans un langage fonctionnel, il n'y a pas de différence de nature entre une fonction et un objet simple (un entier ou un flottant, par exemple). Une fonction est une expression comme une autre et à ce titre pourra elle-même être l'argument ou le résultat d'une autre fonction. En programmation fonctionnelle, on ne dispose pas d'instructions permettant de modifier l'état du programme, il n'y a donc pas de boucles (qui changeraient l'état) : celles-ci sont remplacées par l'usage de la récursivité (capacité d'une fonction à s'appeler elle même).

En pratique, les langages fonctionnels sont écrits dans un langage de plus bas niveau (plus proche du langage machine), et masquent à l'utilisateur la nature impérative du processeur qui va exécuter le programme. OCaml est un langage fonctionnel, écrit en C (comme Python d'ailleurs) qui est un langage impératif de bas niveau.

Exemple : la factorielle. Les deux algorithmes suivants donnent des descriptions impérative et fonctionnelle de la fonction factorielle.

Algorithme 0.1 : Factorielle impérative

Entrée : Un entier $n \geq 0$
$f \leftarrow 1$;
pour *i allant de 1 à n faire* **faire**
  $f \leftarrow f \times i$
Renvoyer f

Algorithme 0.2 : Factorielle fonctionnelle (fact)

Entrée : Un entier $n \geq 0$
si $n = 0$ **alors**
  Renvoyer 1
sinon
  Renvoyer $n \times \text{fact}(n-1)$

En fait, Python est un langage impératif permettant l'usage de la récursivité, tandis que Caml est un langage fonctionnel permettant l'usage de la programmation impérative : le style naturel pour implémenter la factorielle est plutôt le premier pour Python et le deuxième pour Caml, mais il est possible de faire l'inverse.

0.2 Un langage fortement typé

Commençons par un calcul simple dans l'interpréteur :

```
# 4+1 ;;
- : int = 5
```

Le prompt (`#`) est l'invite de l'interpréteur, la ligne `4+1;;` a été écrite au clavier. La ligne `- : int = 5` est le résultat du calcul. Comme on le voit sur cet exemple :

— un calcul en Caml termine par deux points-virgules `;;`

— Caml procède (avant même l'évaluation) par une analyse de types : il sait avant même de calculer que le résultat de l'opération est un entier (`int`).

Caml est un langage *fortement typé* : toute valeur possède un type, les opérateurs et fonctions prennent en paramètre et renvoient des données d'un certain type. Le mélange des genres est interdit : on ne peut, comme en Python, définir une fonction (sans argument, par exemple) qui renvoie dans certains cas un entier, et dans d'autres un booléen, ou considérer des listes d'éléments inhomogènes. Caml permet cependant la généricité (polymorphisme) : par exemple la fonction `max` peut comparer deux entiers ou deux flottants tout comme les relations de comparaisons comme `<=` : une fonction de tri permettra de trier une liste d'entiers comme une liste de flottants [1].

0.3 Types simples

0.3.1 Type « rien »

Le type `unit` en Caml correspond au `None` en Python. Il n'y a qu'une constante de ce type-là, dont l'utilité apparaîtra plus tard. On peut la construire à l'aide de deux parenthèses :

```
# () ;;
- : unit = ()
```

0.3.2 Entier

Le type `int` correspond aux entiers. Sur une implémentation 64 bits, ceux-ci sont restreints à la plage de valeurs [2] $[\![-2^{62}, 2^{62}-1]\!]$. Les opérateurs sur les entiers sont `+`, `*`, `-`, `/` (division entière) et `mod` (modulo).

```
# 2*16 ;;
- : int = 32
# 58 mod 14 ;;
- : int = 2
```

1. Et si on veut faire plus générique, on passera en paramètre de la fonction de tri une fonction de comparaison.
2. Le lecteur attentif aura noté que cette plage contient 2^{63} entiers. En effet, Caml réserve 1 bit pour savoir que le registre contient bien un entier, et non une adresse mémoire. On n'en dira pas plus.

0.3.3 Flottants

Le type `float` correspond aux flottants, essentiellement ce sont les mêmes qu'en Python. Les opérations simples sont les mêmes que pour les entiers (sauf `mod` qui n'a pas d'équivalent), mais suivis d'un point qui les différencie de leurs équivalents sur les entiers. Les fonctions usuelles (`cos`, `sin`, `exp`, `atan`...) sont définies sur les flottants de même que l'opérateur d'exponentiation [3] (`**`).

```
# 2.0 *. 8.9 ;;
- : float = 17.8
# 2.0 ** 8.9 ;;
- : float = 477.712891666845508
# cos 5.0 ;;
- : float = 0.283662185463226246
```

Comme on l'a dit plus haut, le mélange des genres est interdit :

```
# 2 *. 3.0 ;;
Characters 0-1:
  2 *. 3.0 ;;
  ^
Error: This expression has type int but an expression was expected of type float
# 2 ** 5 ;;
Characters 0-1:
  2 ** 5 ;;
  ^
Error: This expression has type int but an expression was expected of type float
```

Dans le premier cas, l'opérateur `*` est défini sur les entiers, mais `3.0` est un flottant. Dans le deuxième, c'est l'inverse : `**` est défini sur les flottants.

0.3.4 Booléens

Semblables aux booléens de Python, le type `bool` de Caml n'a que deux constantes : `true` et `false` (sans majuscule !). Les opérateurs [4] sont `&&` (et logique), `||` (ou logique, la barre s'obtient avec Alt gr + 6) et `not` (non logique). En terme de priorité, `not` est prioritaire sur `&&` qui l'est sur `||`.

```
# true && false ;;
- : bool = false
# false || true ;;
- : bool = true
# not true ;;
- : bool = false
```

Comme en Python, les opérateurs `&&` et `||` sont paresseux : si la partie gauche suffit à déterminer le résultat de l'opération, la partie droite n'est pas évaluée.

```
# false && 1/0>0 ;;
- : bool = false
# true || 1/0>0 ;;
- : bool = true
```

Notez que ceci n'empêche pas Caml d'exiger naturellement la cohérence du type. `false && 1` n'a pas de sens et provoquera une erreur.

0.4 Déclaration, déclaration locale, déclaration simultanée, références

0.4.1 Déclaration

Pour déclarer une variable, on utilise `let`. La déclaration permet de déclarer une variable *globale* qui ne change pas à moins qu'une autre déclaration globale ne l'écrase. Parler de variable ici est donc abusif : on déclare plutôt une constante ! En effet, on effectue simplement une liaison entre un nom (le nom de la variable) et une valeur.

```
# let x = 44 ;;
val x : int = 44
```

3. Non, il n'y a pas d'opérateur d'exponentiation sur les entiers.

4. Variantes : `&` pour le et logique et `or` pour le ou. Comme `and` est utilisé pour la déclaration simultanée de variables (voir la suite), je vous conseille d'utiliser `&&` et `||` pour éviter toute confusion.

0.4.2 Déclaration locale

Le même mot clé `let`, combiné avec `in`, sert à effectuer une déclaration locale. Si la variable utilisée était déja associée à une valeur, celle-ci est temporairement oubliée, mais retrouvée à la fin de l'exécution. Voici un moyen de calculer $\sqrt[4]{5}+\sqrt[4]{5^2}+\sqrt[4]{5^3}$ à l'aide d'une déclaration locale :

```
# let x= 5.0 ** 0.25 in x +. x *. x +. x ** 3.0 ;;
- : float = 7.07511828360311945
# x ;;
- : int = 44
```

Si le nom de la variable n'était pas lié à une valeur avant la déclaration locale, il ne l'est toujours pas après :

```
# let y= exp(1.0) in (y +. 1.0 /. y) /. 2.0 ;;
- : float = 1.54308063481524371
# y ;;
Characters 0-1:
  y ;;
  ^
Error: Unbound value y
```

0.4.3 Déclaration simultanée

On peut déclarer simultanément deux variables avec le mot clé `and` :

```
# let x = 0 and y = 1 ;;
val x : int = 0
val y : int = 1
```

Bien sûr, cette déclaration peut également être locale.

```
# let x=0 and y=1 in x+y ;;
- : int = 1
```

Il ne faut pas confondre déclaration locale et simultanée. Si la variable `z` n'est pas définie avant l'exécution du code suivant, on obtient une erreur :

```
# let z=0 and y=z+2 in y+z ;;
Characters 14-15:
  let z=0 and y=z+2 in y+z ;;
                ^
Error: Unbound value z
```

0.4.4 Références

Les variables de Caml ne sont donc pas des variables au sens traditionnel des langages de programmation, puisqu'il est impossible de modifier leur valeur. Il est pourtant souvent nécessaire d'utiliser dans les programmes des variables modifiables comme en Python. En Caml, on utilise pour cela une référence modifiable vers une valeur, c'est-à-dire une case mémoire dans laquelle on peut lire et écrire le contenu. Pour créer une référence, on applique le constructeur `ref` au contenu initial de la case mémoire. Par exemple :

```
# let x=ref 0 ;;
val x : int ref = {contents = 0}
```

La variable `x` est liée à une valeur (de type `int ref`), qui est une référence pointant vers 0 à la création. Pour lire le contenu d'une référence, on utilise l'opérateur de *déférencement* `!`, qui signifie « contenu de » :

```
# !x ;;
- : int = 0
```

De même qu'avec une déclaration « globale » avec `let`, la liaison entre la variable et la case mémoire pointée est définitive jusqu'à ce qu'une nouvelle déclaration écrase cette liaison : plus précisément, la valeur de `x` est ici l'adresse de la case mémoire (modifiable) qui contient la valeur, et cette adresse est une constante.

Pour modifier le contenu d'une référence on utilise l'opérateur d'affectation `:=`. Par exemple, `x := !x + 1` incrémente le contenu de la case mémoire pointée par `x`, de manière similaire à `x+=1` ou `x=x+1` en Python.

```
# x:= !x + 1 ;;
- : unit = ()
# !x ;;
- : int = 1
```

Les références sont liées à la possibilité de faire de la programmation impérative en Caml : on les utilisera donc souvent dans des boucles. Le type de la valeur pointée par une référence est fixé à la création : on a vu précédemment que `x` était de type `int ref`. On peut de même créer des `bool ref`, `float ref`... et des références vers des types plus complexes. Une remarque pour finir : les opérations `x := !x + 1` et `x:= !x - 1` étant d'usage très courant, elles ont un raccourci :

```
# !x ;;
- : int = 1
# incr x ;
- : unit = ()
# !x
- : int = 2
# decr x ; !x ;;
- : int = 1
```

0.5 Quelques types plus complexes

0.5.1 Tuples

Les tuples (n-uplets en français) sont similaires aux tuples de Python : ils permettent de construire des séquences de valeurs de type possiblement différents.

```
# (true, 0, atan 1.0) ;;
- : bool * int * float = (true, 0, 0.785398163397448279)
```

Le type d'un tuple de la forme $(t_1, t_2, \ldots, t_p)$ est le *produit cartésien* des types de ses éléments, par exemple `bool * int * float` dans l'exemple ci-dessus. Comme on le voit sur la deuxième ligne, les parenthèses sont facultatives. En pratique, les tuples seront souvent utilisés comme valeur de retour de fonctions (voir la suite). On peut récupérer dans des variables les composantes d'un tuple :

```
#  let a, b = (0, 1.0) ;;
val a : int = 0
val b : float = 1.
```

Notons l'existence des fonctions `fst` et `snd` qui permettent de récupérer la première et la deuxième composante d'un couple (tuple de taille 2), et seulement d'un couple :

```
# fst (1, 5.0) ;;
- : int = 1
# snd (0, true) ;;
- : bool = true
```

0.5.2 Tableaux

Les tableaux (array en anglais) sont très similaires aux listes de Python [5] à deux restrictions près :

— la taille du tableau est fixée à la création, et ne peut pas changer : il n'y a donc pas d'équivalent aux fonctions `append` et `pop` pour des tableaux Caml ;

— le type des éléments du tableau est le même pour tous les éléments.

On aura donc des `int array`, `bool array`, `(int * bool) array`, etc... Pour la syntaxe, les éléments sont séparés par des point-virgules, et placés entre `[|` et `|]`.

```
# [|5; 0; 7|] ;;
- : int array = [|5; 0; 7|]
```

5. Les listes Python sont en fait des tableaux redimensionnables inhomogènes.

De même qu'en Python, si n est la taille du tableau, les éléments sont indexés de 0 à $n-1$. Pour l'accès et la modification des éléments, si `t` est un tableau de taille n et i un indice (entre 0 et $n-1$), on utilise `t.(i)` pour récupérer la valeur stockée à l'indice i, et `t.(i) <- x` pour la changer en `x`.

```
# let t = [|0; 5; 7|] ;;
val t : int array = [|0; 5; 7|]
# t.(0) <- 2 ;;
- : unit = ()
# t ;;
- : int array = [|2; 5; 7|]
```

Remarquez, que, naturellement, l'expression `t.(0) <- 2` a pour type `unit`, mais elle a un *effet de bord* : le modification de la première case du tableau. Pour parcourir des tableaux, on utilise fréquemment des références et des boucles : on fait donc de la programmation impérative.

On donne enfin (beaucoup) de fonctions sur les tableaux [6], qui se trouvent dans le module Array. Toutes ne sont pas à connaître, mais elles sont parfois bien pratiques. Retenez comment on construit un tableau par la données de ses éléments, comment on accède à ou modifie une entrée d'un tableau, `Array.make` et `Array.length`, on peut recoder assez facilement toutes les autres avec !

commande	effet
`[|0;1;7;8;9|]`	construit un tableau par donnée explicite des éléments.
`t.(i)`	le i-eme élément de `t` ($0 \leq i < n$ avec n la taille de `t`)
`t.(i) <- x`	remplacer le i-eme élément de t par x.
`Array.length t`	renvoie la longueur de t.
`Array.make n x`	construit un tableau de longueur n contenant des x (attention les éléments sont physiquement tous égaux !)
`Array.init n f`	construit un tableau de longueur n contenant les $f(i)$ pour i entre 0 et $n-1$
`Array.copy t`	renvoie une copie `t`
`Array.sub t i k`	renvoie un tableau de longueur k, égal à la portion de `t` qui démarre à l'indice i
`Array.append t1 t2`	concatène deux tableaux (ne pas confondre avec Python !)
`Array.concat q`	concatène une liste de tableaux (les listes seront vues ultérieurement.)
`Array.make_matrix n m x`	construit une matrice de taille n, m contenant des x (c'est-à-dire un tableau de tableaux. Un élément est accessible par `t.(i).(j)`).
`Array.map f t`	crée un tableau dont les éléments sont les $f(x)$ pour x dans t.
`Array.iter f t`	applique f sur chaque x de t (f est de type `'a -> unit`).
`Array.sort f t`	trie le tableau `t` en place, avec fonction de comparaison `f`. `f x y` renvoie un entier, nul si les éléments sont égaux, strictement positif si $x > y$, et strictement négatif sinon.

0.5.3 Chaînes de caractères

Contrairement à Python, Caml fait la distinction entre un caractère (type `char`) et une chaîne de caractères (type `string`). D'un point de vue syntaxique, les caractères sont encadrés par des apostrophes (touche 4 du clavier), et les chaînes par des guillemets (touche 3). D'un point de vue sémantique, les chaînes sont très semblables à des `char array`, à ceci près qu'elles sont immuables [7], comme en Python.

```
# let s="abc" ;;
val s : string = "abc"
# s.[0] ;;
- : char = 'a'
```

Comme on le voit, l'accès à l'élément d'indice i d'une chaîne `s` se fait avec `s.[i]`. Bien que semblables à des `char array`, les chaînes de caractères ont leur propre syntaxe et leurs propres fonctions associées, voici un récapitulatif de quelques fonctions [8].

6. Liste complète ici : `https://caml.inria.fr/pub/docs/manual-ocaml/libref/Array.html`.

7. Ce comportement est récent. Avant, `s.[i] <- 'a'` permettait de modifier le caractère numéro i par la lettre `'a'`. Aujourd'hui, Ocaml fait la distinction entre chaînes (immuables) et chaînes d'octets (type `Bytes`), modifiables. On verra à l'occasion les fonctions permettant de passer d'un type à l'autre.

8. Liste complète ici : `https://caml.inria.fr/pub/docs/manual-ocaml/libref/String.html`.

commande	effet
`'a'`	le caractère a, entre apostrophes
`"abc"`	la chaîne de caractère abc
`s.[i]`	le i-ème caractère de s
`String.length s`	longueur de la chaîne s
`String.make n c`	création de la chaîne $ccc\cdots c$ de longueur n
`String.sub s i k`	extrait la sous-chaîne de taille k commencant a l'indice i de s.
`s1^s2`	renvoie la concaténation des deux chaînes s_1 et s_2.

Pour terminer, un petit avertissement : la longueur d'une chaîne est directement liée au nombre d'octets utilisés pour représenter un caractère. Attention avec les caractères accentués (qui ne sont pas ASCII) !

```
# String.length "é" ;;
- : int = 2
```

0.6 Fonctions

En Caml, une fonction est une valeur, qui a donc un type. Commençons par regarder quelques fonctions déja existantes avant de créer les notres.

0.6.1 Quelques exemples

```
# int_of_float ;;
- : float -> int = <fun>
# int_of_float (3.5) ;;
- : int = 3
# fst ;;
- : 'a * 'b -> 'a = <fun>
# fst (true, 5.4) ;;
- : bool = true
```

Comme on le voit sur ces deux exemples, le type d'une fonction est de la forme type du paramètre → type du résultat. La fonction `int_of_float` permet d'obtenir un entier à partir d'un flottant (c'est donc la partie entière [9]). La fonction `fst` permet, comme on l'a déja évoqué, d'obtenir la première composante d'un couple. Comme les types des composantes de ce couple peuvent être quelconque, Caml utilise des types *génériques* (on dit aussi *polymorphes*) pour donner le type de la fonction : un couple générique est de type `'a * 'b`, ici le résultat est du type `'a`, car nécessairement du type de la première composante. La fonction est donc bien de type `'a * 'b -> 'a`.

0.6.2 Fonctions à un seul argument. Appel

Pour créer une fonction à un seul argument, on utilise `function`. Par exemple :

```
# function x -> x+1 ;;
- : int -> int = <fun>
```

On crée ici la fonction qui à un entier x associe $x+1$. Caml détecte tout seul que le type est `int -> int` car l'opérateur `+` n'est valable que sur les entiers : le paramètre de la fonction doit donc être un entier. On peut bien sûr associer cette fonction à une variable :

```
# let f = function x -> x+1 ;;
val f : int -> int = <fun>
```

Pour l'appel, on utilise comme en mathématiques la syntaxe `fonction (valeur)`. Les parenthèses sont en fait facultatives :

```
# f (5) ;;
- : int = 6
# f 5 ;;
- : int = 6
# (function x -> x +. 1.0) 2.5 ;;
- : float = 3.5
```

9. Sur les flottants positifs seulement, car `int_of_float (-4.3)` vaut `-4`.

En fait, il existe une autre syntaxe pour déclarer une fonction à un argument, et lui associer simultanément un nom. En mathématiques, on écrit souvent quelque chose comme « Posons $f(x) = x + 1$ ». C'est pareil en Caml, mais les parenthèses sont aussi facultatives. Cette syntaxe est celle que l'on utilisera le plus souvent.

```
# let f x = x + 1 ;;
val f : int -> int = <fun>
```

Il n'y a pas de différence avec la construction précédente de `f`, mais c'est plus pratique !

On déclarera une fonction à un seul argument avec la syntaxe `let f x = expression`

0.6.3 Curryfication des fonctions

En mathématiques, lorsqu'on considère une fonction ayant deux arguments, c'est qu'on considère une application de la forme :

$$\begin{array}{rccc} f: & E \times F & \to & G \\ & (x, y) & \longmapsto & f(x, y) \end{array}$$

On pourrait considérer de telles fonctions systématiquement, lorsqu'on veut écrire des fonctions à plusieurs arguments. C'est le cas de `fst` vue plus haut, par exemple. Mais en informatique, il est pratique de procéder autrement.

Dans la suite, on note $\mathcal{F}(E, F)$ l'ensemble des applications de E dans F. Notons qu'il y a une bijection naturelle entre $\mathcal{F}(E \times F, G)$ (applications de $E \times F$ dans G) et $\mathcal{F}(E, \mathcal{F}(F, G))$ (applications de E vers l'ensemble des applications de F dans G). Cette bijection est la suivante :

$$\begin{array}{rccc} \varphi: & \mathcal{F}(E \times F, G) & \to & \mathcal{F}(E, \mathcal{F}(F, G)) \\ & f & \longmapsto & x \mapsto (y \mapsto f(x, y)) \end{array}$$

L'intérêt est notamment qu'avec la seconde formulation, on peut considérer des applications *partielles*. Par exemple, avec la fonction max (à deux arguments), la deuxième formulation permet de construire facilement la fonction $y \mapsto \max(20, y)$ qui donne le maximum entre y et 20. La plupart des fonctions Caml sont données sous la seconde forme, appelée forme *curryfiée*[10]. Regardons quelques fonctions :

```
# max ;;
- : 'a -> 'a -> 'a = <fun>
# max 20 ;;
- : int -> int = <fun>
# max 20 58 ;;
- : int = 58
# Array.make ;; (* création d'un tableau de taille n initialisé avec l'élément x *)
- : int -> 'a -> 'a array = <fun>
# Array.make 5 0 ;;
- : int array = [|0; 0; 0; 0; 0|]
# Array.append ;; (* concaténation de tableaux *)
- : 'a array -> 'a array -> 'a array = <fun>
# Array.append [|2; 0|] ;;
- : int array -> int array = <fun>
```

Toutes ces fonctions sont curryfiées. Comme on le voit, pour une fonction curryfiée à deux arguments, `f a b` est équivalent à `(f a) b` : *la fonction a priorité dans l'évaluation.* Inversement, une fonction de type `'a -> 'b -> 'c` est en fait une fonction curryfiée de type `'a -> ('b -> 'c)`. Toute la discussion se généralise naturellement à des fonctions à plus de deux arguments.

0.6.4 Création de fonctions curryfiées

Recréons pour l'exemple la fonction somme sur les entiers. L'opérateur `+` de Caml est un opérateur *infixe* (c'est-à-dire qu'il s'utilise sous la forme a op b, mais on peut le transformer en opérateur préfixe (c'est-à-dire qu'il s'utilise sous la forme op a b, donc comme une fonction) en l'encadrant entre parenthèses :

```
# (+) ;;
- : int -> int -> int = <fun>
```

Un moyen d'obtenir une fonction équivalente est de faire un usage répété de `function` :

10. Du nom du mathématicien et logicien américain Haskell Curry, dont le prénom a donné le langage fonctionnel Haskell.

```
# function x -> (function y -> x+y) ;;
- : int -> int -> int = <fun>
```

Le mot clé `fun` permet de construire directement des fonctions curryfiées à plusieurs arguments :

```
# fun x y -> x+y ;;
- : int -> int -> int = <fun>
# (fun x y -> x+y) 1 6 ;;
- : int = 7
```

En général on préfère donner un nom à la fonction, et on peut utiliser `let` de la même manière qu'avec un seul argument :

```
# let somme x y = x + y ;;
val somme : int -> int -> int = <fun>
# somme 1 6 ;;
- : int = 7
```

Les constructeurs `fun` et `function` sont assez piégeux, en général on les évitera.

On déclarera une fonction curryfiée à n arguments avec la syntaxe `let f x1 x2 ... xn = expression`

0.6.5 Création de fonctions non curryfiées

Il sera toutefois utile de temps en temps de créer des fonctions non curryfiées. Par exemple, si on veut travailler avec les points du plan, on travaillera explicitement avec des couples de type `int * int` ou `float * float`. On peut déclarer une fonction non curryfiée avec `let` également :

```
# let f (x,y) = x**2. +. y**2. <= 1. ;;
val f : float * float -> bool = <fun>
```

Cette fonction teste si le couple de flottants passé en paramètre est dans le disque unité fermé. Une définition équivalente déconstruit le couple à l'intérieur du corps de la fonction :

```
# let f z = let x,y=z in x**2. +. y**2. <= 1. ;;
val f : float * float -> bool = <fun>
```

L'usage de `fst` et `snd` était également possible.

0.7 Conditions et boucles en Caml

0.7.1 Expressions conditionnelles

La syntaxe générale d'une expression conditionnelle en Caml est de la forme `if cond then a else b` où `cond` est une expression booléenne, et `a` et `b` sont des expressions *de même type*. Le type de l'expression (détecté avant exécution !) est ce type commun. Le résultat peut être utilisé dans d'autres expressions.

```
# if true then 1 else 0 ;;
- : int = 1
# 1+(if true then 1 else 0) ;;
- : int = 2
```

Si `a` et `b` sont deux expressions de types différentes, l'interpréteur avertit immédiatement d'une erreur.

```
# if true then 1.0 else 2 ;;
Characters 22-23:
  if true then 1.0 else 2 ;;
                      ^
Error: This expression has type int but an expression was expected of type float
```

Ici, l'interpréteur a détecté que la première expression (`1.0`) avait type `float`, la deuxième doit avoir ce type également. L'absence de `else` est compris comme `else ()`, où `()` est de type `unit`. L'expression `a` doit alors être de type `unit` :

```
# if true then print_string "c'est vrai !" ;;
c'est vrai !- : unit = ()
```

Cette possibilité sera par exemple utilisée lorsqu'on programmera de manière impérative : une expression modifiant la valeur pointée par une référence a pour type `unit` :

```
# let x = ref 0 ;;
val x : int ref = {contents = 0}
# if true then x:= !x + 1 ;;
- : unit = ()
```

0.7.2 Séquence d'instructions

En Caml, deux expressions successives sont séparées par un point virgule. Dans une suite d'expressions successives, *toutes doivent avoir le type* `unit`*, sauf peut-être la dernière.* Le type de l'expression totale est celui de la dernière expression.

```
# let x=ref 0 in x:= !x + 1 ; print_int 8 ; print_string " encore une expression  " ; !x ;;
8 encore une expression  - : int = 1
```

Lorsqu'on utilise la construction `if`, `then`, `else`, il ne doit y avoir qu'une seule expression dans les cas `then` et `else`. On peut délimiter une séquence d'instructions par `begin`, `end` si besoin. Par exemple :

```
# let x = ref 0 in if !x > 0 then x:= !x + 1 ; print_int !x  ;;
0- : unit = ()
# let x = ref 0 in if !x > 0 then begin x:= !x + 1 ; print_int !x end ;;
- : unit = ()
```

(Dans le premier cas, `print_int` est exécuté car en dehors du `if ... then`. Dans le deuxième on a encadré avec `begin` et `end`). Une petite remarque : on peut éviter l'usage de `begin` et `end` par parenthésage, c'est un peu plus court :

```
# let x = ref 0 in if !x > 0 then (x:= !x + 1 ; print_int !x) ;;
- : unit = ()
```

0.7.3 Boucles

Boucles for.

La syntaxe d'une boucle `for` est la suivante : `for i = i1 to i2 do instructions done`. Le compteur de boucle (`i` ici) prend toutes les valeurs entre `i1` et `i2`, par pas de 1. Si `i2` $<$ `i1`, aucune instruction n'est exécutée. Les instructions, séparées par des point-virgules, ont toutes type `unit`, qui est aussi le type de la boucle totale. Voici une fonction, écrite dans un style impératif, qui calcule la factorielle d'un entier.

```
let fact n=
  let y=ref 1 in
  for i=1 to n do
     y:= !y * i
  done ;
  !y
;;
```

Regardons l'exécution :

```
# fact 5 ;;
- : int = 120
```

Il n'est pas possible de modifier le compteur de boucle, et c'est très bien comme ça. Petite variante, la syntaxe `for i = i1 downto i2 do instructions done` permet d'avoir un pas de -1 (il faut donc que `i1` $\geq$ `i2` pour qu'au moins une intruction soit exécutée).

Boucles while.

La syntaxe est très similaire : `while condition do instructions done`, où `condition` est une expression booléenne. L'algorithme d'Euclide suivant, de type `int -> int -> int`, est écrit avec une boucle `while`.

```
let pgcd a b=
  let x=ref a and y=ref b in
  while !y > 0 do
    let r= !x mod !y in
    x:= !y ;
    y:= r
  done ;
  !x ;;
```

Notez l'utilisation d'une variable `r` locale au corps de boucle, tandis que les références sont locales à la fonction. Regardons l'exécution :

```
# pgcd 451 123 ;;
- : int = 41
```

0.8 Filtrages

0.8.1 Introduction

Le filtrage peut être vu comme une alternative aux `if`, `else`, qui peuvent s'enchaîner de manière disgracieuse, mais est en réalité beaucoup plus que ça ! D'une part, il peut être utilisé pour construire une fonction « par morceaux ». La fonction suivante

$$\begin{array}{rccl} \text{sgn} : & \mathbb{Z} & \longrightarrow & \mathbb{Z} \\ & x & \longmapsto & \begin{cases} 0 & \text{si } x = 0. \\ 1 & \text{si } x > 0. \\ -1 & \text{si } x < 0. \end{cases} \end{array}$$

peut être implémentée comme suit.

```
let sgn x=if x=0 then 0 else abs(x)/x ;;
```

Une autre manière est d'utiliser un filtrage, avec le mot-clé `function`.

```
let sgn = function
| 0 -> 0
| x -> abs(x) / x
;;
```

Ceci s'étend naturellement à des fonctions de plusieurs arguments, avec l'utilisation de `fun`. Néanmoins, l'utilisation de `fun` et de `function` étant assez trompeuse, on préfère filtrer avec un `match ... with` :

```
let sgn x=match x with
  | 0 -> 0
  | y -> abs(y)/y
;;
```

Plus généralement, un filtrage permet de gérer différents motifs possible d'une expression. Mais d'autres part, il permet d'accéder aux composantes d'un type construit (en particulier récursif), ce qui sera très utile lorsqu'on aura vu les listes et les arbres, par exemple.

0.8.2 Règles du filtrage sur motif

Lors d'un filtrage, les cas successifs sont examinés un à un, et le premier qui « colle » à l'expression examinée est réalisé, pas les autres. Voici par exemple une réécriture du « ou exclusif » en Caml, sur les booléens :

```
let xor a b=match (a,b) with
  | true, true -> false
  | false, true -> true
  | true, false -> true
  | false, false -> false
;;
```

Dans une instruction de la forme `if... else...`, les expressions dans chaque bloc `if` et `else` doivent avoir le même type. Il en va de même des expressions qui sont résultats d'un filtrage :

```
# match true with | true -> 0 | false -> "erreur" ;;
Characters 39-47:
  match true with | true -> 0 | false -> "erreur" ;;
                                         ^^^^^^^^
Error: This expression has type string but an expression was expected of type int
```

Un filtrage se doit d'être exhaustif : c'est-à-dire qu'il doit pouvoir filtrer toutes les valeurs possibles de l'expression en fonction de son type. Par exemple, dans la fonction suivante, qui simule le lancé d'un dé à 6 faces [11] et indique une action, le filtrage n'est pas exhaustif (du point de vue de l'interpréteur Caml, qui ne voit que les types) :

```
let action ()=match 1+Random.int 6 with
  | 1 -> "un pas à droite"
  | 2 -> "un pas à gauche"
  | 3 -> "demi tour"
  | 4 -> "grand écart"
  | 5 -> "saut perilleux"
  | 6 -> "vrille"
;;
```

En effet, la fonction est acceptée, mais le filtrage non exhaustif signalé :

```
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: 0
val action : unit -> string = <fun>
```

Avoir des filtrages non exhaustifs est disgracieux et doit être évité. Le motif « joker » `_` (underscore) permet de filtrer tous les motifs possibles (ou une partie d'un motif). Ainsi, la dernière ligne du filtrage ci-dessus peut avantageusement être remplacée par

```
  | _ -> "vrille"
```

De même que l'interpréteur indique si un filtrage est non exhaustif, il indique aussi si un cas de filtrage est inutile :

```
#match true with | _ -> 0 | false -> 1 ;;
Toplevel input:
>match true with | _ -> 0 | false -> 1 ;;
>                           ^^^^^
Warning 11: this match case is unused.
- : int = 0
```

Le filtrage d'une expression s'effectue par *motifs* et non par valeurs : la « forme » de l'expression située à gauche qui est comparée à la valeur filtrée. Tout cela sera plus clair lorsqu'on aura vu les types construits et les listes, mais un motif est essentiellement :

— une constante (`true`, `0`, `(0, "a")`, etc...) ;

— un identificateur ;

— le joker `_` ;

— une construction de motifs à l'aide de motifs plus simples.

On peut déja faire un exemple avec les couples. Le filtrage suivant réalise des actions différentes suivant que le couple filtré possède une composante nulle ou non :

```
match c with
  | (0,_) -> ...
  | (_,0) -> ...
  | _ ->
```

Lorsqu'un ou plusieurs identificateurs se trouvent dans le motif et que le filtrage réussit, une liaison locale est effectuée entre les identificateurs et les valeurs filtrées. Par exemple :

11. `Random.int n` fournit un entier aléatoire de $[\![0, n-1]\!]$. Le module `Random` fournit aussi `Random.float f` pour un flottant.

```
match c with
  | (0,y) -> y
  | (x,0) -> x
  | (x,y) -> -x-y
```

Notons que ce filtrage est bien exhaustif : `(x,y)` filtre tous les couples possibles. Dans un motif, ne peuvent figurer que des identificateurs distincts : par exemple filtrer `(x,x)` n'a aucun sens ! Le mot clé `when` permet de relâcher un peu la rigidité du filtrage sur motif : une fois la liaison effectuée, on peut réaliser une comparaison de valeurs. Voici une réécriture de la fonction [12] `xor` :

```
let xor a b=match (a,b) with
  | (x,y) when x=y -> false
  | _ -> true
;;
```

Voici une écriture d'une fonction `arg` donnant un argument d'un nombre complexe identifié à un couple de flottants, avec un filtrage (on rappelle que $\pi = 4\arctan(1)$) :

```
let arg (x,y)= match (x,y) with
  | (0., 0.) -> failwith "zero"
  | (0., y) when y>0. -> 2. *. atan 1.
  | (0., y) -> -. 2. *. atan 1.
  | (x,_) when x>0. -> atan (y/. x)
  | _ -> atan (y/. x) +. 4. *. atan 1.
;;
```

0.9 Types enregistrement et types somme

On va maintenant voir comment construire de nouveaux types à partir de types existants ou non.

0.9.1 Types enregistrement

Principe. Les types enregistrement (ou types produit) sont similaires à des n-uplets, donc à un produit cartésien de types, à quelques exceptions près :

— les composantes ont des noms (étiquettes ou champs) ;

— on peut déclarer une ou plusieurs composantes comme modifiable (mutable) ;

— il n'y a pas d'ordre dans les champs d'un enregistrement.

L'utilisation d'enregistrements, en particulier modifiables, se fait beaucoup en programmation *impérative*. Voici un exemple :

```
# type personne = {nom: string ; prenom: string ; age: int} ;;
type personne = { nom : string; prenom : string; age : int; }
# let a = {nom="Dupond" ; age=42 ; prenom="Jean"} ;;
val a : personne = {nom = "Dupond"; prenom = "Jean"; age = 42}
# a.nom ;;
- : string = "Dupond"
```

On a déclaré un type `personne`, dont les champs sont `nom`, `prenom` et `age`, associés à un type particulier. Attention, le nom d'un type et les noms des champs doivent être en minuscules. Pour créer un élément de type `personne`, il suffit de fixer les valeurs des champs. On accède au champ `c` d'un élément `a` d'un type produit avec `a.c`.

Les motifs de filtrage peuvent être des types produits [13], par exemple pour tester si une personne est majeure :

```
let est_majeur p=match p with
  | {prenom = _ ; nom = _ ; age = x} when x>=18 -> true
  | _ -> false
;;
```

En fait, il n'est pas nécessaire de préciser tous les enregistrements dans un filtrage. La fonction suivante est équivalente.

12. Celle ci n'a toutefois pas le même type, elle est polymorphe : `'a -> 'a -> bool`.
13. En pratique, on filtrera rarement des types produit.

```
let est_majeur p=match p with
  | {age = x} when x>=18 -> true
  | _ -> false
;;
```

Bien sûr, `p.age >= 18` était suffisant ici.

Champ modifiable. A priori, l'âge d'une personne peut changer. On aurait pu rendre le champ `age` mutable, il aurait fallu procéder ainsi :

```
# type personne = {nom: string ; prenom: string ; mutable age: int} ;;
type personne = {nom: string ; prenom: string ; mutable age: int;} ;;
# let a = {nom="Dupond" ; age=42 ; prenom="Jean"} ;;
val a : personne = {nom = "Dupond"; prenom = "Jean"; age = 42}
# a.age <- 43 ;;
- : unit = ()
# a;;
- : personne = {nom = "Dupond"; prenom = "Jean"; age = 43}
```

Comme on le voit, la modification du champ `c` de l'élément `a` en la valeur `x` se fait avec `a.c <- x`.

Types paramétrés. On peut faire usage de polymorphisme dans les types produits. Par exemple, définissons nous même un type similaire aux couples, mais en imposant des éléments homogènes :

```
# type 'a couple = {f: 'a ; s: 'a} ;;
type 'a couple = { f : 'a; s : 'a; }
# {f=5; s=2} ;;
- : int couple = {f = 5; s = 2}
# {f=true; s=false} ;;
- : bool couple = {f = true; s = false}
```

Bien sûr, on peut utiliser autant de types polymorphes que nécessaire :

```
# type ('a, 'b, 'c) truc = {un: 'a ; deux: 'b ; trois: ('a * 'c) array ; quatre: bool} ;;
type ('a, 'b, 'c) truc = {un : 'a; deux : 'b; trois : ('a * 'c) array; quatre : bool;}
```

Recréer manuellement les références. Il est intéressant de voir que l'on peut recréer « manuellement » un type semblable au type `ref` de Caml à l'aide d'un enregistrement, contenant un seul champ (naturellement mutable). Pour pouvoir créer des références vers des types quelconques, on fait naturellement usage de polymorphisme en créant un type paramétré.

```
type 'a reference_perso = {mutable contenu: 'a} ;;
let creer_ref x = {contenu = x} ;;
let acceder_ref r = r.contenu ;;
let modifier_ref r x = r.contenu <- x ;;
```

Voici les types obtenus après interprétation :

```
type 'a reference_perso = { mutable contenu : 'a; }
val creer_ref : 'a -> 'a reference_perso = <fun>
val acceder_ref : 'a reference_perso -> 'a = <fun>
val modifier_ref : 'a reference_perso -> 'a -> unit = <fun>
```

On remarque que ceux-ci sont très similaires aux opérations semblables avec les références de Caml (parenthéser un opérateur infixe permet de le transformer en opérateur préfixe, c'est-à-dire en une fonction) :

```
# ref ;;
- : 'a -> 'a ref = <fun>
# ( ! ) ;;
- : 'a ref -> 'a = <fun>
# ( := ) ;;
- : 'a ref -> 'a -> unit = <fun>
```

0.9.2 Types somme

Principe. Les types produit correspondait à des produits cartésiens, les types sommes correspondent à des unions *disjointes*. De même que l'on utilise des champs pour désigner les composantes d'un type produit, on utilise des *constructeurs* pour indiquer dans quelle partie de l'union disjointe on se situe. Les constructeurs peuvent être constants, ou d'un certain type. Voici d'abord un type constitué de constructeurs constants, qui redéfinissent les booléens :

```
# type booleen = Vrai | Faux ;;
type booleen = Vrai | Faux
# Vrai ;;
- : booleen = Vrai
```

Un tel type constitué uniquement de constructeurs constants est dit *énuméré*. Voici un type mélangeant entiers et flottants : on utilise deux constructeurs au nom explicite :

```
# type nombre = Ent of int | Flo of float ;;
type nombre = Ent of int | Flo of float
# Flo 4.5 ;;
- : nombre = Flo 4.5
```

On peut bien sûr mélanger constructeurs constants ou non :

```
# type carte_tarot = Excuse | Roi | Dame | Cavalier | Valet | Atout of int | Petite_carte of int ;;
type carte_tarot = Excuse | Roi | Dame | Cavalier | Valet | Atout of int | Petite_carte of int
# Atout 21 ;;
- : carte_tarot = Atout 21
```

(Oui, il manque la couleur. On pourrait facilement l'intégrer !)

Fonctions sur les types sommes. On fonctionne énormément par filtrage dans l'utilisation des types somme. Voici une fonction de négation sur nos booléens fraichement redéfinis :

```
let negation b=match b with
  | Vrai -> Faux
  | Faux -> Vrai
;;
```

Et une autre sur les nombres :

```
let valeur_absolue x=match x with
  | Ent a -> Ent (abs a)
  | Flo a -> Flo (abs_float a)
;;
```

Et enfin une qui donne la valeur d'une carte de tarot [14] :

```
let point_tarot c=match c with
  | Roi | Excuse -> 4.5
  | Dame -> 3.5
  | Cavalier -> 2.5
  | Valet -> 1.5
  | Atout x when x=1 || x=21 -> 4.5
  | _ -> 0.5
;;
```

Paramétrage. Comme pour les types produit, on peut paramétrer en faisant usage de polymorphisme. Voici comment faire quelque chose qui ressemble à $A \cup B$ avec A et B disjoints.

```
type ('a,'b) union = A of 'a | B of 'b ;;
```

On peut bien sûr l'utiliser pour manipuler deux copies d'un même ensemble : quelque chose comme $\mathbb{Z} \cup \mathbb{Z}'$, où $\mathbb{Z}'$ est une copie de $\mathbb{Z}$, serait représenté par un `(int * int) union`.

14. On remarque que `Roi` et `Excuse` ont été filtrés ensemble. Il est possible de procéder ainsi lorsque dans les motifs de fitrage les identificateurs sont les mêmes (et ont les mêmes types).

Type option : Some / None. Il est courant, en Python par exemple, de vouloir écrire une fonction qui renvoie par exemple tantôt un entier, tantôt rien. Par exemple, une fonction recherchant l'indice de la première apparition d'un élément dans un tableau si l'élément apparaît au moins une fois, ou rien du tout sinon. En Caml, ce n'est pas possible car cela ne respecte pas la cohérence des types. Toutefois, Caml propose le type somme préconstruit `'a option`, dont les constructeurs sont `Some` (paramétré génériquement), et `None` (constructeur constant). Par exemple :

```
# Some 4 ;;
- : int option = Some 4
# None ;;
- : 'a option = None
# Some ("a", 2) ;;
- : (string * int) option = Some ("a", 2)
```

Voici une implémentation de la fonction évoquée, de type `'a -> 'a array -> int option` renvoyant l'indice de la première apparition d'un élément d'un élément dans un tableau, ou `None` s'il n'en existe pas.

```
let indice x t =
  let i=ref 0 and res = ref None in
  while !res = None && !i < Array.length t do
    if t.( !i) = x then res := Some !i ;
    incr i
  done ;
  !res
;;
```

On pourrait créer le type option nous-même facilement comme suit :

```
type 'a option = None | Some of 'a ;;
```

0.10 Exceptions

Un moyen (pas explicitement au programme) pour programmer efficacement en évitant le lourd usage d'une référence vers un booléen est d'utiliser des exceptions. Voici quelques exemples produisant des exceptions.

```
# 1/0 ;;
Exception: Division_by_zero.
# [|5; 2|].(2) ;;
Exception: Invalid_argument "index out of bounds".
# failwith "echec" ;;
Exception: Failure "echec".
```

Il y a ici 3 exceptions différentes dans cet exemple : `Division_by_zero`, `Invalid_argument` et `Failure`. Les deux dernières prennent en paramètre une chaîne de caractères.

Les exceptions en Caml forment un type à part entière, qui est le type `exn` (abbréviation de exception, sans suprise) :

```
# Division_by_zero ;;
- : exn = Division_by_zero
# Failure "truc" ;;
- : exn = Failure "truc"
```

À priori, une exception dans un programme Caml est *levée* lorsqu'on tente de faire une opération interdite : diviser par zéro, accéder à un indice d'un tableau qui n'est pas défini (comme ci-dessus), etc... La levée d'une exception interrompt le déroulement du programme, et l'exception remonte de fonctions appelées en fonctions appelantes jusqu'au programme principal, sauf si elle est *rattrapée*. Pour rattraper une exception, il suffit d'encadrer un code pouvant produire une exception par `try ... with`, et de rattraper l'exception dans le `with`. Par exemple, la fonction suivante

```
let sgn x=
  try
    x/abs(x)
  with Division_by_zero -> 0
;;
```

est l'implémentation Caml (complètement artificielle) de la fonction :

$$\begin{array}{rccl} \text{sgn} : & \mathbb{Z} & \longrightarrow & \mathbb{Z} \\ & x & \longmapsto & \begin{cases} 0 & \text{si } x = 0. \\ 1 & \text{si } x > 0. \\ -1 & \text{si } x < 0. \end{cases} \end{array}$$

Le mécanisme de rattrapage des exceptions est celui d'un filtrage [15]. On a vu que certaines exceptions pouvaient prendre en paramètre une valeur d'un certain type, c'est le cas de `Failure` (exception produite par `failwith`), qui est un constructeur à un argument (de type `string`).

Reprenons la même fonction que précédemment, avec `failwith` :

```
let sgn x=
  try
    if x=0 then failwith "division par zero !" ;
    x/abs(x)
  with Failure s -> 0 (* filtrage: s est liée localement à la valeur associée à Failure *)
;;
```

Il est possible de lever nous même des exceptions (autres que `Failure` avec `failwith`), par l'utilisation de `raise` :

```
#raise ;;
- : exn -> 'a = <fun>
```

On peut donc, dans le code précédent, remplacer le `failwith ...` par `raise (Failure ... )` : c'est strictement équivalent. Refaisons le avec `Division_by_zero` qu'on lève « manuellement » :

```
let sgn x=
  try
    if x=0 then raise Division_by_zero ;
    x/abs(x)
  with Division_by_zero -> 0
;;
```

Enfin, il est possible de créer nous même des exceptions. Voici un code d'une fonction permettant de chercher si un élément se trouve dans un tableau, sans référence. On crée une exception « Trouve » qu'on pourra lever pour indiquer qu'on a trouvé l'élément.

```
exception Trouve ;;

let appartient t x=
  try
  for i=0 to Array.length t - 1 do
    if t.(i) = x then raise Trouve
  done ;
  false
  with Trouve -> true
;;
```

Bien sûr, on aurait pu lever n'importe quelle exception (comme `Division_by_zero`), mais c'est plus parlant ainsi. On peut également créer une exception prenant un paramètre d'un certain type. La fonction suivante renvoie l'indice d'un élément dans un tableau, et -1 si l'élément n'y est pas :

```
exception Indice of int ;;

let indice t x=
  try
    for i=0 to Array.length t - 1 do
      if t.(i) = x then raise (Indice i)
    done ;
    -1
  with Indice a -> a
;;
```

Elle utilise une exception « de type entier ».

15. qui n'a pas à être exhaustif.

Chapitre 1

Structure de données usuelles

1.1 Introduction

Un fil rouge du programme des deux années d'option informatique est l'utilisation de structures de données abstraites classiques dans des algorithmes. Plus basiquement, pour résoudre un problème donné, un informaticien se tournera naturellement vers une structure de données bien connue s'il peut en faire usage efficacement. Donnons deux exemples :

— lors de l'exploration d'un graphe, il faut stocker les sommets découverts mais dont les voisins n'ont pas encore été examinés. Le choix de la structure (pile ou file) mène à une exploration du graphe dite « en profondeur » ou « en largeur ». Pour un graphe dont les arêtes sont munies de poids positifs, la distance entre deux sommets est la somme des poids des arêtes sur un chemin entre les deux sommets (on prend le chemin minimisant cette somme). Une généralisation du parcours en largeur explorant les sommets depuis une origine fixée, par distance à l'origine croissante, se fait via l'algorithme de Dijkstra : une file de priorité est utilisée pour gérer les sommets en cours d'exploration.

— l'informaticien d'une entreprise souhaite référencer les clients, et pouvoir accéder rapidement aux informations concernant un client à l'aide de son identifiant unique, attribué à la création de la fiche client. Une structure de dictionnaire est toute indiquée pour stocker les informations sur les clients.

Définition 1.1. *Une structure de données abstraite est la donnée d'un type, et des opérations que l'on peut effectuer dessus.*

Cette définition est un peu évasive, mais ce qu'il faut retenir est qu'une structure abstraite est indépendante d'une implémentation concrète : ce qui est important est la description des opérations que le type permet, plutôt que la manière dont ces opérations sont réalisées. À cela il y a plusieurs avantages :

— si on possède déja une implémentation d'une structure de données abstraites, on peut programmer des algorithmes en faisant usage sans savoir comment ces structures sont implémentées : on les voit comme des « boites noires » complexes. C'est le point de vue du programmeur utilisateur : il n'a pas besoin de savoir comment ça marche pour utiliser une structure abstraite.

— si on change l'implémentation d'une structure pour une autre implémentation, les algorithmes faisant usage de la structure que l'on a déja implémentés seront compatibles avec la nouvelle implémentation.

Ce qui permet de juger si une implémentation est meilleure qu'une autre est essentiellement la complexité : lorsqu'on voudra implémenter concrètement une structure abstraite (on parle de structure concrète), on cherchera à avoir la meilleure complexité (en temps et/ou en mémoire) possible.

Passons maintenant à la description des quatre structures de données abstraites au programme.

1.2 Piles

Une pile est une structure de donnée linéaire, dans laquelle les éléments sont insérés ou supprimés suivant le principe *LIFO* (last in, first out) : visuellement, le seul élément accessible est celui situé au sommet de la pile, qu'il faudrait retirer pour accéder à l'élément situé en dessous. Inversement, un élément qu'on rajoute à la pile le sera au sommet.

Définition 1.2. *Une pile est une structure abstraite, supportant les opérations suivantes :*

— *création d'une pile vide ;*

— *test d'égalité au vide ;*
— *accès au sommet d'une pile non vide ;*
— *retrait de l'élément au sommet d'une pile non vide ;*
— *ajout d'un élément au sommet de la pile.*

La pile est un outil de base en informatique, dont les utilisations sont multiples. Voici quelques exemples :
— la gestion des appels de fonctions dans l'exécution d'un programme se fait via une *pile d'appels* : lorsqu'une fonction est appelée, les informations relatives à l'appel (adresse de retour, notamment) sont empilées sur la pile d'appels, et seront dépilées lorsque la fonction terminera son exécution ;
— la gestion des boutons « page précédente » et « page suivante » d'un navigateur utilise deux piles ;
— le parcours en profondeur d'un graphe, déja évoqué, fait usage d'une pile.

Il existe une structure déja implémentée en Ocaml : le module `Stack`.

1.3 Files

La file est une autre structure linéaire, assez semblable à la pile, mais qui fonctionne sur le principe *FIFO* (first in, first out) : lorsqu'on insère un élément dans une file, celui-ci ne pourra être retiré qu'après que tous les éléments insérés avant l'aient été.

Définition 1.3. *Une file est une structure abstraite, supportant les opérations suivantes :*
— *création d'une file vide ;*
— *test d'égalité au vide ;*
— *retrait de l'élément en tête d'une file non vide ;*
— *ajout d'un élément en queue de file.*

La file n'est pas d'un usage aussi courant que la pile en informatique, citons néanmoins quelques exemples :
— une imprimante « bas de gamme » (qui ne gère pas les priorités, voir le paragraphe suivant) à qui l'on envoie des documents à imprimer les traitera séquentiellement : le premier document à être envoyé sera imprimé en premier ;
— le parcours en largeur d'un graphe traite les sommets par distance à l'origine (nombre minimal d'arêtes à parcourir depuis l'origine) croissante : il suffit de rajouter les sommets dans une file lorsqu'ils sont découverts pour respecter cet ordre dans le parcours.

Il existe une structure déja implémentée en Ocaml : le module `Queue`.

1.4 Files de priorité

Une file de priorité suit le même principe qu'une file, mais à chaque élément inséré dans la file est attaché une priorité, en général représentée par un entier. La file de priorité stocke donc des couples (e, p), où e est un élément et p un entier (la priorité). Le prochain élément à sortir de la file est celui qui a la plus grande priorité (on parle de file de priorité max ou min suivant si on prend le plus grand ou le plus petit entier p).

Définition 1.4. *Une file de priorité est une structure abstraite, supportant les opérations suivantes :*
— *création d'une file de priorité vide ;*
— *test d'égalité au vide ;*
— *retrait de l'élément de plus grande priorité d'une file non vide ;*
— *ajout d'un élément avec une priorité donnée ;*
— *modification de la priorité d'un élément (cette opération a un sens seulement si on impose l'unicité des éléments e stockés dans la file de priorité, ce qui est le cas en général).*

Quelques exemples d'utilisation :
— le système d'exploitation gère les processus à exécuter via des priorités. Une tâche de moindre importance sera mise en attente pour être exécutée par le processeur après celles de plus grande priorité ;
— il en va de même pour les documents gérés par une imprimante « haut de gamme » : il est possible d'augmenter ou de diminuer la priorité d'un document ;
— le parcours en largeur d'un graphe se généralise à des graphes pondérés (les arêtes sont munies d'un poids supposé positif) via l'algorithme de Dijkstra : les sommets à explorer en priorité sont ceux qui ont la plus petite distance à l'origine, l'ordre d'exploration est donc géré par une file de priorité min. Cet algorithme permet de calculer toutes les distances entre l'origine du parcours et chacun des sommets.

1.5 Dictionnaires

Le dictionnaire est également une structure très courante en informatique. Il stocke des couples (k, e) où k est la clé, et e l'élément associé. Dans un dictionnaire, les clés k sont supposées toutes distinctes.

Définition 1.5. *Un dictionnaire est une structure abstraite, supportant les opérations suivantes :*

- *création d'un dictionnaire vide ;*
- *test d'égalité au vide ;*
- *test de présence d'un élément ayant une clé k donnée ;*
- *retrait de l'élément ayant une clé k donnée (s'il existe) ;*
- *ajout d'un élément e avec une clé k donnée (s'il n'y a pas déja d'élément de clé k).*

Quelques exemples d'utilisation :

- un dictionnaire (au sens usuel) peut être vu comme un dictionnaire informatique : les clés sont les mots de la langue, les éléments sont les définitions ;
- un logiciel de programmation fait usage d'un dictionnaire pour stocker les variables définies par l'utilisateur, et leurs valeurs ;
- dans la réalistion concrète d'une file de priorité, on peut faire usage d'un dictionnaire pour implémenter l'opération « augmenter ou diminuer la priorité d'un élément ». Les clés de ce dictionnaire sont les éléments x de la file de priorité ;
- plus généralement, tout stockage de données où on attribut un identifiant unique (la clé) aux éléments stockés peut être implémenté via un dictionnaire.

Il existe une structure déja implémentée en Ocaml, qui fait usage de tables de hachage (module `Hashtbl`).

Chapitre 2

Programmation impérative en Caml

En informatique, la programmation impérative est un paradigme de programmation qui décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme. Bien que ce ne soit pas la manière de coder la plus utilisée en Caml, qui dérive du langage fonctionnel ML, c'est tout à fait possible. Un objet typique de la programmation impérative est le tableau, est une fonction de tri qui ne retourne rien (le type `unit` en Caml), mais modifie le tableau. Dans ce chapitre, on va donc voir (ou revoir) les boucles, et la manipulation de tableaux. On en profite pour revoir la notion de terminaison des fonctions itératives, les preuves de correction et l'étude de leur complexité. Une application de tout ceci se trouve naturellement dans les tris : on se limite ici aux tris de base qui ne sont pas récursifs.

2.1 Analyse d'algorithmes : terminaison, correction, complexité

Pour décrire un algorithme, on lui donne en général un nom, on précise quels sont les paramètres qu'il peut prendre en entrée et le résultat qu'il est sensé renvoyer. On précise aussi de quelle manière il agit sur son *environnement* : modification de la mémoire, affichage éventuel à l'écran, etc... Tout ceci constitue la *spécification de l'algorithme.*

Dans nos algorithmes, outre les opérations d'affectations, et de manipulations des variables, on trouve un découpage en blocs simples correspondant aux instructions conditionnelles et aux boucles. Ce découpage en blocs simples est essentiel : pour montrer que notre algorithme se termine, qu'il calcule ce qu'il est sensé calculer et estimer son coût, il suffit d'analyser chacun des blocs.

2.1.1 Terminaison

Pour montrer qu'un algorithme termine quel que soit le jeu de paramètres passé en entrée respectant la spécification, il faut montrer que chaque bloc élémentaire décrit ci-dessus termine ! Or, les boucles `for` et les instructions conditionnelles terminent forcément. Le seul souci pourrait venir d'une boucle `while`.

En général, pour montrer la terminaison d'une boucle `while` on procède ainsi : *on exhibe une quantité, dépendant des paramètres, à valeurs dans* $\mathbb{N}$*, qui décroît strictement à chaque passage dans la boucle.* Puisqu'il n'existe pas de suite infinie strictement décroissante dans $\mathbb{N}$, cela prouve que la boucle se termine. Cette quantité est appelée un *variant de boucle.*

2.1.2 Correction

Pour montrer qu'un algorithme est correct, il s'agit de montrer que quels que soient les paramètres vérifiant sa spécification, l'action de l'algorithme correspond à ce qui est attendu. Reprenons notre découpage en blocs. Pour montrer la correction de l'algorithme, il s'agit de montrer que chacun des blocs effectue une action bien précise. Pour les blocs conditionnels (`if`, `else`...), il n'y a en général pas grand chose à dire de plus que l'algorithme lui-même. En revanche, analyser les boucles `for` et `while` est essentiel, car l'action de ces boucles n'est pas forcément évidente en première lecture. La notion essentielle pour montrer la correction des boucles est celle d'*invariant de boucle.*

Définition 2.1. *Un invariant de boucle est une propriété dépendant des variables de l'algorithme, qui est vérifiée à chaque passage dans la boucle.*

Pour une formulation rigoureuse, on (je) préfère distinguer le cas des boucles `while` et celui des boucles `for`. Dans les deux cas le principe est le même : une propriété dépendant des paramètres est un invariant de boucle si les deux conditions suivantes sont vérifiées :

— la propriété est vérifiée avant la boucle ;
— si la propriété est vérifiée en haut du corps de boucle, alors elle est vérifiée en bas du corps de boucle.

On en conclut (notamment) que la propriété est vérifiée *après* la boucle. La démonstration qu'une propriété est bien un invariant de boucle est très similaire à une démonstration par récurrence.

Boucles `while`. Pour la boucle `while`, le principe est exactement celui décrit ci-dessus : on exhibe une propriété vraie avant la boucle, et qui, si elle est vérifiée en haut du corps de boucle, le sera en bas. On en conclut qu'elle est vérifiée après.

```
(* Inv *)
while condition do
    (* Inv *)
    instructions
    (* Inv *)
done ;
(* Inv *)
```

Boucles `for`. Pour la boucle `for`, l'invariant dépend en général du compteur de boucle. Pour celui-ci, le passage $i \leftarrow i+1$ (ou $i-1$) est implicite car exécuté par la boucle elle-même. En fait, une boucle `for` est très semblable à une boucle `while` particulière :

```
for i=id to if do
    instructions
done ;
```

```
let i = ref id ;
while !i <= if do
     instructions ;
     incr i
done ;
```

La séquence `instructions` dans les deux boucles est la même (en particulier, elle ne modifie pas `i`). Ainsi, sur la boucle `while`, si on fait dépendre une propriété Inv de l'indice `i` (on identifie pour simplifier la référence et la valeur pointée...), on s'aperçoit que, pour qu'elle soit un invariant, il faut que :

— $\text{Inv}(i_d)$ soit vérifié avant la boucle ;
— pour tout i entre i_d et i_f, si $\text{Inv}(i)$ est vrai en haut du corps de boucle alors $\text{Inv}(i+1)$ est vrai juste avant l'incrémentation de i.

On en déduit que $\text{Inv}(i_f+1)$ est vérifié après la boucle. C'est donc ainsi qu'on prouvera qu'une propriété est un invariant d'une boucle `for`. Le principe est le même pour les boucles où l'indice est décrémenté à chaque passage : pour une boucle de la forme `for i=id downto if`, on montre que $\text{Inv}(i_d)$ est vrai avant la boucle, et que pour tout $i \in \{i_f, \dots, i_d\}$, si $\text{Inv}(i)$ est vrai en haut du corps de boucle, alors $\text{Inv}(i+1)$ est vrai en bas du corps de boucle. On en déduit que $\text{Inv}(i_f+1)$ est vrai *après* la boucle.

2.1.3 Complexité

Qu'est-ce que la complexité ? L'étude de la complexité d'une fonction consiste à estimer son coût (temporel ou en mémoire), en fonction des entrées. Pour différencier deux entrées entre elles, on compare en général leur taille. Essentiellement pour nous, les entrées seront constituées d'entiers ou de tableaux. Pour les tableaux, la donnée pertinente est la taille. Pour les entiers, cela dépend du contexte. Pour un entier n, on peut en effet exprimer la complexité d'une fonction dépendant de n en fonction :

— de l'entier n lui-même.
— ou de son nombre de chiffres (sa taille), correspondant à $\ln(n)$, à une constante multiplicative près. En général on ne tient pas compte des constantes multiplicatives.

Par exemple, pour une fonction calculant $n!$, le nombre d'opérations est clairement linéaire en n. Pour exprimer la complexité d'une fonction qui renvoie l'écriture en base 2 d'un nombre exprimé en base 10, on se dirigerait plus naturellement vers $\log_2(n)$.

Coûts. Concentrons-nous d'abord sur la complexité en temps. L'exécution d'un algorithme est une séquence d'opérations nécessitant plus ou moins de temps. Pour mesurer ce temps, on considère certaines opérations comme élémentaires : par exemple faire une opération arithmétique de base (addition, multiplication, soustraction, division...), lire ou modifier un élément d'un tableau, ajouter un élément à la fin d'un tableau, affecter un entier ou un flottant, etc... Estimer le coût d'une fonction sur une entrée de taille donnée signifie estimer le nombre de ces opérations élémentaires effectuées par la fonction sur l'entrée. La complexité en mémoire consiste à estimer la mémoire nécessaire à une fonction pour son exécution.

Différentes complexités. Les notions de complexité au programme sont les complexités temporelles dans le meilleur et dans le pire cas, c'est à dire le nombre minimal/maximal que requiert l'algorithme pour s'exécuter sur une entrée de taille n. La complexité dans le meilleur des cas n'est en général pas la plus pertinente. Par exemple pour la fonction de recherche d'un élément `x` dans un tableau `t`, si l'élément cherché se trouve en première position, la fonction effectue (si elle est codée de la bonne façon) un seul test d'égalité de la forme `t.(i)=x`. La complexité (en nombre de tests d'égalité) est donc constante. La complexité au pire est le nombre d'opérations maximales nécessaires pour traiter une entrée de taille donnée. Pour une fonction de recherche de `x` dans un tableau, le cas le pire se produit par exemple lorsque le tableau ne contient pas `x`. Une autre notion de complexité intéressante est la complexité en moyenne, qu'on abordera assez peu : elle demande des connaissances en probabilité, et une distribution de probabilités sur les entrées possibles. En conclusion, lorsqu'on demandera d'exprimer la complexité d'une fonction sans plus de précision, on sous-entendra la complexité dans le pire cas.

Complexité asymptotique et notations de Landau. Tout d'abord, lorsque l'on s'intéresse à la complexité $C(n)$ (n est la taille de l'entrée) d'une fonction, c'est bien souvent pour les grandes valeurs de n qu'il est pertinent de connaître $C(n)$, pour comparer vis à vis d'autres fonctions réalisant le même calcul. On cherche donc un comportement asymptotique de n, qu'on rapportera aux fonctions usuelles : logarithmes, puissances, exponentielles... Ensuite, on ne cherchera pas systématiquement un équivalent : celui-ci est souvent difficile à obtenir et n'est pas le plus important. Si deux fonctions nécessitent respectivement $9n\log(n)$ et $\frac{n^2}{2}$ opérations élémentaires, on retiendra que la première nécessite de l'ordre de $n\log(n)$ opérations, ce qui est bien meilleur que la seconde qui en requiert de l'ordre de n^2. Rappelons les notations de Landau. Soit f et g deux fonctions $\mathbb{N}\to\mathbb{R}$. On note :

- $f(n)=O(g(n))$, si il existe un entier n_0 tel que $g(n)$ est non nul pour $n\geq n_0$ et $\left(\frac{f(n)}{g(n)}\right)_{n\geq n_0}$ est bornée.
- $f(n)=\Omega(g(n))$, si $g(n)=O(f(n))$.
- $f(n)=\Theta(g(n))$, si $f(n)=O(g(n))$ et $f(n)=\Omega(g(n))$.

Pour exprimer la complexité $C(n)$, on se contentera bien souvent d'un O. Par exemple, la recherche dichotomique dans un tableau trié de taille n a une complexité de $O(\ln(n))$. Si l'on veut préciser que cette borne est essentiellement optimale, on dira que la complexité est en $\Theta(\ln(n))$. Attention à ne pas dire « la complexité est au moins en $O(\ln(n))$ », ce qui n'aurait aucun sens.

Complexité arthmétique versus complexité binaire. On voit facilement que le calcul de x^n avec l'algorithme naïf nécessite $O(n)$ multiplications, et $O(\log(n))$ avc l'algorithme d'exponentiation rapide (voir la suite). Ainsi, la complexité (au pire comme au mieux, puisqu'il n'y a qu'une seule entrée de « taille » n ici) est en $O(n)$ ou $O(\log(n))$. Ce raisonnement est exact, en n'oubliant pas que l'on parle ici de complexité arithmétique. En pratique, les entiers intervenant dans le calcul grandissent vite : tenir compte de cette croissance donnerait une complexité dite binaire. On s'intéressera uniquement à la complexité arithmétique, mais il faut avoir à l'esprit que cela ne reflète pas forcément fidèlement les temps de calculs effectifs.

2.1.4 Un exemple important : l'exponentiation

Un exemple important est le calcul de x^n pour un entier naturel n donné. Il n'existe pas de fonction permettant ce calcul sur les entiers en Caml. On présente ici deux fonctions permettant ce calcul ; l'une est naïve, l'autre est plus élaborée.

Algorithme naïf. Il consiste simplement à multiplier n fois une variable par x, en partant de 1.

Algorithme d'exponentiation naïf

```
let expo x n=
  let y=ref 1 in
  (* Inv(0): y=x^0 *)
  for i=0 to n-1 do
    (* Inv(i): y=x^i *)
    y:=!y*x
    (* Inv(i+1): y=x^(i+1) *)
  done ;
  (* Inv(n): y=x^n *)
  !y
;;
```

La terminaison est évidente : l'algorithme repose sur une boucle `for`. Pour la correction, il est simple d'exhiber un invariant : la propriété $y=x^i$ est conservée à chaque passage de boucle. Plus exactement, la propriété au rang 0

est vérifiée avant la boucle, et on montre facilement que si elle est vérifiée au rang i en haut du corps de boucle, elle est vérifiée au rang $i+1$ en bas du corps de boucle. On en déduit que $y = x^n$ après la boucle, et on renvoie y donc l'algorithme est correct. Notez que là encore, dans la preuve de complexité, on a identifié référence et valeur pointée, ce qui est quand même plus pratique.

En terme de complexité arithmétique, celle-ci est clairement de n multiplications.

Algorithme d'exponentiation rapide. L'algorithme d'exponentiation rapide (itératif) se base sur ldécomposition binaire de l'entier n. Prenons un exemple : on souhaite calculer x^{11}. 11 s'écrit en binaire $\overline{1011}^2$. À partir de x et en procédant par élévations au carré successives, il est facile de calculer les x^{2^p} : ici ce sont x, x^2, x^4 et x^8. Comme $11 = \overline{1011}^2$, il suffit de multilplier x^8, x^2 et x pour obtenir x^{11}.

Expliquons son fonctionnement : il suit de près l'algorithme permettant de récupérer les bits d'un entier par division successives par 2. Cet algorithme permet de récupérer les bits 1 par 1, en commençant par les bits de poids faibles. En utilisant une variable annexe que l'on élève au carré à chaque étape, on calcule successivement les x^{2^p}. Il suffit alors de multiplier une variable (`z` dans le code suivant) initialisée à 1 par les x^{2^p} qui conviennent (donnés par les bits de n) pour obtenir x^n. Voici le code Caml :

Algorithme d'exponentiation rapide

```
let expo_rapide x n=
  let m=ref n and y=ref x and z=ref 1 in
  (* Inv: z*y^m = x^n *)
  while !m > 0 do
    (* Inv *)
    if !m mod 2=1 then z:=!z * !y ;
    m:=!m/2 ;
    y:=!y * !y
    (* Inv *)
  done ;
  (* Inv *)
  !z
;;
```

La terminaison de l'algorithme vient du fait que les valeurs prises par m (on identifie encore référence et valeur) forment une suite positive strictement décroissante. L'invariant de boucle `while` est le suivant : à chaque passsage dans la boucle, $zy^m = x^n$. En effet, cette propriété est vérifiée avant la boucle, et pour voir qu'elle est conservée lors d'un passage dans la boucle, il suffit de distinguer suivant la parité de m. À la fin de l'algorithme, puisque m est nul (on est sorti de la boucle), alors $z = x^n$, donc l'algorithme renvoie bien x^n. En terme de complexité, on fait un nombre borné de multiplications (1 ou 2, en fait) à chaque passage de boucle, et il est facile de voir que le nombre de passages dans la boucle est égal au nombre de chiffres dans l'écriture en binaire de n (0 si n est nul), c'est à dire $O(\log(n))$, qui est la complexité de l'algorithme en nombre d'opérations arithmétiques.

2.2 Les tableaux en Caml

Les tableaux ont été décrits dans le chapitre 0. Voyons quelques fonctions sur ce type.

2.2.1 Quelques exemples de fonctions basiques

Voici quelques fonctions de base qu'il faut savoir recoder vite : tester l'appartenance d'un élément à un tableau, calculer le minimum d'un tableau, ou le nombre d'occurences d'un élément dans un tableau (rappel : `incr n` est équivalent à `n:= !n + 1`).

```
let mini t=
  let m=ref t.(0) in
  for i=1 to Array.length t - 1 do
    m:= min !m t.(i)
  done ;
  !m
;;
```

```
let appartient t x=
  let b=ref false in
  for i = 0 to Array.length t - 1 do
    b:= !b || t.(i) = x
```

```
  done ;
  !b
;;
```

```
let occurences t x=
  let c=ref 0 in
  for i=0 to Array.length t -1 do
    c:= !c + (if t.(i) = x then 1 else 0)
  done ;
  !c
;;
```

Remarquez le caractère paresseux de l'opérateur `||` dans la fonction `appartient` : on n'évalue pas `t.(i)=x` si `!b` est `true`. Pour éviter de parcourir toute le tableau pour vérifier si au moins un élément du tableau vérifie une certaine propriété, on peut utiliser une référence vers un booléen et une boucle `while` :

```
let appartient t x=
  let b=ref false and i=ref 0 in
  while not !b && !i< Array.length t do
    b:=t.( !i) = x
  done ;
  !b
;;
```

On peut aussi utiliser une exception, comme vu dans l'introduction :

```
exception Trouve ;;
let appartient t x =
  try
    for i=0 to Array.length t - 1 do
      if t.(i) = x then raise Trouve
    done ;
    false
  with Trouve -> true
;;
```

2.2.2 Recherche dichotomique dans un tableau trié

Si le tableau est trié dans l'ordre croissant, on peut accélérer considérablement la recherche d'un élément : si l'on sait que l'élément à chercher ne peut se trouver qu'entre les indices g inclus et d exclus, on peut discriminer la moitié de la portion en considérant l'élément d'indice $m = \left\lfloor \frac{g+d}{2} \right\rfloor$:

- si l'élément d'indice m est l'élément cherché, on a terminé ;
- si l'élément d'indice m est strictement inférieur à l'élément cherché, celui-ci ne peut se trouver que dans la portion entre l'indice $m+1$ inclus et d exclus ;
- si l'élément d'indice m est strictement supérieur à l'élément cherché, celui-ci ne peut se trouver que dans la portion entre l'indice g inclus et $m-1$ exclus ;

Initialement, on a $g, d = 0, n$ avec n la taille du tableau. On peut s'arrêter lorsque $g = d$: l'élément ne se trouve pas dans le tableau.

```
exception Trouve ;;
let recherche_dicho t x=
  try
    let g, d=ref 0, ref (Array.length t) in
    while !g < !d do
      let m=( !g + !d) / 2 in
      if t.(m) = x then raise Trouve ;
      if t.(m) > x then d:= m else g:= m+1
    done ;
    false
  with Trouve -> true
;;
```

La portion $d - g$ sur laquelle on travaille a initialement une taille $t_0 = n$ (la taille du tableau), et diminue au moins de moitié à chaque itération : $t_{n+1} \leq t_n/2$. Ceci montre que seulement $O(\log n)$ étapes sont nécessaires pour que l'algorithme s'arrête.

2.2.3 Exemples : les algorithmes de tri

On se donne un tableau dont les éléments sont comparables : entiers, flottants, voire même chaînes de caractères pour l'ordre *lexicographique* (celui du dictionnaire...), et on veut le trier dans l'ordre croissant. On détaille ici les algorithmes de tris « naifs » les plus classiques. Ceux-ci sont quadratiques (complexité $O(n^2)$ avec n la taille du tableau) et sont donc inefficaces pour de grands tableaux. On leur préférera l'un des tris récursifs (qu'on verra plus tard) dès que le nombre d'éléments à trier dépasse environ 50.

Tri par sélection

Ce tri particulièrement simple est peut-être celui auquel on pense en premier lorsqu'on écrit un algorithme de tri.

Idée du tri. L'idée est simple : supposons qu'un tableau de taille n est déja en partie trié avec ses i premiers éléments à leur place définitive. On *sélectionne* le plus petit des $n-i$ éléments restants, qu'on amène en position $i+1$. Le tableau a alors ses $i+1$ premiers éléments à leur position définitive. Itérer ce procédé $n-1$ fois suffit pour trier le tableau. La fonction suivante nous sera utile :

échange de deux éléments d'un tableau

```
let echange t i j=
  let a=t.(i) in
  t.(i) <- t.(j) ;
  t.(j) <- a
;;
```

Naturellement, echange possède le type : `'a array -> int -> int -> unit`.

Code Caml.

Le tri par sélection

```
let tri_selection t =
  let n=Array.length t in
  for i=0 to n-2 do
    (* Inv(i): t.(0),...,t.(i-1) dans l'ordre croissant est plus petits que les autres éléments de t *)
    let imin=ref i in
    for j=i+1 to n-1 do
      (* Inv2(j): i_min est l'indice du plus petit élément parmi t.(i),...,t.(j-1) *)
      if t.(j) < t.( !imin) then imin:= j
      (* Inv2(j+1) *)
    done ;
    if !imin>i then echange t i !imin
    (* Inv(i+1) *)
  done ;;
```

Terminaison de l'algorithme. L'algorithme de tri par sélection est constitué de deux boucles `for` imbriquées, il termine donc !

Preuve de l'algorithme. La boucle `for` interne a pour effet de positionner la variable `i_min` à l'indice de l'élément minimal du tableau entre les indices `i` et `n-1`. Ainsi, un passage dans la boucle `for` externe positionne l'élément minimal du tableau entre les indices `i` et `n-1` en position `i`. Cette boucle `for` principale possède l'invariant suivant :

Inv_i : Les éléments du tableau entre les indices `0` et `i-1` sont triés dans l'ordre croissant et plus petits que les autres.

- Tout d'abord, Inv_0 est vrai : en effet, le sous tableau `t[0:0]` est vide.
- Clairement, si Inv_i est vrai en haut du corps de la boucle, Inv_{i+1} est vrai en bas du corps de boucle : en effet, on positionne le plus petit élément de parmi `t.(i),...,t.(n-1)` en position `i`.

Le compteur de boucle `i` prend toutes les valeurs entre 0 et $n-2$. Par suite, l'invariant $\text{Inv}_{n-2+1} = \text{Inv}_{n-1}$ est vérifié en sortie de boucle, ce qui implique que les $n-1$ premiers éléments du tableau sont triés en sortie de boucle, et plus petits que l'autre élément du tableau, à savoir `t.(n-1)`. Ainsi, le tableau est entièrement trié en sortie de boucle, donc en sortie de fonction, et le tri est correct.

En toute rigueur, il faudrait exhiber un invariant de boucle pour la boucle `for` interne, celui-ci est plutôt évident et est marqué dans le code.

Tri par insertion

Idée du tri : On maintient constamment la partie gauche du tableau trié. Lorsqu'on considère un nouvel élément `x` (celui juste à droite de la partie triée), il faut le faire « descendre » de façon à ce que cette portion augmentée de 1 élément soit triée. Pour ce faire, plutôt que de procéder par échanges, on sauvegarde la valeur de l'élément dans une variable. Il suffit ensuite de faire monter un à un les éléments du tableau *tant qu'*ils sont plus grands que `x`. Une fois ceci effectué, on peut positionner `x`.

Code Caml.

Le tri par insertion

```
let tri_insertion t =
  let n=Array.length t in
  for i=1 to n-1 do
    (* t.(0),..,t.(i-1) triés *)
    let x=t.(i) and j=ref i in
    while !j>0 && t.( !j-1)> x do
      (* Inv: Pour tout k vérifiant j<k<=i, L[k]>x *)
      t.( !j) <- t.( !j-1) ;
      decr j
      (* Inv *)
    done ;
    t.( !j) <- x
    (* Inv(i+1) *)
  done ;;
```

Terminaison de l'algorithme. L'algorithme de tri par insertion est constitué d'une boucle `while` dans une boucle `for`. Il faut donc montrer que pour tout $i \in \{1, \dots, n-1\}$, la boucle `while` termine, ce qui est à peu près évident : la variable j (on identifie encore référence et valeur) est initialisée à i juste avant la boucle, la condition de continuation du `while` comporte notamment la condition $j > 0$ et j est décrémenté à chaque tour de boucle. Notons que les indices du tableau considérés ne produisent jamais d'erreurs (d'accès en dehors du tableau). Remarquez que si $j = 0$, au niveau de la condition du `while`, alors la condition `!j>0` n'est pas vérifiée et on n'a pas besoin d'évaluer `t.(!j-1)> !x` (qui produirait un dépassement d'indice) pour s'apercevoir que la condition `!j>0 && t.(!j-1)> !x` est fausse. Ceci est dû au comportement *paresseux* de l'opérateur logique `&&`.

Preuve de l'algorithme. La boucle `while` admet pour invariant :

$$\text{Inv} : \text{Pour tout } k \text{ tel que } j < k \leq i, \texttt{t.(k)>x}.$$

En sortie de boucle `while`, la condition `!j>0 && t.(!j-1)> !x` est fausse, ainsi la boucle `for` possède l'invariant suivant :

$$\text{Inv}_i : \text{Les éléments } \texttt{t.(0),...,t.(i-1)} \text{ sont triés dans l'ordre croissant.}$$

En effet :

- Lorsque i vaut 1, `t.(0)` tout seul forme bien un ensemble trié dans l'ordre croissant, donc Inv_1 est vrai avant la boucle.
- Si, pour $i \in \{1, \dots, n-1\}$, Inv_i est vrai en haut du corps de boucle, alors Inv_{i+1} est vrai en bas du corps de boucle. En effet, après l'exécution de la boucle `while`, les éléments de `t.(k)` pour $j < k \leq i$ sont strictement supérieurs à `x` et ceux avant l'indice j (exclus) sont inférieurs ou égaux (avec `j` éventuellement nul). Ainsi, placer `x` en position `i` dans `t` mène à la portion triée `t.(0),...,t.(i+1)`.

Par suite, après l'exécution de la boucle `for` les éléments `t.(0),...,t.(n-1)` sont dans l'ordre croissant, et la fonction est correcte.

2.3 Implémentation d'une pile et d'une file dans un tableau

On a décrit quelques structures abstraites dans le chapitre précédent, on va voir que deux d'entre elles peuvent s'implémenter facilement à l'aide d'un tableau.

Avertissement : piles et files bornées ou non. De par la finitude de la mémoire utilisant une structure de pile ou de file, le nombre d'éléments que l'on peut mettre dans une pile ou une file est nécessairement borné. Toutefois, la borne est en général conséquente, si bien que l'on peut considérer que la capacité (nombre d'éléments que l'on peut stocker) est infinie : on parle de pile (ou file) non bornée. On va ici implémenter des stucture de pile et de file dans laquelle la capacité est fixée une fois pour toute à la création de l'objet : la pile (ou la file) est bornée. La fonction de création prendra donc en paramètre la capacité. L'explication est simple : on va utiliser des tableaux, dont la taille est elle-même fixée une fois pour toute : la taille du tableau sera égale à la capacité choisie [1].

2.3.1 Piles

Rappels sur la structure

On rappelle qu'une pile suit le principe « LIFO » : dernier arrivé, premier sorti. Les opérations à écrire pour implémenter une structure de pile sont les suivantes :

— création d'une pile vide ;
— test d'égalité au vide ;
— accès au sommet d'une pile non vide ;
— retrait de l'élément au sommet d'une pile non vide ;
— ajout d'un élément au sommet de la pile (non pleine ici).

Représentation dans un tableau

La représentation choisie est celle d'un enregistrement dans lequel sont stockés :

— la capacité `capacite` : nombre maximal d'éléments que l'on peut stocker dans la pile ;
— le nombre d'éléments `nb` présents dans la pile ;
— un tableau `contenu` de taille `capacite`, dont les `nb` premiers éléments sont les éléments présents dans la pile, les autres éléments du tableau ne sont pas des éléments de la pile. Le sommet est l'élément d'indice `nb-1`.

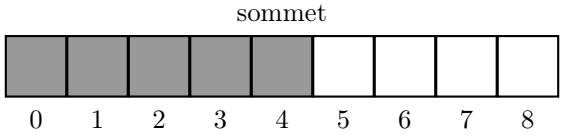


FIGURE 2.1 – Le tableau contenu associé à une pile de capacité 9, comportant 5 éléments. Le fond de la pile est à l'indice 0 (à gauche), le sommet à l'indice 4. Les éléments grisés sont ceux de pile, les éléments blancs sont quelconques. L'ajout d'un élément se ferait à l'indice 5 (il faut aussi incrémenter `nb`) et pour retirer un élément de la pile, il suffit de diminuer `nb`.

Remarque 2.2. *Le champ* `capacite` *est superflu, car la capacité d'une pile peut s'obtenir comme la longueur du tableau* `contenu`*. Néanmoins l'implémentation est plus claire avec ce champ supplémentaire.*

Implémentation concrète

On décide de donner un type pile polymorphe, le type est fixé à la création et la fonction de création prend donc deux paramètres : la capacité et un élément permettant de créer le tableau (néanmoins la pile est vide au départ).

```
type 'a pile = {capacite: int ; mutable nb: int ; contenu: 'a array} ;;

let creer_pile c x={capacite = c ; nb = 0 ; contenu = Array.make c x} ;;

let pile_vide p = p.nb = 0 ;;

let pile_pleine p = p.nb = p.capacite

let empiler p x=match pile_pleine p with
  | true -> failwith "pile pleine"
```

1. On pourrait implémenter une structure non bornée à l'aide d'un tableau redimensionnable, similaire aux listes Python (elles se comportent comme des tableaux mais on peut rajouter un élément avec `append`). Cette structure n'existe pas en Caml, mais on peut l'implémenter nous même !

```
  | false -> p.contenu.(p.nb) <- x ; p.nb <- p.nb + 1
;;

let sommet p=match pile_vide p with
  | true -> failwith "pile vide"
  | false -> p.contenu.(p.nb -1)
;;

let depiler p=match pile_vide p with
  | true -> failwith "pile vide"
  | false -> p.nb <- p.nb - 1 ; p.contenu.(p.nb)
;;
```

Voici un exemple de maniement d'une pile d'entiers :

```
# let p=creer_pile 5 0 ;;
val p : int pile = {capacite = 5; nb = 0; contenu = [|0; 0; 0; 0; 0|]}
# empiler p 1 ;;
- : unit = ()
# empiler p 2 ;;
- : unit = ()
# empiler p 3 ;;
- : unit = ()
# depiler p ;;
- : int = 3
# p ;;
- : int pile = {capacite = 5; nb = 2; contenu = [|1; 2; 3; 0; 0|]}
```

Attention : la pile `p` ne possède que deux éléments à la fin du processus, les deux premiers du tableau `contenu`. Les trois suivants ne font pas partie de la pile.

Complexité

À part à la création, toutes les opérations se font en temps constant ($O(1)$). La création est en $O(c)$, la capacité de la pile.

2.3.2 Files

Rappels sur la structure

On rappelle qu'une file suit le principe « FIFO » : premier arrivé, premier sorti. Les opérations à écrire pour implémenter une structure de pile sont les suivantes :

— création d'une file vide ;

— test d'égalité au vide ;

— retrait de l'élément en tête d'une file non vide ;

— ajout d'un élément en queue d'une file (non pleine ici).

Représentation dans un tableau

On va donner une réalisation d'une file *bornée* à partir d'un tableau, semblable à celle d'une pile. C'est un peu plus complexe pour une file, parce qu'on ne réalise pas l'ajout et la suppression d'éléments au même endroit. L'astuce est ici de considérer le tableau `contenu` comme « circulaire ». On utilise donc deux indices supplémentaires (mutables), qui indiquent les positions de la tête de file (le premier à avoir été inséré dans la file, donc le prochain à sortir), et de la queue (position du prochain élément qui va être inséré). La représentation choisie est donc celle d'un enregistrement dans lequel sont stockés :

— la capacité `capacite` ;

— le nombre d'éléments `nb` présents dans la file ;

— les indices `tete` et `queue` ;

— un tableau `contenu` dont les éléments entre les éléments présents dans la pile, les autres éléments du tableau ne sont pas des éléments de la pile.

FIGURE 2.2 – Deux tableaux associés à des files de capacité 9, comportant 5 éléments. Les éléments effectivement présents dans la file (grisés) sont entre les indices `tete` (inclus) et `queue` (exclus), le tableau est considéré comme circulaire : lorsqu'un élément est enfilé, le champ `queue` est incrémenté de 1 (modulo la capacité), et lorsqu'un élément est défilé, c'est le champ `tete`.

Remarque 2.3. *Là encore, le champ* `capacite` *est superflu. Il en va de même d'un des deux champs* `tete` *ou* `queue` *car la relation suivante est toujours vérifiée :*

$$queue - tete \equiv nb[capacite]$$

Par contre, le champ `nb` *ne peut s'obtenir à partir de* `tete` *et* `queue`*, car lorsque ces deux indices sont égaux, c'est* `nb` *qui permet de faire la distinction entre une file pleine et une file vide.*

Implémentation concrète

```
type 'a file={capacite: int; mutable nb: int; mutable tete: int; mutable queue: int; contenu: 'a array} ;;

let creer_file c x={capacite = c ; nb = 0 ; tete = 0 ; queue = 0 ; contenu=Array.make c x} ;;

let file_vide f=f.nb = 0 ;;

let file_pleine f=f.nb = f.capacite ;;

let enfiler f x=match file_pleine f with
  | true -> failwith "file pleine"
  | false -> f.contenu.(f.queue) <- x ; f.queue <- (f.queue + 1) mod f.capacite ; f.nb <- f.nb + 1
;;

let defiler f = match file_vide f with
  | true -> failwith "file vide"
  | false -> let x=f.contenu.(f.tete) in f.tete <- (f.tete + 1) mod f.capacite ; f.nb <- f.nb - 1 ; x
;;
```

Voici un exemple avec une file d'entiers :

```
# let f=creer_file 5 0 ;;
val f : int file =
  {capacite = 5; nb = 0; tete = 0; queue = 0; contenu = [|0; 0; 0; 0; 0|]}
# for i=1 to 3 do enfiler f i done ;;
- : unit = ()
# defiler f ;;
- : int = 1
# f;;
- : int file =
{capacite = 5; nb = 2; tete = 1; queue = 3; contenu = [|1; 2; 3; 0; 0|]}
```

Les éléments présents dans la file (2 et 3), sont bien entre l'indice de tête inclus et l'indice de queue exclu.

Complexité

Là encore, toutes les opérations se font en temps constant $O(1)$, exceptée la création en $O(c)$, la capacité de la file.

Chapitre 3

Récursivité et listes

3.1 Exemple introductif : la fonction factorielle

Une première définition basique d'une fonction récursive est plutôt simple : c'est une fonction qui s'appelle elle-même. Prenons un exemple classique : le calcul de la factorielle d'un entier positif. On définit, pour $n \geq 0$, $n! = \prod_{i=1}^{n} i$. En Caml, on peut donc définir la factorielle comme :

```
let fact n=
  let f = ref 1 in
  for i=1 to n do
    f:= !f * i
  done ;
  !f
;;
```

Une autre définition mathématique classique de la factorielle se fait par *récurrence* :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon.} \end{cases}$$

En reprenant quasiment mot pour mot cette dernière définition, on obtient la fonction Caml suivante :

```
let rec fact_rec n=match n with
  | 0 -> 1
  | _ -> n * fact_rec (n-1)
;;
```

qui fonctionne tout aussi bien ! Remarquez l'usage de `rec` lors de la définition, qui indique au compilateur Caml que l'objet qu'on définit est une fonction récursive [1]. On distingue clairement deux cas dans cette fonction :

— le cas $n = 0$, appelé cas terminal ;

— le cas $n > 0$, qui produit un appel *récursif* à la fonction `fact_rec`.

3.2 Pratique de la récursivité

3.2.1 Ce qui se passe « en interne » : la pile d'appels

En informatique, la pile d'exécution (ou pile d'appels, *call stack* en anglais) est une structure de données de type pile, qui sert à enregistrer des informations au sujet des fonctions actives dans un programme. Une fonction active est une fonction dont l'exécution n'est pas encore terminée.

L'utilisation principale de la pile d'appels est de garder la trace de l'endroit où chaque fonction active doit retourner à la fin de son exécution. En pratique, lorsqu'une fonction est appelée par un programme, son adresse de retour (adresse de l'instruction qui suit l'appel) est empilée sur la pile d'appels. En plus d'emmagasiner des adresses de retour, la pile d'exécution stocke aussi d'autres valeurs, comme les variables locales de la fonction, les paramètres de la fonction, etc [2]...

1. Ce qui est nécessaire, car sinon la définition de `fact_rec` requiert celle d'une fonction `fact_rec`, non encore définie !

2. Ce mécanisme est totalement transparent pour l'utilisateur d'un langage de haut niveau comme Caml. On ne rentrera pas dans les détails, le lecteur intéressé pourra consulter `https://en.wikipedia.org/wiki/Call_stack`

En particulier, lors d'appels *imbriqués* c'est-à-dire lorsqu'une fonction f appelle une fonction g, ce qui est relatif à l'appel de la fonction g est placé juste au dessus de ce qui est relatif à la fonction f. Lorsque g termine son exécution, ce qui est relatif à l'exécution de g est *dépilé*. Comme l'adresse de retour est contenu dans la pile d'appel, l'exécution de f peut reprendre juste après l'endroit où g a été appelée.

Une fonction récursive est essentiellement une fonction qui s'appelle elle-même, ainsi les appels successifs à f s'empile dans la pile d'appels (voir figure 3.1).

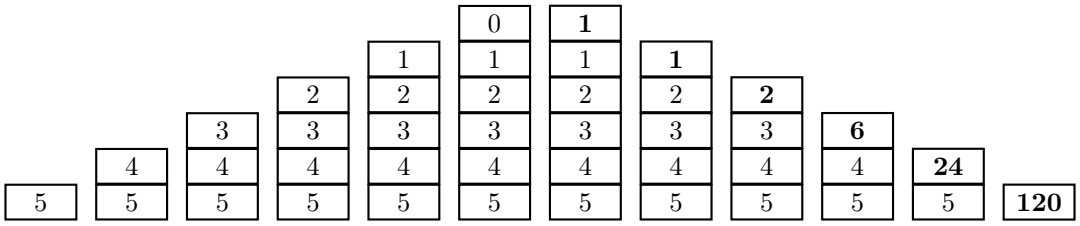


FIGURE 3.1 – La pile d'exécution (partie relative à l'exécution de `fact_rec 5`) : en clair les paramètres d'appels, en gras les valeurs de retour.

Une fois arrivé à un cas terminal (ne produisant pas d'appel récursif), le nombre d'éléments de la pile d'appels se réduit. Dans le cas de la fonction factorielle, comme celle-ci ne rappelle qu'une fois, dés lors qu'on a commencé à dépiler on ne s'arrête plus (ce moment correspond au milieu de la figure 3.1). Enfin, la récursion s'arrête lorsqu'on dépile l'élément correspondant au premier appel de la fonction. On peut vérifier ce comportement en *traçant* la fonction `fact_rec` : cela permet d'imprimer à l'écran les entrées et sorties d'une fonction :

```
# #trace fact_rec ;;
fact_rec is now traced.
# fact_rec 5 ;;
fact_rec <-- 5
fact_rec <-- 4
fact_rec <-- 3
fact_rec <-- 2
fact_rec <-- 1
fact_rec <-- 0
fact_rec --> 1
fact_rec --> 1
fact_rec --> 2
fact_rec --> 6
fact_rec --> 24
fact_rec --> 120
- : int = 120
```

On voit que le nombre *d'appels imbriqués* réalisés par une fonction récursive peut être important : il faut stocker ces appels, ce qui est coûteux en mémoire. En pratique, on peut voir la pile d'appels comme une pile de capacité finie : si la pile est pleine, un appel supplémentaire produit un dépassement de capacité : le fameux *stack overflow* en anglais.

3.2.2 Récursivité terminale

Un exemple : la fonction somme

Voici un autre exemple de fonction récursive : le calcul de la somme des entiers de 0 à n, pour $n > 0$. Elle est facile à écrire :

```
let rec somme n=match n with
  | 0 -> 0
  | _ -> n + somme (n-1)
;;
```

Cette fonction est très proche de la fonction factorielle écrite plus haut. Testons quelques entiers :

```
#somme 100 ;;
- : int = 5050
#somme 1000 ;;
- : int = 500500
# somme 1000000 ;;
Stack overflow during evaluation (looping recursion?).
```

Dans le dernier exemple, la capacité de la pile d'appels a été dépassée [3] : le calcul n'a pu aboutir par manque de mémoire.

Écrivons une autre fonction :

```
let rec somme_avec_acc n acc=match n with
  | 0 -> acc
  | _ -> somme_avec_acc (n-1) (n+acc)
;;
```

Cette fonction utilise un *accumulateur* `acc`, dans lequel se font les additions. Observons quelques appels avec `acc=0` :

```
#somme_avec_acc 100 0 ;;
- : int = 5050
#somme_avec_acc 1000 0 ;;
- : int = 500500
#somme_avec_acc 1000000 0 ;;
- : int = 500000500000
```

Cette fonction permet donc de calculer la somme des entiers de 0 à n, mais ne semble pas souffrir du grossissement de la pile d'appels. En effet, cette fonction est *récursive terminale*.

Définition de la récursivité terminale

Définition 3.1. *Une fonction f récursive est dite récursive terminale si tout appel récursif est le dernier calcul réalisé par f.*

La fonction `somme_avec_acc` respecte bien cette définition : lorsque l'on n'est pas dans le cas terminal $n = 0$, la fonction calcule $n - 1$ et $n +$ acc avant de réaliser l'appel récursif, qui est le dernier calcul effectué lors de cet appel de fonction. L'intérêt d'avoir une fonction f récursive terminale est qu'un appel récursif à f ne nécessite pas d'empilement sur la pile d'appels : l'appel récursif effectué peut prendre la place de l'appel en cours dans la pile d'appels, puisqu'il n'y a plus d'instructions à effectuer une fois l'appel récursif terminé. Certains compilateurs, dont le compilateur Caml [4], sont capables de détecter les fonctions récursives terminales, et d'en tirer profit pour diminuer les empilements dans la pile d'appels.

En Caml, on fait donc souvent une transformation similaire à la précédente pour transformer une fonction récursive en fonction récursive terminale. Il reste un petit détail : il est un peu pénible de devoir passer 0 en paramètre de la fonction pour la valeur initiale de l'accumulateur. Une solution est de définir la fonction `somme_avec_acc` comme interne à une autre fonction, qui se contentera de l'appeler avec `acc=0`. La fonction `somme_avec_acc` n'existant plus dans le programme principal, on peut lui donner un nom plus court (souvent `aux`, pour auxiliaire) :

```
let somme2 n=
  let rec aux n acc = match n with
    | 0 -> acc
    | _ -> aux (n-1) (acc + n)
  in aux n 0
;;
```

3.2.3 Deux exemples de fonctions récursives

Algorithme d'Euclide. Une méthode récursive de calcul du PGCD de deux entiers positifs non tous deux nuls est la suivante :

$$\text{PGCD}(a,b) = \begin{cases} a & \text{si } b = 0\,; \\ \text{PGCD}(b, a \bmod b) & \text{sinon.} \end{cases}$$

Cette méthode mène à la fonction récursive terminale suivante :

```
let rec pgcd a b=match b with
  | 0 -> a
  | _ -> pgcd b (a mod b)
;;
```

Par exemple :

```
#pgcd 1898615 16586155318 ;;
- : int = 1
```

3. L'interpréteur OCaml se demande si cela est du à une fonction récursive qui ne termine pas : ce n'est pas le cas ici !
4. Python, par exemple, ne gère pas la récursivité terminale.

Algorithme d'exponentiation rapide. L'algorithme d'exponentiation rapide se reformule très facilement en faisant usage de récursivité. En effet, pour $n \geq 0$, on a :

$$x^n = \begin{cases} 1 & \text{si } n = 0; \\ \left(x^{n/2}\right)^2 & \text{si } n \text{ est pair;} \\ x \times \left(x^{n/2}\right)^2 & \text{sinon.} \end{cases}$$

On en déduit le code :

```
let rec expo_rapide x n=match n with
  | 0 -> 1
  | _ -> let y=expo_rapide x (n/2) in if n mod 2 = 0 then y*y else y*y*x
;;
```

Cette fonction n'est pas récursive terminale, ce qui n'est pas gênant ici : seulement $O(\log n)$ appels récursifs sont effectués.

3.2.4 Récursivité croisée

On rappelle que `and` permet de définir simultanément plusieurs objets. Il est possible de définir simultanément plusieurs fonctions, qui s'appellent mutuellement (on parle de récursivité mutuelle ou croisée). Par exemple :

```
let rec pair n=match n with
  | 0 -> true
  | _ -> impair (n-1)
and impair n=match n with
  | 0 -> false
  | _ -> pair (n-1)
;;
```

Testons :

```
#pair 5 ;;
- : bool = false
#pair 6 ;;
- : bool = true
```

Cet exemple est assez artificiel, le suivant l'est un peu moins. On suppose les variables globales `u0` et `v0` définies, ce sont deux flottants positifs.

```
let rec u n=match n with
  | 0 -> u0
  | _ -> (u (n-1) +. v(n-1))/. 2.
and v n=match n with
  | 0 -> v0
  | _ -> sqrt (u (n-1) *. v(n-1))
;;
```

Ces fonctions définissent deux suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$. Un exercice classique de mathématiques consiste à montrer qu'elles sont adjacentes, leur limite commune est appelée la *moyenne arithmético-géométrique* de u_0 et v_0. La convergence est très rapide :

```
#u 3 ;;
- : float = 2.24303398875
#v 3 ;;
- : float = 2.24302317183
#u 4 ;;
- : float = 2.24302858029
#v 4 ;;
- : float = 2.24302858028
```

On a pris ici $u_0 = 1$ et $v_0 = 4$. Notons que les deux fonctions `u` et `v` ont malheureusement une grande complexité, comme expliqué dans la sous-section suivante.

Pour prendre en compte la récursivité croisée, on peut donner une autre définition de la récursivité [5] :

Définition 3.2. *Une fonction f est dite récursive lorsque dans la pile d'appels peuvent se trouver simultanément plusieurs appels à f.*

5. qui vaut ce qu'elle vaut...

3.2.5 Attention aux appels qui se chevauchent !

Considérons l'exemple suivant : soit $(F_n)_{n\in\mathbb{N}}$ la suite définie par

$$F_n = \begin{cases} 1 & \text{si } n = 0 \text{ ou } 1. \\ F_{n-1} + F_{n-2} & \text{sinon} \end{cases}$$

Vous aurez probablement reconnu la fameuse suite de Fibonacci. Une transcription récursive en Caml s'obtient aisément :

```
let rec fibo n=match n with
  | 0 | 1 -> 1
  | _ -> fibo (n-1) + fibo (n-2)
;;
```

Le problème de la fonction précédente est sa complexité, qui réside dans le nombre d'appels récursifs effectués. Notons A_n le nombre d'appels récursifs nécessaires pour le calcul de F_n. Alors, la suite (A_n) vérifie la relation de récurrence $A_0 = A_1 = 0$ et pour tout $n \geq 2$, $A_n = 2 + A_{n-1} + A_{n-2}$, soit $A_n + 2 = (A_{n-2} + 2) + (A_{n-1} + 2)$. Autrement dit, la suite $(A_n + 2)_{n\in\mathbb{N}}$ coïncide avec la suite $(2F_n)_{n\in\mathbb{N}}$, d'où $A_n = \Theta(\varphi^n)$, avec $\varphi = \frac{1+\sqrt{5}}{2} > 1$. La complexité de la fonction `fibo` est donc exponentielle en n, ce qui n'est pas étonnant si on schématise les appels récursifs effectués, comme en figure 3.2.

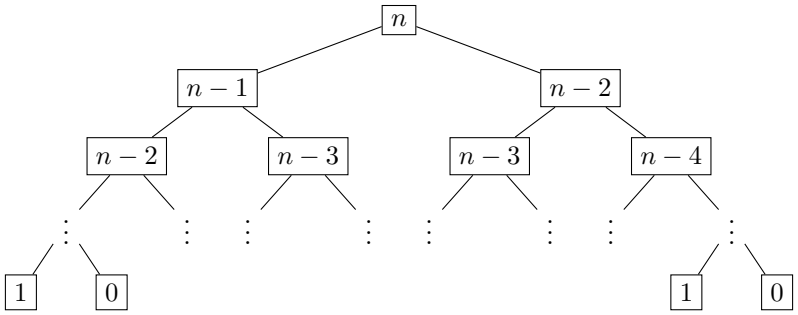


FIGURE 3.2 – L'arbre d'appels récursifs pour le calcul de $(F_n)_{n\in\mathbb{N}}$.

On parle ici de *chevauchement des appels récursifs* : la fonction, bien que correcte, nécessite de réaliser plusieurs fois les mêmes calculs pour aboutir. Elle est impraticable pour des grands n. Déja le calcul de F_{40} nécessite plus de 15 secondes sur mon ordinateur personnel, et chaque incrémentation de n demande une multiplication du temps de calcul par environ 1.618.

```
# let a=Sys.time() in let _ = fibo 40 in Sys.time() -. a ;;
- : float = 15.7569840000000028
# let a=Sys.time() in let _ = fibo 41 in Sys.time() -. a ;;
- : float = 25.4530430000000081
```

Le même phénomène fait que les fonctions `u` et `v` de la sous-section précédente sont inefficaces. Donnons deux méthodes pour éviter cet écueil :

— utiliser une fonction itérative, mais on perd la formulation récursive ;

— utiliser un dictionnaire pour stocker les valeurs déja calculées.

En conclusion, il faut faire attention à ne pas faire des appels récursifs qui se recoupent, sous peine de voir la complexité exploser !

3.3 Un exemple plus complet : les tours de Hanoï

On a déja vu un avantage des fonctions récursives : leur formulation est plus simple que leur équivalent itératif. Montrons un exemple de problème pour lequel une solution récursive est très adaptée, mais pour lequel une solution itérative n'est pas facile à trouver : le problème des tours de Hanoï.

On dispose de n disques troués en leur centre, numérotés de 1 à n, de diamètres croissants. On se donne également 3 piquets (numérotés de A à C). Initialement, tous les disques sont enfilés sur le premier piquet, le plus grand étant à la base, le plus petit au sommet, comme sur la figure 3.3.

Le but du jeu est d'amener les disques sur le troisième piquet, en suivant les règles suivantes :

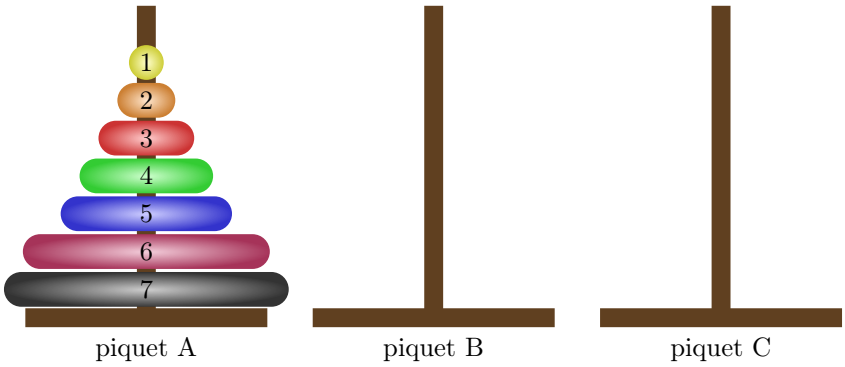


FIGURE 3.3 – Le jeu de Hanoï : comment déplacer les 7 disques du piquet A au piquet C, en suivant les règles ?

— déplacer les disques un à un d'un piquet à un autre ;
— un disque ne doit jamais être posé sur un disque de diamètre inférieur.

La figure 3.4 montre les 7 mouvements à effectuer pour résoudre le jeu avec seulement 3 disques. On peut vérifier qu'il faut par exemple 127 mouvements pour le jeu à 7 disques (voir la suite).

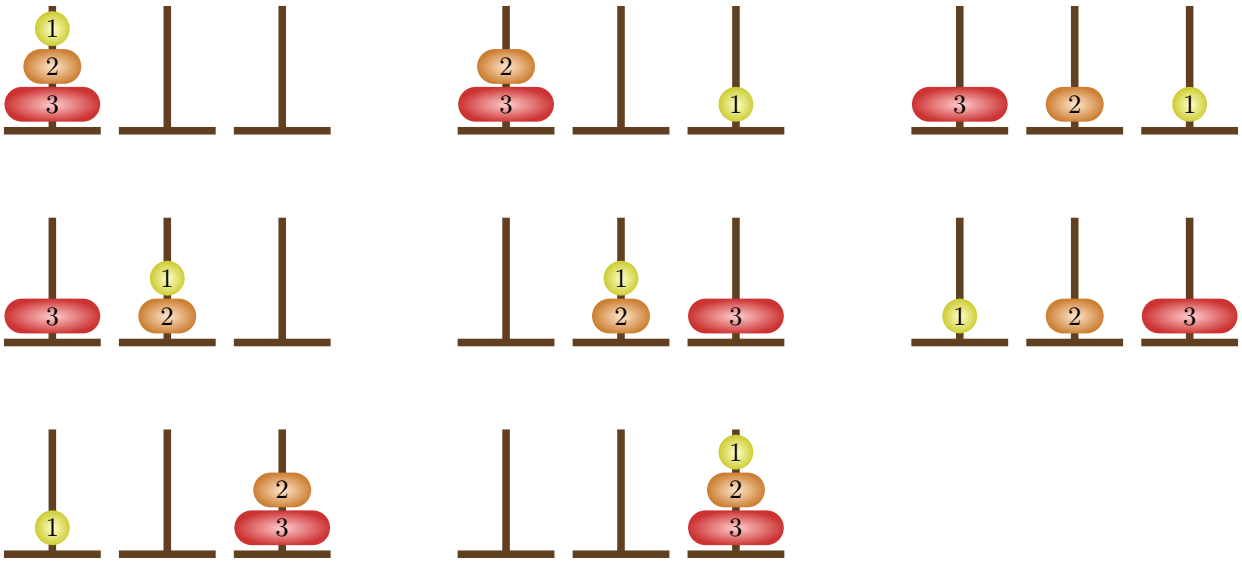


FIGURE 3.4 – Résolution du jeu de Hanoi pour $n = 3$

On cherche à donner les mouvements de disques à effectuer pour résoudre le jeu. De manière itérative, il n'est pas évident à résoudre, mais il est très facile de le faire lorsqu'on pense à la récursivité. Soient i, j et k trois caractères tels que $\{i, j, k\} = \{A, B, C\}$, et $n \in \mathbb{N}$. Pour faire passer n disques du piquet i au piquet j :

— il n'y a rien à faire si $n = 0$;
— pour $n \geq 1$, il suffit de faire passer les $n-1$ disques numérotés de 1 à $n-1$ du piquet i au piquet k, de déplacer ensuite le disque n du piquet i au piquet j, puis de refaire passer les $n-1$ disques du piquet k au piquet j. Le fait de travailler avec les disques les plus petits permet de ne pas violer la deuxième règle.

Écrivons donc une fonction Caml qui imprime à l'écran la suite des mouvements à effectuer pour résoudre le jeu. Un mouvement est décrit comme `i -> j`, ce qui signifie faire passer le disque supérieur du piquet i au piquet j (l'opérateur `^` permet la concaténation de chaînes de caractères) :

```
let deplacement i j=
  print_string (i^" -> "^j^"\n")
;;
```

La discussion précédente invite à écrire une fonction `hanoi n` résolvant le jeu à n disques, qui fait un unique appel à une fonction récursive interne `aux n i j k` qui doit donner la suite des mouvements permettant de faire passer les

n plus petits disques du piquet i au piquet j, avec $\{i, j\} \subset \{A, B, C\}$. Pour des raisons de commodité, il est pratique d'indiquer le dernière lettre parmi $\{A, B, C\}$ dans une variable (k) :

```
let hanoi n=
  let rec aux i j k n=match n with
    | 0 -> ()
    | _ -> aux i k j (n-1) ; deplacement i j ; aux k j i (n-1)
  in aux "A" "C" "B" n ;
;;
```

Testons avec $n = 3$:

```
>>> hanoi 3
A  ->  C
A  ->  B
C  ->  B
A  ->  C
B  ->  A
B  ->  C
A  ->  C
```

On retrouve les mouvements de la figure 3.4. Il est facile de montrer par récurrence que le nombre de mouvements produits est $2^n - 1$: c'est optimal [6].

Comme on le voit sur cet exemple, les fonctions où plusieurs appels récursifs sont nécessaires ne sont pas vraiment faciles à traduire de façon itérative (à moins d'utiliser une pile pour essentiellement réécrire la récursivité...) : c'est un avantage de l'emploi de fonctions récursives.

3.4 Structure de liste chaînée

La liste chaînée est l'exemple le plus simple de *type récursif*, ou *type défini par induction* : le type se retrouve dans la définition. Caml est extrêmement pratique pour définir et manipuler ces types, comme on va le voir par la suite. Naturellement, les fonctions qui manipulent des types récursifs sont le plus souvent récursives.

3.4.1 Définition

Définition 3.3. *Une liste chaînée d'éléments de type t est :*

- *soit la liste vide (souvent notée* `[]`*) ;*
- *soit la donnée d'un élément x de type t et d'une liste chaînée ℓ d'éléments de type t, qu'on notera $Cons(x, \ell)$.*

Dans la définition précédente, on appelle x la *tête* de liste, et ℓ la *queue* de la liste (c'est une liste !). Cette manière récursive de définir le type liste chaînée est essentiellement ce qu'on va faire en Caml. Une définition équivalente (mais inductive) est la suivante :

Définition 3.4. *L'ensemble des listes chaînées d'éléments de type t est le plus petit ensemble $\mathcal{C}$ tel que :*

- *la liste vide* `[]` *appartient à $\mathcal{C}$;*
- *si x est de type t et ℓ une liste chaînée d'éléments de type t, alors $Cons(x, \ell)$ appartient à $\mathcal{C}$.*

En figure 3.5 est représentée une liste chaînée d'entiers, à savoir :

$$\text{Cons}(12, \text{Cons}(99, \text{Cons}(37, \text{Cons}(8, [\,]))))$$

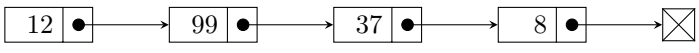


FIGURE 3.5 – La représentation interne d'une liste chaînée

6. La fonction a aussi une complexité en $O(2^n)$, ce qui est exponentiel... Mais là on ne peut pas faire mieux : ce qu'on cherche à calculer est de taille exponentielle en n.

3.4.2 Le type list en Caml

En Caml, le type `list` est le type liste chaînée. On peut définir une liste en donnant la séquence des éléments de la liste, entre crochets. Voici la liste précédente, en Caml.

```
#[12; 99; 37; 8] ;;
- : int list = [12; 99; 37; 8]
```

On remarque que le type est `int list`, les listes chaînées en Caml sont constituées d'éléments homogènes, comme les tableaux. Voici comment obtenir une liste chaînée à partir d'une autre et d'un nouvel élément, en suivant la définition :

```
#12::[99; 37; 8] ;;
- : int list = [12; 99; 37; 8]
```

L'opérateur `::` (qui se lit « Conse ») est un opérateur infixe, `x::q` donne une `'a list` si `x` est de type `'a` et `q` de type `'a list`. La version préfixe permet de s'en convaincre[7] :

```
# let pref_cons (x,q) = x::q ;;
val pref_cons : 'a * 'a list -> 'a list = <fun>
```

Inversement, à partir d'une liste non vide, on peut accéder à sa tête et à sa queue via les fonctions `hd` (head) et `tl` (tête) du module `List` :

```
# List.hd ;;
- : 'a list -> 'a = <fun>
# List.tl ;;
- : 'a list -> 'a list = <fun>
# let q=["a"; "b"; "c"] ;;
val q : string list = ["a"; "b"; "c"]
# List.hd q ;;
- : string = "a"
# List.tl q ;;
- : string list = ["b"; "c"]
```

Ceci dit, les fonctions `hd` et `tl` sont rarement utilisées, on préfère fonctionner par filtrage sur les listes : l'opérateur `::` peut être utilisé comme motif de filtrage, voici comment recoder `hd` et `tl` :

```
let tete q=match q with
  | [] -> failwith "liste vide"
  | x::_ -> x
;;

let queue q=match q with
  | [] -> failwith "liste vide"
  | _::p -> p
;;
```

Complexité des opérations. Les trois opérations `hd`, `tl` et `::` ont toute une complexité constante. Ceci peut paraître étonnant car on crée de nouvelles listes via ces opérations, mais elles ne sont pas recopiées entièrement : les éléments sont partagés au maximum pour diminuer la complexité (en temps comme en mémoire), voir figure 3.6.

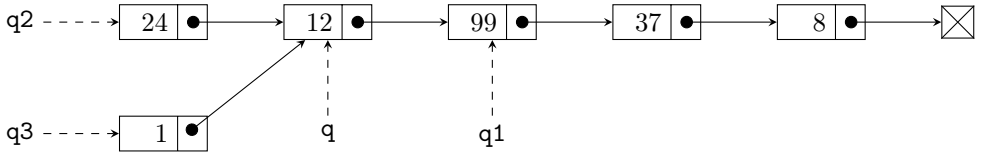


FIGURE 3.6 – La liste `q=[12; 99; 37; 8]`, et les listes `q1=List.tl q`, `q2=24::q` et `q3=1::q`

7. Remarque : la fonction `List.cons` existe depuis la version 4.03 de Ocaml.

Résumé et avertissement. On n'a parlé que de l'élément de tête dans cette présentation : en effet, lorsqu'on manipule une liste chaînée, *seul l'élément en tête de liste est accessible*. Les listes chaînées sont très différentes des tableaux (`array`) :

— un tableau est de taille fixée, et on peut accéder et modifier ses éléments en temps constant (on rappelle que l'on parle de structure mutable) ;

— une liste est immuable (ou non mutable, ou persistante), l'accès à la tête se fait en temps constant, de même que la construction de nouvelles listes avec `List.tl` et `::`.

On fera donc très attention à ne pas confondre les deux structures.

3.4.3 Exemples de fonctions sur les listes

Parcours d'une liste. Le type liste étant récursif, les fonctions sur les listes sont en général récursives. La fonction suivante est l'archétype d'une fonction sur une liste :

```
let rec parcours q=match q with
  | [] -> ()
  | x::p -> parcours p
;;
```

Elle se contente de parcourir la liste passée en paramètre, en entier, et ne fait rien d'autre. Mais elle est typique d'une fonction sur une liste :

— cette fonction est récursive ;

— le filtrage comporte le motif de la liste vide `[]`, et le motif d'une liste non vide `x::p`.

Longueur d'une liste. La fonction suivante donne la longueur d'une liste (qui existe déja en OCaml : `List.length`).

```
let rec longueur q=match q with
  | [] -> 0
  | _::p -> 1 + longueur p
;;
```

Attention : la complexité de cette fonction est linéaire en la taille de la liste (c'est le cas de `List.length` également, mais la version Caml est récursive terminale).

Tester l'appartenance d'un élément à une liste. La fonction suivante implémente la fonction `List.mem` de Caml[8].

```
let rec appartient x q=match q with
  | [] -> false
  | y::p -> y=x || appartient x p
;;
```

Un exemple un peu plus complexe. L'opérateur `::` peut être utilisé plusieurs fois dans un motif de filtrage. La fonction suivante teste si une liste possède deux éléments consécutifs égaux.

```
let rec deux_egaux q = match q with
  | [] | [_] -> false
  | x::y::p when x=y -> true
  | x::p -> deux_egaux p
;;
```

Miroir d'une liste. La fonction `List.rev` de Caml renvoie la liste « miroir » de la liste passée en argument. En voici une implémentation, où on utilise une fonction auxiliaire et un accumulateur qu'on remplit au fur et à mesure que l'on lit la liste passée en entrée.

```
let miroir q=
  let rec aux acc q=match q with
  | [] -> acc
  | y::p -> aux (y::acc) p
  in aux [] q
;;
```

8. « mem » pour member en anglais.

3.4.4 Construction de listes

Il existe plusieurs manières de construire des listes : une possibilité est de donner la liste des éléments « à la main ». Sinon, il faut la construire via une structure itérative ou en faisant usage de récursivité. Donnons deux méthodes pour construire la liste des entiers de 0 à $n-1$.

Fonction récursive. La fonction suivante convient, par exemple. On se force à écrire une fonction récursive terminale ici.

```
let liste_ent n =
  let rec aux acc p = match p<0 with
    | true -> acc
    | false -> aux (p::acc) (p-1)
  in aux [] (n-1)
;;
```

Fonction itérative. Une autre possibilité est d'utiliser une boucle, et une référence vers une liste (rappel : une liste est immuable, par contre on peut changer l'objet pointé par une réféfence).

```
let liste_ent n =
  let q = ref [] in
  for i = n-1 downto 0 do
    q:= i :: !q
  done ;
  !q
;;
```

3.4.5 Une implémentation personnalisée des listes

Il est très facile de réécrire une implémentation personnalisée des listes Caml, à l'aide d'un type somme :

```
type 'a liste = Vide | Cons of 'a * 'a liste ;;
```

On suit ici la définition 3.3. Voici la définition du type et de l'opérateur `cons` et des fonctions `tete` et `queue`, analogues de `::`, `List.hd` et `List.tl` :

```
let tete q=match q with
  | Vide -> failwith "Vide"
  | Cons (x, _) -> x
;;

let queue q=match q with
  | Vide -> failwith "Vide"
  | Cons (_, p) -> p
;;

let cons (x,q)=Cons (x, q) ;;
```

Voici les types des fonctions écrites :

```
tete : 'a liste -> 'a
queue : 'a liste -> 'a liste
cons : 'a * 'a liste -> 'a liste
```

C'est quasiment de cette manière que sont implémentées les listes en Caml, on remarquera que `hd`, `tl` et la version préfixe de `::` ont les mêmes types que ceux-ci dessus, en remplaçant `liste` par `list`.

3.4.6 Implémentation de structures de pile et de file à l'aide de listes chaînées

On va implémenter une structure de pile et une structure de file à l'aide de listes chaînées. La différence avec l'implémentation sur les tableaux est que la taille peut varier : il est peu coûteux (complexité $O(1)$) d'ajouter un élément en tête d'une liste pour obtenir une nouvelle liste. On peut donc réaliser des piles et files *non bornées* : leur capacité est virtuellement [9] infinie. On n'aura pas non plus à préciser un élément à la création pour spécifier le type (comme on l'avait fait pour les tableaux), la liste vide étant polymorphe :

9. En pratique, la capacité reste naturellement bornée par la mémoire.

```
#[] ;;
- : 'a list = []
```

Structure de pile

La structure de pile avec son seul sommet accessible est très similaire à une liste chaînée avec son élément de tête accessible. Ainsi une liste chaînée est quasiment une pile ! On réalise ici une structure mutable, constituée d'un unique champ (modifiable) contenant une liste.

```
type 'a pile = {mutable contenu: 'a list} ;;
```

La seule différence avec le type des opérations de pile écrits pour des tableaux est la création, qui ne prend pas d'argument :

```
let creer_pile () = {contenu=[]} ;;
let pile_vide p = p.contenu = [] ;;
let empiler p x=p.contenu <- x::p.contenu ;;

let sommet p = match p.contenu with
  | [] -> failwith "pile vide"
  | x::_ -> x
;;

let depiler p = match p.contenu with
  | [] -> failwith "pile vide"
  | x::q -> p.contenu <- q ; x
;;
```

Complexité des opérations. Elles se font toutes en temps constant $O(1)$.

Structure de file

Implémenter une structure de file à l'aide de listes n'est pas aussi simple qu'une structure de pile : en effet, seule la tête de liste est accessible. Mais on peut en utiliser deux ! Cela mène au type suivant :

```
type 'a file = {mutable entree: 'a list ; mutable sortie: 'a list} ;;
```

La première est utilisée pour rajouter des éléments (en queue de file), la seconde pour enlever des éléments (en tête de file), comme le montre la figure suivante :

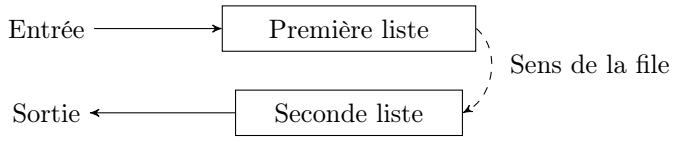


FIGURE 3.7 – Réalisation d'une file à l'aide de deux listes. Le sens de la file est indiqué par une flèche.

La seule difficulté dans l'écriture des opérations se situe dans l'opération « défiler », lorsque la seconde liste est vide : c'est l'élément en bout de la première liste qui doit sortir. Pour garder une complexité (presque) constante, il suffit d'abord de remplacer la deuxième liste (vide) par la première, retournée (et la première par une liste vide). Voici les opérations associées :

```
let creer_file () = {entree = []; sortie = []} ;;
let file_vide f=f.entree = [] && f.sortie = [] ;;
let enfiler f x=f.entree <- x::f.entree ;;

let rec defiler f=match f.sortie with
  | [] when f.entree = [] -> failwith "file vide"
  | [] -> f.sortie <- List.rev f.entree ; f.entree <- [] ; defiler f
  | x::q -> f.sortie <- q ; x
;;
```

Remarquez l'utilisation de la récursivité pour simplifier l'écriture de `defiler`.

Complexité des opérations. Toutes les opérations se font en temps constant, sauf lorsqu'il faut défiler alors que la seconde liste est vide. Dans ce cas, la complexité est linéaire en le nombre d'éléments présents dans la file, ce qui est fâcheux. Toutefois, ce cas-là ne se produit pas trop souvent. En effet lorsqu'on défile alors qu'il y a n éléments dans la file (et que la seconde liste est vide), le coût est $O(n)$, mais néanmoins les $n-1$ prochaines opérations « défiler » se feront en temps constant : sur ces n opérations, il y a un coût total $O(n)$, donc $O(1)$ en moyenne. On dit que la complexité *amortie* de l'opération « défiler » est $O(1)$.

```
# let f=creer_file () ;;
val f : '_a file = {entree = []; sortie = []}
#for i=0 to 999 do enfiler f i done ;;
- : unit = ()
#defiler f ;; (* cette opération est coûteuse *)
- : int = 0
#f ;; (* on peut défiler 999 fois en temps O(1) *)
- : int file =
 {entree = []; sortie = [1; 2; 3; 4; 5; 6; 7; 8; 9;...]}
```

Remarque 3.5. *De manière générale, on peut de la même façon réaliser une structure de file dès que l'on a une implémentation d'une structure de pile, il suffit d'en utiliser deux.*

3.5 Un exemple fondamental : le tri fusion

Principe du tri. On termine ce chapitre par la présentation d'un algorithme fondamental : le tri fusion. C'est un algorithme qui suit le principe « diviser pour régner » dont on reparlera plus tard. Le principe du tri fusion pour le tri d'une liste ℓ est le suivant :

— si la liste ℓ possède zéro ou un élément, elle est triée ;

— sinon, il suffit de couper la liste en deux parties ayant le même nombre d'éléments (à un près), de les trier récursivement, et de fusionner les deux listes triées obtenues.

Écriture en Caml. Pour pouvoir écrire la fonction de tri, il faut au préalable écrire deux fonctions :

— l'une (`fission`), qui divise une liste en deux listes de même taille à un élément près ;

— l'autre (`fusion`), qui prend en entrée deux listes triées dans l'ordre croissant, et renvoie la liste triée constituée des éléments des deux listes.

Voici des implémentations possibles :

```
let rec fission q=match q with
  | [] | [_] -> q, []
  | x::y::p -> let a,b=fission p in x::a, y::b
;;

let rec fusion p1 p2=match p1, p2 with
  | [],_ -> p2
  | _, [] -> p1
  | x::q1, y::_ when x<=y -> x::(fusion q1 p2)
  | _,x::q2 -> x::(fusion p1 q2)
;;
```

Une fois ces deux fonctions écrites, l'implémentation du tri est un jeu d'enfant (attention à ne pas oublier le cas terminal) :

```
let rec tri_fusion p=match p with
  | [] | [_] -> p
  | _ -> let a,b=fission p in fusion (tri_fusion a) (tri_fusion b)
;;
```

Complexité. L'intérêt d'un tel tri par rapport aux tris déja vus (tri par insertion, tri par sélection...) paraît obscur. Néanmoins on montrera dans le chapitre suivant que la complexité du tri fusion est $O(n\log n)$ avec n la taille de la liste à trier : c'est un gain considérable par rapport aux tris évoqués, quadratiques (de complexité $O(n^2)$).

Tri d'une liste en OCaml. La fonction `List.sort` de Caml trie une liste, avec le tri fusion ! Elle est très semblable à l'implémentation donnée précédemment, mais un peu plus générique car elle prend en paramètre une fonction de comparaison. C'est une fonction curryfiée `f x y` à deux arguments et à valeurs dans les entiers. La liste renvoyée est telle que $f(x, y) \leq 0$ si x est placé avant y. Par exemple pour trier une liste d'entiers (ou de flottants) suivant l'ordre usuel :

```
# let f = fun x y -> if x = y then 0 else if x<y then -1 else 1 ;;
val f : 'a -> 'a -> int = <fun>
# List.sort f [5; 2; 3; 7; 59; 2; 8 ; 489; 44; 498; 11566; 16] ;;
- : int list = [2; 2; 3; 5; 7; 8; 16; 44; 59; 489; 498; 11566]
```

On obtient le même résultat avec la fonction $(x, y) \mapsto y - x$.

Chapitre 4

Analyse des fonctions récursives

4.1 Introduction

On rappelle le code Caml de la fonction donnant la factorielle :

```
let rec fact n=match n with
  | 0 -> 1
  | _ -> n* fact (n-1)
;;
```

Cette fonction est très proche de la définition mathématique associée et il est facile de voir qu'elle termine pour tout entier positif passé en argument. En fait, elle termine car *il n'existe pas de suite strictement décroissante dans* $\mathbb{N}$, et elle est correcte par principe de récurrence : le *cas terminal* est correct car $0! = 1$, et ensuite une récurrence immédiate sur $\mathbb{N}$ prouve que la fonction calcule bien $n!$. En ce qui concerne sa complexité (arithmétique), elle vérifie une relation de récurrence de la forme $C(n) = C(n-1) + O(1)$, dont la solution est $C(n) = O(n)$.

Sur cet exemple simpliste sont résumés les trois principes de l'analyse des fonctions récursives :

— la terminaison se montre en exhibant une quantité, fonction des paramètres de la fonction récursive, et dont les valeurs décroîssent à chaque appel récursif. Ces valeurs sont dans un ensemble muni d'un *ordre bien fondé*, pas nécessairement $\mathbb{N}$.

— la correction, en général assez immédiate, repose sur un principe proche du principe de récurrence ;

— la complexité s'étudie en évaluant la formule récurrente donnée par le coût du traitement dans le corps de la fonction auquel s'ajoute le coût des appels récursifs.

4.2 Terminaison

Définition 4.1. *Une relation d'ordre $\preceq$ sur un ensemble E est une relation :*

— *réflexive : pour tout $x \in E$, $x \preceq x$.*

— *transitive : pour tout $x, y, z \in E$, si $x \preceq y$ et $y \preceq z$, alors $x \preceq z$.*

— *antisymétrique : pour tout $x, y \in E$, si $x \preceq y$ et $y \preceq x$ alors $x = y$.*

Dans la définition précédente, si pour toute paire x, y d'éléments, on a soit $x \preceq y$ soit $y \preceq x$, alors la relation d'ordre est dite totale.

Définition 4.2. *Un élément minimal de E est un élément x tel que $y \preceq x \Rightarrow x = y$. Un plus petit élément de E (nécessairement unique) est un élément x tel que $x \preceq y$ pour tout y de E.*

Définition 4.3. *Un ensemble ordonné $(E, \preceq)$ est dit bien fondé (on dit aussi que c'est l'ordre qui est bien fondé) s'il n'existe pas de suite de cet ensemble strictement décroisante (pour $\preceq$). Si de plus, l'ordre est total, E est dit* bien ordonné.

Exemple 4.4. *L'ensemble $(\mathbb{N}, \leq)$ est bien ordonné. L'ensemble $(\mathbb{Q}_+, \leq)$ ne l'est pas car la suite $\left(\frac{1}{n}\right)_{n \in \mathbb{N}^*}$ est strictement décroissante.*

Exemple 4.5. *Passons à quelques exemples d'ordres bien fondés sur $\mathbb{N}^2$. Ils se généralisent à $\mathbb{N}^p$ pour tout $p \geq 3$ et les deux premiers à des produits cartésiens d'ensembles bien fondés.*

— *L'ordre produit sur $\mathbb{N}^2$, défini par*

$$(a,b) \preceq (c,d) \qquad \iff \qquad a \leq c \qquad et \qquad b \leq d$$

est bien fondé. Attention : ce n'est pas un ordre total.

— *L'ordre lexicographique sur $\mathbb{N}^2$, défini par*

$$(a,b) \preceq (c,d) \qquad \iff \qquad a < c \qquad ou \qquad a = c \quad et \quad b \leq d$$

est bien fondé. En effet, partant d'un couple (a,b), il n'existe que b couples de la forme (a,x) et $x < b$, donc une suite strictement décroissante de longueur au moins $b+2$ atteint un couple (c,d) avec $c < a$. On conclut car $\mathbb{N}$ est lui-même bien fondé. C'est un ordre total.

— *L'ordre lexicographique gradué sur $\mathbb{N}^2$, lui aussi total est défini par :*

$$(a,b) \preceq (c,d) \qquad \iff \qquad a+c < b+d \qquad ou \qquad a+b = c+d \qquad et \qquad a \leq c$$

Il est également bien fondé.

Proposition 4.6. *Un ensemble ordonné E est bien ordonné si et seulement si toute partie non vide admet un plus petit élément.*

Démonstration.
- Montrons le sens direct. On se donne une partie A non vide de E, et on veut montrer qu'elle admet un plus petit élément. Soit $x_0 \in A$. Si x_0 n'est pas minimal, il existe $x_1 \prec x_0$. On réitère le procédé pour construire une suite strictement décroissante $x_k \prec x_{k-1} \prec x_{k-2} \prec \cdots \prec x_0$: le procédé s'arrête car il n'existe pas de suite infinie décroissante dans un ensemble bien ordonné : on a donc trouvé un élément minimal.
- Réciproquement, l'ordre sur E est nécessairement total (sinon il existerait une partie à deux éléments sans plus petit élément). Si $(x_n)_{n\in\mathbb{N}}$ est une suite de l'ensemble, alors $\{x_n \mid n \in \mathbb{N}\}$ possède un plus petit élément, donc la suite n'est pas strictement décroissante et l'ensemble est bien ordonné.

□

Passons au principe d'induction, qui est à un ensemble bien fondé ce que la récurrence (forte) est à $\mathbb{N}$.

Définition 4.7. *On appelle prédicat sur un ensemble E une application de E dans l'ensemble des booléens $\{Vrai, Faux\}$.*

Théorème 4.8. *Soit $(E, \preceq)$ un ensemble bien fondé, notons $\mathcal{M}$ l'ensemble de ses éléments minimaux. Si l'application $\mathcal{P}$ vérifie*

- $\forall x \in \mathcal{M}, \quad \mathcal{P}(x)$,
- $\forall x \in E \backslash \mathcal{M}, \quad (\forall y \prec x, \mathcal{P}(y)) \Rightarrow \mathcal{P}(x)$

Alors, pour tout $x \in E$, $\mathcal{P}(x)$.

Démonstration. Par l'absurde, supposons qu'il existe $x_0 \in E$ tel que $\mathcal{P}(x_0)$ soit faux. Alors il existe un élément x_1, nécessairement pas dans $\mathcal{M}$, tel que $x_1 \prec x_0$ et tel que $\mathcal{P}(x_1)$ soit faux. On recommence avec x_1, et on construit ainsi une suite infinie décroissante, en contradiction avec le caractère bien fondé de E. □

Voyons maintenant comment montrer la terminaison d'une fonction récursive f.

Théorème 4.9. *Soit φ une application de l'ensemble des arguments $\mathcal{A}$ de la fonction f vers un ensemble bien fondé E. On peut supposer φ surjective quite à considérer $\varphi(E)$ à la place de E. Supposons que*

— *la fonction f termine pour tous les x dans l'ensemble $\mathcal{M}_\mathcal{A} = \{x \mid \varphi(x) \in \mathcal{M}\}$ où $\mathcal{M}$ est l'ensemble des éléments minimaux de E ;*

— *pour tout x dans $\mathcal{A}\backslash\mathcal{M}_\mathcal{A}$, le calcul de $f(x)$ ne requiert qu'un nombre fini (éventuellement aucun) d'appels à f, sur des arguments y vérifiant $\varphi(y) \prec \varphi(x)$, et la terminaison de ces appels entraîne celle de $f(x)$.*

Alors, la fonction f termine sur tout argument de $\mathcal{A}$.

Démonstration. Il suffit d'appliquer le théorème précédent à la propriété sur E : $\mathcal{P}(z)$: « les appels $f(x)$ avec $\varphi(x) = z$ terminent. » □

Définition 4.10. *Les éléments x de $\mathcal{A}$ pour lesquels le calcul de $f(x)$ ne nécessite aucun appel à f sont appelés cas de base ou cas terminaux (les éléments de $\mathcal{M}_\mathcal{A}$ sont terminaux, mais ce ne sont pas forcément les seuls).*

Remarque 4.11. *On aura souvent des arguments dans un ensemble bien fondé, et la fonction φ sera l'identité où la projection sur une composantes. Pour des structures plus complexes, l'application φ est souvent un indicateur à valeurs dans $\mathbb{N}$ comme la taille d'une liste ou d'un tableau...*

Exemple 4.12. *La fonction* `fact` *ci-dessus termine.*

Exemple 4.13. *La fonction ci-dessous donnant la longueur d'une liste termine.*

```
let rec longueur l=match l with
  | [] -> 0
  | _::q -> 1+longueur q
;;
```

On prend en effet comme application φ celle qui à une liste associe sa longueur.

Prenons quelques exemples légèrement plus complexe :

Exemple 4.14. *Le calcul des coefficients binomiaux par la fonction suivante termine :*

```
let rec binome n p=match (n,p) with
  | _,0 -> 1
  | 0,_ -> 0
  | _ -> (n*binome (n-1) (p-1))/p
```

En effet, on peut prendre pour ensemble E l'ensemble $\mathbb{N}^2$ muni d'un des ordres vus plus haut : ils conviennent tous. On peut aussi prendre pour E l'ensemble $\mathbb{N}$ avec la fonction $\varphi : \mathbb{N}^2 \to \mathbb{N}$ qui a un couple associe la somme des éléments du couple. Les mêmes ordres fonctionne pour la version suivante, bien moins efficace :

```
let rec binome n p=match (n,p) with
  | _,0 -> 1
  | 0,_ -> 0
  | _ -> binome (n-1) p + binome (n-1) (p-1)
```

Exemple 4.15. *La fonction d'Ackermann suivante termine :*

```
let rec ack n p=match (n,p) with
  | 0,_ -> p+1
  | _,0 -> ack (n-1) 1
  | _ -> ack (n-1) (ack n (p-1))
;;
```

En effet, on prend ici $\mathbb{N}^2$ muni de l'ordre lexicographique.

- *Les cas de bases sont les couples dont la première coordonnée est nulle.*
- *Sur $(n, 0)$, la fonction se rappelle une seule fois avec un couple strictement plus petit.*
- *Sur (n, p) avec n et p tous deux non nuls, les appels récursifs sont sur les couples suivants :*

$$(n, p-1) \qquad \text{et} \qquad (n-1, \texttt{ack}(n, p-1))$$

  qui sont bien plus petits.

Remarquez que la fonction d'Ackermann croît très très vite : on pourra à titre d'exercice calculer les `ack i j` *pour $i \leq 3$, et vérifier que* `ack 4 0` *vaut 13,* `ack 4 1` *vaut 65533 et* `ack 4 2` *vaut $2^{65536} - 3$... La terminaison n'implique pas que le calcul effectif soit possible !*

Exemple 4.16. *La fonction de Morris suivante ne termine pas.*

```
let rec morris n p=match (n,p) with
  | 0,_ -> 1
  | _ -> morris n (p-1) + morris (n-1) (morris n p) ;;
```

En effet, `morris n p` *fait appel à* `morris n p`.

Pour conclure, précisons qu'il n'est pas toujours facile de trouver un « bon ensemble » bien fondé et une application φ associée. La terminaison de la fonction suivante est un problème ouvert.

```
let rec syracuse n=match n with
  | 0 | 1 -> 1
  | n when n mod 2 =0 -> syracuse (n/2)
  | _ -> syracuse (3*n+1)
;;
```

4.3 Correction

Montrer la correction d'une fonction signifie montrer qu'elle calcule ce qu'elle est sensée calculer. Tout le préambule mathématique introduit permet de répondre facilement à cette question, dans le même style que la terminaison, avec la propriété suivante. On reprend la fonction φ du théorème sur la terminaison.

Théorème 4.17. *Considérons sur l'ensemble E le prédicat suivant : $\mathcal{P}(z)$: « les $f(x)$ pour $\varphi(x) = z$ ont la bonne valeur ». Supposons que*

- $\forall z \in \mathcal{M}$, $\mathcal{P}(z)$.
- *pour tout x dans $\mathcal{A}\backslash\mathcal{M}_{\mathcal{A}}$, le calcul de $f(x)$ ne requiert qu'un nombre fini d'appels d'arguments $(y_i)_{1\leq i\leq N}$ qui vérifient $\varphi(y_i) \prec \varphi(x)$ et*
$$(\forall i \leq N, \mathcal{P}(\varphi(y_i))) \Longrightarrow \mathcal{P}(\varphi(x))$$
Alors pour tout $x \in \mathcal{A}$, $f(x)$ est correct.

Démonstration. Immédiate. □

En général, prouver la correction d'une fonction récursive sera relativement immédiat. Par exemple, il est facile de voir que la `syracuse` renvoie 1 pour tout entier naturel *si jamais elle termine*. Prenons un autre exemple, le tri par sélection sur les listes. On commence par écrire une fonction prenant en entrée une liste non vide `q`, et renvoyant un couple formé du plus petit élément de `q` et de la liste de ses autres éléments, dans un ordre arbitraire.

```
let mini q=
 let rec aux m reste p=match p with
   | [] -> m, reste
   | x::r when m<=x -> aux m (x::reste) r
   | x::r -> aux x (m::reste) r
 in aux (List.hd q) [] (List.tl q)
;;
```

La terminaison de la fonction `aux` se montre en considérant pour φ la taille de la liste `p`, la correction de `aux` se fait en considérant la propriété suivante sur $\mathbb{N}$: $\mathcal{P}(z)$: « si `p` est une liste de taille z, et si `m` est plus petit que les éléments de `reste`, alors `aux m reste p` renvoie le couple constitué du minimum parmi les éléments de `m::p@reste`, et d'une liste formé des mêmes éléments, moins `m`. » L'appel de `mini` à `aux` montre qu'elle a bien l'effet escompté. Passons au tri par sélection :

```
let rec tri_selection q=match q with
  | [] -> []
  | _ -> let m,p=mini q in m::(tri_selection p)
;;
```

On prend ici pour fonction φ la taille de la liste `q`, et pour propriété la correction du tri sur les listes d'une certaine taille.

4.4 Complexité des fonctions récursives

4.4.1 Introduction

La complexité d'une fonction récursive vérifie une relation de récurrence de la forme $C(n) = \sum C(n_i) + f(n)$, qu'il va falloir résoudre. Les $C(n_i)$ résulte des appels récursifs, le terme $f(n)$ étant le coût de la fonction hors appels récursifs. Par exemple sur les fonctions déja vues :

Exemple 4.18.
- *factorielle : $C(n) = C(n-1) + O(1)$;*
- *tri par sélection récursif : $C(n) = C(n-1) + O(n)$;*
- *tri fusion : $C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + O(n)$;*
- *Hanoi : $C(n) = 2C(n-1) + O(1)$.*

Souvent, le O pourra être précisé en un Θ (c'est le cas pour toutes ces fonctions), et le résultat aussi : les propositions et théorèmes qui suivent sont valables en remplaçant O par Θ. Dans la suite, on introduit les outils permettant de résoudre de telles récurrences.

Avertissement. Dans les récurrences précédentes, il est crucial que les constantes multiplicatives cachées dans le O ne dépendent pas de n ! C'est le cas dans les algorithmes pris en exemple, et ce sera toujours le cas lorsqu'on étudiera la complexité en informatique en général, si bien qu'on ne le précise pas.

4.4.2 Premiers résultats

Les résultats de cette sous-section permettent de résoudre des « récurrences simples » comme celles satisfaites par le tri par sélection ou la fonction résolvant les tours de Hanoï. On verra comment traiter des récurrences comme celle issue du tri fusion dans la sous-section suivante. Le premier résultat qui suit permet de remplacer un terme $f(n)$ par un terme plus simple.

Proposition 4.19. *Soit $(b_n), (b'_n)$ deux suites réelles positives. Si $b_n = O(b'_n)$ alors $\sum_{k=0}^n b_k = O\left(\sum_{k=0}^n b'_k\right)$.*

Démonstration. — Si $(\sum_{k=0}^n b'_k)_n$ est bornée, alors $(\sum_{k=0}^n b'_k)_n = \Theta(1)$ (car la suite est non nulle à partir d'un certain rang), et la majoration assure que $(\sum_{k=0}^n b_k)_n$ est bornée également, d'où le résultat.

— Sinon, $(\sum_{k=0}^n b'_k)_n$ tend vers $+\infty$. Soit $C > 0$ et n_0 tels que $b_n \leq Cb'_n$ à partir du rang n_0. Alors c'est aussi le cas pour la somme : $\sum_{k=n_0}^n b_k \leq C \sum_{k=n_0}^n b'_k$. Puisque la somme de droite est divergente, elle est non nulle à partir d'un certain rang n_1, qu'on peut supposer être supérieur à n_0. Alors pour $n \geq n_1$:

$$\frac{\sum_{k=0}^n b_k}{\sum_{k=0}^n b'_k} \leq \frac{\sum_{k=0}^{n_1-1} b_k}{\sum_{k=n_1}^n b'_k} + \frac{\sum_{k=n_1}^n b_k}{\sum_{k=n_1}^n b'_k} \leq C + \frac{\sum_{k=0}^{n_1-1} b_k}{\sum_{k=n_1}^n b'_k}$$

et le quotient restant tend vers 0 en $+\infty$. On a bien $\sum_{k=0}^n b_k = O\left(\sum_{k=0}^n b'_k\right)$.

□

Proposition 4.20 (Sommations classiques)**.** *Soient $\alpha > 0$, $q > 1$. Alors $\sum_{k=0}^n k^\alpha = \Theta(n^{\alpha+1})$, $\sum_{k=0}^n q^k = \Theta(q^n)$.*

Démonstration.
- $\sum_{k=0}^n q^k = \frac{q^{n+1}-1}{q-1} = \Theta(q^n)$.
- Pour l'autre relation, la fonction $t \mapsto t^\alpha$ est une fonction croissante, ainsi

$$\frac{n^{\alpha+1}}{\alpha+1} = \int_0^n t^\alpha \,\mathrm{d}t \leq \sum_{k=1}^n k^\alpha \leq \int_1^{n+1} t^\alpha \,\mathrm{d}t = \frac{(n+1)^{\alpha+1}-1}{\alpha+1}$$

Les deux termes à gauche et à droite sont des $\Theta(n^{\alpha+1})$, donc $\sum_{k=0}^n k^\alpha$ aussi.

□

Exemple 4.21. *La solution de la récurrence $C(n) = C(n-1) + O(n)$ est $C(n) = O(n^2)$. En effet on a alors $C(n) = O\left(\sum_{k=0}^n k\right) = O(n^2)$.*

Théorème 4.22. *Soit $(u_n)_{n\geq 0}$ vérifiant pour $n > 0$ la relation $u_n = au_{n-1} + b_n$, avec (b_n) une suite strictement positive, $a > 0$, $u_0 \geq 0$. Si $b_n = O(b^n)$, on a suivant les cas :*

— *si $b < a$, alors $u_n = O(a^n)$;*

— *si $b = a$, alors $u_n = O(na^n)$;*

— *si $b > a$, alors $u_n = O(b^n)$.*

(Le résultat est vrai en remplaçant les O par des Θ).

Démonstration. Posons $v_n = \frac{u_n}{a^n}$. Ainsi $v_n = v_{n-1} + \frac{b_n}{a^n}$ pour tout $n \geq 0$. On a donc pour $n \geq 0$:

$$v_n = v_0 + \sum_{k=1}^{n-1} (v_k - v_{k-1}) = v_0 + \sum_{k=1}^{n-1} \frac{b_k}{a^k}$$

et donc $u_n = a^n \left(v_0 + \sum_{k=1}^{n-1} \frac{b_k}{a^k}\right) = O\left(a^n \left(v_0 + \sum_{k=1}^{n-1} \frac{b^k}{a^k}\right)\right)$ avec $b_k = O(b^k)$. Or la sommation classique des suites géométriques donne :

$$\sum_{k=1}^{n-1} \left(\frac{b}{a}\right)^k = \begin{cases} O(1) & \text{si } b < a \\ O(n) & \text{si } b = a \\ O\left(\left(\frac{b}{a}\right)^n\right) & \text{si } b > a \end{cases}$$

Et le résultat s'ensuit, d'après la proposition 4.19. □

Remarque 4.23. *Si $b_k = O(k^\alpha)$ avec α quelconque et $a > 1$, la première conclusion est valable, la démonstration étant la même.*

Exemple 4.24. *La complexité dans l'algorithme résolvant le problème de Hanoï vérifie $C(n) = 2C(n-1) + \Theta(1)$, donc $C(n) = \Theta(2^n)$, ce qu'on avait déja vu directement.*

4.4.3 Récurrences « Diviser pour régner »

On s'intéresse aux récurrences de la forme

$$u_n = au_{\lfloor \frac{n}{2} \rfloor} + bu_{\lceil \frac{n}{2} \rceil} + b_n$$

La complexité dans le tri fusion est un cas particulier : $a = b = 1$ et $b_n = \Theta(n)$.

Proposition 4.25. *Soit $(u_n)_{n\in\mathbb{N}^*}$ une suite vérifiant la relation de récurrence pour $n > 1$:*

$$u_n = au_{\lfloor \frac{n}{2} \rfloor} + bu_{\lceil \frac{n}{2} \rceil} + b_n$$

avec $a, b \geq 0$, entiers non tous deux nuls, $b_n > 0$. Si $b'_n = O(b_n)$, alors la suite définie par $u'_n = au'_{\lfloor \frac{n}{2} \rfloor} + bu'_{\lceil \frac{n}{2} \rceil} + b'_n$ et $u'_1 = u_1$ vérifie $u'_n = O(u_n)$.

Démonstration. Si $b'_n \leq Cb_n$, avec $C > 1$, alors une récurrence immédiate montre que $u'_n \leq Cu_n$. □

Remarque 4.26. *En supposant $b'_n > 0$, on obtient un résultat valable en remplaçant les O par des Θ, par symétrie.*

Proposition 4.27. *On reprend les mêmes notations, et on suppose (b_n) croissante. Alors (u_n) est croissante.*

Démonstration. Elle se fait par récurrence :

— $u_2 \geq u_1$, car $(a + b) \geq 1$;

— pour $n > 2$, on a

$$u_n = au_{\lfloor \frac{n}{2} \rfloor} + bu_{\lceil \frac{n}{2} \rceil} + b_n \geq au_{\lfloor \frac{n-1}{2} \rfloor} + bu_{\lceil \frac{n-1}{2} \rceil} + b_{n-1} = u_{n-1}$$

donc $u_n \geq u_{n-1}$ et (u_n) est croissante.

□

Théorème 4.28. *Considérons une récurrence de la forme :*

$$u_n = au_{\lfloor \frac{n}{2} \rfloor} + bu_{\lceil \frac{n}{2} \rceil} + b_n$$

On suppose a, b entiers positifs non tous deux nuls, $u_1 \geq 0$. On pose $\alpha = \log_2(a + b)$. Alors, si $(b_n)_{n\geq 1}$ est une suite strictement positive vérifiant $b_n = O(n^\beta)$, on a disjonction suivant les valeurs de β.

— *$\beta < \alpha$, $u_n = O(n^\alpha)$.*

— *$\beta = \alpha$, $u_n = O(n^\alpha \log(n))$.*

— *$\beta > \alpha$, $u_n = O(n^\beta)$.*

Démonstration. On se ramène donc à la même récurrence, mais avec $b_n = n^\beta$ à la place de $O(n^\beta)$, ce qui est légitime d'après la proposition 4.25. Remarquons que la récurrence est plus simple si $n = 2^p$ est une puissance de 2 : $u_{2^p} = (a+b)u_{2^{p-1}}+b_{2^p}$. Étudions d'abord la suite $v_p = (a+b)v_{p-1}+2^{\beta p}$. D'après le théorème 4.22, on a donc le comportement :

— si $2^\beta < (a + b)$, alors $v_p = O((a + b)^p)$

— si $2^\beta = (a + b)$, alors $v_p = O((a + b)^p p)$

— si $2^\beta > (a + b)$, alors $v_p = O(2^{p\beta})$

Or $p = \log_2(n)$ donc $2^{p\beta} = n^\beta$et $(a + b)^p = 2^{p\log_2(a+b)} = 2^{\log_2(n)\log_2(a+b)} = n^{\log_2(a+b)} = n^\alpha$. La conclusion est donc exacte si p est une puissance de 2. Ensuite, il suffit de voir que $(n^\alpha)_n$ est croissante, donc d'après la proposition 4.27, $(u_n)_n$ l'est aussi. On peut alors encadrer n par deux puissances de deux successives (ce qui fait sortir un facteur constant) pour aboutir à la même conclusion, pour n quelconque. □

Remarque 4.29. *Expliquons le résultat obtenu en traçant* l'arbre d'appels récursifs *effectués. En figure 4.1 est représenté celui du tri fusion, satisfaisant la relation $C(n) = C(\lfloor n/2 \rfloor)+C(\lceil n/2 \rceil)+O(n)$. On notera qu'ici $a+b = 2 = 2^\beta$: on se trouve dans le cas $2^\beta = a + b$. Dans ce cas, l'arbre a une hauteur de $\log_2(n)$, et chaque niveau contribue à la complexité totale de manière équilibrée, pour un coût $O(n)$. Discutons des trois cas du théorème :*

— *si $2^\beta < (a + b)$, alors c'est le bas de l'arbre qui contribue le plus à la complexité : le nombre d'appels récursifs effectué fait que le coût hors appels récursifs est petit.*

— *si $2^\beta > (a + b)$ à l'inverse, c'est le haut de l'arbre qui contribue le plus à la complexité : le coût hors appels récursifs est élevé et l'emporte sur ceux-ci.*

— *si $2^\beta = a + b$ comme ici, les $O(\log n)$ niveaux de l'arbres contribuent de manière équilibrée, chacun pour un coût $O(n^\alpha)$, d'où la complexité $O(n^\alpha \log n)$.*

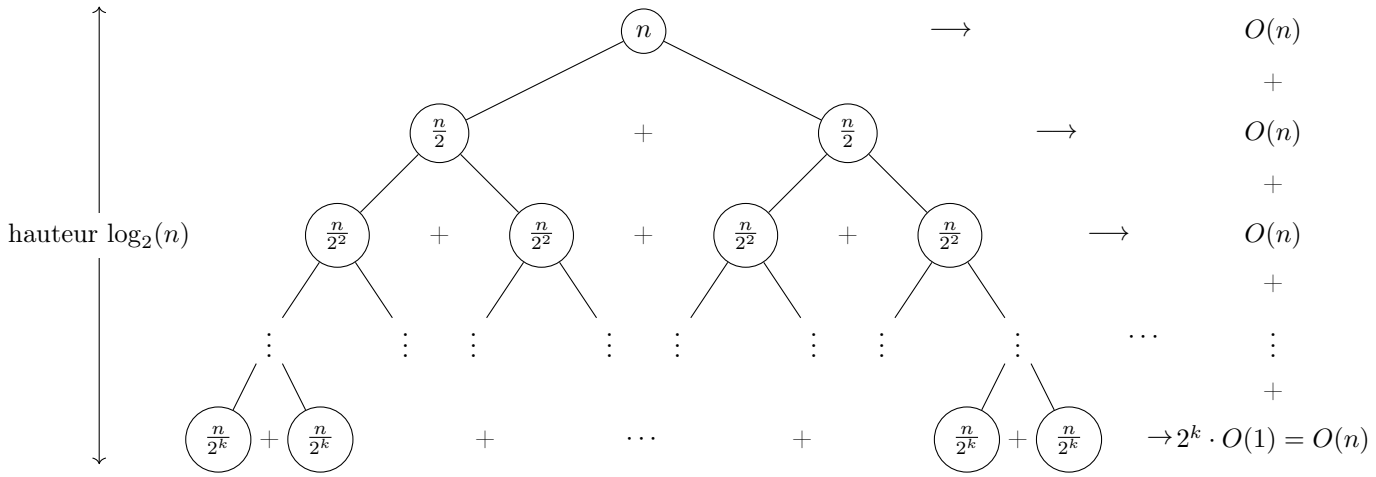


FIGURE 4.1 – Complexité dans le tri fusion sur une liste de taille $n = 2^k$.

Chapitre 5

Algorithmes « Diviser pour régner »

5.1 Introduction

Le chapitre précédent nous a donné des outils pour montrer qu'une fonction récursive termine, est correcte, et estimer sa complexité. Dans le chapitre 3, nous avons notamment vu le tri fusion, qui est un bel exemple d'algorithme « diviser pour régner ». Le présent chapitre montre d'autres exemples de résolution de problèmes via des algorithmes de ce type, dont on rappelle le fonctionnement :

— découper l'instance à résoudre en instances plus petites ;

— résoudre récursivement le problème sur les petites instances ;

— reconstituer une solution du problème sur l'instance initiale à partir des solutions précédemment obtenues.

5.2 Tri fusion : analyse

Pour le problème du tri d'une liste de taille $n > 1$, on avait suivi le principe précédent :

— découper la liste à trier en deux de tailles égales à un près via la fonction `fission` ;

— la fusion de deux listes triées via la fonction `fusion` ;

— appeler récursivement le tri sur les deux portions (étape « division ») avant de les fusionner (« règne »).

5.2.1 Rappel : le code du tri

```
let rec fission q=match q with
  | [] | [_] -> q, []
  | x::y::p -> let a,b=fission p in x::a, y::b
;;

let rec fusion p1 p2=match p1, p2 with
  | [],_ -> p2
  | _, [] -> p1
  | x::q1, y::_ when x<=y -> x::(fusion q1 p2)
  | _,x::q2 -> x::(fusion p1 q2)
;;

let rec tri_fusion p=match p with
  | [] | [_] -> p
  | _ -> let a,b=fission p in fusion (tri_fusion a) (tri_fusion b)
;;
```

5.2.2 Analyse des fonctions auxiliaires

Terminaison. La fonction `fission` termine sur des listes de tailles 0 ou 1, et sur une liste de taille $n > 1$, se rappelle sur une liste de taille $n-2$: elle termine. De même, en notant n_1 et n_2 les tailles des listes en paramètre de la fonction `fusion`, celle-ci termine si $n_1 = 0$ ou $n_2 = 0$, et sinon se rappelle récursivement sur deux listes dont la somme des tailles est $n_1 + n_2 - 1$: la quantité « somme des tailles des deux listes » décroît lors de l'appel récursif et la fonction termine.

Correction. Les propriétés « la fonction `fission q` renvoie un couple de listes `(q_1,q_2)` de tailles égales à 1 près, dont les éléments sont ceux de `q` » et « la fonction `fusion p1 p2` renvoie une liste triée dont les éléments sont ceux de `p1` et `p2` si ces listes sont triées » se montre de manière immédiate par récurrence sur les quantités évoquées au paragraphe précédent.

Complexité. En notant $C(n)$ la complexité de `fission`, l'équation $C(n) = C(n-2) + O(1)$ a pour solution $C(n) = O(n)$. De même, avec $n = n_1 + n_2$, la complexité de la fonction `fusion` est solution de $C(n) = C(n-1) + O(1)$, d'où également une complexité $O(n)$ pour cette fonction.

5.2.3 Analyse de la fonction de tri

Terminaison et correction. La fonction `tri_fusion` est correcte sur des listtes de tailles 0 ou 1, et pour $n > 1$ elle se rappelle récursivement sur des listes de tailles $\lfloor n/2 \rfloor < n$ et $\lceil n/2 \rceil$, donc la fonction termine. Elle est correcte par récurrence sur n et correction des fonctions `fusion` et `fission`.

Complexité. L'équation de la complexité de la fonction `tri_fusion` sur une liste de taille $n > 1$ est donc $C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + O(n)$, dont le théorème du chapitre précédent montre que la solution est $C(n) = O(n \log n)$. Refaisons brièvement la démonstration dans ce cas précis, lorsque n est une puissance de 2. n s'écrit alors 2^k, avec $k \geq 1$. On a donc pour tout $i \geq 1$, $C(2^i) = 2C(2^{i-1}) + O(2^i)$. En divisant par 2^i, on obtient

$$\frac{C(2^i)}{2^i} = \frac{C(2^{i-1})}{2^{i-1}} + O(1)$$

$$\text{Ainsi,} \qquad \frac{C(2^k)}{2^k} = \sum_{i=1}^{k} \underbrace{\left[\frac{C(2^i)}{2^i} - \frac{C(2^{i-1})}{2^{i-1}} \right]}_{O(1)} + C(1) = O(k)$$

D'où $C(n) = C(2^k) = O(k2^k) = O(n \log n)$. Cette relation s'étend à n quelconque, comme montré dans le chapitre précédent [1].

5.3 Algorithmes de multiplication rapide : polynômes et matrices

Cette section aborde deux autres exemples, historiquement très importants, d'algorithmes « diviser pour régner ». Le but est de multiplier rapidement deux polynômes ou deux matrices, de même taille.

Pour le premier problème (multiplication de deux polynômes de même taille n), la méthode naïve a une complexité $O(n^2)$, et il a longtemps été conjecturé qu'il n'existait pas de meilleur algorithme. Karatsuba [2] découvrît la méthode présentée ci-dessous en 1960 : celle-ci a une complexité en $O(n^{1.59})$. Historiquement, la question que s'était posée Karatsuba portait sur la multiplication d'entiers réprésentés en binaire. Mais un entier représenté dans une base b comme $N = \sum a_k b^k$ et le polynôme $\sum a_k X^k$ sont deux objets très proches. Néanmoins la multiplication de polynômes est légèrement plus facile car il n'y a pas à gérer les retenues.

De la même manière, on a longtemps cru que la multiplication de deux matrices de taille $n \times n$ ne pouvait se faire avec une complexité meilleure que $O(n^3)$. Strassen [3] découvrît les formules menant à son algorithme de multiplication de complexité $O(n^{2.81})$ en 1969.

Ces deux algorithmes ont eu une importance considérable, et les problèmes asssociés ont fait l'objet d'intenses recherches depuis. La meilleure borne connue pour la multiplication de deux polynômes de taille n est quasi-optimale : $O(n \log n \log\log n)$. Par contre, le problème de la multiplication matricielle résiste toujours, on sait [4] aujourd'hui multiplier deux matrices de taille $n \times n$ avec une complexité $O(n^{2.3728639})$. Il est conjecturé qu'il existe un algorithme de complexité $O(n^{2+\varepsilon})$ pour tout $\varepsilon > 0$, mais la recherche actuelle en est encore très loin !

Enfin, mentionnons pour terminer un résultat établi en mars 2019, par David Harvey et Joris Van Der Hoeven : il est possible de multiplier deux entiers de n bits en temps [5] $O(n \log n)$. L'optimalité de cette borne est encore une conjecture.

1. On peut supposer C croissante, il suffit alors d'encadrer n entre deux puissances de 2 successives.
2. Anatoli Alekseïevitch Karatsouba, mathématicien soviétique puis russe (1937-2008).
3. Volker Strassen, mathématicien allemand né en 1936.
4. La constante cachée dans le O est tellement énorme que l'algorithme associé n'est pas utilisable en pratique. Celui-ci a été publié par François le Gall en 2014.
5. Ce résultat a « résisté » longtemps ! Les dernières publications donnaient des bornes extrêmement proches, de la forme $O(n \log n f(n))$, où f est une fonction de croissance extrêmement lente (bien plus que le log), que les auteurs ont réussi à éliminer.

5.3.1 Multiplication rapide de polynômes : algorithme de Karatsuba

Le problème

Soit n un entier strictement positif. Considérons $P = \sum_{k=0}^{n-1} p_k X^k$ et $Q = \sum_{k=0}^{n-1} q_k X^k$ deux polynômes de même taille n (la taille est égale au degré, plus 1 : c'est le nombre de coefficients). Les coefficients sont dans un anneau commutatif $\mathbb{A}$ quelconque, qu'on pourra considérer comme étant l'ensemble des entiers $\mathbb{Z}$: une opération dans $\mathbb{A}$ (multiplication, addition, soustraction) est considérée comme élémentaire. On représente les polynômes P et Q par les tableaux `[|`$p_0; p_1; \ldots; p_{n-1}$`|]` et `[|`$q_0; q_1; \ldots; q_{n-1}$`|]`. Le produit $P \times Q$ est un polynôme de degré $2n-2$. Comment obtenir efficacement le tableau de ses $2n-1$ coefficients ?

Méthode naïve de résolution

Pour calculer naïvement le produit, on utilise simplement la relation $PQ = \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} p_k q_j X^{k+j}$. On en déduit un algorithme [6] de complexité $O(n^2)$ répondant au problème :

Algorithme 5.1 : Multiplication naïve

Entrée : Deux polynômes P et Q de même taille n, représentés par leurs tableaux de coefficients.
Sortie : Le tableau de taille $2n-1$ associé au produit $P \times Q$.
$T \leftarrow$ un tableau de $2n-1$ zéros;
pour k *entre 0 et* $n-1$ **faire**
  pour j *entre 0 et* $n-1$ **faire**
    $T[j+k] \leftarrow T[j+k] + p_k \times q_j$
retourner T

Algorithme de Karatsuba.

Si $n = 1$ (les deux polynômes sont des constantes), le produit est simplement le produit des deux constantes. Sinon, posons $m = \lfloor \frac{n}{2} \rfloor$, et découpons nos polynômes en 2. On écrit donc $P_0 = \sum_{k=0}^{m-1} p_k X^k$ et $P_1 = \sum_{k=m}^{n-1} p_k X^{k-m}$ de sorte que $P = P_0 + X^m P_1$, et de même $Q = Q_0 + X^m Q_1$. Posons ensuite $T_0 = P_0 Q_0$, $T_1 = P_1 Q_1$ et $T_2 = (P_0 + P_1) \times (Q_0 + Q_1)$.

Comme $P \times Q = (P_0 + X^m P_1)(Q_0 + X^m Q_1) = P_0 Q_0 + X^m (P_1 Q_0 + P_0 Q_0) + X^{2m} P_1 Q_1$, on obtient le produit en combinant les facteurs (T_i) de la façon suivante : $P \times Q = T_0 + X^m (T_2 - T_1 - T_0) + X^{2m} T_1$. Les produits (T_i) sont eux-même calculés récursivement en exploitant cette idée. L'algorithme, en pseudo-code, est le suivant :

Algorithme 5.2 : L'algorithme de Karatsuba

Entrée : Deux polynômes P et Q de même taille n, représentés par leurs tableaux de coefficients.
Sortie : Le tableau de taille $2n-1$ associé au produit $P \times Q$.
si $n = 1$ **alors**
  retourner `[|`$p_0 \times q_0$`|]`
sinon
  $m = \lfloor n/2 \rfloor$;
  Décomposer $P = P_0 + X^m P_1, Q = Q_0 + X^m Q_1$;
  $T_0 =$Karatsuba(P_0, Q_0);
  $T_1 =$Karatsuba(P_1, Q_1);
  $T_2 =$Karatsuba$(P_0 + P_1, Q_0 + Q_1)$;
  retourner $T_0 + X^m (T_2 - T_1 - T_0) + X^{2m} T_1$

Terminaison et correction de l'algorithme de Karatsuba

La terminaison et la correction repose sur les mêmes principes que pour le tri fusion : pour $n = 1$, l'algorithme termine (et est correct !), et pour deux polynômes de taille $n > 1$, algorithme fait 3 appels récursifs :

— P_0 et Q_0 sont de tailles $1 \leq \lfloor n/2 \rfloor < n$;
— P_1 et Q_1, et donc $P_0 + P_1$ et $Q_0 + Q_1$ sont de tailles $1 \leq \lceil n/2 \rceil < n$;

Donc l'algorithme termine, et est correct car le produit PQ est recomposé correctement à l'aide de T_0, T_1 et T_2.

6. On laisse au lecteur le soin de l'implémenter en Caml.

Étude de la complexité de l'algorithme de Karatsuba

Supposons pour simplifier que n est une puissance de 2, donc s'écrit 2^k. Dans chaque appel récursif, les instances ont des tailles divisées exactement par 2. Pour déterminer la complexité globale, il est essentiel d'estimer la complexité des deux étapes diviser et régner. Ici, on suppose qu'on ne compte que les opérations arithmétiques dans l'anneau $\mathbb{A}$ (additions, multiplications, soustractions).

— Pour diviser, il faut créer les tableaux associés aux polyômes P_0 et P_1, et calculer les polynômes $P_0 + P_1$ et $Q_0 + Q_1$. Ceci se fait en temps linéaire en n.

— Pour régner, il faut créer un tableau T de taille $(2n-1)$, et combiner les trois tableaux associées à T_0, T_1 et T_2 pour obtenir le tableau T. Ceci se fait également en temps linéaire en n puisqu'il suffit de parcourir les tableaux T_0, T_1 et T_2 et de modifier un ou deux éléments de T pour chacun des éléments des trois tableaux.

Pour le calcul du produit, on fait trois appels récursifs pour résoudre des problèmes de taille divisée par 2. Ainsi, la complexité $C(n)$ de l'algorithme de Karatsuba satisfait à l'équation : $C(n) = 3 \times C(n/2) + O(n)$ dont la solution est $C(n) = O\left(n^{\log_2(3)}\right)$. Là encore, les résultats du chapitre précédent permettent d'étendre le résultat à n quelconque.

Code Caml

Voici le code. Il faut faire attention, la taille de `p1` est la même que celle de `p0` si n est pair, un de plus si n est impair. De même pour `q0` et `q1`. Ainsi `t1` et `t2` ont la même taille, supérieure à celle de `t0` si n est impair.

```
let rec karatsuba p q=
  let n=Array.length p in
  if n=1 then [|p.(0)*q.(0)|] else begin
    let k=n/2 in
    let p0, p1=Array.sub p 0 k, Array.sub p k (n-k) and q0, q1=Array.sub q 0 k, Array.sub q k (n-k) in
    let t0, t1=karatsuba p0 q0, karatsuba p1 q1 in
    for i=0 to k-1 do
      p1.(i) <- p0.(i) + p1.(i) ;
      q1.(i) <- q0.(i) + q1.(i)
    done ;
    let t2=karatsuba p1 q1 in
    let res=Array.make (2*n-1) 0 in
    for i=0 to Array.length t0 - 1 do
      res.(i) <- res.(i)+t0.(i) ;
      res.(i+k) <- res.(i+k)-t0.(i)
    done ;
    for i=0 to Array.length t1 - 1 do
      res.(i+k) <- res.(i+k)-t1.(i)+t2.(i) ;
      res.(i+2*k) <- res.(i+2*k)+t1.(i)
    done ;
    res
  end
;;
```

En pratique ?

Le tableau suivant montre les temps en secondes. nécessaires en Python pour calculer sur mon ordinateur personnel (Pocket PC de 2012) le produit de deux polynômes P et Q avec de petits coefficients entiers (tirés aléatoirement dans l'intervalle $[\![-1000, 1000]\!]$), de degrés variables, avec un algorithme naïf et avec l'algorithme de Karatsuba.

$\deg(P) = \deg(Q) = n$	100	500	1000	5000	10000	50000
multiplication naïve	0.006	0.16	0.63	15.96	63.3	1692
Karatsuba	0.07	0.15	0.48	7.75	23.2	288

Pour mieux apercevoir les variations, on peut tracer le diagramme log-log de ces temps (voir figure 5.1). Une complexité $C(n) = n^\alpha$ donne une droite car $\log(C(n)) = \alpha \cdot \log(n)$, ce qu'on observe sur le graphe. Une régression linéaire fait apparaître des coefficients directeurs proches des valeurs théoriques 2 et $1.58 \simeq \log_2(3)$.

Pour de petits polynômes, l'algorithme de Karatsuba est sans intérêt, mais il devient assez vite intéressant : il est plus efficace que l'algorithme naïf pour des polynômes de degré au moins 500.

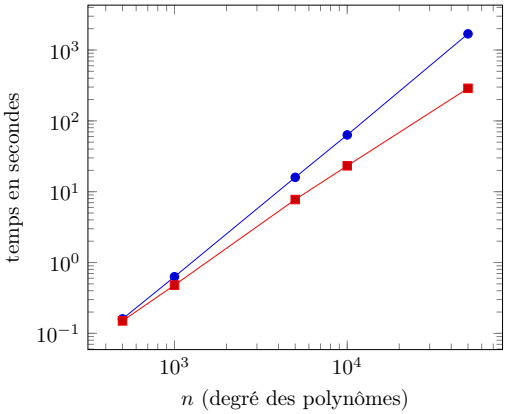


FIGURE 5.1 – Comparaison des temps (algorithme de Karatsuba, multiplication naïve), échelle log-log

5.3.2 Algorithme de Strassen

Le problème

Soit n un entier strictement positif. On se donne $A = (a_{i,j})$ et B deux matrices carrées de même taille $n \times n$, à coefficients dans un anneau commutatif $\mathbb{A}$. Le produit $C = A \times B$ est une matrice de taille $n \times n$, le but est de calculer ses n^2 coefficients efficacement.

Algorithme naïf

En notant $A = (a_{i,j})_{0 \leq i,j < n}$, $B = (b_{i,j})_{0 \leq i,j < n}$ et $C = (c_{i,j})_{0 \leq i,j < n}$, la formule du produit vue en cours de mathématiques $c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$ mène à un algorithme de complexité $O(n^3)$.

Formules de Strassen

Supposons que n soit pair, alors A, B et C se décomposent chacune en 4 blocs de taille $\frac{n}{2} \times \frac{n}{2}$:

$$A = \left(\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array}\right), \quad B = \left(\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \end{array}\right) \quad \text{et} \quad C = \left(\begin{array}{c|c} C_{1,1} & C_{1,2} \\ \hline C_{2,1} & C_{2,2} \end{array}\right)$$

On considère alors les 7 produits suivants :

$$\begin{cases} P_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\ P_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\ P_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\ P_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\ P_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\ P_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\ P_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \end{cases}$$

On vérifie alors que les $C_{i,j}$ s'obtiennent à partir des P_k comme suit :

$$\begin{cases} C_{1,1} &= P_1 + P_4 - P_5 + P_7 \\ C_{1,2} &= P_3 + P_5 \\ C_{2,1} &= P_2 + P_4 \\ C_{2,2} &= P_1 - P_2 + P_3 + P_6 \end{cases}$$

Notons que le simple produit par blocs effectue 8 multiplications : c'est là que se situe le gain en complexité de l'algorithme de Strassen.

Algorithme de Strassen : analyse

L'algorithme de Strassen utilise simplement les formules récursivement, dans le cas $n = 1$ on effectue simplement le produit des deux coefficients. Pour n pair, on obtient donc l'équation de complexité $C(n) = 7C(n/2) + O(n^2)$: en effet, les additions et soustractions de matrices nécessaires avant le calcul des P_i ont un coût $O(n^2)$, de même que le calcul des $C_{i,j}$ à partir des P_k.

La solution de cette récurrence est $C(n) = O(n^{\log_2(7)})$, et $\log_2(7) \simeq 2.81 < 3$. Pour n qui n'est pas une puissance de 2, on peut simplement compléter les matrices avec des zéros pour obtenir une matrice d'une taille une puissance de 2, ce qui ne change pas la complexité asymptotique [7].

Implémentation pratique ?

La constante cachée dans le $O(n^{\log_2(7)})$ étant assez élevée, il faut implémenter finement l'algorithme pour obtenir un gain par rapport à l'algorithme naïf, et ce gain ne s'observe que pour des matrices de tailles déja conséquentes. En pratique, on n'utilise pas les formules de Strassen pour de petites matrices : on se contente de l'algorithme naïf.

5.4 Calcul de la paire de points les plus proches dans un nuage de points

Pour clore ce chapitre, on conclut par un problème géométrique : la recherche du couple de points les plus proches dans un nuage de points. Le problème est le suivant : on se donne un tableau de taille n contenant des couples de flottants, représentant un nuage de points, et on veut identifier les deux points les plus proches, naturellement avec la meilleure complexité possible.

5.4.1 Approche naïve

Il y a $\binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$ couples de points distincts dans le nuage : on peut donc tous les examiner en conservant le couple le plus proche. On obtient donc la solution au problème avec une complexité $O(n^2)$.

5.4.2 Approche « diviser pour régner »

Avant de discuter d'un algorithme efficace, regardons déja ce que l'on peut faire avec comme nuage de points un sous-ensemble de la droite réelle.

Intermède : approche en dimension 1

On sait que l'on peut trier les points d'un nuage réel de taille n en temps $O(n \log n)$ avec le tri fusion. Trouver le couple de points les plus proches est alors très facile car ils seront placés contigüment dans le tableau : un parcours en $O(n)$ suffit. On en déduit donc que le problème à une dimension possède une solution en $O(n \log n)$. De manière assez spectaculaire, on peut obtenir la même complexité pour le problème en dimension 2 qu'en dimension 1, ce qu'on va établir.

Algorithme efficace : partie théorique

On va appliquer une stratégie « diviser pour régner ». Si le nuage a peu de points (disons moins que quatre), on applique l'algorithme naïf. Sinon, on peut séparer le nuage de points en deux parties (presque) égales, autour d'un axe vertical. Remarquons tout de suite qu'il est commode que les points soient triés par abscisse croissante.

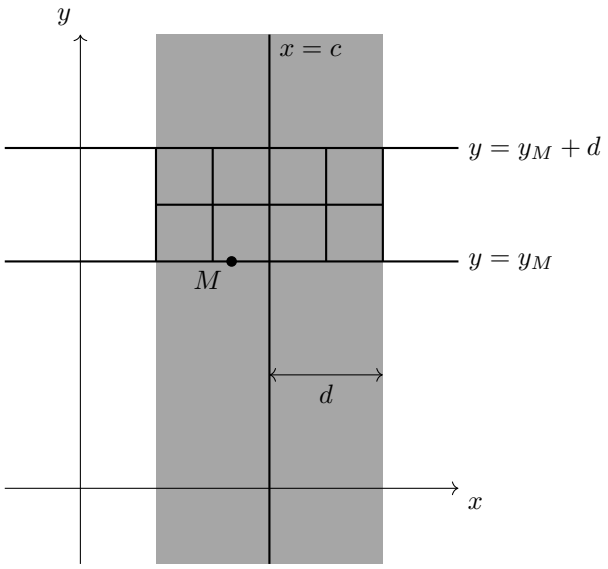
On calcule ensuite (récursivement) la distance minimale et le couple de points correspondant dans les parties gauches et droites. Il faut ensuite calculer la distance entre les points situés de part et d'autre de la droite verticale.

Si on calcule naïvement cette distance, on a environ $n/2$ points à gauche et $n/2$ à droite, on obtient donc une complexité $O(n^2)$, ce qui n'est pas avantageux. Toutefois, on peut optimiser cette partie :

7. En pratique, on se contenterait de rajouter une ligne et une colonne de zéros dans le cas n impair.

— Si d est la distance minimale entre deux points de la partie gauche et deux points de la partie droite, alors deux points du nuage à distance inférieure à d sont situés dans la bande d'abscisse $[c-d, c+d]$, où c est l'abscisse de la droite centrale ;

— Soit $M(x_M, y_M)$ un point de cette bande. Dans le rectangle délimité par $c - d \leq x \leq c + d$ et $y_M \leq y \leq y_M + d$ se trouvent 8 carrés de coté $d/2$ (voir figure ci-contre). La distance maximale entre 2 points d'un petit carré est la longueur de la diagonale : $\sqrt{d^2/4 + d^2/4} = d/\sqrt{2}$. Il ne peut y avoir qu'un seul point du nuage par petit carré au maximum. Donc il suffit de calculer la distance entre M et les sept points suivants, si les points de la bande sont triés par ordonnée croissante.



On va donc pouvoir considérer tous les points M de la bande un par un. Pour chacun, on calcule la distance avec les sept points suivants de la bande (par ordonnée croissante). Ainsi l'analyse de la bande se fait en temps $O(n)$. La complexité totale de l'algorithme est alors :

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + O(n) = O(n \log n)$$

Encore faut-il avoir la structure de donnée adéquate, ce dont on discute dans la sous-section suivante.

Algorithme efficace : idées pour l'implémentation pratique

Résumons les idées de l'algorithme :

— pour séparer les points du nuage autour d'un axe vertical, il est commode que les points soient triés par abscisse croissante ;

— pour examiner les points de la bande dans l'ordre, il est commode que les points soient triés par ordonnée croissante.

Plutôt que de procéder à de multiples tris à chaque appel récursif [8], il vaut mieux introduire de la redondance dans la manière de stocker les points du nuage : on duplique initialement le nuage sous la forme de deux tableaux, l'un trié par abscisse croissante, l'autre par ordonnée croissante [9]. Ce précalcul a un coût $O(n \log n)$ avec le tri fusion, qui ne sera appelé que deux fois en tout.

Notre fonction récursive prendra en entrée ces deux tableaux. A chaque étape, avant les appels récursifs, il faut :

— couper le tableau trié sur l'abscisse en deux ;

— en notant c l'abscisse qui sépare en deux les points du nuage, il faut répartir en deux les éléments du tableau trié sur l'ordonnée (ceci se fait en $O(n)$).

Une fois les appels récursifs effectués et la distance d calculée :

— on extrait les points de la bande $[c-d, c+d]$ à partir du tableau trié sur l'ordonnée ;

— on parcourt la bande en temps $O(n)$.

8. Un examen attentif de la complexité montre que l'on obtiendrait un algorithme qui n'est pas en $O(n \log n)$.

9. On peut supposer les tableaux triés pour les ordres lexicographiques, le premier en regardant d'abord la composante x, et le second la composante y. Ceci permet de gérer aussi les cas « pathologiques », par exemple si tous les points se retrouvent sur une même droite $x = c$. On ne rentre pas dans les détails d'implémentation ici.

Chapitre 6

Introduction aux arbres

6.1 Les arbres comme objets mathématiques

6.1.1 Définitions

Définition 6.1. *Un arbre A est un ensemble non vide muni d'une relation binaire $\prec$ vérifiant :*

— $\exists! r \in A \quad \forall x \in A \quad \neg(r \prec x)$. *L'élément r s'appelle la racine de l'arbre.*

— $\forall x \in A \backslash \{r\} \quad \exists! t \in A \quad x \prec y$. *On dit que y est* le *parent (ou père) de x, et x est* un *fils de y.*

— $\forall x \in A \backslash \{r\} \quad \exists n > 0 \quad \exists (x_1, \dots, x_n) \in A^n \quad x \prec x_1 \prec x_2 \prec \cdots \prec x_n = r$.

La relation binaire $x \prec y$ signifie que x est un enfant de y. Les conditions peuvent se résumer ainsi : mis à part la racine r, chaque élément a un unique parent, et en suivant ces liens de parenté on aboutit à la racine. Visuellement, on représente un arbre avec des nœuds reliés par des arêtes, la racine de l'arbre étant situé tout en haut [1] Par exemple :

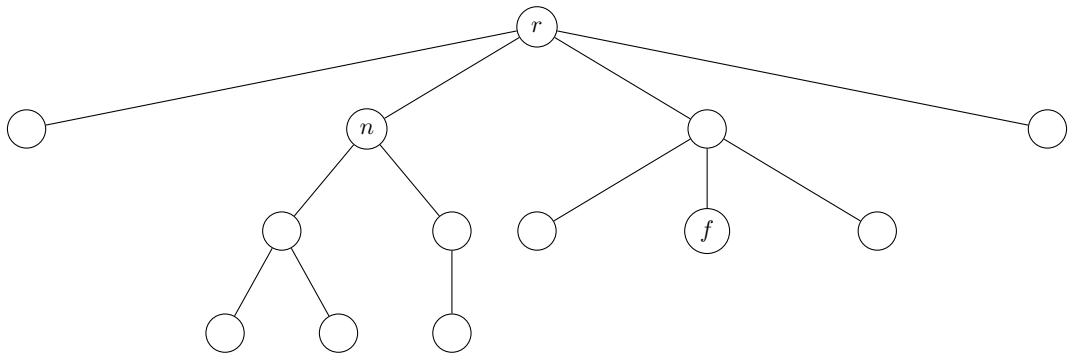


FIGURE 6.1 – Un arbre

Définition 6.2 (feuilles et nœuds internes)**.** *Les éléments de A sont appelés les* nœuds *de l'arbre. Pour x un nœud, on appelle* arité *de x le nombre de fils de x. Un nœud d'arité 0 est appelé une* feuille*, sinon c'est un* nœud interne*.*

Par exemple en figure 6.1, n est un nœud interne, f est une feuille. On étudiera plus particulièrement les arbres évoqués dans la définition suivante.

Définition 6.3 (arbre binaire)**.** *On appelle arbre binaire un arbre dont les nœuds sont d'arité* au plus deux*, et arbre binaire entier un arbre binaire dont les nœuds sont tous d'arité* zéro ou deux*.*

Définition 6.4 (Profondeur et hauteur)**.** *Avec r la racine d'un arbre A et x un de ses nœuds, on a vu qu'il existait un unqiue entier $n \geq 0$ et d'uniques nœuds $x_1, \dots, x_{n-1}$ tels que $x \prec x_1 \prec \cdots \prec x_{n-1} \prec x_n = r$.*

— *On appelle* profondeur *de x l'entier $n \geq 0$;*

— *On appelle* hauteur *d'un arbre la profondeur maximale de ses nœuds.*

Exemple 6.5. *La racine est l'unique nœud à profondeur 0. Les arbres des figures 6.1 et 6.2 ont tous hauteur 3.*

1. En informatique, les arbres poussent de haut en bas.

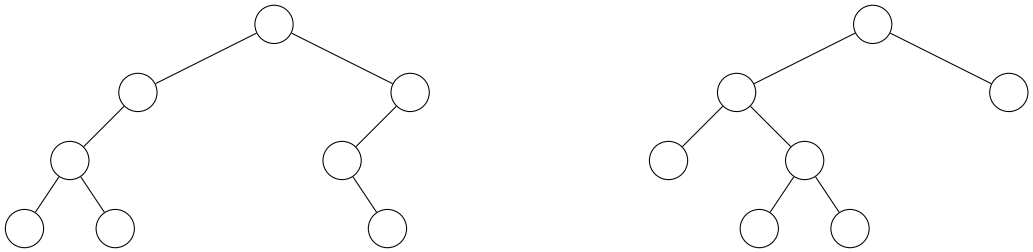


FIGURE 6.2 – Un arbre binaire, et un arbre binaire entier

Définition 6.6 (sous-arbre enraciné)**.** *Soit x un nœud d'un arbre A. On considère l'ensemble*

$$A_x = \{y \in A, \exists n \in \mathbb{N}, \exists x_1, \dots, x_{n-1} \in A \mid y = x_0 \prec x_1 \prec \cdots \prec x_n = x\}$$

Alors on vérifie aisément que la restriction de $\prec$ à A_x munit A_x d'une structure d'arbre, de racine x. Cet arbre se nomme le sous-arbre de A enraciné en x. On dit aussi que les éléments de A_x forment la descendance de x dans A.

6.1.2 Un peu de dénombrement

Inégalités entre hauteur et nombre de nœuds

On donne ici des encadrements faisant intervenir la hauteur et le nombre de nœuds d'un arbre, en fonction de l'arité maximale des nœuds.

Proposition 6.7. *Pour A un arbre de hauteur h dont les nœuds sont d'arité au plus a, le nombre n de nœuds de l'arbre vérifie si $a > 1$:*

$$h + 1 \leq n \leq \frac{a^{h+1} - 1}{a - 1}$$

Démonstration. — Considérons un nœud à profondeur maximale h. Sur le chemin de ce nœud à la racine, il y a $h + 1$ nœuds, d'où $n \geq h + 1$. Avec a l'arité maximale, on montre aisément par récurrence qu'il y a au plus a^p nœuds à profondeur p. L'autre inégalité s'obtient donc par somme : $\sum_{p=0}^{h} a^p = \frac{a^{h+1}-1}{a-1}$.

□

Remarque 6.8. *Si un arbre est d'arité maximale 1, il a exactement $h + 1$ nœuds, avec h sa hauteur.*

Corollaire 6.9. *La hauteur h d'un arbre à n nœuds tous d'arité au plus $a > 1$ vérifie*

$$\log_a((a-1)n + 1) - 1 \leq h \leq n - 1$$

Appliquons ce résultat dans le cas $a = 2$:

Corollaire 6.10. *Soit A un arbre binaire à n nœuds. Sa hauteur h vérifie $\lfloor \log_2(n) \rfloor \leq h \leq n - 1$*

Démonstration. On prend donc $a = 2$ dans le corollaire précédent. Ainsi $h + 1 \geq \log_2(n + 1) > \log_2(n) \geq \lfloor \log_2(n) \rfloor$. Ainsi $h + 1$ est un entier strictement supérieur à l'entier $\lfloor \log_2(n) \rfloor$, donc $h \geq \lfloor \log_2(n) \rfloor$. □

Feuilles et nœuds internes dans un arbre binaire

Proposition 6.11. *Un arbre binaire entier ayant p nœuds internes possède $p + 1$ feuilles.*

Démonstration. La démonstration se fait par récurrence forte (sur p par exemple).

— Si $p = 0$, l'arbre a une seule feuille (sa racine) donc la relation est vérifiée.

— Sinon soit $p > 0$. Considérons un arbre A ayant $p > 0$ nœuds internes. La racine étant un nœud interne, notons alors n_g et n_d le nombre de nœuds internes des sous-arbres enracinés en les deux fils de la racine. Ces arbres sont également binaires entiers, et vérifient $n_g < p$ et $n_d < p$, donc par hypothèse de récurrence, ils ont respectivement $n_g + 1$ et $n_d + 1$ feuilles. Dans l'arbre A, il y a donc $n_g + n_d + 1$ nœuds internes et $(n_g + n_d + 1) + 1$ feuilles, donc la propriété est vraie pour un arbre de hauteur p ;

— Par principe de récurrence, la propriété est démontrée.

□

Corollaire 6.12. *Dans un arbre binaire à p nœuds internes, il y a au plus $p + 1$ feuilles.*

Démonstration. Rajouter un fils (une feuille) aux nœuds d'arité 1 transforme l'arbre en arbre binaire entier, sans changer le nombre de nœuds internes. □

6.2 Les arbres en Caml

En informatique, les arbres sont utilisés pour stocker de l'information : à chaque nœud est attaché une *étiquette*, qui peut être un entier, une chaîne de caractères, voire même un couple (voir chapitre suivant). De plus, les fils d'un nœud sont en général ordonnés : par exemple pour un arbre binaire entier, on parlera du fils gauche et du fils droit d'un nœud interne. On propose dans cette section une implémentation *persistante* des arbres : comme pour les listes chaînées, les fonctions sur la structure d'arbre renverront de nouveaux arbres plutôt que de les modifier (une implémentation impérative sera vue au chapitre suivant).

6.2.1 Arbres généraux

Pour des arbres généraux, on peut représenter un nœud par la liste de ses fils, qui sont eux même des arbres. On distinguera alors une feuille et un nœud interne suivant si la liste des fils est vide ou non. Pour que les nœuds de l'arbre puissent porter des étiquettes d'un certain type, on peut définir le type (récursif) suivant :

```
type 'a arbre = N of 'a * 'a arbre list;;
```

Voici un exemple d'arbre à étiquettes entières :

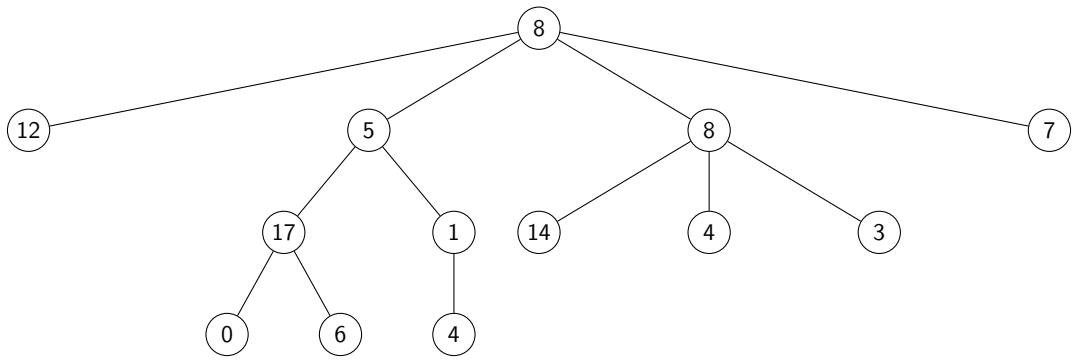


FIGURE 6.3 – Un arbre à étiquettes entières

En Caml cet arbre est implémenté comme suit :

```
#ex_arbre ;;
- : int arbre =
 N (8,
    [N (12, []);
     N (5, [N (17, [N (0, []); N (6, [])]); N (1, [N (4, [])])]);
     N (8, [N (14, []); N (4, []); N (3, [])]);
     N (7, [])])
```

Pour parcourir un tel arbre, on utilise en général deux fonctions : l'une qui prend en entrée un arbre, et l'autre une liste d'arbres. Elles vont s'appeler l'une l'autre et donc être mutuellement récursives. Voici un exemple de parcours, pour calculer la hauteur d'un tel arbre :

```
let rec hauteur a=match a with
  | N(_,q) -> 1+max_h q
and max_h q=match q with
  | [] -> -1
  | x::p -> max (hauteur x) (max_h p)
;;
```

Si on veut faire la distinction entre feuilles et nœuds internes (et leur donner des étiquettes de type différents), on peut utiliser par exemple le type suivant.

```
type ('a, 'b) arbre = F of 'a | N of 'b * ('a, 'b) arbre list;;
```

6.2.2 Arbres binaires entiers

En informatique, les arbres sont souvent des arbres binaires (entiers ou non). On va donc utiliser une implémentation un peu moins générale que celle de la section précédente. Pour les arbres binaires entiers, un arbre (informatique) est :

— soit une feuille ;

— soit la donnée d'une étiquette, et de deux arbres (ses sous-arbres gauche et droit).

En suivant cette description, on obtient le type suivant, où l'on distingue les étiquettes des feuilles et des nœuds internes.

```
type ('a, 'b) arbre =  F of 'a | N of 'b*('a, 'b) arbre * ('a, 'b) arbre;;
```

Voici un exemple d'utilisation : une expression arithmétique se représente naturellement par un arbre binaire entier, les étiquettes des nœuds internes étant les opérateurs et celles des feuilles étant les opérandes, voir figure 6.4.

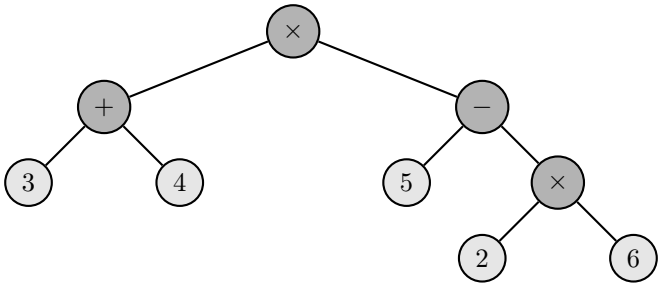


FIGURE 6.4 – L'arbre associé à l'expression arithmétique $(3+4)\times(5-(2\times 6))$.

En Caml on représente l'expression de la figure 6.4 comme :

```
type op = Plus | Moins | Fois | Div | Mod ;;
let expr = N (Fois, N (Plus, F 3, F 4), N (Moins, F 5, N (Fois, F 2, F 6))) ;;
```

On a utilisé un type énuméré pour définir les opérateurs. L'évaluation d'une telle expression se fait récursivement, par filtrage :

```
let rec evalue e=match e with
  | F x -> x
  | N (op, a, b) -> (traduit op) (evalue a) (evalue b)
;;
```

Où la fonction `traduit` renvoie la fonction `int -> int -> int` associée à l'opérateur `op`, qu'on laisse au lecteur le soin d'écrire. Testons :

```
# evalue ;;
- : (int, op) arbre -> int = <fun>
# evalue expr ;;
- : int = -49
```

6.2.3 Arbres binaires

On propose une implémentation des arbres binaires qui sera reprise dans le chapitre suivant, pour l'implémentation des arbres binaires de recherche. Elle est très proche de l'implémentation précédente des arbres binaires entiers : en effet, si on considère un arbre binaire, et qu'en chaque nœud x on fait pousser $2-a$ fils où a est l'arité de x, on obtient un arbre binaire entier. On note « Vide » les nœuds que l'on fait pousser, de sorte que les feuilles du nouvel arbre sont toutes « Vide ».

Avec cette représentation, un arbre binaire est :

— soit vide ;

— soit la donnée d'une étiquette, et deux deux sous-arbres binaires.

Ceci mène à la définition du type suivant :

```
type 'a arbre = Vide | N of 'a * 'a arbre * 'a arbre ;;
```

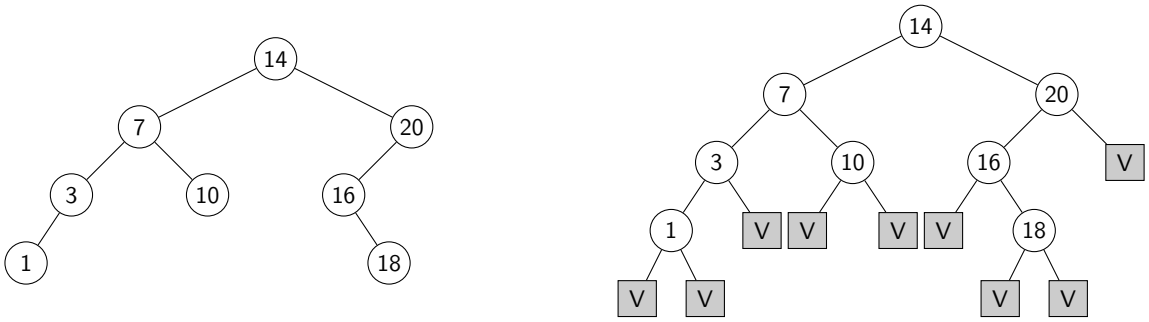


FIGURE 6.5 – Un arbre binaire à étiquettes entières, et l'arbre binaire entier associé.

Voici l'implémentation de l'arbre correspondant à l'exemple de la figure 6.5 :

```
let ex_ab = N (14,
   N(7, N(3, N(1, Vide, Vide), Vide), N(10, Vide, Vide)),
   N(20, N(16, Vide, N(18, Vide, Vide)), Vide)) ;;
```

Donnons pour terminer une fonction permettant de calculer la hauteur [2] d'un tel arbre :

```
let rec hauteur a=match a with
  | Vide -> -1
  | N(_,g,d) -> 1 + max (hauteur g) (hauteur d)
;;
```

Par exemple :

```
#hauteur ex_ab ;;
- : int = 3
```

6.3 Parcours d'arbres binaires entiers

On veut énumérer les nœuds d'un arbre binaire entier, de même type que défini en sous-section 6.2.2. On va stocker le résultat dans une liste. On introduit un type pour stocker dans une même liste les étiquettes des nœuds internes et des feuilles :

```
type ('a, 'b) etiq = A of 'a | B of 'b ;;
```

6.3.1 Parcours en profondeur

Le parcours en profondeur d'un arbre consiste à s'enfoncer le plus possible dans l'arbre avant de revenir en arrière explorer les autres branches. Pour un arbre binaire entier de racine r, et de sous-arbres gauche et droit g et d, il y a trois choix naturels :

— racine r, enumération de g, énumération de d : on parle de parcours préfixe ;

— enumération de g, racine r, énumération de d : on parle de parcours infixe ;

— enumération de g, énumération de d, racine r : on parle de parcours postfixe (ou suffixe).

Voici une écriture du parcours préfixe :

```
let rec prefixe a=match a with
  | F x -> [A x]
  | N(x,g,d) -> (B x)::(prefixe g)@(prefixe d)
;;
```

Testons avec l'expression arithmétique de la figure 6.4.

2. On convienr que l'arbre « Vide » a pour hauteur -1 : ceci est cohérent avec le fait que les feuilles « Vide » ne font pas vraiment partie de l'arbre.

```
# prefixe expr ;;
- : (int, op) etiq list =
[B Fois; B Plus; A 3; A 4; B Moins; A 5; B Fois; A 2; A 6]
```

On peut de même écrire des fonctions `infixe` et `postfixe` :

```
# infixe expr ;;
- : (op, int) etiq list =
[A 3; B Plus; A 4; B Fois; A 5; B Moins; A 2; B Fois; A 6]
# postfixe expr ;;
- : (op, int) etiq list =
[A 3; A 4; B Plus; A 5; A 2; A 6; B Fois; B Moins; B Fois]
```

Il est notable de voir que l'énumération donnée par le parcours préfixe ou le parcours postfixe permet de reconstruire l'arbre, contrairement à celle du parcours infixe.

Proposition 6.13. *Si les étiquettes des nœuds internes et des feuilles ont des types différents, alors à une énumération préfixe ou postfixe correspond un seul arbre binaire entier.*

Voici une preuve dans le cas de l'énumération postfixe, on laisse au lecteur le soin de l'adapter à l'énumération préfixe. Elle débute par un lemme.

Lemme 6.14. *Considérons l'énumération postfixe d'un arbre binaire entier. On parcourt l'énumération avec un compteur s initialisé à 0, on ajoute $+1$ pour une feuille, et -1 pour un nœud interne. Alors s est toujours strictement positif après le début de l'énumération, et termine par* 1.

Démonstration. Le lemme se démontre par récurrence sur la longueur de l'énumération.

— pour une énumération de longueur 1 (correspondant à une unique feuille), c'est immédiat ;

— sinon, l'énumération est constituée de l'énumération du sous-arbre gauche (par récurrence, s est toujours strictement positif et termine à 1), celle du sous-arbre droit (s reste supérieur à 1 et termine à 2), et enfin la racine (s termine à 1).

— Par principe de récurrence, le lemme est démontrée.

□

Preuve de la proposition 6.13. On procède par récurrence sur la longueur de l'énumération.

— Une énumération de taille 1 correspond à un arbre à une unique feuille, l'unicité est donc évidente.

— Donnons nous maintenant une énumération postfixe d'un arbre binaire entier, de taille strictement supérieure à 1, et supposons la propriété démontrée pour des énumérations plus petites. Par nature de l'énumération, se trouve d'abord toute l'énumération du sous-arbre gauche, puis celle du sous-arbre droit, et enfin la racine de l'arbre. Pour pouvoir appliquer l'hypothèse de récurrence et terminer la preuve, il suffit de savoir où se situe la frontière entre les énumérations des sous-arbres gauche et droit. Or, on déduit du lemme que cette frontière se situe juste après que le compteur s du lemme ait pris la valeur 1 pour la dernière fois (avant la racine) : on peut donc reconstruire l'arbre par hypothèse de récurrence.

— Par principe de récurrence, la propriété est démontrée.

□

Remarque 6.15. *Cette propriété sur l'énumération postfixe a été utilisée dans certaines calculatrices*[3]*, et s'étend à des expressions faisant usage d'opérateurs d'arité différente de 2. L'intérêt est que les parenthèses sont inutiles pour donner l'expression arithmétique, ce qui fournit un gain de temps à l'utilisateur. Une pile suffit pour écrire une fonction d'évaluation d'une expression donnée sous la forme du parcours postfixe, qu'on laisse au lecteur le soin d'implémenter :*

```
#evalue_postfixe (postfixe expr) ;;
- : int = -49
```

Remarque 6.16. *L'énumération infixe ne suffit pas pour reconstruire l'arbre, comme le montre l'exemple des deux arbres ci-dessous, ayant même énumération :*

3. À notation *polonaise inversée*, qui est un autre nom pour l'énumération postfixe de l'arbre associé à l'expression.
4. La priorité de $\times$ sur $+$ et $-$ permet de s'affranchir de certaines parenthèses... Mais ce n'est qu'une convention !

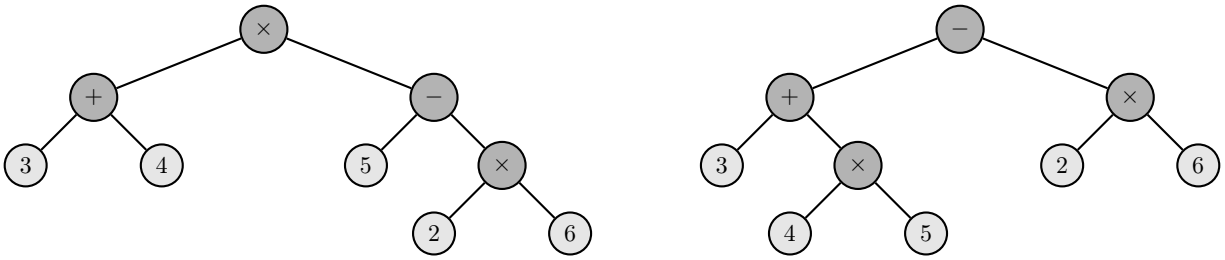


FIGURE 6.6 – Deux arbres d'énumération infixe $3 + 4 \times 5 - 2 \times 6$: les parenthèses sont obligatoires [4]pour donner un sens à l'expression.

6.3.2 Parcours en largeur

Contrairement au parcours en profondeur, le parcours en largeur liste les nœuds par profondeur croissante. Par convention, les nœuds situés à gauche sont énumérés en premier. L'énumération de ce parcours est un peu plus délicate à obtenir qu'une énumération d'un parcours en profondeur. On commence par écrire deux fonctions qui prennent en entrée une liste d'arbres, et renvoient respectivement l'énumération de leurs racines, et la liste de leurs sous-arbres.

```
let rec racines q=match q with
    | [] -> []
    | (F x)::p -> (A x)::(racines p)
    | N(x,_,_)::p -> (B x)::(racines p)
;;

let rec ss_arbres q=match q with
  | [] -> []
  | (F _)::p -> ss_arbres p
  | N(_,g,d)::p -> g::d::ss_arbres p
;;
```

On peut maintenant écrire une fonction de parcours en largeur :

```
let largeur p=
  let rec aux q=match q with
    | [] -> []
    | _ -> (racines q)@(aux (ss_arbres q))
  in aux [p]
;;
```

La fonction `aux` de `largeur` renvoie l'énumération « en largeur » d'une liste d'arbres : si la liste est non vide, elle énumère les racines, et se rappelle récursivement sur la liste des sous-arbres. `largeur` se contente d'un unique appel à `aux`. Testons :

```
#largeur expr ;;
- : (int, op) etiq list =
[B Fois; B Plus; B Moins; A 3; A 4; A 5; B Fois; A 2; A 6]
```

Remarque 6.17. *Les parcours en profondeur préfixe et postfixe, ainsi que le parcours en largeur, se généralisent à d'autres arbres que les arbres binaires entiers, avec des fonctions assez semblables à celles vues dans ce chapitre. Le parcours infixe se généralise à des arbres binaires (non nécessairement binaires entiers), mais pas à des arbres quelconques.*

Chapitre 7

Preuves par induction

7.1 Introduction

Nous avons défini précédemment un arbre, comme un objet mathématique précis : un ensemble muni d'une certaine relation binaire (de parenté). Ceci dit, lors de l'implémentation concrète, il a fallu s'éloigner de cette définition pour ordonner les fils. Une autre possibilité (peut-être plus naturelle) est de donner une définition *inductive* : un arbre est soit réduit à un sommet, soit constitué de sa racine est de la liste (ordonnée !) des sous-arbres de la racine. On pourrait donner des définitions du même style pour les autres types d'arbres un peu plus restreints comme les arbres binaires. Ce genre de définition a le mérite d'être facilement manipulable, à la fois pour prouver des propriétés mathématiques (par exemple, un arbre binaire entier possède une feuille de plus que de nœuds internes) et transposable aisément en une implémentation.

Dans la suite, on utilise la notion de mot sur un alphabet A (fini ou dénombrable), qui sera vue plus tard. Ce dont on a besoin est assez intuitif : un mot sur A est un n-uplet d'éléments de A, qu'on préfère noter $a_1a_2\dots a_n$ plutôt que $(a_1,\dots,a_n)$. L'entier n peut être nul : on note ε le mot vide. Voici deux exemples : *abcaab* est un mot sur $\{a,b,c\}$, et $(4+5)\times 2$ un mot sur $\mathbb{N}\cup\{+,\times,(,)\}$.

7.2 Définitions inductives

Une définition inductive définit une partie d'un certain ensemble E, comme la plus petite contenant un certain sous-ensemble de base, et stable par application de certaines règles de construction. Avant de formaliser, une petite remarque.

Remarque 7.1. *L'ensemble E ne joue pas un rôle prépondérant. Néanmoins, il est nécessaire de supposer son existence, pour éviter de se retrouver coincé par des considérations du type « ensemble de tous les ensembles », qui n'existe pas. Pour l'ensemble E, on prendra souvent l'ensemble des mots sur un certain alphabet, ensemble qui a le mérite d'exister, en confondant les objets avec leurs écritures syntaxiques.*

7.2.1 Le théorème du point fixe

Définition 7.2 (Outils pour la définition inductive)**.** *Soit E un ensemble. On appelle :*

- *ensemble de base un certain sous-ensemble $B\subset E$;*
- *règle une application partielle $r : E^n \to E$, avec n un entier strictement positif, appelé l'arité de r.*

La définition inductive d'une partie de E repose sur le théorème suivant :

Théorème 7.3 (Théorème de point fixe)**.** *Soit E un ensemble. Considérons :*

- *$B\subset E$ un ensemble de base ;*
- *$R=\{r_j \mid j\in J\}$ un ensemble de règles, avec $r_j : E^{n_j}\to E$. Ces règles sont appelées* règles d'inférence.

Alors il existe un plus petit ensemble $X\subset E$, tel que :

- *(B) : $B\subset X$;*
- *(I) : pour tout $j\in J$, pour tout élément $(x_1,\dots,x_{n_j})\in X^{n_j}$ appartenant à l'ensemble de définition de r_j, on a $r_j(x_1,\dots,x_{n_j})\in X$.*

Démonstration. L'ensemble des sous-ensembles de E vérifiant les deux propriétés (B) et (I) est non vide, car il contient E lui-même. On peut donc considérer X, l'intersection de tous ces sous-ensembles. Alors :

— X vérifie (B) ;

— si $(x_1, \dots, x_{n_j})$ est un n_j-uplet d'éléments de X appartenant à l'ensemble de définition de r_j, alors par définition $r_j(x_1, \dots, x_{n_j})$ appartient à l'intersection de tous les sous-ensembles de E vérifiant (B) et (I), donc à X.

X est bien le plus petit sous ensemble de E vérifiant (B) et (I). □

Définition 7.4. *L'ensemble X donné par le théorème précédent s'appelle l'ensemble défini par induction avec l'ensemble de base B et les règles de R.*

Notation. Dans la suite, on notera les définitions d'un ensemble inductif X comme ceci :

— (B) : $B \subset X$;

— (I) : $(x_1, \dots, x_n) \in X^n \Rightarrow r(x_1, \dots, x_n) \in X$, pour chaque règle.

7.2.2 Des exemples

Exemple 7.5 (Entiers naturels pairs)**.** *L'ensemble P des entiers naturels pairs est défini par :*

— $0 \in P$

— $x \in P \Rightarrow x + 2 \in P$

Comme on le voit, l'ensemble E n'a pas un rôle très important, on peut prendre $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{C}$...

Exemple 7.6 (Mots de Dyck)**.** *L'ensemble $\mathcal{D}$ des mots de Dyck sur l'alphabet $\{a, b\}$ est défini par :*

— *le mot vide ε est dans $\mathcal{D}$;*

— *$x, y \in \mathcal{D} \Rightarrow axby \in \mathcal{D}$*

Exemple 7.7 (Listes chaînées)**.** *Les listes chaînées sur un ensemble F sont très semblables à celles sur les entiers naturels :*

— *la liste vide est une liste ;*

— *si x est un élément de F, et ℓ une liste, alors Cons(x, L) est une liste.*

On peut définir plus proprement les listes comme un ensemble de mots sur l'alphabet constitué des éléments de F et de $\{(,), \square\}$, par exemple.

Exemple 7.8 (Arbres binaires)**.** *Soit F un ensemble. Les arbres binaires étiquetés par F sont définis ainsi :*

— *l'arbre vide est un arbre binaire ;*

— *si x est un élément de F, $\mathcal{A}_g$ et $\mathcal{A}_d$ deux arbres binaires, alors Noeud$(\mathcal{A}_g, x, \mathcal{A}_d)$ est un arbre binaire.*

Exemple 7.9 (Arbres quelconques)**.** *L'ensemble des arbres (non étiquetés) peuvent être définis sur l'alphabet $\{(,)\} \cup \{a_0, a_1, a_2, a_3, \dots\}$ par :*

— *a_0 est un arbre.*

— *Pour tout $n > 0$ et pour tout n-uplet $t_1, \dots, t_i$ d'arbres, $a_n(t_1, \dots, t_n)$ est un arbre.*

Visuellement, on représente avec la règle d'inférence un arbre dont la racine possède n sous-arbres qui sont dans l'ordre $t_1, \dots, t_n$. Le nombre de règles étant ici non dénombrable, cette construction théorique est intéressante, mais pas utile pour une implémentation. En voici une autre.

Exemple 7.10 (Arbres et forêts)**.** *Les règles d'inférence sur les arbres et forêts (ensemble d'arbres) étiquetés par des éléments d'un ensemble F sont :*

— *l'arbre vide* `Vide` *est un arbre ;*

— *la foret vide* `[]` *est une forêt ;*

— *si $\mathcal{A}$ est un arbre et $\mathcal{F}$ une forêt, Cons$(\mathcal{A}, \mathcal{F})$ est une forêt.*

— *si $\mathcal{F}$ est une forêt et x un élément de F, Noeud$(x, \mathcal{F})$ est un arbre.*

Exemple 7.11 (Expressions arithmétiques sur les entiers)**.** *On note $\mathcal{N}$ l'ensemble des expressions sur l'alphabet $\{0, \dots, 9\}$ représentant les entiers naturels (essentiellement, à part 0, aucune chaîne ne commence par un 0). Cet ensemble peut être construit inductivement, on laisse au lecteur le soin de décrire la construction. L'ensemble des expressions arithmétiques $\mathcal{A}$ sur $\mathcal{N}$ peut être défini ainsi, sur l'alphabet $\{0, \dots, 9, +, -, \times, /, (,)\}$:*

— *$\mathcal{N} \subset \mathcal{A}$;*

— *pour tout op $\in \{+, \times, -, /\}$, on a $g, h \in \mathcal{A} \Rightarrow g$ op $h \in \mathcal{A}$;*

— *$g \in \mathcal{A} \Rightarrow (g) \in \mathcal{A}$;*

Autres exemples. Les expressions rationnelles ou les formules logiques (vues plus tard) peuvent être définies de la même façon.

7.3 Preuves par induction

Le principe d'une preuve par induction (structurelle) repose sur le théorème suivant :

Théorème 7.12. *Soit $X \subset E$ un ensemble inductif, défini par l'ensemble de base B et les règles d'induction R. On considère un prédicat P sur les éléments de E. Supposons que :*

- *$P(x)$ est vrai pour tout $x \in B$;*
- *P est héréditaire pour les règles de R : pour tout $r \in R$, si $(x_1, \dots, x_n)$ est dans l'ensemble de définition de r, avec n l'arité de r, et si $P(x_i)$ est vrai pour tout $i \in [\![1, n]\!]$, alors $P(r(x_1, \dots, x_n))$ est vrai ;*

alors P est vérifiée sur l'ensemble des éléments de X.

Démonstration. Considérons l'ensemble Y des éléments de E sur lesquels P est vraie. Alors Y contient B et est stable par les règles de R, donc par définition de l'ensemble inductif X, celui-ci est inclus dans Y. □

Remarque 7.13. *Le théorème précédent généralise la preuve par récurrence, car l'ensemble $\mathbb{N}$ peut être décrit par les deux assertions : $0 \in \mathbb{N}$ et $x \in \mathbb{N} \Rightarrow x + 1 \in \mathbb{N}$.*

Exemple 7.14. *L'ensemble des arbres binaires entiers peut-être défini inductivement comme suit (sur l'alphabet $\{N, \emptyset, (,)\}$) :*

- *$\emptyset$ est un arbre binaire entier ;*
- *Pour g et d deux arbres binaires entiers, $N(g, d)$ en est un également.*

Alors en notant n le nombre de N dans l'écriture d'un arbre binaire et f le nombre de $\emptyset$, on a $f = n + 1$. En effet, cette propriété est vérifiée pour $\emptyset$ et stable par application de la règle d'inférence.

7.4 Induction non ambiguë et définitions de fonctions sur un ensemble inductif

7.4.1 Définition non ambiguë d'un ensemble inductif

Revenons sur l'ensemble des expressions arithmétiques sur $\mathbb{N}$ définie un peu plus haut dans l'exemple 7.11. On voit sans peine que $3 - 4 + 5$ où $1 + 2 \times 6$ sont des expressions arithmétiques valides. Dans la suite, on aimerait définir par exemple l'évaluation d'une telle expression, dont le résultat est un élément de $\mathbb{Q}$ (ou de $\mathbb{Q} \cup \{\text{Erreur !}\}$ pour gérer les divisions par zéro). Naturellement, on voudrait définir une fonction f sur un ensemble inductif X en exploitant la définition inductive de l'ensemble. Ainsi, il faudrait pouvoir définir $f(r(x_1, \dots, x_n))$ à partir des $f(x_i)$, où r est une règle d'arité n. On voit ici apparaître un problème pour les expressions arithmétiques que l'on a défini : l'expression $3 - 4 + 5$ peut être *dérivée* à partir de $3 - 4$ et 5, mais aussi de 3 et de $4 + 5$. Qu'importe la valeur que l'on veut donner à $3 - 4 + 5$: celle-ci est incompatible avec les deux dérivations différentes proposées. La définition que l'on a donnée d'une expression arithmétique sur $\mathbb{N}$ est *ambiguë* car il est possible d'obtenir un élément de plusieurs façons. Il est nécessaire de lever toute ambiguïté pour définir une fonction sur un ensemble inductif.

Définition 7.15. *Une définition d'un ensemble inductif X est dite* non ambiguë *si chaque élément de X ne peut s'obtenir que d'une seule façon à partir de B et des règles d'inférences de R.*

Remarque 7.16. *Cette définition est assez peu formelle, bien qu'intuitive. Il est possible de donner une définition beaucoup plus rigoureuse, ce que je ne ferai pas ici. Voici simplement l'idée : la construction des arbres généraux de l'exemple 7.9 est en quelque sorte le prototype d'une définition inductive quelconque, l'application d'une règle d'inférence d'arité n dans une définition inductive pouvant être vue comme la construction d'un arbre dont la racine possède n fils. Dans une définition inductive d'un ensemble X, on peut voir les élément de B comme des feuilles et l'application d'une règle d'inférence r d'arité n comme un arbre*[1] *de racine r, celle-ci possédant n fils. On obtient ainsi une surjection de l'ensemble des arbres dont les nœuds internes sont étiquetés par les éléments de R et les feuilles par les éléments de B dans X. La définition inductive de X est non ambiguë si cette surjection est injective.*

Exemple 7.17. *Parmi les définitions inductives vues plus haut, seule celles des expressions arithmétiques de l'exemple 7.11 est ambiguë. Voici une autre définition, non ambiguë :*

1. On parle plutôt de terme dans la littérature.

— $\mathcal{N} \subset \mathcal{A}$;

— *pour tout* $op \in \{+, \times, -, /\}$, *on a* $g, h \in \mathcal{A} \Rightarrow (g\ op\ h) \in \mathcal{A}$;

Par exemple, les deux dérivations différentes qui donnaient $3-4+5$ *précédemment donnent maintenant* $((3-4)+5)$ *et* $(3-(4+5))$.

7.4.2 Ordre sur un ensemble inductif

Un ensemble inductif défini de manière non ambiguë peut-être muni d'une relation d'ordre (non totale) directement liée à la définition, comme on va le voir. On va d'abord définir la relation entre les x_i et $r(x_1, \dots, x_n)$ où r est une règle d'inférence, et prolonger cette relation.

Définition 7.18. *Soit* $\mathcal{R}$ *une relation sur un ensemble* E. *On appelle fermeture réflexive-transitive de la relation la relation* $\mathcal{R}'$ *définie par :*

$$x\mathcal{R}'y \iff \exists\, n \geq 0, \quad \exists\, x_0, x_1, \dots, x_n \in E, \quad x = x_0, \quad y = x_n \quad et \quad x_i \mathcal{R} x_{i+1} \text{ pour tout } i \in [\![0, n-1]\!]$$

Proposition 7.19. *La fermeture réflexive-transitive d'une relation est une relation réflexive et transitive.*

Démonstration. Évident ! Pour la réflexivité, remarquer que l'on peut prendre $n = 0$ dans la définition. □

Théorème 7.20. *Soit* X *un ensemble inductif défini de manière non ambiguë. On introduit la relation* $\prec_1$ *telle que* $x_i \prec_1 r(x_1, \dots, x_n)$ *pour toute règle d'inférence* r *d'arité* n, *et tout* n*-uplet* $(x_1, \dots, x_n)$. *Notons* $\preceq$ *la fermeture réflexive transitive de la relation* $\prec_1$. *Alors* $\preceq$ *est une relation d'ordre sur* X.

Démonstration. Il ne reste à montrer que l'anti-transitivité. Mais si $x \preceq y$ et $y \preceq x$, alors $x = y$ car sinon la non ambiguïté serait contredite, puisqu'on pourrait obtenir x à nouveau à partir de x et d'une suite non triviale d'applications des règles d'inférence. □

On rappelle qu'un ensemble ordonné est bien fondé s'il ne possède pas de suite infinie strictement décroissante. C'est le cas ici.

Proposition 7.21. *Sous les hypothèses du théorème précédent, l'ensemble* $(X, \preceq)$ *est bien fondé.*

Démonstration. On peut construire une suite de sous-ensembles de X de la façon suivante :

$$X_0 = B \qquad \text{et} \qquad \forall\, n > 0 \quad X_n = \{r(x_1, \dots, x_p) \mid r \in R \text{ d'arité } p, x_1, \dots, x_p \in \cup_{i<n} X_i, \text{ l'un des } x_i \text{ est dans } X_{n-1}\}$$

Alors $X = \cup_{i>0} X_i$. En effet, en notant Y cette union, on voit facilement que les X_i sont inclus dans X, donc Y aussi, et comme Y contient les éléments de base B et et est stable par les règles, il contient X. La non ambiguïté implique que les X_i sont disjoints, et par définition de $\preceq$, si $x \prec y$ avec $x \in X_p$ et $y \in X_q$, alors $p < q$. Une suite strictement décroissante démarrant par un élément de X_p est donc de longueur au plus $p+1$. □

Autre preuve. On note $P(x)$ la propriété sur X définie par « x n'est pas à l'origine d'une suite infinie strictement décroissante ».

— P est vérifiée pour x dans B. C'est immédiat par non ambiguïté, car aucun élément y de X ne vérifie $y \prec x$.

— Montrons que P est héréditaire. Soit $x_1, \dots, x_n$ vérifiant P, dans l'ensemble de définition d'une règle r d'arité n, et soit $y = r(x_1, \dots, x_n)$. Supposons y à l'origine d'une suite strictement décroissante pour l'ordre $\preceq$, notée $y = y_0 \succ y_1 \succ y_2 \cdots$. Par définition de $\preceq$, il existe $m > 0$ et des éléments y'_j tels que $y_0 \succ_1 y'_1 \succ_1 \cdots \succ_1 y'_m = y_1$. Puisque X est non ambigü, y'_1 est l'un des x_i, et en particulier $x_i \succ y_1 \succ y_2 \succ \cdots$, donc x_i est à l'origine d'une suite infinie strictement décroissante. C'est absurde, d'où l'hérédité de P.

Par principe d'induction, $(X, \preceq)$ est bien fondé. □

7.4.3 Fonctions sur un ensemble inductif

Il est facile de définir une fonction sur un ensemble inductif, pourvu que l'on en possède une définition inductive non ambiguë, comme le prouve le théorème suivant.

Théorème 7.22. *Soit* X *un ensemble défini inductivement, avec ensemble de base* B *et règles de* R. *On suppose l'ensemble défini de manière non ambiguë. La donnée de :*

— *valeurs* $f(x)$ *pour tout* x *dans* B ;

— *valeurs de* $f(r(x_1, \dots, x_n))$ *en fonction des* x_i *et des* $f(x_i)$ *pour toute règle* r *d'arité* n ;

permet de définir une fonction f sur X.

Démonstration. Il suffit de montrer que ces hypothèses définissent la valeur $f(x)$ de manière unique, propriété sur les éléments de X que l'on montre par induction :

— c'est vrai pour les éléments x de B ;

— Tout autre élément x de X s'obtenant de manière unique sous la forme $x = r(x_1, \ldots, x_n)$, $f(x)$ est bien défini de manière unique.

□

Exemple 7.23 (Suite de l'exemple 7.8)**.** *On peut définir la hauteur h d'un arbre binaire de la façon suivante : l'abre vide a pour hauteur -1, et ensuite $h(Noeud(g, x, d)) = 1 + \max(h(g), h(d))$.*

7.4.4 Analyse d'une fonction récursive

Considérons un ensemble inductif X défini de manière non ambigüe. Une fonction récursive sur X utilisant la structure à la manière du théorème 7.22 est facile à analyser :

— sa terminaison est évidente en vertu de la propriété 7.21 ;

— sa correction se montre par induction.

— sa complexité s'analyse de la même manière qu'une fonction récursive quelconque.

Chapitre 8

Implémentation des dictionnaires

La structure de dictionnaire est une structure abstraiet qui doit gérer des couples (clé, élément). Les clés sont à valeurs dans un ensemble ordonné (entiers, chaînes de caractères), supposées deux à deux distinctes, contrairement aux éléments qui sont quelconques. La structure doit garantir les opérations :

— création d'un dictionnaire vide ;
— test d'égalité d'un dictionnaire au vide ;
— recherche de la présence d'un couple (clé, élément) à partir de la clé ;
— insertion d'un couple (clé, élément), si la clé n'est pas déja présente ;
— suppression d'un couple (clé, élément) à partir de la clé.
— remplacement de l'élément d'un couple (clé, élément) par un autre de même clé.

Voici quelques exemples d'utilisation de cette structure : un dictionnaire peut être utilisé pour gérer un dictionnaire (usuel) ! Plus généralement, ils sont très utilisés, par exemple pour gérer des listes d'utilisateurs d'un site web. En algorithmique, on peut par exemple se servir d'un dictionnaire pour implémenter la fonction de modification de la priorité d'un élément dans une file de priorité (ce qu'on verra dans ce chapitre).

Une fois donnée une structure abstraite, le but du jeu est d'en donner une implémentation efficace, ce que l'on va faire maintenant.

8.1 Tas et file de priorité

8.1.1 Une file de priorité à l'aide d'une liste ?

Une première idée pour implémenter une FP-max (c'est pareil pour une FP-min) est d'utiliser une simple liste.

Utilisation d'une liste non triée. Avec une liste non triée, quelques-unes des opérations sont faciles (notamment l'insertion d'un nouvel élément), par contre l'opération consistant à retirer l'élément prioritaire n'est pas évidente car il faut parcourir toute la liste pour calculer le maximum. Utiliser une liste non triée n'est donc pas une bonne idée d'implémentation.

Utilisation d'une liste triée. Avec une liste triée (dans l'ordre décroissant), il est maintenant facile de retirer l'élément de plus grande priorité d'une FP-max. Par contre, l'insertion d'un nouvel élément nécessite de chercher où insérer le nouvel élément dans la liste, ce qui prend un temps linéaire en la taille de la liste dans le pire cas. Utiliser une liste triée n'est donc pas non plus une bonne idée.

8.1.2 La structure de tas

Dans la suite, on se concentre sur l'implémentation d'une FP-max, pour laquelle on va utiliser un tas-max (on ferait usage de manière symétrique d'un tas-min pour une FP-min).

Définition 8.1. *Un tas-max est un arbre binaire complet à gauche, tel que l'étiquette d'un nœud quelconque de l'arbre soit supérieure ou égale à celles de ses fils.*

Remarque 8.2. *Deux remarques :*

— *pour un tas-min, remplacer « supérieure ou égale » par « inférieure ou égale » ;*

— *la propriété de tas-max implique que l'étiquette d'un nœud s est supérieure ou égale à celles de tous les nœuds appartenant au sous-arbre enraciné en s. En particulier la racine de l'arbre a la plus grande étiquette.*

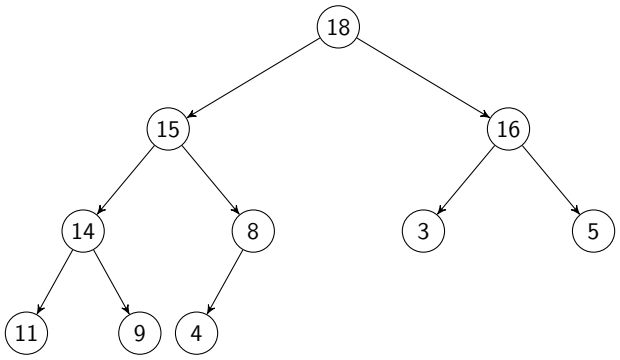


FIGURE 8.1 – Un tas-max

La figure 10.1 présente un tel tas-max. Pour mémoire, un arbre est dit *binaire* lorsque tous ses nœuds ont au plus deux fils. Il est dit complet si toutes ses feuilles se trouvent à la même profondeur (tous les niveaux de l'arbre sont remplis). Un arbre binaire complet à gauche a ses feuilles sur les deux derniers niveaux, l'avant dernier étant rempli au maximum et le dernier rempli le plus à gauche possible.

Avant de passer à l'implémentation concrète d'un tel arbre, donnons un encadrement de sa hauteur en fonction de son nombre de nœuds.

Définition 8.3. *La profondeur d'un nœud dans un arbre binaire est définie inductivement :*

— *la racine est à profondeur zéro ;*

— *la hauteur d'un nœud qui n'est pas la racine est celle de son parent, plus un.*

Définition 8.4. *La hauteur d'un arbre est la profondeur maximale de ses feuilles.*

Proposition 8.5. *Un arbre binaire complet à gauche de hauteur h a entre 2^h et $2^{h+1}-1$ nœuds.*

Démonstration. On numérote les niveaux de l'arbre par profondeur croissante. L'arbre possède $h+1$ niveaux, les h premiers étant remplis cela représente $1+2+\cdots+2^{h-1}$ nœuds, soit 2^h-1. Le dernier niveau comporte entre 1 et 2^h nœuds, ce qui donne l'encadrement désiré. □

Corollaire 8.6. *La hauteur d'un arbre binaire complet à gauche à n nœuds est $\lfloor \log_2(n) \rfloor$.*

Démonstration. Immédiat. □

Remarque 8.7. *Ainsi, la hauteur d'un arbre binaire complet à gauche à n nœuds est en $O(\log n)$. Toutes les fonctions que l'on écrira dans la suite auront cette complexité, ce qui mène à une implémentation de la structure de file de priorité bien plus intéressante qu'avec des listes !*

8.1.3 Opérations sur un tas

Dans cette partie, on décrit de manière théorique comment maintenir la structure de tas-max lorsque l'on rajoute/supprime un élément, ou lorsqu'on modifie un nœud. Essentiellement, il y a deux mécanismes à décrire, dont les autres vont découler. Il s'agit de rétablir la structure de tas lorsqu'on diminue ou augmente l'étiquette d'un nœud. On suppose que l'échange des étiquettes contenues dans deux nœuds peut se faire en complexité constante dans le tas.

Augmentation de l'étiquette. Imaginons que l'on augmente l'étiquette d'un nœud. Alors la propriété de tas-max n'est plus nécessairement vérifiée, mais le seul endroit où elle peut être violée est entre ce nœud et son parent : il se peut que l'étiquette du parent soit maintenant strictement inférieure. Dans ce cas, on peut échanger l'étiquette du nœud avec celle de son parent, et réitérer l'opération : l'étiquette augmentée va progressivement remonter jusqu'à ce que la structure soit rétablie. Essentiellement, on applique l'algorithme récursif 10.8 pour rétablir la structure de tas ($E(s)$ désigne l'étiquette du nœud s) :

Analysons l'algorithme :

- la terminaison est immédiate, car si un appel récursif est effectué, c'est avec un nœud dont la profondeur est strictement inférieure à celle de s ;

Algorithme 8.8 : monter_nœud

Entrées : Un (presque) tas-max t, un nœud s
si *s n'est pas la racine et $E(parent(s)) < E(s)$* **alors**
  Échanger les étiquettes de s et parent(s) ;
  monter_nœud$(t, \text{parent}(s))$

- la correction est immédiate, pour peu que l'on précise ce qu'est un « presque tas-max », dont un sommet s est marqué : il s'agit d'un arbre vérifiant toutes les propriétés de tas-max, sauf peut-être $E(\text{parent}(s)) \geq E(s)$. Il est clair que si l'algorithme s'arrête alors t est un tas-max, et si un appel récursif est effectué, c'est clairement sur un presque tas-max et sur le sommet pouvant violer la condition de tas ;
- la complexité est clairement linéaire en la profondeur de s, majorée par la hauteur du tas.

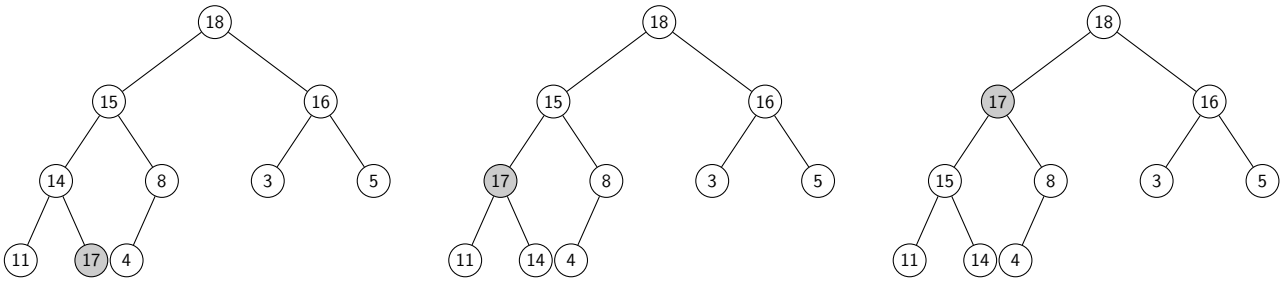


FIGURE 8.2 – L'effet de monter_nœud sur le nœud d'étiquette (augmentée) 17

Diminution de l'étiquette. À l'inverse, lorsqu'on diminue l'étiquette d'un nœud, la propriété de tas peut être violée car l'étiquette du nœud est plus grande que l'un (au moins) de ses enfants. Dans ce cas, il suffit d'échanger le nœud avec son fils ayant la plus grande étiquette pour que le problème descende d'un cran, voir l'algorithme 10.9.

Algorithme 8.9 : descendre_nœud

Entrées : Un (presque) tas-max t, un sommet s
$u \leftarrow s$;
si *s a un fils gauche g et $E(g) > E(s)$* **alors**
  $u \leftarrow g$
si *s a un fils gauche d et $E(d) > E(u)$* **alors**
  $u \leftarrow d$
si $u \neq s$ **alors**
  Échanger les étiquettes de s et u ;
  descendre_nœud(t, u)

De même, la terminaison et la correction sont faciles à montrer par récurrence décroissante sur la profondeur du nœud s : si s est une feuille, il n'y a rien à faire, et sinon on reporte éventuellement le problème à un nœud de plus grande profondeur. La complexité est clairement linéaire en la distance (maximale) entre s et une feuille, majorée par la hauteur du tas.

FIGURE 8.3 – L'effet de descendre_nœud sur le nœud d'étiquette (diminuée) 7

Rajout d'un nœud. Il suffit de placer le nœud de façon à maintenir la structure d'arbre binaire complet à gauche, et d'appeler monter_nœud pour rétablir la structure de tas.

Suppression de la racine. Pour rétablir la structure de tas-max facilement si l'on supprime la racine, il suffit d'y placer le nœud situé en dernière feuille, puis d'appeler descendre_nœud.

8.1.4 Stockage d'un arbre binaire complet à gauche dans un tableau

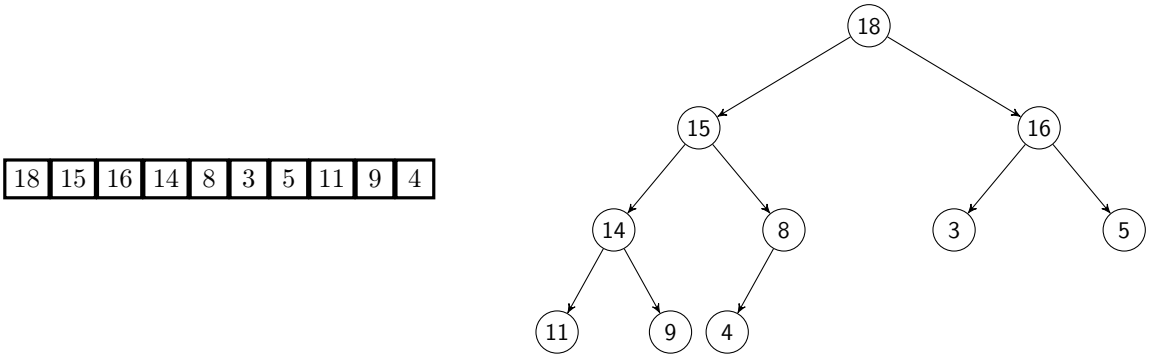


FIGURE 8.4 – Un tableau et l'arbre associé

On peut facilement stocker un arbre binaire complet à gauche dans un tableau (ce qui rendra facile les échanges d'étiquettes) : il suffit de stocker les éléments de l'arbre dans l'ordre du parcours en largeur. On vérifie alors aisément que, en identifiant nœud de l'arbre et indice dans le tableau :

Proposition 8.10. *Si i est l'indice d'un élément du tableau, correspondant au i-ème nœud dans l'énumération du parcours en largeur (initialisée à partir de la racine d'indice 0) :*

- *si $i \neq 0$ (donc i n'est pas la racine), le père de i est $\left\lfloor \frac{i-1}{2} \right\rfloor$;*
- *si i possède un fils gauche, c'est $2i+1$;*
- *de même, son fils droit éventuel est $2i+2$.*

Démonstration. Il suffit de montrer la proposition sur les indices des fils, l'assertion sur l'indice du parent en résulte. On vérifie (par récurrence immédiate) que les nœuds à profondeur h ont pour indices $2^h - 1, 2^h, \cdots, 2^{h+1} - 2$. Ainsi le nœud le plus à gauche à la profondeur $h+1$ est $2^{h+1} - 1$, le suivant 2^{h+1}, qui s'obtiennent bien comme $2(2^{h-1} - 1)$ auquel on ajoute 1 ou 2. Il en va donc de même des suivants. □

Remarque 8.11. *Il est assez courant lorsqu'on implémente un tas dans un tableau, de commencer à remplir le tableau à partir de l'indice 1 en ignorant la première case, ce qui donne $\left\lfloor \frac{i}{2} \right\rfloor$, $2i$ et $2i+1$ dans la proposition précédente. On ne le fera pas dans ce cours.*

8.1.5 Implémentation de la structure de file de priorité max dans un tableau

Pour la réalisation d'une file de priorité, il est nécessaire d'utiliser un tableau qui n'est pas entièrement rempli, pour laisser de la place aux éléments que l'on va rajouter à la file.

On présente ici une implémentation où la taille du tableau est fixée une fois pour toute à la création [1], et l'on maintient une variable indiquant le nombre d'éléments du tableau qui font effectivement partie du tas : si cette variable est n, les éléments d'indice 0 à $n-1$ sont ceux du tas, les suivants pouvant être quelconques.

On utilise donc le type suivant :

```
type 'a fp = {mutable n: int ; tab: 'a array} ;;
```

Le champ `n` désigne le nombre d'éléments effectivement présents dans la file de priorité, le tableau `tab` a alors ses n premiers éléments qui forment le tas.

1. Une implémentation plus générale, sans hypothèse sur la capacité de la file de priorité, consiste à utiliser un tableau redimensionnable (similaire à une liste Python) : il n'est pas très compliqué d'implémenter une telle structure, il suffit de doubler la taille du tableau utilisé lorsque l'on manque de place. Il faut alors recopier tous les éléments de l'ancien tableau vers le nouveau, ce qui est coûteux, mais cette opération est faite suffisamment peu souvent pour que l'ajout d'un élément à la structure se fasse en complexité *amortie* constante.

Couples (priorité, élément). En pratique pour implémenter une structure de file de priorité, il faut stocker dans le tableau des couples (priorité, élément). La comparaison d'éléments en Caml se faisant avec l'ordre lexicographique, toutes les fonctions que l'on va écrire compareront les couples comme il se doit.

Fonctions basiques. Les fonctions `fg`, `fd`, `pere` et `echanger` nous serviront à manipuler facilement le tableau associé au tas. La fonction `creer_fp` prend en paramètre la taille du tableau (la capacité de la file), ainsi qu'un élément `x` utile pour fixer le type de la file de priorité, même si la file est vide à la création. Enfin, pour tester si une file de priorité est vide, il suffit de vérifier que le champ `n` est nul.

```
let fg i = 2*i+1 ;;
let fd i = 2*i+2 ;;
let pere i = (i-1)/2 ;;
let echanger t i j = let a=t.(i) in t.(i) <- t.(j) ; t.(j) <- a ;;

let creer_fp taille x = {n=0 ; tab=Array.make taille x} ;;

let est_vide_fp f = f.n=0 ;;
```

Monter et descendre. Pour implémenter les fonctions `monter_noeud` et `descendre_noeud`, il suffit de suivre les algorithmes 10.8 et 10.9. Les deux fonctions prennent en paramètre le tableau sur lequel travailler et un indice `i` qui est celui du nœud pouvant violer la structure de tas. La fonction `descendre_noeud` prend également en paramètre le nombre d'éléments `n` du tas associé (seuls les `n` premiers éléments du tableau forment le tas).

```
let rec monter_noeud t i=
  if i<>0 && t.(pere i)<t.(i) then begin
    echanger t i (pere i) ;
    monter_noeud t (pere i)
  end
;;

let rec descendre_noeud t n i=
  let j=ref i in
  if fg i < n && t.(fg i)>t.(i) then j:=(fg i) ;
  if fd i < n && t.(fd i)>t.( !j) then j:=(fd i) ;
  if i<> !j then begin
    echanger t i !j ;
    descendre_noeud t n !j
  end
;;
```

Fonctions de file de priorité. On donne maintenant l'implémentation des deux principales opérations de file de priorité max (ajouter un élément, supprimer le maximum), exceptées celles qui modifient une clé, dont on parlera un peu plus loin.

Pour enfiler un élément, on suppose qu'il y a assez de place dans le tableau, et on rajoute un élément à la suite des autres éléments du tas. On incrémente le nombre d'éléments contenus dans la file, et on appelle `monter_noeud` sur le dernier élément du tas. À l'inverse, pour retirer le maximum (la racine du tas), on choisit de l'échanger avec le dernier élément du tas, en décrémentant le nombre d'éléments contenus dans la file. On appelle `descendre_noeud` sur la nouvelle racine, et on renvoie l'élément supprimé.

```
let enfiler_fp f x=
  f.tab.(f.n) <- x ;
  f.n <- f.n + 1 ;
  monter_noeud f.tab (f.n - 1)
;;

let supprimer_max_fp f=
  f.n <- f.n - 1 ;
  echanger f.tab 0 f.n ;
  descendre_noeud f.tab f.n 0 ;
  f.tab.(f.n)
;;
```

8.1.6 Complexité

Ainsi implémentée, la structure de file de priorité possède une complexité $O(\log n)$ pour toutes ses opérations, où n est le nombre d'éléments présents dans la file de priorité, exceptée la création qui est linéaire en la capacité choisie, et le test d'égalité au vide qui s'exécute en temps constant.

8.1.7 Intermède : le tri par tas

Les fonctions précédemment écrites suffisent à implémenter le fameux « tri par tas » : pour trier un tableau, on crée une file de priorité dans laquelle on rajoute les éléments du tableau un par un, puis on supprime les éléments de la file un par un, ce qui fournit les éléments du tableau dans l'ordre décroissant, que l'on peut donc stocker à la bonne place.

```
let tri_par_tas t=
  let n=Array.length t in
  let f=creer_fp n t.(0) in
  for i=0 to n-1 do
    enfiler_fp f t.(i)
  done ;
  for i=n-1 downto 0 do
    t.(i) <- supprimer_max_fp f
  done ;;
```

La correction est relativement évidente. En terme de complexité, les opérations `enfiler_fp` et `supprimer_max_fp` s'exécutent en temps $O(\log n)$ avec n la taille du tableau (qui est aussi la taille de la capacité de la file de priorité que l'on crée) : on en déduit une complexité totale $O(n \log n)$.

Remarque 8.12. *Il est en fait possible de trier le tableau « en place » avec ces idées : essentiellement on transforme le tableau en tas à l'aide de* `monter_noeud`*, puis on le trie en réécrivant essentiellement le code de la fonction* `supprimer_max_fp`*.*

8.1.8 Modification d'une clé

Dans une file de priorité, les éléments stockés sont des couples (clé, élément). L'une des opérations possibles de la structure est la modification de la clé associée à un élément présent dans la file. Il faut donc être capable de retrouver le nœud du tas à partir de l'élément. On indique deux solutions.

- Un cas qui se produit souvent est celui où les éléments sont à valeurs dans un ensemble de la forme $[\![0, B-1]\!]$, avec B un entier pas trop grand. On peut alors utiliser un tableau de taille B pour stocker les positions des éléments présents dans la file de priorité dans le tableau associé au tas. Lorsqu'on utilise la fonction `echanger` dans une des fonctions ci-dessus, il ne faut pas oublier de répercuter le changement sur ce tableau des positions. De même, il faut gérer l'insertion et la suppression d'un élément dans la file de priorité avec ce tableau. La complexité des opérations de file de priorité restent alors logarithmique en le nombre d'éléments, de même que la modification de la priorité : il suffit de monter ou descendre l'élément dont on a modifié la priorité. Un exemple où cette approche est très efficace est celle de l'algorithme de Djikstra sur un graphe dont les sommets sont numérotés de 0 à $N-1$.
- Plus généralement, on peut utiliser une structure de dictionnaire dans laquelle on stocke des couples (élément, position dans le tableau), ce qui amène à la section suivante ! La complexité des opérations de file de priorité s'en trouve modifiée suivant la structure choisie, signalons qu'avec une table de hachage elle est quasiment inchangée [2]

8.2 Arbres binaires de recherche, Arbres AVL

8.2.1 Implémentation d'une structure de dictionnaire avec une liste chaînée

Pour les mêmes raisons que pour une file de priorité, utiliser une structure de liste chaînée pour implémenter une structure de dictionnaire n'est pas efficace du point de vue de la complexité : il faut stocker des couples (clé, élément) où les clés sont des éléments distincts, et rechercher si une clé est présente se fait en temps linéaire en le nombre d'éléments présents dans le pire cas. On va, là aussi, proposer une structure efficace à l'aide d'arbres, plus précisément d'*arbres binaires de recherche*.

2. Bien implémentée, une table de hachage effectue ses opérations en temps constant amorti, sous des hypothèses naturelles de répartition « aléatoire » des clés de hachage.

8.2.2 Structure d'arbre binaire de recherche

Définition 8.13. *un arbre binaire de recherche (abrégé ABR) est un arbre binaire éventuellement vide dont les nœuds sont des couples (clé, valeur), tel que*

- *deux nœuds n'ont jamais la même clef ;*
- *les clefs sont à valeur dans un ensemble totalement ordonné (on peut ordonner les clefs) ;*
- *pour tout nœud de l'arbre, les clés du sous arbre gauche sont strictement inférieures à celle du nœud, celles du sous arbre droit sont strictement supérieures.*

Deux exemples d'ABR avec comme ensemble de clés {canari, castor, chameau, chat, cheval, chien, chouette} (ordonné pour l'ordre naturel sur les chaînes de caractères) sont donnés figure 8.5. On n'y a pas mis les valeurs associées, qui pourraient être les définitions de ces mots dans la langue française.

FIGURE 8.5 – Deux exemples d'ABR avec un même ensemble de clés

La condition sur les clés d'un ABR implique la proposition suivante.

Proposition 8.14. *Un arbre binaire est un ABR si et seulement si l'énumération infixe de ses nœuds est strictement croissante.*

Remarque 8.15. *L'énumération infixe n'est pas unique, elle correspond à plusieurs ABR. C'est le cas par exemple pour les ABR de la figure 8.5 : les clés sont les mêmes donc les deux arbres ont même énumération infixe.*

Définition 8.16. *La hauteur d'un ABR est définie inductivement comme suit :*

- *la hauteur de l'arbre vide est -1 ;*
- *pour un arbre non vide, elle est égale au maximum des hauteurs des sous-arbres gauche et droit, plus un.*

Remarque 8.17. *On peut considérer qu'un ABR non vide est un arbre binaire entier, les « feuilles » de l'arbre étant des arbres vides. C'est cohérent avec la définition de la hauteur, et pratique avec l'implémentation Caml que l'on va voir au paragraphe suivant.*

Proposition 8.18. *Le nombre de nœuds n d'un ABR (non vide) de hauteur h est compris entre $h+1$ et $2^{h+1}-1$*

Démonstration. La borne $2^{h+1}-1$ est obtenue pour un arbre binaire complet, comme dans le cas des tas. Un chemin de la racine a une feuille à profondeur h comportant $h+1$ nœuds, l'autre borne est démontrée. □

Corollaire 8.19. *La hauteur h d'un ABR (non vide) à n nœuds vérifie $\lfloor \log_2(n) \rfloor \leq h \leq n-1$.*

Démonstration. La propriété précédente fournit l'encadrement $\log_2(n+1)-1 \leq h \leq n-1$. Or $h \geq \log_2(n+1)-1 > \log_2(n)-1 \geq \lfloor \log_2(n) \rfloor - 1$. Donc $h \geq \lfloor \log_2(n) \rfloor$. □

8.2.3 Implémentation des ABR en Caml

On choisit une implémentation *persistante* des ABR, identique à celle classique des arbres binaires. Contrairement à l'implémentation des tas de la section précédente, les fonctions qui travailleront sur les arbres ne modifieront pas la structure mais renverront de nouveaux arbres. En pratique, on implémente les ABR comme des arbres binaires entiers (tout nœud interne a 2 fils), les feuilles correspondent à un arbre vide (remarque : on oublie ces nœuds dans la définition de la hauteur [3]).

```
type 'a abr = Vide | N of 'a abr * 'a * 'a abr;;
```

3. Attention, ce n'est pas forcément standard.

On peut facilement calculer la hauteur :

```
let rec hauteur a=match a with
  | Vide -> -1
  | N(g,_,d) -> 1 + max (hauteur g) (hauteur d)
;;
```

8.2.4 Implémentation de la structure de dictionnaire avec des ABR

En Caml, on pourrait représenter les nœuds comme :

```
type ('a, 'b) noeud = {cle : 'a; mutable valeur: 'b};;
```

On a rendu le deuxième champ mutable, car dans un dictionnaire on peut vouloir changer la valeur associée à une clé : dans ce cas il suffit de retrouver le nœud à partir de la clé et modifier la valeur associée. Ainsi la modification d'une valeur est essentiellement une recherche de clé. Dans la suite on oubliera les valeurs pour ne considérer que les clés, d'un point de vue algorithmique les opérations sont les mêmes.

Fonctions basiques. Créer un arbre binaire de recherche (vide) et tester si un arbre binaire de recherche est vide sont évidentes :

```
let creer_abr () = Vide ;;
let est_vide_abr a = a=Vide ;;
```

Recherche et insertion dans un ABR. Pour rechercher un élément ou insérer un élément dans un ABR, on suit le même principe de cheminement dans l'arbre ; la condition sur les clés impose que si l'on cherche x dans un arbre de racine y, on est dans l'un des trois cas suivants :

— soit $y = x$, auquel cas on a trouvé x ;

— soit $y < x$, auquel cas x ne peut se trouver que dans le sous-arbre droit du nœud étiqueté par y ;

— soit $y > x$, et x ne peut se trouver que dans le sous-arbre gauche.

```
let rec rechercher a x=match a with
  | Vide -> false
  | N(g,y,_) when y>x -> rechercher g x
  | N(_,y,d) when y<x -> rechercher d x
  | _ -> true
;;
```

Pour l'insertion de `x` dans l'ABR, on chemine dans l'arbre jusqu'à arriver à une feuille `Vide`, que l'on remplace par un arbre contenant simplement `x`. On choisit par convention de renvoyer l'arbre à l'identique si `x` est déja présent [4]

```
let rec inserer a x=match a with
  | Vide -> N(Vide,x,Vide)
  | N(g,y,d) when y>x -> N(inserer g x, y, d)
  | N(g,y,d) when y<x -> N(g, y, inserer d x)
  | _ -> a
;;
```

Suppression d'un élément dans un ABR. La suppression d'un élément est légèrement plus complexe. Comme pour l'insertion, on se ramène à la suppression de la racine d'un ABR. On propose alors deux solutions.

- Suppression par fusion : si `x` est la racine de l'ABR et l'élément à supprimer, on peut *fusionner* les deux sous-arbres gauche et droit de l'ABR. Cette fusion est immédiate si l'un des deux arbres est vide, et sinon elle peut se faire en suivant le schéma de la figure 8.6, qui donne bien un ABR si les deux arbres situés à gauche en sont bien, et que les clés du premier sont strictement inférieures aux clés du second. Bien sûr le choix de x_1 comme nouvelle racine est arbitraire et on pourrait prendre symétriquement x_2.

  Écrire les fonctions est alors facile :

4. On rappelle que les clés sont supposées distinctes.

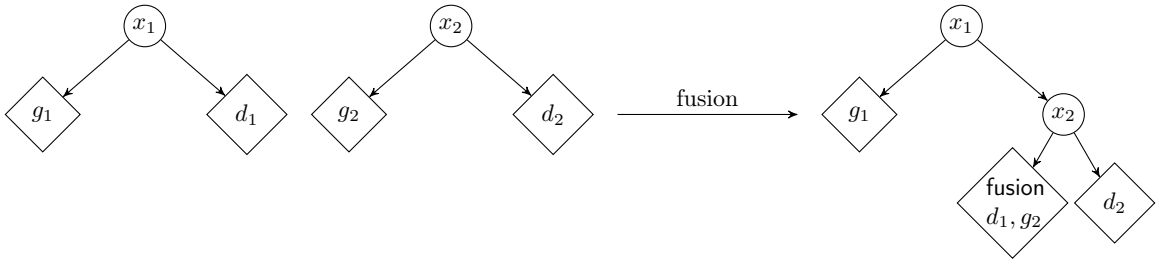


FIGURE 8.6 – Fusion de deux ABR : les clés du premier arbre sont toutes strictement inférieures à celles du deuxième.

```
let rec fusion a1 a2=match a1,a2 with
  | Vide,_ -> a2
  | _,Vide -> a1
  | N(g1,x1,d1), N(g2,x2,d2) -> N(g1,x1,N(fusion d1 g2,x2,d2))
;;


let rec supprimer a x=match a with
  | Vide -> Vide
  | N(g,y,d) when y>x -> N(supprimer g x, y, d)
  | N(g,y,d) when y<x -> N(g, y, supprimer d x)
  | N(g,y,d) -> fusion g d
;;
```

De même que pour l'insertion, si le nœud à supprimer n'est pas présent dans l'arbre, celui-ci est renvoyé à l'identique.

- Suppression du maximum du sous-arbre gauche : supposons à nouveau que la racine `x` de soit l'élément à supprimer. Si le sous-arbre gauche est vide, l'arbre obtenu après suppression est simplement le sous-arbre droit. Sinon, on peut remplacer la racine par le maximum de son sous-arbre gauche (préalablement supprimé) : on maintient bien ainsi la structure d'ABR. Ainsi, on écrit d'abord une fonction qui renvoie le maximum d'un arbre binaire de recherche : il suffit de descendre le plus à droite possible.

```
let rec max_abr a=match a with
  | Vide -> failwith "vide"
  | N(_,x,Vide) -> x
  | N(_,_,d) -> max_abr d
;;

let rec supprimer a x=match a with
  | Vide -> Vide
  | N(g,y,d) when y>x -> N(supprimer g x, y, d)
  | N(g,y,d) when y<x -> N(g, y, supprimer d x)
  | N(Vide,y,d) -> d
  | N(g,y,d) -> let z=max_abr g in N(supprimer g z, z, d)
;;
```

Complexité des opérations. Toutes les opérations insertion/suppression/recherche prennent un temps $O(h)$, avec h la hauteur de l'arbre. Malheureusement, h peut être proche du nombre de nœuds, ce qui est pas mauvais en terme de complexité. Par exemple, en partant d'un arbre vide, si on insère successivement n nœuds dans l'ordre croissant, on obtient un arbre de hauteur $n-1$, et la construction se fait en temps $O(n^2)$ (on construit un « peigne »).

Il y a plusieurs solutions pour remédier à ce problème :

— si on se donne un ensemble de clés que l'on insère dans un ordre aléatoire (avec distribution uniforme sur les $n!$ permutations possibles), on peut montrer que l'espérance de la hauteur de l'arbre obtenu est $O(\log n)$: en pratique les choses se passent bien si on laisse faire le hasard ;

— on peut rajouter de l'information dans l'ABR. Ces informations permettent, en utilisant des « rotations » bien choisies, d'équilibrer l'arbre pour garder une hauteur $O(\log n)$.

La suite du chapitre est dévolue à l'étude des arbres AVL, qui s'inscrivent dans la stratégie du deuxième point.

8.2.5 Arbres AVL

Définition 8.20. *Un arbre AVL*[5] *est un ABR tel que pour tout nœud, ses sous-arbres gauche et droit aient même hauteur, à 1 près.*

Cette condition est suffisante pour que l'arbre soit approximativement équilibré, comme le montre le théorème suivant :

Théorème 8.21. *La hauteur h d'un arbre AVL à n nœuds vérifie $h = O(\log n)$.*

Démonstration. Il suffit de voir qu'un arbre AVL a toujours un grand nombre de nœuds, relativement à sa hauteur. Notons N_h le nombre de nœuds minimal d'un arbre AVL de hauteur h. On a $N_0 = 1$, $N_1 = 2$ et $N_h = N_{h-1} + N_{h-2} + 1$. La suite $(N_h + 1)$ vérifie donc la même relation que la suite de Fibonacci. D'où $N_h = C\varphi^h + D(-1/\varphi)^h - 1$, avec $\varphi = \frac{1+\sqrt{5}}{2}$ et C et D deux constantes. Asymptotiquement, $N_h \sim C\varphi^h$. Par suite un arbre AVL à n nœuds possède une hauteur majorée par $\log_\varphi(n)$ (à une constante additive près), d'où le résultat. □

Remarque 8.22. $\frac{1}{\log_2(\varphi)} \simeq 1.44$*, donc asymptotiquement un arbre AVL à n nœuds a une hauteur majorée par une quantité de l'ordre de* $1.44\log_2(n)$*. On n'est pas très loin des arbres binaires complets pour lesquels la hauteur est bornée par* $\log_2(n)$*.*

8.2.6 Rotations et maintien de la structure d'arbre AVL

Les opérations de recherche, insertion, suppression dans un arbre AVL à n nœuds se font donc en temps $O(\log n)$. Mais il faut modifier les fonctions d'insertion et de suppression pour que la structure d'arbre AVL soit maintenue. On va procéder en utilisant des rotations sur l'arbre, expliquées à la figure 8.7. On remarque que ces rotations maintiennent bien la structure d'ABR.

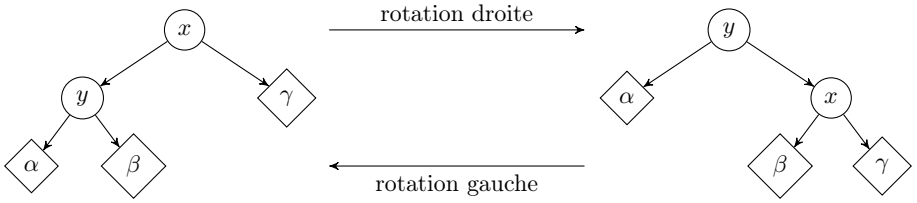


FIGURE 8.7 – Rotations dans un ABR : x et y sont des nœuds, α, β et γ des ABR.

En fait, on va insérer/supprimer les nœuds comme dans les ABR, mais on opérera un rééquilibrage éventuel « à la remontée », en procédant par rotations. Lorsqu'on insère/supprime un nœud, il se peut que l'on introduise un déséquilibre qui fait perdre la structure d'arbre AVL. Comme insertion et suppression font varier les hauteurs des sous-arbres d'au plus 1, s'il y a déséquilibre c'est qu'un nœud (dont l'arbre associé a pour hauteur h) possède deux sous-arbres de hauteurs respectives $h-1$ et $h-3$. On suppose que le sous-arbre de hauteur $h-1$ est celui de gauche, l'autre cas s'en déduit par symétrie. Le sous-arbre de gauche se décompose en une racine (y) et deux sous-arbres gauche et droit α et β. On distingue alors deux cas :

- la hauteur de α est supérieure ou égale à celle de β (voir figure 8.8). Ainsi $h(\alpha) = h-2$ et $h(\beta) = h-2-p$, avec $p \in \{0, 1\}$. Une rotation droite suffit alors, et l'arbre total passe d'une hauteur h à $h-p$: si $p = 1$ le déséquilibre se propage potentiellement plus haut.
- la hauteur de α est strictement inférieure à celle de β (voir figure 8.9). β se décompose en une racine (z) et deux sous-arbres δ et ε, de hauteurs respectives $h-3-p$ et $h-3-q$, avec $p, q \in \{0, 1\}$ l'un au moins étant nul. On vérifie qu'une rotation gauche sur le sous-arbre enraciné en y nous ramène (presque) au cas précédent. Une rotation droite équilibre l'arbre total, qui passe d'une hauteur h à une hauteur $h-1$.

8.2.7 Opérations sur les arbres AVL en Caml

On donne simplement l'idée de l'implémentation des opérations sur les AVL dans ce cours. On peut par exemple utiliser le type suivant :

```
type 'a avl = Vide | N of int * 'a avl * 'a * 'a avl
```

5. du nom de ses inventeurs : Georgii Adelson-Velsky et Evguenii Landis.

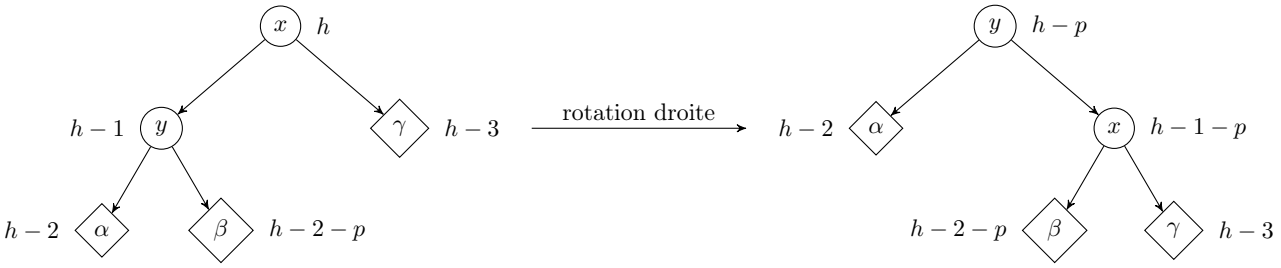


FIGURE 8.8 – Déséquilibre dans un arbre AVL, cas 1 : une rotation droite suffit.

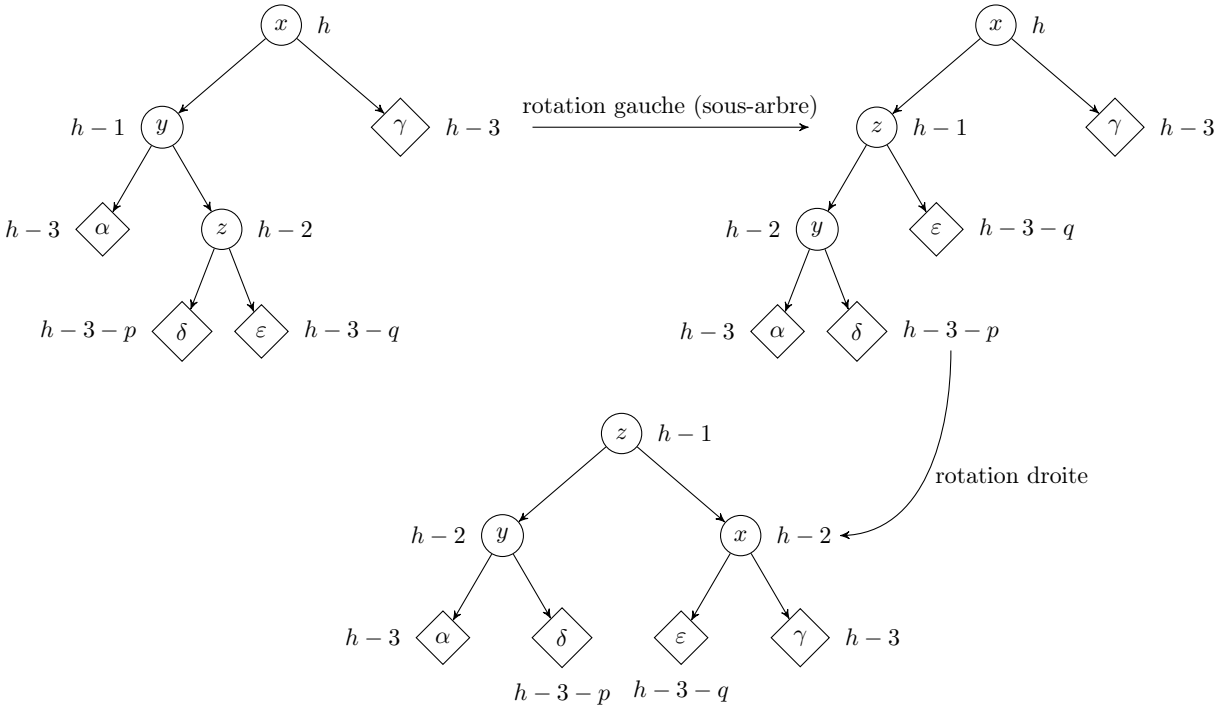


FIGURE 8.9 – Déséquilibre dans un arbre AVL, cas 2 : une rotation gauche sur le sous-arbre gauche nous ramène quasiment au cas précédent, une rotation droite résout le déséquilibre. La hauteur de l'arbre a diminué de 1.

Le champ entier indique la hauteur du sous-arbre associé au nœud. Ainsi on n'a pas à recalculer la hauteur des sous-arbres d'un arbre à équilibrer (cette opération serait linéaire en le nombre de nœuds, donc très mauvaise car on veut une complexité logarithmique pour les opérations d'ABR !). La discussion précédente permet d'écrire une fonction `equilibrer : 'a avl -> 'a avl`, qui prend en paramètre un arbre qui est presque un AVL : les sous-arbres gauche et droit sont supposés être des AVL de hauteur qui diffèrent d'au plus 2. Le champ hauteur de l'arbre est également possiblement erroné. La fonction procède si nécessaire à une ou deux rotations, et renvoie un arbre AVL (avec champ hauteur correct), et ce *en temps constant*. Une fois écrite cette fonction, il est facile de réécrire des versions pour arbres AVL des fonctions sur les ABR. Voici par exemple le code de la fonction d'insertion :

```
let rec inserer a x=match a with
  | Vide -> N(0,Vide,x,Vide)
  | N(h,g,y,d) when y=x -> a
  | N(h,g,y,d) when y<x -> equilibrer (N(h,g,y,inserer d x))
  | N(h,g,y,d) -> equilibrer (N(h,inserer g x,y,d))
;;
```

À l'aide du théorème 8.21 on voit que l'on a bien créé une structure de dictionnaire où toutes les opérations s'effectuent en temps logarithmique en le nombre d'entrées.

Remarque 8.23. *Une autre implémentation classique des dictionnaires est celle faisant usage d'une* table de hachage.

Chapitre 9

Logique

9.1 Introduction

Trois collègues déjeunent ensemble à midi. A, B, C. Les faits suivants sont vrais :

— si A prend un dessert, B aussi ;

— soit B, soit C prennent un dessert, mais pas les deux ;

— A ou C prend un dessert ;

— si C prend un dessert, A aussi.

On va voir que A et B prennent un dessert, pas C.

9.2 Syntaxe des expressions logiques et représentation arborescente

9.2.1 Définitions

Définition 9.1. *Soit V un ensemble au plus dénombrable de variables logiques, qu'on notera $V = \{v_1, \dots, v_n, ..\}$. L'ensemble des expressions logiques sur V est défini inductivement :*

— *0 et 1 sont des expressions logiques (aussi notées Faux et Vrai, $\bot$ et $\top$...) ;*

— *tout $v \in V$ est une expression logique ;*

— *pour ψ et φ deux expressions logiques :*

* *$(\psi \wedge \phi)$ est une expression logique (lire « et ») ;*
* *$(\psi \vee \phi)$ est une expression logique (lire « ou ») ;*
* *$\neg\psi$ est une expression logique (lire « non »).*

Exemple 9.2. *$\neg(v_1 \wedge v_2)$ ou encore $(v_1 \vee \neg(v_2 \wedge v_2))$ sont des expressions logiques. $\neg(v_1 \wedge v_2)$ et $(\neg v_1 \vee \neg v_2)$ sont deux expressions logiques différentes.*

9.2.2 Représentation arborescente

Les expressions logiques admettent naturellement une représentation arborescente : les variables logiques et les constantes 0 et 1 sont les étiquettes des feuilles, les nœuds internes ayant pour étiquettes les opérateurs. Un nœud d'étiquette $\wedge$ ou $\vee$ est d'arité 2, un nœud d'étiquette $\neg$ est d'arité 1.
Cette représentation permet de définir la longueur et la hauteur d'une expression logique :

Définition 9.3. *On appelle :*

— *hauteur d'une expression logique la hauteur de l'arbre binaire associé ;*

— *longueur d'une expression logique le nombre de nœuds de l'arbre associé.*

La hauteur ou la longueur peut, de manière équivalente, également être définie par induction.

9.2.3 Simplification de l'écriture

Pour raccourcir l'écriture d'une expression logique, on définit une priorité des opérateurs : $\neg$ est prioritaire sur $\wedge$, lui même prioritaire sur $\vee$.

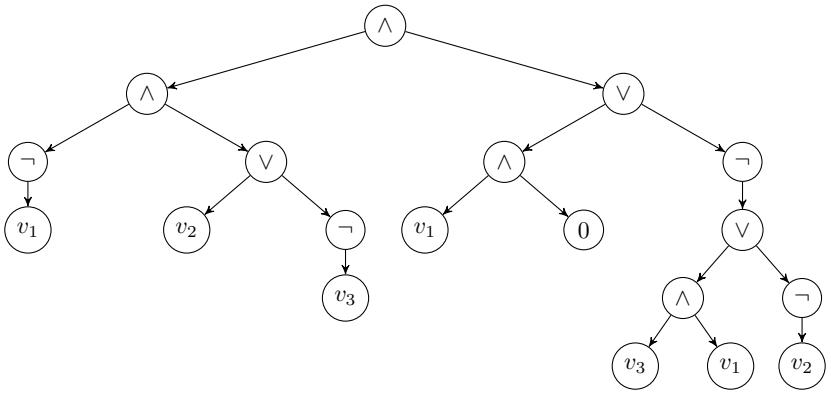


FIGURE 9.1 – La représentation arborescente d'une expression logique en $\{v_1, v_2, v_3\}$

9.2.4 Implémentation

On utilise naturellement une implémentation proche de la structure d'arbres pour définir un type `exp_log`. On utilise un type polymorphe pour pouvoir avoir des variables logiques indexées par des entiers, des chaînes de caractères, ou encore tout type à notre convenance.

```
type 'a exp_log =
  Zero | Un
  | V of 'a
  | Et of 'a exp_log * 'a exp_log
  | Ou of 'a exp_log * 'a exp_log
  | Non of 'a exp_log
;;
```

Voici deux fonctions [1] permettant de calculer la hauteur et la longueur d'une expression logique :

```
let rec longueur e=match e with
  | Zero | Un | V _ -> 1
  | Et (g,d) | Ou (g,d) -> 1 + longueur g + longueur d
  | Non g -> 1 + longueur g
;;

let rec hauteur e=match e with
  | Zero | Un | V _ -> 0
  | Et (g,d) | Ou (g,d) -> 1 + max (hauteur g) (hauteur d)
  | Non g -> 1 + hauteur g
;;
```

Avec l'exemple de la figure 9.1, on obtient bien hauteur 5 et longueur 19 :

```
let e = Et
  (Et (Non (V 1), Ou (V 2, Non (V 3))),
   Ou (Et (V 1, Zero),
       Non (Ou (Et (V 3, V 1), Non (V 2)))));
;;

print_int (hauteur e) ;;
print_int (longueur e) ;;
```

```
# val e : int exp_log =
  Et (Et (Non (V 1), Ou (V 2, Non (V 3))),
   Ou (Et (V 1, Zero), Non (Ou (Et (V 3, V 1), Non (V 2)))))
# 5- : unit = ()
# 19- : unit = ()
```

9.3 Sémantique des expressions logiques

9.3.1 Distribution de vérité

Définition 9.4. *Soit V un ensemble fini de variables. Une distribution de vérité sur V est une application $V \to \{0,1\}$.*

Proposition 9.5. *Si $|V| = n$, il y a 2^n distributions de vérité sur V.*

1. Un motif multiple (de la forme `m1 | m2`) avec identificateurs est possible, à condition que les mêmes identificateurs figurent des deux côtés.

9.3.2 Évaluation d'une expression logique. Équivalence sémantique

Définition 9.6. *Soit d une distribution de vérité sur V. On définit inductivement l'évaluation $\mathcal{E}_d(\varphi)$ d'une expression logique φ à variables dans V suivant la distribution d comme suit :*

- $\mathcal{E}_d(0) = 0$;
- $\mathcal{E}_d(1) = 1$;
- $\mathcal{E}_d(v) = d(v)$ *pour tout* $v \in V$;
- $\mathcal{E}_d(\varphi \wedge \psi) = \mathcal{E}_d(\varphi)\mathcal{E}_d(\psi)$;
- $\mathcal{E}_d(\varphi \vee \psi) = \mathcal{E}_d(\varphi) + \mathcal{E}_d(\psi) - \mathcal{E}_d(\varphi)\mathcal{E}_d(\psi)$;
- $\mathcal{E}_d(\neg\varphi) = 1 - \mathcal{E}_d(\varphi)$.

Voici les tables des deux opérateurs $\wedge$ et $\vee$ sur $\{0,1\}$:

$\wedge$	0	1
0	0	0
1	0	1

$\vee$	0	1
0	0	1
1	1	1

Remarque 9.7. *L'ensemble $\{0,1\}$, muni des deux lois de compositions internes $\wedge$ et $\vee$ a une structure dite d'*algèbre de Boole. *Ce n'est pas une algèbre au sens usuel (il faudrait utiliser $\oplus$, défini plus loin, à la place de $\vee$ pour obtenir une $(\mathbb{Z}/2\mathbb{Z})$-algèbre).*

Définition 9.8. *La table de vérité d'une expression logique φ sur V est la donnée des $\mathcal{E}_d(\varphi)$ pour toute distribution de vérité d sur V.*

On représente en général la table de vérité comme un tableau, les lignes indexées par les distributions de vérité, et les colonnes par les éléments de V et φ.

Exemple 9.9. *Voici la table de vérité de l'expression $\varphi = \neg v_1 \vee (v_2 \wedge \neg v_3)$ sur $V = \{v_1, v_2, v_3\}$:*

v_1	v_2	v_3	φ
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Définition 9.10. *Deux expressions φ et ψ sur V sont dites* sémantiquement équivalentes *si elles ont la même table de vérité. On notera $\varphi \equiv \psi$ dans ce cas.*

Proposition 9.11. *À équivalence sémantique près, il y a (au plus) 2^{2^n} expressions logiques sur un ensemble de n variables logiques.*

Démonstration. Il y a 2^n distributions de vérité possibles, donc au plus 2^{2^n} tables de vérité associées à des expressions logiques. □

Remarque 9.12. *On verra dans la suite comment construire une expression logique associée à une table de vérité, il y a donc exactement 2^{2^n} expressions logiques à équivalence sémantique près.*

9.3.3 Expressions logiques à 2 variables

Avec deux variables logiques v_1 et v_2, il y a $16 = 2^{2^2}$ expressions logiques, à équivalence sémantique près. Dénombrons-les, en fonction du nombre de 1 dans la table de vérité.

- Sans 1 dans la table de vérité, il y a la formule logique 0.
- Avec un seul 1, il y en a 4 : $v_1 \wedge v_2$, $\neg v_1 \wedge v_2$, $v_1 \wedge \neg v_2$ et $\neg v_1 \wedge \neg v_2$.

- Avec deux 1, il y a 6 formules logiques à équivalence sémantique près. Outre les formules $v_1, \neg v_1, v_2$ et $\neg v_2$, il y en a deux autres : $v_1 \wedge v_2 \vee \neg v_1 \wedge \neg v_2$ et $\neg v_1 \wedge v_2 \vee v_1 \wedge \neg v_2$:

v_1	v_2	$v_1 \wedge v_2 \vee \neg v_1 \wedge \neg v_2$	$\neg v_1 \wedge v_2 \vee v_1 \wedge \neg v_2$
0	0	1	0
0	1	0	1
1	0	0	1
1	1	1	0

  On notera $v_1 \Leftrightarrow v_2$ toute formule logique sémantiquement équivalente à $v_1 \wedge v_2 \vee \neg v_1 \wedge \neg v_2$ et $v_1 \oplus v_2$ toute formule logique sémantiquement équivalente à $\neg v_1 \wedge v_2 \vee v_1 \wedge \neg v_2$. On prononce « v_1 équivalente à v_2 » et « v_1 xor v_2 ».
- Avec trois 1, il y en a 4 : $v_1 \vee v_2$, $\neg v_1 \vee v_2$, $v_1 \vee \neg v_2$ et $\neg v_1 \vee \neg v_2$. On note $v_1 \Rightarrow v_2$ pour une expression logique équivalente à $\neg v_1 \vee v_2$.
- Avec quatre 1, on retrouve la formule 1.

Définition 9.13. *Lorsqu'on ne considère que les équivalences sémantiques, on note également :*
— $v_1 \uparrow v_2$ *pour une formule équivalente à* $\neg(v_1 \wedge v_2)$ *(pronconcer« nand ») ;*
— $v_1 \downarrow v_2$ *pour une formule équivalente à* $\neg(v_1 \vee v_2)$ *(pronconcer« nor »).*

Proposition 9.14 (Lois de De Morgan)**.** *On a* $v_1 \uparrow v_2 \equiv \neg v_1 \vee \neg v_2$ *et* $v_1 \downarrow v_2 \equiv \neg v_1 \wedge \neg v_2$.

9.3.4 Retour au problème de l'introduction

On reprend le problème de l'introduction, et en notant égaement x la variable indiquant que x prend un dessert. On cherche à savoir pour quelles distributions de vérité les quatre propositions s'évaluent en 1.

Reformulons ces propositions en expressions logiques en les variables A, B et C.

— si A prend un dessert, B aussi : $A \Rightarrow B$;
— soit B, soit C prennent un dessert, mais pas les 2 : $B \oplus C$;
— A ou C prend un dessert : $A \vee C$
— si C prend un dessert, A aussi : $C \Rightarrow A$.

Dressons la table de vérité de ces 4 formules logiques en l'ensemble $V = \{A, B, C\}$.

A	B	C	$A \Rightarrow B$	$B \oplus C$	$A \vee C$	$C \Rightarrow A$
0	0	0	1	0	0	1
0	0	1	1	1	1	0
0	1	0	1	1	0	1
0	1	1	1	0	1	0
1	0	0	0	0	1	1
1	0	1	0	1	1	1
1	1	0	1	1	1	1
1	1	1	1	0	1	1

La ligne grisée correspond à la seule distribution de vérité pour laquelle l'évaluation des quatre formules logiques donne 1. On en déduit que A et B prennent un dessert, pas C.

9.4 Tautologies, antilogies et formules satisfiables

9.4.1 Définitions

Définition 9.15. *Soit* φ *une expression logique sur un ensemble de variables* V*. On dit que :*
— φ *est une tautologie si pour toute distribution de vérité d,* $\mathcal{E}_d(\varphi) = 1$ *;*
— φ *est une antilogie si pour toute distribution de vérité d,* $\mathcal{E}_d(\varphi) = 0$ *;*
— φ *est satisfiable s'il existe une distribution de vérité d telle que* $\mathcal{E}_d(\varphi) = 1$.

Remarque 9.16.
— *On dit aussi formule valide pour une tautologie et formule fausse pour une antilogie.*
— *Les problèmes de détermination des expressions tautologiques, antilogiques et satisfiables sont reliés, car :*
  — φ *tautologie* $\iff$ $\neg\varphi$ *antilogie ;*
  — φ *antilogie* $\iff$ φ *non satisfiable.*

9.4.2 Les tautologies : la base du raisonnement mathématique

Il est utile de posséder une liste de tautologies, bien qu'il soit impossible de « toutes » les lister, car elles sont en nombre infini. Donnons-en quelques-unes, mais avant ça une petite remarque sous forme de proposition :

Proposition 9.17. *Si φ est une expression tautologique en les variables $v_1, \dots, v_n$, et $\psi_1, \dots, \psi_n$ des expressions logiques en les variables $w_1, \dots, w_p$, alors l'expression logique obtenue en substituant ψ_i à v_i pour tout i est une tautologie en les variables $w_1, \dots, w_p$.*

Démonstration. Prenons une distribution de vérité d sur les variables $w_1, \dots, w_p$. Puisque φ est une tautologie, l'évaluation de φ avec la distribution de vérité associant chaque v_i à $\mathcal{E}_d(\psi_i)$ donne 1. Ainsi l'expression logique obtenue en substituant les ψ_i aux variables v_i est bien une tautologie en $w_1, \dots, w_p$. □

Remarque 9.18. *Il en va de même pour une antilogie, mais pas pour une expression satisfiable.*

Voici une liste de tautologies :

- négation de la négation : $\neg\neg v \Leftrightarrow v$
- équivalence à vrai/faux : $(v \Leftrightarrow 1) \Leftrightarrow v$ et $(v \Leftrightarrow 0) \Leftrightarrow \neg v$
- idempotence : $v \wedge v \Leftrightarrow v$ et $v \vee v \Leftrightarrow v$
- implication et équivalence d'une même variable : $(v \Rightarrow v) \Leftrightarrow 1$ et $(v \Leftrightarrow v) \Leftrightarrow 1$
- commutativité : $v_1 \wedge v_2 \Leftrightarrow v_2 \wedge v_1$. De même avec $\vee$,$\oplus$, $\uparrow$, $\downarrow$ et $\Leftrightarrow$.
- équivalence des négations : $(\neg v_1 \Leftrightarrow \neg v_2) \Leftrightarrow (v_1 \Leftrightarrow v_2)$.
- lois de de Morgan : $\neg(v_1 \wedge v_2) \Leftrightarrow \neg v_1 \vee \neg v_2$ et $\neg(v_1 \vee v_2) \Leftrightarrow \neg v_1 \wedge \neg v_2$.
- associativité : $(v_1 \vee (v_2 \vee v_3)) \Leftrightarrow ((v_1 \vee v_2) \vee v_3)$, de même pour $\wedge$, $\oplus$ et $\Leftrightarrow$.
- distributivité de $\vee$ sur $\wedge$ et réciproquement : $(v_1 \vee (v_2 \wedge v_3)) \Leftrightarrow (v_1 \vee v_2) \wedge (v_1 \vee v_3)$, de même $(v_1 \wedge (v_2 \vee v_3)) \Leftrightarrow (v_1 \wedge v_2) \vee (v_1 \wedge v_3)$.
- transitivité : $((v_1 \Rightarrow v_2) \wedge (v_2 \Rightarrow v_3)) \Rightarrow (v_1 \Rightarrow v_3)$.
- simplification : $v_1 \vee (v_1 \wedge v_2) \Leftrightarrow v_1$, et $v_1 \vee (\neg v_1 \wedge v_2) \Leftrightarrow v_1 \vee v_2$. De même $v_1 \wedge (v_1 \vee v_2) \Leftrightarrow v_1$, et $v_1 \wedge (\neg v_1 \vee v_2) \Leftrightarrow v_1 \wedge v_2$.

Voici les bases du raisonnement mathématique :

- tiers exclus : $v \vee \neg v \Leftrightarrow 1$ et $v \wedge \neg v \Leftrightarrow 0$
- contraposée : $(v_1 \Rightarrow v_2) \Leftrightarrow (\neg v_2 \Rightarrow \neg v_1)$.
- double implication : $(v_1 \Rightarrow v_2) \wedge (v_2 \Rightarrow v_1) \Leftrightarrow (v_1 \Leftrightarrow v_2)$.
- disjonction de cas : $((v_1 \Rightarrow v_2) \wedge (\neg v_1 \Rightarrow v_2)) \Rightarrow v_2$.
- démonstration par l'absurde : $(\neg v_1 \Rightarrow 0) \Leftrightarrow v_1$.
- Modus ponens : $(v_1 \wedge (v_1 \Rightarrow v_2)) \Rightarrow v_2$.

9.5 Formes normales

Soit φ une expression logique. On cherche à trouver une expression sémantiquement équivalente à φ de la forme la plus « simple » possible.

9.5.1 Formes normales conjonctives et disjonctives

Définition 9.19. *Soit V un ensemble de variables logiques. On appelle* littéral *de V une expression logique de la forme v ou $\neg v$, pour $v \in V$.*

Définition 9.20. *On dit qu'une expression logique est* en forme normale disjonctive *si c'est une disjonction de conjonctions de littéraux.*

Exemple 9.21. *Avec $V = \{v_1, v_2, v_3\}$, l'expression $\varphi = (v_1 \wedge v_2 \wedge \neg v_3) \vee (\neg v_1 \wedge v_2)$ est sous forme normale disjonctive : c'est une disjonction de deux conjonctions de littéraux.*

Définition 9.22. *On appelle* clause *une disjonction de littéraux. On dit qu'une expression logique est* en forme normale conjonctive *si c'est une conjonction de clauses (donc une conjonction de disjonctions de littéraux).*

Exemple 9.23. *$v_1 \wedge (\neg v_1 \vee v_2 \vee v_3) \wedge (v_1 \vee \neg v_2)$ est sous forme normale conjonctive, composée de trois clauses à 1, 2 et 3 littéraux.*

En général, on suppose que les variables apparaissant dans une clause ou une conjonction de littéraux sont toutes différentes. En effet, d'une part on peut utiliser l'idempotence des opérateurs $\wedge$ et $\vee$ pour supprimer les littéraux égaux, et d'autre part une conjonction de littéraux comportant v et $\neg v$ est sémantiquement équivalente à 0, de même qu'une clause comportant v et $\neg v$ est sémantiquement équivalente à 1.

Remarque 9.24. *0 est le neutre pour $\vee$ et 1 est le neutre pour $\wedge$. On considère donc 0 comme une disjonction et 1 comme une conjonction.*

Sans hypothèses supplémentaires, il existe plusieurs (une infinité, en fait) formes normales disjonctives ou conjonctives équivalente à une expression logique donnée. Les formes normales sont en pratiques fournies en entrée des algorithmes décidant si une expression logique est satisfiable, tautologique ou antilogique : il faut donc être capable de mettre une expression sous forme normale.

9.5.2 Mise sous forme canonique

Avant de décrire un algorithme de mise sous forme normale conjonctive ou disjonctive, prouvons l'existence d'une telle écriture.

Théorème 9.25. *Soit φ une expression logique sur un ensemble de variables $V = \{v_1, \dots, v_n\}$. Alors φ est sémantiquement équivalente à une disjonction de conjonctions de littéraux et à une conjonction de clauses.*

Avant de montrer ce théorème, énonçons un lemme :

Lemme 9.26. *La négation d'une conjonction de disjonctions est une disjonction de conjonctions, et réciproquement.*

Démonstration. Il suffit d'appliquer les loi de De Morgan. □

Démonstration du théorème 9.25. On montre ce théorème par induction :

- 0 est une dijonction (sans conjonction). C'est également une conjonction (d'une seule disjonction, vide).
- de même pour 1, qui est une conjonction (sans disjonction) et la disjonction d'une seule conjonction, vide.
- v_i (et $\neg v_i$) est la disjonction d'une conjonction à un littéral, ou la conjonction d'une disjonction à un littéral.
- si φ admet une telle écriture, c'est le cas également pour $\neg\varphi$ d'après le lemme.
- si $\varphi = \varphi_1 \wedge \varphi_2$ avec φ_1 et φ_2 vérifiant le lemme, alors φ est également une conjonction de disjonctions. De plus si $\varphi_1 \equiv c_1^1 \vee \dots \vee c_{n_1}^1$ et $\varphi_2 \equiv c_1^2 \vee \dots \vee c_{n_2}^2$ avec les c_i^k des conjonctions de littéraux, alors en utilisant la distributivité de $\wedge$ sur $\vee$,
$$\varphi = \varphi_1 \wedge \varphi_2 \equiv \bigvee_{\substack{1\leq i\leq n_1\\ 1\leq j\leq n_2}} c_i^1 \wedge c_j^2$$
On peut éliminer les littéraux en doublons dans les conjonctions de la forme $c_i^1 \wedge c_j^2$, et éliminer totalement les conjonctions contenant une variable et sa négation.
- La démonstration est la même pour $\varphi_1 \vee \varphi_2$, en remplaçant conjonction par disjonction, et en échangeant $\wedge$ et $\vee$.

□

La démonstration du théorème précédent fournit une méthode pour calculer une forme normale (conjonctive ou disjonctive), il suffit :

- de supprimer les 0 et les 1 ;
- de reformuler $\Rightarrow$, $\oplus$, etc... à l'aide de $\vee$, $\wedge$ et $\neg$;
- d'utiliser les lois de De Morgan pour faire rentrer $\neg$ dans les littéraux ;
- d'utiliser la distributivité de $\vee$ sur $\wedge$ et celle de $\wedge$ sur $\vee$;
- de simplifier en éliminant les doublons de littéraux dans une clause/conjonction de littéraux et en utilisant les relations $v \wedge \neg v \equiv 0$ et $v \vee \neg v \equiv 1$.

• Mettons par exemple le problème de l'introduction sous forme normale conjonctive. L'expression logique associée est :

$$(A \Rightarrow B) \wedge (B \oplus C) \wedge (A \vee C) \wedge (C \Rightarrow A)$$

On reformule d'abord les opérateurs autres que $\neg$, $\wedge$ et $\vee$:

— $A \Rightarrow B \equiv \neg A \vee B$;
— $B \oplus C \equiv \neg B \wedge C \vee B \wedge \neg C$;
— $C \Rightarrow A \equiv \neg C \vee A$.

La formule obtenue est quasiment sous forme normale conjonctive, il ne reste plus qu'à utiliser les lois de De Morgan sur la formule issue de $B \oplus C$:

$$\begin{aligned}(\neg B \wedge C) \vee (B \wedge \neg C) &= ((\neg B \wedge C) \vee B) \wedge ((\neg B \wedge C) \vee \neg C) \\ &= (\neg B \vee B) \wedge (C \vee B) \wedge (\neg B \vee \neg C) \wedge (C \vee \neg C)\end{aligned}$$

Enfin, puisque $B \vee \neg B \equiv C \vee \neg C \equiv 1$, on obtient :

$$(A \Rightarrow B) \wedge (B \oplus C) \wedge (A \vee C) \wedge (C \Rightarrow A) \equiv (\neg A \vee B) \wedge (B \vee C) \wedge (\neg B \vee \neg C) \wedge (A \vee C) \wedge (\neg C \vee A)$$

La formule de droite est bien sous forme normale conjonctive : il y a 5 clauses à 2 littéraux.

• Reprenons l'exemple précédent pour obtenir une forme normale disjonctive, à partir de $(\neg A \vee B) \wedge (\neg B \wedge C \vee B \wedge \neg C) \wedge (A \vee C) \wedge (\neg C \vee A)$. On utilise maintenant la distributivité de $\wedge$ sur $\vee$:

$$\begin{aligned}\text{D'une part,} \quad (A \vee C) \wedge (\neg C \vee A) &\equiv (A \wedge \neg C) \vee (A \wedge A) \vee (C \wedge \neg C) \vee (A \wedge C) \\ &\equiv A \vee (A \wedge \neg C) \vee (A \wedge C) \quad \text{car } A \wedge A \equiv A \text{ et } C \wedge \neg C \equiv 0 \\ &\equiv A \quad \text{(simplification)}\end{aligned}$$

$$\text{D'autre part,} \quad (\neg A \vee B) \wedge A \equiv A \wedge B \quad \text{(simplification)}$$

$$\begin{aligned}\text{Enfin,} \quad A \wedge B \wedge ((\neg B \wedge C) \vee (B \wedge \neg C)) &\equiv (A \wedge B \wedge \neg B \wedge C) \vee (A \wedge B \wedge B \wedge \neg C) \quad \text{(loi de De Morgan)} \\ &\equiv A \wedge B \wedge \neg C \quad \text{car } B \wedge B \equiv B \text{ et } B \wedge \neg B \equiv 0\end{aligned}$$

Cette dernière expression est à la fois sous forme normale disjonctive (une seule conjonction de trois littéraux) et sous forme normale conjonctive (conjonction de trois clauses comportant un unique littéral).

Remarque 9.27. *On remarque via l'exemple précédent qu'une forme normale n'est pas unique (on a montré deux formes normales conjonctives équivalentes à la formule initiale). De plus, il n'est pas toujours facile d'obtenir une forme normale, car celle-ci peut être de taille exponentielle en la formule initiale. Par exemple, sur $V = \{x_1, \dots, x_n, y_1, \dots, y_n\}$ un ensemble de variables, l'expression*

$$\varphi = (x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \cdots \vee (x_n \wedge y_n)$$

est sous forme normale disjonctive, et de taille $O(n)$. On peut montrer qu'une forme normale conjonctive équivalente est :

$$\bigwedge_{z_i \in \{x_i, y_i\} \text{ pour } 1 \leq i \leq n} (z_1 \vee z_2 \vee \cdots \vee z_n)$$

qui comporte 2^n clauses, et on ne peut pas trouver plus petit.

Remarque 9.28. *Il est courant d'alléger les expressions pour utiliser des formes arithmétiques plus proches de celles qu'on utilise habituellement, ce qui a l'avantage de simplifier les écritures lorsqu'on travaille à équivalence sémantique près. Ainsi :*

— *on remplace le $\wedge$ par un point, voire par rien du tout ;*
— *on remplace le $\vee$ par $+$;*
— *on écrit $\bar{v}$ à la place de $\neg v$;*
— *on s'autorise à écrire $=$ à la place de $\equiv$.*

Cette forme rend plus intuitive la distributivité de $\wedge$ sur $\vee$. Il ne faut cependant pas oublier l'autre distributivité, l'absorbance de 1 pour $\vee$, l'idempotence, le fait que $a + \bar{a} = 1, a\bar{a} = 0$ et les autres règles de simplification. Par exemple :

$$(a+b)(a+\bar{b}+c) = a^2 + ab + a\bar{b} + b\bar{b} + ac + bc = a(1 + b + \bar{b} + c) + bc = a + bc$$

9.5.3 Formes normales conjonctives et disjonctives canoniques

Il est possible d'assurer l'unicité d'une forme normale conjonctive ou disjonctive, en rajoutant une condition sur les conjonction et disjonction de littéraux : celles-ci doivent être de taille maximale. Commençons par les formes normales disjonctives canoniques.

Formes normales disjonctives canoniques

Définition 9.29. *Soit $V = \{v_1, \dots, v_n\}$ un ensemble fini de variables logiques. Un min-terme sur V est une conjonction de n littéraux dans laquelle chaque variable apparaît exactement une fois.*

Exemple 9.30. *Avec $n = 3$, $\neg v_1 \wedge v_2 \wedge \neg v_3$ est un min-terme. $v_1 \wedge \neg v_2$ et $v_1 \wedge \neg v_1 \wedge v_2$ n'en sont pas.*

Proposition 9.31. *La table de vérité d'un min-terme contient un seul 1.*

Démonstration. En effet, pour m un min-terme, la seule distribution de vérité d telle que $\mathcal{E}_d(m) = 1$ est celle donnée par : $d(v_i) = \begin{cases} 1 & \text{si } v_i \text{ apparaît dans } m \\ 0 & \text{si } \neg v_i \text{ apparaît dans } m \end{cases}$. □

Théorème 9.32. *Soit φ une expression logique sur un ensemble fini de variables V. Alors φ est sémantiquement équivalente à une unique (à l'ordre près) disjonction de min-termes différents (deux-min termes sont considérés comme différents si les littéraux qui apparaîssent dans la conjonction diffèrent).*

Démonstration. Pour l'existence, on vérifie que

$$\varphi \equiv \bigvee_{d \mid \mathcal{E}_d(\varphi)=1} m_d$$

où m_d est le min-terme associé à la distribution d, défini par $m_d = \ell_1 \wedge \ell_2 \wedge \cdots \wedge \ell_n$ avec $\ell_i = \begin{cases} v_i & \text{si } d(v_i) = 1 \\ \neg v_i & \text{si } d(v_i) = 0 \end{cases}$.
Pour l'unicité, il suffit de voir que si d est une distribution de vérité telle que $\mathcal{E}_d(\varphi) = 1$, alors m_d est le seul min-terme tel que $\mathcal{E}_d(m_d) = 1$, il doit donc apparaître dans la décomposition. Réciproquement si $\mathcal{E}_d(\varphi) = 0$, alors m_d ne peut apparaître dans l'écriture. □

Ce théorème mène à la définition suivante :

Définition 9.33. *L'expression logique $\bigvee_{d \mid \mathcal{E}_d(\varphi)=1} m_d$ est la forme normale disjonctive* canonique *(en abrégé FNDC) sémantiquement équivalente à φ.*

On peut paraphraser la preuve précédente en disant que pour trouver la FNDC équivalente à φ, il suffit de regarder la table de vérité : chaque ligne avec un 1 fournit un min-terme apparaîssant dans la décomposition.

Exemple 9.34. *$A \wedge B \wedge \neg C$ est la FNDC de la formule $(A \Rightarrow B) \wedge (B \oplus C) \wedge (A \vee C) \wedge (C \Rightarrow A)$. Elle n'est composée que d'un seul min-terme.*

Exemple 9.35. *Considérons $\varphi = (v_1 \Rightarrow \neg v_2) \wedge (v_1 \vee v_3)$. Dressons sa table de vérité :*

v_1	v_2	v_3	$v_1 \Rightarrow \neg v_2$	$v_1 \vee v_3$	φ
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	1	0	0
0	1	1	1	1	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	0	1	0
1	1	1	0	1	0

Ainsi, $\varphi \equiv (\neg v_1 \wedge \neg v_2 \wedge v_3) \vee (\neg v_1 \wedge v_2 \wedge v_3) \vee (v_1 \wedge \neg v_2 \wedge \neg v_3) \vee (v_1 \wedge \neg v_2 \wedge v_3)$

Remarque 9.36. *Cette écriture permet de montrer qu'il y a bien 2^{2^n} formules logiques sur un ensemble de variables de taille n, à équivalence sémantique près.*

Formes normales conjonctives canoniques

La négation établit une dualité entre les formes normales disjonctives et les formes normales conjonctives. Ce qui suit est donc très proche de ce qu'on vient de voir sur les FNDC.

Définition 9.37. *Un max-terme sur un ensemble de variables de taille n est une disjonction de n littéraux dans laquelle chaque variable apparaît exactement une fois.*

Proposition 9.38. *La négation d'un min-terme est un max-terme, et réciproquement.*

Démonstration. C'est une simple application de la loi de De Morgan. □

Proposition 9.39. *La table de vérité d'un max-terme ne contient qu'un seul 0.*

Démonstration. Immédiat par négation. □

Théorème 9.40. *Une expression logique sur un ensemble fini de variables V est sémantiquement équivalente à une unique conjonction de max-termes différents. Cette écriture se nomme la forme normale conjonctive canonique (FNCC).*

Démonstration. Soit φ une expression logique sur V. Considérons la forme normale disjonctive canonique de $\neg\varphi$. Celle ci s'écrit $\bigvee_{d \mid \mathcal{E}_d(\varphi)=0} m_d$. Ainsi, par négation

$$\varphi \equiv \bigwedge_{d \mid \mathcal{E}_d(\varphi)=0} (\neg m_d)$$

et chacun des $\neg m_d$ est équivalent à un unique max-terme. □

Pour trouver la FNCC équivalente à φ, il suffit de regarder la table de vérité : chaque ligne avec un 0 fournit un max-terme apparaîssant dans la décomposition.

Exemple 9.41 (Suite de l'exemple 9.34)**.** *La FNDD de la formule $(A \Rightarrow B) \wedge (B \oplus C) \wedge (A \vee C) \wedge (C \Rightarrow A)$ est composée de 7 max-termes : tous sauf $\neg A \vee \neg B \vee C$*

Exemple 9.42 (Suite de l'exemple 9.35)**.** *La forme normale conjonctive canonique de $\varphi = (v_1 \Rightarrow \neg v_2) \wedge (v_1 \vee v_3)$ s'obtient en regardant les zéros dans sa table de vérité :*

$$\varphi \equiv (v_1 \vee v_2 \vee v_3) \wedge (v_1 \vee \neg v_2 \vee v_3) \wedge (\neg v_1 \vee \neg v_2 \vee v_3) \wedge (\neg v_1 \vee \neg v_2 \vee \neg v_3)$$

Application aux problèmes de décision

À partir d'une forme normale canonique (conjonctive ou disjonctive) il est facile de décider si une expression logique est satisfiable / tautologique / antilogique. En effet :

Proposition 9.43. *Soit φ une formule logique en un ensemble fini de variables. Alors :*

- *φ tautologie $\iff$ sa FNDC contient tous les min-termes $\iff$ sa FNCC ne contient aucun max-terme ;*
- *φ antilogie $\iff$ sa FNDC ne contient aucun min-terme $\iff$ sa FNCC contient tous les max-termes ;*
- *φ satisfiable $\iff$ sa FNDC contient au moins un min-terme $\iff$ sa FNCC ne contient pas tous les max-termes.*

Démonstration. Cela provient du fait que les min-termes sont liés aux uns dans la table de vérité, et les max-termes aux zéros. □

Le problème dans la mise en œuvre pratique (algorithmique) de cette proposition réside dans la taille potentiellement exponentielle des formes normales canoniques : le nombre de min-termes dans la FNDC additionné au nombre de max-termes dans la FNDD fait toujours 2^n avec n le nombre de variables, donc au moins l'une des deux est de taille exponentielle en n. Il n'est donc en général pas facile (algorithmiquement) de calculer ces formes normales canoniques.

9.6 Le problème SAT

Le problème SAT (pour satisfiabilité) est d'avoir un algorithme permettant de décider si une expression logique est satisfiable :

Entrée : une formule logique φ sur $V = \{v_1, \dots, v_n\}$.
Sortie : une distribution de vérité d sur V telle que $\mathcal{E}_d(\varphi) = 1$ si elle existe, la réponse « non satisfiable » sinon.

Une idée pour résoudre ce problème est de tester toutes les distributions de vérité. On obtient un algorithme de complexité $\Theta(2^n \ell)$ avec ℓ la taille de φ. Mais peut-on faire mieux ? L'idéal serait un algorithme de complexité dans le pire cas polynomiale en n (plus la taille de φ), c'est-à-dire de complexité $O((n+\ell)^k)$ pour un certain $k > 0$.

Considérons un problème (à priori) plus simple :

Définition 9.44. *Soit V un ensemble fini de variables. On rappelle qu'une clause est une disjonction de littéraux sur V, chaque variable apparaissant au plus une fois. On dit que c'est une p-clause si p variables apparaissent exactement.*

Définition 9.45. *Le problème p-SAT pour $p > 0$ est la restriction du problème SAT sur les conjonctions de p-clauses.*

Exemple 9.46. *La reformulation $(\neg A \vee B) \wedge (B \vee C) \wedge (\neg B \vee \neg C) \wedge (A \vee C) \wedge (\neg C \vee A)$ de l'expression logique issue du problème de l'introduction est une instance du problème 2-SAT.*

Proposition 9.47. *Le nombre de p clauses sur un ensemble de n variables est polynomial en n, à p fixé.*

Démonstration. En effet, il y a exactement $\binom{n}{p}2^p$ p-clauses différentes : $\binom{n}{p}$ correspond au choix des variables apparaîssant dans la clause, le facteur 2^p correspond au choix de v ou $\neg v$ pour chaque variable v apparaîssant dans la clause. Or $\binom{n}{p}2^p = \frac{2^p}{p!} n \times (n-1) \times \cdots \times (n-p+1) = O(n^p)$ à p-fixé. □

On peut supposer que les p-clauses d'une instance du problème p-SAT toutes différentes (il est très facile de supprimer les doublons). On est donc ramené à la question : « Existe t-il un algorithme de complexité $O(n^k)$ pour le problème p-SAT, avec $k > 0$? »

Les réponses à cette question sont :

— si $p = 1$, oui : il suffit que la formule ne contienne pas à la fois v_i et $\neg v_i$;
— si $p = 2$, oui : on verra un très joli algorithme dérivé d'algorithmes sur les graphes ;
— si $p \geq 3$: on ne sait pas. Celui qui résoudra la question recevra 1 million de dollars !

En fait, on peut montrer que le problème 3-SAT est aussi dur que les problèmes p-SAT et même que le problème SAT lui-même (si l'on sait résoudre 3-SAT avec un algorithme de complexité polynomiale, on peut résoudre en temps polynomial en la taille de l'entrée les problèmes p-SAT ou même SAT). Le problème 3-SAT fait partie de la grande classe des problèmes NP (à vérification polynomiale : si l'on fournit une distribution supposée satisfaire $\mathcal{E}_d(\varphi) = 1$, il est très facile de le vérifier en temps polynomial en la taille de φ). Une autre classe de problèmes est celle des problèmes résolubles en temps polynomial, qu'on note P. Clairement P $\subset$ NP, mais la plus grande question de l'informatique théorique est :

$$\text{A-t-on P} = \text{NP} ?$$

C'est pour la résolution de cette question que l'institut de mathématiques Clay offre 1 million de dollars. On peut en fait montrer que le problème 3-SAT est *NP-complet*, c'est-à-dire qu'un algorithme de complexité polynomiale pour l'un quelconque des problèmes NP-complets fournit un algorithme de complexité polynomiale pour n'importe quel problème NP. Trouver un tel algorithme pour le problème 3-SAT fournirait donc une solution positive à la question. Cependant, la plupart des informaticiens pensent que P $\neq$ NP (et donc que 3-SAT $\notin$ NP). En effet, on connaît aujourd'hui un très grand nombre de problèmes NP-complets (par exemple le problème du sac à dos, le calcul du nombre chromatique d'un graphe, résolution d'un système d'équations polynomiales...), mais toujours aucun algorithme de complexité polynomiale dans le pire cas : il est donc probable qu'il n'en existe pas.

On ne peut donc raisonnablement pas espérer trouver dans le cadre du cours un algorithme polynomial pour résoudre le problème SAT : on se contentera d'algorithmes de complexité exponentielle. Cependant, la complexité exponentielle dans le pire cas d'un algorithme de résolution de SAT n'implique en rien qu'il soit systématiquement difficile de résoudre SAT sur toutes les instances. Sur certaines « classes » d'instances particulières, il est possible de trouver des algorithmes de complexité polynomiale en la taille de l'entrée.

Deuxième partie

Programme de deuxième année

Chapitre 10

Implémentation des files de priorité

On rappelle que la structure de file de priorité (abrégé FP) doit gérer des couples de la forme (p, élément). Les premières composantes p sont à valeurs dans un ensemble ordonné (en pratique ce sont des entiers, qui indiquent la priorité de l'élément). Les éléments présents dans la FP sont tous distincts, par contre ce n'est pas nécessairement le cas des composantes p. Il y a une distinction à faire, suivant si les éléments prioritaires sont ceux ayant un p maximal (on parle de FP max) ou minimal (FP min). La structure doit garantir les opérations suivantes :

— création d'une FP vide ;

— test d'égalité d'une FP au vide ;

— insertion d'un nouvel élément, avec sa priorité ;

— retrait de l'élément de plus grande priorité ;

— modification de la priorité d'un couple (augmentation ou diminution).

Voici quelques exemples d'utilisation : une file de priorité peut être utilisée pour gérer un agenda (les taches les plus urgentes à effectuer doivent être effectuées en premier). On retrouve cette structure dans les imprimantes de bureau, ou dans la gestion des processus à effectuer sur un ordinateur. En algorithmique, on utilisera une file de priorité (min) pour l'implémentation efficace d'un algorithme de recherche de plus courts chemins dans un graphe pondéré (algorithme de Dijkstra, déja vu en première année).

10.1 Une file de priorité à l'aide d'une liste ?

Une première idée pour implémenter une FP-max (c'est pareil pour une FP-min) est d'utiliser une simple liste.

Utilisation d'une liste non triée. Avec une liste non triée, quelques-unes des opérations sont faciles (notamment l'insertion d'un nouvel élément), par contre l'opération consistant à retirer l'élément prioritaire n'est pas évidente car il faut parcourir toute la liste pour calculer le maximum. Utiliser une liste non triée n'est donc pas une bonne idée d'implémentation.

Utilisation d'une liste triée. Avec une liste triée (dans l'ordre décroissant), il est maintenant facile de retirer l'élément de plus grande priorité d'une FP-max. Par contre, l'insertion d'un nouvel élément nécessite de chercher où insérer le nouvel élément dans la liste, ce qui prend un temps linéaire en la taille de la liste dans le pire cas. Utiliser une liste triée n'est donc pas non plus une bonne idée.

10.2 La structure de tas

Dans la suite, on se concentre sur l'implémentation d'une FP-max, pour laquelle on va utiliser un tas-max (on ferait usage de manière symétrique d'un tas-min pour une FP-min).

Définition 10.1. *Un tas-max est un arbre binaire complet à gauche, tel que l'étiquette d'un nœud quelconque de l'arbre soit supérieure ou égale à celles de ses fils.*

Remarque 10.2. *Deux remarques :*

— *pour un tas-min, remplacer « supérieure ou égale » par « inférieure ou égale » ;*

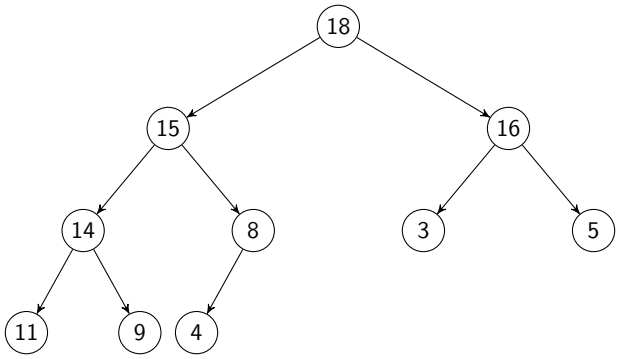


FIGURE 10.1 – Un tas-max

— *la propriété de tas-max implique que l'étiquette d'un nœud s est supérieure ou égale à celles de tous les nœuds appartenant au sous-arbre enraciné en s. En particulier la racine de l'arbre a la plus grande étiquette.*

La figure 10.1 présente un tel tas-max. Pour mémoire, un arbre est dit *binaire* lorsque tous ses nœuds ont au plus deux fils. Il est dit complet si toutes ses feuilles se trouvent à la même profondeur (tous les niveaux de l'arbre sont remplis). Un arbre binaire complet à gauche a ses feuilles sur les deux derniers niveaux, l'avant dernier étant rempli au maximum et le dernier rempli le plus à gauche possible.

Avant de passer à l'implémentation concrète d'un tel arbre, donnons un encadrement de sa hauteur en fonction de son nombre de nœuds.

Définition 10.3. *La profondeur d'un nœud dans un arbre binaire est définie inductivement :*

— *la racine est à profondeur zéro ;*

— *la hauteur d'un nœud qui n'est pas la racine est celle de son parent, plus un.*

Définition 10.4. *La hauteur d'un arbre est la profondeur maximale de ses feuilles.*

Proposition 10.5. *Un arbre binaire complet à gauche de hauteur h a entre 2^h et $2^{h+1}-1$ nœuds.*

Démonstration. On numérote les niveaux de l'arbre par profondeur croissante. L'arbre possède $h+1$ niveaux, les h premiers étant remplis cela représente $1+2+\cdots+2^{h-1}$ nœuds, soit 2^h-1. Le dernier niveau comporte entre 1 et 2^h nœuds, ce qui donne l'encadrement désiré. □

Corollaire 10.6. *La hauteur d'un arbre binaire complet à gauche à n nœuds est $\lfloor \log_2(n) \rfloor$.*

Démonstration. Immédiat. □

Remarque 10.7. *Ainsi, la hauteur d'un arbre binaire complet à gauche à n nœuds est en $O(\log n)$. Toutes les fonctions que l'on écrira dans la suite auront cette complexité, ce qui mène à une implémentation de la structure de file de priorité bien plus intéressante qu'avec des listes !*

10.3 Opérations sur un tas

Dans cette partie, on décrit de manière théorique comment maintenir la structure de tas-max lorsque l'on rajoute/supprime un élément, ou lorsqu'on modifie un nœud. Essentiellement, il y a deux mécanismes à décrire, dont les autres vont découler. Il s'agit de rétablir la structure de tas lorsqu'on diminue ou augmente l'étiquette d'un nœud. On suppose que l'échange des étiquettes contenues dans deux nœuds peut se faire en complexité constante dans le tas.

Augmentation de l'étiquette. Imaginons que l'on augmente l'étiquette d'un nœud. Alors la propriété de tas-max n'est plus nécessairement vérifiée, mais le seul endroit où elle peut être violée est entre ce nœud et son parent : il se peut que l'étiquette du parent soit maintenant strictement inférieure. Dans ce cas, on peut échanger l'étiquette du nœud avec celle de son parent, et réitérer l'opération : l'étiquette augmentée va progressivement remonter jusqu'à ce que la structure soit rétablie. Essentiellement, on applique l'algorithme récursif 10.8 pour rétablir la structure de tas ($E(s)$ désigne l'étiquette du nœud s) :

Analysons l'algorithme :

Algorithme 10.8 : monter_nœud

Entrées : Un (presque) tas-max t, un nœud s
si *s n'est pas la racine et $E(\text{parent}(s)) < E(s)$* **alors**
  Échanger les étiquettes de s et parent(s) ;
  monter_nœud(t, parent(s))

- la terminaison est immédiate, car si un appel récursif est effectué, c'est avec un nœud dont la profondeur est strictement inférieure à celle de s ;
- la correction est immédiate, pour peu que l'on précise ce qu'est un « presque tas-max », dont un sommet s est marqué : il s'agit d'un arbre vérifiant toutes les propriétés de tas-max, sauf peut-être $E(\text{parent}(s)) \geq E(s)$. Il est clair que si l'algorithme s'arrête alors t est un tas-max, et si un appel récursif est effectué, c'est clairement sur un presque tas-max et sur le sommet pouvant violer la condition de tas ;
- la complexité est clairement linéaire en la profondeur de s, majorée par la hauteur du tas.

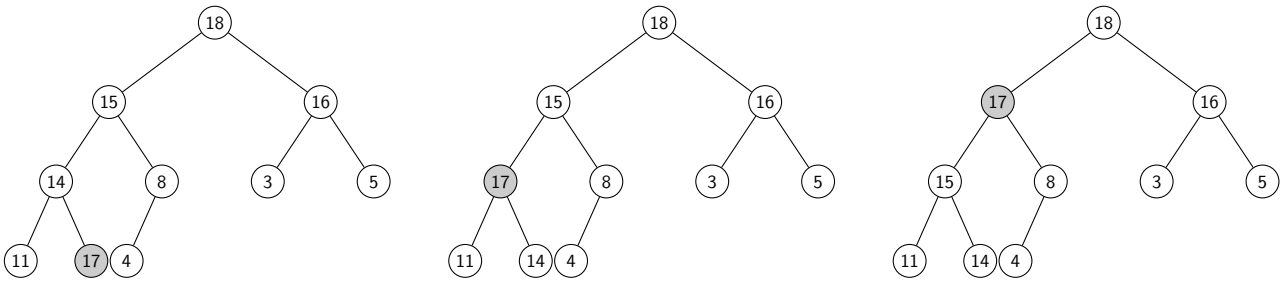


FIGURE 10.2 – L'effet de monter_nœud sur le nœud d'étiquette (augmentée) 17

Diminution de l'étiquette. À l'inverse, lorsqu'on diminue l'étiquette d'un nœud, la propriété de tas peut être violée car l'étiquette du nœud est plus grande que l'un (au moins) de ses enfants. Dans ce cas, il suffit d'échanger le nœud avec son fils ayant la plus grande étiquette pour que le problème descende d'un cran, voir l'algorithme 10.9.

Algorithme 10.9 : descendre_nœud

Entrées : Un (presque) tas-max t, un sommet s
$u \leftarrow s$;
si *s a un fils gauche g et $E(g) > E(s)$* **alors**
  $u \leftarrow g$
si *s a un fils gauche d et $E(d) > E(u)$* **alors**
  $u \leftarrow d$
si $u \neq s$ **alors**
  Échanger les étiquettes de s et u ;
  descendre_nœud(t, u)

De même, la terminaison et la correction sont faciles à montrer par récurrence décroissante sur la profondeur du nœud s : si s est une feuille, il n'y a rien à faire, et sinon on reporte éventuellement le problème à un nœud de plus grande profondeur. La complexité est clairement linéaire en la distance (maximale) entre s et une feuille, majorée par la hauteur du tas.

Rajout d'un nœud. Il suffit de placer le nœud de façon à maintenir la structure d'arbre binaire complet à gauche, et d'appeler monter_nœud pour rétablir la structure de tas.

Suppression de la racine. Pour rétablir la structure de tas-max facilement si l'on supprime la racine, il suffit d'y placer le nœud situé en dernière feuille, puis d'appeler descendre_nœud.

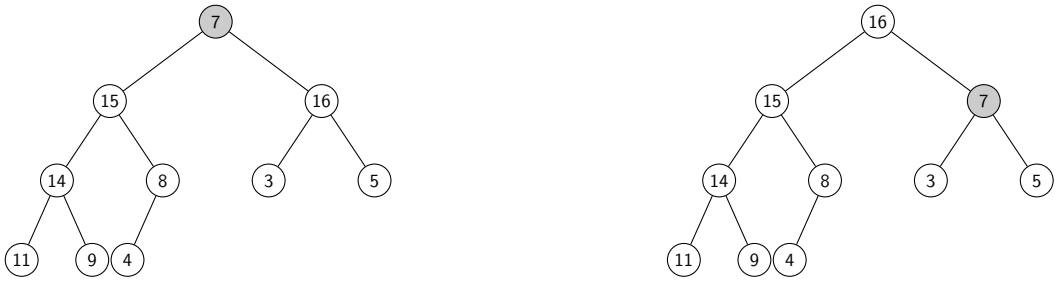


FIGURE 10.3 – L'effet de descendre_nœud sur le nœud d'étiquette (diminuée) 7

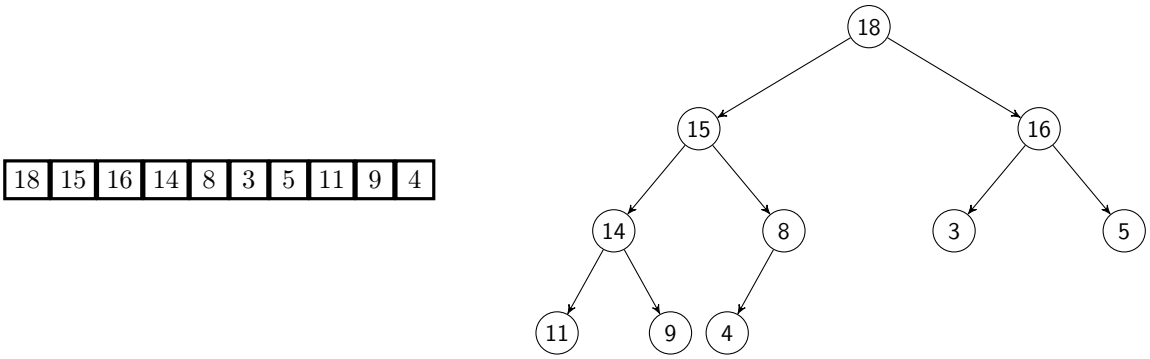


FIGURE 10.4 – Un tableau et l'arbre associé

10.4 Stockage d'un arbre binaire complet à gauche dans un tableau

On peut facilement stocker un arbre binaire complet à gauche dans un tableau (ce qui rendra facile les échanges d'étiquettes) : il suffit de stocker les éléments de l'arbre dans l'ordre du parcours en largeur. On vérifie alors aisément que, en identifiant nœud de l'arbre et indice dans le tableau :

Proposition 10.10. *Si i est l'indice d'un élément du tableau, correspondant au i-ème nœud dans l'énumération du parcours en largeur (initialisée à partir de la racine d'indice 0) :*

- *si $i \neq 0$ (donc i n'est pas la racine), le père de i est $\left\lfloor \frac{i-1}{2} \right\rfloor$;*
- *si i possède un fils gauche, c'est $2i+1$;*
- *de même, son fils droit éventuel est $2i+2$.*

Démonstration. Il suffit de montrer la proposition sur les indices des fils, l'assertion sur l'indice du parent en résulte. On vérifie (par récurrence immédiate) que les nœuds à profondeur h ont pour indices $2^h - 1, 2^h, \cdots, 2^{h+1} - 2$. Ainsi le nœud le plus à gauche à la profondeur $h+1$ est $2^{h+1} - 1$, le suivant 2^{h+1}, qui s'obtiennent bien comme $2(2^{h-1} - 1)$ auquel on ajoute 1 ou 2. Il en va donc de même des suivants. □

Remarque 10.11. *Il est assez courant lorsqu'on implémente un tas dans un tableau, de commencer à remplir le tableau à partir de l'indice 1 en ignorant la première case, ce qui donne $\left\lfloor \frac{i}{2} \right\rfloor$, $2i$ et $2i+1$ dans la proposition précédente. On ne le fera pas dans ce cours.*

10.5 Implémentation de la structure de file de priorité max dans un tableau

Pour la réalisation d'une file de priorité, il est nécessaire d'utiliser un tableau qui n'est pas entièrement rempli, pour laisser de la place aux éléments que l'on va rajouter à la file.

On présente ici une implémentation où la taille du tableau est fixée une fois pour toute à la création [1], et l'on maintient une variable indiquant le nombre d'éléments du tableau qui font effectivement partie du tas : si cette variable est n, les éléments d'indice 0 à $n-1$ sont ceux du tas, les suivants pouvant être quelconques.

1. Une implémentation plus générale, sans hypothèse sur la capacité de la file de priorité, consiste à utiliser un tableau redimensionnable (similaire à une liste Python) : il n'est pas très compliqué d'implémenter une telle structure, il suffit de doubler la taille du tableau utilisé lorsque l'on manque de place. Il faut alors recopier tous les éléments de l'ancien tableau vers le nouveau, ce qui est coûteux, mais cette opération est faite suffisamment peu souvent pour que l'ajout d'un élément à la structure se fasse en complexité *amortie* constante.

On utilise donc le type suivant :

```
type 'a fp = {mutable n: int ; tab: 'a array} ;;
```

Le champ `n` désigne le nombre d'éléments effectivement présents dans la file de priorité, le tableau `tab` a alors ses n premiers éléments qui forment le tas.

Couples (priorité, élément). En pratique pour implémenter une structure de file de priorité, il faut stocker dans le tableau des couples (priorité, élément). La comparaison d'éléments en Caml se faisant avec l'ordre lexicographique, toutes les fonctions que l'on va écrire compareront les couples comme il se doit.

Fonctions basiques. Les fonctions `fg`, `fd`, `pere` et `echanger` nous serviront à manipuler facilement le tableau associé au tas. La fonction `creer_fp` prend en paramètre la taille du tableau (la capacité de la file), ainsi qu'un élément `x` utile pour fixer le type de la file de priorité, même si la file est vide à la création. Enfin, pour tester si une file de priorité est vide, il suffit de vérifier que le champ `n` est nul.

```
let fg i = 2*i+1 ;;
let fd i = 2*i+2 ;;
let pere i = (i-1)/2 ;;
let echanger t i j = let a=t.(i) in t.(i) <- t.(j) ; t.(j) <- a ;;

let creer_fp taille x = {n=0 ; tab=Array.make taille x} ;;

let est_vide_fp f = f.n=0 ;;
```

Monter et descendre. Pour implémenter les fonctions `monter_noeud` et `descendre_noeud`, il suffit de suivre les algorithmes 10.8 et 10.9. Les deux fonctions prennent en paramètre le tableau sur lequel travailler et un indice `i` qui est celui du nœud pouvant violer la structure de tas. La fonction `descendre_noeud` prend également en paramètre le nombre d'éléments `n` du tas associé (seuls les `n` premiers éléments du tableau forment le tas).

```
let rec monter_noeud t i=
  if i<>0 && t.(pere i)<t.(i) then begin
    echanger t i (pere i) ;
    monter_noeud t (pere i)
  end
;;

let rec descendre_noeud t n i=
  let j=ref i in
  if fg i < n && t.(fg i)>t.(i) then j:=(fg i) ;
  if fd i < n && t.(fd i)>t.( !j) then j:=(fd i) ;
  if i<> !j then begin
    echanger t i !j ;
    descendre_noeud t n !j
  end
;;
```

Fonctions de file de priorité. On donne maintenant l'implémentation des deux principales opérations de file de priorité max (ajouter un élément, supprimer le maximum), exceptées celles qui modifient une clé, dont on parlera un peu plus loin.

Pour enfiler un élément, on suppose qu'il y a assez de place dans le tableau, et on rajoute un élément à la suite des autres éléments du tas. On incrémente le nombre d'éléments contenus dans la file, et on appelle `monter_noeud` sur le dernier élément du tas. À l'inverse, pour retirer le maximum (la racine du tas), on choisit de l'échanger avec le dernier élément du tas, en décrémentant le nombre d'éléments contenus dans la file. On appelle `descendre_noeud` sur la nouvelle racine, et on renvoie l'élément supprimé.

```
let enfiler_fp f x=
  f.tab.(f.n) <- x ;
  f.n <- f.n + 1 ;
  monter_noeud f.tab (f.n - 1)
;;

let supprimer_max_fp f=
```

```
  f.n <- f.n - 1 ;
  echanger f.tab 0 f.n ;
  descendre_noeud f.tab f.n 0 ;
  f.tab.(f.n)
;;
```

10.6 Complexité

Ainsi implémentée, la structure de file de priorité possède une complexité $O(\log n)$ pour toutes ses opérations, où n est le nombre d'éléments présents dans la file de priorité, exceptée la création qui est linéaire en la capacité choisie, et le test d'égalité au vide qui s'exécute en temps constant.

10.7 Intermède : le tri par tas

Les fonctions précédemment écrites suffisent à implémenter le fameux « tri par tas » : pour trier un tableau, on crée une file de priorité dans laquelle on rajoute les éléments du tableau un par un, puis on supprime les éléments de la file un par un, ce qui fournit les éléments du tableau dans l'ordre décroissant, que l'on peut donc stocker à la bonne place.

```
let tri_par_tas t=
  let n=Array.length t in
  let f=creer_fp n t.(0) in
  for i=0 to n-1 do
    enfiler_fp f t.(i)
  done ;
  for i=n-1 downto 0 do
    t.(i) <- supprimer_max_fp f
  done ;;
```

La correction est relativement évidente. En terme de complexité, les opérations `enfiler_fp` et `supprimer_max_fp` s'exécutent en temps $O(\log n)$ avec n la taille du tableau (qui est aussi la taille de la capacité de la file de priorité que l'on crée) : on en déduit une complexité totale $O(n \log n)$.

Remarque 10.12. *Il est en fait possible de trier le tableau « en place » avec ces idées : essentiellement on transforme le tableau en tas à l'aide de* `monter_noeud`*, puis on le trie en réécrivant essentiellement le code de la fonction* `supprimer_max_fp`*.*

10.8 Modification d'une clé

Dans une file de priorité, les éléments stockés sont des couples (clé, élément). L'une des opérations possibles de la structure est la modification de la clé associée à un élément présent dans la file. Il faut donc être capable de retrouver le nœud du tas à partir de l'élément. On indique deux solutions.

- Un cas qui se produit souvent est celui où les éléments sont à valeurs dans un ensemble de la forme $[\![0, B-1]\!]$, avec B un entier pas trop grand. On peut alors utiliser un tableau de taille B pour stocker les positions des éléments présents dans la file de priorité dans le tableau associé au tas. Lorsqu'on utilise la fonction `echanger` dans une des fonctions ci-dessus, il ne faut pas oublier de répercuter le changement sur ce tableau des positions. De même, il faut gérer l'insertion et la suppression d'un élément dans la file de priorité avec ce tableau. La complexité des opérations de file de priorité restent alors logarithmique en le nombre d'éléments, de même que la modification de la priorité : il suffit de monter ou descendre l'élément dont on a modifié la priorité. Un exemple où cette approche est très efficace est celle de l'algorithme de Djikstra sur un graphe dont les sommets sont numérotés de 0 à $N-1$.
- Plus généralement, on peut utiliser une structure de dictionnaire dans laquelle on stocke des couples (élément, position dans le tableau), ce qui amène à la section suivante ! La complexité des opérations de file de priorité s'en trouve modifiée suivant la structure choisie, signalons qu'avec une table de hachage elle est quasiment inchangée [2]

2. Bien implémentée, une table de hachage effectue ses opérations en temps constant amorti, sous des hypothèses naturelles de répartition « aléatoire » des clés de hachage.

Chapitre 11

Introduction au Backtracking

11.1 Introduction

Le retour sur trace, ou *backtracking* en anglais, est une méthode de résolution de problème par exploration exhaustive, où l'on interrompt intelligemment le processus dès que les choix effectués précédemment ne pourront mener à une solution. Voici deux exemples de problèmes assez difficiles que l'on peut résoudre par backtracking :

- le problème bien connu du Sudoku, que l'on prendra en exemple dans ce cours, demande de compléter une grille comme la suivante, avec des chiffres de 1 à 9, de sorte que chaque ligne, chaque colonne et chaque petit carré de taille 3×3 contiennent exactement chaque chiffre de 1 à 9 une et une seule fois, voir figure 11.1. Les grilles présentes dans le commerce possède une unique solution, on peut généraliser à des entrées où les grilles possèdent plusieurs solutions ou aucune. Le problème se généralise également à des grilles de tailles $n^2 \times n^2$ et les chiffres de $[\![1, n^2]\!]$.
- le problème des 8 dames consiste à placer 8 dames sur un échiquier, de sorte que deux d'entre-elles ne soient jamais en prise, c'est-à-dire sur la même ligne, même colonne, ou même diagonale. La figure 11.2 montre un exemple de solution, il y en a 92 au total. Le problème se généralise à des échiquiers $n \times n$, le nombre de solutions n'est connu exactement que jusqu'à $n = 27$ (et vaut 234 907 967 154 122 528). Signalons toutefois l'avancée spectaculaire réalisée par Michael Simkin en 2021, qui a montré qu'asymptotiquement, le nombre de solutions est de l'ordre de $(0.143 \times n)^n$, voir ici `https://en.wikipedia.org/wiki/Eight_queens_puzzle`.

	9		2			6		5
3	2				7			
	7		9		5			8
	1							
		7					9	4
6								
		8						7
	3		4	9	1	5		
					3			

8	9	1	2	3	4	6	7	5
3	2	5	6	8	7	4	1	9
4	7	6	9	1	5	3	2	8
9	1	4	7	5	2	8	6	3
2	5	7	3	6	8	1	9	4
6	8	3	1	4	9	7	5	2
1	4	8	5	2	6	9	3	7
7	3	2	4	9	1	5	8	6
5	6	9	8	7	3	2	4	1

FIGURE 11.1 – Une grille de sudoku et son unique solution

11.2 Modélisation par un arbre et algorithme générique

Concrètement, le problème à résoudre se résume à explorer un ensemble de possibilités E. Par exemple pour le problème du sudoku, E est l'ensemble des grilles obtenues en choisissant un chiffre de $[\![1, 9]\!]$ pour chaque case vide de la grille, de cardinal 9^v.

L'algorithme de Backtracking consiste à effectuer des choix successifs pour parcourir l'ensemble E. Ces choix vont restreindre le sous-ensemble de E étudié en introduisant des contraintes. Les choix effectués peuvent se représenter comme un arbre, où effectuer un choix consiste à choisir un fils d'un nœud. L'exploration dans l'arbre se poursuit jusqu'à ce qu'on aboutisse à une feuille (un seul élément de E, plus de choix possibles) :

— si cette feuille est une solution, on peut arrêter l'algorithme ;

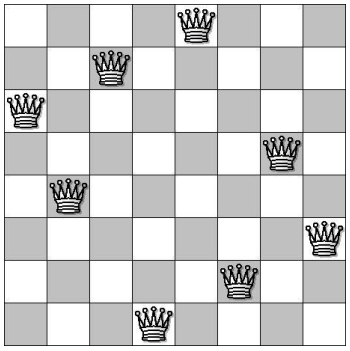


FIGURE 11.2 – Une solution au problème des 8 dames

— sinon, les choix effectués précédemment ne conviennent pas, il faut donc revenir en arrière et effectuer d'autres choix.

Par exemple, pour la résolution du Sudoku, on peut parcourir celui-ci de haut en bas puis de gauche à droite. Les règles imposent que l'on ne peut écrire en première case (en haut à gauche) qu'un chiffre de l'ensemble $\{1, 4, 8\}$. Choisissons 1. Il n'y a pas de choix à effectuer sur la deuxième case déja préremplie. Sur la troisième, on peut écrire uniquement 4 en suivant les règles, etc... (voir figure 11.3).

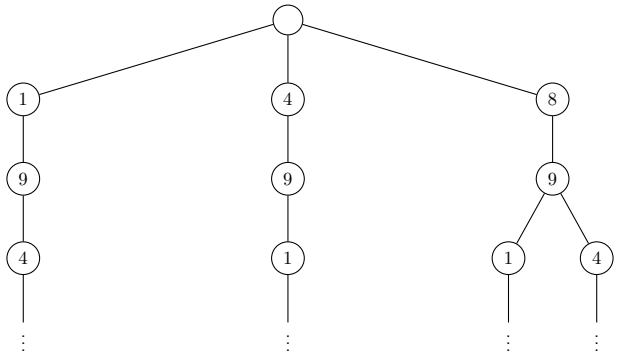


FIGURE 11.3 – L'arbre associé à l'exploration du sudoku par Backtracking. Chaque niveau indique un choix effectué sur une case.

L'algorithme 11.1 présente un algorithme générique de Backtracking, consistant à explorer cet arbre avec un parcours préfixe. Cette exploration est naturellement gérée par récursivité. Dès qu'une solution est trouvée, l'algorithme s'arrête.

Algorithme 11.1 : *Backtrack*(n) : Résolution générique par Backtracking

```
Entrée : Un nœud n de l'arbre (élément partiellement construit de l'ensemble E du problème)
Sortie  : Un booléen indiquant si on peut compléter n en une solution
si n est une feuille alors
    Renvoyer un booléen indiquant si n est solution
sinon
    pour tout fils f de n faire
        si Backtrack(f) alors
            Renvoyer Vrai
    Renvoyer Faux
```

11.3 Sudoku : résolution pratique

Représentation d'une grille. On représente une grille de Sudoku comme une matrice (tableau de 9 tableaux à 9 éléments). Chaque entrée de la grille est représentée par son chiffre, une entrée vide est représentée par zéro. Voici donc la grille initiale de la figure 11.1.

```
let grille_initiale =
[|[|0; 9; 0; 2; 0; 0; 6; 0; 5|];
  [|3; 2; 0; 0; 0; 7; 0; 0; 0|];
  [|0; 7; 0; 9; 0; 5; 0; 0; 8|];
  [|0; 1; 0; 0; 0; 0; 0; 0; 0|];
  [|0; 0; 7; 0; 0; 0; 0; 9; 4|];
  [|6; 0; 0; 0; 0; 0; 0; 0; 0|];
  [|0; 0; 8; 0; 0; 0; 0; 0; 7|];
  [|0; 3; 0; 4; 9; 1; 5; 0; 0|];
  [|0; 0; 0; 0; 0; 3; 0; 0; 0|]|] ;;
```

Case suivante. Pour remplir la grille, on va parcourir celle-ci de haut en bas puis de gauche à droite. La fonction suivante permet de passer d'une case (couple d'indices de $[\![0,8]\!]$) à la prochaine (celle située à droite si elle existe, sinon celle en dessous). On remarque que sur la dernière case $(8,8)$, la fonction renvoie $(9,0)$, ce qui sera utile pour marquer la fin de l'exploration.

```
let case_suivante (a,b) =
  if b=8 then a+1, 0 else a, b+1
;;
```

Valeurs interdites. Considérons une case (i,j) vide de la grille t (elle contient 0). Les règles imposent qu'on ne peut écrir en case (i,j) de t aucune valeur qui apparaît déja sur la ligne numéro i. Voici comment récupérer la liste de ces valeurs interdites :

```
let interdites_ligne t i j =
  let q=ref [] in
  for j2=0 to 8 do
    if t.(i).(j2) > 0 then q:=t.(i).(j2) :: !q
  done ;
  !q
;;
```

On laisse au lecteur le soin d'écrire également `interdites_colonne` et `interdites_bloc`, prenant les mêmes arguments et renvoyant respectivement la liste des entrées déja présentes sur la colonne j et dans le bloc auquel appartient la case (i,j). Par exemple avec la case $(4,1)$, qui est la case centrale du bloc situé au milieu, à gauche.

```
# interdites_ligne grille_initiale 4 1 ;;
- : int list = [4; 9; 7]
# interdites_colonne grille_initiale 4 1 ;;
- : int list = [3; 1; 7; 2; 9]
# interdites_bloc grille_initiale 4 1 ;;
- : int list = [6; 7; 1]
```

Valeurs permises. À l'aide de ces fonctions, on peut écrire une fonction renvoyant la liste (croissante) des valeurs que l'on peut écrire en case (i,j), en respectant les règles, en fonction des valeurs déja écrites (entrées initiales et essais précédents) :

```
let valeurs_permises t i j =
  let q=interdites_ligne t i j @ interdites_colonne t i j @ interdites_bloc t i j in
  let perm = ref [] in
  for i=9 downto 1 do
    if not (List.mem i q) then perm := i :: !perm
  done ;
  !perm
;;
```

Voici le résultat sur la grille prise en exemple :

```
# valeurs_permises grille_initiale 4 1 ;;
- : int list = [5; 8]
# valeurs_permises grille_initiale 0 0 ;;
- : int list = [1; 4; 8]
```

Résolution par backtracking. On a tout le nécessaire pour implémenter l'algorithme 11.1 dans ce contexte. On préfère modifier la grille directement plutôt que de procéder à de nombreuses recopies inutiles et coûteuses.

```
let rec backtrack_sudoku t i j =
  if i=9 (* grille remplie ! *) then
    true
  else begin
      let a,b = case_suivante (i,j) in
      if t.(i).(j) > 0 then (* (i,j) donnee initiale *)
        backtrack_sudoku t a b
      else (* essai des valeurs autorisées *)
        let rec parcours_valeurs perm = match perm with
          | [] ->  t.(i).(j) <- 0 ;  false
          | x::r -> (t.(i).(j) <- x ; backtrack_sudoku t a b) || parcours_valeurs r
        in parcours_valeurs (valeurs_permises t i j)
    end
;;
```

Attention : dans le cas où on se trouve dans une branche qui ne mène pas à une solution : une fois toutes les valeurs possibles testées dans la case (i, j), il faut prendre garde à réécrire 0 en case (i, j) avant de terminer l'appel courant.

Voici le résultat de l'appel sur la grille prise en exemple et la case $(0, 0)$:

```
# backtrack_sudoku grille_initiale 0 0 ;;
- : bool = true
```

Ceci montre qu'une solution a été détectée. Voici la valeur de `grille_initiale` après l'appel :

```
# grille_initiale ;;
- : int array array =
[|[|8; 9; 1; 2; 3; 4; 6; 7; 5|];
  [|3; 2; 5; 6; 8; 7; 4; 1; 9|];
  [|4; 7; 6; 9; 1; 5; 3; 2; 8|];
  [|9; 1; 4; 7; 5; 2; 8; 6; 3|];
  [|2; 5; 7; 3; 6; 8; 1; 9; 4|];
  [|6; 8; 3; 1; 4; 9; 7; 5; 2|];
  [|1; 4; 8; 5; 2; 6; 9; 3; 7|];
  [|7; 3; 2; 4; 9; 1; 5; 8; 6|];
  [|5; 6; 9; 8; 7; 3; 2; 4; 1|]|]
```

Le sudoku est résolu !

Chapitre 12

Graphes non pondérés : compléments et rappels

12.1 Introduction

L'intérêt pour les graphes remonte au 18ème siècle d'un point de vue mathématiques. L'exemple historique est le problème qu'Euler s'était posé dans la ville de Königsberg (aujourd'hui Kaliningrad). Peut-on partir d'un point de la ville, faire une promenade en passant par tous les ponts une seule fois ? La réponse est non. On peut montrer relativement facilement qu'un parcours de graphe *eulérien* (qui passe par toutes les arêtes, une seule fois) est possible si et seulement si le nombre de sommets de degré impair est 0 ou 2, ce qui n'est pas le cas ici.

FIGURE 12.1 – Modélisation du problème des ponts de Königsberg par des graphes. Les images proviennent de Wikipédia

Aujourd'hui, d'un point de vue pratique, les graphes sont présents partout :

- réseaux routiers ;
- réseaux de distribution d'eau, électricité ;
- Web et liens entre les pages ;
- Facebook et autres réseaux sociaux ;
- ...

Dans ce cours, on va s'intéresser aux questions algorithmiques, comme les suivantes.

- Comment parcourir un graphe ?
- Quelle est la plus courte distance entre 2 sommets d'un graphe ?

Mais on ne s'intéressera pas aux questions mathématiques.

- Combien de couleurs sont nécessaires pour colorer un graphe planaire sans que deux sommets adjacents aient la même couleur ?
- Sous quelles conditions un graphe est-il plongeable (c'est-à-dire qu'il peut être représenté sans croisement) dans le plan (ou la sphère, c'est le même chose), le tore, etc... ?

12.2 Vocabulaire des graphes et propriétés mathématiques

12.2.1 Graphes non orientés

Définition 12.1. *Un graphe est un couple (V, E) tel que :*

- *V est un ensemble fini non vide, ses éléments sont appelés* sommets *ou* nœuds.
- *E est un sous-ensemble de $\mathcal{P}_2(V)$, les parties à 2 éléments distincts de V. Les éléments de E sont appelés les* arêtes *du graphe.*

On utilise la notation anglo-saxonne : V pour vertices (sommets) et E pour edges (arêtes).

Remarque 12.2. *Puisque $E \subseteq V$, un graphe à n sommets a au plus $\binom{n}{2} = \frac{n(n-1)}{2}$ arêtes.*

Remarque 12.3. *Les graphes au programme ne présentent ni boucles (une arête reliant un sommet à lui-même), ni multi-arêtes (plusieurs arêtes possibles entre deux sommets donnés). Le graphe associé au problème des ponts de Königsberg présente des multi-arêtes.*

Incidence et degré. On dit :

- que deux sommets v et w sont *adjacents* (ou *voisins*) s'ils sont reliés par une arête (c'est-à-dire $\{v, w\} \in E$) ;
- qu'une arête est *incidente* aux sommets qu'elle relie.

Le nombre de sommets $|V|$ du graphe s'appelle l'*ordre du graphe*, que l'on notera en général n dans ce cours. Le *degré* d'un sommet v, noté $\deg(v)$ est le nombre d'arêtes qui lui sont incidentes.

Chemins et cycles. Un chemin de longueur p dans le graphe est une suite de $p+1$ sommets $v_0, v_1, \dots, v_p$, telle que $\{v_i, v_{i+1}\} \in E$ pour tout $0 \leq i \leq p-1$.

Définition 12.4. *Un cycle de G est un chemin $v_0, v_1, \dots, v_p$ de longueur au moins trois, tel que $v_0 = v_p$ et les sommets $(v_0, \dots, v_{p-1})$ sont distincts deux à deux*[1].

Définition 12.5. *Un graphe est dit acyclique s'il ne possède pas de cycle.*

On va caractériser par la suite les graphes acycliques : ce sont des forêts, c'est-à-dire des unions d'arbres (au sens des graphes, définis ci-après).

Graphe induit et composantes connexes. Il est souvent utile de restreindre un graphe à un sous-ensemble de sommets, ce qui mène à la définition suivante.

Définition 12.6. *Soit $G = (V, E)$ un graphe, et $S \subseteq V$ un ensemble de sommets non vide du graphe. Le sous-graphe de G induit par S est le graphe (S, A) avec $A = E \cap \mathcal{P}_2(S)$.*

Un graphe dans lequel il existe deux sommets non reliés par un chemin se décompose de manière non triviale en sous-graphes, ce que l'on formalise avec les propriétés et définitions qui suivent.

Proposition 12.7. *Dans $G = (V, E)$, la relation $u\mathcal{R}v$ donnée par « il existe un chemin entre u et v » est une relation d'équivalence sur V.*

Démonstration.
- La relation est réflexive : $u\mathcal{R}u$, car u est un chemin de longueur 0 ;
- La relation est symétrique : si $u = v_0, v_1, \dots, v_p = v$ est un chemin de u à v, alors $v_p, v_{p-1}, \dots, v_0$ est un chemin de v à u ;
- La relation est transitive : si $u = v_0, v_1, \dots, v_p = v$ et $v = w_0, \dots, w_q = w$ sont des chemins de u à v et de v à w, on obtient un chemin de u à w en concaténant ces deux chemins.

□

Définition 12.8. *Les composantes connexes du graphe sont les classes d'équivalence pour cette relation.*

Remarque 12.9. *Dans l'appelation « composante connexe », on confondera souvent l'ensemble des sommets et le sous-graphe induit par cet ensemble.*

Exemple 12.10. *En figure 12.2 est illustré un graphe à trois composantes connexes.*

1. Ce n'est pas forcément standard. Certains auteurs se contentent de $v_0 = v_p$. Mais pour parler de graphes acyliques comme dans la suite, il faut éliminer les cycles « triviaux », comme par exemple un chemin parcouru dans les deux sens, ce qui est assez pénible à écrire convenablement. Une autre définition possible est celle d'un chemin constitué d'arêtes distinctes dont les extrémités coïncident.

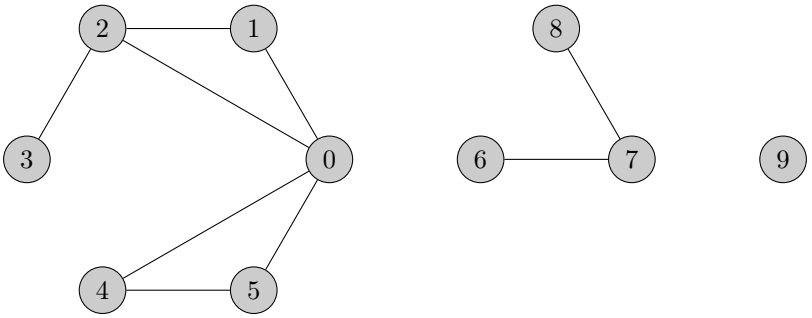


FIGURE 12.2 – Un graphe non orienté à 3 composantes connexes

Définition 12.11. *Un graphe est dit connexe s'il ne possède qu'une seule composante connexe. Autrement dit, pour tout couple de sommets, il existe un chemin entre ces deux sommets.*

Exemple 12.12. *Voici deux graphes « classiques » sur* $[\![0, n-1]\!]$ *(voir figure 12.3) :*

- *le cycle* C_n*, dont les arêtes sont* $\{i, i+1\}$ *pour tout* $i \in [\![0, n-2]\!]$ *ainsi que* $\{0, n-1\}$ *;*
- *la clique* K_n*, constituée des* $\binom{n}{2}$ *arêtes possibles.*

FIGURE 12.3 – Le cycle C_6 et la clique K_6

12.2.2 Arbres au sens des graphes non orientés.

Un arbre peut être vu comme un cas particulier de graphe non orienté. Le but de cette sous-section est de caractériser les arbres parmi les graphes : ce sont les graphes connexes acycliques.

Proposition 12.13. *Soit* $G = (V, E)$ *un graphe, on a* $\sum_{v \in V} \deg(v) = 2\,|E|$.

Démonstration. On va effectuer un double comptage : on note $\mathbb{1}_{v,e}$ le symbole d'incidence de l'arête e au sommet v, c'est-à-dire :

$$\mathbb{1}_{v,e} = \begin{cases} 1 & \text{si } e \text{ incidente à } v \\ 0 & \text{sinon.} \end{cases}$$

On a alors d'une part $\sum_{v \in V} \sum_{e \in E} \mathbb{1}_{v,e} = \sum_{v \in V} \deg(v)$. D'autre part, $\sum_{e \in E} \sum_{v \in V} \mathbb{1}_{v,e} = 2\,|E|$. □

Proposition 12.14. *Un graphe connexe d'ordre* n *possède au moins* $n-1$ *arêtes.*

Démonstration. La preuve se fait par récurrence sur n.

- si $n = 1$, c'est trivial.
- considérons $n \geq 2$ et la propriété démontrée pour l'entier $n-1$. Soit $G = (V, E)$ un graphe d'ordre n, connexe. Alors tout sommet est de degré non nul.
  - ∗ S'il existe un sommet v de degré 1, alors le graphe induit par $V \backslash \{v\}$ est toujours connexe [2] et possède $n-1$ sommets. Par hypothèse de récurrence, il possède au moins $n-2$ arêtes. Donc le graphe G a au moins $n-1$ arêtes.
  - ∗ Sinon, tout sommet est de degré au moins 2, et le graphe possède $|E| = \frac{1}{2} \sum_{v \in V} \deg(v) \geq n$ arêtes.

2. Pourquoi ?

— Par principe de récurrence, la propriété est démontrée.

□

Corollaire 12.15. *Déterminer le maximum d'une liste par comparaisons requiert au moins $n-1$ comparaisons avec n la taille de la liste.*

On va maintenant parler des graphes qui possèdent des cycles, dans le but de caractériser les graphes acycliques.

Proposition 12.16. *Soit G un graphe tel que le degré de chaque sommet soit au moins 2. Alors G possède un cycle.*

Démonstration. On construit une suite de sommets dans le graphe en partant d'un sommet quelconque v_0, un voisin v_1, et en posant pour tout $i \geq 1$, v_{i+1} un voisin de v_i qui n'est pas v_{i-1}. Cette construction est possible car le degré de chaque sommet est au moins 2. Puisque V est fini, un sommet v apparaît au moins deux fois dans la suite $(v_i)_{i\in\mathbb{N}}$, et supposons que ce soit le premier à apparaître pour la deuxième fois. Notons alors i l'indice de sa première occurence, et j l'indice de la deuxième. Alors le chemin $v_i, v_{i+1}, \ldots, v_j = v_i$ ne contient que des sommets distincts : c'est un cycle car $j > i+2$ par construction. □

Proposition 12.17. *Un graphe acyclique d'ordre n possède au plus $n-1$ arêtes.*

Démonstration. La démonstration se fait également par récurrence sur $n \geq 1$.

— Si $n = 1$, il n'y a rien à montrer.

— Supposons la propriété vraie jusqu'au rang n non inclus, et considérons un graphe acyclique d'ordre n. Alors G possède un sommet v de degré 0 ou 1 (car sinon, il aurait un cycle d'après la proposition précédente). On applique l'hypothèse de récurrence au graphe induit par $V\backslash\{v\}$ (évidemment acyclique !), qui est d'ordre $n-1$ et possède donc au plus $n-2$ arêtes. Ainsi G a bien au plus $n-1$ arêtes.

— Par principe de récurrence, la propriété est bien démontrée.

□

Proposition 12.18. *Soit G un graphe d'ordre n. Alors les propriétés suivantes sont équivalentes :*

(1) il est acyclique et connexe ;

(2) il est acyclique et a n-1 arêtes ;

(3) il est connexe et a n-1 arêtes ;

Démonstration.

(1) ⇒ (2), (3) Puisque le graphe est connexe, il possède au moins $n-1$ arêtes. Puisqu'il est acyclique, il en possède au plus $n-1$;

(3) ⇒ (1) Supposons que $G = (V, E)$ possède un cycle $v_0, \ldots, v_p$. Tout chemin dans le graphe passant par l'arête $\{v_0, v_1\}$ peut être transformé en un chemin dont les arêtes sont dans $E\backslash\{\{v_0, v_1\}\}$, en remplaçant l'arête $\{v_0, v_1\}$ par le chemin $v_0 = v_p, \ldots, v_1$. Ainsi, supprimer $\{v_0, v_1\}$ du graphe ne change pas sa connexité, ce qui est absurde car avec $n-2$ arêtes le graphe ne saurait être connexe.

(2) ⇒ (1) Notons r le nombre de composantes connexes de G, et $n_1, \ldots, n_r$ le nombre de sommets de chaque composante. Chaque composante étant acyclique et connexe, elle possède $n_i - 1$ arêtes. On obtient donc au total $\sum_{i=1}^{r}(n_i - 1) = n - r$ arêtes, donc $r = 1$ et G est connexe.

□

Définition 12.19. *Un arbre (au sens des graphes non orientés) est un graphe vérifiant l'une des conditions équivalentes précédentes.*

Remarque 12.20. *Parmi les graphes, les arbres sont donc les graphes acycliques maximaux (rajouter une arête crée un cycle) et connexe minimaux (enlever une arête fait perdre la connexité). La proposition précédente montre que les composantes connexes d'un graphe acyclique sont des arbres, un graphe acyclique est donc également appelé une* forêt.

12.2.3 Graphes orientés

Les graphes orientés sont des graphes où les arêtes ont un sens. La plupart des définitions s'adaptent à ce cadre, à quelques modifications près.

Définition 12.21. *Un graphe orienté est un couple $G = (V, E)$ où E est un ensemble de couples d'éléments distincts de V. Les éléments de E sont alors appelés* arcs *(au lieu d'arêtes).*

Remarque 12.22. *Un graphe d'ordre n possède donc au plus $n(n-1)$ arcs.*

Les arcs sont alors représentés comme des flèches entre deux sommets.

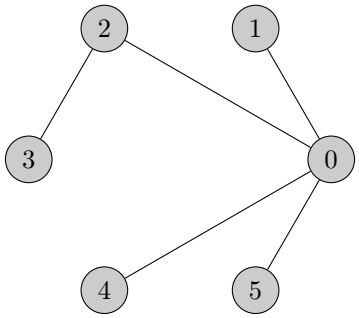


FIGURE 12.4 – Un exemple d'arbre (au sens des graphes non orientés)

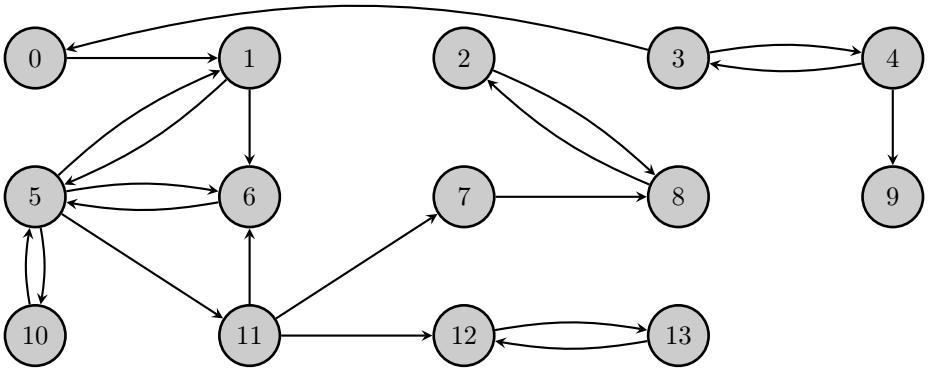


FIGURE 12.5 – Un graphe orienté

Degré entant et sortant. Puisque les arcs ont une orientation, on ne parle plus du degré d'un sommet v, mais de son degré entrant (nombre d'arcs de la forme (u, v)) et de son degré sortant (nombre d'arcs de la forme (v, u)). La formule reliant le nombre d'arêtes aux degrés des sommets s'écrit maintenant

$$2\,|E| = \sum_{v \in V} (\deg_e(v) + \deg_s(v))$$

avec $\deg_e(v)$ et $\deg_s(v)$ les degrés entrant et sortant de v.

Chemin et circuit. Dans un graphe orienté, une suite $v_0, v_1, \ldots, v_p$ telle que $(v_i, v_{i+1}) \in E$ est également appelée un chemin. On appelle circuit un chemin non réduit à un sommet, dont les sommets aux extrémités sont les mêmes.

Composantes fortement connexes. La relation $\mathcal{R}$ que l'on avait défini sur les graphes non orientés n'est plus symétrique dans ce cadre. On remplace cette relation par la suivante : $u\mathcal{R}'v$ si il existe un chemin de u à v et un chemin de v à u, qui est bien une relation d'équivalence.

Définition 12.23. *Les classes d'équivalence pour cette relation se nomment les composantes fortement connexes du graphe.*

Proposition 12.24. *Les composantes fortement connexes d'un graphe orienté sans circuit sont réduites à des singletons.*

Démonstration. Si $u \neq v$ étaient dans la même composante fortement connexe d'un graphe G sans circuit, alors on pourrait constuire un circuit en concaténant un chemin de u à v à un chemin de v à u, ce qui est absurde. □

Il est intéressant de voir qu'on peut munir l'ensemble des composantes fortement connexes d'un graphe orienté d'une structure de graphe sans circuit.

Proposition 12.25. *Soit $G = (V, E)$ un graphe orienté. Notons $\mathcal{C}$ l'ensemble des composantes fortement connexes, et considérons $G^{CFC} = (\mathcal{C}, \mathcal{E})$ avec $\mathcal{E}$ l'ensemble de couples de composantes fortement connexes distinctes vérifiant : $(C_1, C_2) \in \mathcal{E}$ si et seulement si il existe un arc entre un sommet de C_1 et un sommet de C_2. Alors G^{CFC} est un graphe orienté acylique.*

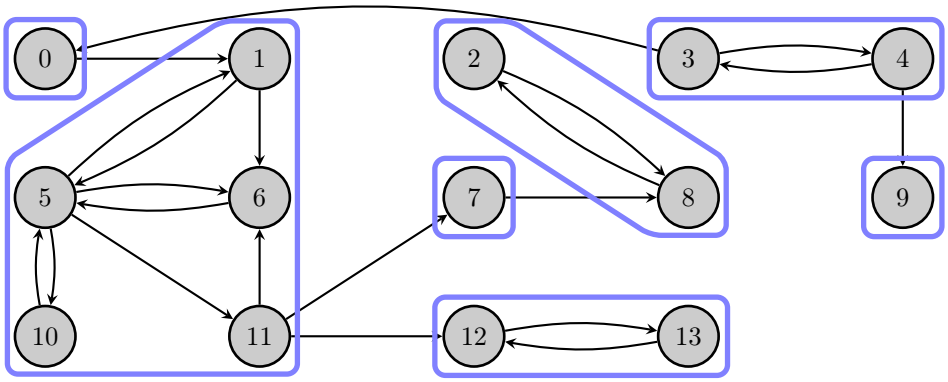


FIGURE 12.6 – Les composantes fortement connexes du graphe ci-dessus

Démonstration. Supposons l'existence d'un circuit dans G^{CFC}. Alors deux éléments C_1 et C_2 distincts du circuit sont dans la même composante connexe de G^{CFC}. Par définition, il existe v_1, v_1' dans C_1 et v_2, v_2' dans C_2, tels que dans G il y a un chemin de v_1 à v_2 et un autre de v_2' à v_1'. Par définition des composantes connexes, il y a aussi un chemin de v_1' à v_1 et un autre de v_2 à v_2' dans G. Par suite, v_1, v_1', v_2, v_2' sont sur un même circuit dans G, donc dans la même composante fortement connexe : c'est absurde. □

Exemple 12.26. *En figure 12.7 est représenté le graphe des composantes fortement connexes du graphe précédent.*

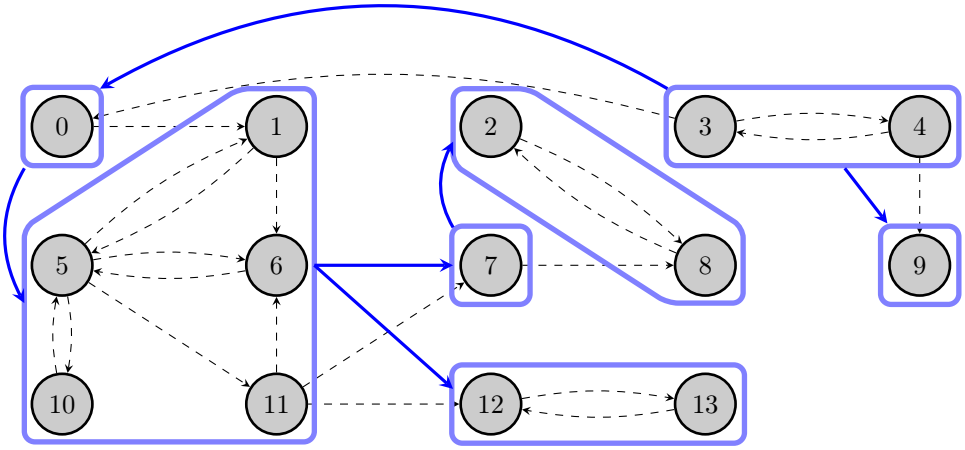


FIGURE 12.7 – Graphe sans circuit des composantes fortement connexes

Arbre au sens des graphes orientés. Le travail fait sur les graphes non orientés montre que dans un arbre, il y a essentiellement un seul chemin d'un sommet vers un autre (sinon on pourrait créer un cycle). Ainsi, le choix d'un sommet particulier de l'arbre (on parle d'*enracinement* de l'arbre) permet d'orienter sans ambiguïté les arêtes pour en faire un graphe orienté.

Remarque 12.27. *Après enracinement, on retrouve les arbres définis « mathématiquement », c'est-à-dire comme un ensemble muni d'une relation de parenté vérifiant les propriétés idoines. Ils sont toutefois différents des arbres usuellement manipulés en informatique, car dans ce cas les fils d'un nœud sont ordonnés.*

Voici pour terminer quelques exemples de graphes orientés :

— réseaux routiers (il y a des routes à sens unique) ;

— (Humanité, $\{(h_1, h_2) \mid h_1 \text{ connaît } h_2\}$) : nous sommes nombreux à connaître Alain Chabat, mais *a priori* il ne nous connaît pas !

— graphe divisoriel sur $[\![0, n-1]\!]$: $i \to j$ si $i|j$ et $i \neq j$.

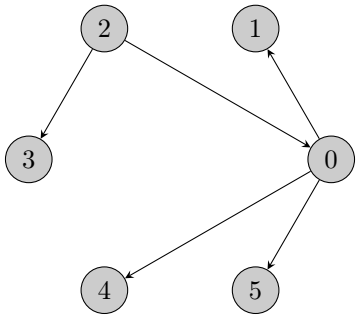


FIGURE 12.8 – L'arbre vu plus haut, orienté par enracinement du sommet 2

12.3 Implémentation des graphes

Le programme officiel impose d'écrire une implémentation qui permet la suppression de sommets. On commence donc par une implémentation très générale, avant de se restreindre à des implémentations plus classiques où les sommets sont fixés : ce sont les entiers de $[\![0, n-1]\!]$.

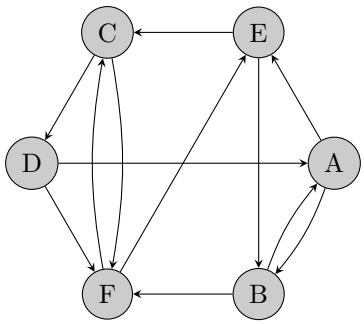
Une petite remarque avant de débuter : il est un peu plus difficile de manipuler des graphes non orientés que des graphes orientés, car il faut faire attention à maintenir la non orientation. Comme il n'est pas facile de manipuler des ensembles à deux éléments, l'information « $\{u, v\}$ est une arête du graphe » est dupliquée sous la forme « u est voisin de v, v est voisin de u ». Lorsqu'on rajoute l'arête $\{u, v\}$, il faut en pratique rajouter v aux voisins de u et u aux voisins de v, et inversement lorsqu'on la supprime.

12.3.1 Implémentation « générale »

Dans cette implémentation, on décrit simplement un graphe par la liste de ses sommets, et de leurs voisins. Ceci mène aux types suivants (persistants) :

```
type 'a sommet = {id: 'a ; voisins: 'a list} ;;
type 'a graphe_gen = 'a sommet list ;;
```

Ni la liste des voisins d'un sommet, ni la liste des sommets ne sont supposées ordonnées.



Par exemple, le graphe ordonné de la figure précédente peut être implémenté comme suit.

```
[{id = "a"; voisins = ["b"; "e"]}; {id = "b"; voisins = ["a"; "f"]};
 {id = "c"; voisins = ["d"]}; {id = "d"; voisins = ["a"; "f"]};
 {id = "e"; voisins = ["b"; "c"]}; {id = "f"; voisins = ["c"; "e"]}]
```

Voici les fonctions d'ajout/suppression d'arcs et d'arêtes et de sommets que l'on peut écrire sur un tel graphe :

Ajout/suppression d'arcs/d'arêtes. On donne à chaque fois deux fonctions, la deuxième est valable pour un graphe non orienté. Pour ajouter l'arête (u, v), il faut d'abord trouver le sommet d'index u et rajouter v dans sa liste d'adjacence. Pour la suppression, on utilise `suppr` permettant de retirer si elle existe la première occurence d'un élément dans une liste.

```
let rec ajoute_arc g u v=match g with
    | [] -> failwith "u non present"
    | x::q when x.id = u && List.mem v x.voisins -> g
    | x::q when x.id = u -> {id=u ; voisins=v::x.voisins}::q
    | x::q -> x::(ajoute_arc q u v)
;;

let ajout_arete g u v=ajoute_arc (ajoute_arc g u v) v u ;;

let rec suppr x q=match q with
  | [] -> []
  | y::p when y=x -> p
  | y::p -> y::(suppr x p)
;;

let rec supprime_arc g u v=match g with
    | [] -> failwith "u non present"
    | x::q when x.id = u -> {id=u ; voisins= (suppr v x.voisins)}::q
    | x::q -> x::(supprime_arc q u v)
;;

let supprime_arete g u v=supprime_arc (supprime_arc g u v) v u ;;
```

Ajout/suppression de sommets. Ajouter un sommet est facile, il est initialement sans voisins. Supprimer un sommet est plus délicat qu'une arête car il faut supprimer le sommet de la liste, mais également toutes les apparitions du sommet dans les listes d'adjacences des autres éléments. Par contre, il n'y a pas de distinction à faire entre les cas orienté et non orienté.

```
(* est_present teste si l'étiquette u est déja présente *)
let est_present g u = List.mem u (List.map (function x -> x.id) g) ;;

let ajoute_sommet g u= if est_present g u then g else {id=u ; voisins = [] }::g ;;

let rec supprime_sommet g u=match g with
  | [] -> []
  | x::q when x.id=u -> supprime_sommet q u
  | x::q -> ({id=x.id ; voisins=suppr u x.voisins})::(supprime_sommet q u)
;;
```

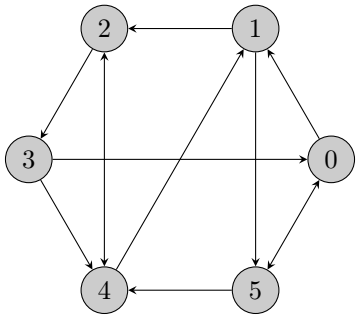
Un mot sur l'efficacité. Une telle implémentation des graphes n'est pas très efficace, car il faut parcourir toute la liste des sommets pour trouver celui qui nous intéresse. Ceci dit, ce type d'implémentation où les sommets sont dynamiques est très utile lorsqu'on travaille sur un graphe dont l'ensemble des sommets n'est pas statique. C'est par exemple le cas lorsqu'on explore une petite partie d'un très gros graphe en partant d'un sommet. Initialement, on va travailler avec le sous-graphe contenant uniquement le sommet de départ, puis on va faire grossir le sous-graphe et s'arrêter dès que nécessaire. Une application serait par exemple de chercher le plus court chemin entre vous et Barack Obama dans Facebook : hors de question de charger en mémoire tout le graphe de Facebook, alors que le chemin entre vous et Barack Obama est probablement inférieur à 6 ![3] Une meilleure implémentation consiste à faire usage de tables de hachages (ou plus généralement d'un dictionnaire) à la place d'une liste pour stocker les voisins d'un sommet : retrouver un sommet se fait alors en temps constant en moyenne, sous certaines hypothèses naturelles.

12.3.2 Implémentation « creuse »

Dorénavant, on suppose que l'ensemble des sommets est fixé, et l'on suppose que c'est $[\![0, n-1]\!]$. On propose une autre implémentation où les voisins d'un sommet sont stockés dans une liste, mais on utilise un tableau pour stocker les listes d'ajdacence, ce qui permet d'accéder à la liste d'adjacence d'un sommet donné en temps constant. Le type est donc le suivant :

```
type graphe_creux = int list array ;;
```

3. Oui, c'est petit ! On pourra consulter `https://fr.wikipedia.org/wiki/Six_degr%C3%A9s_de_s%C3%A9paration` pour en savoir plus. En partant de ce principe, la meilleure approche en terme de mémoire et de calculer les gens qui sont à distance au plus 3 de vous, et ceux à distance au plus 3 de Barack Obama, et de chercher un point commun.



Par exemple, le graphe ordonné de la figure précédente peut être implémenté comme suit.

```
let g=[| [1; 5]; [2; 5]; [3]; [0; 4]; [1; 2]; [0; 4] |] ;;
```

La liste d'adjacence du sommet i est simplement donnée par `g.(i)`. Par contre les `g.(i)` ne sont pas supposées ordonnées en général. Il est très facile de gérer l'ajout/suppression d'un arc dans un graphe représenté ainsi, on pourrait ainsi écrire des fonctions d'ajout/suppression d'arcs/d'arêtes, très similaires aux précédentes.

Écrivons une fonction de *désorientation* du graphe, qui prend en entrée un graphe orienté et qui le modifie pour rajouter si nécessaire les arcs (v, u) correspondant aux arcs (u, v) présents.

```
let desoriente g=
  let n=Array.length g in
  let ajoute g i j=if not (List.mem j g.(i)) then g.(i) <- j::g.(i) in
  let rec aux i q=match q with
    | [] -> ()
    | j::p -> ajoute g j i ; aux i p
  in
  for i=0 to n-1 do
    aux i g.(i)
  done ;;
```

Remarque : si l'on ne souhaite pas modifier le graphe, on peut le copier simplement avec `Array.copy`.

Complexités. Cette représentation est également économique, car elle nécessite un espace mémoire en $O(|V|+|E|)$. Parcourir les voisins d'un sommet se fait en temps linéaire en son degré, par contre il faut parcourir une liste d'adjacence pour tester l'existence d'un arc ou une arête. Là encore pour cette opération, utiliser des tables de hachage serait préférable.

12.3.3 Représentation « dense »

Une manière de représenter un graphe dont les sommets sont $[\![0, n-1]\!]$ est d'utiliser une matrice pour indiquer l'existence d'un arc : la matrice d'adjacence.

Définition 12.28. *La matrice d'un adjacence d'un graphe* $([\![0, n-1]\!], E)$ *est la matrice* $M = (m_{i,j})_{0\leq i,j\leq n-1}$ *définie par* $m_{i,j} = \begin{cases} 1 & \text{si } (i,j) \in E. \\ 0 & \text{sinon.} \end{cases}$

Par exemple, la matrice du graphe précédent est :

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

La diagonale de la matrice est constituée de zéros (il n'y a pas de boucles dans nos graphes). Pour un graphe non orienté, la matrice est symétrique. En représentant ainsi un graphe en Caml, on obtient le type :

```
type graphe_dense = int array array ;;
```

La matrice précédente est donnée par :

```
let m=[| [|0; 1; 0; 0; 0; 1|]; [|0; 0; 1; 0; 0; 1|]; [|0; 0; 0; 1; 1; 0|]; [|1; 0; 0; 0; 1; 0|];
[|0; 1; 1; 0; 0; 0|]; [|1; 0; 0; 0; 1; 0|] |] ;;
```

Écrivons une fonction de *désorientation* du graphe, similaire à la précédente mais sur des graphes représentés de manière dense : il suffit de « symétriser » la matrice d'adjacence.

```
let desoriente m=
  let n=Array.length m in
  for i=0 to n-1 do
    for j=0 to n-1 do
      m.(i).(j) <- max m.(i).(j) m.(j).(i)
    done
  done
;;
```

Là encore, si on ne souhaite pas modifier le graphe, on peut faire une copie. Comme il est nécessaire de copier les éléments de `m`, (tableau de tableaux), on propose la fonction suivante :

```
let copie m =
  Array.map Array.copy m
;;
```

avec `Array.map` de type `('a -> 'b) -> 'a array -> 'b array`, appliquant une même fonction à tous les éléments d'un tableau pour obtenir un nouveau tableau. Évidemment on pouvait faire une copie à la main en recréant une matrice.

Complexités. Cette représentation n'est pas très économique en mémoire pour les graphes ayant peu d'arcs, car elle nécessite toujours un espace en $O(|V|^2)$. Parcourir les voisins d'un sommet se fait en temps $O(|V|)$ quel que soit le degré du sommet, par contre tester l'existence d'un arc ou une arête se fait en temps constant.

12.3.4 D'une représentation à une autre

Il faut être capable de passer d'une représentation creuse à une représentation dense. Rappelons un principe simple :

— lorsqu'on parcourt une liste, on utilise en général une fonction récursive ;

— lorsqu'on parcourt un tableau, on utilise en général une fonction itérative.

Ce n'est bien sûr pas obligatoire, mais en général c'est plus commode. Ainsi, pour passer d'une représentation creuse à une représentation dense, on utilisera une fonction récursive (interne) pour parcourir les listes. Inversement, pour passer de la représentation dense à la représentation creuse, on construira les listes d'adjacence à l'aide d'une boucle.

```
let creux_vers_dense g=
  let n=Array.length g in
  let m=Array.make_matrix n n 0 in
  for i=0 to n-1 do
    let rec aux q=match q with
      | [] -> ()
      | x::r -> m.(i).(x) <- 1 ; aux r
    in aux g.(i)
  done ;
  m
;;

let dense_vers_creux m=
  let n=Array.length m in
  let g=Array.make n [] in
  for i=0 to n-1 do
    for j=0 to n-1 do
      if m.(i).(j)=1 then g.(i) <-j::g.(i)
    done ;
  done ;
  g
;;
```

12.4 Parcours de graphes donnés par liste d'adjacence

Les parcours de graphes sont à la base de nombreux algorithmes sur les graphes. Ils vont nous permettre de calculer des plus courts chemins, des composantes connexes ou même fortement connexes, tester l'existence d'un cycle (circuit) dans un graphe, etc... On suppose que le graphe est donné par listes d'adjacence, et que l'ensemble de ses sommets est $[\![0, n-1]\!]$.

12.4.1 Parcours générique de graphe depuis un sommet source

On se donne un sommet source s_0, depuis lequel on veut explorer le graphe. On va donc suivre les arcs ou les arêtes et découvrir de nouveaux sommets. On suppose donnée une structure (de données) `a_traiter` dans laquelle on stocke les sommets à traiter : lorsqu'on traite un nouveau sommet, on ajoute à `a_traiter` la liste de ses sommets qui n'ont pas déja été traités. Pour savoir les sommets déja traités, on utilise un tableau de booléens.

Algorithme 12.29 : Parcours générique de graphe

Entrée : Un graphe G donné par listes d'adjacences, un sommet de départ s_0

```
a_traiter ← {s_0};
B ← [Faux, ..., Faux] ;
B[s_0] ← Vrai;
tant que a_traiter est non vide faire
    s ← sortir un élément de a_traiter;
    pour tout voisin s' de s tel que B[s'] est Faux faire
        a_traiter ← a_traiter ∪ {s'};
        B[s'] ← Vrai
```

Si les opérations d'ajout et de retrait d'un élément dans `a_traiter` se font en temps constant, la complexité du parcours générique est linéaire en $O(|V| + |E|)$ (en fait plutôt que $|E|$ c'est même le nombre d'arcs/arêtes du sous-graphe induit par l'ensemble des sommets accessibles depuis s_0). En effet, chaque arc (resp. arête) est exploré au plus une fois (resp. 2 fois). L'algorithme précédent ne renvoie rien, mais on peut l'adapter pour obtenir des informations sur le graphe : ceci dépend de la structure de données utilisée. Il y a deux choix naturels :

— une file mène à un parcours dit *en largeur*

— une pile mène à un parcours ressemblant au parcours *en profondeur* vu plus loin.

Un tel parcours permet d'explorer uniquement des sommets accessibles depuis un sommet s_0 donné (un sommet s est dit accessible depuis s_0 s'il existe au moins un chemin de s_0 à s). Vérifions qu'il les explore tous.

Proposition 12.30. *Un parcours de graphe avec l'algorithme 12.29 lancé en s_0 visite tous les sommets accessibles depuis s_0.*

Démonstration. Posons $\delta(s) = \inf\{$ longueur d'un chemin de s_0 à $s\} \in \mathbb{N} \cup \{+\infty\}$, défini pour tout sommet du graphe. Naturellement, $\delta(s) < +\infty$ si s est accessible depuis s_0. Raisonnons par l'absurde et supposons qu'il existe au moins un sommet s accessible depuis s_0 mais non visité. Considérons un de ces sommets s tel que $\delta(s)$ est minimal. $s \neq s_0$ car s_0 est visité. Considérons un plus court chemin de s_0 à s, noté $s_0, s_1, \ldots, s_k = s$. Clairement $\delta(s_i) \leq i$, mais il y a en fait égalité car sinon on pourrait construire un chemin plus court. Ainsi $\delta(s_{k-1}) < \delta(s)$, et s_{k-1} est visité par l'algorithme, donc s l'est aussi. □

12.4.2 Parcours en largeur et plus courts chemins

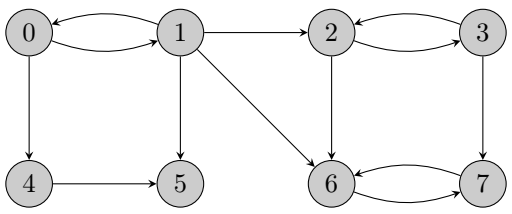


FIGURE 12.9 – Un exemple de graphe à parcourir

Donnons nous une structure de file pour l'ensemble `a_traiter`. On va utiliser le module `Queue` de Ocaml, fournissant les opérations classiques :

— `Queue.create : unit -> 'a Queue.t`
— `Queue.is_empty : 'a Queue.t -> bool`
— `Queue.add : 'a -> 'a Queue.t -> unit`
— `Queue.pop : 'a Queue.t -> bool`

En partant du sommet 0, tous les sommets sont accessibles. En supposant les listes d'adjacences données dans l'ordre croissant, voici l'ordre dans lequel sont traités les sommets :

à traiter	0	1, 4	4, 2, 5, 6	2, 5, 6	5, 6, 3	6, 3	3, 7	7	vide
déja vus	0	0, 1, 4	0, 1, 2, 4, 5, 6	idem	0, 1, 2, 3, 4, 5, 6	idem	tous	tous	tous

Une application du parcours en largeur est le calcul de plus courts chemins depuis l'origine s_0 du parcours. Il suffit de rajouter les deux informations suivantes dans le parcours en largeur :

— un tableau `dist` de distance à la source s_0. Lorsqu'on découvre un nouveau sommet t à partir d'un sommet s, `dist.(t)` prend la valeur `dist.(s)+1`.
— un tableau `pred` de prédecesseur : si t est découvert à partir de s, on pose $\pi(t) = s$. Le tableau des prédecesseurs fournit un plus court chemin entre la source s_0 et tout sommet accessible t : il suffit de remonter via `pred` jusqu'à s_0 pour avoir le chemin à l'envers.

Avant de montrer que cette approche est correcte, donnons un code complet en Caml.

```
let plus_courts_chemins g s0=
  let a_traiter=Queue.create () in
  let n=Array.length g in
  let deja_traites=Array.make n false and pred = Array.make n s0 and dist = Array.make n (-1) in
  deja_traites.(s0) <- true ;
  Queue.add s0 a_traiter ;
  dist.(s0) <- 0 ;
  while not (Queue.is_empty a_traiter) do
    let s=Queue.pop a_traiter in
    let rec aux v = match v with
      | [] -> ()
      | t::q when deja_traites.(t) ->  aux q
      | t::q -> Queue.add t a_traiter ; deja_traites.(t) <- true ; dist.(t) <- dist.(s) + 1 ;
         pred.(t) <- s ; aux q
    in aux g.(s)
  done ;
  dist, pred
;;
```

Dans le code précédent, tous les sommets à distance finie de s_0 ont une valeur associée dans `dist` qui est positive : c'est la distance d'un plus court chemin entre s_0 et ce sommet. Le tableau `pred` permet de reconstruire facilement un plus court chemin.

```
# let g = [|[1; 4]; [0; 2; 5; 6]; [3; 6]; [2; 7]; [5]; []; [7]; [6]|] ;;
val g : int list array =
  [|[1; 4]; [0; 2; 5; 6]; [3; 6]; [2; 7]; [5]; []; [7]; [6]|]
# plus_courts_chemins g 0 ;;
- : int array * int array =
([|0; 1; 2; 3; 1; 2; 2; 3|], [|0; 0; 1; 2; 0; 1; 1; 6|])
```

Proposition 12.31. *À la fin de l'algorithme, si s est un sommet accessible,* `dist.(s)` *contient la distance entre s_0 et s, et le chemin formé par les prédecesseurs de s via le tableau* `pred` *est un plus court chemin de s_0 à s.*

Démonstration. Tout d'abord, les sommets sont insérés dans la file avec une distance au sommet s_0 croissante : en effet, ceci se montre facilement par récurrence sur la distance à s_0, en considérant la propriété $\mathcal{P}(d)$: « les sommets à distance d sont insérés dans la file avant tous ceux à distance au moins $d+1$ ». $\mathcal{P}(0)$ est vraie, et si $\mathcal{P}(d)$ est vraie, un sommet à distance $d+2$ est inséré à partir d'un sommet à distance (au moins) $d+1$, donc après traitement de tous les sommets à distance d et donc insertion des sommets à distance $d+1$, donc $\mathcal{P}(d)$ implique $\mathcal{P}(d+1)$.

Considérons maintenant le chemin donné par les prédécesseurs dans le parcours, entre s_0 et un sommet $s \neq s_0$, qu'on note $s_0 \to s_1 \to \cdots \to s_n = s$. Considérons de plus un plus court chemin de s_0 à s, qu'on écrit $s_0 \rightsquigarrow t \to s$. t a été inséré après s_{n-1} dans la file (car s est découvert par s_{n-1}), donc $\delta(s_0, s_{n-1}) \leq \delta(s_0, t)$ d'après la propriété précédente. Ainsi $\delta(s_0, s) = \delta(s_0, t) + 1 \geq \delta(s_0, s_{n-1}) + 1$, ce qui prouve que le chemin $s_0 \to s_1 \to \cdots \to s_n = s$ est un plus court chemin de s_0 à s. □

Avant de parler du parcours en profondeur, montrons une propriété sur les prédecesseurs :

Proposition 12.32. *Considérons le graphe non orienté induit par les sommets accessibles depuis s_0 et les arêtes $\{s, \pi(s)\}$. Ce graphe est un arbre.*

Démonstration. Le graphe possède un sommet de plus que d'arêtes et est connexe : c'est donc un arbre. □

On parle de *l'arbre des prédecesseurs*, voir figure 12.10. Il est en fait naturellement orienté vers la source.

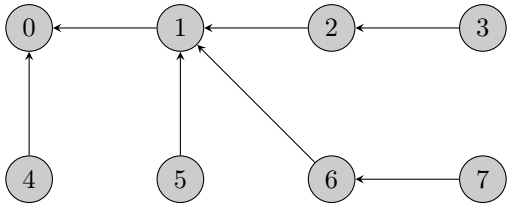


FIGURE 12.10 – L'arbre des prédécesseurs dans le parcours en largeur depuis 0

En fait cela ne dépend pas de la structure de données utilisées et est valable également pour l'algorithme générique.

12.4.3 Parcours en profondeur

Le parcours en profondeur d'un graphe consiste à s'enfoncer le plus possible dans le graphe avant de remonter vers des sommets déja vus, dont les voisins n'ont pas tous été découverts. On pourrait écrire un algorithme utilisant une pile à la place d'une file pour `a_traiter`, mais on découvrirait tous les voisins d'un sommet d'un coup, ce qui n'est pas un vrai parcours en profondeur (et il est important de respecter ceci dans les applications). Deux solutions :

— modifier légèrement l'algorithme générique : marquer un sommet comme traité une fois qu'il a été dépilé pour la première fois, pas dès qu'il est découvert. Ceci implique qu'un sommet peut se trouver plusieurs fois dans la pile (mais ne sera traité qu'une fois). L'inconvénient est une complexité spatiale $O(|V| + |E|)$ pour la pile, au lieu de $O(|V|)$;

— utiliser une formulation récursive (solution choisie !)

De plus, dans les applications du parcours en profondeur, on ne fait en général pas seulement un parcours élémentaire (sur un seul sommet comme dans le parcours générique) mais un parcours en profondeur « complet », qui consiste à relancer des parcours élémentaires tant que tout le graphe n'a pas été découvert. Pour assurer une complexité $O(|V| + |E|)$, on mutualise le tableau des booléens, comme dans le pseudo-code ci-dessous.

Algorithme 12.33 : Parcours en profondeur complet du graphe

```
Entrée : Un graphe G donné par liste d'adjacence
Sortie : Une énumération des sommets correspondant au parcours en profondeur, par date de fin de
         parcours décroissante
deja_vu[v] ← Faux pour tout sommet v;
Enum ← [];
Fonction pp(u) :
    deja_vu[u] ← Vrai;
    pour tout voisin v de u faire
        si deja_vu[v] = Faux alors
            pp(v)
    Rajouter u en tête de Enum;
pour tout sommet u du graphe faire
    si deja_vu[u] = Faux alors
        pp(u)
Renvoyer Enum
```

De plus, il est parfois utile de stocker plus d'informations pendant le parcours. On peut par exemple calculer les dates de début et de fin de traitement des sommets : on utilise un compteur (entier) que l'on incrémente lorsqu'on découvre un nouveau sommet où lorsqu'on a terminé de traiter l'un des sommets. On stocke les dates de début et de

fin de traitement dans des tableaux. De même que pour le parcours en largeur, il est courant de stocker le prédecesseur d'un sommet dans un tableau. La définition du prédécesseur d'un sommet dans le parcours est la même que pour le parcours générique : si v est découvert dans la boucle principale de la fonction `pp` appelée sur u, alors le prédecesseur de v est u. Pour le même graphe que précédemment, le parcours élémentaire lancé depuis le sommet 0 découvre tout le graphe, et on obtient l'arbre des prédécesseurs suivant de la figure 12.11.

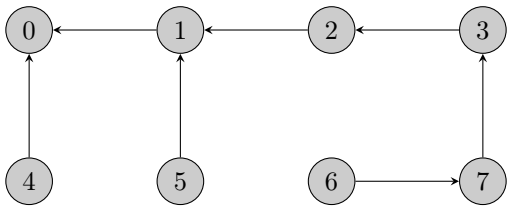


FIGURE 12.11 – L'arbre des prédécesseurs dans le parcours en profondeur depuis 0

On va voir dans la section suivante des variantes du parcours en profondeur permettant de résoudre les problèmes suivants :

— calcul des composantes connexes d'un graphe non orienté ;

— tri topologique d'un graphe orienté sans circuit ;

— calcul des composantes fortement connexes d'un graphe orienté.

Complexité. Tout comme le parcours en largeur, le parcours en profondeur a une complexité $O(|V|+|E|)$, pour les mêmes raisons : l'initialisation de `deja_vu` prend un temps $O(|V|)$, et ensuite la complexité est linéaire en le nombre d'arêtes/arcs $|E|$.

12.5 Applications du parcours en profondeur

12.5.1 Composantes connexes d'un graphe non orienté

Le calcul des composantes connexes d'un graphe non orienté est facile : lancer un parcours en profondeur « élémentaire » depuis un sommet permet de découvrir toute sa composante connexe. Ainsi, il suffit de pratiquer un parcours en profondeur complet du graphe, en stockant les résultats des parcours élémentaires dans des listes distinctes. On obtient donc une complexité $O(|V|+|E|)$, voici une implémentation Caml.

```
let composantes_connexes g=
  let n=Array.length g and liste_comp=ref [] and comp=ref [] in
  let deja_vu=Array.make n false in
  let rec pp y=match deja_vu.(y) with
    | true -> ()
    | false -> deja_vu.(y) <- true ; aux g.(y) ; comp:=y::!comp ;
  and aux p=match p with
      | [] -> ()
      | z::q ->  pp z ; aux q
  in
  for i=0 to n-1 do
    if not deja_vu.(i) then begin
      pp i ;
      liste_comp:=!comp::!liste_comp ;
      comp:=[]
    end
  done ;
  !liste_comp
;;
```

Voici le résultat, avec le graphe de la figure 12.2.

```
# let g = [|[1; 2; 4; 5]; [0; 2]; [0; 3]; [2]; [0; 5]; [0; 4]; [7]; [6; 8]; [7]; []|] ;;
val g : int list array =
  [|[1; 2; 4; 5]; [0; 2]; [0; 3]; [2]; [0; 5]; [0; 4]; [7]; [6; 8]; [7]; []|]
# composantes_connexes g ;;
- : int list list = [[9]; [6; 7; 8]; [0; 4; 5; 1; 2; 3]]
```

12.5.2 Tri topologique d'un graphe orienté sans circuit

Théorème 12.34. *Soit $G = (V, E)$ un graphe orienté, sans circuit. Alors il existe une énumération $s_0, \dots, s_{n-1}$ des sommets telle que si $(s_i, s_j) \in E$ alors $i < j$.*

La proposition précédente nous dit que l'on peut ordonner les sommets d'un graphe orienté sans circuit « en ligne », de sorte que tous les arcs aillent de gauche à droite. Avant de montrer ce théorème, énonçons et montrons le lemme suivant.

Lemme 12.35. *Dans un graphe orienté sans circuit, il existe un sommet de degré sortant nul.*

Démonstration. Par l'absurde, si tout sommet était de degré sortant non nul, alors on pourrait construire une suite $(v_i)_{i\in\mathbb{N}}$ de sommets de la façon suivante :

— v_0 un sommet quelconque.

— v_i un voisin quelconque de v_{i-1} pour $i \geq 1$ (dans un graphe orienté, cela signifie $(v_{i-1}, v_i) \in E$).

Puisque le nombre de sommets est fini, un sommet au moins apparaît deux fois, ce qui fournit un circuit : c'est absurde. □

Preuve du théorème 12.34. On montre cette propriété par récurrence sur le nombre de sommets n.

— si $n = 1$, c'est trivial.

— supposons donc $n \geq 2$, et considérons un graphe $G = (V, E)$ d'ordre n, sans circuit. Posons s_{n-1} un sommet de degré sortant nul. Le graphe induit par $V\backslash\{s_{n-1}\}$ est sans circuit, il existe donc par hypothèse de récurrence une énumération $s_0, \dots, s_{n-2}$ des sommets de ce graphe dont les arcs sont de la forme (s_i, s_j) avec $i < j$. Ainsi, $s_0, \dots, s_{n-1}$ est une énumération convenable des sommets de G, car les arcs impliquant s_{n-1} sont orientés vers s_{n-1}.

— Par principe de récurrence, le théorème est démontré.

□

Évidemment, la réciproque de ce théorème est vraie car un circuit rend une telle énumération impossible.

Définition 12.36. *Une énumération $s_0, \dots, s_{n-1}$ des sommets d'un graphe sans circuit, telle que si $(s_i, s_j) \in E$ alors $i < j$, se nomme un* ordre topologique *du graphe.*

On a montré l'existence d'un ordre topologique dans un graphe sans circuit, et en fait la démonstration du théorème fournit un algorithme pour en calculer un. Il est possible d'implémenter cette idée avec une complexité $O(|V| + |E|)$, mais la proposition suivante indique comment en calculer un facilement à l'aide d'un simple parcours en profondeur.

Proposition 12.37. *Considérons un parcours en profondeur complet d'un graphe sans circuit $G = (V, E)$, dans lequel les dates de fin de parcours sont stockées. Ordonner les sommets par date de fin de parcours décroissante fournit un ordre topologique du graphe.*

Démonstration. Considérons deux sommets u et v du graphe, tels que $(u, v) \in E$.

— si le parcours en profondeur découvre u avant v, alors comme v est voisin de u, v est découvert pendant le parcours de u, et donc l'exploration depuis v termine avant celle de u.

— si v est découvert avant u, comme il n'existe pas de chemin dans le graphe reliant v à u (sinon il y aurait un circuit) alors l'exploration depuis v termine également avant celle de u.

Par suite, l'énumération suivant les dates de parcours décroissantes fournit bien un ordre topologique. □

À l'inverse, il est également possible de détecter facilement l'existence d'un circuit dans un graphe orienté, en stockant également les dates de début de parcours en profondeur. En effet, considérons dans le parcours en profondeur d'un graphe ayant un circuit le premier sommet s_0 contenu dans un circuit à être découvert. Notons $s_0, s_1, \dots, s_{n-1} = s_0$ ce circuit. Tous les sommets $(s_i)_{1\leq i\leq n-2}$ seront découverts pendant le parcours en profondeur de s_0, et l'arc (s_{n-1}, s_0) sera examiné pendant ce parcours : lorsque c'est le cas, les parcours en profondeur des deux sommets ne sont pas terminés, et la date de début de parcours de s_{n-1} est postérieure à celle de s_0, ce qui signifie qu'il y a un chemin de s_0 à s_{n-1} : on peut donc détecter le circuit.

En fait, il est inutile de stocker des informations aussi précises pour le calcul d'un ordre topologique ou la détection de circuit : il suffit d'attribuer 3 couleurs aux sommets pendant le parcours en profondeur :

— blanche pour un sommet non découvert ;

— grise pour un sommet découvert mais donc le parcours en profondeur est en cours ;

Algorithme 12.38 : Ordre topologique d'un graphe orienté sans circuit

Entrée : Un graphe G orienté, sans circuit, donné par liste d'adjacence
Sortie : Un ordre topologique du graphe
`Couleur`$[v] \leftarrow$ Blanc pour tout sommet v;
`tri_top` $\leftarrow [\,]$;
Fonction *pp(u)* **:**
  `Couleur`$[u] \leftarrow$ `Gris`;
  pour *tout voisin v de u* **faire**
    si *`Couleur`*$[v] =$ *`Blanc`* **alors**
      pp(v)
    si *`Couleur`*$[v] =$ *`Gris`* **alors**
      Erreur : il y a un cycle !
  `Couleur`$[u] \leftarrow$ `Noir`;
  Rajouter u à la fin de `tri_top`;
pour *tout sommet u du graphe* **faire**
  si *`Couleur`*$[u] =$ *Blanc* **alors**
    pp(u)
Renvoyer `tri_top`, en sens inverse

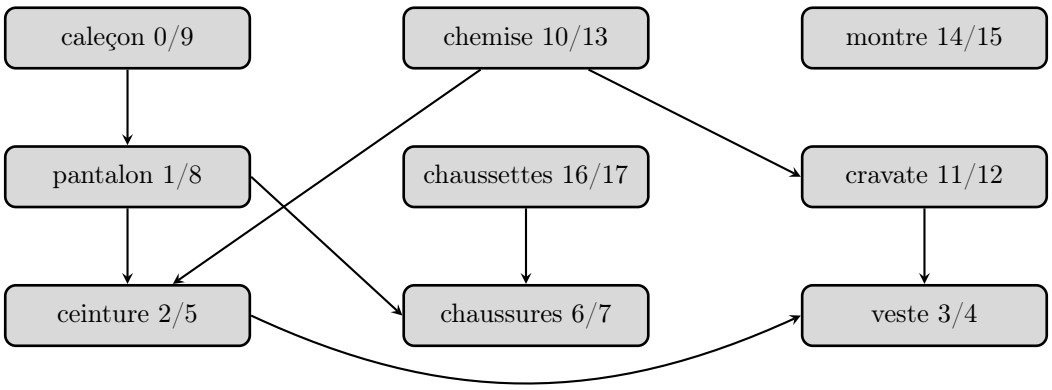


FIGURE 12.12 – Les vêtements du savant Cosinus

— noire pour un sommet dont le parcours en profondeur est terminé.

Un arc qui mène d'un sommet gris à un autre indique l'existence d'un circuit. S'il n'en existe pas, stocker les sommets au fur et à mesure qu'on leur attribue la couleur noire donne un ordre topologique (dans l'ordre inverse où on les stocke), voir algorithme 12.38.

Exemple 12.39 (Comment s'habiller le matin ?)**.** *Il faut aider le savant Cosinus à s'habiller. Dans le graphe de la figure 12.12, un arc entre le vêtement u et le vêtement v indique qu'il faut impérativement enfiler u avant v (typiquement, le caleçon avant le pantalon, les chaussettes avant les chaussures...). Mais dans quel ordre enfiler tout ça ? Un tri topologique fournit la solution. En lançant un parcours en profondeur complet sur les vêtements (de haut en bas et de gauche à droite), on obtient les dates de début et dates de fin de parcours de chaque nœud indiqué à côté des vêtements. Par ordre* décroissant *de date de fin du parcours en profondeur, on obtient l'ordre topologique de la figure 12.13, qui n'est pas forcément l'ordre le plus naturel, mais fonctionne !*

12.5.3 Calcul des composantes fortement connexes d'un graphe orienté (HP)

Le calcul des composantes fortement connexes d'un graphe orienté est plus technique que le calcul des composantes connexes d'un graphe non orienté : en effet, lorsqu'on lance un parcours en profondeur élémentaire depuis un sommet dans un graphe non orienté, les sommets découverts sont uniquement ceux de sa composante connexe, alors que dans un graphe orienté on découvre tous les sommets des composantes fortement connexes accessibles depuis le sommet.

On présente ici un algorithme dû à Kosaraju qui fait usage de *deux* parcours en profondeur pour calculer les composantes fortement connexes. L'un se fait sur le graphe, l'autre sur le *graphe transposé*, qui est le graphe obtenu en changeant le sens des arcs :

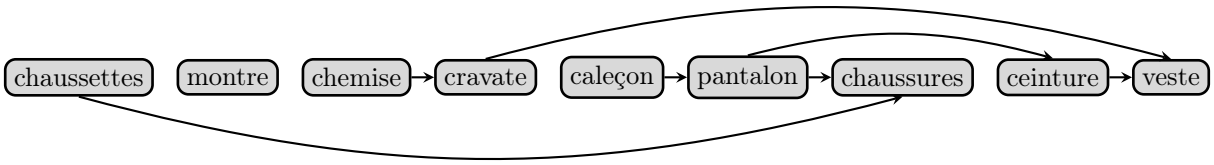


FIGURE 12.13 – Les vêtements du savant Cosinus, triés topologiquement

Définition 12.40. *Soit $G = (V, E)$ un graphe orienté. Le graphe transposé de G, noté ${}^tG$ est le graphe $(V, {}^tE)$ où ${}^tE = \{(v, u) \mid (u, v) \in E\}$.*

Une propriété essentielle pour le bon fonctionnement de l'algorithme est le lemme suivant.

Lemme 12.41. *Pour G un graphe orienté, les partitions des sommets de G et ${}^tG$ données par les composantes fortement connexes sont les mêmes.*

Démonstration. Si u et v sont dans une même composante fortement connexe dans G, cela signifie qu'il existe un chemin de u à v et un chemin de v à u. En changeant le sens des arcs, on obtient un chemin de v à u et un chemin de u à v dans ${}^tG$, donc u et v sont dans une même composante fortement connexe dans ${}^tG$. Comme ${}^{tt}G = G$, la réciproque est vraie, d'où le lemme. □

Donnons l'algorithme de Kosaraju en pseudo-code (algorithme 12.42).

Algorithme 12.42 : Algorithme de Kosaraju pour le calcul de composantes fortement connexes

Entrée : Un graphe G orienté
Sortie : Ses composantes fortement connexes
Effectuer un parcours en profondeur de G, en stockant les sommets dans l'ordre de leur date de fin de parcours d_f dans une liste L;
Effectuer un parcours en profondeur de ${}^tG$, mais dans la boucle principale du parcours, prendre les sommets par date d_f décroissante. Chaque parcours élémentaire fournit une composante fortement connexe.

Avant de montrer qu'il est correct, effectuons un exemple complet sur le graphe de la figure 12.14.

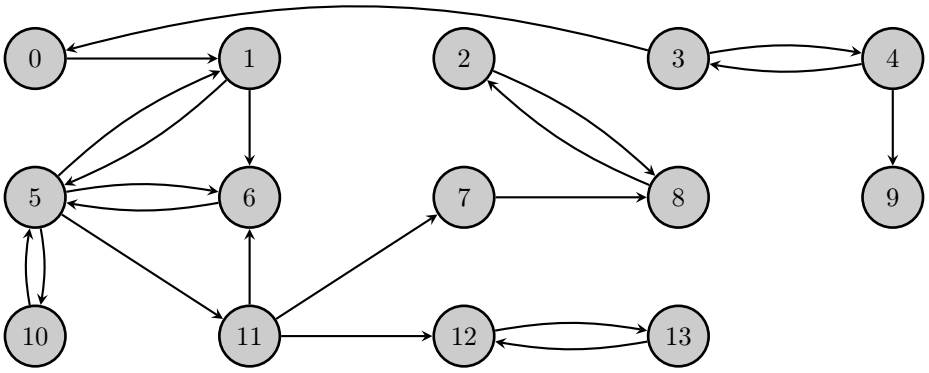


FIGURE 12.14 – Un graphe dont on veut calculer les composantes fortement connexes

Le tableau suivant donne les dates de début et fin de parcours en profondeur du graphe, en supposant que les listes d'adjacence sont triées dans l'ordre croissant (on effectue donc un premier parcours en profondeur élémentaire depuis 0, puis un autre depuis 3). Dans le tableau, les sommets sont ordonnés par date de découverte (début de l'exploration) croissante.

sommet	0	1	5	6	10	11	7	8	2	12	13	3	4	9
debut	0	1	2	3	5	7	8	9	10	14	15	22	23	24
fin	21	20	19	4	6	18	13	12	11	17	16	27	26	25

L'énumération des sommets, ordonnés par date de fin de parcours en profondeur décroissante est donc :

3, 4, 9, 0, 1, 5, 11, 12, 13, 7, 8, 2, 10, 6.

En relançant un parcours en profondeur complet sur le graphe transposé, l'ordre des sommets dans la boucle principale étant celui-ci, on obtient précisément les composantes fortement connexes :

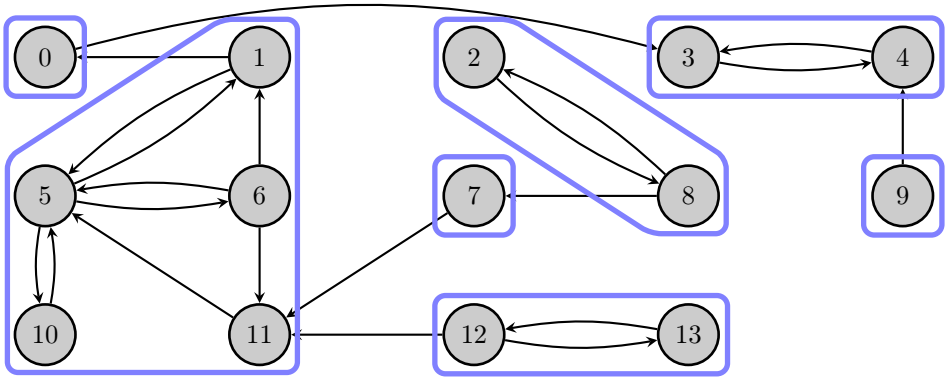


FIGURE 12.15 – Le graphe transposé, et les composantes fortement connexes

Proposition 12.43. *Soit G un graphe orienté, notons $C_1, \dots, C_k$ ses composantes fortement connexes. Lors du parcours en profondeur de G, notons v_i le premier sommet de C_i à être découvert. Quitte à réordonner les composantes, supposons que $v_1, \dots, v_k$ soient ordonnés par date de fin de parcours décroissante. Alors $C_1, \dots, C_k$ est un ordre topologique du graphe des composantes fortement connexes.*

Démonstration. La démonstration est essentiellement la même que pour montrer que le parcours en profondeur fournit un ordre topologique dans un graphe orienté sans circuit. Soit $\mathcal{C}$ et $\mathcal{C}'$ deux composantes fortement connexes, notons c et c' les deux sommets découverts en premier dans les deux composantes, et supposons qu'il existe un arc entre $\mathcal{C}$ et $\mathcal{C}'$.

- si c est découvert avant c' dans le parcours en profondeur, le parcours depuis c découvre tous les sommets de $\mathcal{C}$, et donc un arc entre $\mathcal{C}$ et $\mathcal{C}'$, et donc toute la composante $\mathcal{C}'$. Ainsi le parcours en profondeur depuis c termine après celui depuis c' ;
- si c' est découvert avant c, comme il n'existe pas de chemin entre c' et un sommet quelconque de $\mathcal{C}$, le parcours en profondeur découvrira c après que le traitement de c' soit terminé. Ainsi le parcours en profondeur depuis c termine après celui depuis c'.

□

Théorème 12.44. *L'algorithme de Kosaraju fournit bien les composantes fortement connexes d'un graphe orienté.*

Démonstration. On reprend les notations de la proposition 12.43. Puisqu'on traite dans la boucle principale du parcours en profondeur sur ${}^tG$ les sommets par date de fin de parcours sur G décroissante, chaque nouveau parcours en profondeur lancé dans la boucle principale l'est sur l'un des sommets v_i, en commençant par v_1. De plus puisque $C_1, \dots, C_k$ forme un ordre topologique des composantes connexes de G, un ordre topologique des composantes connexes de ${}^tG$ est $C_k, \dots, C_1$. Ainsi, le parcours lancé sur v_1 ne découvre que C_1, celui sur v_2 ne découvre que C_2, etc... L'algorithme de Kosaraju est bien correct. □

Proposition 12.45. *Le calcul des composantes fortement connexes d'un graphe orienté $G = (V, E)$ via l'algorithme de Kosaraju se fait en complexité $O(|V| + |E|)$.*

Démonstration. L'algorithme consiste à effectuer deux parcours en profondeur sur G et ${}^tG$, ce qui se fait en complexité $O(|V| + |E|)$. Le calcul de ${}^tG$ se fait également avec une complexité $O(|V| + |E|)$ (il faut parcourir toutes les listes d'adjacence et en créer de nouvelles), d'où le résultat. □

12.6 Résolution du problème 2-SAT

On se donne une formule logique f qui est une instance du problème 2-SAT, c'est à dire une conjonction de 2-clauses (des clauses de la forme $\ell_i \vee \ell_j$ où ℓ_i et ℓ_j sont des littéraux, égaux à une variable logique ou sa négation). On suppose que les variables apparaîssant dans les clauses sont distinctes (sinon on pourrait simplifier facilement -et rapidement !- la formule en propageant les littéraux unitaires). Peut-on décider rapidement si la formule f est satisfiable, et le cas échéant, calculer une distribution de vérité qui rende la formule vraie ? La réponse est oui, et on peut le voir comme une application du calcul des composantes fortement connexes des graphes orientés.

12.6.1 Construction d'un graphe à partir d'une instance de 2-SAT

À partir d'une conjonction de 2-clauses en les variables $v_1, \dots, v_n$, on construit un graphe à $2n$ sommets étiquetés par les littéraux $v_1, \dots, v_n, \neg v_1, \dots, \neg v_n$. Chaque clause $\ell \vee \ell'$ de la formule se réécrit $(\neg\ell \Rightarrow \ell') \wedge (\neg\ell' \Rightarrow \ell)$. On relie alors les littéraux $\neg\ell$ et ℓ' par un arc, de même que $\neg\ell'$ et ℓ. Rechercher les variables logiques dans la formule et construire le graphe s'effectue en temps polynomial en la taille de la formule. La formule est satisfiable si on peut trouver une affectation des variables qui ne crée aucune implication de la forme Vrai $\Rightarrow$ Faux.

Proposition 12.46. *Dans le graphe associé à une expression logique instance de 2-SAT, si $\{\ell_1, \dots, \ell_k\}$ sont les littéraux apparaîssant dans une composante fortement connexe, alors $\{\neg\ell_1, \dots, \neg\ell_k\}$ est également une composante fortement connexe.*

Démonstration. Par contruction du graphe, s'il y a un arc de ℓ à ℓ', il y a aussi un arc de $\neg\ell'$ à $\neg\ell$. Ainsi si deux littéraux sont dans la même composante fortement connexe, les littéraux conjugués sont également dans la même composante fortement connexe. □

Remarque 12.47. *En notant G le graphe ainsi construit, et $\tilde{G}$ le graphe où on échange v_i et $\neg v_i$ pour tout i, alors ${}^t\tilde{G} = G$.*

12.6.2 À la recherche d'une distribution de vérité

Théorème 12.48. *Une formule logique f instance de 2-SAT est satisfiable si et seulement si, dans le graphe associé, aucune composante fortement connexe ne contient à la fois une variable logique v et sa négation $\neg v$.*

Démonstration.
— La condition est nécessaire car si v et $\neg v$ se trouvent dans la même composante fortement connexe, cela signifie que la formule logique implique $v \Leftrightarrow \neg v$, qui est équivalente à 0 : f est donc antilogique.

— Pour montrer que la condition est suffisante, voici une preuve constructive : l'algorithme qui suit construit une distribution de vérité qui satisfait la formule logique.

  — calculer un ordre topologique $C_1, \dots, C_k$ des composantes connexes ;

  — remonter l'ordre topologique à l'envers, en donnant la valeur de vérité Vrai aux littéraux apparaîssant dans C_i s'ils n'en ont pas déja.

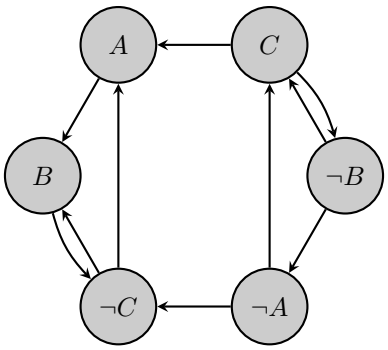
  Montrons que cet algorithme ne produit aucune implication de la forme Vrai $\Rightarrow$ Faux. Puisqu'aux littéraux apparaîssant dans une même composante est associée la même valeur de vérité, on peut supposer que dans chacune des composante apparaît un unique littéral. Le graphe a donc $2n$ composantes fortement connexes, qui sont les v_i et les $\neg v_i$. Considérons une implication $\ell \Rightarrow \ell'$ présente dans le graphe. Alors l'implication $\neg\ell' \Rightarrow \neg\ell$ est également présente.

  — Si ℓ' apparaît en dernier dans l'ordre topologique, l'algorithme affecte à ℓ' la valeur Vraie, et donc à $\neg\ell'$ la valeur Faux. Quelle que soit la valeur de vérité associée à ℓ, les deux implications sont satisfaites ;

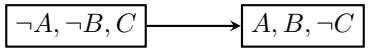
  — si $\neg\ell$ apparaît en dernier, le raisonnement est le même.

□

Exemple 12.49 (Exemple des desserts du cours de logique)**.** *On avait vu que le problème introductif du cours de logique pouvait se reformuler sous l'instance 2-SAT suivante : $(\neg A \vee B) \wedge (B \vee C) \wedge (\neg B \vee \neg C) \wedge (A \vee C) \wedge (\neg C \vee A)$. En reformulant chacune des clauses sous la forme de deux implications, on obtient le graphe suivant :*

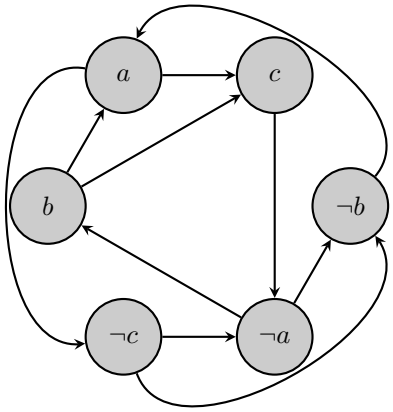


Un ordre topologique pour les composantes connexes est le suivant :



Ainsi, une valuation qui satisfait la formule est donc $d(A) = 1, d(B) = 1$ et $d(C) = 0$ (et c'est d'ailleurs la seule).

Exemple 12.50 (Autre exemple)**.** *On considère* $(a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee a) \wedge (\neg b \vee c) \wedge (\neg a \vee \neg c)$.



Les 6 littéraux sont tous dans la même composante fortement connexe : la formule logique n'est pas satisfiable.

12.6.3 Complexité

Notons c le nombre de clauses apparaîssant dans la formule, et notons n le nombre de variables logiques. Notons que $n = O(c)$. Le graphe se construit en temps $O(n + c) = O(c)$, et le calcul des composantes connexes et d'un ordre topologique (dans le cas où la formule est satisfiable) se fait également en temps $O(n + c) = O(c)$. Le calcul d'une distribution de vérité se fait ensuite en temps $O(n)$. Ainsi, on obtient un algorithme de complexité $O(n + c) = O(c)$ pour la résolution du problème 2-SAT, qui se résout donc en temps linéaire.

Chapitre 13

Couplage de cardinal maximal dans un graphe biparti

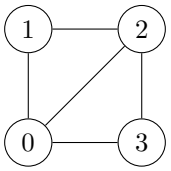
13.1 Définitions

Dans la suite, on travaille avec un graphe $G = (V, E)$, non orienté et non pondéré. E est donc un sous-ensemble de $\mathcal{P}_2(V)$, parties à deux éléments de V.

Définition 13.1 (Couplage dans un graphe général)**.** *Pour $G = (V, E)$ un graphe non orienté, on appelle* couplage *de G un sous-ensemble $C \subset E$ d'arêtes deux à deux non adjacentes, c'est-à-dire :*

$$\forall\, e_1, e_2 \in C, \quad e_1 \neq e_2 \Longrightarrow e_1 \cap e_2 = \emptyset$$

Exemple 13.2. *Considérons le graphe G suivant :*



Alors $C = \{\{0, 2\}\}$ est un couplage de G. $C' = \{\{0, 1\}, \{2, 3\}\}$ en est un également.

Définition 13.3 (Couplage maximal, couplage de cardinal maximum)**.** *On appelle :*

- couplage maximal *un couplage maximal au sens de l'inclusion parmi les couplages : il n'est inclus dans aucun couplage de cardinal strictement supérieur.*
- couplage de cardinal maximum *un couplage dont le nombre d'arêtes est maximal parmi tous les couplages.*

Évidemment, un couplage de cardinal maximum est un couplage maximal. La réciproque est fausse comme le montre l'exemple suivant.

Exemple 13.4. *Dans le graphe G pris en exemple précédemment, C et C' sont tous les deux des couplages maximaux : on ne peut rajouter aucune autre arête de G à l'un ou à l'autre sans qu'elle ait une extrémité commune avec l'une des arêtes de C ou C'. C' est de cardinal maximum. Il n'y a pas unicité du couplage de cardinal maximum car $\{\{0, 3\}, \{1, 2\}\}$ en est un également.*

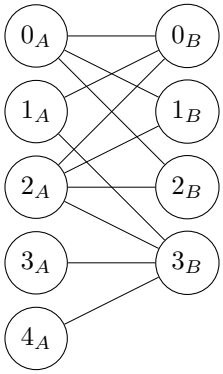
Déterminer un couplage maximal est plutôt facile en appliquant l'algorithme suivant : partir du couplage trivial $C = \emptyset$, considérer les arêtes e une à une, et dès qu'on en trouve une dont les deux extrémités n'appartiennent à aucune arête de C, rajouter e à $\mathcal{C}$.

Déterminer un couplage de cardinal maximal sur un graphe général est plus compliqué [1], néanmoins un cas plus facile est celui d'un graphe biparti.

1. Ce problème peut néanmoins être résolu avec une complexité polynomiale en les entrées V et E du graphe, ce n'est donc pas un problème difficile au sens de la théorie de la complexité. Mais les algorithmes en question sont plutôt ardus. Le plus accessible est l'algorithme d'Edmonds, ou *blossom algorithm* en anglais, qui généralise la recherche dans un graphe biparti étudiée dans la suite. Voir ici : `https://fr.wikipedia.org/wiki/Algorithme_d%27Edmonds_pour_les_couplages`.

Définition 13.5 (Graphe Biparti)**.** *Un graphe $G = (V, E)$ est dit* biparti *s'il existe une partition de V en deux sous-ensembles $V_1 \cup V_2$, telle que toute arête du graphe relie un sommet de V_1 à un sommet de V_2.*

Exemple 13.6. *Le graphe suivant est un graphe biparti.*



Remarque 13.7. *Les graphes bipartis interviennent fréquemment dans la pratique, par exemple dans les problèmes d'affectations : si par exemple, les sommets de gauche (de la forme x_A) sont des candidats à des formations post-bac, et les nœuds de droite (de la forme x_B) ces mêmes formations, et une arête entre deux tels nœuds un intérêt du candidat à la formation, trouver un couplage de cardinal maximum permet de contenter le plus de candidats. Les candidats peuvent aussi classer les formations par préférence, les formations font de même avec les candidats, et on retrouve le problème de Parcoursup*[2]*.*

Dans la suite, on se concentre sur le calcul d'un couplage de cardinal maximal dans un graphe biparti.

13.2 Chemin augmentant et recherche d'un couplage de cardinal maximum

Définition 13.8 (Sommet exposé, sommet couvert)**.** *Soit $G = (V, E)$ un graphe biparti, et C un couplage de G. Un sommet $v \in V$ est dit :*

— couvert *par le couplage C s'il appartient à une arête de C ;*

— exposé *sinon.*

Définition 13.9 (Chemin alternant)**.** *Soit $G = (V, E)$ un graphe biparti, et C un couplage de G. Un chemin $v_0, v_1, \ldots, v_k$ dans G est dit* alternant *(relativement à C), si :*

— $\{v_0, v_1\} \notin C$;

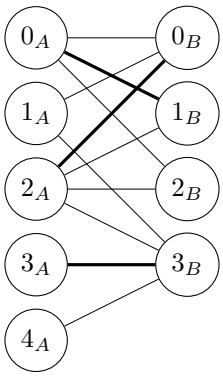
— $\{v_1, v_2\} \in C$;

— $\{v_2, v_3\} \notin C$;

— *etc...*

Autrement dit, le chemin alterne entre des arêtes qui ne sont pas dans C et des arêtes qui sont dans C, en commençant par une arête qui n'est pas dans C.

Exemple 13.10. *Dans le graphe suivant, les arêtes d'un couplage maximal ont été marquées en gras.*



2. Pour lequel il existe des algorithmes efficaces, mais pour de mauvaises raisons (masquer le manque de place dans le supérieur ?) la politique de Parcoursup est de ne pas demander aux candidats leurs préférences.

Le chemin $0_A, 0_B, 2_A, 2_B$ est un chemin alternant.

Définition 13.11 (Chemin augmentant)**.** *Soit $G = (V, E)$ un graphe biparti, et C un couplage de G. Un chemin $v_0, v_1, \dots, v_k$ dans G est dit* augmentant *(relativement à C), si c'est un chemin alternant et si de plus :*

- *il est de longueur impaire ;*
- *ces deux extrémités sont exposées, c'est-à-dire qu'elles n'appartiennent à aucune arête du couplage.*

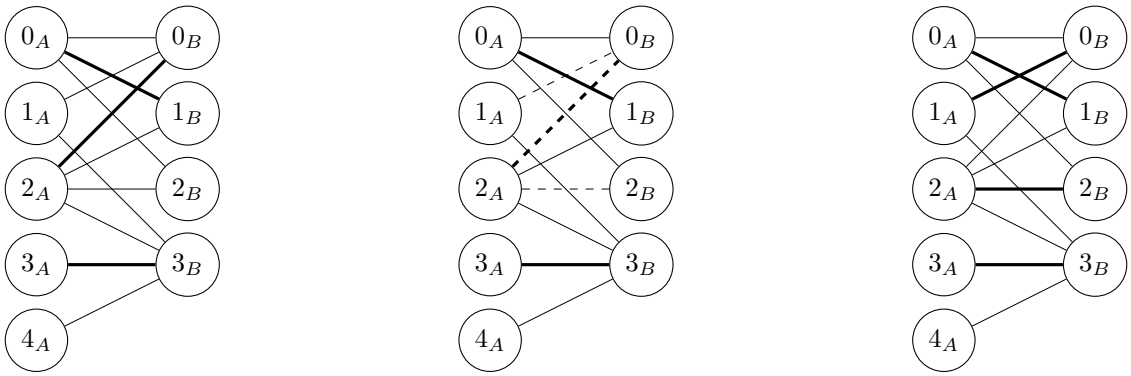
Exemple 13.12. *Dans l'exemple précédent, le chemin n'est pas augmentant puisque la première extrémité 0_A est couverte. Par contre, le chemin $1_A, 0_B, 2_A, 2_B$ est un chemin augmentant.*

Comme son nom l'indique, un chemin augmentant permet d'augmenter le cardinal d'un couplage.

Proposition 13.13. *Soit C un couplage d'un graphe biparti $G = (V_1 \cup V_2, E)$, possédant un chemin augmentant $v_0^1, v_0^2, v_1^1, v_1^2, \dots, v_1^k, v_2^k$. Alors il existe un couplage de G de cardinal $|C| + 1$.*

Démonstration. Dans la notation du chemin, les sommets $v_0^1, \dots, v_k^1$ appartiennent à V_1, les sommets $v_0^2, \dots, v_k^2$ appartiennent à V_2 (quite à retourner le chemin, on peut en effet supposer qu'il démarre dans V_1). Par définition, les sommets v_0^1 et v_k^2 ne sont pas couplés, alors que C contient les arêtes $\{v_i^2, v_{i+1}^1\}$ pour $i \in [\![0, k-1]\!]$. Par conséquent, l'ensemble obtenu en supprimant de C les k arêtes $\{v_i^2, v_{i+1}^1\}$ et en lui rajoutant les $k+1$ arêtes $\{v_i^1, v_i^2\}$ pour $i \in [\![0, k]\!]$ est bien un couplage de G, de cardinal $|C| + 1$. □

Exemple 13.14. *Voici le couplage obtenu dans le graphe précédent, via augmentation du chemin $1_A, 0_B, 2_A, 2_B$ (à gauche, le couplage initial, au milieu en pointillé le chemin augmentant, à droite le couplage obtenu). Le couplage est de cardinal maximum, puisque le graphe est biparti à 5+4 sommets et les 4 sommets de droite sont couplés.*



L'intérêt de la notion de chemin augmentant, c'est que l'existence d'un tel chemin caractérise le fait qu'un couplage ne soit pas de cardinal maximum, comme le montre la proposition suivante.

Proposition 13.15. *Soit $G = (V_1 \cup V_2, E)$ un graphe biparti, et C un couplage de G qui n'est pas de cardinal maximal. Alors G possède un chemin augmentant relativement à C.*

Démonstration. On considère C un couplage de G qui n'est pas de cardinal maximum parmi les couplages de G. Soit C' un couplage de cardinal maximum, qu'on peut supposer posséder le maximum d'arêtes en commun avec C parmi les couplages de cardinal maximum de G. Soit v_0^1 un sommet de V_1 couplé dans C' mais pas dans C. On construit un chemin augmentant d'origine v_0^1 ainsi :

- il existe $v_0^2 \in V_2$ tel que $\{v_0^1, v_0^2\} \in C'$. Si v_0^2 est non couplé dans C alors v_0^1, v_0^2 est un chemin augmentant pour C ;
- sinon, il existe $v_1^1 \in V_1$ tel que $\{v_0^2, v_1^1\} \in C$. Le sommet v_1^1 est couplé dans C' car sinon, en remplaçant l'arête $\{v_0^1, v_0^2\}$ de C' par l'arête $\{v_0^2, v_1^1\}$, on obtient un couplage de même cardinal que celui de C' ayant une arête en commun supplémentaire avec C.
- Ainsi, il existe $v_1^2 \in V_2$, tel que $\{v_1^1, v_1^2\} \in C'$. Si v_1^2 est non couplé dans C alors $v_0^1, v_0^2, v_1^1, v_1^2$ est un chemin augmentant pour C ;
- sinon, il existe $v_2^1 \in V_1$ tel que $\{v_1^2, v_2^1\} \in C$. De même que précédemment, le sommet v_2^1 est couplé dans C' car sinon, en remplaçant les arêtes $\{v_0^1, v_0^2\}$ et $\{v_1^1, v_1^2\}$ de C' par les arêtes $\{v_0^2, v_1^1\}$ et $\{v_1^2, v_2^1\}$ de C, on obtient un couplage de même cardinal que celui de C' ayant deux arêtes en commun supplémentaires avec C.
- ...

La finitude du nombre de sommets implique que le processus précédent s'arête : lorsque c'est le cas un chemin augmentant relativement à C a été trouvé. □

Algorithme 13.16 : Calcul d'un couplage de cardinal maximum dans un graphe biparti

Entrée : Un graphe $G = (V_1 \cup V_2, E)$ biparti
Sortie : Un couplage de cardinal maximum de G
$C \leftarrow \emptyset$;
tant que *il existe un chemin c augmentant relativement à C* **faire**
  Utiliser c pour augmenter le cardinal de C de 1
Renvoyer C;

De la proposition précédente, on tire l'algorithme 13.16 permettant de trouver un couplage de cardinal maximum dans un graphe biparti.

Bien sûr, pour appliquer l'algorithme, il faut savoir tester s'il existe un chemin augmentant et en calculer un le cas échéant. Ceci peut être fait à l'aide d'une variante du parcours en profondeur.

13.3 Détection de chemins augmentants et complexité de l'algorithme

À partir d'un graphe $G = (V_1 \cup V_2, E)$, et d'un couplage C de G, on peut associer un graphe orienté G' en procédant comme suit :

— pour chaque arête $\{v_1, v_2\}$ de G qui n'appartient pas à C, avec $(v_1, v_2) \in V_1 \times V_2$, on ajoute l'arc (v_1, v_2) dans G' ;

— pour chaque arête $\{v_1, v_2\}$ de C, avec $(v_1, v_2) \in V_1 \times V_2$, on ajoute l'arc (v_2, v_1) dans G' ;

Autrement dit, on peut aller de V_1 à V_2 avec toute arête hors du couplage, et revenir avec une arête du couplage. La propriété suivante montre qu'un simple parcours de graphe permet de détecter l'existence d'un chemin augmentant :

Proposition 13.17. *Avec cette définition, il existe dans G' un sommet de V_2 de degré sortant nul, accessible depuis un sommet de V_1 de degré entant nul, si et seulement si il existe dans G un chemin augmentant pour C.*

Démonstration. Immédiat par définition des chemins augmentants et définition du graphe G'. □

La proposition précédente montre que pour détecter un chemin augmentant, on peut simplement lancer un parcours en profondeur sur G', en se restreignant dans la boucle principale aux sommets de V_1 de degré entrant nul. Pour calculer effectivement un chemin augmentant, il suffit d'utiliser un tableau de prédecesseurs : remonter du sommet de V_2 trouvé jusqu'au sommet de V_1 à l'origine du parcours élémentaire donne un chemin augmentant.

Proposition 13.18. *La complexité de l'algorithme 13.16 est $O(\min(|V_1|, |V_2|)(|V| + |E|))$ sur un graphe biparti $G = (V, E)$, avec $V = V_1 \cup V_2$, représenté par listes d'adjacence.*

Démonstration. Un couplage est de cardinal inférieur à $\min(|V_1|, |V_2|)$, il faut donc appliquer au plus $\min(|V_1|, |V_2|)$ fois un algorithme de parcours de complexité $O((|V| + |E|))$, d'où le résultat. □

Remarque 13.19. *En pratique, on ne recalcule pas de graphe G' à chaque étape ! Pour l'encodage, on peut :*

— *stocker le graphe biparti par listes d'adjacence, celles des sommets de V_1 suffisent ;*

— *coder un couplage comme un tableau d'entiers, associant à chaque sommet celui auquel il est couplé le cas échéant (et une autre valeur comme -1 s'il n'est pas couplé, par exemple).*

Cette encodage suffit pour calculer efficacement un couplage de cardinal maximum.

Chapitre 14

Graphes pondérés : compléments et arbres couvrants

14.1 Introduction

Dans ce chapitre, on considère des graphes où les arêtes sont munies d'un *poids*. Les applications sont nombreuses :

— dans le plan euclidien, si on considère un nuage de points, on peut considérer le graphe complet sur ce nuage, les arêtes ont alors pour poids la distance entre ces sommets.

— une variante s'applique aux problématiques du transport : on a un réseau de routes (des arcs), dont le poids est la distance entre ces villes. Les problèmes de plus courts chemins apparaîssent naturellement dans ce contexte.

— un poids sur une arête/un arc peut modéliser une capacité de flux, etc...

On a vu dans le chapitre précédent comment calculer la distance $\delta(s,t)$ entre un sommet source s et un sommet quelconque t dans un graphe non pondéré, à l'aide d'un parcours en largeur. Le but de ce chapitre est de généraliser cet algorithme au cas des graphes pondérés.

14.2 Définition et représentations des graphes pondérés

14.2.1 Pondération

Définition 14.1. *On considère un graphe $G = (V, E)$ orienté ou non. Une* fonction de pondération *de G est une fonction $\omega : E \to \mathbb{R}$. Le réel $\omega(e)$ est appelé le* poids *de l'arête ou de l'arc e. Le graphe $G = (V, E, \omega)$ est appelé un graphe* pondéré.

On étend naturellement la fonction de pondération à tout couple de sommets pour obtenir une fonction $V^2 \to \mathbb{R} \cup \{+\infty\}$, avec la définition suivante :

$$\omega'(u,v) = \begin{cases} \omega((u,v)) & \text{si } (u,v) \in E \\ 0 & \text{si } u = v \\ +\infty & \text{sinon.} \end{cases}$$

14.2.2 Implémentation

En pratique, on se ramène très souvent au cas où les poids sont des entiers. L'implémentation en Caml est assez immédiate.

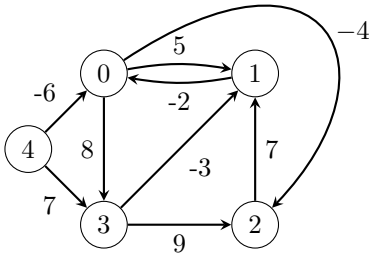
- Dans le cas d'une implémentation creuse, il suffit dans la liste d'adjacence d'un sommet u, de stocker des couples (v,p) à la place du seul sommet v : ceci signifie que $(u,v) \in E$ et $\omega(u,v) = p$. Le type utilisé est donc le suivant avec des poids entiers :

```
type graphe_pondere_creux = (int * int) list array ;;
```

- Dans le cas d'une implémentation dense, on utilise la généralisation de la fonction ω à tout couple de sommets : la matrice d'adjacence est maintenant une matrices à valeurs dans $\mathbb{R} \cup \{+\infty\}$. En pratique, comme les poids sont souvent des entiers, on utilise plutôt une matrice à coefficients dans $\mathbb{Z} \cup \{+\infty\}$, ce qui mène au type suivant en Caml :

```
type zbar = Z of int | Inf ;;
type graphe_pondere_dense = zbar array array ;;
```

Voici un exemple de graphe pondéré, et sa matrice d'adjacence (figure 14.1).



$$\begin{pmatrix} 0 & 5 & -4 & 8 & +\infty \\ -2 & 0 & +\infty & +\infty & +\infty \\ +\infty & 7 & 0 & +\infty & +\infty \\ +\infty & -3 & 9 & 0 & +\infty \\ -6 & +\infty & +\infty & 7 & 0 \end{pmatrix}$$

FIGURE 14.1 – Un graphe pondéré et sa matrice d'adjacence

Remarque 14.2. *Attention à ne pas confondre la matrice d'adjacence dans les graphes pondérés et non pondérés : les zéros dans la matrice d'adjacence d'un graphe non pondéré deviennent des $+\infty$ dans la matrice d'un graphe pondéré, sauf sur la diagonale où ils restent des zéros.*

14.3 Définitions et premières propriétés sur les plus courts chemins

14.3.1 Définition

Définition 14.3. *La notion de poids d'un arc s'étend au poids d'un chemin : pour $p = v_0, v_1, \ldots, v_n$ un chemin dans un graphe, on définit son poids comme $\omega(p) = \sum_{i=0}^{n-1} \omega(v_i, v_{i+1}) \in \mathbb{R}$.*

Venons-en à la définition de la distance entre deux sommets.

Définition 14.4. *Pour s et t deux sommets dans le graphe, on appelle distance de s à t, notée $\delta(s,t)$, comme*

$$\delta(s,t) = \inf\{\omega(p) \mid p \text{ est un chemin de } s \text{ à } t\} \in \mathbb{R} \cup \{\pm\infty\}$$

Remarque 14.5. *Pour s et t deux sommets du graphe, $\delta(s,t)$ peut prendre les valeurs $+\infty$ et $-\infty$:*

- *si t n'est pas accessible depuis s, il n'existe pas de chemin entre s et t. L'ensemble définissant $\delta(s,t)$ étant vide, on a bien $\delta(s,t) = +\infty$.*
- *si t est un sommet accessible depuis s et si, sur un chemin de s à t, il existe un circuit de poids strictement négatif, le poids d'un chemin de s à t n'est pas borné inférieurement, car on peut construire des chemins de poids arbitrairement petit en bouclant sur ce circuit. Ainsi, $\delta(s,t) = -\infty$.*

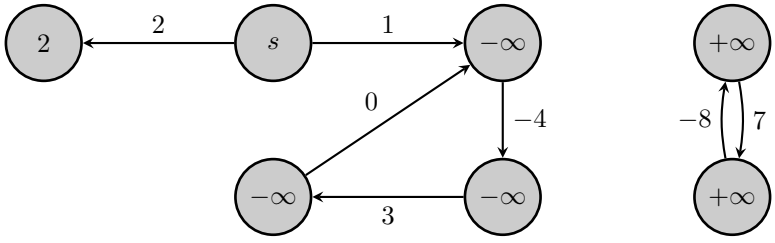


FIGURE 14.2 – Un exemple de graphe avec des circuits de poids strictement négatifs : on a fait figurer les distances à la source s. Trois sommets sont accessibles et situés sur un circuit de poids strictement négatif. Les sommets de droite sont également sur un tel circuit, mais non accessibles.

Définition 14.6. *Pour s et t deux sommets tels que $\delta(s,t) \notin \{\pm\infty\}$, on appelle plus court chemin de s à t un chemin de poids $\delta(s,t)$ entre s et t.*

14.3.2 Différents problèmes de plus courts chemins

Lors des calculs de plus courts chemins, nos algorithmes donneront une réponse erronée pour les sommets situés sur des circuits (accessibles) de poids strictement négatifs. On supposera donc qu'il n'y en a pas, même si en pratique il est facile de les détecter.

Dans le problème du calcul de plus courts chemins, on peut énumérer 4 cas :

1. calcul d'un plus court chemin depuis une source s et une destination t ;
2. calcul de plus courts chemins depuis une origine unique s fixée ;
3. calcul de plus courts chemins vers une destination unique s fixée ;
4. calcul de plus courts chemins entre deux couples quelconques de sommets s et t.

Le problème (3) est symétrique du problème (2) : en effet, on se ramène au problème (2) à partir du problème (3) en travaillant sur le graphe transposé. On verra par la suite des algorithmes pour résoudre les problèmes (2) et (4). Pour le problème (1), on se contentera du problème (2) : même s'il est *a priori* plus facile, on ne connaît pas à l'heure actuelle d'algorithme de résolution de (1) plus efficace asymptotiquement qu'un algorithme qui résout (2).

14.3.3 Optimalité des solutions aux sous-problèmes

Le problème du calcul d'un plus court chemin entre deux sommets est un problème *d'optimisation*. Mais calculer un plus court chemin entre deux sommets fournit une solution à des sous-problèmes :

Proposition 14.7 (Optimalité des sous-chemins d'un plus court chemin)**.** *Soit $s \overset{c}{\rightsquigarrow} t$ un plus court chemin passant par un sommet u. Alors les deux morceaux $s \overset{c_1}{\rightsquigarrow} u$ et $u \overset{c_2}{\rightsquigarrow} t$ sont des plus courts chemins de s à u et de u à t.*

Démonstration. La démonstration est facile, et classique : s'il existait un chemin plus court de s à u, noté c_1', on obtiendrait un chemin de s à t en considérant $s \overset{c_1'}{\rightsquigarrow} u \overset{c_2}{\rightsquigarrow} t$, de poids $\omega(c_1') + \omega(c_2) < \omega(c_1) + \omega(c_2) = \omega(c)$, ce qui est absurde. De même pour le chemin c_2. □

La propriété précédente montre que dans le problème du calcul de plus courts chemins réside une *optimalité des solutions aux sous-problèmes* : déterminer un plus court chemin de s à t fournit en même temps des plus courts chemins depuis s (ou vers t) pour tous les sommets situés sur le chemin. C'est une caractéristique des problèmes pour lesquels les méthodes de programmation dynamique peuvent s'appliquer. Parmi les algorithmes que l'on va voir, trois d'entre eux font usage de programmation dynamique, le dernier étant un algorithme glouton (un algorithme pour lequel il y a un choix localement optimal qui est globalement optimal, et évite le recours à la programmation dynamique, plus lourde).

Comme souvent en programmation dynamique, on commence par calculer les valeurs optimales (ici les $\delta(s,t)$) avant de modifier l'algorithme pour calculer les solutions optimales (pour tout (s,t), un chemin c de s à t pour lequel $\omega(c) = \delta(s,t)$). On va donc se concentrer sur le calcul des $\delta(s,t)$.

14.4 Plus courts chemins à origine unique

Le but de cette section est de calculer les distances entre une origine s fixée et tous les sommets du graphe, donc $(\delta(s,t))_{t\in G}$. On va voir deux algorithmes : l'algorithme de Bellman-Ford (HP) et l'algorithme de Dijkstra. Le premier a une moins bonne complexité, mais a le mérite d'être très simple et de s'appliquer lorsqu'il y a des arcs de poids strictement négatifs, alors que l'algorithme de Dijkstra nécessite que tous les arcs soient de poids positifs. Les deux fonctionnent sur le même principe : il s'agit de *relâcher* les arcs du graphe, dans un certain ordre.

Dans la suite, on suppose que le graphe ne possède pas de circuit de poids strictement négatif.

14.4.1 Relâchement d'arcs

Lemme 14.8 (Inégalité triangulaire)**.** *Soit s, t et u trois sommets d'un graphe pondéré G. Alors $\delta(s,t) \leq \delta(s,u) + \omega(u,t)$.*

Démonstration. S'il existe un chemin de s à u et un arc de u à t, on a $\delta(s,u) < +\infty$ et $\omega(u,t) < +\infty$. On obtient un chemin de s à t de poids $\delta(s,u) + \omega(u,t)$ en prenant un plus court chemin de s à u et l'arc (u,t), donc $\delta(s,t) \leq \delta(s,u) + \omega(u,t)$. Sinon, on a $\delta(s,u) + \omega(u,t) = +\infty$, et l'inégalité reste valable. □

Lemme 14.9. *Soit t un sommet accessible depuis s. Alors il existe un chemin de poids $\delta(s,t)$ entre s et t, composé de sommets tous distincts.*

Démonstration. Considérons un chemin c entre s et t, de poids $\delta(s,t)$, et de longueur (nombre d'arcs) minimale parmi les chemins de poids $\delta(s,t)$ reliant s à t. S'il existait deux sommets égaux sur le chemin, on obtiendrait un circuit. Comme il n'y a pas de circuit de poids strictement négatif dans le graphe par hypothèse, ce circuit est de poids nul (sinon on pourrait le supprimer pour obtenir un chemin de s à t de poids strictement inférieur à $\delta(s,t)$, ce qui est exclu). Mais le supprimer mène alors à un chemin de même poids mais avec strictement moins d'arcs, ce qui est exclu également. Donc c est composé de sommets distincts. □

Définition 14.10. *Supposons que le tableau $(d_s(t))_{t\in G}$ soit une estimation des distances $\delta(s,t)$ (c'est-à-dire $d_s(t) \geq \delta(s,t)$ pour tout t). Relâcher l'arc (u,v) consiste à réaliser l'affectation $d_s[v] \leftarrow \min(d_s[v], d_s[u] + \omega(u,v))$.*

Lemme 14.11. *(Mêmes notations) On a toujours $d_s[v] \geq \delta(s,v)$ après relâchement de l'arc (u,v).*

Démonstration. Avant relâchement, on a $\delta(s,u) \leq d_s[u]$ par hypothèse. Ainsi, par inégalité triangulaire, $\delta(s,v) \leq d_s[u] + \omega(u,v)$. □

Proposition 14.12. *Soit t un sommet accessible depuis s et $c = (s_0 = s, s_1, \ldots, s_k = t)$ un chemin de poids $\delta(s,t)$ entre s et t. En partant d'un tableau $(d_s[u])_{u\in G}$ quelconque tel que $d_s[u] \geq \delta(s,u)$ pour tout u, avec $d_s[s] = 0$, si on relâche successivement les arcs $(s_0,s_1),\ldots,(s_{k-1},s_k)$ dans cet ordre, alors $d_s[t]$ contient $\delta(s,t)$ à la fin du processus.*

Démonstration. La proposition 14.7 (optimalité des sous-chemins) montre que pour tout $i \in \{0,\ldots,k\}$, $(s_0,s_1,\ldots,s_i)$ est un plus court chemin de s à s_i. Montrons par récurrence sur i qu'après relâchement de l'arc (s_{i-1},s_i), $d_s[s_i]$ contient $\delta(s,s_i)$:

- $i = 0$: $d_s[s]$ vaut 0, qui est bien $\delta(s,s)$.
- soit $i > 1$ et supposons la propriété démontrée au rang $i-1$. Alors juste avant le relâchement de (s_{i-1},s_i), $d_s[s_{i-1}]$ contient $\delta(s,s_{i-1})$. Comme $\delta(s,s_i) = \delta(s,s_{i-1}) + \omega(s_{i-1},s_i)$, on a $d[s_i] \leq \delta(s,s_i)$ après relâchement de l'arc (s_{i-1},s_i). Or le lemme 14.11 prouve que $d_s[s_i]$ était supérieur ou égal à $\delta(s,s_i)$ avant relâchement. Donc la propriété est vraie au rang i.
- Par principe de récurrence, elle est vraie pour tout $i \in \{0,\ldots,k\}$, et en particulier $d_s[t]$ contient $\delta(s,t)$ à la fin du processus.

□

Remarque 14.13. *La propriété précédente reste vraie si on relâche d'autres arcs entre les relâchements des (s_i,s_{i+1}) : en effet, relâcher un arc ne peut que faire baisser les $d_s[u]$, mais le lemme 14.11 assure que l'on ne descendra pas en dessous de $\delta(s,u)$!*

14.4.2 Algorithme de Bellman-Ford (HP)

L'algorithme de Bellman-Ford est directement inspiré de la proposition 14.12 précédente, et du lemme 14.9 : avec $n = |V|$, il suffit de relâcher $n-1$ fois tous les arcs du graphe $G = (V,E,\omega)$ pour calculer les $\delta(s,t)$. Voici l'algorithme en pseudo code :

Algorithme 14.14 : Algorithme de Bellman-Ford

Entrée : Un graphe pondéré $G = (V,E,\omega)$ donné par listes d'adjacences, un sommet s
Sortie : Les distances $\delta(s,t)$ pour tout $t \in V$
$d[t] \leftarrow +\infty$ pour tout $t \in V$; $d[s] \leftarrow 0$;
$n \leftarrow |V|$;
pour *i entre 1 et $n-1$ faire* **faire**
  pour *j entre 0 et $n-1$ faire* **faire**
    pour *tout voisin u de j faire* **faire**
      $d[u] \leftarrow \min(d[u], d[j] + \omega(j,u))$
Renvoyer d

Correction de l'algorithme de Bellman-Ford. Montrons que l'algorithme de Bellman-Ford est correct si le graphe n'a pas de circuit de poids strictement négatif. Soit t un sommet quelconque du graphe, accessible depuis s. Comme on ne fait que relâcher des arcs dans l'algorithme, le lemme 14.11 implique $d[t] \geq \delta(s,t)$ à la fin de l'algorithme. Le lemme 14.9 assure l'existence d'un plus court chemin $s = s_0, s_1, \ldots, s_k = t$ où les sommets sont distincts : on a donc $k \leq n-1$. Ainsi, l'arc (s_0, s_1) est relâché lors du tour $i = 1$ de la boucle principale, l'arc (s_1, s_2) est relâché pendant le tour $i = 2$, et de même jusqu'à l'arc (s_k, s_{k-1}) relâché lors du tour $i = k$. Puisque les arcs sont relâchés dans l'ordre du chemin, on a bien $d[t] \leq \delta(s,t)$ à la fin de l'algorithme. La propriété « $d[t] = +\infty$ si t n'est pas accessible depuis s » est clairement un invariant des boucles de l'algorithme, donc celui-ci calcule correctement toutes les distances $\delta(s,t)$.

Complexité. L'algorithme de Bellman-Ford est de complexité $O(n(a+n))$, puisqu'il relâche $n-1$ fois tous les arcs de G, et un relâchement de tous les arcs nécessitant de parcourir toutes les listes d'adjacence, il a un coût $O(n+a)$.

Détection des circuits de poids total strictement négatif. Il est en fait facile de tester l'existence d'un circuit de poids total strictement négatif accessible depuis s, via la propriété suivante.

Proposition 14.15. *Dans l'algorithme de Bellman-Ford, effectuons un relâchement supplémentaire de tous les arcs (boucle principale de 1 à n). Alors le tableau d est modifié pendant le dernier tour de boucle si et seulement si il existe un circuit de poids total strictement négatif accessible depuis s.*

Démonstration. Dans le cas où il n'y a pas de tel circuit, on a vu qu'après les $n-1$ tours de la boucle principale, $d[t]$ contient $\delta(s,t)$ pour tout sommet t. Un relâchement d'arc ne peut donc diminuer aucun $d[t]$. Supposons maintenant qu'il existe un circuit de poids total strictement négatif, mais qu'aucun $d[u]$ ne soit modifié. Ceci signifie qu'avant le dernier tour de boucle, on a $d[u] \leq d[j] + \omega(j,u)$ pour tout arc (j,u). Considérons un circuit $s_0, \ldots, s_k = s_0$ de poids total strictement négatif, accessible depuis s. Alors avant le dernier tour de boucle, $d[s_{i+1}] \leq d[s_i] + \omega(s_i, s_{i+1})$ pour tout $i \in [\![0, k-1]\!]$. On a de plus $d_s[s_i] < +\infty$ pour tout $i \in [\![0, k-1]\!]$ car chaque s_i est accessible, donc $d[s_{i+1}] - d[s_i] \leq \omega(s_i, s_{i+1})$. Ainsi en sommant ces inégalités, on obtient, comme $s_0 = s_k$:

$$0 = \sum_{i=0}^{k-1} (d[s_{i+1}] - d[s_i]) \leq \sum_{i=0}^{k-1} \omega(s_i, s_{i+1})$$

Ce qui est absurde. □

Ainsi, il suffit de faire n tours de la boucle principale au lieu de $n-1$ pour pouvoir tester l'existence d'un circuit de poids total strictement négatif accessible depuis s.

Calcul effectif de plus courts chemins. Pour calculer effectivement des plus court chemins depuis s, il suffit comme dans le parcours en largeur d'un graphe non pondéré d'ajouter un tableau de prédecesseurs π : dans la boucle interne de l'algorithme, si $d[u] > d[j] + \omega(j,u)$ alors $d[u]$ prend la valeur $d[j] + \omega(j,u)$ et $\pi[u]$ prend la valeur j. À la fin de l'algorithme, il suffit de remonter depuis un sommet accessible vers s en suivant le tableau des prédecesseurs pour obtenir un plus court chemin, à l'envers.

Exemple. On considère le graphe de la figure 14.3, constitué de 5 sommets : la source s, ainsi que t, x, y, z. On suppose que l'on relâche les arcs dans l'ordre lexicographique, excepté ceux de la forme (s,t) et (s,y) que l'on relâche à la fin. Ainsi, l'ordre de relâchement des arcs est (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y). Comme il y a 5 sommets, les arcs doivent être relâchés 4 fois. On indique dans chaque sommet a l'estimation $d[a]$, après un relâchement de tous les arcs, et en gras ceux qui ont été modifiés après une étape. Dans cet exemple, il est bien nécessaire de relâcher 4 fois tous les arcs. Un relâchement supplémentaire ne change rien : il n'y a pas de circuit de poids strictement négatif (néanmoins, s, y, x, t, z, s est de poids nul). Les arcs en gras représentent le tableau des prédecesseurs.

14.4.3 Algorithme de Dijkstra

L'algorithme précédent a une complexité très élevée comparée au parcours en largeur du chapitre précédent, qui s'effectuait en temps linéaire $O(a+n)$. L'algorithme de Dijkstra [1] que l'on va voir est une généralisation du parcours en largeur : il consiste également à traiter les sommets un par un, par distance à l'origine s croissante. Pour fonctionner, l'algorithme a besoin d'une hypothèse non nécessaire pour l'algorithme de Bellman-Ford :

On suppose pour l'algorithme de Dijkstra que les arcs ont un poids positif.

1. prononcer « Daïjkstra »

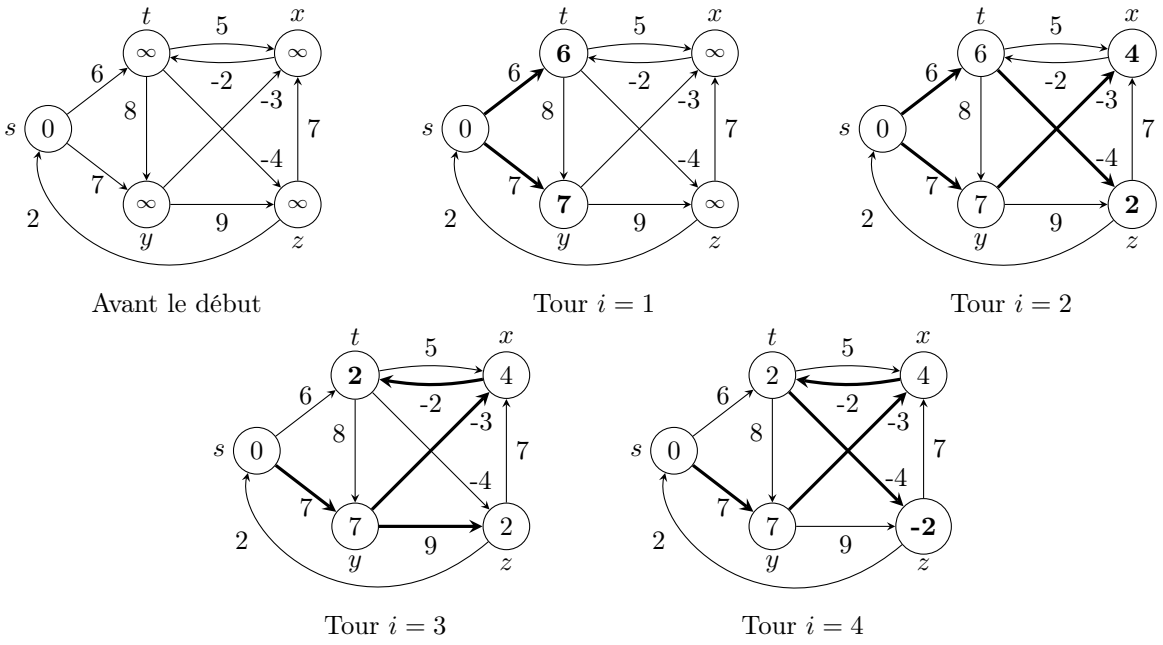


FIGURE 14.3 – Déroulement de l'algorithme de Bellman-Ford

Algorithme 14.16 : Algorithme de Dijkstra

Entrée : Un graphe pondéré $G = (V, E, \omega)$ donné par listes d'adjacences, avec $\omega(E) \subset \mathbb{R}_+$, un sommet s
Sortie : Les distances $\delta(s, t)$ pour tout $t \in V$
$d[t] \leftarrow +\infty$ pour tout $t \in V$; $d[s] \leftarrow 0$;
$H \leftarrow \emptyset$; $F \leftarrow \{s\}$;
tant que $F \neq \emptyset$ **faire**
  $u \leftarrow$ Retirer de F un sommet v vérifiant $d[v]$ minimal parmi les sommets de F;
  pour *tout voisin* v *de* u **faire**
    si v *n'est ni dans* F *ni dans* H **alors**
      Ajouter v à F
    $d[v] \leftarrow \min(d[v], d[u] + \omega(u, v))$
  Ajouter u à H
Renvoyer d

Terminaison de l'algorithme. L'algorithme fait un parcours du graphe depuis le sommet s : on peut retrouver le parcours générique du chapitre précédent en supprimant ce qui a trait au tableau d, et en remplaçant H par l'ensemble des sommets déja vus. Ainsi, l'algorithme termine.

Correction de l'algorithme. Faisons déja plusieurs observations :

— l'algorithme, pour travailler sur le tableau d, se contente de relâcher des arcs : on aura donc $d[t] \geq \delta(s, t)$ pour tout sommet t à la fin de l'algorithme.

— l'algorithme faisant notamment un parcours générique, à la fin de l'algorithme, tous les sommets t accessibles depuis s sont dans l'ensemble H, et vérifient tous $d[t] < +\infty$.

Montrons maintenant que l'algorithme est correct, via la proposition suivante :

Proposition 14.17. *Dans l'algorithme de Dijkstra, la propriété « tout sommet t de H vérifie $d[t] = \delta(s, t)$ » est un invariant de la boucle* **tant que**.

Démonstration. — la propriété est vraie avant la boucle, car l'ensemble H est vide.

— Pour montrer l'hérédité, il suffit de montrer qu'un sommet u de F vérifiant $d[u]$ minimal vérifie en fait $d[u] = \delta(s, u)$. Distinguons deux cas :

  — si $u = s$ (c'est le premier tour de boucle), on a $d[s] = \delta(s, s) = 0$;

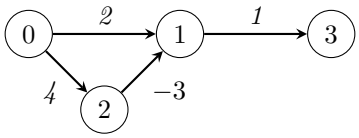
— sinon, considérons un plus court chemin de s à u, noté $s \rightsquigarrow u$. Considérons sur ce chemin le premier sommet y qui n'appartient pas à H (qui existe bien, car $s \in H$ et $u \notin H$) et x son prédeceseur. Le chemin se décompose en $s \rightsquigarrow x \to y \rightsquigarrow u$ (on peut avoir $s = x$ et/ou $y = u$). Puisque x est dans H, $\delta(s,x) = d[x]$ (hypothèse de récurrence) et lorsqu'on a considéré x dans la boucle, on a rajouté y à F s'il n'y était pas déja et relaché l'arc $x \to y$: ainsi $d[y] \leq \delta(s,x) + \omega(x,y) = \delta(s,y)$, donc il y a en fait égalité. Le morceau de s à y étant un plus court chemin et les poids positifs, on a $\delta(s,y) \leq \delta(s,u)$. Puisque $d[u]$ minimal parmi les sommets dans F, on a $d[u] \leq d[y]$. Ainsi :

$$d[u] \leq d[y] = \delta(s,y) \leq \delta(s,u) \leq d[u]$$

Il y a donc égalité partout, et en particulier $\delta(s,u) = d[u]$.

□

Remarque 14.18. *Le fait que les arcs aient un poids positif est un ingrédient essentiel de la preuve précédente, pour pouvoir affirmer que $\delta(s,y) \leq \delta(s,u)$. L'exemple minimaliste suivant montre un graphe pour lequel l'algorithme de Dijkstra lancé sur le sommet $s = 0$ ne fonctionne pas : l'algorithme relâche successivement les arcs $(0,1)$, $(1,3)$, $(0,2)$, $(2,1)$. Il faudrait relâcher à nouveau l'arc $(1,3)$ pour que le tableau des distances soit correct.*



Complexité de l'algorithme. La complexité de l'algorithme dépend de la structure de données utilisée pour gérer l'ensemble F du pseudo-code.

- Si on utilise simplement un tableau de booléens pour marquer les éléments de F, retirer l'élément de F vérifiant $d[u]$ minimal a un coût $O(n)$. Cette action est effectuée au plus n fois pour un coût total $O(n^2)$. À part cela, la complexité de l'algorithme est la même que celle d'un parcours en largeur classique : on parcourt au plus une fois toutes les listes d'adjacence, pour un coût total $O(a+n) = O(n^2)$ car $a = O(n^2)$. Ainsi avec cette implémentation l'algorithme de Dijkstra a un coût $O(n^2)$.
- Si on utilise une file de priorité min pour gérer F, implémentée avec un tas (min), chaque opération de file de priorité a une complexité $O(\log n)$. Il faut effectuer une opération sur la file de priorité lorsqu'on retire le minimum (affectation de u, donc au plus une fois par sommet) mais aussi lorsqu'on diminue $d[v]$ (donc au plus une fois par arc). Attention : il faut avoir implémenté l'opération de diminution de la clé d'un élément quelconque de la file. Ici, avec les sommets supposés être dans $[\![0, n-1]\!]$, il suffit d'utiliser un tableau `pos` dans lequel est stocké la position de chaque nœud i dans le tableau associé au tas. On accède ainsi facilement à la position de chaque nœud, qu'on peut diminuer en temps $O(\log n)$, en répercutant les permutations effectuées dans le tableau `pos`. Ainsi la complexité totale est $O((a+n)\log n)$.

La méthode à choisir dépend du graphe : pour un graphe « dense » (avec a proche de n^2), on aura intérêt à utiliser un tableau pour éviter les trop nombreuses opérations de file de priorité, par contre si le graphe est plus « creux » ($a = o(n^2/\log n)$), on aura intérêt à utiliser une file de priorité.

Remarque 14.19. *Une implémentation de la structure de file de priorité min où les opérations de diminution de clé ont une complexité amortie constante existe (et a été en fait inventée historiquement pour l'algorithme de Dijkstra) : les tas de Fibonacci. Avec cette implémentation, la complexité se réduit à $O(a + n\log n)$.*

Calcul effectif de plus courts chemins. L'adaptation est la même que pour l'algorithme de Bellman-Ford : il suffit d'utiliser un tableau de prédécesseurs.

Exemple. En figure 14.4 est représenté le déroulement de l'algorithme de Dijkstra sur un graphe à 5 sommets, dont la source s. Les arcs en gras représentent l'évolution du tableau des prédecesseurs.

14.5 Plus courts chemins pour tous couples de sommets

On souhaite maintenant calculer $\delta(s,t))$ pour tout $(s,t) \in V^2$. Avec un graphe représenté par listes d'adjacence, il est possible d'appliquer n fois les algorithmes de Bellman-Ford et de Dijkstra. Ce dernier reste très efficace si le graphe est « creux » (complexité $O((a+n)n\log n)$) mais suppose les poids positifs. On va donner ici deux algorithmes fonctionnant sur des graphes donnés par matrices d'adjacence. Le premier est assez simple et possède une complexité $O(n^3 \log n)$. Le second (Floyd-Warshall) est plus astucieux et plus efficace, car sa complexité se réduit à $O(n^3)$.

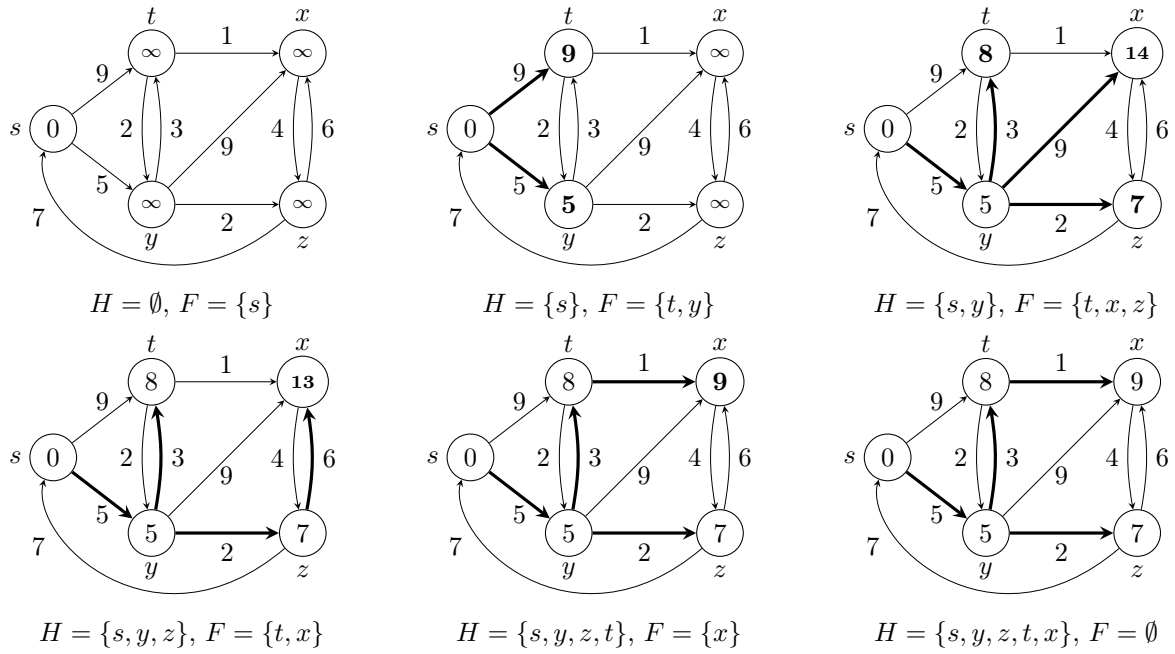


FIGURE 14.4 – Déroulement de l'algorithme de Dijkstra

14.5.1 Multiplication de matrices (HP)

Soit M la matrice d'adjacence, et considérons la suite de matrice $(D_i)_{i\geq 1}$ définie par

$$D_1 = M \quad \text{et} \quad D_m = (d^m_{i,j})_{0\leq i,j\leq n-1} \text{ avec } \begin{array}{rcll} d^m_{i,j} & = & \min\{d^{m-1}_{i,j}\} \cup \{d^{m-1}_{i,k} + \omega(k,j) \mid k \in [\![0, n-1]\!]\} & \\ & = & \min\{d^{m-1}_{i,k} + \omega(k,j) \mid k \in [\![0, n-1]\!]\} & \text{car } \omega_{j,j} = 0 \end{array}$$

Proposition 14.20. *Avec cette construction, $d^m_{i,j}$ est la longueur d'un plus court chemin de longueur au plus m dans le graphe, entre les sommets i et j.*

Démonstration. Immédiat par récurrence. □

Ainsi, si le graphe ne possède pas de circuit de poids strictement négatif, il suffit de calculer la matrice D_{n-1} pour obtenir tous les $\delta(i,j)$. En suivant la définition, le calcul de D_m à partir de D_{m-1} se fait avec une complexité $O(n^3)$: il y a n^2 coefficients, et le calcul de chaque $d^m_{i,j}$ se fait en complexité $O(n)$. On en déduit donc un algorithme de complexité $O(n^4)$ pour le calcul de D_{n-1}. On peut en fait accélérer le processus en calquant la multiplication matricielle usuelle :

Proposition 14.21. $(\mathbb{R} \cup \{+\infty\}, \min, +, +\infty, 0)$ *est un semi-anneau commutatif, c'est-à-dire que :*

- $(\mathbb{R} \cup \{+\infty\}, \min)$ *est un monoïde commutatif, de neutre $+\infty$;*
- $(\mathbb{R} \cup \{+\infty\}, +)$ *est également un monoïde commutatif, de neutre 0 ;*
- $+$ *est distributive par rapport à* $\min$ *;*
- $+\infty$ *est absorbant pour $+$.*

Démonstration. Tout s'écrit facilement, vérifions la distributivité de $+$ sur min : $a + \min(b,c) = \min(a+b, a+c)$. L'associativité de min est aussi immédiate : $\min(a, \min(b,c)) = \min(a,b,c) = \min(\min(a,b),c)$. □

La proposition précédente montre que l'on peut définir un produit matriciel associatif pour les matrices de $\mathcal{M}_n(\mathbb{R} \cup \{+\infty\})$. La loi de multiplication de deux telles matrices $A = (a_{i,j})_{0\leq i,j<n}$ et $B = (b_{i,j})_{0\leq i,j<n}$ est la suivante : $A \star B = C$ où $C = (c_{i,j})_{0\leq i,j<n}$ avec $c_{i,j} = \min\{a_{i,k} + b_{k,j} \mid 0 \leq k \leq n-1\}$. Le neutre pour ce produit est la matrice avec des zéros sur la diagonale et des $+\infty$ ailleurs. Avec ce produit, on a $D_i = M \star \cdots \star M = M^i$. On peut donc calculer $D_{n-1} = M^{n-1}$ par exponentiation rapide, avec une complexité $O(n^3 \log n)$. En fait, il nous suffit d'avoir M^k pour $k \geq n-1$, ce que fournit l'algorithme suivant :

Algorithme 14.22 : Algorithme de multiplication matricielle

Entrée : Un graphe pondéré $G = (V, E, \omega)$ donné par sa matrice d'adjacence M
Sortie : La matrice $(\delta(i,j))_{0\leq i,j<n}$ des plus courtes distances entre deux sommets quelconques du graphe
$A \leftarrow M$;
$k \leftarrow 1$;
tant que $k < n-1$ **faire**
  $A \leftarrow A \star A$;
  $k \leftarrow 2 \times k$
Renvoyer A

Détection d'un circuit de poids strictement négatif. Pour détecter un circuit de poids strictement négatif, on peut procéder de manière similaire à l'algorithme de Bellman-Ford : s'il existe un tel circuit, alors il en existe un tel que tous les sommets soient distincts, excepté le premier sommet qui coïncide avec le dernier, notons le s_i. On a alors un chemin $s_i \rightsquigarrow s_i$ de poids strictement négatif, et de longueur au plus n. Ainsi, le coefficient diagonal de la matrice M^n en case (i,i) est strictement négatif. On peut donc légèrement modifier l'algorithme pour calculer M^k avec $k \geq n$ (condition $k < n$ à la place de $k < n-1$ dans la boucle), et vérifier s'il existe un coefficient diagonal strictement négatif.

Calcul effectif de plus courts chemins. Il existe plusieurs méthodes pour retrouver les plus courts chemins dans l'algorithme précédent, la plus économique en mémoire consiste là encore à stocker dans une *matrice de liaison* les prédécesseurs dans des plus courts chemins entre deux sommets. On pose donc $\pi_{i,j}$=le prédecesseur de j dans un plus court chemin de i à j, s'il existe. Initialement, $\pi_{i,j} = i$ si (i,j) est un arc du graphe, et a une valeur arbitraire sinon. Lors du calcul de A^2, si le coefficient en case (i,j) est abaissé (on a $a_{i,j} > a_{i,k} + a_{k,j}$), alors $\pi_{i,j}$ prend la valeur $\pi_{k,j}$. En fin d'algorithme, s'il existe un chemin de i à j pour $i \neq j$, alors $\pi_{i,j}$ contient le dernier sommet intermédiaire (ou i si le plus court chemin est l'arc $i \to j$). Le calcul de la matrice de liaison ne change pas la complexité en temps comme en mémoire, et il est facile de calculer effetivement un plus court chemin entre i et j en remontant depuis j.

Exemple. Le graphe de la figure 14.5 est un graphe d'ordre 5, il suffit donc de faire deux multiplications pour obtenir les poids des plus courts chemins (une troisième permet de s'assurer qu'il n'y a pas de circuit de poids strictement négatif). Seuls les coefficients pertinents de la matrice $\Pi = (\pi_{i,j})_{0\leq i,j<5}$ sont indiqués. Une fois le calcul effectué, on a par exemple qu'un plus court chemin entre les sommets 1 et 4 est de poids -1. Pour le calcul effectif d'un plus court chemin, on voit que $\pi_{1,4} = 0$, $\pi_{1,0} = 3$ et $\pi_{1,3} = 1$, donc $1 \to 3 \to 0 \to 4$ convient.

Remarque 14.23. *On peut se demander s'il est possible d'utiliser un algorithme de multiplication sous-cubique (comme l'agorithme de Strassen, de complexité $O(n^{\log_2(7)})$) pour améliorer la complexité précédente. En fait, la réponse est oui, mais pas tel quel car $\mathbb{R} \cup \{+\infty\}$ n'a qu'une structure de semi-anneau et l'algorithme de Strassen demande d'effectuer des soustractions. Mais ces questions sortent très largement du cadre de ce cours !*

14.5.2 Algorithme de Floyd-Warshall

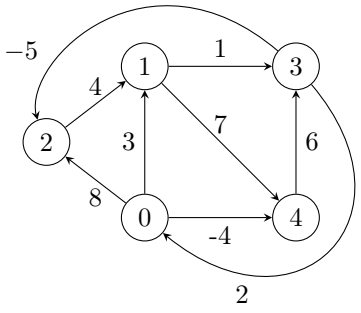
L'algorithme de Floyd-Warshall est un autre algorithme de programmation dynamique qui calcule une suite de matrices (M_i), mais le passage de M_i à M_{i+1} se fait simplement avec une complexité $O(n^2)$.

Définition 14.24. *Dans l'algorithme de Floyd-Warshall sur un graphe $G = (V, E, \omega)$, de matrice d'adjacence M, on pose pour tout $k \in [\![0, n]\!]$, $M_k = (m^k_{i,j})$ avec $m^k_{i,j}$ le poids minimal d'un chemin de i à j dont tous les sommets intermédiaires (c'est-à-dire i et j exclus) sont dans $[\![0, k-1]\!]$ (on a donc notamment $M_0 = M$ car les chemins sans sommets intermédiaires sont simplement les arcs).*

Ces matrices sont à valeurs dans $\mathbb{R} \cup \{\pm\infty\}$, mais bien sûr la valeur $-\infty$ ne se produit que dans le cas où il y a un circuit de poids total strictement négatif. En excluant ce cas, la proposition suivante indique comment calculer M^{k+1} à partir de M^k :

Proposition 14.25. *En l'absence de circuit de poids strictement négatif dans le graphe, on a pour tout $(i,j,k) \in [\![0, n-1]\!]^3$: $m^{k+1}_{i,j} = \min(m^k_{i,j}, m^k_{i,k} + m^k_{k,j})$.*

Démonstration.
- S'il n'y a pas de chemin entre i et j ne passant que par des sommets de $[\![0,k]\!]$, $m^k_{i,j} = m^{k+1}_{i,j} = +\infty$, et l'un des deux $m^k_{i,k}$ ou $m^k_{k,j}$ vaut aussi $+\infty$ (car sinon on aurait un chemin entre i et k et un autre entre k et j, dont la concaténation fournirait un chemin entre i et j).



Initialement : $A = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$ $\Pi = \begin{pmatrix} . & 0 & 0 & . & 0 \\ . & . & . & 1 & 1 \\ . & 2 & . & . & . \\ 3 & . & 3 & . & . \\ . & . & . & 4 & . \end{pmatrix}$

Première multiplication : $A^2 = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$ $\Pi = \begin{pmatrix} . & 0 & 0 & 4 & 0 \\ 3 & . & 3 & 1 & 1 \\ . & 2 & . & 1 & 1 \\ 3 & 2 & 3 & . & 0 \\ 3 & . & 3 & 4 & . \end{pmatrix}$

Deuxième multiplication : $A^4 = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$ $\Pi = \begin{pmatrix} . & 2 & 3 & 4 & 0 \\ 3 & . & 3 & 1 & 0 \\ 3 & 2 & . & 1 & 0 \\ 3 & 2 & 3 & . & 0 \\ 3 & 2 & 3 & 4 & . \end{pmatrix}$

FIGURE 14.5 – Calcul de plus courts chemins dans un graphe, à l'aide de multiplications matricielles

- Sinon, un chemin de poids minimal entre i et j ne passant que par des sommets de $[\![0,k]\!]$ peut être supposé ne passer qu'au plus une fois par k (il suffit de supprimer le circuit - nécessairement de poids nul - entre les première et dernière occurences de k dans un chemin de poids minimal pour en obtenir un de même poids dans lequel k apparaît au plus une fois). Si ce chemin ne passe pas par k, on a $m_{i,j}^{k+1} = m_{i,j}^k$, et sinon le chemin se décompose en $i \overset{c_1}{\rightsquigarrow} k \overset{c_2}{\rightsquigarrow} j$ avec c_1 et c_2 des chemins dont les sommets intermédiaires sont dans $[\![0,k-1]\!]$, ainsi $m_{i,j}^{k+1} = m_{i,k}^k + m_{j,k}^k$.

□

On obtient ainsi un algorithme de complexité $O(n^3)$ pour le calcul de tous les $\delta(i,j)$. La remarque suivante indique que l'on peut se contenter de mettre à jour une unique matrice :

Remarque 14.26. *En l'abscence de circuit de poids total négatif, on a également, pour tout* $(i,j,k) \in [\![0,n-1]\!]^3$ *:* $m_{i,j}^{k+1} = \min(m_{i,j}^k, m_{i,k}^k + m_{k,j}^k) = \min(m_{i,j}^k, m_{i,k}^{k+1} + m_{k,j}^k) = \min(m_{i,j}^k, m_{i,k}^k + m_{k,j}^{k+1}) = \min(m_{i,j}^k, m_{i,k}^{k+1} + m_{k,j}^{k+1})$.

En effet, remplacer $m_{i,k}^k$ par $m_{i,k}^{k+1}$ ne change rien, car il n'y a pas de circuit de poids strictement négatif bouclant sur k. Voici donc l'algorithme de Floyd-Warshall :

Complexité. Elle est clairement de $O(n^3)$, en temps comme en mémoire.

Détection des circuits de poids total négatif. S'il existe un circuit de poids total négatif, prenons en un sans sommet en double excepté les extrémités. Soit i le sommet aux extrémités, on aura alors $a_{i,i} < 0$ à la fin de l'algorithme. Il suffit donc de tester l'existence d'un élément diagonal strictement négatif à la fin de l'algorithme pour tester l'existence d'un circuit de poids total strictement négatif.

Algorithme 14.27 : Algorithme de Floyd-Warshall

Entrée : Un graphe pondéré $G = (V, E, \omega)$ donné par sa matrice d'adjacence M
Sortie : La matrice $(\delta(i,j))_{0\leq i,j<n}$ des plus courtes distances entre deux sommets quelconques du graphe

```
A ← copie(M);
pour tout k entre 0 et n − 1 faire faire
    pour tout i entre 0 et n − 1 faire faire
        pour tout j entre 0 et n − 1 faire faire
            a_{i,j} ← min(a_{i,j}, a_{i,k} + a_{k,j})
Renvoyer A
```

Calcul effectif des plus courts chemins. Parmi d'autres méthodes, on peut là encore calculer une matrice de liaison, comme dans l'algorithme par multiplications matricielles. Lorsqu'on fait $a_{i,j} \leftarrow a_{i,k} + a_{k,j}$, on effectue parallèlement $\pi_{i,j} \leftarrow \pi_{k,j}$, la matrice $\Pi = (\pi_{i,j})$ étant initialisée comme dans l'algorithme précédent.

Exemple. On reprend le même exemple que précédemment, pour naturellement obtenir le même résultat.

Application : fermeture transitive d'un graphe. Une dérivation de l'algorithme de Floyd-Warshall permet de répondre facilement au problème de l'accessibilité dans un graphe, et fournit l'algorithme de Warshall (historiquement antérieur à l'algorithme de Floyd-Warshall...)

Définition 14.28. *Soit $G = (V, E)$ un graphe orienté, non pondéré. On appelle fermeture transitive de G le graphe $\tilde{G} = (V, \tilde{E})$, où pour $u \neq v$ deux sommets de $\tilde{G}$, (u, v) appartient à $\tilde{E}$ s'il existe un chemin de u à v dans G.*

Algorithme 14.29 : Algorithme de Warshall : calcul de la fermeture transitive d'un graphe

Entrée : Un graphe $G = (V, E)$ donné par sa matrice d'adjacence M
Sortie : La matrice $(b_{i,j})_{0\leq i,j<n}$ d'accessibilité dans G : $b_{i,j}$ est vrai si et seulement si il existe un chemin entre i et j dans G.

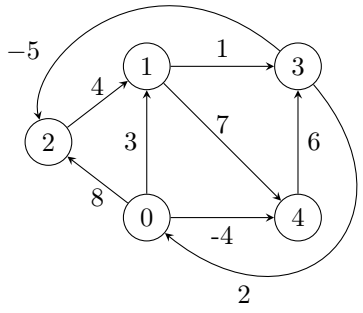
```
b_{i,j} ← Vrai si i = j ou m_{i,j} = 1, Faux sinon.;
pour tout k entre 0 et n − 1 faire faire
    pour tout i entre 0 et n − 1 faire faire
        pour tout j entre 0 et n − 1 faire faire
            b_{i,j} ← b_{i,j} ∨ (b_{i,k} ∧ b_{k,j})
Renvoyer A
```

14.5.3 Résumé des algorithmes de plus courts chemins

On a vu dans ce chapitre 4 algorithmes, voici un résumé des complexités.

	Bellman-Ford	Dijkstra	Multiplication matricielle	Floyd-Warshall
Limitation	–	poids positifs	–	–
problème (2)	$O(n(a+n))$	$O((a+n)\log n)$ (tas min) $O(n^2)$ (tableau)	–	–
problème (4)	$O(n^2(a+n))$	$O((a+n)n\log n)$ (tas min) $O(n^3)$ (tableau)	$O(n^3 \log n)$	$O(n^3)$

Remarque 14.30. *Pour la résolution du problème (4) sur un graphe dense, la complexité asymptotique de l'algorithme de Dijkstra utilisé n fois (en gérant la file de priorité avec un tableau) est la même que celle de l'algorithme de Floyd-Warshall ($O(n^3)$). Néanmoins, la constante cachée dans le O est plus faible pour l'algorithme de Floyd-Warshall, et celui-ci a le mérite de s'appliquer même s'il y a des arcs de poids négatifs. Pour un graphe dense, on préférera l'algorithme de Floyd-Warshall pour calculer une solution au problème (4) !*



$$M_0 = M = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} . & 0 & 0 & . & 0 \\ . & . & . & 1 & 1 \\ . & 2 & . & . & . \\ 3 & . & 3 & . & . \\ . & . & . & 4 & . \end{pmatrix}$$

$$M_1 = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} . & 0 & 0 & . & 0 \\ . & . & . & 1 & 1 \\ . & 2 & . & . & . \\ 3 & 0 & 3 & . & 0 \\ . & . & . & 4 & . \end{pmatrix}$$

$$M_2 = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} . & 0 & 0 & 1 & 0 \\ . & . & . & 1 & 1 \\ . & 2 & . & 1 & 1 \\ 3 & 0 & 3 & . & 0 \\ . & . & . & 4 & . \end{pmatrix}$$

$$M_3 = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} . & 0 & 0 & 1 & 0 \\ . & . & . & 1 & 1 \\ . & 2 & . & 1 & 1 \\ 3 & 2 & 3 & . & 0 \\ . & . & . & 4 & . \end{pmatrix}$$

$$M_4 = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} . & 0 & 3 & 1 & 0 \\ 3 & . & 3 & 1 & 0 \\ 3 & 2 & . & 1 & 0 \\ 3 & 2 & 3 & . & 0 \\ 3 & 2 & 3 & 4 & . \end{pmatrix}$$

$$M_5 = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} . & 2 & 3 & 4 & 0 \\ 3 & . & 3 & 1 & 0 \\ 3 & 2 & . & 1 & 0 \\ 3 & 2 & 3 & . & 0 \\ 3 & 2 & 3 & 4 & . \end{pmatrix}$$

FIGURE 14.6 – Calcul de plus courts chemins dans un graphe, à l'aide de l'algorithme de Floyd-Warshall

14.6 Arbre couvrant de poids minimal

Définition 14.31. *Soit $G = (V, E, \omega)$ un graphe pondéré, non orienté, connexe, dont la fonction de pondération est à valeurs dans $\mathbb{R}_+$. On appelle graphe couvrant de G un sous-graphe connexe $G' = (V, E')$ tel que $E' \subseteq E$. On appelle poids du graphe couvrant la somme $\sum_{e \in E'} \omega(e)$. Si G' est un arbre, on dit que c'est un arbre couvrant.*

Le problème de « l'arbre couvrant minimal » est de trouver un graphe couvrant de poids minimal. La propriété suivante indique que l'on peut chercher un arbre.

Proposition 14.32. *En reprenant les notations de la définition précédente, il existe un graphe couvrant de G de poids minimal qui est un arbre.*

Démonstration. Soit G' un graphe couvrant de poids minimal de G (ce graphe existe car il existe au moins un graphe couvrant de G : G lui-même). En notant $n = |V|$, G' possède au moins $n-1$ arêtes car il est connexe. On peut de plus supposer G' minimal en nombre d'arêtes parmi les arbres couvrants de poids minimal. S'il existait un cycle dans G', on pourrait retirer une arête du cycle sans perdre la connexité en diminuant le poids : c'est absurde. Donc G' est un arbre. □

En pratique, si les valeurs de ω sont strictement positives, un graphe couvrant minimal est un arbre. S'il y a des arêtes de poids nul, un graphe couvrant minimal peut ne pas être un arbre, mais il existe au moins un graphe couvrant minimal qui est un arbre.

14.6.1 Algorithme de Prim (HP)

L'algorithme 14.33 (algorithme de Prim), très proche de l'algorithme de Dijkstra, permet de trouver un tel arbre.

Algorithme 14.33 : Algorithme de Prim

Entrée : Un graphe pondéré, non orienté, connexe $G = (V, E, \omega)$ donné par listes d'adjacence, avec $\omega(E) \subset \mathbb{R}_+$, un sommet s
Sortie : Un ensemble d'arêtes formant un arbre couvrant minimal

```
n ← |V| ; d[t] ← +∞ pour tout t ∈ V ; d[s] ← 0;
p[t] ← t pour tout t ∈ V;
H ← ∅ ; F ← {s};
E' ← ∅;
pour i de 0 à n − 1 faire
    u ← Retirer de F un sommet v vérifiant d[v] minimal parmi les sommets de F;
    si u ≠ s alors
        E' ← E' ∪ {{u, p[u]}}
    pour tout voisin v de u faire
        si v n'est ni dans F ni dans H alors
            Ajouter v à F
        si d[v] > ω(u, v) alors
            d[v] ← ω(u, v);
            p[v] ← u
    Ajouter u à H
Renvoyer E'
```

Théorème 14.34. *Avec E' l'ensemble renvoyé par l'algorithme de Prim, (V, E') est un arbre couvrant de poids minimal.*

Démonstration. L'algorithme réalise un parcours du graphe, qui atteindra tous les sommets car G est connexe : les n sommets vont donc passer par F. L'algorithme ajoute à E' une arête pour chaque sommet excepté le sommet de départ, ainsi on a à la fin de l'algorithme $|E'| = n-1$, de plus (V, E') est connexe car tous les sommets sont raccordés au sommet initial s. Donc (V, E') est un arbre. Notons $s = v_0, v_1, \dots, v_{n-1}$ les sommets dans l'ordre de leur ajout à H, et pour $i \in \{1, \dots, n-1\}$ notons e_i l'arête ajoutée à E', reliant v_i au graphe induit par $\{v_j \mid j < i\}$. Supposons que (V, E') ne soit pas un arbre couvrant de poids minimal et considérons un arbre (V, A) couvrant de poids minimal, et notons

$$i = \max\{j \mid e_1, \dots, e_j \in A\}$$

On peut sans perte de généralités supposer que (V, A) est un arbre couvrant minimal pour lequel i est maximal, et on va aboutir à une absurdité. L'arête e_{i+1}, reliant v_{i+1} à un sommet v_j (avec $j \leq i$) n'est pas dans A. Travaillons sur le graphe $(V, A \cup \{e_{i+1}\})$: ce graphe possède un cycle car (V, A) est un arbre. En considérant le cycle comme un chemin bouclant de v_j sur lui même, en terminant par l'arête e_{i+1}, notons v le premier sommet qui n'est pas dans $\{v_0, \dots, v_i\}$, et e l'arête le reliant au sommet précédent.

— $\omega(e) \geq \omega(e_{i+1})$, car sinon e_{i+1} n'aurait pas été rajoutée à E'. En effet, on aurait eu $d[v] < d[v_{i+1}]$ à ce moment là ;

— $\omega(e) \leq \omega(e_{i+1})$, car sinon $(V, A \cup \{e_{i+1}\}\backslash\{e\})$ serait un arbre couvrant de poids strictement inférieur à celui de (V, A).

Ainsi, e a le même poids que e_{i+1}, et $(V, A \cup \{e_{i+1}\}\backslash\{e\})$ est un arbre couvrant de même poids que (V, A) mais tel que $\max\{j \mid e_1, \ldots, e_j \in A \cup \{e_{i+1}\}\backslash\{e\}\} > i$, ce qui est absurde par hypothèse. Ainsi (V, E') est un arbre couvrant de poids minimal.

□

Proposition 14.35. *La complexité de l'algorithme précédent est en $O(n^2)$ avec une implémentation avec tableaux, $O(a \log n)$ avec une file de priorité implémentée avec un tas binaire.*

Démonstration. La complexité est exactement la même que dans l'algorithme de Dijkstra, avec ici $n = O(a)$ car le graphe est connexe. □

Exemple. Voici un exemple, où l'on part de s pour trouver un arbre couvrant minimal. On a fait figurer dans chaque sommet u qui n'est pas dans H la valeur $d[u]$, les arêtes en gras forment l'arbre couvrant minimal.

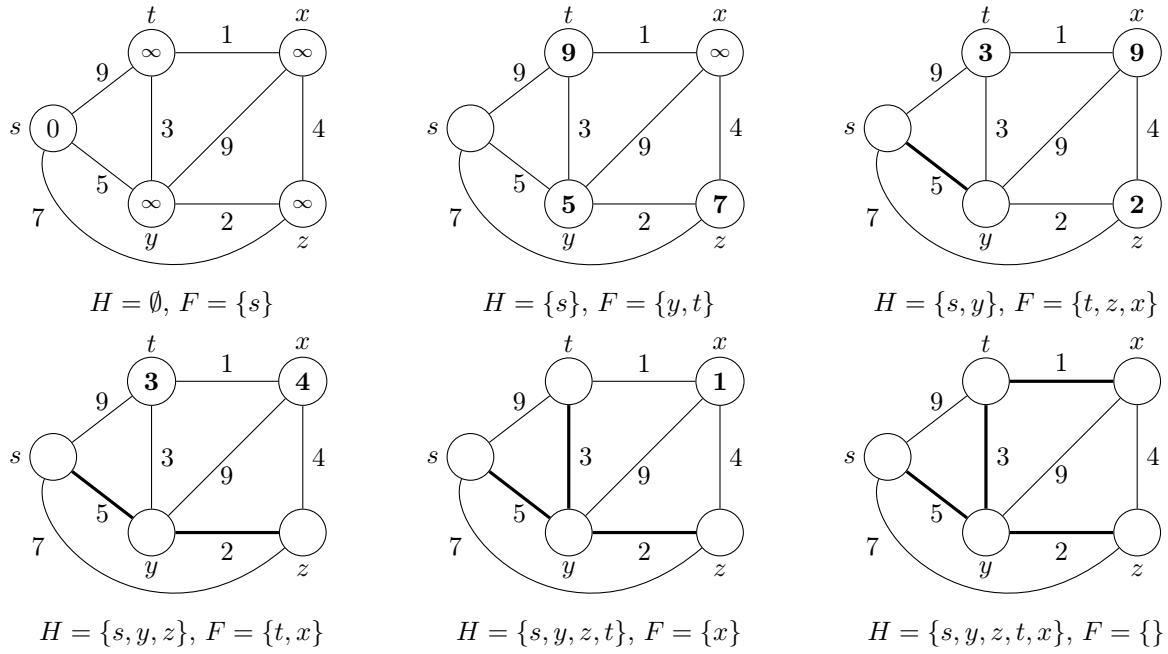


FIGURE 14.7 – Déroulement de l'algorithme de Prim : le poids minimal d'un arbre couvrant est 11

14.6.2 Algorithme de Kruskall

L'algorithme de Kruskall (algorithme 14.36), un peu différent, permet lui aussi de calculer un graphe couvrant minimal. C'est également un algorithme glouton.

Algorithme 14.36 : Algorithme de Kruskall

Entrée : Un graphe pondéré, non orienté, connexe $G = (V, E, \omega)$ donné par listes d'adjacence, avec $\omega(E) \subset \mathbb{R}_+$
Sortie : Un ensemble d'arêtes formant un arbre couvrant minimal
$\tilde{E} \leftarrow$ arêtes de E, triées par valeurs croissantes de ω;
$A \leftarrow \emptyset$; `#arêtes de l'ACM en construction`
pour *toute arête a de $\tilde{E}$* **faire**
  si $(V, A \cup \{a\})$ *est sans cycle* **alors**
    $A \leftarrow A \cup \{a\}$
Renvoyer A

Terminaison de l'algorithme de Kruskall. Elle est évidente.

Correction de l'algorithme de Kruskall. Montrons d'abord que l'algorithme renvoie bien un ensemble d'arêtes d'un arbre couvrant. Cela repose sur la propriété suivante.

Proposition 14.37. *Soit $G = (V, E)$ un graphe acyclique. Lorsqu'on rajoute une arête à E :*

- *soit le nombre de composantes connexes est réduit de 1, et G reste sans cycle ;*
- *soit le nombre de composantes connexes reste le même, et G possède maintenant un cycle.*

Démonstration. Tout dépend si l'arête rajoutée relie deux sommets situés dans la même composante connexe ou non :

- si c'est le cas, notons k le nombre de sommets de la composante connexe en question. Puisque le graphe est acyclique, cette composante possède $k-1$ arêtes puisqu'elle est connexe acyclique (on se réfère au travail effectué dans le chapitre sur les graphes non orientés). Lui rajouter une arête donne un graphe qui n'est plus acyclique, mais ne change pas le nombre de composantes connexes du graphe total ;
- sinon, notons k et k' le nombre de sommets des composantes que l'on va relier. Par hypothèse, ces composantes possèdent $k-1$ et $k'-1$ arêtes (ce sont des arbres). Une fois reliées, le sous-graphe obtenu est connexe, à $k+k'$ sommets et $k+k'-1$ arêtes : c'est également un arbre, donc connexe acyclique. Le graphe total possède bien une composante connexe de moins.

□

Proposition 14.38. *L'algorithme de Kruskall renvoie l'ensemble des arêtes d'un arbre couvrant*

Démonstration. Déja, le graphe (V, A) obtenu est acyclique (cette propriété est un invariant de l'algorithme). Reste à montrer qu'il est connexe. Supposons qu'il possède au moins deux composantes connexes différentes, et soient u et v deux sommets appartenant à des composantes distinctes. Le graphe $G = (V, E)$ étant connexe, il existe dans G un chemin de u à v. Le long de ce chemin, on peut trouver une arête qui relie deux sommets appartenant à des composantes distinctes dans (V, A) : au moment où elle a été considérée par l'algorithme, elle aurait du être rajoutée à A : c'est absurde. □

Proposition 14.39. *L'algorithme de Kruskall est correct.*

Démonstration. Il reste à montrer que le graphe (V, A) obtenu est un ACM de G. Supposons que ce ne soit pas le cas. Notons $e_1, \dots, e_{n-1}$ les arêtes de A, supposées ordonnées par valeurs de ω croissantes. Considérons un ACM (V, A') de G, qui est donc de poids strictement inférieur à (V, A), et considérons e_i la première arête qui n'appartient pas à A'. On suppose de plus que parmi les ACM de G, (V, A') est celui qui maximise l'indice i : c'est un ACM possédant un maximum d'arêtes de petits poids en commun avec (V, A). Rajoutons e_i à A' : $(V, A' \cup \{e_i\})$ possède un cycle, sur lequel se situe une arête e qui n'est pas dans A. $(V, A' \cup \{e_i\} \backslash \{e\})$ est encore un arbre couvrant de G, donc $\omega(e) \leq \omega(e_i)$. Si on avait égalité, cela contredirait l'hypothèse sur le fait que A' partage un maximum d'arêtes de plus petits poids avec A, donc $\omega(e) < \omega(e_i)$. Mais, lorsque e a été considéré par l'algorithme, A ne contenait que $e_1, \dots, e_{i-1}$, et e aurait du lui être rajouté, ne créant pas de cycle car n'en créant pas dans A' : c'est absurde. □

Implémentation et complexité. Examinons les coûts de l'algorithme :

- récupérer la liste d'arêtes se fait en $O(a)$ si le graphe est stocké sous forme de listes d'adjacence, $O(n^2)$ s'il est stocké sous forme de tableaux ;
- trier cette liste a un coût $O(a \log a) = O(a \log n)$ si on utilise un algorithme comme le tri fusion ;
- pour pourvoir tester si une arête crée un cycle dans le graphe, il faut pouvoir tester si les deux extrémités sont dans la même composante connexe. Plutôt que de faire des parcours inutiles de graphe, il suffit de donner un numéro à chaque composante connexe (initialement, le sommet i se trouve seul dans la composante i), et de stocker pour chaque sommet le numéro de sa composante. Lorsqu'on rajoute une arête à A, il suffit de donner le même numéro à tous les sommets des deux composantes fusionnées. Avec une implémentation naïve, cette étape se fait en $O(n)$, et devra être réalisée $O(n)$ fois, soit un surcoût $O(n^2)$. Ce surcoût n'est pas gênant si on travaille avec un graphe donné par matrice d'adjacence, néanmoins cela devient le coût majoritaire si le graphe est plutôt creux.

Structure d'Union-Set. Il existe une structure appelée « union-set », ou « partition-fusion » en français, permettant de gérer efficacement le troisième point ci-dessus. Plus précisément, on s'intéresse aux les éléments de $[\![0, n-1]\!]$, dont on veut pouvoir gérer facilement les *partitions*. Les opérations de la structure sont :

- `creer n` : renvoie une structure `u` où chaque élément de $[\![0, n-1]\!]$ est seul dans sa partie.
- `rep u x` : avec `u` la structure, renvoie le représentant de $x \in [\![0, n-1]\!]$ (deux éléments ont même représentant si et seulement si ils sont dans la même partie. À la création, le représentant de x est x lui-même)
- `fusion u x y` : *fusionne* les deux parties de $[\![0, n-1]\!]$ contenant x et y, supposés appartenir à deux parties différentes.

Une implémentation efficace de cette structure peut se faire de sorte que la complexité *amortie* des opérations soit quasi-constante, de sorte que la troisième étape a finalement un coût $O(a)$ et devient négligeable par rapport au tri des arêtes. Elle sera vue en TP !

Remarque 14.40. *Dans certains cas, on peut trier plus rapidement qu'avec le tri fusion : c'est le cas par exemple si on sait que les poids des arêtes sont des entiers pas trop grands (la borne est supposée connue), on peut utiliser le tri par comptage. Dans ces conditions, l'algorithme de Kruskall devient meilleur que l'algorithme de Prim, car la complexité du tri des arêtes diminue.*

Chapitre 15

Preuves en logique

15.1 Introduction

On a vu en première année une méthode pour prouver qu'une formule logique φ donnée était une tautologie : il s'agissait essentiellement de vérifier que la table de vérité de cette formule ne contient que des uns. Il existe plusieurs algorithmes capables d'effectuer ce calcul, néanmoins on n'en connaît aucun qui soit de complexité polynomiale en la taille de φ. C'est en effet l'objet de la célèbre conjecture $\mathrm{P} = \mathrm{NP}$ [1]. Sous l'hypothèse $\mathrm{P} \neq \mathrm{NP}$, il est vain d'essayer de trouver un algorithme de complexité polynomiale.

L'algorithme naïf a une complexité $O(2^n \times \ell)$ où n est le nombre de variables logiques apparaissant dans φ et ℓ la longueur de φ : pour chaque distribution de vérité, on vérifie que la formule logique est bien satisfaite par cette distribution, et cette vérification se fait en temps linéaire en la taille de la formule, avec les structures de données adéquates (par exemple si la formule logique est représenté sous forme d'arbre et une distribution de vérité sous forme de tableau).

Dans ce chapitre, on va développer une autre approche pour travailler avec des formules logiques, qui se rapproche davantage du raisonnement mathématique : lorsqu'on veut montrer un théorème, on énumère rarement toutes les possibilités pour les vérifier une à une. Pour modéliser le raisonnement du mathématicien, on introduit la notion de *preuve formelle*. Certains énoncés, les *axiomes*, sont tenus pour vrais, et d'autres énoncés sont obtenus à partir de ces axiomes et de *règles d'inférence*. La preuve est donc un jeu d'écriture permettant d'obtenir un énoncé donné en jouant avec les axiomes et les règles. Une preuve peut être courte à obtenir alors que la vérification par table de vérité est coûteuse. Par exemple, $(v_1 \wedge v_2 \wedge \cdots \wedge v_n) \Rightarrow (v_1 \vee v_2 \vee \cdots \vee v_n)$ est facile à prouver, alors que la vérification par table de vérité est exponentielle en n.

On appelle système de preuve le choix d'un ensemble d'axiomes et d'un ensemble de règles d'inférence. Il y a plusieurs systèmes usuels de preuves, permettant (ou non !) de prouver les mêmes théorèmes. Le programme officiel se place dans le cadre de la *déduction naturelle*. Dans ce cadre, l'implication joue un rôle central, symbolisant son importance en mathématiques (on suppose P, on en déduit Q, on a montré $P \Rightarrow Q$), alors qu'en première année $P \Rightarrow Q$ n'était qu'un synonyme de $\neg P \vee Q$.

15.2 Séquent

Définition 15.1. *Un séquent est un terme de la forme $\Gamma \vdash C$ où :*

- $\Gamma = H_1, \ldots, H_n$ *est un ensemble de formules appelées* hypothèses*, on appelle également Γ le contexte.*
- C *est une formule appelée* conclusion*.*

Remarque 15.2. *Dans l'idée, au séquent $\Gamma \vdash C$ est associé la formule logique $\Gamma \models C$, c'est-à-dire $(H_1 \wedge \cdots \wedge H_n) \Rightarrow C$. Si $n = 0$, on remarque que cette formule est sémantiquement équivalente à $\top \Rightarrow C$, elle-même équivalente à C.*

Exemple 15.3. *Voici des exemples de séquents :*

- $x \in \mathbb{R}_+, x^2 = 1 \vdash x = 1$ *;*
- *il pleut, s'il pleut le sol est mouillé $\vdash$ le sol est mouillé.*
- $a \Rightarrow b \vdash \neg b \Rightarrow \neg a$*.*
- $a \Rightarrow b \vdash b \Rightarrow a$*.*

1. https://en.wikipedia.org/wiki/P_versus_NP_problem

Remarque : *ce dernier séquent paraît « faux ». Pour le moment, les séquents ne sont qu'un jeu d'écriture, on discutera de leur « validité » plus tard, uniquement dans le cadre de la logique propositionnelle.*

15.3 Règles de la déduction naturelle pour les formules propositionnelles

En déduction naturelle, il n'y a pas d'axiomes. Les règles d'inférence sont notées sous la forme suivante :

$$\frac{\text{premisse 1} \qquad \text{premisse 2} \qquad \cdots}{\text{conclusion}} \text{ (nom de la règle)}$$

où les prémisses et la conclusion sont donnés sous forme de séquents : on dit qu'on dérive le séquent conclusion des prémisses par application de la règle. Il se peut qu'il n'y ait pas de prémisse : ce sera le cas de la règle d'hypothèse. On peut enchaîner l'application des règles pour construire un *arbre de preuve* où les feuilles (en haut de l'arbre) sont les prémisses et la racine (en bas de l'arbre) est le séquent à montrer, la conclusion d'une règle servant comme prémisse d'une autre. Un tel arbre ressemble à :

$$\dfrac{\text{prémisses} \qquad \dfrac{\text{prémisses}}{C_1}\,(R) \qquad \cdots \qquad \dfrac{\text{prémisses}}{C_k}\,(R')}{C}\,(R'')$$

Dans la suite Γ dénote un contexte quelconque (éventuellement vide), les autres lettres (φ, ψ,...) dénotent des formules logiques. $\Gamma \cup \{\varphi\}$ est également un contexte, qu'on préfère noter Γ, φ.

Règle d'hypothèse. La règle d'hypothèse permet d'extraire une formule d'un contexte.

$$\frac{}{\Gamma, \varphi \vdash \varphi}\ (\text{H})$$

Règle d'affaiblissement. Rajouter une formule au contexte ne change pas sa conclusion.

$$\frac{\Gamma \vdash \varphi}{\Gamma, \psi \vdash \varphi}\ (\text{Aff})$$

Règles pour $\wedge$. Pour les opérateurs op qui vont suivre, on a deux types de règles, l'introduction, notée I_{op} et l'élimination, notée E_{op}.

- Introduction de $\wedge$: si on peut prouver φ et ψ, on peut prouver $\varphi \wedge \psi$:

$$\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\ (I_\wedge)$$

- Élimination de $\wedge$: à l'inverse, si on peut prouver $\varphi \wedge \psi$, on peut prouver φ et ψ :

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi}\ (E^1_\wedge) \qquad\qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}\ (E^2_\wedge)$$

Exemple 15.4. *Montrons $\varphi \wedge \psi \vdash \psi \wedge \varphi$. Pour cette preuve, on note $\Gamma = \varphi \wedge \psi$.*

$$\dfrac{\dfrac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}\,(E^2_\wedge) \qquad \dfrac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi}\,(E^1_\wedge)}{\Gamma \vdash \psi \wedge \varphi}\,(I_\wedge)$$

Remarque 15.5. *Une fois une telle preuve effectuée, on peut utiliser le résultat dans d'autres preuves sans avoir à refaire toute la démarche. On obtient donc une nouvelle règle, appelée* règle dérivée*, qui est conséquence des règles précédentes. Par exemple, le résultat précédent peut s'écrire, avec Γ un contexte quelconque :*

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi \wedge \varphi}\ \textit{(commutativité de } \wedge)$$

Règles pour $\Rightarrow$.

- Introduction de l'implication : elle est essentielle en mathématiques. Si on montre ψ à partir de φ, cela revient à dire qu'on a montré $\varphi \Rightarrow \psi$.

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \Rightarrow \psi}\ (I_\Rightarrow)$$

- Élimination de l'implication (aussi appelée *modus ponens*) : de φ et $\varphi \Rightarrow \psi$, on déduit ψ.

$$\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \varphi \Rightarrow \psi}{\Gamma \vdash \psi}\ (E_\Rightarrow)$$

Exemple 15.6. *Montrons la transitivité de $\Rightarrow$, à savoir $(\varphi \Rightarrow \psi), (\psi \Rightarrow \chi) \vdash (\varphi \Rightarrow \chi)$. On note $\Gamma = (\varphi \Rightarrow \psi), (\psi \Rightarrow \chi)$.*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \varphi \vdash \varphi \Rightarrow \psi}\ (H) \quad \dfrac{}{\Gamma, \varphi \vdash \varphi}\ (H)}{\Gamma, \varphi \vdash \psi}\ (E_\Rightarrow) \quad \dfrac{}{\Gamma, \varphi \vdash \psi \Rightarrow \chi}\ (H)}{\Gamma, \varphi \vdash \chi}\ (E_\Rightarrow)}{\Gamma \vdash \varphi \Rightarrow \chi}\ (I_\Rightarrow)$$

Règles pour $\vee$.

- Introduction du ou logique : elle est symétrique de l'élimination du $\wedge$.

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}\ (I^1_\vee) \qquad\qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}\ (I^2_\vee)$$

- Élimination du ou logique. Elle est aussi couramment employée en mathématiques : Si on sait que $\varphi \vee \psi$, prouver $\varphi \Rightarrow \chi$ et $\psi \Rightarrow \chi$ suffit à montrer χ.

$$\frac{\Gamma \vdash \varphi \vee \psi \qquad \Gamma \vdash \varphi \Rightarrow \chi \qquad \Gamma \vdash \psi \Rightarrow \chi}{\Gamma \vdash \chi}\ (E_\vee)$$

La contradiction et la négation. On introduit le symbole $\bot$ pour la formule logique toujours fausse (*contradiction*), qui, d'un point de vue syntaxique, n'est qu'une formule logique supplémentaire. On introduit alors la négation comme une abbréviation : $\neg\varphi$ sera une abbréviation de $\varphi \Rightarrow \bot$.

- Élimination de $\bot$: essentiellement, il s'agit d'affirmer que d'une contradiction, on peut montrer n'importe quoi :

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \psi}\ (E_\bot)$$

Exercice 1. Les règles qui vont suivre sont en fait des règles dérivées, dans le sens où elle peuvent être déduites des règles précédentes. Les démontrer !

- Introduction de $\bot$: on introduit $\bot$ comme le résultat d'une formule logique et sa négation :

$$\frac{\Gamma \vdash \varphi \wedge \neg\varphi}{\Gamma \vdash \bot}\ (I_\bot)$$

- Introduction de $\neg$: si un contexte non vide permet de montrer $\bot$, ce même contexte privé d'une formule logique φ montre sa négation $\neg\varphi$.

$$\frac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ (I_\neg)$$

Exemple 15.7. *Montrons que $\vdash (\varphi \Rightarrow \psi) \Rightarrow \neg(\varphi \wedge \neg\psi)$. On note $\Gamma = (\varphi \Rightarrow \psi), \varphi \wedge \neg\psi$.*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \varphi \Rightarrow \psi}\ (H) \quad \dfrac{\dfrac{}{\Gamma \vdash \varphi \wedge \neg\psi}\ (H)}{\Gamma \vdash \varphi}\ (E^1_\wedge)}{\Gamma \vdash \psi}\ (E_\Rightarrow) \quad \dfrac{\dfrac{}{\Gamma \vdash \varphi \wedge \neg\psi}\ (H)}{\Gamma \vdash \psi}\ (E^2_\wedge)}{\Gamma \vdash \psi \wedge \neg\psi}\ (I_\wedge)}{\Gamma \vdash \bot}\ (I_\bot)$$

De ce dernier séquent, on déduit successivement $\varphi \Rightarrow \psi \vdash \neg(\varphi \wedge \neg\psi)$ *par* $I_\neg$ *et* $\vdash (\varphi \Rightarrow \psi) \Rightarrow \neg(\varphi \wedge \neg\psi)$ *par* $I_\Rightarrow$.

Exercice 2. Montrer la règle de la contraposée :

$$\frac{\Gamma \vdash \varphi \Rightarrow \psi}{\Gamma \vdash \neg\psi \Rightarrow \neg\varphi} \text{ (contraposée)}$$

Exercice 3. Montrer les règles dérivées suivantes :

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \neg\neg\varphi} \text{ (double négation)} \qquad \frac{\Gamma \vdash \neg\neg\neg\varphi}{\Gamma \vdash \neg\varphi} \text{ (triple négation)}$$

15.4 Correction de la déduction naturelle

En première année, nous avons défini les propositions logiques, et leur avons donné une sémantique, et défini ce qu'était une tautologie. Pour Γ un ensemble de formules logiques et φ une formule logique, on note $\Gamma \models \varphi$ si φ est conséquence de Γ, c'est-à-dire si $\left(\bigwedge_{\psi\in\Gamma} \psi\right) \Rightarrow \varphi$ est une tautologie. Lorsque Γ est vide, la conjonction est la formule vraie $\top$, et cette implication est équivalente à φ : écrire $\models \varphi$ signifie que φ est une tautologie.

Dans ce chapitre, nous avons défini des règles permettant d'obtenir des séquents de la forme $\Gamma \vdash \varphi$, qui se résumaient pour le moment en un jeu d'écriture. On se pose donc les questions suivantes :

- *Correction.* Si $\Gamma \vdash \varphi$, a-t-on $\Gamma \models \varphi$? Autrement dit, si on prouve φ à partir de Γ, est-ce que φ est conséquence logique de Γ ?
- *Complétude.* Inversement, si $\Gamma \models \varphi$, a-t-on $\Gamma \vdash \varphi$? C'est-à-dire, si φ est conséquence logique de Γ, peut-on prouver φ à partir de Γ en suivant les règles de la déduction ?

Le théorème suivant répond à la première question.

Théorème 15.8. *La déduction logique est correcte : tout séquent* $\Gamma \vdash \varphi$ *obtenu à partir des règles d'inférence vérifie* $\Gamma \models \varphi$.

Démonstration. Pour un séquent $S = \Gamma \vdash \varphi$, notons $P(S)$ la propriété $\Gamma \models \varphi$. L'ensemble des séquents que l'on peut constuire via les règles est un ensemble inductif. Il suffit donc de montrer pour chaque règle, que si les séquents prémisses vérifient P, alors le séquent conclusion vérifie P. Cette démonstration est simple, faisons le pour $I_\wedge$ par exemple, les autres cas sont laissés en exercice.

Soient $\Gamma \vdash \varphi$ et $\Gamma \vdash \psi$ deux séquents vérifiant la propriété P. Soit d une distribution de vérité sur l'ensemble des variables logiques intervenant dans Γ, φ et ψ. Par hypothèse, d satisfait $\left(\bigwedge_{\psi\in\Gamma} \psi\right) \Rightarrow \varphi$ et $\left(\bigwedge_{\psi\in\Gamma} \psi\right) \Rightarrow \psi$. Montrons qu'elle satisfait $\Phi = \left(\bigwedge_{\psi\in\Gamma} \psi\right) \Rightarrow \varphi \wedge \psi$:

— Si $\mathcal{E}_d\left(\bigwedge_{\psi\in\Gamma} \psi\right) = 0$, alors $\mathcal{E}_d(\Phi) = 1$;

— Sinon, $\mathcal{E}_d\left(\bigwedge_{\psi\in\Gamma} \psi\right) = 1$, mais alors $\mathcal{E}_d(\varphi) = \mathcal{E}_d(\psi) = \mathcal{E}_d(\varphi \wedge \psi) = \mathcal{E}_d(\Phi) = 1$.

Et donc la propriété est vérifiée pour le séquent $\Gamma \vdash \varphi \wedge \psi$. □

Définition 15.9. *On appelle théorème un séquent de la forme* $\vdash \varphi$ *(le contexte est vide), obtenu par déduction naturelle.*

Le théorème précédent montre qu'un théorème (au sens de la déduction naturelle) est bien une tautologie.

15.5 Complétude de la déduction naturelle

On passe maintenant à la deuxième question évoquée précédemment. Toute tautologie est-elle prouvable ? La réponse est (pour le moment) non.

Exercice 4. *Loi de Peirce.* Montrer que $((\varphi \Rightarrow \psi) \Rightarrow \varphi) \Rightarrow \varphi$ est une tautologie.

On peut montrer (c'est pénible) que la loi de Peirce, bien que tautologique, n'est pas démontrable avec les règles précédentes. Elle le devient si on rajoute la règle du *tiers exclu*.

Tiers exclu. On rajoute aux règles précédentes la règle suivante (*exm* pour excluded middle, tiers exclu en anglais), sans prémisse.

$$\frac{}{\Gamma \vdash \varphi \vee \neg\varphi}\ (\text{exm})$$

Il est clair que les séquents que l'on peut obtenir en rajoutant cette règle sont toujours valides, dans le sens où les formules logiques associées sont tautologiques : il suffit de traiter le cas du tiers exclu dans la preuve par induction, ce qui est facile. Voici quelques exemples de séquents prouvables, alors qu'ils ne le sont pas sans tiers exclu (on note $\varphi \Leftrightarrow \psi$ comme une abbréviation de $(\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$

- Simplification de la négation de la négation : $\neg\neg\varphi \vdash \varphi$
- Loi de De Morgan : $\vdash \neg(\varphi \wedge \psi) \Leftrightarrow (\neg\varphi \vee \neg\psi)$
- Contraposée : $\vdash (\varphi \Rightarrow \psi) \Leftrightarrow (\neg\psi \Rightarrow \neg\varphi)$

Avec le tiers exclu, on peut montrer le théorème suivant (hors programme), dont on ne fera pas la preuve.

Théorème 15.10. *Muni du tiers exclu, la déduction naturelle est complète : tout séquent valide est prouvable.*

Remarque 15.11. *Ce traitement « à part » du tiers exclu n'est pas anodin : initié comme position plutôt philosophique par le mathématicien hollandais Brouwer, il a donné naissance à la* logique intuitionniste*, qui s'est révélé très utile en informatique théorique. Dans la logique intuitionniste, un raisonnement comme le suivant est rejeté :*

> *Montrons qu'il existe un irrationnel x tel que $x^{\sqrt{2}}$ est rationnel. Posons $y = \sqrt{2}^{\sqrt{2}}$. Si y est rationnel, alors $x = \sqrt{2}$ convient. Sinon, posons $x = y$. Alors $x^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\cdot\sqrt{2}} = 2$, donc x convient.*

En effet, lorsqu'un mathématicien intuitionniste énonce A, il affirme qu'il possède une preuve pour A. Ici, on ne possède de preuve ni pour $\sqrt{2}^{\sqrt{2}}$ rationnel, ni pour $\sqrt{2}^{\sqrt{2}}$ irrationnel. En notant $A = \sqrt{2}^{\sqrt{2}}$ rationnel et B l'énoncé à prouver, on a montré les deux implications $A \Rightarrow B$ et $\neg A \Rightarrow B$, mais on ne peut en déduire B.

Résumé des règles. Le tableau suivant résume les règles introduites dans le chapitre.

Symbole	Introduction / Autre	Élimination
Hypothèse (H)	$\dfrac{}{\Gamma, \varphi \vdash \varphi}\ (\text{H})$	
Affaiblissement (Aff)	$\dfrac{\Gamma \vdash \varphi}{\Gamma, \psi \vdash \varphi}\ (\text{Aff})$	
$\wedge$	$\dfrac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\ (I_\wedge)$	$\dfrac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi}\ (E^1_\wedge) \qquad \dfrac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}\ (E^2_\wedge)$
$\Rightarrow$	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \Rightarrow \psi}\ (I_\Rightarrow)$	$\dfrac{\Gamma \vdash \varphi \qquad \Gamma \vdash \varphi \Rightarrow \psi}{\Gamma \vdash \psi}\ (E_\Rightarrow)$
$\vee$	$\dfrac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}\ (I^1_\vee) \qquad \dfrac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}\ (I^2_\vee)$	$\dfrac{\Gamma \vdash \varphi \vee \psi \qquad \Gamma \vdash \varphi \Rightarrow \chi \qquad \Gamma \vdash \psi \Rightarrow \chi}{\Gamma \vdash \chi}\ (E_\vee)$
$\bot$	$\dfrac{\Gamma \vdash \varphi \wedge \neg\varphi}{\Gamma \vdash \bot}\ (I_\bot)$	$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash \psi}\ (E_\bot)$
$\neg$	$\dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ (I_\neg)$	
Tiers exclu (exm)	$\dfrac{}{\Gamma \vdash \varphi \vee \neg\varphi}\ (\text{exm})$	

Chapitre 16

Langages et expressions rationnelles

16.1 Introduction

Ce chapitre, et le suivant, traite de mots, et de langages : un langage est simplement un ensemble de mots. La question qui va le plus nous intéresser est la suivante : un mot donné appartient-il à un langage donné ? Malgré sa simplicité apparente, cette question est, en toutes généralités, difficile, il est même parfois impossible d'y répondre algorithmiquement ! Néanmoins elle est dans certains cas facile à traiter d'un point de vue algorithmique. Donnons quelques exemples où cette question apparaît.

Reconnaissance de motifs. Dans le tronc commun d'informatique de première année, on s'est posé la question de savoir si un mot s contenait un autre mot m. Cette question peut être décidée par un algorithme naïf en temps $O(|m|\,|s|)$, produit des tailles des deux mots. On peut reformuler la question ainsi : le mot s appartient-il au langage des mots qui contiennent m ? On verra notamment une approche plus efficace que l'algorithme naïf dans le prochain chapitre.

Analyse syntaxique. Lorsqu'on essaie d'exécuter un programme dans un langage de programmation comme Python ou Caml, l'interpréteur réalise avant d'exécuter toute instruction une analyse syntaxique : un exemple courant est l'oubli d'une parenthèse qui se traduit par la sentence « Invalid syntax ». Vérifier que les parenthèses sont correctement imbriquées est une traduction de : « le programme forme-t-il un mot bien parenthésé ? ». L'analyse syntaxique a aussi pour but de vérifier par exemple qu'un mot réservé du langage (comme « else ») n'est pas utilisé comme une variable. L'analyse syntaxique peut donc être vue comme la question d'appartenance du programme à l'ensemble des programmes syntaxiquement corrects.

Analyse lexicale. Pour mener à bien l'analyse syntaxique décrite précédemment, l'interpréteur réalise au préalable une analyse lexicale : il s'agit de découper le programme en morceaux : 123.54 doit être reconnu comme un flottant, 78 comme un entier, = comme l'opérateur d'affectation ou test d'égalité suivant le langage de programmation, etc... Là aussi, les questions d'appartenance d'un mot à un langage (langage des entiers, langage des flottants, etc...) est omniprésente.

Formulaires WEB et recherche dans des fichiers. Sur un site web où un utilisateur souhaite s'enregistrer, il faut vérifier que ce qu'il entre au clavier dans les champs à renseigner a la forme voulue. On s'intéresse donc à l'appartenance au langage des adresses mail, des dates, etc... Par exemple, `^[0-3][0-9]/(0[1-9]|1[0-2])/[1-2][0-9]{3}$` est une expression régulière désignant les dates entre 1000 et 2999 (et un peu plus...). Ce genre d'expressions régulières est également utilisée pour filtrer les lignes d'un fichier texte qui vérifient une certaine condition, la commande `grep` (*Get Regular ExPression*), permet à l'aide de l'expression précédente de filtrer les lignes d'un fichier qui sont des dates.

10ème problème de Hilbert. Ce problème énoncé en 1900 soulevait la question de l'existence d'un algorithme capable de prendre en entrée une équation *diophantienne* (une équation algébrique à coefficients entiers) et de décider si elle possédait ou non une solution rationnelle. Il a fallu attendre 1970 pour qu'un mathématicien russe, Youri Matiiassevitch, montre l'inexistence d'un tel algorithme : il n'existe donc pas d'algorithme capable de prendre en entrée une équation diophantienne, et décidant si cette équation appartient au langage des équations ayant des solutions rationnelles.

Biologie. En génétique, on s'intéresse aux mots sur l'alphabet $\{A, C, T, G\}$. Des questions intéressantes (un peu éloignées de la question initiale) émergent naturellement de ce contexte, comme l'existence de facteurs communs ou de proximité entre deux individus.

16.2 Mots sur un alphabet

16.2.1 Définition et structure mathématique

Définition 16.1. *Un* alphabet *est un ensemble fini non vide de symboles (les lettres). Dans tout le chapitre, on notera* Σ *un alphabet.*

Exemple 16.2.
- $\Sigma = \{A, C, T, G\}$ *langage utilisé pour décrire des séquences d'ADN ;*
- $\Sigma = \{0, 1, 2, \dots, 9\}$ *chiffres romains ;*
- $\Sigma = \{a, b, \dots, z\}$ *lettres minuscules ;*
- *caractères ASCII ;*
- *etc...*

Définition 16.3. *On appelle* mot *sur un alphabet* Σ *une suite finie de symboles de* Σ*. On notera* $m = m_1 \cdots m_k$ *une telle suite et* $|m|$ *la longueur d'un mot. Le mot vide (de longueur zéro) sera noté* ε*. On note aussi* $|m|_a$ *pour le nombre d'occurences de la lettre* $a \in \Sigma$ *dans le mot* m*.*

Définition 16.4. *On note* Σ^* *l'ensemble des mots sur* Σ*, et* $\Sigma^+ = \Sigma^* \backslash \{\varepsilon\}$ *l'ensemble des mots non vides.*

Définition 16.5. *Pour* $u = u_1 u_2 \cdots u_n$ *et* $v = v_1 v_2 \cdots v_k$ *deux mots sur un alphabet* Σ*, on appelle concaténation de* u *et* v *le mot* $u_1 u_2 \cdots u_n v_1 v_2 \cdots v_k$*. Ce mot sera noté* $u \cdot v$ *voire simplement* uv*.*

Proposition 16.6. $(\Sigma^*, \cdot)$ *est un monoïde (ensemble muni d'une l.c.i associative et d'un neutre), de neutre* ε*.*

Démonstration. immédiat. □

Proposition 16.7. *Pour deux mots* u *et* v *sur* Σ*, on a* $|uv| = |u| + |v|$ *et* $|uv|_a = |u|_a + |v|_a$ *pour toute lettre* a *de* Σ*. (ainsi* $|.|$ *et* $|.|_a$ *sont des morphismes de monoïde.*

Remarque 16.8. *À part* ε*, il n'y a pas d'élément inversible dans ce monoïde. Toutefois, il est simplifiable à gauche et à droite :* $uv = uw \Rightarrow v = w$ *et* $vu = wu \Rightarrow v = w$*.*

Définition 16.9. *Pour* m *un mot de* Σ^**, on appelle :*
- *préfixe de* m *un mot* u *tel qu'il existe un mot* v *vérifiant* $m = uv$ *;*
- *suffixe de* m *un mot* u *tel qu'il existe un mot* v *vérifiant* $m = vu$ *;*
- *facteur de* m *un mot* u *tel qu'il existe deux mots* v *et* w *vérifiant* $m = vuw$ *;*
- *sous-mot de* $m = m_1 \cdots m_k$ *un mot* $u = u_1 \cdots u_p$ *tel qu'il existe une application* $\varphi : [\![1, p]\!] \to [\![1, k]\!]$ *strictement croissante vérifiant* $u_1 \cdots u_p = m_{\varphi(1)} \cdots m_{\varphi(p)}$*.*

16.2.2 Lemme de Lévi et application à la commutativité

Lemme 16.10. *Soit* m *un mot, et* x *et* y *deux préfixes de* m*. Alors* x *est préfixe de* y *ou* y *est préfixe de* x*.*

Démonstration. Si x est de longueur inférieure à celle de y, x est préfixe de y car ses lettres sont les premières de m qui sont aussi les premières de y. Sinon c'est l'inverse. □

Lemme 16.11 (Lévi)**.** *Soient* u, v, w *et* z *quatre mots de* Σ^* *tels que* $uv = wz$*. Alors il existe un unique mot* $t \in \Sigma^*$ *tel que l'une des conditions suivantes est vérifiée :*
- $u = wt$ *et* $z = tv$ *;*
- $w = ut$ *et* $v = tz$*.*

Démonstration. u et w sont préfixes de $uv = wz$, donc l'un est préfixe de l'autre, disons w préfixe de u. Il existe alors un unique mot $t \in \Sigma^*$ tel que $u = wt$. On a ensuite $uv = wtv = wz$, donc $z = tv$. Évidemment, les deux conditions s'excluent, sauf si $u = w$ et $v = z$, auquel cas $t = \varepsilon$. □

Théorème 16.12. *Deux mots* x *et* y *de* Σ^* *commutent (pour la concaténation) si et seulement si ce sont deux puissances d'un même mot, autrement dit il existe* $z \in \Sigma^*$ *et deux entiers* $i, j \geq 0$ *tels que* $x = z^i$ *et* $y = z^j$*.*

Démonstration. La condition est suffisante : deux puissances d'un même mot commutent. Montrons qu'elle est nécessaire, par récurrence sur $|x| + |y|$:

— si l'un des deux mots est vide, disons $x = \varepsilon$, alors $x = y^0$ et $y = y^1$: la propriété est vérifiée.

— sinon, appliquons le lemme de Lévi à $xy = yx$. Si $|x| \leq |y|$, il existe t tel que $y = xt = tx$. Comme x est non vide, $|x| + |t| = |y| < |x| + |y|$. Donc par hypothèse de récurrence, x et t s'écrivent tous deux z^i et z^j, et donc $y = z^{i+j}$. Le raisonnement est symétrique si $|x| > |y|$.

— Par principe de récurrence, l'équivalence est démontrée.

□

16.2.3 Mots de Dyck

Définition 16.13. *Sur $\Sigma = \{a, b\}$, on considère la valuation définie par $\nu(a) = 1$ et $\nu(b) = -1$. Elle s'étend aux mots de Σ^* : pour $m = m_1 \cdots m_k$, on pose $\nu(m) = \sum_{i=1}^{k} \nu(m_i)$. Un mot m sur Σ^* est dit* de Dick *s'il satisfait les deux conditions suivantes :*

— $\nu(m) = 0$

— *tout préfixe p de m vérifie $\nu(p) \geq 0$.*

Exemple 16.14. *aabbab est un mot de Dyck, alors que aab ou abba n'en sont pas.*

Remarque 16.15. *Ces mots sont également appelés mots bien parenthésés : si, dans une expression arithmétique on ne garde que les parenthèses que l'on remplace par a (pour une ouvrante) et b (pour une fermante) alors on obtient un mot de Dyck. Par exemple $((3-4) \times 5 + 2 \times (5 - (7+3)))/2 \to aabaabbb$.*

Dans la suite, on note D l'ensemble des mots de Dyck. Voyons comment construire des mots de Dyck par concaténation.

Proposition 16.16. *Le mot vide ε est dans D, et pour deux mots u et v quelconques de D, le mot $aubv$ est également dans D.*

Démonstration. C'est clair pour ε. Soient u et v deux mots de D.

— D'une part, $\nu(aubv) = 1 + \nu(u) - 1 + \nu(v) = 0$ car $\nu(u) = \nu(v) = 0$.

— D'autre part, si p est un préfixe de $aubv$:

  — soit $p = \varepsilon$ et $\nu(p) = 0$;

  — soit p est un préfixe de au de la forme ap' avec p' préfixe de u, et donc $\nu(p) = 1 + \nu(p') > 0$;

  — soit p est un préfixe de la forme $aubp'$ avec p' préfixe de v, et donc $\nu(p) = \nu(p') \geq 0$.

  Ce qui prouve que $aubv$ est bien dans D.

□

En fait, cette manière de procéder est essentiellement unique, comme le montre la proposition qui suit.

Proposition 16.17. *Soit m un mot de Dyck non vide. Alors m se décompose de manière unique en $aubv$, avec $u, v \in D$.*

Démonstration.

- Existence. L'ensemble des préfixes non vides de m de valuation nulle est non vide, car il contient m. Considérons p le plus petit élément de l'ensemble, et notons v le suffixe de m tel que $m = pv$. $\nu(v) = \nu(m) - \nu(p) = 0$, et pour v' un préfixe de v on a $\nu(v') = \nu(pv') \geq 0$ car pv' est un préfixe de $m \in D$. Donc $v \in D$. De plus, p étant non vide, sa première lettre est un a (sinon on aurait un préfixe de valuation strictement négative). Sa dernière lettre est un b car sinon p s'écrirait $ap'a$ avec $\nu(ap') = \nu(p) - 1 < 0$, ce qui n'est pas. Donc p est de la forme aub. D'une part $\nu(u) = \nu(p) = 0$. De plus, un préfixe strict de p non vide étant de valuation strictement positive par définition de p, les préfixes de u sont de valuation positive. Donc $u \in D$ et l'existence de la décomposition est démontrée.
- Unicité. Donnons nous deux décompositions $m = aubv = au'bv'$. On peut supposer que u est préfixe de u'. Si u était préfixe strict de u', u' aurait pour préfixe ub, ce qui est exclus car $\nu(ub) = -1$. Donc $u = u'$, et $v = v'$ par régularité.

□

Cette décomposition permet de dénombrer les mots de Dyck. Ceux-ci sont clairement tous de longueur paire, comptons-les.

Lemme 16.18. *Notons C_n le nombre de mots de Dyck de taille $2n$, pour $n \geq 0$. Alors $(C_n)_{n\in\mathbb{N}}$ vérifie :*

$$C_0 = 1 \qquad et \qquad C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k} \text{ pour } n \geq 1$$

Démonstration. Immédiat. □

Théorème 16.19. *Le nombre de mot de Dyck de taille $2n$ est $C_n = \frac{1}{n+1}\binom{2n}{n}$ pour tout $n \geq 0$.*

Démonstration. Il est facile de vérifier le résultat par récurrence. Donnons une preuve combinatoire. On note $M_{i,j}$ les mots sur $\{a, b\}$ ayant i fois la lettre a et j fois la lettre b, et D_n l'ensemble des mots de Dyck de $M_{n,n}$. Dénombrons les mots de longueur $2n \geq 2$ ayant autant de a que de b mais n'étant *pas* de Dyck, c'est-à-dire $M_{n,n}\backslash D_n$: pour un tel mot m, il existe un préfixe de valuation -1, notons le plus petit p et considérons l'application :

$$\begin{array}{rccc} \varphi : & M_{n,n}\backslash D_n & \longrightarrow & M_{n+1,n-1} \\ & m = pv & \longmapsto & \bar{p}v \end{array}$$

où $\bar{p}$ est le mot obtenu à partir de p en échangeant les a et les b. Comme p est de valuation -1, il s'ensuit que v est de valuation 1, de même que $\bar{p}$. Ainsi $\varphi(m) = \bar{p}v$ est de valuation 2 et est bien dans $M_{n+1,n-1}$. Inversement, pour un mot m de $M_{n+1,n-1}$, l'ensemble de ses préfixes de valuation 1 est non vide, car $\nu(\varepsilon) = 0$ et $\nu(m) = 2$, et la valuation évolue par pas de ± 1 lorsqu'on parcourt le mot. Notons q le plus petit préfixe de m de valuation 1, et considérons l'application :

$$\begin{array}{rccc} \psi : & M_{n+1,n-1} & \longrightarrow & M_{n,n}\backslash D_n \\ & m = qv & \longmapsto & \bar{q}v \end{array}$$

Pour les mêmes raisons que précédemment, ψ est bien à valeur dans $M_{n,n}\backslash D_n$. De plus, ψ et φ sont réciproques l'une de l'autre. Il s'ensuit que $M_{n+1,n-1}$ et $M_{n,n}\backslash D_n$ ont même cardinal. Or $M_{i,j}$ a pour cardinal $\binom{i+j}{i}$. Ainsi,

$$C_n = |D_n| = |M_{n,n}| - |M_{n+1,n-1}| = \binom{2n}{n} - \binom{2n}{n+1} = \frac{(2n)!}{n!^2} - \frac{(2n)!}{(n+1)!(n-1)!} = \binom{2n}{n}\left(1 - \frac{n}{n+1}\right) = \frac{1}{n+1}\binom{2n}{n}$$

□

Remarque 16.20. *Les nombres (C_n) sont les nombres de* Catalan [1]*. Ils interviennent très souvent en combinatoire, et donc en informatique. Par exemple, ils dénombrent :*

- *les mots bien parenthésés ;*
- *les chemins de $(0, 0)$ à $(2n, 0)$ dans le demi-plan $y \geq 0$, qui utilisent les deux déplacements de vecteurs $(1, 1)$ et $(1, -1)$;*
- *les arbres binaires entiers à n nœuds internes ;*
- *les triangulations d'un polygone convexe à $n + 2$ côtés ;*
- *...*

16.3 Langages

16.3.1 Définition et cardinalité

Fixons un alphabet Σ. L'ensemble des mots (suites finies de lettres) sur l'alphabet Σ est noté Σ^*.

Définition 16.21. *Un langage sur un alphabet Σ est une partie de Σ^**

Exemple 16.22. *Les ensembles suivants sont des langages :*

- *langage des mots de Dyck sur $\{a, b\}$;*
- *langage des mots de la forme $a^n b^n$ sur $\{a, b\}$;*
- *langage des mots contenant plus de a que de b sur $\{a, b\}$;*
- *langage des mots contenant bracada mais pas abracadabra comme facteur (sur l'alpahabet usuel, par exemple) ;*
- *langage des mots contenant bracada mais pas abracadabra comme sous-mot ;*
- *...*

1. Eugène Charles Catalan, 1814-1894, mathématicien belge.

Proposition 16.23. *L'ensemble de tous les langages est indénombrable.*

Démonstration. Σ^* est en bijection avec $\mathbb{N}$, donc l'ensemble de tous les langages avec $\mathcal{P}(\mathbb{N})$. Classiquement, un ensemble n'est jamais en bijection avec l'ensemble de ses parties [2], donc il y a une infinité non dénombrable de langages sur un alphabet Σ. □

Remarque 16.24. *L'ensemble des algorithmes étant dénombrable (ils forment un langage !), pour la plupart des langages il n'existe pas d'algorithme capable de prendre en entrée un mot quelconque et de décider s'il appartient ou non au langage. Un langage pour lequel il existe un tel algorithme est dit* récursif. *Dans la suite, on va s'intéresser à une toute petite partie des langages récursifs : les langages rationnels.*

16.3.2 Opérations sur les langages

Fixons un alphabet Σ. Sur l'ensemble des langages d'alphabet Σ, on peut considérer les opérations suivantes :

— opérations ensemblistes (union, intersection, différence, différence symétrique, etc...) ;

— concaténation : $L_1 \cdot L_2 = \{m_1 \cdot m_2 \mid m_1 \in L_1 \quad \text{et} \quad m_2 \in L_2\}$;

— puissance : on note $L^0 = \{\varepsilon\}$ et pour $n \geq 1$, $L_n = L \cdot L_{n-1}$ (remarque : ne pas confondre L^2 avec $\{u \cdot u \mid u \in L\} \subset L^2$).

— étoile de Kleene : on note $L^* = \cup_{n\geq 0} L^n$, appelée l'étoile de Kleene du langage L, ensembe de mots obtenus par concaténation de mots de L. On note aussi $L^+ = \cup_{n=1}^{+\infty} L^n$, qui ne contient ε que si L le contient. Ces notations sont cohérentes avec Σ^* et Σ^+.

16.3.3 Expressions rationnelles et langages rationnels

Définition 16.25. *L'ensemble des expressions rationnelles sur un alphabet Σ est défini inductivement :*

— *$\emptyset$ et ε sont des expressions rationnelles ;*

— *pour tout $a \in \Sigma$, a est une expression rationnelle ;*

— *pour e_1, e_2 deux expressions rationnelles, les expressions $(e_1 + e_2)$, $(e_1 e_2)$ et e_1^* sont rationnelles.*

Exemple 16.26. *$((((ab) + c)a)^* + (cb^*)) + \varepsilon$ est rationnelle sur $\Sigma = \{a, b, c\}$.*

Remarque 16.27. *Une manière plus propre de les définir est d'utiliser des arbres binaires : les feuilles sont $\emptyset$, ε et les a pour $a \in \Sigma$, les opérateurs $+$, $\cdot$ et $*$ sont associés à des nœuds internes, d'arité 2 pour $+$ et $\cdot$, et d'arité 1 pour $*$. Par exemple l'expression précédente se représente comme :*

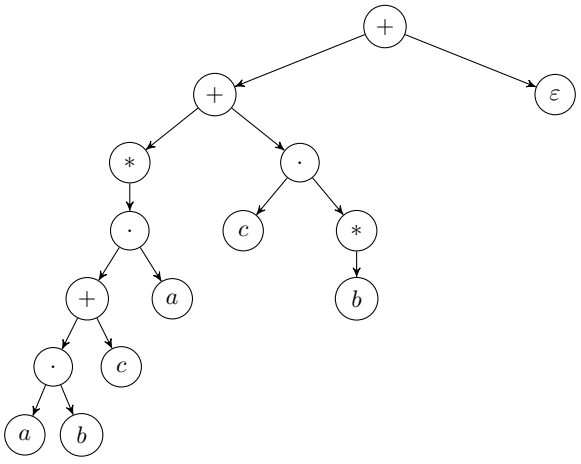


FIGURE 16.1 – L'arbre associé à l'expression rationnelle $((((ab) + c)a)^* + (cb^*)) + \varepsilon$.

Définition 16.28. *À une expression rationnelle e, on associe $L(e)$, un langage de Σ^*, défini également inductivement :*

— *$L(\emptyset) = \emptyset$, $L(\varepsilon) = \{\varepsilon\}$, et $L(a) = \{a\}$ pour tout $a \in \Sigma$;*

— *pour e_1, e_2 deux expressions rationnelles, $L(e_1 + e_2) = L(e_1) \cup L(e_2)$, $L(e_1 e_2) = L(e_1) \cdot L(e_2)$ et $L(e_1^*) = L(e_1)^*$.*

2. Si on suppose l'existence d'une sujection $\varphi : E \to \mathcal{P}(E)$, on aboutit à une contradiction en considérant un antécédent de $\{\omega \in E \mid \omega \notin \varphi(\omega)\}$. C'est le théorème de Cantor. Plus simplement, $\mathcal{P}(\mathbb{N})$ est en bijection avec $\mathbb{R}$, qui n'est pas dénombrable.

Définition 16.29. *Un langage est dit rationnel s'il est associé à une expression rationnelle.*

Remarque 16.30. *Lorsqu'on s'intéresse aux langages dénotés par des expressions rationnelles, on évite les parenthèses qui alourdissent l'écriture d'une expression rationnelle. Par exemple,* $((((ab)+c)a)^* + (cb^*)) + \varepsilon$ *sera notée plus simplement* $((ab+c)a)^* + cb^* + \varepsilon$.

Proposition 16.31. *L'ensemble* $Rat(\Sigma)$ *des langages rationnels sur* Σ *est la plus petite partie de* Σ^* *contenant* $\emptyset$, $\{\varepsilon\}$, *les langages* $\{a\}$ *pour* $a \in \Sigma$ *et stable par union, concaténation, et étoile de Kleene.*

Démonstration. Immédiat. □

Exemple 16.32.
— *Les langages finis sont rationnels : en effet le langage contenant le seul mot* $m = m_1, \dots, m_k$ *est obtenu comme* $L(m_1) \cdots L(m_k)$, *et un langage fini* L *s'obtient comme l'union finie* $L = \cup_{m \in L}\{m\}$.
— Σ^* *est rationnel.* $\Sigma^+ = \cup_{a \in \Sigma} a\Sigma^*$ *est rationnel.*
— *L'ensemble des mots qui ont un mot* m *comme facteur est* $\Sigma^* m \Sigma^*$, *et est donc rationnel.*

Remarque 16.33. *Les deux expressions rationnelles* $(a+b)^*$ *et* $a(a+b)^* + b(a+b)^* + \varepsilon$ *dénotent toutes deux le langage* $\{a,b\}^*$. *Il n'y a donc pas unicité de l'expression rationnelle dénotant un langage rationnel. On pourra faire le parallèle avec la différence entre syntaxe et sémantique d'une expression logique.*

Réglons tout de suite la question de savoir s'il existe un langage non rationnel [3].

Proposition 16.34. *L'ensemble* $Rat(\Sigma)$ *des langages rationnels est dénombrable.*

Démonstration. L'ensemble des expressions rationnelles sur Σ est dénombrable : en effet, pour chaque entier $h \in \mathbb{N}$, il existe un nombre fini (non nul) d'arbres de hauteur h représentant une expression rationnelle : l'ensemble des expressions rationnelles est dénombrable [4], ainsi $\mathrm{Rat}(\Sigma)$ est au plus dénombrable. De plus, si $a \in \Sigma$, $\mathrm{Rat}(\Sigma)$ contient l'infinité de langages $(\{a^p\})_{p \in \mathbb{N}}$, et est donc dénombrable. □

Corollaire 16.35. *Il existe des langages non rationnels.*

Démonstration. $\mathrm{Rat}(\Sigma)$ est dénombrable (donc en bijection avec $\mathbb{N}$), $\mathcal{P}(\Sigma^*)$ ne l'est pas, car il est en bijection avec $\mathcal{P}(\mathbb{N})$. □

Exemple 16.36. *On verra plus tard que les langages constitués des mots de Dyck sur* $\{a,b\}$ *ou encore* $\{a^n b^n \mid n \geq 0\}$ *ne sont pas rationnels.*

16.3.4 Quelques réductions

Si un langage dénoté par une expression rationnelle n'est ni vide ni réduit à $\{\varepsilon\}$, on va montrer qu'il existe une expression rationnelle dénotant ce langage (à ε près) sans $\emptyset$ et sans ε. Ceci permet d'alléger l'implémentation.

Proposition 16.37. *Soit* e *une expression rationnelle dénotant un langage* $L(e) \neq \emptyset$. *Alors il existe* e' *sans* $\emptyset$ *telle que* $L(e') = L(e)$.

Démonstration. La démonstration se fait par induction :
— si $e = a \in \Sigma$ ou $e = \varepsilon$, il n'y a rien à montrer.
— si e s'écrit $e_1 + e_2$ avec e_1, e_2 deux expressions rationnelles, avec $L(e) \neq \emptyset$, alors l'un au moins des deux langages $L(e_1)$ et $L(e_2)$ est non vide :
  — si $L(e_1) = \emptyset$, alors $L(e_2) \neq \emptyset$, donc par hypothèse d'induction il existe e_2' dénotant $L(e_2)$ avec e_2' sans $\emptyset$. Comme $L(e) = L(e_2')$, c'est terminé ;
  — de même si $L(e_2) = \emptyset$;
  — sinon, il existe e_1' et e_2' sans $\emptyset$ telles que $L(e_i) = L(e_i')$ pour $i \in \{1,2\}$. Alors $e' = e_1 + e_2$ est sans $\emptyset$ et dénote $L(e)$.
— si e s'écrit $e_1 e_2$ avec e_1, e_2 deux expressions rationnelles, avec $L(e) \neq \emptyset$, alors $L(e_1)$ et $L(e_2)$ sont tous deux non vides : on procède de même que précédemment, en écrivant $L(e) = L(e_1' e_2')$.
— si $e = e_1^*$, alors :
  — soit $L(e_1) = \emptyset$, auquel cas $L(e) = \{\varepsilon\}$ et $e' = \varepsilon$ convient.

3. La démonstration suivie est classique : pour montrer qu'un ensemble est non vide, il suffit de montrer qu'il est gros !
4. Un autre argument est que les expressions rationnelles sur Σ forment un langage sur $\Sigma \cup \{\emptyset, \varepsilon, +, *, \cdot, (,)\}$.

— soit $L(e_1) \neq \emptyset$, donc par induction il existe e'_1 sans $\emptyset$ telle que $L(e_1) = L(e'_1)$ et $e' = e'^*_1$ est sans $\emptyset$ et dénote $L(e)$.

Par principe d'induction, la propriété est démontrée. □

Proposition 16.38. *Soit e une expression rationnelle dénotant un langage $L(e) \neq \emptyset$ et $L(e) \neq \{\varepsilon\}$. Alors il existe e' sans $\emptyset$ ni ε telle que $L(e) = L(e')$ ou $L(e) = L(e') \cup \{\varepsilon\}$.*

Démonstration. Là encore, la propriété se démontre par induction. D'après la proposition précédente, on peut supposer que e est de la forme $a \in \Sigma$, ou de la forme $e_1 + e_2$, e_1e_2, e_1^*, avec e_1 et e_2 sans $\emptyset$. Il n'y a rien à montrer pour $a \in \Sigma$, dans les trois autres cas on distingue les cas $L(e_i) = \{\varepsilon\}$ et $L(e_i) \neq \{\varepsilon\}$, auquel cas on se donne e'_i tel que $L(e_i) = L(e'_i)$ ou $L(e_i) = L(e'_i) \cup \{\varepsilon\}$, et on construit e' sans ε ni $\emptyset$ telle que $L(e)$ soit égal à $L(e')$ ou $L(e') \cup \{\varepsilon\}$.

$e_1 + e_2$			
$e_1 \backslash e_2$	ε	e'_2	$e'_2 + \varepsilon$
ε	ε(exclus)	$e'_2 + \varepsilon$	$e'_2 + \varepsilon$
e'_1	$e'_1 + \varepsilon$	$e'_1 + e'_2$	$e'_1 + e'_2 + \varepsilon$
$e'_1 + \varepsilon$	$e'_1 + \varepsilon$	$e'_1 + e'_2 + \varepsilon$	$e'_1 + e'_2 + \varepsilon$

e_1e_2			
$e_1 \backslash e_2$	ε	e'_2	$e'_2 + \varepsilon$
ε	ε(exclus)	e'_2	$e'_2 + \varepsilon$
e'_1	e'_1	$e'_1e'_2$	$e'_1e'_2 + e'_1$
$e'_1 + \varepsilon$	$e'_1 + \varepsilon$	$e'_1e'_2 + e'_2$	$e'_1e'_2 + e'_1 + e'_2 + \varepsilon$

e_1^*			
e_1	ε	e'_1	$e'_1 + \varepsilon$
e_1^*	ε(exclus)	e'^*_1	e'^*_1

□

16.4 Langages locaux et expressions rationnelles linéaires

16.4.1 Expressions rationnelles linéaires

Définition 16.39. *Une expression rationnelle e sur un alphabet Σ est dite linéaire si chaque lettre de Σ apparaît au plus une fois dans e.*

Exemple 16.40. *L'expression $\varepsilon + a(b^* + c)$ est linéaire. aa^* ne l'est pas.*

16.4.2 Langages locaux

Proposition 16.41. *Soit L un langage sur l'alphabet Σ. On définit :*

— *$P(L) = \{a \in \Sigma \mid a\Sigma^* \cap L \neq \emptyset\}$ l'ensemble des premières lettres des éléments de L ;*

— *$S(L) = \{a \in \Sigma \mid \Sigma^*a \cap L \neq \emptyset\}$ l'ensemble des dernières lettres des éléments de L ;*

— *$F(L) = \{f \in \Sigma^2 \mid \Sigma^*f\Sigma^* \cap L \neq \emptyset\}$ l'ensemble des facteurs de taille 2 des éléments de L ;*

— *$N(L) = \Sigma^2 \backslash F(L)$ l'ensemble des mots de taille 2 qui ne sont facteurs d'aucun mot de L.*

Alors $L\backslash\{\varepsilon\} \subseteq (P(L)\Sigma^* \cap \Sigma^*S(L))\backslash\Sigma^*N(L)\Sigma^*$.

Démonstration. Un élément de L non réduit à ε commence par une lettre de $P(L)$ et termine par une lettre de $S(L)$, il est donc dans $(P(L)\Sigma^* \cap \Sigma^*S(L))$. Aucun de ses facteurs de taille 2 n'est dans $N(L)$, donc il n'appartient pas à $\Sigma^*N(L)\Sigma^*$. D'où le résultat. □

Exemple 16.42. *Pour $L = L(e)$ avec $e = (ab)^*c + bca + \varepsilon$, on a : $P(L) = \{a, b, c\}$, $S(L) = \{a, c\}$, $F(L) = \{ab, ba, bc, ca\}$, et $N(L) = \Sigma^2\backslash F(L)$. Le mot $abca$ est dans $(P(L)\Sigma^* \cap \Sigma^*S(L))\backslash\Sigma^*N(L)\Sigma^*$ mais pas dans L.*

Définition 16.43. *Un langage sur Σ^* est dit* local *s'il y a égalité dans l'inclusion de la proposition précédente.*

Exemple 16.44. *Le langage $L(e)$ de l'exemple précédent n'est donc pas local. Par contre, $L((ab)^*c + \varepsilon)$ l'est.*

Proposition 16.45. *Soient $P \subset \Sigma$, $S \subset \Sigma$, $F \subset \Sigma^2$, et $N = \Sigma^2\backslash F$ Alors $L = (P\Sigma^* \cap \Sigma^*S)\backslash\Sigma^*N\Sigma^*$ est local.*

Démonstration. Clairement, $P(L) \subset P, S(L) \subset S$, et $F(L) \subset F$, donc comme $\varepsilon \notin L$,

$$L \subset (P(L)\Sigma^* \cap \Sigma^*S(L))\backslash\Sigma^*N(L)\Sigma^* \subset L$$

et L est local. □

Remarque 16.46.

— *Dans la propriété précédente, il n'y a pas a priori égalité entre les inclusions comme $P \subset P(L)$: par exemple sur $\Sigma = \{a, b, c\}$, avec $P = \{a, b\}$, $S = \{a, c\}$ et $F = \{ab, ba, ca\}$, on a $P(L) = \{a, b\}$, $S(L) = \{a\}$ et $F(L) = \{ab, ba\}$. Les éléments « inutiles » ont été supprimés.*

— *La définition d'un langage local montre que ceux-ci sont en nombre fini.*

16.4.3 Propriétés de clôture des langages locaux

L'ensemble des langages locaux est stable par certaines opérations. Voyons comment construire des langages locaux à partir d'autres langages locaux. Dans la suite, on fera des opérations entre deux langages L_1 et L_2, supposés locaux. On note pour $i \in \{1,2\}$ P_i, S_i, F_i et N_i les ensembles $P(L_i)$, $S(L_i)$, $F(L_i)$ et $N(L_i)$.

a) Intersection

Proposition 16.47. *L'intersection de deux langages locaux est local.*

Démonstration. On considère deux langages L_1 et L_2 locaux. Si $m \neq \varepsilon$ est dans $L_1 \cap L_2$, sa première lettre est dans $P_1 \cap P_2$, ses facteurs sont dans $F_1 \cap F_2$, et sa dernière lettre est dans $S_1 \cap S_2$. La réciproque est immédiate, donc avec $N = \Sigma^2 \backslash (F_1 \cap F_2)$.

$$L \backslash \{\varepsilon\} = (P_1 \cap P_2)\Sigma^* \cap \Sigma^*(S_1 \cap S_2) \backslash \Sigma^* N \Sigma^*$$

La propriété 16.45 montre que L est local. □

b) Union et Concaténation sur alphabets disjoints

Les langages locaux ne sont pas stables par union et concaténation, par contre si L_1 et L_2 sont locaux sur alphabets disjoints, c'est le cas.

Proposition 16.48. *Si L_1 et L_2 sont deux langages locaux sur Σ_1 et Σ_2 avec $\Sigma_1 \cap \Sigma_2 = \emptyset$, alors $L = L_1 \cup L_2$ est local sur $\Sigma = \Sigma_1 \cup \Sigma_2$.*

Démonstration. Posons $P = P_1 \cup P_2$, $S = S_1 \cup S_2$ et $F = F_1 \cup F_2$, et $N = \Sigma^2 \backslash F$. Ce sont clairement les ensembles caractéristiques de L, donc $L \backslash \{\varepsilon\} \subset P\Sigma^* \cap \Sigma^* S \backslash \Sigma^* N \Sigma^*$. Soit $m \neq \varepsilon$ un mot de $P\Sigma^* \cap \Sigma^* S \backslash \Sigma^* N \Sigma^*$. Si la première lettre de m est dans P_1, on montre de proche en proche que tous les facteurs de taille 2 sont dans F_1, et sa dernière lettre dans S_1. Donc $m \in L_1 \subset L_1 \cup L_2$. De même si la première lettre est dans P_2. Ainsi $L_1 \cup L_2$ est local. □

Remarque 16.49. *Si $\Sigma_1 \cap \Sigma_2 \neq \emptyset$, la propriété est fausse : par exemple $\{ab\}$ et $\{bc\}$ sont tous deux locaux mais pas leur union, qui devrait contenir abc.*

Proposition 16.50. *Si L_1 et L_2 sont deux langages locaux sur Σ_1 et Σ_2 avec $\Sigma_1 \cap \Sigma_2 = \emptyset$, alors $L_1 \cdot L_2$ est local sur $\Sigma = \Sigma_1 \cup \Sigma_2$.*

Démonstration. Le tableau suivant donne les ensembles $P(L)$, $S(L)$ et $F(L)$ avec $L = L_1 \cdot L_2$ en fonction des ensembles caractéristiques de L_1 et L_2, dans le cas où les deux langages sont non vides (sinon L est vide, et donc local) :

	$\varepsilon \notin L_2$	$\varepsilon \in L_2$
$\varepsilon \notin L_1$	$P(L) = P_1$ $S(L) = S_2$ $F(L) = F_1 \cup F_2 \cup S_1P_2$	$P(L) = P_1$ $S(L) = S_1 \cup S_2$ $F(L) = F_1 \cup F_2 \cup S_1P_2$
$\varepsilon \in L_1$	$P(L) = P_1 \cup P_2$ $S(L) = S_2$ $F(L) = F_1 \cup F_2 \cup S_1P_2$	$P(L) = P_1 \cup P_2$ $S(L) = S_1 \cup S_2$ $F(L) = F_1 \cup F_2 \cup S_1P_2$

Soit m un mot de $P(L)\Sigma^* \cup \Sigma^* S(L) \backslash \Sigma^* N(L) \Sigma^*$.

- si sa première lettre est dans P_1, considérons m_1 son plus grand préfixe dans Σ_1^*. Ses facteurs de taille 2 sont tous dans Σ_1^2, donc dans F_1.
  - si $m_1 = m$ (auquel cas $\varepsilon \in L_2$), alors la dernière lettre de m est dans S_1, donc $m \in L_1 \subseteq L_1L_2$.
  - sinon, notons m_2 tel que $m_1m_2 = m$. Le facteur de taille 2 constitué de la dernière lettre de m_1 et la première de m_2 est nécessairement dans S_1P_2, donc $m_1 \in L_1$. On montre de proche en proche que les facteurs suivants sont tous dans Σ_2^2, donc dans F_2. La dernière lettre de m est dans Σ_2, donc dans S_2. Ainsi $m_2 \in L_2$, et $m \in L$.
- sinon, on a $\varepsilon \in L_1$ et on montre facilement que $m \in L_2 \subseteq L_1L_2$.

Ainsi, L est local. □

Remarque 16.51. *Là encore, si $\Sigma_1 \cap \Sigma_2 \neq \emptyset$, la propriété est fausse : par exemple $\{ab\}$ et $\{bc\}$ sont tous deux locaux mais pas leur concaténation $\{abbc\}$, qui devrait contenir abc.*

c) Étoile de Kleene

Proposition 16.52. *Si L est local, alors L^* est local.*

Démonstration. Avec P, S, et F les ensembles caractéristiques de L, ceux de L^* sont : $P(L^*) = P$, $S(L^*) = S$, et $F(L^*) = F \cup S \cdot P$. Montrons que l'on a bien $P(L^*)\Sigma^* \cup \Sigma^* S(L^*) \backslash \Sigma^* N(L^*) \Sigma^* \subset L^* \backslash \{\varepsilon\}$: Considérons un tel mot m, et regardons ses facteurs qui sont dans $S \cdot P \backslash F$: ceux-ci induisent une factorisation de m en mots $m_1 m_2 \cdots m_k$, où la dernière lettre de m_i est dans S, la première lettre de m_i dans P, et tous les facteurs de taille 2 de m_i dans F : donc $m_i \in L$ et $m \in L^*$. □

d) Corollaire On déduit des points précédents le théorème qui suit :

Théorème 16.53. *Le langage associé à une expression linéaire est un langage local.*

Démonstration. ε et $\emptyset$ sont des langages locaux (les ensembles caractéristiques sont vides). Le langage $L_a = \{a\}$ sur Σ est local, avec $P(L_a) = S(L_a) = \{a\}$ et $F(L_a) = \emptyset$. Ensuite, une expression linéaire est construite comme $e_1 + e_2$ ou $e_1 e_2$ avec e_1 et e_2 deux expressions linéaires sur alphabets disjoints, ou encore comme e^* avec e linéaire. On conclut par induction. □

Remarque 16.54. *La réciproque est fausse : le langage dénoté par aa^* est local, car égal à $\Sigma^+ = \Sigma^* \backslash \{\varepsilon\}$. En revanche, il n'est pas local. De même pour $L(baa^*c)$, par exemple.*

Remarque 16.55. *Pour $L = L_1 L_2$, $L = L_1 \cup L_2$ ou $L = L_1^*$, les expressions donnant $P(L), S(L)$ et $F(L)$, en fonction des $P(L_i), S(L_i)$ et $F(L_i)$ vues dans les démonstrations ci-dessus restent valables, mais pas pour $L = L_1 \cap L_2$.*

16.5 Implémentation

On implémente les expressions rationnelles représentant un langage *non vide* sur un alphabet Σ à l'aide du type `erat` suivant :

```
type 'a erat =
    Eps
    | S of 'a
    | Plus of 'a erat * 'a erat
    | Conc of 'a erat * 'a erat
    | Etoile of 'a erat
;;
```

On a pris un type polymorphe, de sorte que l'alphabet peut-être représenté par des entiers ou des chaînes de caractères. Voici par exemple l'expression rationnelle $(a+b)^* + ac^*$:

```
let e=Plus (Etoile (Plus (S "a", S "b")), Conc (S "a", Etoile (S "c"))) ;;
```

Voici un exemple d'une fonction testant si le langage associé à une expression rationnelle contient ε.

```
let rec contient_eps e=match e with
  | Eps -> true
  | S _ -> false
  | Plus (a,b) -> contient_eps a || contient_eps b
  | Conc (a,b) -> contient_eps a && contient_eps b
  | Etoile _ -> true
;;
```

Décrivons une fonction qui calcule simultanément les ensembles $P(L)$, $S(L)$ et $F(L)$ d'un langage L donné par une expression rationnelle. On utilise :

— des listes d'éléments de type `'a` pour les ensembles $P(L)$ et $S(L)$;

— une liste de couples d'éléments de type `'a` pour $F(L)$.

Pour mener à bien le calcul, on doit :

— faire l'union (sans doublons) de deux listes ;

— calculer le produit cartésien de deux listes.

```
let rec union l1 l2=match l1 with
  | [] -> l2
  | x::q when List.mem x l2 -> union q l2
  | x::q -> x::(union q l2)
;;

let rec prod p s=match p with
  | [] -> []
  | x::q -> (List.map (function y -> (x,y)) s)@(prod q s)
;;
```

avec `List.map f t` qui produit la liste de `f(x)` pour `x` dans la liste `t`.

On peut maintenant écrire la fonction principale, prenant en entrée une expression rationnelle `e` et qui renvoie les ensembles `p, s, f` ainsi qu'un booléen indiquant si ε est dans le langage dénoté par `e`. Comme évoqué dans la remarque 16.55, cette fonction est valable même pour des expressions dénotant des langages non locaux.

```
let rec calcule e=match e with
  | Eps -> [], [], [], true
  | S x -> [x], [x], [], false
  | Plus (f, g) -> let p1,s1,f1, b1=calcule f and p2,s2,f2,b2=calcule g in
                (union p1 p2),(union s1 s2),(union f1 f2), b1 || b2
  | Conc (f, g) -> let p1,s1,f1,b1=calcule f and p2,s2,f2,b2=calcule g in begin match b1,b2 with
    | false, false -> p1, s2, union f1 (union f2 (prod s1 p2)), false
    | false, true -> p1, union s1 s2, union f1 (union f2 (prod s1 p2)), false
    | true, false -> union p1 p2, s2, union f1 (union f2 (prod s1 p2)), false
    | true, true -> union p1 p2, union s1 s2, union f1 (union f2 (prod s1 p2)), true
  end
  | Etoile f -> let p, s, f, _=calcule f in p, s, union f (prod s p), true
;;
```

Une estimation de la complexité est $O(|\Sigma|^4 |e|)$, avec $|e|$ la taille de l'expression `e` (pouvant être définie comme le nombre de nœuds de l'arbre associé) : les tailles des listes p_i et s_i sont au plus Σ, et celles des listes f_i au plus Σ^2. L'opération la plus coûteuse est l'union de listes encodant les ensembles de facteurs de taille 2 des langages, de complexité $O(\Sigma^4)$. Ensuite, on fait appel à ces fonctions sur les listes un nombre borné de fois par nœud de l'arbre associé, d'où la complexité.

On pourrait en fait réduire la complexité à $O(\left|\Sigma^2\right| |e|)$ en utilisant des tableaux à la place de listes, et si l'alphabet est gros, on peut encore réduire cette complexité en quelque chose de linéaire en la taille du résultat (multiplié par le nombre de nœuds) à l'aide de tables de hachage.

Chapitre 17

Automates

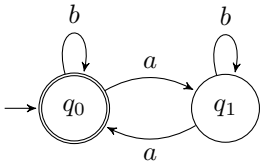
17.1 Introduction

Essentiellement, un automate est un programme, qui prend en entrée un mot (donné lettre par lettre) sur un alphabet Σ fixé, et qui l'accepte ou le rejette. Une contrainte forte est que la quantité de mémoire utilisable par l'automate est finie.

Le langage des mots reconnus par l'automate est appelé le langage reconnu par l'automate. Un langage sur Σ est dit reconnaissable s'il existe un automate qui le reconnaît.

Concrètement, la finitude de la mémoire utilisable par l'automate se traduit par un nombre fini d'*états* : on démarre à un état *initial*, et la lecture des lettres du mot fait changer d'état, en suivant des *transitions*. Le mot est accepté si après sa lecture, on se retrouve dans un état *final* (ou acceptant). Visuellement, on représente un automate comme un graphe : les sommets du graphe forment les états de l'automate, ses arcs (étiquetées par des lettres de Σ) les transitions.

Exemple 17.1. *Voici par exemple un automate à deux états reconnaissant le langage des mots sur $\{a, b\}$ ayant un nombre pair de a. On démarre à l'état q_0 (état initial) et on lit le mot lettre par lettre en suivant les transitions. Lire un a fait changer d'état, lire un b nous fait rester dans le même état. L'état initial est aussi le seul état final dans cet exemple.*



Le fait que la mémoire disponible soit finie impose des contraintes fortes sur l'ensemble des langages reconnaissables par un automate. Par exemple, on montrera que $\{a^n b^n \mid n \in \mathbb{N}\}$ n'est pas reconnaissable : il faudrait pouvoir compter jusqu'à n pour reconnaître $a^n b^n$, ce qui nécessite $1 + \lfloor \log_2(n) \rfloor$ bits. Or, $(\log_2(n))_{n \in \mathbb{N}^*}$ n'est pas bornée.

17.2 Automates finis déterministes

17.2.1 Définitions

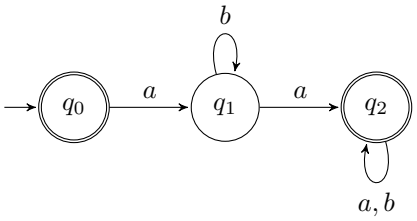
Remarque 17.2. *Les automates au programme sont finis. On verra un peu plus loin dans le cours des automates* non déterministes.

Définition 17.3. *Un automate déterministe (AFD en abrégé) est un quintuplet $(\Sigma, Q, q_0, F, \delta)$ où :*

- *Σ est un alphabet (fini) ;*
- *Q est un ensemble non vide, dont les éléments sont les états de l'automate ;*
- *$q_0 \in Q$ est l'état initial de l'automate ;*
- *$F \subset Q$ est l'ensemble des états finaux ;*
- *δ est la fonction de transition. C'est une application* partielle *de $Q \times \Sigma \to Q$, c'est-à-dire que $\delta(q, a)$ peut ne pas être définie pour tout (q, a) (concrètement, c'est donc une application d'un sous-ensemble de $Q \times \Sigma$ dans Q). On dira que $\delta(q, a)$ est un blocage de l'automate si $\delta(q, a)$ n'est pas défini.*

On représente l'automate comme un graphe (avec des boucles et des multi-arêtes) dont les sommets sont les états, l'état initial est indiqué par une flêche entrante et les états finaux par des doubles cercles [1] Si $\delta(q,a)=q'$, il y a un arc $q \to q'$ étiqueté par a.

Exemple 17.4. *Voici un autre exemple d'automate :*



L'alphabet est $\Sigma=\{a,b\}$, l'ensemble des états est $\{q_0,q_1,q_2\}$, l'état initial q_0, l'ensemble des états finaux est $\{q_0,q_2\}$, et la fonction de transition est décrite ci-dessous :

$Q\backslash\Sigma$	a	b
q_0	q_1	*blocage*
q_1	q_2	q_1
q_2	q_2	q_2

Le lecteur se convaincra facilement que le langage reconnu est celui des mots commençant par a et contenant au moins un autre a, auquel on ajoute le mot vide ε.

Définition 17.5. *La fonction de transition δ s'étend en une fonction partielle $\delta^* : Q\times\Sigma^* \to Q$ par récurrence : on pose $\delta^*(q,\varepsilon)=q$ pour tout $q\in Q$, et ensuite, pour tout $(q,a,m)\in Q\times\Sigma\times\Sigma^*$:*

- *$\delta^*(q,am)$ est un blocage si $\delta(q,a)$ en est un ou si $\delta^*(\delta(q,a),m)$ en est un ;*
- *dans le cas contraire, $\delta^*(q,am)=\delta^*(\delta(q,a),m)$.*

Par exemple, pour l'automate précédent, $\delta^*(q_0,baab)$ est un blocage, et $\delta^*(q_0,abb)=q_1$.

Définition 17.6 (Langage reconnu par un automate)**.** *Pour $A=(\Sigma,Q,q_0,F,\delta)$ un AFD, on appelle langage reconnu par l'automate le langage :*

$$L(A)=\{m\in\Sigma^* \mid \delta^*(q_0,m)\in Q\}$$

Définition 17.7. *On appelle langage reconnaissable un langage reconnu par un AFD.*

On verra que les langages reconnaissables sur un alphabet Σ coïncident avec les langages rationnels. Seule l'implication langage rationnel $\Rightarrow$ langage reconnaissable est au programme. Il est déja facile de voir que tout langage sur un alphabet Σ ne peut être reconnaissable : il n'y a, à renommage des états près, qu'un nombre fini d'automates à n états pour un entier n donné. L'ensemble des langages reconnaissables est donc dénombrable, contrairement à $\mathcal{P}(\Sigma^*)$.

17.2.2 Équivalence d'automates

Définition 17.8. *Deux automates sur un même alphabet Σ sont dits équivalents s'ils reconnaissent le même langage.*

On va maintenant définir quelques propriétés que peuvent avoir les automates, et montrer qu'on peut toujours supposer qu'un automate donné possède une telle propriété, du point de vue du langage reconnu. Ceci permet de simplifier certaines preuves et constructions.

Automate complet

Définition 17.9. *Un automate est dit* complet *lorsque la fonction de transition δ est sans blocage.*

Ceci implique que la fonction de transition étendue δ^* est également sans blocage, donc définie sur $Q\times\Sigma^*$.

Proposition 17.10. *Un automate est équivalent à un automate complet.*

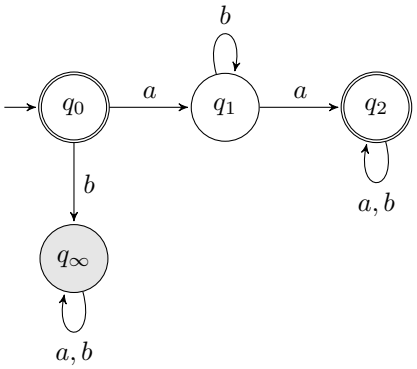
1. Certains auteurs préfèrent une flêche sortante.

Démonstration. Soit $A = (\Sigma, Q, q_0, F, \delta)$ un automate. Notons q_∞ un élément qui n'est pas dans Q. Alors l'automate $A' = (\Sigma, Q \cup \{q_\infty\}, q_0, F, \delta')$, où δ' est définie par :

$$\delta'(q,a) = \begin{cases} q_\infty & \text{si } q = q_\infty \text{ ou si } \delta(q,a) \text{ est un blocage de } A; \\ \delta(q,a) & \text{sinon} \end{cases}$$

est un automate complet équivalent à A. En effet, on a simplement rajouté un état « puits » q_∞ où l'on fait aboutir toutes les transitions non définies de A. Comme q_∞ n'est pas final dans A', le langage reconnu est le même. □

Exemple 17.11. *Voici un automate complet équivalent à l'automate précédent :*



Automate standard

Définition 17.12. *Un automate est dit standard si aucune transition n'aboutit à l'état initial.*

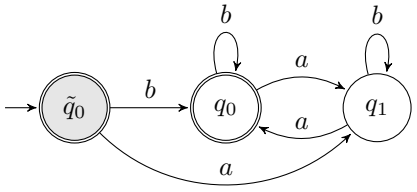
Proposition 17.13. *Un automate est équivalent à un automate standard.*

Démonstration. Pour construire un automate standard reconnaissant le même langage que $A = (\Sigma, Q, q_0, F, \delta)$, il suffit d'introduire un nouvel état initial $\tilde{q}_0 \notin Q$. Considérons l'automate $A' = (\Sigma, Q \cup \{\tilde{q}_0\}, \tilde{q}_0, \tilde{F}, \delta')$, où δ' est définie comme suit :

$$\delta'(q,a) = \begin{cases} \delta(q,a) & \text{si cette transition est définie et } q \neq \tilde{q}_0. \\ \delta(q_0,a) & \text{si cette transition est définie et } q = \tilde{q}_0. \\ \text{blocage} & \text{sinon.} \end{cases}$$

L'ensemble $\tilde{F}$ vaut F si q_0 n'est pas un état final, et $F \cup \{\tilde{q}_0\}$ sinon (avec seulement F on perdrait le mot vide du langage reconnu par l'automate). Alors A' reconnaît le même langage que A : en effet, pour tout mot m de taille au moins 1, on a $\delta^*(\tilde{q}_0, m) = \delta^*(q_0, m)$ dans A', et ε est reconnu par A si et seulement si il l'est par A'. □

Exemple 17.14. *Voici un automate standard reconnaissant le langage des mots sur $\{a, b\}$ ayant un nombre pair de a :*



Remarque 17.15. *« Standardiser » un automate complet ne change pas son caractère complet : tout automate est donc équivalent à un automate standard complet.*

Automate émondé

Définition 17.16. *Soit $A = (\Sigma, Q, q_0, F, \delta)$ un AFD. Un état q de A est dit :*

- accessible*, s'il existe un mot m de Σ^* tel que $\delta^*(q_0, m) = q$;*
- co-accessible*, s'il existe un mot m de Σ^* tel que $\delta^*(q, m) \in F$;*

Une conséquence immédiate de la définition est que l'état initial est accessible, et que tout état final est co-accessible. Les états non accessibles ou non co-accessibles sont un peu inutiles du point de vue du langage reconnu : on ne peut pas y accéder ou bien n'accéder à aucun état final depuis eux.

Définition 17.17. *Un automate est dit* émondé *si tout état de l'automate est à la fois accessible et co-accessible.*

« Presque » tous les automates sont équivalents à des automates émondés, comme on va le voir avec la proposition qui suit. Avant de l'énoncé, montrons un lemme.

Lemme 17.18. *Soit $A = (\Sigma, Q, q_0, F, \delta)$ un AFD, tel que $L(A) \neq \emptyset$ (le langage reconnu par A est non vide). Alors q_0 est co-accessible et l'automate possède un état final accessible.*

Démonstration. Soit $m \in L(A)$. Comme $\delta^*(q_0, m) \in F$, l'état q_0 est co-accessible. De même, l'état $\delta^*(q_0, m)$ est un état final accessible. □

Proposition 17.19. *Un automate reconnaissant un langage non vide est équivalent à un automate émondé.*

Démonstration. Soit $A = (\Sigma, Q, q_0, F, \delta)$ un automate reconnaissant un langage non vide. On considère Q' l'ensemble des états de Q à la fois accessibles et co-accessibles (non vide car il contient q_0), et δ' la fonction de transition de $Q' \times \Sigma \to Q'$ définie par :

$$\delta'(q, a) = \begin{cases} \delta(q, a) & \text{si } \delta(q, a) \in Q' \\ \text{blocage} & \text{sinon.} \end{cases}$$

On pose également $F' = Q' \cap F$. Alors l'automate $A' = (\Sigma, Q', q_0, F', \delta')$ reconnaît le même langage que A :

- $L(A') \subset L(A)$ est évident car on a supprimé des états et des transitions de A pour obtenir A'.
- $L(A) \subset L(A')$. Soit $m \in L(A)$. Alors $q = \delta^*(q_0, m) \in F \cap Q' = F'$. Tous les états du chemin de q_0 à q étiqueté par les lettres de m sont à la fois accessibles et co-accessibles, donc sont dans Q', et les transitions étiquetées par les lettres de m depuis q_0 ne sont pas des blocages, ainsi $\delta'^*(q_0, m) \in F'$ et $m \in L(A')$.

□

Remarque 17.20. *Émonder un automate standard ne change pas cette propriété, donc un automate reconnaissant un langage non vide est équivalent à un automate émondé standard.*

Par contre, il n'est en général pas possible de construire un automate émondé complet équivalent à un automate donné, comme le montre la proposition suivante.

Proposition 17.21. *Soit $A = (\Sigma, Q, q_0, F, \delta)$ un automate complet émondé. Alors $Pref(L(A))$ (le langage constitué des préfixes des mots de $L(A)$) est égal à Σ^* tout entier.*

Démonstration. Soit $m \in \Sigma^*$. Notons $q = \delta^*(q_0, m)$. L'état q est co-accessible, donc il existe un mot m' tel que $\delta^*(q, m') \in F$. Ainsi mm' est un mot de $L(A)$ dont m est un préfixe. □

17.2.3 Automates locaux

Définition 17.22. *Un automate est dit* local *si*

$$\forall a \in \Sigma,\ \exists q_a \in Q,\ \forall q \in Q, \qquad \delta(q, a) \text{ est un blocage ou } \delta(q, a) = q_a$$

Autrement dit, toutes les transitions étiquetées par une lettre donnée aboutissent à un même état.

Quitte à supprimer les états non accessibles d'un automate local, on voit que ceux-ci possèdent au plus $1 + |\Sigma|$ états : un état par lettre, plus éventuellement l'état initial. Comme leur nom l'indique, les automates locaux sont liés aux langages locaux.

Théorème 17.23. *Un langage local est reconnu par un automate local standard.*

Démonstration. Soit L un langage local, donné par le triplet $(P, S, F) \subseteq \Sigma \times \Sigma \times \Sigma^2$ de ses premières et dernières lettres et de ses facteurs de longueur 2. On considère l'automate local standard $A = (\Sigma, Q, q_0, F, \delta)$ défini par :

— $Q = \Sigma \cup \{q_0\}$ (il y a donc $|\Sigma| + 1$ états).

— $F = S$ si $\varepsilon \notin L$, $F = S \cup \{q_0\}$ sinon.

— $\delta(q_0, x) = x$ pour tout $x \in P$, $\delta(q_0, x)$ est un blocage pour tout $x \notin P$.

— Pour tout $(x, y) \in \Sigma \times \Sigma$, $\delta(x, y) = y$ si $xy \in F$, et $\delta(x, y)$ est un blocage sinon.

Alors $L(A) = L$. En, effet, $\varepsilon \in L(A)$ si et seulement si $\varepsilon \in L$. De plus, soit $m \neq \varepsilon$ un mot non vide du langage L. Les transitions indexées par les lettres de m depuis q_0 mènent à un état final, donc m appartient au langage $L(A)$. Réciproquement, si m est un mot non vide de $L(A)$, alors m est bien dans le langage local associé au triplet (P, S, F), car sa première lettre est dans P, sa dernière dans S et ses facteurs de taille 2 dans F. □

Remarque 17.24. *La réciproque est vraie : le langage reconnu par un automate local (standard) est local.*

Corollaire 17.25. *Les langages dénotés par des expressions linéaires sont reconnaissables.*

La preuve précédente fournit directement un moyen de construire un automate local reconnaissant le langage dénoté par une expression rationnelle linéaire.

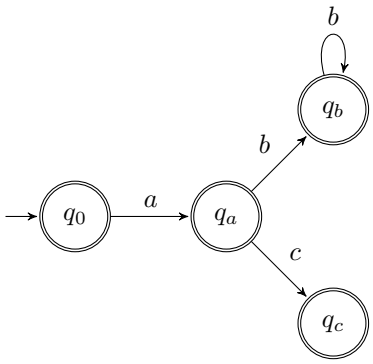
Exemple 17.26. *Soit $e = \varepsilon + a(b^* + c)$, construisons un automate reconnaissant $L = L(e)$:*

— $P(L) = \{a\}$;

— $S(L) = \{a, b, c\}$

— $F(L) = \{ab, ac, bb\}$

Comme $\varepsilon \in L$, l'automate suivant reconnaît L :



17.3 Automates non déterministes

Dans un AFD, il y a au plus un chemin étiqueté par un mot depuis un état donné. La notion de *non déterminisme* pour un automate signifie qu'il n'y a plus cette unicité du chemin. Ceci se traduit par :

— la possibilité de plusieurs transitions étiquetées par une même lettre depuis un même état ;

— la possibilité d'avoir plusieurs états initiaux.

On verra que cette généralisation ne change pas l'ensemble des langages reconnus. Il y a plusieurs avantages à l'introduction du non déterminisme :

— d'un point de vue pratique, les automates non déterministes sont faciles à construire, et ils ont souvent moins d'états que les automates déterministes. On verra comment construire facilement un automate non déterministe reconnaissant le langage dénoté par une expression rationnelle.

— d'un point de vue théorique, l'introduction du non déterminisme facilite considérablement certaines preuves de stabilité des langages reconnaissables (par exemple : si L est reconnaissable, le langage ${}^tL$, constitué des miroirs des mots de L, est également reconnaissable).

Par contre, il est difficile de déterminer algorithmiquement si un mot est reconnu par un automate non déterministe : il faudrait examiner tous les chemins possibles étiquetés par le mot. Heureusement, on verra une procédure permettant de déterminiser un automate non déterministe.

17.3.1 Définitions

Définition 17.27. *Un automate fini non déterministe (AFND) est un quintuplet $(\Sigma, Q, I, F, \delta)$ où :*

— *Σ est l'alphabet associé à l'automate ;*

— *Q est l'ensemble des états de l'automate ;*

— *I est l'ensemble des états initiaux ;*

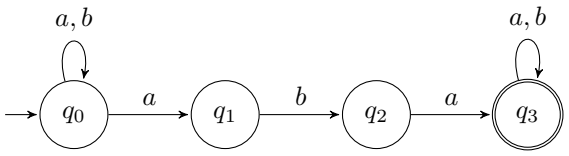
— *F est l'ensemble des états finaux ;*

— *δ est la fonction de transition, c'est une application de $Q \times \Sigma \to \mathcal{P}(Q)$*

La différence est essentiellement celle citée plus haut : il peut y avoir plusieurs états initiaux, et la fonction de transition n'est plus à valeurs dans Q mais dans l'ensemble des parties de Q, c'est à dire que plusieurs transitions depuis un état donné peuvent être étiquetées par la même lettre. La notion de « blocage » vue précédemment est légèrement modifiée : δ n'étant plus une fonction partielle, $\delta(q,a)$ est toujours défini mais peut valoir $\emptyset$.

La fonction de transition s'étend par récurrence de la même manière que précédemment : on pose $\delta^*(q,\varepsilon) = \{q\}$ pour tout $q \in Q$, et ensuite $\delta^*(q,am) = \cup_{q' \in \delta(q,a)} \delta^*(q',m)$ pour tout $(a,m) \in \Sigma \times \Sigma^*$. Autrement dit, $\delta^*(q,m)$ est l'ensemble des états que l'on peut atteindre depuis q par un chemin étiqueté par m.

Exemple 17.28. *L'automate suivant est un automate fini non déterministe :*



On a par exemple $\delta(q_0,a) = \{q_0,q_1\}$.

Définition 17.29. *Un mot m est accepté par un AFND $\mathcal{A} = (\Sigma,Q,I,F,\delta)$ s'il existe $q_0 \in I$ tel que $\delta^*(q_0,m) \cap F \neq \emptyset$. Autrement dit, un mot m est accepté par l'automate s'il existe un chemin étiqueté par m depuis l'un des états initiaux vers un état final.*

Le lecteur se convaincra rapidement que l'automate précédent reconnaît $\Sigma^* aba \Sigma^*$.

Remarque 17.30. *Les notions d'automate standard et d'automate émondé s'étendent au cas non déterministe.*

17.3.2 Détérminisation d'un automate non déterministe

On va montrer que du point de vue des langages reconnus, l'introduction du non déterminisme ne change rien, en donnant un procédé pour construire un automate déterministe équivalent à un automate non déterministe. Cette construction est la preuve du théorème suivant.

Théorème 17.31. *Soit A un automate fini non déterministe. Alors il existe un automate déterministe A' tel que $L(A) = L(A')$.*

Démonstration. Soit $A = (\Sigma,Q,I,F,\delta)$. On va construire un automate déterministe (dit *des parties*) reconnaissant $L(A)$: ses états sont l'ensemble des parties de Q. Dans l'idée, l'exécution de cet automate avec un mot m revient à simuler toutes les exécutions possibles de A avec m. Soit donc $A' = (\Sigma,\mathcal{P}(Q),I,F',\delta')$ défini par :

- l'ensemble des états de A' est l'ensemble des parties de Q ;
- l'état initial de A' est $I \in \mathcal{P}(Q)$, ensemble des états initiaux de A ;
- $F' = \{E \in \mathcal{P}(Q) \mid E \cap F \neq \emptyset\}$: l'ensemble des états finaux de A' est constitué des parties de Q contenant au moins un état final de A ;
- δ' est définie comme suit : $\delta'(E,a) = \cup_{q \in E} \delta(q,a)$.

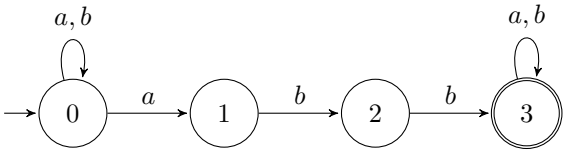
Notons que A' est déterministe complet. Pour montrer que A et A' reconnaissent le même langage, montrons par récurrence sur $|m|$ la propriété :

> Il existe un chemin étiqueté par m, depuis un état de I, vers un état q dans A si et seulement si le chemin dans A' étiqueté par m depuis I aboutit à un état E contenant q.

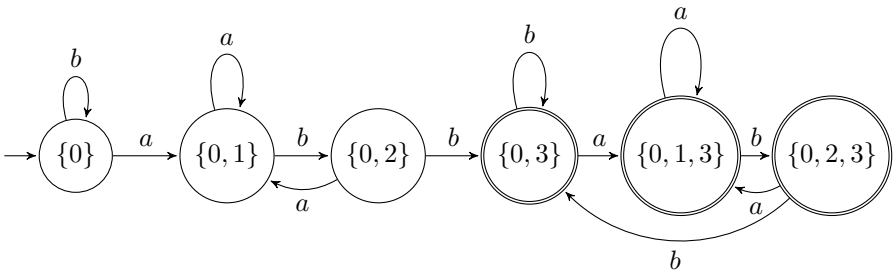
- Si $|m| = 0$, la propriété est vraie (les deux morceaux sont équivalents à $q \in I$).
- Soit maintenant un mot $m = m_1 \cdots m_n$ de longueur $n \geq 1$, et supposns la propriété vraie au rang $n-1$.
  - Supposons qu'il existe un chemin dans A étiqueté par m d'un état $q_0 \in I$ vers q. Notons $q_0 \overset{m_1}{\to} q_1 \cdots \overset{m_n}{\to} q_n = q$ ce chemin. Par hypothèse de récurrence, le chemin dans A' étiqueté par $m_1 \cdots m_{n-1}$ de I aboutit à un état E contenant q_{n-1}. Mais $\delta(E,m_n)$ contient alors q_n, donc $\delta'^*(I,m)$ aussi.
  - Réciproquement, soit $I \overset{m_1}{\to} Q_1 \cdots \overset{m_n}{\to} Q_n$ le chemin dans A' étiqueté par m, depuis I. Soit $q \in Q_n$. Comme $\delta'(Q_{n-1},m_n)$ contient q, il existe un état q_{n-1} dans Q_{n-1} tel que $q \in \delta(q_{n-1},m_n)$. Par hypothèse de récurrence, il existe un chemin étiqueté par $m_1,\ldots,m_{n-1}$ dans A, depuis un état de I vers q_{n-1}. On complète ce chemin avec la transition $q_{n-1} \overset{m_n}{\to} q$ pour terminer la preuve.
- Par principe de récurrence, la propriété est démontrée.

D'après la définition des états finaux de A', cette propriété implique le résultat : A et A' reconnaissent le même langage. □

Exemple 17.32. *Déterminisons l'automate qui suit :*



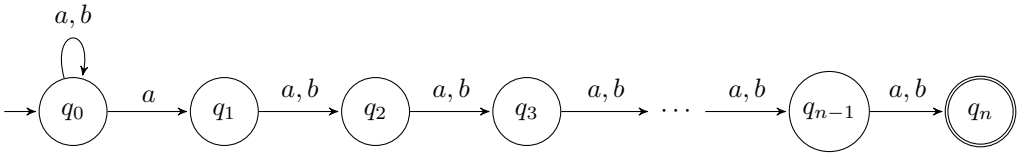
Comme il y a 4 états dans cet automate, le déterminisé en possède $2^4 = 16$*. On va se contenter de construire uniquement les états accessibles depuis* $I = \{0\}$*. Voici le résultat :*



Il y a 10 sommets non accessibles dans le déterminisé, qu'il est inutile de construire. Remarquons aussi que dans l'automate précédent, les trois derniers états pourraient être jumelés en un seul, pour obtenir un automate plus petit reconnaissant le même langage.

L'automate déterminisé peut avoir *a priori* un nombre d'états exponentiel en le nombre d'états de l'automate non déterministe initial, même après émondage. Voici un exemple.

Exemple 17.33. *Soit* $n \geq 2$*. Le langage des mots ayant un* a *à la* n*-ème position en partant de la fin, c'est à dire* $\mathcal{L}(\Sigma^* a(a+b)^{n-1})$*. Il est reconnu par l'automate non déterministe suivant, à* $n+1$ *états :*



Montrons qu'un automate déterministe $A = (\Sigma, Q, q_0, F, \delta)$ *reconnaissant* L *possède au moins* 2^n *états. Remarquons que l'ensemble des préfixes des mots de* L *est* Σ^**, donc la fonction* δ *n'a pas de blocage, ainsi la fonction suivante :*

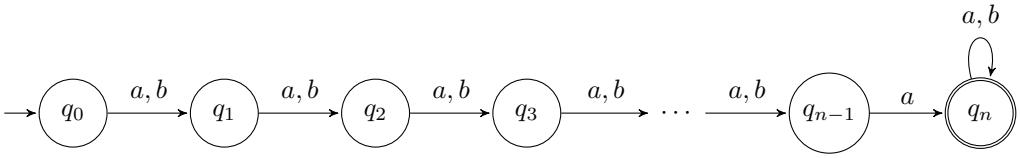
$$\begin{array}{rccc} \varphi : & \Sigma^n & \longrightarrow & Q \\ & m & \longmapsto & \delta^*(q_0, m) \end{array}$$

est bien définie. Montrons qu'elle est injective. Supposons l'existence de deux mots distincts $u = u_0 u_1 \cdots u_{n-1}$ *et* $v = v_0 v_1 \cdots v_{n-1}$ *de* Σ^n *tels que* $\varphi(u) = \varphi(v)$*. Notons* i *le premier indice où* u *et* v *diffèrent, on peut supposer* $u_i = a$ *et* $v_i = b$*. Considérons un mot quelconque* s *de taille* i*. Alors* $us \in L$*, et* $vs \notin L$*. Or*

$$F \ni \delta^*(q_0, us) = \delta^*(\varphi(u), s) = \delta^*(\varphi(v), s) = \delta^*(q_0, vs) \notin F$$

ce qui est absurde. Donc φ *est injective et* A *possède au moins* 2^n *états.*

Remarque 17.34. *Le langage précédent n'est autre que le langage miroir de* $\mathcal{L}((a+b)^{n-1}a\Sigma^*)$*, mots de* $\{a, b\}^*$ *ayant un* a *en* n*-ème position. Cet automate est reconnu par l'automate* déterministe *suivant :*



17.3.3 Automate de Glushkov et algorithme de Berry-Sethi

Cette partie est fondamentale, et explicitement au programme. Nous allons voir comment obtenir un automate reconnaissant le langage dénoté par une expression rationnelle donnée, ce qui montrera l'implication langage rationnel $\Rightarrow$ langage reconnaissable. On se donne donc e une expression rationnelle.

Rappel. Si $L(e) \neq \emptyset$, il existe une expression e' sans ε ni $\emptyset$ telle que $L(e) = L(e')$ ou $L(e) = L(e') \cup \{\varepsilon\}$. On peut donc supposer e sans ε ni $\emptyset$: en effet, si $L(e) = \emptyset$, il n'est pas dur de construire un automate reconnaissant $L(e)$, et si $L(e) = L(e') \cup \{\varepsilon\}$, il suffit de rajouter l'état initial [2] d'un automate reconnaissant $L(e')$ dans les états finaux pour obtenir un automate reconnaissant $L(e)$.

Linéarisation d'une expression rationnelle

Définition 17.35. *Soit e une expression rationnelle sans $\emptyset$ ni ε sur un alphabet Σ. Notons k le nombre de lettres de Σ apparaissant dans e,* multiplicité comprise. *On se donne un alphabet $\Sigma' = \{c_1, \dots, c_k\}$ de taille k. La linéarisation de e consiste à remplacer chaque lettre apparaîssant dans e par une lettre de Σ' pour obtenir une expression rationnelle linéaire e' de Σ', de sorte que chaque lettre de e' n'apparaîsse qu'une fois.*

Exemple 17.36. *Considérons $e = a(ba + b)^*b$ sur l'alphabet $\{a, b\}$. Sa linéarisation est $e' = c_1(c_2c_3 + c_4)^*c_5$ sur l'alphabet $\Sigma' = \{c_i \mid 1 \leq i \leq 5\}$.*

Proposition 17.37. *Soit e un expression sans $\emptyset$ ni ε sur un alphabet Σ, et e' sa linéarisée sur Σ'. Notons $\varphi(c_i)$ la lettre de Σ associée à c_i, pour tout $1 \leq i \leq k$. On peut prolonger φ en un morphisme de monoïde de Σ'^* dans Σ^*. Alors $\varphi(L(e')) = L(e)$.*

Démonstration. La preuve se fait par induction sur e. Elle est évidente si e est réduite à un seul caractère. Traitons les autres cas :

— $e = f + g$. Notons f' et g' les linéarisés (sur des alphabets disjoints) de f et g. Par hypothèse d'induction, on a $\varphi(L(f')) = L(f)$ et $\varphi(L(g')) = \varphi(L(g))$. Ainsi

$$\varphi(L(e')) = \varphi(L(f') \cup L(g')) = \varphi(L(f')) \cup \varphi(L(g')) = L(f) \cup L(g) = L(e)$$

.

— $e = fg$. Notons également f' et g' les linéarisés de f et g. Par hypothèse d'induction, on a $\varphi(L(f')) = L(f)$ et $\varphi(L(g')) = \varphi(L(g))$. Ainsi

$$\begin{array}{rcl} \varphi(L(e')) & = & \varphi(L(f')L(g')) \\ & = & \{\varphi(m_1m_2) \mid (m_1, m_2) \in (L(f'), L(g'))\} \\ & = & \{\varphi(m_1)\varphi(m_2) \mid (m_1, m_2) \in (L(f'), L(g'))\} \\ & = & \varphi(L(f'))\varphi(L(g')) \\ \varphi(L(e')) & = & L(e) \end{array}$$

.

— $e = f^* : \varphi(L(e')) = \varphi(L(f'^*)) = \varphi(L(f')^*) = \{\varphi(m_1 \dots m_k) \mid m_i \in L(f')\} = \{\varphi(m_1) \dots \varphi(m_k) \mid m_i \in L(f')\} = L(f)^* = L(e)$

□

Obtention d'un automate reconnaissant une expression rationnelle

On sait construire des automates associés à des expressions linéaires. La suppression du marquage permet de construire un automate (a priori non déterminisite) associé à un expression rationnelle quelconque.

Proposition 17.38. *Soit e une expression rationnelle sans $\emptyset$ ni ε sur un alphabet Σ, e' sa linéarisée sur $\Sigma' = \{c_1, \dots, c_k\}$. On note toujours $\varphi(c_i)$ la lettre de Σ associée à une lettre de Σ'. Considérons A' un automate local standard reconnaissant $L(e')$. Alors l'automate A obtenu en remplaçant chaque transition de A de la forme $q \xrightarrow{c_i} q'$ par la transition $q \xrightarrow{\varphi(c_i)} q'$ est un automate (non déterminisite a priori) reconnaissant $L(e)$.*

Démonstration. • Soit m un mot de $L(e) = \varphi(L(e'))$. Alors il existe un mot $m' \in L(e')$ tel que $m = \varphi(m')$. Par construction, il existe dans A' un chemin étiqueté par les lettres de m' depuis l'état initial jusqu'à un état final. Par suite, il existe dans A un chemin depuis l'état initial jusqu'à un état final étiqueté par les lettres de m, donc $m \in L(A)$.

2. ou l'un des états initiaux dans le cas non déterministe.

- Réciproquement, soit m un mot de $L(A)$. Il existe donc dans A un chemin étiqueté par les lettres de m depuis l'état initial jusqu'à un état final. En repassant par A', on construit un mot $m' \in \Sigma'^*$ tel que $\varphi(m') = m$ et m' reconnu par A'. Ainsi $m' \in L(e')$, donc $m \in \varphi(L(e')) = L(e)$.

□

Automate de Glushkov et algorithme de Berry-Sethi

Les noms de Victor Glushkov [3], Gérard Berry [4], et Ravi Sethi [5] sont restés attachés à la procédure d'obtention d'un automate déterminisite reconnaissant le langage dénoté par une expression rationnelle quelconque.

Définition 17.39. *L'automate de Glushkov d'une expression rationnelle e est le déterminisé de l'automate reconnaissant $L(e')$ obtenu précédemment.*

La procédure d'obtention de l'automate de Glushov s'appelle *algorithme de Berry-Sethi*. On le résume ci-dessous (algorithme 17.40).

Algorithme 17.40 : Algorithme de Berry-Sethi

Entrée : Une expression linéaire e sans ε ni $\emptyset$ sur un alphabet Σ.
Sortie : Un automate reconnaissant $L(e)$
Linéariser e en une expression linéaire e' sur un alphabet Σ';
Calculer un automate local standard A' reconnaissant $L(e')$;
Supprimer les marques de la linéarisation en rétablissant les lettres de Σ à la place des lettres de Σ' ;
Déterminiser l'automate obtenu ;
Renvoyer l'automate déterministe calculé.

Exemple

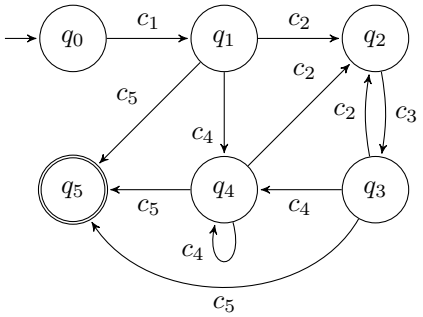
Prenons un exemple complet. Considérons l'expression $e = a(ba + b)^*b$ sur l'alphabet $\Sigma = \{a, b\}$.

Linéarisation. On linéarise e sur l'alphabet $\Sigma' = \{c_1, c_2, c_3, c_4, c_5\}$ en $c_1(c_2c_3 + c_4)^*c_5$.

Calcul des ensembles caractéristiques. On calcule les ensembles $P(L(e'))$, $S(L(e'))$, $F(L(e'))$:

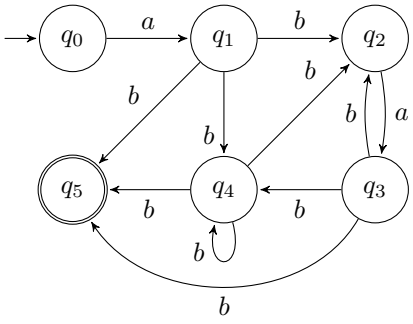
$$P(L(e')) = \{c_1\} \qquad S(L(e')) = \{c_5\} \qquad F(L(e')) = \{c_1c_2, c_1c_4, c_1c_5, c_2c_3, c_3c_2, c_3c_4, c_3c_5, c_4c_2, c_4^2, c_4c_5\}$$

Calcul d'un automate local standard reconnaissant $L(e')$. Une fois les ensembles caractéristiques calculés, on produit l'automate reconnaissant $L(e')$.

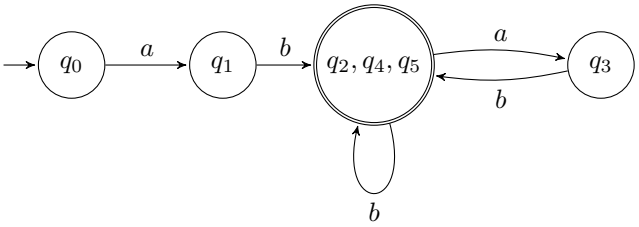


Suppression des marques de la linéarisation. Il suffit de rétablir les lettres de $\{a, b\}$ à la place des lettres de Σ' pour obtenir un automate non déterministe reconnaissant $L(e)$.

3. Mathématicien et informaticien soviétique.
4. Informaticien français.
5. Informaticien indien.



Déterminisation. Il reste alors à déterminiser l'automate obtenu. En ne construisant que les états accessibles, on obtient ici un automate ayant un nombre particulièrement faible d'états, mais c'est du à la simplicité de l'expression choisie.



Complexité.

La complexité d'un algorithme permettant de linéariser une expression rationnelle e et calculer un automate local standard reconnaissant le langage dénoté par l'expression linéarisée dépend un peu de la structure utilisée (listes ? tableaux ? dictionnaires ?). Sans rentrer dans les détails, notons simplement que cette complexité est polynomiale en la longueur $\ell(e)$ de l'expression e.

L'automate local obtenu possède $\ell'(e)+1$ états, où $\ell'(e)$ est le nombre de lettres dans l'expression e (on ne compte pas union, concaténation et étoile de Kleene). L'étape coûteuse est la déterminisation de l'automate obtenu par suppression des marques de la linéarisation, puisque le nombre d'états est $2^{\ell'(e)+1}$.

Pour résumer, la complexité de l'algorithme de Berry-Sethi est $O(2^{\ell(e)}P(\ell(e)))$, où P est un polynôme.

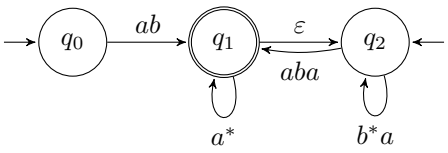
17.4 Automates généralisés et élimination des états

17.4.1 Automates étiquetés par des expressions rationnelles

On va généraliser encore un peu plus la notion d'automates, en autorisant que les transitions soient étiquetées non plus par des lettres de Σ, mais par des expressions rationnelles.

Définition 17.41. *Un automate généralisé est défini de la même manière qu'un automate fini non déterministe, sauf que les transitions entre états sont étiquetées par des expressions rationnelles plutôt que par des lettres.*

Exemple 17.42. *Voici un exemple d'automate généralisé sur $\{a, b\}$:*



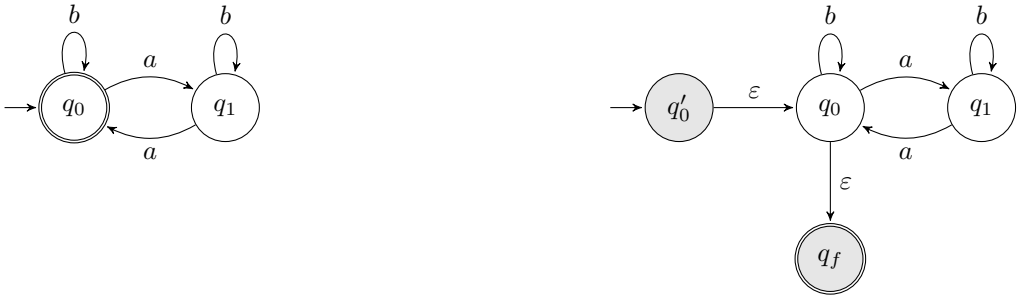
Définition 17.43. *Un mot w est accepté par un tel automate généralisé s'il existe un chemin dans l'automate, d'un état initial à un état final, tel que w est la concaténation de mots appartenant successivement aux langages dénotés par chacune des expressions rationnelles étiquetant le chemin. Deux tels automates sont dits équivalents s'ils reconnaissent le même langage.*

Exemple 17.44. *L'automate précédent reconnaît le mot $w = abbbbaaba$. En effet, avec $w_0 = ab$, $w_1 = \varepsilon$, $w_2 = bbba$, $w_3 = aba$, $w = w_0w_1w_2w_3$, et le chemin dans l'automate $q_0 \xrightarrow{ab} q_1 \xrightarrow{\varepsilon} q_2 \xrightarrow{b^*a} q_2 \xrightarrow{aba} q_1$ reliant un état initial à un état final montre que w est accepté, avec w_0 (resp. w_1, w_2, w_3) dans le langage dénoté par ab (resp. ε, b^*a, aba).*

Remarque 17.45. *(Hypothèses supplémentaires) Sans pertes de généralités, on suppose dorénavant que :*

- *entre deux états il y a au plus une transition (deux transitions $q \xrightarrow{e} q'$ et $q \xrightarrow{e'} q'$ sont équivalentes à $q \xrightarrow{e+e'} q'$) ;*
- *il y a un seul état initial et un seul état final, aucune transition n'aboutit à l'état initial et aucune ne provient de l'état final. En effet, à partir d'un automate A dont l'ensemble des états initiaux est I et l'ensemble des états finaux est F, on peut obtenir un automate équivalent respectant cette condition en rajoutant un état initial q_0, et des transitions étiquetées par ε de q_0 aux états de I (qui ne sont plus initiaux !), et de même pour un état final q_f et des transitions étiquetées par ε des états de F vers q_f.*

Exemple 17.46. *Voici un automate déterministe et l'automate de Thomson équivalent.*

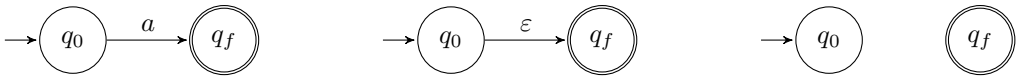


Remarque 17.47. *Comme on le voit, les transitions étiquetées par ε, aussi appelées ε-transitions ou transitions instantanées, sont importantes dans cette construction.*

17.4.2 Automate de Thomson associé à une expression rationnelle

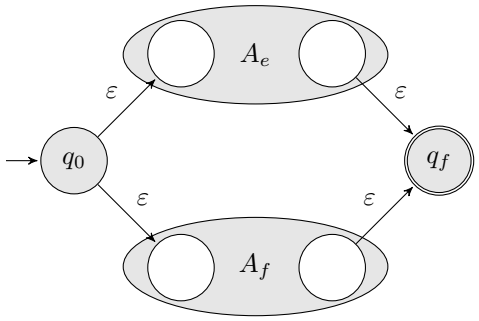
À partir d'une expression rationnelle e, il est facile de créer un automate généralisé reconnaissant $L(e)$, l'automate en question ayant toutes ses transitions étiquetées par des lettres de Σ ou qui sont instantanées. On obtient une construction différente de celle donnée par l'algorithme de Berry-Sethi. Cette construction est inductive.

Automate associé à une lettre, ε, $\emptyset$. Un automate possédant deux états q_0 initial et q_f final, et une transition $q \to q'$ étiquetée par $a \in \Sigma$ (resp. ε, resp. pas de transition) reconnaît le langage $\{a\}$, (resp. ε, resp. $\emptyset$).

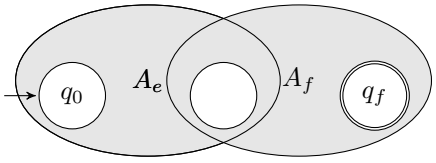


Dans la suite, on se donne deux automates généralisés A_e et A_f reconnaissant $L(e)$ et $L(f)$, avec e et f deux expressions rationnelles. Rappel : un seul état initial et un seul état final pour chaque, aucune transition n'aboutit aux états initiaux et n'est issue d'un état final.

Automate associé à $e + f$. On rajoute un état initial q_0, un état final q_f, des ε-transitions de q_0 vers les anciens états initiaux de A_e et A_f, et des anciens états finaux vers q_f. L'automate obtenu reconnaît $L(e + f)$.



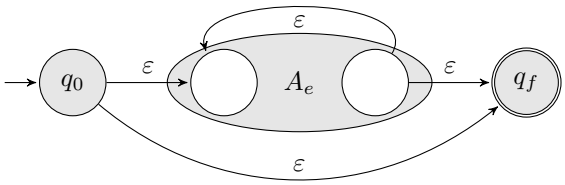
Automate associé à $e \cdot f$. Il suffit d'identifier l'état terminal de A_e avec l'état initial de A_f. L'état initial du nouvel automate est celui de A_e, l'état final celui de A_f.



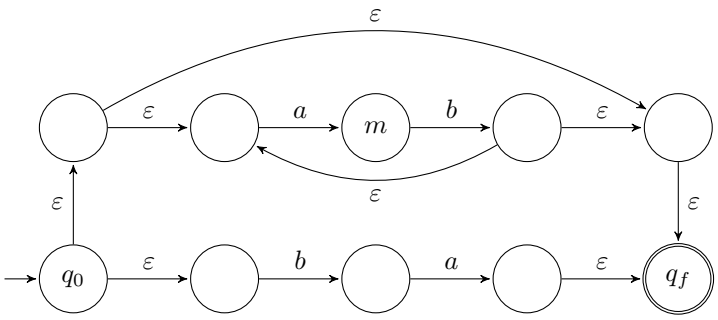
Automate associé à e^*. On rajoute deux états q_0 et q_f (un état initial et un état final), et des ε transitions de q_0 vers l'ancien état initial de A_e, et de même de l'ancien état final vers q_f. On rajoute :

— une ε-transition de l'ancien état final vers l'ancien état initial ;

— une ε-transition de q_0 à q_f.



Exemple 17.48. *Voici l'automate de Thomson obtenu via l'expression rationnelle $(ab)^* + ba$.*



Obtention d'un automate déterministe reconnaissant e. Pour obtenir un automate déterministe reconnaissant e, il suffirait de déterminiser l'automate précédent. Toutefois, il faut gérer les ε-transitions :

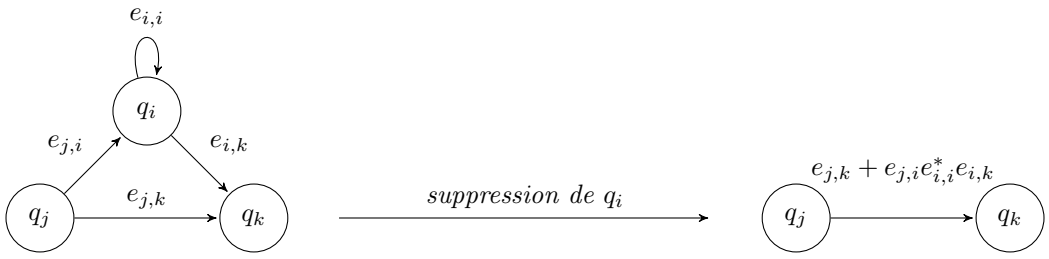
— dans la pratique, lorsqu'on procède « à la main », il suffit de voir les ε-transitions comme un moyen de passer d'un état à l'autre sans utiliser de lettre. Dans l'exemple précédent, depuis l'état q_0, l'état m est accessible via la lettre a (c'est le seul). Depuis m, via la lettre b, on peut accéder à 4 états en suivant en plus entre zéro et deux ε transitions.

— algorithmiquement, on pourrait supprimer préalablement les ε-transitions en appliquant un algorithme de calcul d'une *fermeture transitive* de graphe : il s'agit de calculer pour chaque état tous les états accessibles en ne suivant que des ε-transitions.

17.4.3 Algorithme d'élimination des états

On a vu que le langage dénoté par une expression rationnelle est reconnaissable par un automate. Dans cette sous-section, on se pose la question inverse : que peut-on dire du langage reconnu par un automate (généralisé ?). On va voir qu'il s'agit d'un langage rationnel, dont on peut calculer une expression rationnelle associée via l'algorithme suivant, de *Brzozowski* et *McCluskey*. Il consiste à éliminer un à un les états d'un automate généralisé, jusqu'à n'avoir plus que l'état initial et l'état final. Bien sûr, les automates obtenus à chaque étape sont tous équivalents. Rappel : dans un automate généralisé, il n'y a qu'un état initial, sans transition qui y aboutit et un seul état final, sans transition qui en est issue. Entre deux états, il y a au plus une transition. Une étape de l'élimination repose sur la propriété suivante :

Proposition 17.49. *Soit A un automate généralisé, ayant au moins trois états. Pour q_i, q_j deux états éventuellement égaux de l'automate, on note $e_{i,j}$ l'expression rationnelle étiquetant la transition de q_i à q_j (on peut toujours considérer que cette transition existe quite à poser $e_{i,j} = \emptyset$). Considérons un état q_i qui n'est ni l'état initial, ni l'état final. L'automate A' obtenu en remplaçant chacune des transitions $q_j \xrightarrow{e_{j,k}} q_k$ pour tout couple d'états (q_j, q_k) éventuellement égaux mais distincts de q_i, par $e_{j,k} + e_{j,i}e_{i,i}^* e_{i,k}$ et en supprimant l'état q_i, est un automate généralisé équivalent à A. L'opération est résumée dans le schéma ci-dessous.*



Démonstration.

- Soit w un mot de $L(A)$. Notons $q_0 \to \cdots \to q_f$ un chemin dans l'automate, de l'état initial à l'état final montrant que w est reconnu : w se décompose en facteurs, ceux-ci étant dans les langages dénotés par les expressions rationnelles successives. Le chemin dans l'automate se décompose en portions de la forme

$$q_j \overset{e_{j,i}}{\to} q_i \overset{e_{i,i}}{\to} \cdots \overset{e_{i,i}}{\to} q_i \overset{e_{i,k}}{\to} q_k$$

  avec $i \notin \{j,k\}$ (le sommet q_i apparaît un nombre quelconque de fois, qui peut être zéro). Le facteur m de w concerné par ce chemin est dans $L(e_{j,k})$ si q_i n'apparaît pas, dans $L(e_{j,i}e_{i,i}^*e_{i,k})$ sinon. Dans tous les cas, il est dans $L(e_{j,k} + e_{j,i}e_{i,i}^*e_{i,k})$. On en déduit un chemin dans A' qui montre que w est dans $L(A')$.
- Réciproquement, si $w \in L(A')$, à un chemin dans A' et une décomposition en facteurs montrant que w est reconnu, on associe un chemin dans A : pour un tel facteur $m \in L(e_{j,k} + e_{j,i}e_{i,i}^*e_{i,k})$ associé à la transition $q_j \to q_k$ de A', on associe la même transition et le même facteur si $m \in L(e_{j,k})$. Sinon, m est de la forme $m_{j,i}m_{i,i}^1 \cdots m_{i,i}^p m_{i,k}$, avec $m_{j,i} \in L(e_{j,i})$, $m_{i,k} \in L(e_{i,k})$ et $m_{i,i}^\ell \in L(e_{i,i})$ pour $\ell \in [\![1,p]\!]$. À cette décomposition de m on associe le chemin $q_j \overset{e_{j,i}}{\to} q_i \overset{e_{i,i}}{\to} \cdots \overset{e_{i,i}}{\to} q_i \overset{e_{i,k}}{\to} q_k$ où q_i apparaît $p+1$ fois. Le chemin et la décomposition de w obtenus montre que $w \in L(A)$.

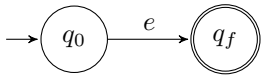
□

De la proposition précédente, on déduit le théorème suivant :

Théorème 17.50. *Le langage reconnu par un automate généralisé est rationnel.*

Démonstration. On le montre par récurrence sur le nombre d'états de l'automate généralisé. La récurrence démarre à 2 puisqu'il y a au moins l'état initial et l'état final.

— Un automate généralisé à deux états est de la forme



où e est une expression rationnelle. Mais le langage reconnu par un tel automate est clairement $L(e)$, rationnel.

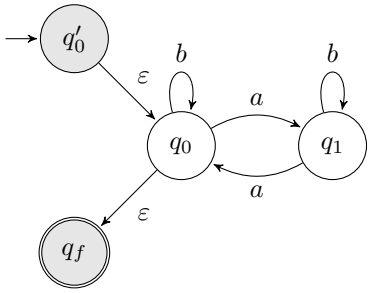
— Soit $n \geq 2$ et considérons un automate généralisé A à $n+1$ états, en supposant la propriété démontrée pour l'entier n. La proposition précédente montre que l'on peut supprimer un état de A, en gardant le même langage reconnu. Par hypothèse de récurrence, ce langage est donc rationnel.

— Par principe de récurrence, la propriété est démontrée.

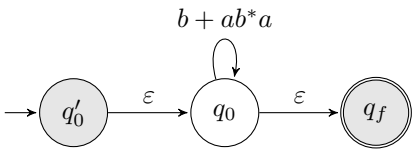
□

La preuve du théorème fournit un algorithme permettant de calculer une expression rationnelle dénotant le langage reconnu par un automate généralisé : il suffit de supprimer un à un tous les états, sauf l'état initial et l'état final.

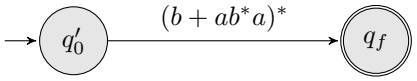
Exemple 17.51. *Calculons une expression rationnelle dénotant le langage reconnu par l'automate suivant.*



On élimine d'abord q_1 pour obtenir



On élimine q_0 :



Le langage reconnu par l'automate est donc $L((b + ab^*a)^*)$.

Remarque 17.52. *Éliminer q_0 en premier aurait donné l'expression rationnelle $b^*+b^*a(b+ab^*a)^*ab^*$, plus compliquée, mais bien sûr équivalente.*

17.4.4 Théorème de Kleene

On a introduit dans ce chapitre et dans le précédent plusieurs types de langages :

— les langages rationnels (dénotés par une expressions rationnels) ;

— les langages reconnus par un automate déterministe ;

— les langages reconnus par un automate non déterministe ;

— les langages reconnus par un automate « de Thomson » étiqueté par des expressions rationnelles.

Les résultats montrés dans les deux dernières sections montrent le théorème suivant.

Théorème 17.53. *(Kleene) Sur un alphabet Σ donné, les langages rationnels coïncident avec les langages reconnaissables.*

Démonstration. On a déja vu que l'ajout du non déterminisme ne change pas l'ensemble des langages reconnus. L'algorithme de Berry-Sethi montre qu'un langage rationnel est reconnu par un automate non déterministe, qui est un cas particulier d'automate de Thomson. Enfin, l'algorithme d'élimination des états montre qu'un langage reconnu par un automate de Thomson est rationnel. □

Ce théorème est remarquable, car il relie deux propriétés très différentes sur les langages, l'une définie de manière algébrique (rationnalité) et l'autre procédurale (reconnaissabilité).

17.5 Stabilité des langages rationnels

L'équivalence entre langages rationnels et reconnaissables permet de montrer certaines stabilités de l'ensemble des langages rationnels, qui auraient été difficiles à établir sans recourir aux automates.

17.5.1 Opérations ensemblistes

Naturellement, l'ensemble des langages rationnels est stable par union, concaténation et passage à l'étoile, parce que les langages rationnels sont définis ainsi. Mais cet ensemble est stable par beaucoup d'autres opérations ensemblistes.

Complémentation

Proposition 17.54. *Soit L un langage rationnel sur l'alphabet Σ. Alors $\Sigma^*\backslash L$ est rationnel.*

Démonstration. Soit $A = (\Sigma, Q, q_0, F, \delta)$ un automate déterministe *complet* reconnaissant L. Alors $\bar{A} = (\Sigma, Q, q_0, Q\backslash F, \delta)$ reconnaît $\Sigma^*\backslash L$, car pour tout mot $m \in \Sigma^*$, on a $\delta^*(q_0, m) \notin F \Leftrightarrow \delta^*(q_0, m) \in Q\backslash F$. □

Remarque 17.55. *Dans la preuve précédente, il est essentiel de supposer l'automate complet, sinon les mots induisant un blocage dans A en induisent aussi un dans $\bar{A}$.*

Intersection

Proposition 17.56. *Soit L_1, L_2 deux langages rationnels sur l'alphabet Σ. Alors $L_1 \cap L_2$ est rationnel.*

Démonstration. On se donne $A_i = (\Sigma, Q_i, q_0^i, F_i, \delta_i)$ un automate déterministe reconnaissant L_i, pour $i \in \{1, 2\}$. On va considérer l'*automate produit* de A_1 et A_2, c'est à dire : $A = (\Sigma, Q_1 \times Q_2, q_0 = (q_0^1, q_0^2), F = F_1 \times F_2, \delta)$, où δ est définie par :

$$\delta((q_1, q_2), a) = \begin{cases} (\delta_1(q_1, a), \delta_2(q_2, a)) & \text{si ni l'un ni l'autre n'est un blocage} \\ \text{blocage} & \text{sinon.} \end{cases}$$

Alors $L(A) = L(A_1) \cap L(A_2)$. En effet

$$\delta^*(q_0, m) \in F \qquad \Longleftrightarrow \qquad \delta_1^*(q_0^1, m) \in F_1 \quad \text{et} \quad \delta_2^*(q_0^2, m) \in F_2$$

□

Remarque 17.57. *L'automate produit consiste essentiellement à exécuter simultanément m sur les deux automates.*

17.5.2 Préfixes, suffixes, facteurs, sous-mots, miroir...

On termine cette section par le théorème suivant. Pour L un langage sur l'alphabet Σ, on note $\mathrm{Pref}(L)$, $\mathrm{Suff}(L)$, $\mathrm{Fact}(L)$, $\mathrm{SM}(L)$ et ${}^tL$ les langages constitués des préfixes des mots de L, des suffixes des mots de L, des facteurs des mots de L, des sous-mots des mots de L et enfin des miroirs des mots de L :

$${}^tL = \{m_k m_{k-1} \cdots m_1 \mid m_1 m_2 \cdots m_k \in L\}$$

Théorème 17.58. *Si L est un langage rationnel sur Σ, alors $\mathrm{Pref}(L)$, $\mathrm{Suff}(L)$, $\mathrm{Fact}(L)$, $\mathrm{SM}(L)$ et ${}^tL$ sont rationnels.*

Démonstration. La preuve utilise encore l'équivalence entre langage rationnel et langage reconnaissable. On suppose L non vide (sinon il n'y a rien à montrer) et on se donne un automate déterministe *émondé* $A = (Q, \Sigma, q_0, F, \delta)$ reconnaissant L. Alors :

- $(Q, \Sigma, q_0, Q, \delta)$ est un automate déterministe reconnaissant $\mathrm{Pref}(L)$: le fait que l'automate soit émondé est essentiel, ensuite ce n'est pas difficile à vérifier.
- $(Q, \Sigma, Q, F, \delta')$ avec δ' la fonction de transition similaire à δ mais en version non déterministe, est un automate non déterministe reconnaissant $\mathrm{Suff}(L)$.
- $(Q, \Sigma, Q, Q, \delta')$ est un automate non déterministe reconnaissant $\mathrm{Fact}(L)$.
- on laise au lecteur le soin de rajouter des transitions à l'automate précédent pour en obtenir un qui reconnaît $\mathrm{SM}(L)$.
- $(Q, \Sigma, F, \{q_0\}, {}^t\delta)$ est un automate non déterministe reconnaissant ${}^tL$, avec ${}^t\delta$ définie par :

  $${}^t\delta(q, a) = \{q' \in Q \mid \delta(q', a) = q\}$$

  En clair : on a inversé état initial et états terminaux, et changé le sens des transitions.

□

17.6 Lemme de l'étoile

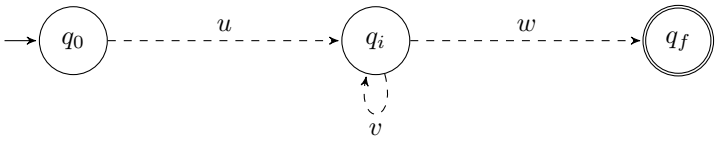
On sait maintenant que les langages reconnaissables sont rationnels, mais on ne sait pas déterminer si un langage donné est reconnaissable. Une condition nécessaire (mais non suffisante) que doit satisfaire un langage pour être reconnaissable est qu'il satisfasse le lemme de l'étoile, qu'on présente maintenant.

Lemme 17.59 (Lemme de l'étoile)**.** *Soit L un langage rationnel. Alors il existe un entier N, tel que tout mot m de longueur supérieur ou égale à N de L s'écrive sous la forme $m = uvw$, avec :*

$$|uv| \leq N, \quad |v| \geq 1 \qquad \textit{et} \qquad \textit{pour tout } k \geq 0, \quad uv^k w \in L.$$

Démonstration. On utilise l'équivalence entre le langage rationnel et reconnaissable. Soit A un automate déterministe reconnaissant L, notons N son nombre d'états. considérons un mot m de L de longueur au moins N, et considérons $m_0, \ldots, m_{N-1}$ ses N premières lettres. Considérons q_0 l'état initial, et $q_{i+1} = \delta(q_i, m_i)$ pour $i \in \{0, \ldots, N-1\}$. On obtient une séquence de $N+1$ états, donc deux d'entre eux sont égaux, disons q_i et q_j avec $i < j$. On pose alors $u = m_0 m_1 \cdots m_{i-1}$ le préfixe de longueur i de m, $v = m_i \cdots m_{j-1}$ le facteur de longueur $j - i$ qui suit, et enfin $w = m_j \ldots m_{n-1}$ le suffixe de m tel que $m = uvw$. Alors par construction, $|v| \geq 1$ et $|uv| \leq N$. De plus, $\delta(q_0, uvw) \in F$. Or $\delta(q_0, u) = \delta(q_0, uv) = q_i = q_j$ Il s'ensuit par récurrence immédiate que $\delta(q_0, uv^k) = q_i$ pour tout $k \geq 0$, et $\delta(q_0, uv^k w) \in F$, donc $uv^k w \in L$. □

Remarque 17.60. *La preuve du lemme, qui s'appelle aussi lemme de pompage ou lemme de gonflement*[6]*, se mémorise bien avec l'image suivante :*



Remarque 17.61. *Il existe des langages non rationnels qui vérifient le lemme de l'étoile, qui n'est donc pas une condition suffisante (voir feuille d'exercices). On peut donner des versions plus fortes où on « pompe » sur n'importe quel facteur de longueur N.*

Exemple 17.62. *On donne quelques exemples d'application du lemme.*

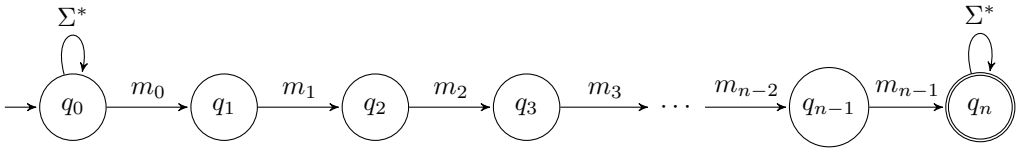
- *Le langage $L = \{a^n b^n \mid n \in \mathbb{N}\}$ n'est pas rationnel. Supposons en effet qu'il vérifie le lemme de l'étoile, donnons nous un mot de longueur au moins N du langage, avec N l'entier du lemme. On décompose $m = uvw$ comme dans le lemme. Alors v ne peut contenir à la fois les lettres a et b car sinon on aurait ba comme motif de $uv^2w \in L$ (absurde). Donc v n'est constitué que de a ou que de b, mais alors uv^2w contient un nombre différent de a et de b (absurde de nouveau). Donc L n'est pas rationnel.*
- *Le langage D des mots de Dyck n'est pas rationnel. En effet, s'il l'était, le précédent le serait aussi car c'est $D \cap L(a^*b^*)$.*
- *Le langage $L = \{a^p \mid p \text{ premier}\}$ n'est pas rationnel. Dans le cas contraire le lemme de l'étoile donnerait l'existence de p et $q > 0$ tel que $a^{p+qn} \in L$ pour tout $n \geq 0$. On peut supposer p arbitrairement grand (donc supérieur à 2), et a^{p+qp} serait dans L, absurde.*

17.7 Application à la reconnaissance de motifs. Expressions régulières étendues.

17.7.1 Reconnaissance de motifs

On s'intéresse au problème de savoir si un mot m est facteur d'une chaîne s. En termes de langages, il s'agit de savoir si $s \in L(\Sigma^* m \Sigma^*)$. En général lorsqu'on souhaite répondre à cette question, la chaîne s est grande par rapport à m. L'algorithme naïf a une complexité $O(|m|\,|s|)$. On souhaite faire mieux, l'idéal serait de n'avoir à parcourir s qu'une seule fois, donc un algorithme en $O(|s| +$ coût indépendant de $s)$. Il est facile d'obtenir un tel algorithme à l'aide d'automates.

Automate reconnaissant $L(\Sigma^* m \Sigma^*)$. Notons $m_0, \ldots, m_{n-1}$ les lettres du mot m. Alors l'automate suivant (non déterministe) reconnaît $L(\Sigma^* m \Sigma^*)$.



Algorithme de reconnaissance de motifs. Pour tester si m est motif de s, il suffit donc de :

— construire l'automate précédent reconnaissant $L(\Sigma^* m \Sigma^*)$;

— le déterminiser ;

— exécuter l'automate sur s, pour tester si $s \in L(\Sigma^* m \Sigma^*)$.

Complexité. Une fois l'automate déterministe calculé, l'exécution de l'automate sur s se fait en temps $O(|s|)$, indépendamment de la taille de Σ si les transitions sont implémentées par une table de hachage[7]. Par contre, la construction de l'automate déterminisite a *a priori* une complexité exponentielle en $|m|$. Ce n'est en général pas génant car m est petit dans la pratique, mais un peu décevant quand même.

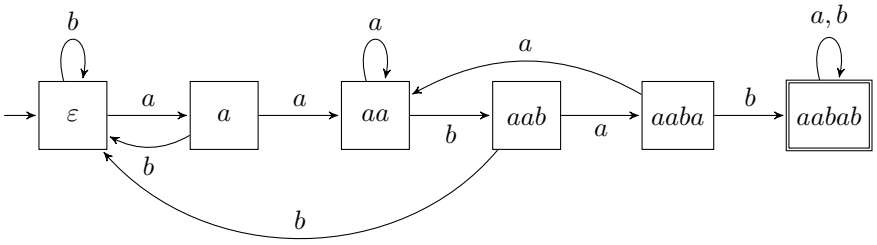
6. Dénominations assez explicites, mais à titre personnel je préfère « lemme de l'étoile ».
7. Complexité constante amortie seulement.

17.7.2 Algorithme KMP

L'algorithme KMP (voir TP) revient à construire directement un automate déterministe à $O(|m|)$ états reconnaissant $\Sigma^* m \Sigma^*$. Dans la suite, on note $P(m)$ l'ensemble des préfixes du mot m. On considère l'automate déterministe à $|m|+1$ états $K = (\Sigma, P(m), \varepsilon, \{m\}, \delta)$, où la fonction de transition δ est définie comme suit, avec $s(v)$ le plus long suffixe de v qui est aussi préfixe de m.

$$\delta(p,a) = \begin{cases} m & \text{si } p = m \\ s(pa) & \text{sinon.} \end{cases}$$

Exemple 17.63. *L'automate suivant reconnaît $\Sigma^* aabab \Sigma^*$*



Théorème 17.64. *L'automate K reconnaît $\Sigma^* m \Sigma^*$.*

Démonstration. On va montrer par récurrence sur $|u|$ que la propriété suivante est vraie : « Pour $p \in P(m)$ et $u \in \Sigma^*$, si pu ne contient pas m comme facteur, alors $\delta^*(p,u) = s(pu)$ ».

- la propriété est claire pour $u = \varepsilon$, car $s(p) = p$;
- supposons $u \neq \varepsilon$, u s'écrit alors va avec $v \in \Sigma^*$ et $a \in \Sigma$. Par hypothèse de récurrence, $\delta^*(p,v) = s(pv)$. Puisque pu ne contient pas m comme facteur, $s(pv) \neq m$. On a donc $\delta^*(p,u) = \delta(s(pv),a) = s(s(pv)a)$. Il reste à montrer que $s(s(pv)a) = s(pva) = s(pu)$:
  - $s(s(pv)a)$ est un suffixe de pva qui est préfixe de m, donc $s(s(pv)a)$ est de taille inférieure à $s(pva) = s(pu)$;
  - $s(pu)$ est le plus long suffixe de pu qui est préfixe de m. Si $s(pu) = \varepsilon$, alors $s(s(pv)a) = \varepsilon$ et il n'y a rien à montrer. Sinon, la dernière lettre de $s(pu)$ est a, donc $s(pu)$ s'écrit wa, où w est un suffixe de pv, tel que wa préfixe de m. Or w est également un préfixe de m, suffixe de pv, donc de taille inférieure à $s(pv)$.

  Ainsi, $s(s(pv)a) = s(pu)$ et l'hérédité est démontrée.
- Par principe de récurrence, on a bien $\delta^*(p,u) = s(pu)$ pour tout $p \in P(m)$ et $u \in \Sigma^*$, tel que pu ne contient pas m comme facteur.

Il reste à conclure la preuve à partir de cette propriété. La démonstration précédente s'étend au cas où m est suffixe de pu sans être facteur de pu à une autre position (on a donc $\delta(p,u) = m$). Comme l'état m est un puits de l'automate, on montre ainsi que si m est facteur de u, alors $\delta(\varepsilon, u) = m$. Réciproquement, si m n'est pas facteur de u, alors $\delta(\varepsilon,u) = s(u) \neq m$. Cette discussion montre que le langage reconnu par l'automate est exactement $\Sigma^* m \Sigma^*$. □

Remarque 17.65. *L'algorithme KMP (voir TP) permet en fait de construire un simple tableau contenant les tailles des* bords maximaux *des préfixes de m, ce qui encode en fait l'automate sous une forme plus compacte. Mais l'idée est exactement celle ci-dessus. On en déduit un algorithme de complexité $O(|m| + |u|)$ permettant de tester si m est facteur de u. De plus une fois l'automate (ou de manière équivalente, le tableau des bords maximaux) construit, on peut tester si m est facteur d'un mot quelconque en temps linéaire en la taille de ce mot : le prétraitement n'est effectué qu'une fois.*

17.7.3 Autres problèmes de reconnaissance

On a défini l'ensemble des expressions rationnelles (et les langages rationnels associés), par induction, en autorisant les suymboles de concaténation, d'union, et l'étoile de Kleene. Les théorèmes de stabilité montre que l'on peut autoriser d'autres symboles, comme par exemple l'intersection et la différence, sans changer l'ensemble des langages dénotés par ces expressions : cet ensemble est toujours celui des langages rationnels. On peut parler *d'expressions rationnelles étendues* pour les expressions rationnelles construites avec ces nouveaux symboles.

On a donné un moyen pratique de construire des automates associés à ces expressions rationnelles étendues : on peut donc algorithmiquement tester si un mot appartient à un langage dénoté par une telle expression. Ceci permet par exemple de répondre aux questions qui suivent :

— lignes d'un texte contenant toutes les voyelles, sauf y ?
— lignes d'un texte contenant père, mère et fils, mais pas fille ?
— etc...

Chapitre 18

Structures usuelles en Ocaml

18.1 Introduction

Au chapitre 1, on a décrit les structures abstraites au programme : piles, files, files de priorité et dictionnaires. Le long du présent ouvrage, on s'est attaché à donner une ou plusieurs implémentations de ces structures. Évidemment, elles sont déja présentes en Ocaml : le but du chapitre est simplement de donner les différentes modules et fonctions associées.

18.2 Piles

Pile se traduit par Stack en anglais : c'est le nom du module en Ocaml.

18.2.1 Fonctions usuelles

Voici les fonctions usuelles de pile :

— `Stack.create` de type `unit -> 'a Stack.t`, crée et renvoie une pile initialement vide ;

— `Stack.is_empty` de type `'a Stack.t -> bool` teste si la pile passée en entrée est vide ;

— `Stack.push` de type `'a -> 'a Stack.t -> unit` ajoute `x` à la pile `p` via l'appel `Stack.push x p` ;

— `Stack.pop` de type `'a Stack.t -> 'a` supprime et renvoie le sommet de la pile, si celle-ci est non vide ;

— `Stack.top` de type `'a Stack.t -> 'a` renvoie le sommet de la pile, si celle-ci est non vide.

L'exception levée si on essaie d'accéder au sommet ou de dépiler une pile vide est `Stack.Empty`.

18.2.2 Fonctions additionnelles

Le module `Stack` offre la possibilité de réaliser quelques opérations supplémentaires, en voici quelques-unes :

— `Stack.length` de type `'a Stack.t -> int`, renvoie le nombre d'éléments de la pile, s'exécute en temps constant.

— `Stack.clear` de type `'a Stack.t -> unit`, vide la pile passée en entrée ;

— `Stack.copy` de type `'a Stack.t -> 'a Stack.t` renvoie une copie de la pile.

18.3 Files

Une file est appelée queue en anglais. Là-encore, c'est le nom du module. Les fonctions sont assez semblables à celles sur les piles.

18.3.1 Fonctions usuelles

Voici les fonctions usuelles de file :

— `Queue.create` de type `unit -> 'a Queue.t`, crée et renvoie une file initialement vide ;

— `Queue.is_empty` de type `'a Queue.t -> bool` teste si la file passée en entrée est vide ;

— `Queue.add` de type `'a -> 'a Queue.t -> unit` ajoute `x` à la file `f` via l'appel `Queue.add x f` ;

— `Queue.pop` de type `'a Queue.t -> 'a` supprime et renvoie l'élément en tête de file, si celle-ci est non vide. `Queue.take` est un synonyme ;

L'exception levée si on essaie de défiler une file vide est `Queue.Empty`.

18.3.2 Fonctions additionnelles

De même que pour les piles, le module `Queue` possède les fonctions `Queue.length`, `Queue.clear` et `Queue.copy`. Signalons aussi `Queue.peek`, de type `'a Queue.t -> 'a` renvoyant l'élément en tête de file sans le supprimer.

18.4 Dictionnaires

Les dictionnaires en Ocaml sont implémentés via des tables de hachage, on trouvera donc dans le module `Hashtbl` les fonctions adéquates. Les tables de hachage en Ocaml on une petite spécificité par rapport aux dictionnaire abstraits décrits dans le chapitre 2 : il est possible d'associer plusieurs valeurs à une clé donnée. Lorsqu'on essaie d'ajouter le couple (k, e) à la table et que k est déja présente (disons que k est associé à e'), alors (k, e) masque (k, e'). Si on supprime l'entrée de clé k, alors seul (k, e) sera supprimé, et le dictionnaire associe alors k à e'. On ne donne que quelques fonctions ici, il y en a d'autres (comme `clear` et `copy`...)

— `Hashtbl.create` de type `int -> ('a, 'b) Hashtbl.t`, crée et renvoie un dictionnaire vide. L'entier passé en paramètre est une estimation du nombre d'entrées qu'il y aura dans la table, mais un appel avec paramètre 0 est tout à fait possible.

— `Hashtbl.length` donne le nombre d'entrées dans la table (le nombre de clés hachées).

— `Hashtbl.mem` de type `('a, 'b) Hashtbl.t -> 'a -> bool` teste s'il existe un couple de clé donnée dans la table.

— `Hashtbl.add` de type `('a, 'b) Hashtbl.t -> 'a -> 'b -> unit`. `Hashtbl.add h x y` ajoute le couple (x, y) au dictionnaire, et masque l'éventuelle valeur déja associée à la clé x.

— `Hashtbl.find` de type `('a, 'b) Hashtbl.t -> 'a -> 'b` renvoie l'élément associé à la clé passée en paramètre si la clé est dans la table, où lève l'exception `Not_found` sinon.

— `Hashtbl.remove` de type `('a, 'b) Hashtbl.t -> 'a -> unit` supprime l'élément de clé passée en paramètre (le premier s'il y en a plusieurs), ne fait rien s'il n'y en a pas.

— `Hashtbl.replace` de type `('a, 'b) Hashtbl.t -> 'a -> 'b -> unit`. `Hashtbl.replace h x y` remplace l'élément courant (le premier s'il y en a plusieurs) associé à la clé x par y. Similaire à `remove` suivi de `add`.

18.5 Files de priorité

Curieusement, la bibliothèque standard de Ocaml ne propose pas d'implémentation de file de priorité. En voici une (file de priorité min !), écrite par mes soins, faisant usage d'un tableau redimensionnable (également implémenté), utilisé comme un tas (min), et d'une table de hachage pour associer à chaque élément présent dans la file de priorité sa position dans le tas, ce qui permet de modifier sa priorité (utile pour l'algorithme de Dijkstra, par exemple !).

```
module type RESIZABLEARRAY_SIG =
    sig
      type 'a t
      val create : unit -> 'a t
      val is_empty : 'a t -> bool
      val length : 'a t -> int
      val get : 'a t -> int -> 'a
      val set : 'a t -> int -> 'a -> unit
      val add : 'a t -> 'a -> unit
      exception ResizableArray_is_empty
      val pop : 'a t -> 'a
    end;;

module ResizableArray = (
  (* une structure de tableau redimensionnable *)
  struct
    type 'a t = {mutable nb: int ; mutable cp: int ; mutable elem: 'a array}
    let create () = {nb = 0; cp = 0; elem = [| |]}
    let is_empty a = a.nb = 0
    let length a = a.nb
    let get a i = a.elem.(i)
```

```
    let set a i x = a.elem.(i) <- x
    let add a x = match a.nb = a.cp with
      | true -> let e2=Array.make (2*a.cp + 1) x in
                for i=0 to a.nb - 1 do
                  e2.(i) <- a.elem.(i)
                done ;
                a.nb <- a.nb + 1 ;
                a.cp <- 2*a.cp + 1 ;
                a.elem <- e2
      | false -> a.elem.(a.nb) <- x ; a.nb <- a.nb + 1
    exception ResizableArray_is_empty
    let pop a = match a.nb with
      | 0 -> raise ResizableArray_is_empty
      | _ -> a.nb <- a.nb - 1 ; a.elem.(a.nb)
  end : RESIZABLEARRAY_SIG)
;;

module type PRIOQUEUE_SIG =
    sig
      type 'a t
      val create : unit -> 'a t
      val is_empty : 'a t -> bool
      exception PrioQueue_is_empty
      exception AlreadyMemberInPrioQueue
      exception NotMemberInPrioQueue
      val add : 'a t -> 'a -> int -> unit
      val remove_prio : 'a t -> 'a
      val change_prio : 'a t -> 'a -> int -> unit
      val mem : 'a t -> 'a -> bool
      end;;

module PrioQueue = (
  (* file de priorite min *)
  struct
    type 'a t = {tr: (int * 'a) ResizableArray.t ; pos: ('a, int) Hashtbl.t}
    let create () = {tr = ResizableArray.create () ; pos = Hashtbl.create 0}
    let is_empty f=ResizableArray.is_empty f.tr
    let fg i = 2*i+1 and fd i = 2*i+2 and pere i = (i-1)/2
    let echanger t i j h=
      (* t tableau f.tr, h la table de hachage f.pos *)
      (* echange les elements d'indice i et j de t et met a jour les positions dans la table de hachage *)
      let a=ResizableArray.get t i in ResizableArray.set t i (ResizableArray.get t j) ;
                      ResizableArray.set t j a ;
      Hashtbl.replace h (snd (ResizableArray.get t i)) i ;
      Hashtbl.replace h (snd (ResizableArray.get t j)) j
    let monter t i h =
      (* t: le tableau redimensionnable f.tr, h la table de hachage *)
      (* les elements sont compares suivant l'ordre lexicographique, donc sur la priorite *)
      let j=ref i in
      while !j>0 && ResizableArray.get t !j < ResizableArray.get t (pere !j) do
        echanger t !j (pere !j) h;
        j:=pere !j
      done
    let rec descendre t i h =
      let n=ResizableArray.length t in
      (* t: le tableau elem dans le tableau redimensionnable f.tr, h la table de hachage *)
      (* n le nombre d'elements dans la fp (donc interessant dans t) *)
      let imax=ref i in
      if fg i < n && ResizableArray.get t (fg i) < ResizableArray.get t i then imax := fg i ;
      if fd i < n && ResizableArray.get t (fd i) < ResizableArray.get t !imax then imax :=fd i ;
      if !imax <> i then begin
        echanger t i !imax h;
        descendre t !imax h
      end
    exception PrioQueue_is_empty
    exception AlreadyMemberInPrioQueue
    exception NotMemberInPrioQueue
    let add f x p =
      (* ajoute avec priorite p l'element x. x ne doit pas etre deja present *)
      if Hashtbl.mem f.pos x then raise AlreadyMemberInPrioQueue
      else begin
        ResizableArray.add f.tr (p,x) ;
        Hashtbl.add f.pos x (ResizableArray.length f.tr - 1) ;
        monter f.tr (ResizableArray.length f.tr - 1) f.pos
```

```
      end
    let remove_prio f =
      if is_empty f then raise PrioQueue_is_empty else
        begin
          let n=ResizableArray.length f.tr in
          echanger f.tr 0 (n-1) f.pos ;
          let _,x=ResizableArray.pop f.tr in
          Hashtbl.remove f.pos x ;
          if n>1 then descendre f.tr 0 f.pos ;
          x
        end
    let change_prio f x p=
      (* on modifie la priorite de l'élément x (en general on baisse l'entier, c'est à dire que *)
      (* l'élément doit remonter dans le tableau, mais l'inverse est possible) *)
      if not (Hashtbl.mem f.pos x) then raise NotMemberInPrioQueue ;
      let i=Hashtbl.find f.pos x in
      let pprev = fst (ResizableArray.get f.tr i) in
      ResizableArray.set f.tr i (p,x) ;
      if pprev > p then monter f.tr i f.pos else descendre f.tr i f.pos
    let mem f x=Hashtbl.mem f.pos x
  end : PRIOQUEUE_SIG)
;;
```

Les fonctions de file de priorité disponibles sont les suivantes (précisées dans le module *signature*). Les priorités sont des entiers dans cette implémentation.

- `create : unit -> 'a PrioQueue.t`
- `is_empty : 'a PrioQueue.t -> bool`
- `add : 'a PrioQueue.t -> 'a -> int -> unit`
- `remove_prio : 'a PrioQueue.t -> 'a`
- `change_prio : 'a PrioQueue.t -> 'a -> int -> unit`
- `mem : 'a PrioQueue.t -> 'a -> bool`

On a aussi défini les exceptions suivantes :

- `exception PrioQueue_is_empty`
- `exception AlreadyMemberInPrioQueue`
- `exception NotMemberInPrioQueue`

18.6 Créer un module en Ocaml

On explique ici brièvement comment créer un module en Ocaml, essentiellement on va expliquer le module précédent. La déclaration d'un module se fait de la façon suivante :

```
module nom_du_module = struct ... end ;;
```

Entre `struct` et `end` se trouve les déclarations de types et de fonctions. On appelera ces fonctions en dehors du module par `nom_du_module.nom_de_la_fonction`. *A priori*, tous les types et fonctions définis dans le module seront alors accessibles en dehors du module. Or, lorsqu'on implémente une structure abstraite par exemple, on préfère masquer à l'utilisateur le mécanisme interne des fonctions, et même le type manipulé. C'est là qu'interviennent les modules de signature :

```
module type nom_du_module_de_signature =  sig ... end ;;
```

Pour les deux modules implémentés plus haut, vous voyez que certaines fonctions ont été choisies pour figurer dans le module de signature, mais pas toutes. De même le type n'est pas décrit explicitement. Par exemple pour la file de priorité (implémentée comme un couple tableau redimensionnable, table de hachage), cette description ne figure pas dans la signature. Pour préciser les signatures révélées, il suffit de déclarer le module comme on l'a fait :

```
module nom_du_module = (struct ... end : nom_du_module_de_signature) ;;
```

Voici un exemple :

```
# let f = PrioQueue.create () ;;
val f : '_a PrioQueue.t = <abstr>
# PrioQueue.add f 81 10 ;; (* ajout de 81 avec priorite 10 *)
- : unit = ()
# f ;;
- : int PrioQueue.t = <abstr>
# f.tr ;;
Error: Unbound record field tr
```

Le type `PrioQueue.t` est considéré comme abstrait, en particulier les champs `tr` et `pos` utilisés dans sa définition ne sont pas accessibles, car n'apparaissent pas dans le module de signature.