2022–2023

Informatique

MPI*

Hugo Salou

TABLE DES MATIÈRES

Table des matières i
Table des figures viii
Liste des tables xii
Liste des algorithmes xiv
Liste des codes xvii

I Cours **1**

−1 Ordres et induction **3**
−1.1 Motivation 3
−1.2 Ordre 4
−1.2.1 Ordre produit 6
−1.2.2 Ordre lexicographique 8
−1.3 Induction nommée 11
−1.4 Synthèse du chapitre 17

0 Logique **19**
0.1 Motivation 19
0.2 Syntaxe 20
0.3 Sémantique 23
0.3.1 Algèbre de Boole 23
0.3.2 Fonctions booléennes 24
0.3.3 Interprétation d'une formule comme une fonction booléenne 25
0.3.4 Liens sémantiques 26
0.4 Le problème Sat – Le problème Validité 27
0.4.1 Résolution par tables de vérité 27
0.5 Représentation des fonction booléennes 28
0.5.1 Par des formules ? 28
0.5.2 Par des formules sous formes normales ? 30
0.6 Algorithme de Quine 32
0.7 Synthèse du chapitre 37

1 Langages réguliers et Automates **39**
1.1 Motivation 40
1.1.1 $1^{\underline{\text{ère}}}$ motivation 40
1.1.2 $2^{\underline{\text{nde}}}$ motivation 40

1.2 Mots et langages, rappels . . . 41
1.3 Langage régulier . . . 42
1.3.1 Opérations sur les langages . . . 42
1.3.2 Expressions régulières . . . 44
1.4 Automates finis (sans ε-transitions) . . . 46
1.4.1 Définitions . . . 46
1.4.2 Transformations en automates équivalents . . . 49
1.5 Automates finis avec ε-transitions . . . 52
1.5.1 Cloture par concaténation . . . 54
1.5.2 Cloture par étoile . . . 55
1.5.3 Cloture par union . . . 55
1.6 Théorème de Kleene . . . 57
1.6.1 Langages locaux . . . 58
1.6.2 Expressions régulières linéaires . . . 62
1.6.3 Automates locaux . . . 64
1.6.4 Algorithme de Berry-Sethi : les langages réguliers sont reconnaissables 67
1.6.5 Les langages reconnaissables sont réguliers . . . 68
1.7 La classe des langages réguliers . . . 72
1.7.1 Limite de la classe/Lemme de l'étoile . . . 73
Annexe 1.A Comment prouver la correction d'un programme ? . . . 75
Annexe 1.B Hors-programme . . . 76

2 Algorithmes probabilistes **79**
2.1 Introduction . . . 79
2.2 Algorithme de Monte-Carlo . . . 83
2.3 Algorithme de type Las-Vegas . . . 84
Annexe 2.A Hors-programme . . . 89

3 Apprentissage **91**
3.1 Motivation . . . 91
3.2 Vocabulaire . . . 91
3.3 Apprentissage supervisé . . . 92
3.3.1 k plus proches voisins . . . 93
3.3.2 Arbres k-dimensionnels . . . 94
3.3.3 Algorithme id3 . . . 95
3.4 Apprentissage non supervisé . . . 101
3.4.1 Algorithme hac, classification hiérarchique ascendant . . . 101
3.4.2 k-moyenne . . . 102

4 Calculabilité, Décidabilité, Complexité **105**
4.1 Remarques mathématiques . . . 105
4.2 Problèmes . . . 106
4.3 Décidabilité . . . 107
4.3.1 Modèles de calcul . . . 108
4.3.2 Décidabilité . . . 109
4.3.3 Langages et problèmes de décision . . . 110
4.3.4 Sérialisation . . . 111
4.3.5 Machine universelle . . . 112
4.3.6 Théorème de l'Arrêt . . . 113
4.3.7 Réduction . . . 114
4.4 Classe **P** et **NP** . . . 115
4.4.1 Complexité d'une machine . . . 115
4.4.2 Classe **P** . . . 116
4.4.3 Classe **NP** . . . 117
4.4.4 **NP**-difficile . . . 118

5 Trois exemples d'algorithmes de graphes **123**
5.1 Composantes fortement connexes (cfc) . . . 123
5.1.1 Rappels . . . 125
5.1.2 Rangement particulier de graphe . . . 127
5.1.3 Graphe transposé et cfc . . . 128
5.1.4 Calcul de tri préfixe . . . 130

5.1.5 Algorithme de Kosaraju . . . 131
5.1.6 Applications . . . 131
5.2 Arbres couvrants de poids minimum . . . 132
5.3 Couplage dans un graphe biparti . . . 137
Annexe 5.A Remarques supplémentaires . . . 140

6 Preuves **141**
6.0 Motivation . . . 141
6.0.1 Tables de vérité . . . 141
6.0.2 Équations . . . 142
6.0.3 Raisonnement mathématiques . . . 142
6.1 La déduction naturelle en logique propositionnelle . . . 142
6.1.1 Séquents . . . 142
6.1.2 Preuves . . . 143
6.1.3 Déduction naturelle . . . 145
6.2 La logique du premier ordre . . . 148
6.2.1 Syntaxe de la logique du premier ordre . . . 148
6.2.2 Substitution . . . 151
6.2.3 Extension au premier ordre de la déduction naturelle . . . 153
6.2.4 Règles dérivées . . . 155
6.2.5 Sémantique . . . 155
6.3 Synthèse du chapitre . . . 159

7 Tentative de réponse à la NP-complétude **161**
7.0 Motivation . . . 161
7.1 Problèmes d'optimisation . . . 161
7.2 Algorithmes d'approximations . . . 163
7.3 *Branch and Bound* — Séparation et évaluation . . . 167
Annexe 7.A Programmation dynamique . . . 169

8 Jeux **171**
8.1 Jeux sur un graphe . . . 171
8.2 Résolution par heuristique . . . 177

9 Grammaires non contextuelles **181**
9.1 Définition, vocabulaire, propriétés . . . 182
9.1.1 Grammaires non contextuelles . . . 182
9.1.2 Dérivation . . . 183
9.1.3 Preuves par induction . . . 184
9.1.4 Définitions équivalentes . . . 185
9.2 La hiérarchie de Chomsky . . . 188
9.2.1 Avec les langages réguliers . . . 188
9.2.2 Lien avec les langages décidables . . . 190

10 Concurrence **193**
10.1 Motivation . . . 193
10.2 *Mutex* . . . 196
10.2.1 À deux fils d'exécutions . . . 196
10.2.2 À N fils d'exécutions . . . 197
10.3 Sémaphore . . . 198

II Travaux Dirigés **201**

TD 1 Ordre & Induction **203**
TD 1.1 Listes, listes, listes ! . . . 203
TD 1.2 Ensembles définis inductivement . . . 204
TD 1.3 Arbres, Arbres, Arbres ! . . . 204
TD 1.4 Ordre sur *powerset* . . . 205
TD 1.5 Ordres bien fondés en vrac . . . 206
TD 1.6 Définition inductive des mots et ordre préfixe . . . 206
TD 1.7 $\mathcal{N}$ . . . 206

TD 1.8 Résultats manquants du cours 207

TD 2 Logique propositionnelle **209**
TD 2.1 Logique avec `If` 209
TD 2.1.1 Représentatibilité des fonctions booléennes par formules de $\mathcal{F}_{\texttt{if}}$ 209
TD 2.2 Définitions de cours : syntaxe 210
TD 2.3 Formules duales 212
TD 2.4 Conséquence sémantique 212
TD 2.5 Axiomatisation algèbre de Boole 213
TD 2.6 Exercice 6 : Barre de Scheffer 213
TD 2.7 Énigmes en logique propositionnelle 213
TD 2.7.1 Fraternité 213
TD 2.7.2 Alice au pays des merveilles 214
TD 2.7.3 Socrate et Cerbère 214
TD 2.8 Compléments de cours, en vrac 214

TD 3 Langages et expressions régulières **217**
TD 3.1 Propriétés sur les mots 217
TD 3.2 Construction d'automates 218
TD 3.3 Déterminisation 1 219
TD 3.4 Déterminisation 2 220
TD 3.5 Une équivalence sur les mots 220
TD 3.6 Langages 220
TD 3.7 Propriétés sur les opérations régulières 220
TD 3.8 Habitants d'expressions régulières 222
TD 3.9 Regexp Crossword 222
TD 3.10 Description d'automates au moyen d'expression régulières 222
TD 3.11 Vocabulaire des automates 222
TD 3.12 Complétion d'automate 223
TD 3.13 Exercice supplémentaire 1 223

TD 4 Langages et expressions régulières (2) **225**
TD 4.1 Déterminisation de taille exponentielle 225
TD 4.2 Suppression des ε-transitions 226
TD 4.3 Déterminisation d'automates avec ε-transitions 226
TD 4.4 Automates pour le calcul de modulo 227
TD 4.5 Automates pour le calcul de l'addition en binaire 227
TD 4.5.1 Nombres de même tailles 227

TD 5 Langages et expressions régulières (3) **229**
TD 5.1 Exercice 4 229
TD 5.2 Exercice 5 230
TD 5.3 Exercice 6 : Langages reconnaissables ou non 230

TD 6 Algorithmes probabilistes **231**
TD 6.1 Exercice 1 : Vérification d'égalité polynomiale 231
TD 6.2 Test de primalité probabiliste 232
TD 6.2.1 Résultats mathématiques 232
TD 6.2.2 Algorithme 232
TD 6.2.3 Implémentation 232
TD 6.3 Exercice 3 : Échantillonnage 233

TD 7 Décidabilité, Calculabilité **235**
TD 7.1 Quelques problèmes décidables 235
TD 7.2 Sérialisation de types énumérés 236
TD 7.3 Stabilité de la classe des langages décidables 236
TD 7.4 Non monotonie du caractère décidable des langages 237
TD 7.5 Réduction 237
TD 7.6 Amélioration de code 239
TD 7.7 Théorème de Rice 240
TD 7.8 Un changement de modèle de calcul 240

TD 8 Classe P, classe NP **243**
TD 8.1 Problèmes de partitions . . . 243
TD 8.2 Optimisation linéaire en nombres entiers . . . 244
TD 8.3 Clique, Stable et Couv.Sommets . . . 244
TD 8.4 . . . 245

TD 9 Algorithmique des graphes **247**
TD 9.1 Arbres et forêts . . . 247
TD 9.2 Tri topologique . . . 247
TD 9.3 Détection de graphe biparti . . . 248
TD 9.4 Exploration du graphe $\mathfrak{S}_n$ . . . 248
TD 9.5 Parcours selon le miroir d'un tri préfixe . . . 248

TD 10Preuves en logique propositionnelle **249**
TD 10.1Premiers arbres de preuves . . . 250
TD 10.2Divers arbres de preuves . . . 250
TD 10.3Lois de de Morgan . . . 251
TD 10.4Distributivités entre $\wedge$ et $\vee$ . . . 251
TD 10.5Implications . . . 252
TD 10.6Implications (partie 1 : simplifications) . . . 253
TD 10.7Implications (partie 2 : transformations) . . . 254

TD 11Preuves **257**
TD 11.1Formalisation . . . 257
TD 11.2Variables libres et liées, clôture universelle . . . 257
TD 11.3Substitution . . . 259
TD 11.4Quelques arbres de preuves . . . 260
TD 11.5Distributivité des quantificateurs . . . 260
TD 11.6Semi-distributivité des quantificateurs . . . 261
TD 11.7Logique classique du premier ordre . . . 261

TD 12Algorithmes d'approximation **263**
TD 12.1Un problème proche de Knapsack . . . 263
TD 12.2Le problème BinPacking . . . 264
TD 12.3Le problème VoyageurCommerce . . . 265

TD 13Jeux **267**
TD 13.1Attracteurs pour le jeu de la soustraction généralisé . . . 267
TD 13.2Chomp . . . 268
TD 13.3Hex . . . 268
TD 13.4Calcul de MinMax . . . 268
TD 13.5Calcul de MinMax avec mémoïsation . . . 268

TD 14Grammaires non contextuelles (1) **269**
TD 14.1Exemples de dérivations . . . 269
TD 14.2Arbres de dérivations . . . 269
TD 14.3Construction de grammaires . . . 270
TD 14.4Raisonner par induction sur une grammaire . . . 271
TD 14.5Ambigüité . . . 271
TD 14.6Langage de Dyck . . . 271
TD 14.7Listes OCaml . . . 271
TD 14.8Mots de Łukasiewicz . . . 272

TD 15Grammaires non contextuelles (2) **273**
TD 15.1Forme normale conjonctive . . . 273
TD 15.2Réduction de grammaire et systèmes de conséquences . . . 274
TD 15.2.1 Digression OCaml . . . 274
TD 15.3Grammaires propres . . . 274
TD 15.4Un lemme d'itération . . . 274
TD 15.5Les langages réguliers sont non contextuels . . . 274
TD 15.5.1 Avec des automates . . . 274
TD 15.5.2 Avec des expressions régulières . . . 275

TD 16 Concurrence **277**
TD 16.1 Entrelacements . . . 277
TD 16.2 Généralisation de l'algorithme de Peterson à N fils . . . 277
TD 16.3 Parallélisation pour le produit de deux matrices . . . 278
TD 16.4 Calcul du maximum par "diviser pour régner" . . . 278
TD 16.5 Un très mauvais algorithme de tri . . . 278

TD 17 Concurrence **279**
TD 17.1 Addition binaire en parallèle . . . 279
TD 17.2 Producteurs–consommateurs en mémoire non bornée . . . 279
TD 17.2.1 Version allégée de la solution vue en cours . . . 279
TD 17.3 Rendez-vous à l'aide de sémaphores . . . 280
TD 17.3.1 Deux fils d'exécutions se rencontrent une fois . . . 280
TD 17.3.2 Plusieurs fils se rencontrent une fois . . . 280
TD 17.3.3 Plusieurs fils d'exécution se rencontrent plusieurs fois . . . 281

TD BONUS 1 Complexité amortie **283**
TD BONUS 1.1 Complexité amortie . . . 283
TD BONUS 1.2 Incrementation compteur binaire . . . 284
TD BONUS 1.3 Tableaux dynamiques . . . 284
TD BONUS 1.4 Tableaux dynamiques, 2 . . . 284

TD BONUS 2 Diviser pour régner **285**
TD BONUS 2.1 Suites récurrentes de complexité . . . 285
TD BONUS 2.2 Multiplication d'entiers par algorithme de KARATSUBA . . . 286

TD BONUS 3 Invariants plus complexes **287**

III Travaux Pratiques **289**

TP 1 Logique propositionnelle **291**
TP 1.1 Syntaxe et sémantique . . . 291
TP 1.1.1 Syntaxe . . . 291
TP 1.1.2 Sémantique . . . 292
TP 1.2 Construction d'une formule à partir d'une fonction . . . 294
TP 1.3 Application de règles de réécriture . . . 294

TP 3 Langages et expressions régulières (2) **295**

TP 7 Algorithme de Kosaraju en OCAML **297**
TP 7.1 Gestion du graphe et du graphe transposé . . . 297
TP 7.2 Gestion des points de régénération dun parcours . . . 297

IV Annexes **299**

Annexe A Complexité amortie **301**

Annexe B Algorithmes DIJKSTRA et $A^\star$ **307**

Annexe C Diviser pour régner **311**

Annexe D Lemme d'ARDEN et retour sur le théorème de KLEENE **313**

Annexe E Tas et files de priorités **315**

Annexe F Arithmétique **317**

Annexe G Arbres rouges-noirs **319**

Annexe H Complexité moyenne **321**

Annexe I Preuves de correction pour les fonctions récursives **323**

Index 325

TABLE DES FIGURES

−1.1 État des variables `b` et `c` 4
−1.2 Diagramme de Hasse 5
−1.3 Contre exemple : $(A^{\mathbb{N}}, \preccurlyeq_{\times})$ est il bien fondé ? 7
−1.4 Ordre lexicographique sur $\mathbb{N}^2$ 9
−1.5 Contre exemple : $(A^{\mathbb{N}}, \preccurlyeq_{\ell})$ est il bien fondé ? 10
−1.6 Contre exemple : $\big((A^\star)^{\mathbb{N}}, \preccurlyeq_{\ell}\big)$ est il bien fondé ? (2) 11
−1.7 Structure des ensembles $X_1, \ldots, X_n$ 14
−1.8 Ensemble obtenu avec les règles S et 0 15
−1.9 Ensemble obtenu avec les règles :: et [ ] 15

0.1 Grille de Sudoku 2×2 19
0.2 Arbre syntaxique d'une expression logique 21

1.1 États d'un ordinateur 40
1.2 Exemple d'automate 46
1.3 Exemple d'automate (2) 47
1.4 Automates minimaux pour différentes valeurs de $\mathscr{L}(\mathscr{A})$ 48
1.5 Automate non déterministe ayant pour expression régulière $a^\star \cdot (a \mid bab)$ 49
1.6 Nœuds possibles par rapport à l'expression lue 49
1.7 Automate déterministe ayant pour expression régulière $a^\star \cdot (a \mid bab)$ 50
1.8 Automate non déterministe 51
1.9 Non-exemples d'états accessibles et co-accessibles 52
1.10 Exemple d'automate avec ε-transition 52
1.11 Automate reconnaissant le langage $(ba)^\star \cdot \big(c \mid a(ba)^\star\big)$ 53
1.12 Automate reconnaissant le langage $(a \mid baa)(bbaa)^\star \mid (b \mid abb)(aabb)^\star$ 53
1.13 Automate reconnaissant la concaténation des deux précédents 54
1.14 Automate reconnaissant $\mathscr{L}(\mathscr{A})^\star$ 55
1.15 Automate reconnaissant $\{a\}$ avec $a \in \Sigma$ 56
1.16 Automate reconnaissant $\varnothing$ 56
1.17 Automate avec ε-transition 56
1.18 Automate sans ε-transition 57
1.19 Automate local reconnaissant le langage $(ab)^\star$ 65
1.20 Automate local reconnaissant $(ab)^\star \mid c^\star$ 66
1.21 Automate déduit de la table 1.4 67
1.22 Application de φ à l'automate de la figure 1.21 68
1.23 Succession d'états 68

1.24 Automate exemple . . . 70
1.25 Application de l'algorithme à un exemple . . . 71
1.26 Automate résultat de l'application du lemme . . . 72
1.27 Ensembles de langages . . . 72
1.28 Automate reconnaissant le langage $\{w \in \Sigma^\star \mid |w|_a \equiv |w|_b \ [3]\}$ . . . 74
1.29 Codage d'un automate par une chaîne de caractères . . . 74
1.30 Automate reconnaissant les mots valides . . . 75
1.31 Automate reconnaissant $\mu^{-1}(P) = L$ . . . 77

2.1 Algorithme de Monte-Carlo pour approximer π . . . 80
2.2 Arbre des appels récursifs de "TriRapide" avec le pivot à gauche . . . 85
2.3 Arbre des appels récursifs de "TriRapide" avec le pivot à la médiane . . . 86

3.1 Représentation de l'algorithme des k plus proches voisins . . . 93
3.2 Représentation de la "dichotomie" en dimension 2 . . . 94
3.3 Arbre 2-dimensionnel représentant la "dichotomie" précédente . . . 94
3.4 Représentation de *bordures* entre les différentes classes . . . 96
3.5 Arbre de décision pour la classification . . . 96
3.6 Représentation graphique de $\mathrm{H}(X)$ en fonction de p . . . 97
3.7 Arbre de décision possible se basant sur le moteur . . . 98
3.8 Arbre de décision possible se basant sur les rails . . . 98
3.9 Arbre de décision possible se basant sur sous-terrain . . . 99
3.10 Arbre de décision partiel . . . 99
3.11 Arbre de décision possible se basant sur le moteur puis la vitesse . . . 99
3.12 Arbre de décision possible se basant sur le moteur puis sous-terrain . . . 99
3.13 Arbre de décision possible se basant sur le moteur puis les rails . . . 99
3.14 Arbre de décision final pour la classification de trains . . . 100
3.15 Représentation graphique du problème de *sur-apprentissage* . . . 100
3.16 Arbre de décision pour la table de données précédente . . . 101

4.1 Structure d'un sous-problème . . . 114
4.2 Structure d'un sous-problème . . . 118

5.1 Graphe non fortement connexe . . . 124
5.2 Exemple de graphe orienté – parcours en largeur . . . 127
5.3 Tri topologique d'un graphe . . . 128
5.4 Exemple de tri préfixe . . . 129
5.5 Représentation d'une formule 2-cnf-sat par un graphe . . . 131
5.6 Arbre pondéré . . . 133
5.7 Représentation par des arbres . . . 136
5.8 Exemple de couplage . . . 137
5.9 Chaîne augmentante . . . 138

6.1 Arbre syntaxique de la formule $G = \forall x,\ \Big(((x > 0) \wedge (\exists y,\ x = y + 1)) \vee (x = 0)\Big)$ 148
6.2 Arbre de syntaxe de la formule $P(x, y) \wedge \big(\forall x,\ Q(x, y)\big)$, application de la substitution $\sigma = (x \mapsto x + y,\ y \mapsto 0)$ . . . 152
6.3 Arbre de syntaxe de la formule $P(x, y) \wedge \big(\forall x,\ Q(x, y)\big)$, application de la substitution $\sigma = (y \mapsto x + x)$ directement et avec α-renommage . . . 153
6.4 Arbre de syntaxe exemple . . . 156

7.1 Entrée du problème du plus court chemin . . . 162
7.2 Algorithme d'approximation pour un problème de maximisation . . . 164
7.3 Algorithme d'approximation pour un problème de minimisation . . . 164
7.4 ρ-approximation différentielle . . . 164
7.5 Calcul de $\mathrm{OPT}(e)$ . . . 164
7.6 Contre-exemple à l'algorithme 7.2 . . . 165
7.7 Exemple de couverture par sommets . . . 166
7.8 Stratégie *branch and bound* appliquée au problème Knapsack . . . 168

8.1 États du jeu des allumettes . . . 172
8.2 Stratégie gagnante pour Bob . . . 174

8.3 Stratégie de minimisation pour Bob, et de maximisation pour Alice 175

9.1 Exemple d'arbres de dérivation 186
9.2 Arbres de dérivations de « $1 - 1 + 9$ » 188

TD 3.1 Automate décrit dans l'énoncé de l'exercice 8 223
TD 3.2 Automate complet équivalent à $\mathcal{A}$ 223

TD 8.1 Représentation du graphe G pour l'entrée $\{x \vee x \vee y, \neg x \vee \neg y \vee \neg y, x \vee y \vee y\}$. 245

TD 9.1 Exemple de graphe 247

TD 11.1 Arbre de syntaxe de la formule $p\big(f(x,y)\big) \vee \forall z,\ r(z,z)$ 258
TD 11.2 Arbre de syntaxe de la formule $\forall x,\ \exists y,\ \big(r(x,y) \to \forall z,\ q(x,y,z)\big)$ 258
TD 11.3 Arbre de syntaxe de la formule $\forall x,\ (x,y,z) \wedge \forall z,\ \big(q(z,y,x) \to r(z,z)\big)$ 258
TD 11.4 Calcul de la substitution $F\big[x \mapsto f(y,z)\big]$, pour la formule 1 259
TD 11.5 Calcul de la substitution $F\big[x \mapsto f(y,z)\big]$, pour la formule 2 259
TD 11.6 Calcul de la substitution $F\big[x \mapsto f(y,z)\big]$, pour la formule 3 260

TD 12.1 Contre exemple 265
TD 12.2 Arbre couvrant de poids minimal 266

TD 13.1 Arbre remplit avec l'algorithme MinMax 268

TD 14.1 Arbre de dérivation de $ab + bc$ dans la grammaire $\mathcal{G}$ 270
TD 14.2 270

Annexe C.1 311

Annexe D.1 Automate exemple ($\mathcal{A}$) 313

LISTE DES TABLEAUX

−1.1 Exemples et non-exemples d'ordres bien fondés 5

0.1 Opération · sur les booléens 24
0.2 Opération + sur les booléens 24
0.3 Opération $\bar{\square}$ sur les booléens 24
0.4 Règles dans $\mathbb{B}$ 24
0.5 Table de vérité de $(a \wedge b) \rightarrow (\neg b \vee \neg c)$ 27
0.6 Table de vérité d'une formule inconnue 28
0.7 Table de vérité de $p \wedge (\neg q \vee p)$ 31
0.8 Table de vérité d'une formule inconnue (2) 31

1.1 Table de transition de l'automate ci-avant 51
1.2 Exemples et non-exemples d'expressions régulières linéaires 62
1.3 Construction de Λ, P, S et F dans différents cas 63
1.4 Λ, S, P et F pour les différents mots reconnus 67
1.5 Fonction T équivalente à l'automate de la figure 1.24 70

3.1 Matrice de confusion dans le cas d'une classification en $\boldsymbol{V}$ et $\boldsymbol{F}$ 94
3.2 Exemple de données 96
3.3 Test de l'arbre de décision créé 100
3.4 Table de données d'exemple 101

6.1 Table de vérité pour montrer $(p \rightarrow q) \wedge (q \rightarrow r) \models p \rightarrow r$ 142
6.2 Règles d'introduction et d'élimination 146
6.3 Extension au premier ordre de la déduction naturelle 153
6.4 Règles d'introduction et d'élimination 159
6.5 Déduction naturelle classique 159
6.6 Extension au premier ordre de la déduction naturelle 159

7.1 Entrée du problème Knapsack 168

10.1 Problème de l'entreblocage 195

Annexe F.1 Valeurs de r_i avec invariant $r_i = au_i + bv_i$ 318

LISTE DES ALGORITHMES

0.1 Algorithme *Assume* . . . 35
0.2 Algorithme de Quine . . . 36
1.1 Suppression des ε-transitions . . . 57
2.1 Bozosort . . . 80
2.2 Algorithme de Monte-Carlo pour répondre au problème . . . 81
2.3 Algorithme de Las-Vegas pour répondre au problème . . . 81
2.4 Algorithme de Monte-Carlo répondant au problème . . . 83
2.5 Fonction “Partitionner” utilisée dans le tri rapide . . . 84
2.6 Tri rapide . . . 85
3.1 k-NN (k *nearest neighbors*) . . . 93
3.2 “F” : Fabrication d'un arbre k-dimensionnel . . . 95
3.3 “R” : Recherche du point le plus proche . . . 95
3.4 Algorithme hac . . . 102
3.5 k-moyenne . . . 103
5.1 Calcul d'un tri préfixe . . . 130
5.2 Algorithme de Kosaraju . . . 131
5.3 Solution au problème 2cnfsat . . . 132
5.4 Algorithme de Kruskal . . . 134
5.5 Algorithme de Kruskal – version 2 . . . 137
5.6 ChaîneAugmentante : Trouver une chaîne augmentante dans un graphe biparti $G = (S, A)$ muni d'un couplage C partant d'un sommet $s \in S$ . . . 139
5.7 Calcul d'un couplage maximum . . . 139
7.1 Solution à un problème de seuil . . . 163
7.2 Algorithme glouton de recherche de stables . . . 165
7.3 Calcul d'un couplage maximal (CouplageMaximal) . . . 166
7.4 Approximation de couverture par sommets . . . 166
7.5 Algorithme glouton $\mathcal{G}^{\mathbb{N}}$ répondant au problème Knapsack . . . 167
7.6 Algorithme glouton $\mathcal{G}^{\mathbb{R}}$ répondant au problème $\text{Knapsack}_{\mathbb{R}}$ . . . 168
8.1 Algorithme MinMax . . . 178
8.2 Algorithme MinMax avec élagage α, β . . . 179
10.1Tentative 1 d'implémentation du type verrou . . . 196
10.2Tentative 2 d'implémentation du type verrou . . . 196
10.3Tentative 3 d'implémentation du type verrou . . . 197
10.4Tentative 4 d'implémentation du type verrou – Algorithme de Peterson . . . 197
10.5Tentative 1 d'implémentation du type verrou à N fils . . . 198
10.6Tentative 2 d'implémentation du type verrou à N fils . . . 198
10.7Tentative 3 d'implémentation du type verrou à N fils – algorithme de la boulangerie 198

10.8 Utilisation de la structure sémaphore dans une problématique producteur/consommateur . . . 199
TD 6.1 Algorithme déterministe pout tester l'égalité polynomiale en $\mathcal{O}(n^2)$ . . . 231
TD 6.2 Algorithme probabiliste pout tester l'égalité polynomiale en $\mathcal{O}(n)$ . . . 232
TD 6.3 Algorithme Monte-Carlo testant la primalité d'un nombre en $\mathcal{O}(k\,(\ln k)^3$ . . . 232
TD 6.4 Échantillonnage naïf . . . 233
TD 9.1 Génération d'un tri topologique d'un graphe acyclique . . . 248
TD 9.2 Génération d'un tri topologique d'un graphe . . . 248
TD 12.1 Algorithme glouton pour résoudre le problème SommeMax$_O$ en $\mathcal{O}(n)$ . . . 264
TD 12.2 Algorithme glouton pour résoudre le problème SommeMax$_O$ en $\mathcal{O}(n \ln n)$ . . . 264
TD 12.3 Algorithme glouton pour résoudre le problème SommeMax$_O$ en $\mathcal{O}(n)$ . . . 264
TD 12.4 Réduction polynômiale de Partition à BinPacking . . . 265
TD 16.1 Proposition 1 . . . 277
TD 16.2 Proposition 2 . . . 278
TD 17.1 Fil d'exécution P_1 . . . 280
TD 17.2 Fil d'exécution P_2 . . . 280
TD 17.3 Fil d'exécution P_1 . . . 280
TD 17.4 Fil d'exécution P_2 . . . 280
TD 17.5 Fil d'exécution P_3 . . . 280
TD 17.6 Fil P_1 . . . 280
TD 17.7 Fil P'_1 . . . 280
TD 17.8 Fil P_2 . . . 280
TD 17.9 Fil P'_2 . . . 280
TD 17.10 Fil P_3 . . . 280
TD 17.11 Fil P'_3 . . . 280
TD 17.12 Programme principal . . . 280
TD 17.13 Implémentation de la structure barrière $\mathcal{b}$ . . . 281
TD 17.14 Implémentation de la structure barrière robuste $\mathcal{b}$ . . . 281
Annexe A.1 Calcul de $n+1$ avec un tableau de *bit*s . . . 301
Annexe A.2 Agrandit(T), fonction agrandissant le tableau t . . . 303
Annexe A.3 Ajout(T, x), ajout d'un élément dans le tableau . . . 303
Annexe B.1 Algorithme $A^\star$ (partiel) . . . 309
Annexe H.1 Calcul d'inverse d'une permutation . . . 321

LISTE DES CODES

−1.1 Calcul de factorielle . . . 3
−1.2 Un programme mystère (2) . . . 3
−1.3 Inverser une liste . . . 4
−1.4 Un programme mystère (3) . . . 4
−1.5 Calcul du PGCD . . . 7
−1.6 La fonction Ackermann . . . 11
−1.7 Une fonction mystère (5) . . . 11
0.1 Algorithme de Quine *version zéro* . . . 34
1.1 $\sqrt{2}$ sous forme de structure . . . 40
1.2 Règles des expressions régulières en OCamL . . . 44
1.3 Fonction `affiche` affichant un automate . . . 74
4.1 Machine décidant le problème Prime . . . 108
4.2 Généralisation des machines ayant pour entrée un ensemble $\mathscr{E}$ et sortie $\mathscr{F}$ . . . 109
4.3 Machine calculant la fonction $\sqrt{\ }$ . . . 109
4.4 Machine décide le problème Existe_0 . . . 110
4.5 Fonction OCamlreconnaissant l'union de deux langages décidables . . . 111
4.6 Fonction OCaml sérialisant le produit cartésien de deux types sérialisables . . . 112
4.7 Fonction factorielle en OCaml . . . 112
4.8 Programme paradoxe prouvant que le problème Arrêt est indécidable . . . 113
4.9 Fonction décidant un sous-problème . . . 114
4.10Machine calculant la composée en temps polynômial . . . 116
5.1 Implémentation du type UnionFind en OCaml . . . 135
6.1 Définition des formules de premier ordre en OCaml . . . 149
8.1 Représentation machine des jeux, types . . . 174
8.2 Représentation machine du jeux des allumettes . . . 174
8.3 Type `int_bar` représentant un élément l'ensemble $\bar{\mathbb{Z}}$ . . . 177
9.1 Programme reconnaissant une grammaire $\mathscr{G}$ . . . 190
10.1Création de *thread*s en C . . . 193
10.2Mémoire dans les *thread*s en C . . . 194
TD 7.1 Sérialisation de listes . . . 236
TD 7.2 Serialisation de types énumérés . . . 236
TD 7.3 Fonction décidant d'un langage fini . . . 236
TD 7.4 Fonction décidant d'une concaténation de langages décidables . . . 236
TD 7.5 Réduction de ArrêtUniv au problème de l'Arrêt . . . 238
TD 7.6 Machine reconnaissant le langage $\{a^n \cdot b^n \mid n \in \mathbb{N}\}$ . . . 238
TD 7.7 Fonction morte . . . 239
TD 7.8 Réduction de ArrêtUniv à CodeMort . . . 239

TD 7.9 Variable constante . . . 240
TD 7.10 *Super-machine* résolvant le problème de l'ARRÊT sur une machine classique . . . 240
TD 7.11 Programme paradoxe proubant que le problème de l'arrêt des *super-machines* est indécidable . . . 241
TD 14.1 Fonction `est_luka` testant si w est un mot de Łukasiewicz . . . 272
TD 15.1 Expressions OCAML . . . 273
TD 15.2 Parsing des fonctions OCAML . . . 274
TP 1.1 Définition du type `formule` représentant les éléments de $\mathcal{F}$ . . . 291
TP 1.2 Affichage du type `formule` . . . 291
TP 1.3 Ensemble de variables d'une formule de type `formule` . . . 292
TP 1.4 Définition du type `env_prop` représentant un environnement propositionnel . . . 292
TP 1.5 Affichage du type `env_prop` . . . 292
TP 1.6 Interprétation d'une formule . . . 292
TP 1.7 Génération des environnements propositionnels . . . 293
TP 1.8 Résolution du problème SAT . . . 293
TP 1.9 Résolution du problème VALIDE . . . 293
TP 1.10 Vérification de "conséquence sémantique" . . . 293
TP 1.11 Vérification de "équivalence" . . . 293
TP 1.12 Calcul de modèles d'une formule . . . 293
TP 1.13 Calcul de modèles d'une formule . . . 293
TP 1.14 Déterminaison d'une formule dont l'interprétation est `f` sous forme FND . . . 294
TP 1.15 Déterminaison d'une formule dont l'interprétation est `f` sous forme FNC . . . 294
TP 7.1 Type `graphe` . . . 297
TP 7.2 Transposée d'un graphe . . . 297
TP 7.3 Générateur du tableau $[\![0, \mathtt{n}]\!]$ . . . 298
TP 7.4 Générateur d'un tableau `tab` . . . 298
Annexe E.1 Définition du type `btree` . . . 315
Annexe F.1 Algorithme d'Euclide calculant le PGCD . . . 317
Annexe G.1 Arbre binaire de recherche . . . 319

Partie I

Cours

CHAPITRE

−1 ORDRES ET INDUCTION

Sommaire

−1.1 Motivation 3
−1.2 Ordre 4
−1.2.1 Ordre produit 6
−1.2.2 Ordre lexicographique 8
−1.3 Induction nommée 11
−1.4 Synthèse du chapitre 17

−1.1 Motivation

```
let rec fact n =
  if n = 0 then 1
  else n * (fact (n-1));;
```

Code −1.1 – Calcul de factorielle

Ce programme calcule la factorielle d'un nombre ? En développant, l'expression de `fact 3`, on a

$$\begin{aligned}(\texttt{fact } 3) &= 3 \times (\texttt{fact } 2)\\ &= 3 \times (2 \times (1 \times 1))\end{aligned}$$

On en déduit que ce programme calcule la factorielle car ce développement s'arrête à un certain point. Comment en être sûr ? En effet, avec $\texttt{n} = -1$, on obtient une `Stack Overflow Error` ; on n'a plus de mémoire. Pour en être sûr, il faut définir un *invariant*.

À faire : Ajouter 2ème exemple

Un autre exemple :

```
let mystere2 n m =
  let rec aux c b =
    if c = 0 and b = m then 0
    else if c = 0 then aux (b * m) (b + 1)
    else 1 + aux (c - 1) b
```

```
6   in aux n 0;;
```

Code −1.2 – Un programme mystère (2)

Ce programme a beaucoup plus de variables : les variables augmentent dans certains cas, puis diminuent... On peut représenter l'état des variables `b` et `c` dans une figure :

À faire : Figure à faire

Figure −1.1 – État des variables `b` et `c`

On en conclut que ce programme calcule la valeur de

$$\mathtt{n} + \mathtt{m} + 2\mathtt{m} + \cdots + \mathtt{m}^2 = \mathtt{n} + \frac{\mathtt{m}^2 \times (\mathtt{m}-1)}{2}.$$

Cherchons un variant. On peut penser à `b − m` mais il ne diminue pas à chaque étape. Nous verrons quel est ce variant plus tard dans le chapitre.

Nouvel exemple : retourner une liste. Essayons de distinguer les différents cas possibles : si la liste est vide, on la renvoie ; sinon, on extrait un élément `x` et on note le reste de la liste `xs`, on retourne `xs` puis on concatène à droite `x`. On peut donc écrire

```
1  let rec rev l =
2    match l with
3    | [] -> []
4    | x :: xs -> (rev xs) @ [x];;
```

Code −1.3 – Inverser une liste

Cependant, le `@` est une opération lente en OCamL. En effet ce programme a une complexité en $\mathcal{O}(n^2)$, où n est la taille de la liste. Cette complexité est douteuse pour une opération aussi simple. On peut également se demander si cette opération se termine. Cela paraît très simple : la taille de la liste diminue mais nous n'avons pas le coté mathématique d'une liste. En effet, qu'est ce qu'une liste et la taille de cette liste ? On doit formaliser l'explication de pourquoi cet algorithme se termine.

Continuons avec un autre exemple :

```
1  let rec mystere3 m n =
2    if m = 0 then n + 1
3    else if n = 1 then mystere (n - 1) 1
4    else mystere (m - 1) (mystere m (n-1));;
```

Code −1.4 – Un programme mystère (3)

Il s'agit de la fonction Ackermann. Sa complexité est très importante mais ce n'est pas le sujet de cette introduction. En effet, on a

$$A_{0,m} = n + 1$$
$$A_{m,0} = A_{n-1,1}$$
$$A_{m,n} = A_{m-1,A_{m,\dots}}$$

Malgré ce que l'on peut penser, cette fonction se termine mais comment le prouver ?

À faire : Exemple arbres binaires

On en conclut que, avec les outils de l'année passée, il est difficile de prouver que ces algorithmes se terminent rigoureusement.

−1.2 Ordre

Définition (Élements minimaux) **:** Lorsque $(E, \preccurlyeq)$ est un espace ordonné, et $A \subseteq E$ ("inclut ou égal") est une partie de E, on appelle *élément minimal* de A un élément $x \in A$ tel que

$$\forall y \in A,\ y \preccurlyeq x \implies y = x.$$

Exemple :
La figure ci-dessous est un diagramme de Hasse : c'est un diagramme où les points représente les éléments de l'ensemble E et où les segments représentent une comparaison entre les deux éléments connectés : l'élément inférieur est représenté plus bas. Dans l'exemple ci-dessous, les éléments minimaux de A sont les points b et f.

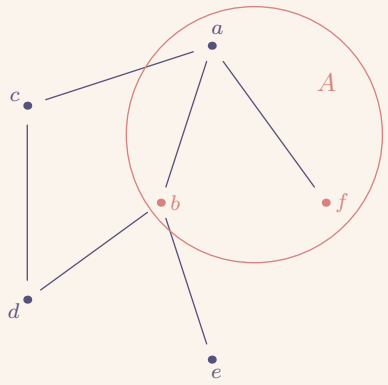


Figure –1.2 – Diagramme de Hasse

Définition (Ordre bien fondé) **:** Un ordre est *bien fondé* s'il n'existe pas de suite infiniment strictement croissante.

Exemple :

Oui	Non
$(\mathbb{N}, \leqslant)$	$(\mathbb{Z}, \leqslant)$
	$(\mathbb{R}, \leqslant)$
	$(\mathbb{R}^+, \leqslant)$ $(1/2^n)$
$(E, \subseteq)$ (si E est fini)	$(E, \subseteq)$ (en général)

Table –1.1 – Exemples et non-exemples d'ordres bien fondés

Propriété : Une relation d'ordre $\preccurlyeq$ sur un ensemble E bien fondé si et seulement si toute partie non vide de E admet un élément minimal.

Preuve " $\Longleftarrow$ " Supposons que toute partie non vide de E admet un élément minimal. Supposons, de plus, qu'il existe une suite $(x_n)_{n\in\mathbb{N}}$ infiniment strictement décroissante. Soit alors $A = \{u_n \mid n \in \mathbb{N}\}$ qui admet un élément minimal ; soit n_0 son indice. Or, $x_{n_0+1} \preccurlyeq x_{n_0}$ ce qui est absurde.

" $\Longrightarrow$ " Supposons que $(E, \preccurlyeq)$ est un ensemble bien fondé. Supposons également qu'il existe un sous-

ensemble A de E non vide n'admettant pas d'élément minimal. Comme A est non vide, on pose alors $x_0 \in A$. Et, comme A n'admet pas d'élément minimal, donc il existe $x \in A$ tel que $x \preccurlyeq x_0$. Notons un tel x, x_1. En itérant ce procédé, on crée la suite $(x_i)_{i\in\mathbb{N}}$ qui est infiniment strictement décroissante ; ce qui est absurde.

□

Théorème (Induction bien fondée) **:** Soit $(E, \preccurlyeq)$ un ensemble ordonné et bien fondé. Soit P une propriété sur les éléments de E. Si $x \in E$, on note $E^{\preccurlyeq x} = \{y \in E \mid y \preccurlyeq x\}$. Si $\forall x \in E,\ \big(\forall y \in E^{\preccurlyeq x},\ P(y)\big) \implies P(x)$, alors $\forall x \in E,\ P(x)$.

Remarque :
Si $(E, \preccurlyeq) = (\mathbb{N}, \leqslant)$, alors le théorème précédent se traduit par :

$$\text{si } \forall n \in \mathbb{N},\ \big(\forall p < n,\ P(p)\big) \implies P(n), \text{ alors } \forall n \in \mathbb{N}, P(n).$$

Ce résultat correspond à la "récurrence forte." Décomposons ce "∀" : on extrait le cas où $n = 0$

$$\text{si } P(0) \text{ et } \forall n \in \mathbb{N}^\star,\ \big(\forall p < n, P(p)\big) \implies P(n), \text{ alors } \forall n \in \mathbb{N}, P(n).$$

On peut donc utiliser le principe de la récurrence pour tout ensemble ordonné bien fondé.

Preuve :
Soit $A = \{x \in E \mid P(x) \text{ n'est pas vrai}\}$.

Cas 1 $A = \varnothing$, alors OK.

Cas 2 $A \neq \varnothing$. Soit alors $x \in A$ un élément minimal de A (c.f. proposition d'avant). On a que $\forall y \in E,\ y \preccurlyeq x,\ P(y)$ est vrai donc $P(x)$ est vrai par hypothèse. Ce qui est absurde.

□

–1.2.1 Ordre produit

Définition : Soit $(A, \preccurlyeq_A)$ et $(B, \preccurlyeq_B)$ deux ensembles ordonnés on définit alors $\preccurlyeq_\times$ sur $A \times B$ par

$$\forall (a,b),(a',b') \in A \times B,\ (a,b) \preccurlyeq_\times (a',b') \overset{\text{def.}}{\iff} (a \preccurlyeq_A a' \text{ et } b \preccurlyeq_B b').$$

Propriété : $\preccurlyeq_\times$ est une relation d'ordre. □

La proposition précédente est facilement vérifiée comme $\preccurlyeq_A$ et $\preccurlyeq_B$ sont, elles aussi, des relations d'ordre.

Remarque (⚠) :
Si $\preccurlyeq_A$ et $\preccurlyeq_B$ sont des ordres totaux, $\preccurlyeq_\times$ ne l'est pas forcément.

Propriété : Soient $(A, \preccurlyeq_A)$ et $(B, \preccurlyeq_B)$ bien fondés, alors $(A \times B, \preccurlyeq_\times)$ l'est aussi.

Preuve :
Supposons alors que $(A\times B, \preccurlyeq_\times)$ ne soit pas bien fondée. Nous avons donc une suite infiniment strictement décroissante

$$(a_0, b_0) \succ_\times (a_1, b_1) \succ_\times (a_2, b_2) \succ_\times \cdots.$$

On a donc $a_0 \succcurlyeq_A a_1 \succcurlyeq_A a_2 \succcurlyeq \cdots$. Or, $(A, \preccurlyeq_A)$ est bien fondée donc il existe $n_0 \in \mathbb{N}$ tel que $\forall i \geqslant n_0, a_i = a_{n_0}$. On a donc $\forall i \geqslant n_0,\ b_i \prec_B b_{i-1}$. Considérons $(b_i)_{i\geqslant n_0}$ est infiniment strictement décroissante dans $(B, \preccurlyeq_B)$, ce qui est absurde. □

Remarque :
On a défini une relation "produit," on peut se demander si ces résultats s'appliquent aussi si la relation est "somme." Ce n'est pas le cas : soit $\preccurlyeq_+$ définie comme

$$(a, b) \preccurlyeq_+ (a', b') \overset{\text{def.}}{\iff} a \preccurlyeq_A a' \text{ ou } b \preccurlyeq_B b'.$$

On peut démontrer que ce n'est pas une relation d'ordre.

Remarque :
Si $(A, \preccurlyeq_A)$ est une relation d'ordre alors $(A^n, \preccurlyeq_\times)$ a les même propriétés.

Remarque :
Sur $A^{\mathbb{N}}$, on définit $(u_n)_{n\in\mathbb{N}} \preccurlyeq_\times (v_n)_{n\in\mathbb{N}}$ si et seulement si

$$\forall i,\ u_i \preccurlyeq_A v_i.$$

L'ensemble ordonné $(A^{\mathbb{N}}, \preccurlyeq_\times)$ est il bien fondé ? La réponse est non.

Voici un contre-exemple : on pose $A = \{0, 1\}$.

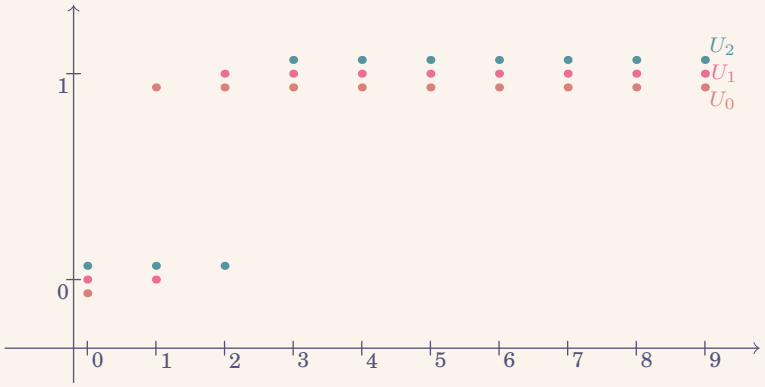


Figure –1.3 – Contre exemple : $(A^{\mathbb{N}}, \preccurlyeq_\times)$ est il bien fondé ?

On considère la suite U_0 qui a pour tout $n \in \mathbb{N}$ la valeur de 1. Puis, on considère la suite U_1 qui a, pour $n = 0$, la valeur de 0 puis pour les autres valeurs de n, la valeur de 1. Ensuite, on considère la suite U_2 qui, pour $n = 0, 1$, la valeur de 0 puis, pour les autres valeurs de n, la valeur de 1. En itérant ce procédé, on crée une suite de suite $(U_n)_{n\in\mathbb{N}}$ infiniment strictement décroissante :

$$U_0 \succcurlyeq_\times U_1 \succcurlyeq_\times U_2 \succcurlyeq_\times \cdots.$$

On considère le programme suivant :

```
let rec pgcd a b =
  if a = b then a
  else if a > b then pgcd (a-b) b
  else pgcd a (b-a);;
```

Code −1.5 – Calcul du PGCD

Étudions ce programme. Ce programme se termine si et seulement si a $= 0$ et b $= 0$ où si a > 0 et b > 0. Prouvons-le rigoureusement. On choisit comme variant (a, b) vivant dans l'ensemble ordonné $(\mathbb{N}^\star \times \mathbb{N}^\star, \preccurlyeq_\times)$ où $\preccurlyeq_\times$ est la relation d'ordre produit. À faire : Recopier une partie du cours ici. On a donc bien une décroissance stricte de la valeur de l'expression (a, b)) valeurs dans un espace bien fondé. D'où terminaison.

Démontrons maintenant la correction, c'est-à-dire, démontrons que

$$\forall (a, b) \in \mathbb{N}^\star \times \mathbb{N}^\star,\ (\texttt{pgcd}\ a\ b) = a \wedge b.$$

Pour cela, on procède par induction sur $\left((\mathbb{N}^\star)^2, \preccurlyeq_\times\right)$ pour démontrer la proposition

$$P(a, b) = (\texttt{pgcd}\ a\ b) = a \wedge b.$$

— Soit $(a, b) = (1, 1)$. On a

$$(\texttt{pgcd}\ a\ b) \underset{\text{(code)}}{=} a = a \wedge b.$$

— Soit $(a, b) \neq (1, 1) \in (\mathbb{N}^\star)^2$ tel que pour tout $(c, d) \in (\mathbb{N}^\star)^2$ tel que $(c, d) \prec_\times (a, b)$ on ait $P(c, d)$. Montrons donc $P(a, b)$.

— Si $a = b$:

$$(\texttt{pgcd}\ a\ b) \underset{\text{(code)}}{=} a = a \wedge b.$$

— Si $a > b$:

$$\begin{aligned}(\texttt{pgcd}\ a\ b) &\underset{\text{(code)}}{=} (\texttt{pgcd}\ (a-b)\ b)\\ &\underset{\text{(hypothèse)}}{=} (a-b) \wedge b\\ &\underset{\text{(maths)}}{=} a \wedge b.\end{aligned}$$

−1.2.2 Ordre lexicographique

Définition : Soit $(A, \preccurlyeq_A)$ et $(B, \preccurlyeq_B)$ deux ensembles ordonnés, on définit alors sur $A \times B$ l'ordre

$$(a, b) \preccurlyeq_\ell (a', b') \overset{\text{def.}}{\iff} (a \prec_A a') \text{ ou } (a = a' \text{ et } b \preccurlyeq_B b').$$

Exemple :
Dans $(\mathbb{N}^2, \preccurlyeq_\times)$, on cherche les éléments $(x, y) \in \mathbb{N}^2$ tels que $(x, y) \preccurlyeq_\ell (3, 4)$:

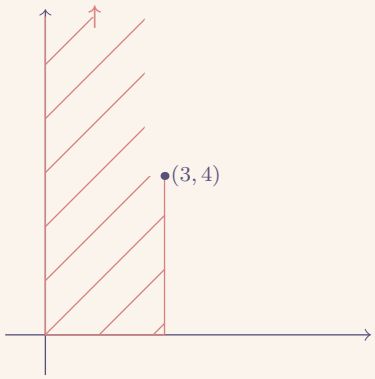


FIGURE –1.4 – Ordre lexicographique sur $\mathbb{N}^2$

Propriété : $\preccurlyeq_\ell$ est une relation d'équivalence.

Preuve :
À faire □

Propriété : Si $\preccurlyeq_A$ est totale et $\preccurlyeq_b$ est totale alors $\preccurlyeq_\ell$ est totale.

Preuve :
Soit (a, b) et $(c, d) \in (A \times B)$.

- Si $a \prec_A c$, alors $(a, b) \prec_\ell (c, d)$.
- Si $a = c$,
  - si $b \preccurlyeq_B d$ alors $(a, b) \preccurlyeq_\ell (c, d)$.
  - sinon $(b \succcurlyeq_B d)$ alors $(a, b) \succ_\ell (c, d)$.
- si $a \succ_A c$ alors $(c, d) \prec_\ell (a, b)$.

□

Propriété : Si $(A, \preccurlyeq_A)$ et $(B, \preccurlyeq_B)$ sont bien fondés, alors $(A \times B, \preccurlyeq_\ell)$ l'est aussi.

Preuve :
À rédiger. □

REMARQUE :
On peut généraliser à un ensemble $(A^n, \preccurlyeq_\ell)$.

Par exemple, avec $n = 3$, on a

$$(a, b, c) \preccurlyeq_\ell (a', b', c') \overset{\text{def.}}{\iff} a \prec_A a' \text{ ou } (a = a' \text{ et } b \prec_B b') \text{ ou } (a = a' \text{ et } b = b' \text{ et } c \prec_C c').$$

Même question qu'avec l'ordre produit, l'ensemble $(A^{\mathbb{N}}, \preccurlyeq_\ell)$ est-il bien fondé? De même, la

réponse est non, la même suite de suite est un contre-exemple.

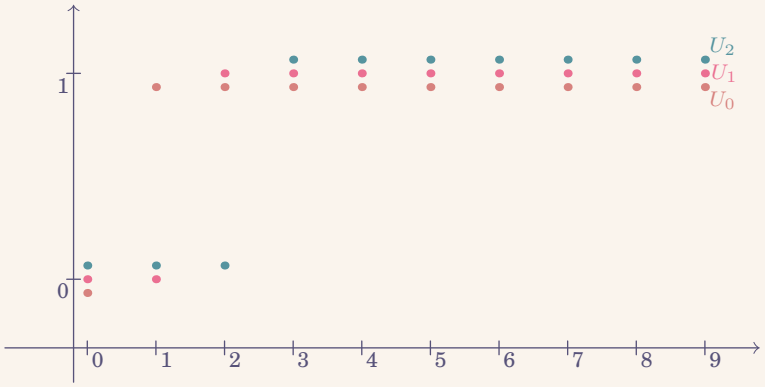


FIGURE −1.5 – Contre exemple : $(A^{\mathbb{N}}, \preccurlyeq_\ell)$ est il bien fondé ?

L'ordre lexicographique est, comme son nom l'indique, l'ordre utilisé dans le dictionnaire. La seule différence est que l'on peut comparer des mots de longueurs différentes.

RAPPEL :
Si A est un ensemble, alors $A^\star = \bigcup_{n\in\mathbb{N}} A^n$. L'ensemble $A^\star$ contient toutes les suites finies d'éléments de A.

Par exemple, avec $A = \{0, 1\}$, on a

$$A^\star \supseteq \{(\,), (1), (0), (0,1), (1,0), (1,1), (0,0), (1,1,0), (0,0,0), \dots\}.$$

Définition : Si $(A, \preccurlyeq_A)$ est un ensemble ordonné, on définit sur $A^\star$:

$$(u_p)_{p\in[\![1,n]\!]} \prec_\ell (v_p)_{p\in[\![1,m]\!]} \overset{\text{def.}}{\iff} \begin{array}{l} \exists i \in [\![1, \min(n,m)+1]\!], \\ \big(\forall j \in [\![1, i-1]\!],\ u_j = v_j\big) \\ \text{et } (i = n+1 \text{ ou } u_i \prec_A v_i). \end{array}$$

Propriété : C'est une relation d'ordre. Elle est totale si $\preccurlyeq_A$ est totale. □

Même question avec cette nouvelle définition de l'ordre lexicographique, l'ensemble $((A^\star)^{\mathbb{N}}, \preccurlyeq_\ell)$ est-il bien fondé ? De même, la réponse est non. Voici un contre-exemple : on considère la suite de suite $U_0 = (1)$, $U_1 = (0,1)$, $U_2 = (0,0,1)$. On crée une suite infiniment strictement décroissante.

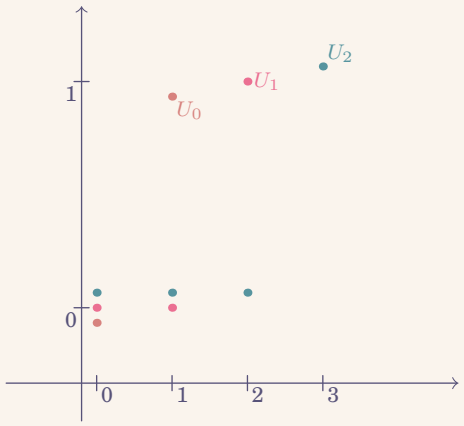


FIGURE –1.6 – Contre exemple : $\left((A^\star)^{\mathbb{N}}, \preccurlyeq_\ell\right)$ est il bien fondé ? (2)

```
let rec mystere3 m n =
  if m = 0 then n + 1
  else if n = 1 then mystere (n - 1) 1
  else mystere (m - 1) (mystere m (n-1));;
```

CODE –1.6 – La fonction ACKERMANN

La fonction ACKERMANN utilise l'ordre lexicographique ; dans ce cas ci, l'ensemble ordonné est bien fondé. C'est comme cela que l'on a la terminaison de cette fonction.

Prenons un autre exemple :

```
let rec mystere n m p =
  if m > 0 then 1 + mystere n (m - 1) p
  else if m = 0 && n > 0 then 1 + mystere (n-1) p (p+1)
  else 0
```

CODE –1.7 – Une fonction mystère (5)

À faire :

— s'assurer de la terminaison (comme celle du PGCD)

— démontrer (par induction) que

$$\forall (m,n,p) \in \mathbb{N}^3,\ (\texttt{mystere}\ n\ m\ p) = \frac{n(n+1)}{2} + pn + m.$$

Les preuves de correction et de terminaison sont basées sur la supposition qu'un entier en OCamL est identique à un entier mathématique.

–1.3 Induction nommée

Définition (Règle de Construction nommée) **:** On appelle *règle de construction nommée* la donnée de

— un symbole S,

— un entier $r \in \mathbb{N}$,

— un ensemble non vide C.

On écrira alors cette règle

$$\text{“ } S\big|_C^r \text{ ”} \qquad \text{ou encore} \qquad \text{“}S(y, \underbrace{\square, \square, \dots, \square}_{r}) \text{ pour } y \in C\text{.”}$$

Remarque :
On a parfois besoin d'un ensemble C trivial (de taille 1 et contenant un objet inutile), on note alors la règle $S\big|^r$.

Exemple :
Les symboles sont écrits en rouge afin de les différencier.

$$0\big|^0 \qquad S\big|^1$$
$$[\,]\big|^0 \qquad ::\big|_{\mathbb{N}}^1$$
$$V\big|^0 \qquad N\big|_{\mathbb{N}}^2$$

Définition : — On appelle *règle de base* une règle de la forme $S\big|_C^0$.
— On appelle *règle d'induction* une règle de la forme $S\big|_C^n$.

Définition : Étant donné un ensemble fini de règles $R = \underbrace{B}_{\text{règle de base}} \cup \overbrace{I}^{\text{règle d'induction}}$ avec $B \neq \varnothing$, on définit alors

$$X_0 = \left\{ (S, a) \;\middle|\; S\big|_C^r \text{ et } a \in C \right\}$$

puis, pour tout $n \in \mathbb{N}$,

$$X_{n+1} = X_n \cup \left\{ (S, a, t_1, t_2, \dots, t_r) \;\middle|\; S\big|_C^r \in \underbrace{R}_{C},\ a \in C,\ t_1 \in X_n,\ t_2 \in X_n, \dots, t_r \in X_n \right\}.$$

On appelle alors $\bigcup_{n\in\mathbb{N}} X_n$ l'ensemble défini par induction nommée à partir des règles de R.

Remarque (Notation) :
On note un n-uplet ayant pour premier élément un symbole S puis $n-1$ éléments $(a_1, \dots, a_{n-1})$. Au lieu de $(S, a_1, \dots, a_{n-1})$, on note $S(a_1, \dots, a_{n-1})$.

Exemple :
On pose

$$R = \left\{ A\big|_{\mathbb{N}}^0, B\big|^1, C_{\{0,1\}}^2 \right\}.$$

À faire : À finir

Remarque :
Pour l'ensemble A défini par induction à partir de $R = \left\{0|^0, S|^1\right\}$, on dira plutôt

"Soit A l'ensemble défini par induction tel que $0 \in A$ et $\forall a \in A,\ S(a) \in A$."

Définition : Soit R un ensemble fini de règles et A l'ensemble défini par induction à partir de ces règles. Sur A, on définit la relation binaire $\diamond$ par

$$x \diamond y \overset{\text{def.}}{\iff} y = S(\dots, x, \dots) \text{ avec } S|_C^r \in R.$$

On définit alors

$$x \preccurlyeq y \overset{\text{def.}}{\iff} \exists p \in \mathbb{N},\ \exists (a_1, \dots, a_p) \in A^p,\ x \diamond a_1 \text{ et } a_1 \diamond a_2 \text{ et } \dots \text{ et } a_p \diamond y \text{ ou } x = y.$$

Définition (hauteur) **:** Soit R un ensemble fini de règles d'induction nommée définissant un ensemble $A = \bigcup_{n\in\mathbb{N}} X_n$. On définit alors

$$\begin{aligned} h : A &\longrightarrow \mathbb{N} \\ x &\longmapsto \min\{n \in \mathbb{N} \mid x \in X_n\}. \end{aligned}$$

Remarque :
Si $x \in a$, il existe alors $n_0 \in \mathbb{N}$ tel que $x \in X_{n_0}$ donc $\{n \in \mathbb{N} \mid x \in X_n\} \neq \varnothing$ donc le minimum existe.

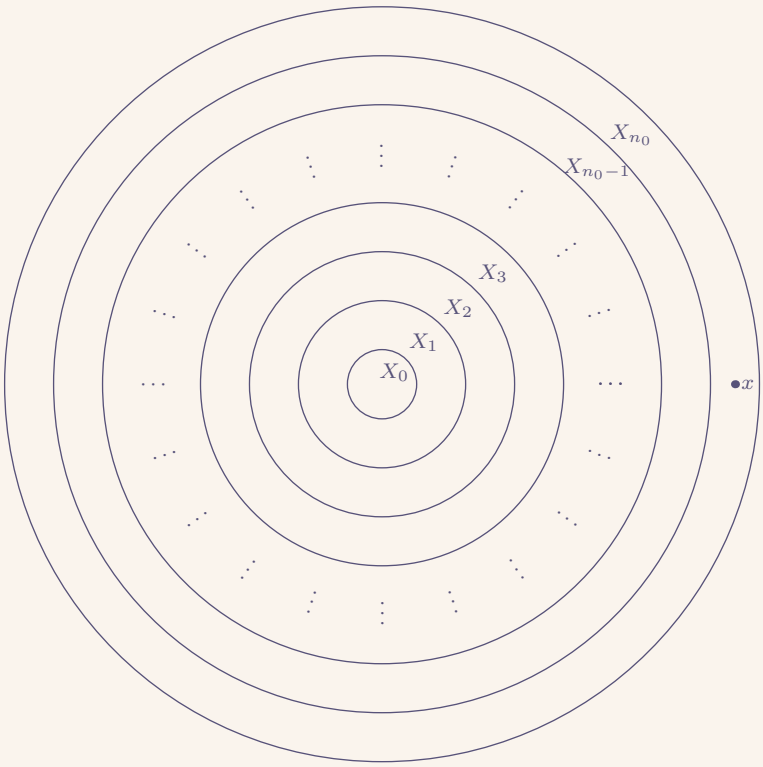


FIGURE −1.7 – Structure des ensembles $X_1, \dots, X_n$

Propriété : Si $x \diamond y$, alors $h(x) < h(y)$.

Preuve :
Soit $x \diamond y$. Alors, $y \in A$. Soit $n_0 = h(y) \neq 0$, on a donc $y \in X_{n_0}$.

- Si $y \in X_{n_0-1}$ ce qui est absurde par définition de h.
- Si $y \in \left\{ S(a, t_1, \dots, t_r) \,\middle|\, S|_C^r \in R \text{ et } a \in C \text{ et } t_1 \in X_{n_0-1} \text{ et } \dots \text{ et } t_r \in X_{n_0-1} \right\}$. Or, soit i_0 tel que $x = t_{i_0}$ donc $x \in X_{n_0-1}$ et donc $h(x) \leqslant n_0 - 1$.

□

Corollaire : Si $n \leqslant y$, alors $h(x) = h(y) \iff x = y$ et $h(x) < h(y) \iff x \prec y$.

Corollaire : La relation $\preccurlyeq$ est antisymétrique.

REMARQUE :
La relation $\preccurlyeq$ est trivialement transitive et reflective. Elle est donc d'ordre.

EXEMPLE :
On prend $R = \left\{ A|^0, B|^0 \right\}$ et $X_0 = \{A, B\}$, $X_1 = X_0$, … L'ensemble S défini par

induction sur R est $\{A, B\}$. On en déduit que $\preccurlyeq$ n'est pas totale.

Exemple :
On prend $R = \left\{0|^0,\ S|^1\right\}$, $X_0 = \{0\}$, $X_1 = \{0,\ S(0)\}$ (on a $0 \prec S(0)$), $X_2 = \{0,\ S(0),\ S(S(0))\}$ (on a $0 \prec S(0) \prec S(S(0))$). L'ensemble obtenu est

Figure −1.8 – Ensemble obtenu avec les règles S et 0

Exemple :
On pose $R = \left\{::|^1_{\mathbb{N}},\ [\,]|^0\right\}$ et $X_0 = \{[\,]\}$, $X_1 = \{::(0, [\,]), [\,], ::(1, [\,], ::(2, [\,])\}$, et $X_2 = \{[\,], ::(0, [\,]), ::(1, ::(0, [\,]))\}$. L'ensemble obtenu est

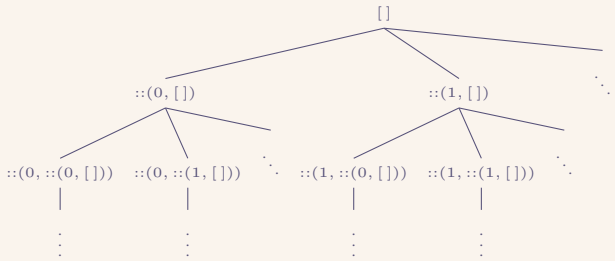


Figure −1.9 – Ensemble obtenu avec les règles :: et $[\,]$

Propriété : La relation $\preccurlyeq$ est bien fondée.

Preuve :
Soit $(x_n)_{n\in\mathbb{N}}$ une suite telle que $x_0 \succ x_1 \succ x_2 \succ \cdots \succ x_n \succ \cdots$ alors

$$\underbrace{h(x_0) > h(x_1) > \cdots > h(x_n) > \cdots}_{\in(\mathbb{N},\leqslant)}.$$

Or, $(\mathbb{N}, \leqslant)$ est bien fondé, c'est donc absurde. □

Remarque :
On peut faire des preuves par induction bien fondée.

Exemple : — Soit l'ensemble trivial défini par $\left\{A|^0, B|^0\right\}$, le théorème est trivial.

— Soit $\mathcal{N}$ défini par $\mathcal{N} = \left\{0|^0,\ S|^1\right\}$. Le théorème donne :

si $P(0)$ vrai et $\forall n \in \mathbb{N},\ (\forall p \in \mathbb{N},\ p < n-1,\ P(\underbrace{S(S(\cdots(S}_{p}(0)\cdots)))) \implies P(\underbrace{S(S(\cdots(S}_{n}(0)\cdots)))$

alors

$$\forall n \in \mathbb{N},\ P(\underbrace{S(S(\cdots(S}_{n}(0)\cdots))).$$

— Soit $\mathscr{L}$ défini par $\mathscr{L} = \left\{[\,]|^0,\ ::|^1_{\mathbb{N}}\right\}$. Si $P([\,])$, et $\forall \ell \in \mathscr{L}, \forall n \in \mathbb{N}, P(\ell) \implies P(::(n,\ell))$ alors $\forall \ell \in \mathscr{L},\ P(\ell)$.

Propriété : Étant donné,
— un ensemble A défini par induction à partir d'un ensemble de règles nommé R,
— un ensemble $\mathbb{I}$,
— une fonction $f_T : C \times \mathbb{I}^r \to \mathbb{I}$ pour chaque règle $T = S|^r_C \in R$,
on défini de manière unique une fonction $f : A \to \mathbb{I}$ telle que, pour tout $x = S(a, t_1, \ldots, t_r) \in A$, soit $T = S|^r_C \in R$ alors $f(x) = f_T(a, f(t_1), \ldots, f(t_r))$.

Exemple :
Sur $\mathscr{L}$ défini par $\left\{\underbrace{[\,]|^0}_{R_1},\ \underbrace{::|^1_{\mathbb{N}}}_{R_2}\right\}$, on choisit $\mathbb{I} = \mathbb{N}$ et

$$f_{R_1} : \overbrace{(\_,\_)}^{\in \text{Inutile}\times\mathbb{N}^0} \mapsto 0$$
$$f_{R_2} : (\underbrace{t}_{\mathbb{N}}, i) \mapsto t + i.$$

On définit alors la fonction

$$f : \mathscr{L} \longrightarrow \mathbb{N}$$
$$::(a_1, ::(a_2, :: \cdots ::(a_n, [\,]) \cdots) \longmapsto \sum_{i=1}^{n} a_i.$$

–1.4 Synthèse du chapitre

— Ordre —

Ordre bien fondé

Un ordre est *bien fondé* s'il n'existe pas de suite infiniment strictement décroissante, *i.e.* toute partie non vide de E admet un élément minimal.

Ordre produit

On définit l'*ordre produit* $\preccurlyeq_\times$ de $(A, \preccurlyeq_A)$ et $(B, \preccurlyeq_B)$ comme

$$(a,b) \preccurlyeq_\times (a',b') \overset{\text{def.}}{\iff} a \preccurlyeq_A a' \text{ et } b \preccurlyeq_B b'.$$

Cet ordre préserve le caractère bien fondé de $\preccurlyeq_A$ et $\preccurlyeq_B$, mais pas le caractère total de la relation. On étend cet ordre à l'ensemble $(A^n, \preccurlyeq_\times)$, en itérant l'ordre produit.

Ordre lexicographique

On définit l'*ordre lexicographique* $\preccurlyeq_\ell$ de $(A, \preccurlyeq_A)$ et $(B, \preccurlyeq_B)$ comme

$$(a,b) \preccurlyeq_\ell (a',b') \overset{\text{def.}}{\iff} \left| \begin{array}{l} (a \prec_A a') \\ \text{ou} \\ (a = a' \text{ et } b \preccurlyeq_B b'). \end{array} \right.$$

Cet ordre conserve le caractère bien fondé de $\preccurlyeq_A$ et $\preccurlyeq_B$, ainsi que le caractère total. On peut également étendre cet ordre à un ensemble $(A^n, \preccurlyeq_\ell)$. On étend également cet ordre à l'ensemble $(A^\star, \preccurlyeq_\ell)$ comme

$$\begin{array}{ccc} u & \prec_\ell & v \\ \cap\!| & & \cap\!| \\ A^n & & A^{n+m} \end{array} \overset{\text{def.}}{\iff} \left| \begin{array}{l} \exists i \in [\![1, n+1]\!], \\ (\forall j < i,\ u_j = u_i) \text{ et} \\ (i = n+1 \text{ ou } u_i \prec_A v_i). \end{array} \right.$$

Le théorème de l'induction bien fondée nous autorise à faire une récurrence forte sur un ensemble ayant un ordre bien fondé.

— Induction nommée —

Une *règle de construction nommée* est de la forme

$$\boldsymbol{S}\big|_C^r \quad \text{ou} \quad \boldsymbol{S}(y, \underbrace{\square, \dots, \square}_{r})$$

où $\boldsymbol{S}$ est un symbole, $r \in \mathbb{N}$ est l'arité de $\boldsymbol{S}$, et C est un ensemble non vide. On omettra parfois l'ensemble C s'il est trivial (*i.e.* de taille 1). Une *règle de base* est de la forme $\boldsymbol{S}\big|_C^0$; une *règle d'induction* est de la forme $\boldsymbol{S}\big|_C^n$.

On définit la *hauteur* h comme

$$h : \bigcup_{n\in\mathbb{N}} X_n \longrightarrow \mathbb{N}$$
$$x \longmapsto \min\{n \in \mathbb{N} \mid x \in X_n\}.$$

On définit la relation $\preccurlyeq$ bien fondée telle que $x \preccurlyeq y$ si x est défini à partir de y. Si $x \preccurlyeq y$, alors $h(x) \leqslant h(y)$.

Un ensemble $\bigcup_{n\in\mathbb{N}} X_n$ est dit *défini par induction nommée* à partir des règles R, si $X_0 = \{\boldsymbol{S}(a) \mid \boldsymbol{S}_C^0 \in R,\ a \in C\}$, et

$$X_{n+1} = X_n \cup \{\boldsymbol{S}(a, t_1, \dots, t_r) \mid \boldsymbol{S}_C^r \in R,\ a \in C, (t_1, \dots, t_r) \in (X_n)^r\}.$$

CHAPITRE

0

LOGIQUE

Sommaire

0.1 Motivation **19**
0.2 Syntaxe **20**
0.3 Sémantique **23**
0.3.1 Algèbre de Boole 23
0.3.2 Fonctions booléennes 24
0.3.3 Interprétation d'une formule comme une fonction booléenne . . . 25
0.3.4 Liens sémantiques 26
0.4 Le problème Sat – Le problème Validité **27**
0.4.1 Résolution par tables de vérité 27
0.5 Représentation des fonction booléennes **28**
0.5.1 Par des formules ? 28
0.5.2 Par des formules sous formes normales ? 30
0.6 Algorithme de Quine **32**
0.7 Synthèse du chapitre **37**

0.1 Motivation

Considérons la grille de Sudoku 2×2 ci-dessous.

3			2
	4	1	
	3	2	
4			1

Figure 0.1 – Grille de Sudoku 2×2

On modélise ce problème : on considère $P_{i,j,k}$ une variable booléenne, c'est à dire un élément de $\{V, F\}$, définie telle que

$$P_{i,j,k} : \text{“}m(i,j) \stackrel{?}{=} k\text{” avec } (i,j,k) \in [\![1,4]\!]^3 \text{ ..}$$

On peut définir des contraintes logiques (des expressions logiques) pour résoudre le Sudoku. Les opérateurs ci-dessous seront définis plus tard.

$$\begin{aligned} & P_{113} \\ \wedge\ & P_{1,4,2} \\ \wedge\ & P_{2,2,4} \\ \wedge\ & P_{2,3,1} \\ & \vdots \\ \wedge\ & P_{1,2,1} \to (\neg P_{1,2,2} \wedge \neg P_{1,2,3} \wedge \neg P_{1,2,4}) \\ & \vdots \end{aligned}$$

Pour résoudre le Sudoku, on peut essayer chaque cas possible. Mais, ces possibilités sont très nombreuses.

En mathématiques, on utilise une certaine logique. Il en existe d'autre, certaines où tout est vrai, certaines où il est plus facile de montrer des théorèmes, etc. On va définir une logique ayant le moins d'opérateurs possibles.

0.2 Syntaxe

Définition : On suppose donné un ensemble $\mathscr{P}$ de variables propositionnelles.

Définition : On définit alors l'ensemble des formules de la logique propositionnelle par induction nommée avec les règles :

- $\neg|^1$;
- $\wedge|^2$;
- $\vee|^2$;
- $\to|^2$;
- $\leftrightarrow|^2$;
- $\top|^0$;
- $\bot|^0$;
- $V|^0_{\mathscr{P}}$.

On nomme l'ensemble des formules $\mathscr{F}$.

Exemple :

$$\vee(\wedge(\to(V(P), \top(\,), \neg(\bot(\,))), \vee(\leftrightarrow(\top(\,), \top(\,)), V(r)).$$

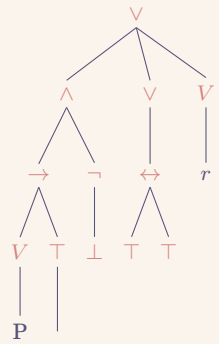


FIGURE 0.2 – Arbre syntaxique d'une expression logique

Pour simplifier la syntaxe, on écrit plutôt

$$((p \to \top) \wedge \neg\bot) \vee ((\top \leftrightarrow \top) \vee r).$$

Définition (taille d'une formule) **:** On définit, par induction, la taille notée "taille" comme

$$\begin{aligned}
\text{taille} : \mathcal{F} &\longrightarrow \mathbb{N} \\
p \in \mathcal{P} &\longmapsto 1 \\
\top &\longmapsto 1 \\
\bot &\longmapsto 1 \\
\neg G &\longmapsto 1 + \text{taille}(G) \\
G \to H &\longmapsto 1 + \text{taille}(G) + \text{taille}(H) \\
G \leftrightarrow H &\longmapsto 1 + \text{taille}(G) + \text{taille}(H) \\
G \wedge H &\longmapsto 1 + \text{taille}(G) + \text{taille}(H) \\
G \vee H &\longmapsto 1 + \text{taille}(G) + \text{taille}(H).
\end{aligned}$$

Définition (Ensemble des variables propositionnelles) **:** On définit inductivement

$$\begin{aligned}
\text{vars} : \mathcal{F} &\longrightarrow \wp(\mathcal{P}) \quad [1] \\
p \in \mathcal{P} &\longmapsto \{p\} \\
\top, \bot &\longmapsto \varnothing \\
\neg G &\longmapsto \text{vars}(G) \\
G \odot H &\longmapsto \text{vars}(G) \cup \text{vars}(H)
\end{aligned}$$

où $\odot$ correspond à $\cup$, $\cap$, $\to$ ou $\leftrightarrow$.

Définition : On appelle *substitution* une fonction de $\mathcal{P}$ dans $\mathcal{F}$ qui est l'identité partout

1. Le $\wp(E)$ représente ici l'ensemble des parties de E.

sauf sur un ensemble fini de variables. On la note alors

$$(p_1 \mapsto H_1,\, p_2 \mapsto H_2, \ldots, p_n \mapsto H_n)$$

qui est la substitution

$$\begin{aligned} \mathscr{P} &\longrightarrow \mathscr{F} \\ p &\longmapsto \begin{cases} H_i & \text{si } p = p_i \\ p & \text{sinon.} \end{cases} \end{aligned}$$

Exemple :
La fonction

$$\sigma = (p \mapsto p \vee q,\, r \mapsto p \wedge \top)$$

est une substitution. On a $\sigma(p) = p \vee q$, $\sigma(r) = p \wedge \top$, $\sigma(q) = q$ et, pour toute autre variable logique a, $\sigma(a) = a$.

Définition (Application d'une substitution à une formule) **:** Étant donné une formule $G \in \mathscr{F}$ et une substitution σ, on définit inductivement $G[\sigma]$ par

$$\begin{cases} \top[\sigma] = \top \\ \bot[\sigma] = \bot \\ p[\sigma] = \sigma(p) \\ (\neg G)[\sigma] = \neg(G[\sigma]) \\ (G \odot H)[\sigma] = (G[\sigma]) \odot H[\sigma] \end{cases}$$

où $\odot$ correspond à $\cup$, $\cap$, $\to$ ou $\leftrightarrow$.

Exemple :
Avec $G = p \wedge (q \vee \top)$ et $\sigma = (p \mapsto p,\, q \mapsto r \wedge \top)$, on a

$$G[\sigma] = q \wedge \big((r \wedge \top) \vee \bot\big).$$

Définition : On appelle parfois *clés* d'une substitution de σ, l'ensemble des variables propositionnelles sur lequel elle n'est pas l'identité.

Définition : On définit la *composée* de deux substitutions σ et σ' par

$$\begin{aligned} \sigma \cdot \sigma' : \mathscr{P} &\longrightarrow \mathscr{F} \\ p &\longmapsto \big(p[\sigma]\big)[\sigma]. \end{aligned}$$

Exemple :
Avec $\sigma = (p \mapsto q)$ et $\sigma = (q \mapsto r)$, on a

$$\sigma' \cdot \sigma = (p \mapsto r,\, q \mapsto r).$$

En effet,

$$\sigma' \cdot \sigma(x) = \begin{cases} r & \text{si } x = p \\ r & \text{si } x = q \\ x & \text{sinon.} \end{cases}$$

Exemple :
Avec $\sigma = (p \mapsto q \wedge \top)$, $\sigma' = (q \mapsto \bot,\ r \mapsto p)$, on a

$$\begin{aligned} \sigma' \cdot \sigma(x) &= \begin{cases} \bot \wedge \top & \text{si } x = p \\ \bot & \text{si} x = q \\ p & \text{si } x = r \\ x & \text{sinon} \end{cases} \\ &= (p \mapsto \bot \wedge \top,\ q \mapsto \bot,\ r \mapsto p). \end{aligned}$$

Remarque :
L'opération $\cdot$ est associative.

Propriété : Soient σ et σ' deux substitutions, on a, pour toute formule $H \in \mathcal{F}$,

$$(H[\sigma])[\sigma'] = H[\sigma' \cdot \sigma].$$

Preuve :
Notons P_G la propriété "$(G[\sigma])[\sigma'] = G[\sigma' \cdot \sigma]$"

Montrons que, pour toute formule $G \in \mathcal{F}$, P_G est vraie par induction :
- $(\top[\sigma])[\sigma'] =^{(\text{def})} \top = \top[\sigma' \cdot]$
- $(p[\sigma])[\sigma'] = p[\sigma' \cdot \sigma]$
- à faire à la maison : le cas $\neg$ et un cas $\wedge$.

□

Définition : On appelle *relation sous-formule*, la relation $\diamond$ définie dans la section 3 du chapitre −1.

0.3 Sémantique

0.3.1 Algèbre de Boole

Définition : On note $\mathbb{B} = \{\boldsymbol{V}, \boldsymbol{F}\}$ l'*ensemble des booléens*.

Définition : Sur $\mathbb{B}$, on définit les opérateurs

a	b	$a \cdot b$
$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{F}$
$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{F}$
$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{F}$
$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{V}$

TABLE 0.1 – Opération $\cdot$ sur les booléens

a	b	$a + b$
$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{F}$
$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{V}$
$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{V}$
$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{V}$

TABLE 0.2 – Opération $+$ sur les booléens

a	$\bar{a}$
$\boldsymbol{F}$	$\boldsymbol{V}$
$\boldsymbol{V}$	$\boldsymbol{F}$

TABLE 0.3 – Opération $\bar{\square}$ sur les booléens

REMARQUE :

NOM	$\cdot$	$+$
Commutativité	$a \cdot b = b \cdot a$	$a + b = b + a$
Neutre	$\boldsymbol{V} \cdot a = a$	$\boldsymbol{F} + a = a$
Absorbant	$\boldsymbol{F} \cdot a = \boldsymbol{F}$	$\boldsymbol{V} \cdot a = \boldsymbol{V}$
Associativité	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$	$a + (b + c) = (a + b) + c$
Idempotence	$a \cdot a = a$	$a + a = a$
Distributivité	$a \cdot (b + c) = a \cdot b + a \cdot c$	$a + (b \cdot c) = (a + b) \cdot (a + c)$
Complémentaire	$a \cdot \bar{a} = \boldsymbol{F}$	$a + \bar{a} = \boldsymbol{V}$
MORGAN	$\overline{a \cdot b} = \bar{a} + \bar{b}$	$\overline{a + b} = \bar{a} \cdot \bar{b}$

TABLE 0.4 – Règles dans $\mathbb{B}$

0.3.2 Fonctions booléennes

Définition (Environnement propositionnel) **:** On appelle *environnement propositionnel* une fonction de $\mathscr{P}$ dans $\mathbb{B}$.

Définition : On appelle *fonction booléenne* une fonction de $\mathbb{B}^{\mathscr{P}}$ dans $\mathbb{B}$. On note l'ensemble des fonctions booléennes $\mathbb{F}$.

REMARQUE :
Si $|\mathscr{P}| = n$, alors $\left|\mathbb{B}^{\mathscr{P}}\right| = 2^n$ et donc $|\mathbb{F}| = 2^{2^n}$.

Exemple :
La fonction

$$f : \begin{pmatrix} (p \mapsto \boldsymbol{F},\, q \mapsto \boldsymbol{F}) \mapsto \boldsymbol{F} \\ (p \mapsto \boldsymbol{F},\, q \mapsto \boldsymbol{V}) \mapsto \boldsymbol{V} \\ (p \mapsto \boldsymbol{V},\, q \mapsto \boldsymbol{F}) \mapsto \boldsymbol{V} \\ (p \mapsto \boldsymbol{V},\, q \mapsto \boldsymbol{V}) \mapsto \boldsymbol{V} \end{pmatrix} \in \mathbb{F}$$

est une fonction booléenne.

0.3.3 Interprétation d'une formule comme une fonction booléenne

Définition (Interprétation) : Étant donné une formule $G \in \mathscr{F}$ et un environnement propositionnel $\rho \in \mathbb{B}^{\mathscr{P}}$, on définit l'*interprétation* de G dans l'environnement ρ par

- $[\![\top]\!]^\rho = \boldsymbol{V}$;
- $[\![\bot]\!]^\rho = \boldsymbol{F}$;
- $[\![p]\!]^\rho = \rho(p)$ où $p \in \mathscr{P}$;
- $[\![\neg G]\!]^\rho = \overline{[\![G]\!]^\rho}$;
- $[\![G \wedge H]\!]^\rho = [\![G]\!]^\rho \cdot [\![H]\!]^\rho$;
- $[\![G \vee H]\!]^\rho = [\![G]\!]^\rho + [\![H]\!]^\rho$;
- $[\![G \to H]\!]^\rho = \overline{[\![G]\!]^\rho} + [\![H]\!]^\rho$;
- $[\![G \leftrightarrow H]\!]^\rho = \left(\overline{[\![G]\!]^\rho} + [\![H]\!]^\rho\right) \cdot \left(\overline{[\![H]\!]^\rho} + [\![G]\!]^\rho\right)$.

Exemple :
Avec $\rho = (p \mapsto \boldsymbol{V}, q \mapsto \boldsymbol{F})$, et $G = (p \wedge \top) \vee (q \wedge \bot)$, on a

$$\begin{aligned} [\![G]\!]^\rho &= [\![(p \wedge \top) \vee (q \wedge \bot)]\!]^\rho \\ &= [\![p \wedge \top]\!]^\rho + [\![q \wedge \bot]\!]^\rho \\ &= [\![p]\!]^\rho \cdot [\![\top]\!]^\rho + [\![q]\!]^\rho \cdot [\![\bot]\!]^\rho \\ &= \rho(p) \cdot \boldsymbol{V} + \rho(q) \cdot \boldsymbol{F} \\ &= \boldsymbol{V} + \boldsymbol{F} \\ &= \boldsymbol{V}. \end{aligned}$$

Définition (Fonction booléenne associée à une formule) : Étant donné une formule G, on note

$$\begin{aligned} \mathbb{F} \ni [\![G]\!] : \mathbb{B}^{\mathscr{P}} &\longrightarrow \mathbb{B} \\ \rho &\longmapsto [\![G]\!]^\rho . \end{aligned}$$

Exemple :
La fonction booléenne associée à $p \vee q$ est

$$f : \begin{pmatrix} (p \mapsto \boldsymbol{F},\, q \mapsto \boldsymbol{F}) \mapsto \boldsymbol{F} \\ (p \mapsto \boldsymbol{F},\, q \mapsto \boldsymbol{V}) \mapsto \boldsymbol{V} \\ (p \mapsto \boldsymbol{V},\, q \mapsto \boldsymbol{F}) \mapsto \boldsymbol{V} \\ (p \mapsto \boldsymbol{V},\, q \mapsto \boldsymbol{V}) \mapsto \boldsymbol{V} \end{pmatrix} \in \mathbb{F}.$$

La fonction booléenne associée à $p \vee (q \wedge \top)$ est aussi f ; tout comme $(p \vee \bot) \vee (q \wedge \top)$.

0.3.4 Liens sémantiques

Définition : On dit que G et H sont *équivalents* si et seulement si $[\![G]\!] = [\![H]\!]$. On note alors $G \equiv H$.

Définition (Conséquence sémantique) **:** On dit que H est *conséquence sémantique* de G dès lors que

$$\forall \rho \in \mathbb{B}^{\mathscr{P}},\ \left([\![G]\!]^\rho = \boldsymbol{V}\right) \implies \left([\![H]\!]^\rho = \boldsymbol{V}\right).$$

On le note $G \models H$.

Propriété : On a

$$G \equiv H \iff (G \models H \text{ et } H \models G).$$

Preuve : " $\implies$ " On suppose $G \equiv H$. Soit $\rho \in \mathbb{B}^{\mathscr{P}}$. On suppose $[\![G]\!]^\rho = \boldsymbol{V}$ alors $[\![H]\!]^\rho = \boldsymbol{V}$ car $[\![G]\!] = [\![H]\!]$. On suppose maintenant $[\![H]\!]^\rho = \boldsymbol{V}$, et alors $[\![G]\!]^\rho = \boldsymbol{V}$ car $[\![G]\!] = [\![H]\!]$.
" $\impliedby$ " On suppose $G \models H$ et $H \models G$. Soit $\rho \in \mathbb{B}^{\mathscr{P}}$. On suppose $[\![G]\!]^\rho = \boldsymbol{V}$ alors $[\![H]\!]^\rho = \boldsymbol{V}$ car $H \models H$ et donc $[\![G]\!] = [\![H]\!]$. On suppose maintenant $[\![H]\!]^\rho = \boldsymbol{V}$ alors $[\![G]\!]^\rho = \boldsymbol{V}$ car $G \models H$. Par contraposée, si $[\![G]\!]^\rho = \boldsymbol{F}$, alors $[\![H]\!]^\rho = \boldsymbol{F}$. On en déduit que $[\![G]\!] = [\![H]\!]$.

□

Remarque :
$\models$ n'est pas une relation d'ordre.

Remarque :
La relation $\equiv$ est une relation d'équivalence. De plus, si $G \equiv G'$ et $H \equiv H'$, alors

- $G \wedge H \equiv G' \wedge H'$;
- $G \vee H \equiv G' \vee H'$;
- $G \to H \equiv G' \to H'$;
- $G \leftrightarrow H \equiv G' \leftrightarrow H'$;
- $\neg G \equiv \neg G'$.

Une telle relation est parfois appelée une *congruence*.

Définition : On dit d'une formule $H \in \mathscr{F}$ qu'elle est

- *valide* ou *tautologique* dès lors que $\forall \rho \in \mathbb{B}^{\mathscr{P}}$, $[\![H]\!]^\rho = \boldsymbol{V}$;
- *satisfiable* dès lors qu'il existe $\rho \in \mathbb{B}^{\mathscr{P}}$, $[\![H]\!]^\rho = \boldsymbol{V}$;
- *insatisfiable* dès lors qu'il n'est pas satisfiable.

On dit de $\rho \in \mathbb{B}^{\mathscr{P}}$ tel que $[\![H]\!]^\rho = \boldsymbol{V}$ que ρ est un *modèle* de H.

Exemple :
- $p \vee \neg p$ est une tautologie. En effet, soit $\rho \in \mathscr{B}^{\mathscr{P}}$, on a

$$[\![p \vee \neg p]\!]^\rho = [\![p]\!]^\rho + \overline{[\![p]\!]^\rho} = \boldsymbol{V}.$$

— p est satisfiable mais non valide. En effet,

$$[\![p]\!]^{(p\mapsto \boldsymbol{V})} = \boldsymbol{V} \qquad \text{et} \qquad [\![p]\!]^{(p\mapsto \boldsymbol{F})} = \boldsymbol{F}.$$

— $p \wedge \neg p$ est insatisfiable. En effet, soit $\rho \in \mathbb{B}^{\mathscr{P}}$, on a

$$[\![p \wedge \neg p]\!]^{\rho} = [\![p]\!]^{\rho} \cdot \overline{[\![p]\!]^{\rho}} = \boldsymbol{F}.$$

Définition : Si Γ est un ensemble de formules, on écrit $\Gamma \models H$ pour dire que

$$\forall \rho \in \mathbb{B}^{\mathscr{P}},\ (\forall G \in \Gamma,\ [\![G]\!]^{\rho} = \boldsymbol{V}) \implies [\![H]\!]^{\rho} = \boldsymbol{V}.$$

Remarque :
Si Γ est fini, alors on a

$$\Gamma \models H \iff \Big(\bigwedge_{G\in\Gamma} G\Big) \models H.$$

On doit faire la preuve, pour $n \geqslant 1$,

$$\{G_1, G_2, \ldots, G_n\} \models H \iff (\cdots((G_1 \wedge G_2) \wedge G_3) \cdots \wedge G_n) \models H.$$

0.4 Le problème Sat – Le problème Validité

On définit le problème Sat comme ayant pour donnée une formule H et pour question "H est-elle satisfiable ?" et le problème Valide comme ayant pour donnée une formule H et pour question "H est-elle valide ?"

0.4.1 Résolution par tables de vérité

a	b	c	$a \wedge b$	$\neg b$	$\neg c$	$\neg b \vee \neg c$	$(a \wedge b) \to (\neg b \vee \neg c)$
$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{F}$
$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{V}$
$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{V}$
$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{V}$
$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{V}$
$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{V}$
$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{V}$
$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{V}$

Table 0.5 – Table de vérité de $(a \wedge b) \to (\neg b \vee \neg c)$

Exemple :

Le problème Sat lit la colonne résultat, on cherche un $\boldsymbol{V}$. Le problème Valide lit la colonne résultat et vérifie qu'il n'y a que des $\boldsymbol{V}$.

Remarque :
Deux formules sont équivalent si et seulement si elles ont la même colonne résultat.

On essaie d'énumérer toutes les possibilités : si $|\mathscr{P}| = n \in \mathbb{N}$, alors le nombre de classes d'équivalences pour $\equiv$ est au plus 2^{2^n}. On cherche donc un meilleur algorithme.

0.5 Représentation des fonction booléennes

0.5.1 Par des formules ?

p	q	r	S
$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{V}$
$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{F}$
$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{F}$
$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{V}$
$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{V}$
$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{F}$
$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{V}$
$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{F}$

Table 0.6 – Table de vérité d'une formule inconnue

On regarde les cas où la sortie est $\boldsymbol{V}$ et on crée une formule permettant de tester cette combinaison de p, q et r uniquement. On unie toutes ces formules par des $\vee$. Dans l'exemple ci-dessus, on obtient

$$(\neg p \wedge \neg q \wedge \neg r) \vee (\neg p \wedge \neg q \wedge r) \vee (p \wedge \neg q \wedge \neg r) \vee (p \wedge q \wedge \neg r).$$

Théorème : Soit $f : \mathbb{B}^{\mathscr{P}} \to \mathbb{B}$ une fonction booléenne avec $\mathscr{P}$ fini. Il existe une formule $H \in \mathscr{F}$ telle que $[\![H]\!] = f$.

Avant de prouver ce théorème, on démontre d'abord les deux lemme suivants et on définit lit_ρ.

Définition : Soit $\rho \in \mathbb{B}^{\mathscr{P}}$. On définit

$$\mathrm{lit}_\rho(p) = \begin{cases} p & \text{si } \rho(p) = \boldsymbol{V}; \\ \neg p & \text{sinon.} \end{cases}$$

Lemme :

$$\forall \rho \in \mathbb{B}^{\mathscr{P}},\ \exists G \in \mathscr{F},\ \left(\forall \rho' \in \mathbb{B}^{\mathscr{P}},\ [\![G]\!]^{\rho'} = \boldsymbol{V} \iff \rho = \rho'\right).$$

On prouve ce lemme :

Preuve :

$\mathscr{P}$ est fini. Notons donc $\mathscr{P} = \{p_1, \dots, p_n\}$ ses variables. Soit alors $\rho \in \mathbb{B}^{\mathscr{P}}$, on définit

$$H_\rho = \bigwedge_{i=1}^{n} \mathrm{lit}_\rho(p_i).$$

Montrons que $[\![H_\rho]\!]^{\rho'} = \boldsymbol{V} \iff \rho = \rho'$. Soit $\rho' \in \mathbb{B}^{\mathscr{P}}$.

— Si $\rho = \rho'$, alors

$$\begin{aligned}[\![H_\rho]\!]^{\rho'} &= \Big[\!\Big[\bigwedge_{i=1}^{n} \mathrm{lit}_\rho(p_i) \Big]\!\Big]^{\rho'} \\ &= \bigbullet_{i=1}^{n} [\![\mathrm{lit}_\rho(p_i)]\!]^{\rho'}\end{aligned}$$

Soit $i \in [\![1, n]\!]$. Si $\rho(p_i) = \boldsymbol{V}$ alors $\rho'(p_i) = \boldsymbol{V}$, or, $\mathrm{lit}_\rho(p_i) = p_i$ et donc $[\![\mathrm{lit}_\rho(p_i)]\!]^{\rho'} = [\![p_i]\!]^{\rho'} = \boldsymbol{V}$; sinon si $\rho(p_i) = \boldsymbol{F}$, alors $\rho'(p_i) = \boldsymbol{F}$, or, $\mathrm{lit}_\rho(p_i) = \neg p_i$ et donc

$$[\![\mathrm{lit}_\rho(p_i)]\!]^{\rho'} = [\![\neg p_i]\!]^{\rho'} = [\![p_i]\!]^{\rho'} = \rho'(p_i) = \bar{\boldsymbol{F}} = \boldsymbol{V}.$$

et comme ceci étant vrai pour tout $i \in [\![1, n]\!]$, on a

$$\bigbullet_{i=1}^{n} [\![\mathrm{lit}_\rho(p_i)]\!]^{\rho'} = \boldsymbol{V}.$$

— Sinon ($\rho \neq \rho'$), soit donc $p_i \in \mathscr{P}$ tel que $\rho(p_i) \neq \rho'(p_i)$. Si $\rho(p_i) = \boldsymbol{V}$ alors $\rho'(p_i) = \boldsymbol{F}$ et donc $\mathrm{lit}_\rho(p_i) = p_i$ et $[\![\mathrm{lit}_\rho\, p_i]\!]^{\rho'} = \rho'(p_i) = \boldsymbol{F}$; sinon si $\rho(p_i) = \boldsymbol{F}$, alors $\rho'(p_i) = \boldsymbol{V}$ et donc $\mathrm{lit}_\rho(p_i) = \neg p_i$ et $[\![\mathrm{lit}_\rho(p_i)]\!]^{\rho'} = [\![\neg p_i]\!]^{\rho'} = \overline{[\![p_i]\!]^{\rho'}} = \bar{\boldsymbol{V}} = \boldsymbol{F}$.

On en déduit donc que

$$[\![H_\rho]\!]^{\rho'} = \bigbullet_{j=1}^{n} [\![\mathrm{lit}_\rho(p_j)]\!]^{\rho'} = \boldsymbol{F}$$

car il existe $i \in [\![1, n]\!]$ tel que $[\![\mathrm{lit}_\rho(p_i)]\!]^{\rho'} = \boldsymbol{F}$. □

On peut donc maintenant prouver le théorème :

Lemme : Considérons alors la formule

$$H = \bigvee_{\substack{\rho \in \mathbb{B}^{\mathscr{P}} \\ f(\rho) = \boldsymbol{V}}} H_\rho.$$

On a $[\![H]\!] = f$.

Preuve : — Soit $\rho \in \mathbb{B}^{\mathscr{P}}$ tel que $f(\rho) = \boldsymbol{V}$, on a donc

$$[\![H]\!]^{\rho} = \Big[\!\Big[\bigvee_{\substack{\rho' \in \mathbb{B}^{\mathscr{P}} \\ f(\rho') = \boldsymbol{V}}} H_{\rho'} \Big]\!\Big]^{\rho}.$$

H_ρ apparaît donc dans cette disjonction. Or, $[\![H_\rho]\!] = \boldsymbol{V}$ et donc $[\![H]\!]^\rho = \boldsymbol{V}$.
Si $f(\rho) = \boldsymbol{F}$, alors on a vu que $\forall \rho'$ tel que $f(\rho') = \boldsymbol{V}$, alors $\rho' \neq \rho$ et donc $[\![H_{\rho'}]\!]^\rho = \boldsymbol{F}$ et donc

$$\Big[\!\Big[\bigvee_{\substack{\rho' \in \mathbb{B}^{\mathscr{P}} \\ f(\rho') = \boldsymbol{V}}} H_{\rho'} \Big]\!\Big] = \boldsymbol{F}.$$

Finalement $[\![H]\!] = f$.

□

Le théorème est prouvé directement à l'aide des deux lemmes précédents.

On connaît donc la réponse à la question du nom de ce paragraphe, à savoir "peut-on représenter les fonctions booléennes par des formules ?" Oui.

0.5.2 Par des formules sous formes normales ?

Définition : On dit d'une formule de la forme
- p ou $\neg p$ avec $p \in \mathcal{P}$, que c'est un *littéral* ;
- $\bigwedge_{i=1}^{n} \ell_i$ où les ℓ_i sont des littéraux que c'est une *clause conjonctive* ;
- $\bigvee_{i=1}^{n} \ell_i$ où les ℓ_i sont des littéraux que c'est une *clause disjonctive* ;
- $\bigwedge_{i=1}^{n} D_i$ où les D_i qui sont des clauses disjonctives est appelée une *forme normale conjonctive* ;
- $\bigvee_{i=1}^{n} C_i$ où les C_i qui sont des clauses conjonctives est appelée une *forme normale disjonctive*.

Remarque :
On prend, comme convention, que $\bigwedge_{i=1}^{0} G_i = \top$ et $\bigvee_{i=1}^{0} G_i = \bot$.

Exemple :
La formule $\overbrace{(p \wedge \neg q)}^{\text{clause conjonctive}} \vee \overbrace{(r \wedge p)}^{\text{clause conjonctive}}$ est donc une clause normale disjonctive.

Remarque :
On écrit FMD pour une forme normale disjonctive et FNC pour une forme normale conjonctive.

Exemple :
La formule $p \wedge q \wedge \neg r$ est une clause conjonctive donc une FNC mais c'est aussi une FND.

Exemple :
La formule $\top$ est une clause conjonctive de taille 0, donc c'est une FND. Mais, c'est aussi une clause conjonctive de taille 0, donc c'est une FNC. De même, la formule $\bot$ est une FNC et une FND.

Théorème : Toute formule est équivalente à une formule sous FND et à une formule sous FNC.

Preuve :
Soit $G \in \mathcal{F}$ une formule. Soit $[\![G]\!]$ la fonction booléenne associée à G. Alors, par le théorème précédent, il existe une formule H telle que $[\![H]\!] = [\![G]\!]$ (i.e. $H \equiv G$) avec H construit dans la preuve précédente sous forme normale disjonctive. □

Exemple :
La formule $G = p \wedge (\neg q \vee p)$ a pour table de vérité la table suivante.

p	q	$[\![G]\!]$
F	**F**	**F**
F	**V**	**F**
V	**F**	**V**
V	**V**	**V**

Table 0.7 – Table de vérité de $p \wedge (\neg q \vee p)$

La forme normale disjonctive équivalente à G est $(p \wedge \neg q) \vee (p \wedge q)$.

Nous n'avons pas encore prouvé la deuxième partie du théorème mais, on essaie de trouver une formule sous fnc :

Exemple :
On reprend l'exemple de la table de vérité d'une fonction inconnue.

p	q	r	f	$\bar{f}$
F	**F**	**F**	**V**	**F**
F	**F**	**V**	**F**	**V**
F	**V**	**F**	**F**	**V**
F	**V**	**V**	**V**	**F**
V	**F**	**F**	**V**	**F**
V	**F**	**V**	**F**	**V**
V	**V**	**F**	**V**	**F**
V	**V**	**V**	**F**	**V**

Table 0.8 – Table de vérité d'une formule inconnue (2)

On analyse la formule $\bar{f}$ au lieu de f. Grâce à la première partie du théorème (et de la méthode pour générer cette fnd), on a

$$\bar{f} = (\neg p \wedge \neg q \wedge r) \vee (\neg p \wedge q \wedge \neg r) \vee (p \wedge \neg q \wedge r) \vee (p \wedge q \wedge r).$$

Et, à l'aide des lois de De Morgan, on a

$$\bar{f} = (p \vee q \vee \neg r) \wedge (p \vee \neg q \vee r) \wedge (\neg p \vee q \vee \neg r) \wedge (\neg p \vee \neg q \vee \neg r),$$

ce qui est une fnc.

À l'aide de cet algorithme, on prouve facilement la $2^{\underline{\text{nde}}}$ partie du théorème.

Remarque :
Il est en fait possible de transformer une formule en fnd en appliquant les règles suivantes à toutes les sous-formules jusqu'à obtention d'un point fixe.

- $\neg\neg H \rightsquigarrow H$;
- $\neg(G \wedge H) \rightsquigarrow G \vee H$;
- $\neg(G \vee H) \rightsquigarrow G \wedge H$;
- $(G \vee H) \wedge I \rightsquigarrow (G \wedge I) \vee (H \wedge I)$;
- $I \wedge (G \vee H) \rightsquigarrow (I \wedge G) \vee (I \wedge H)$;

— $H \wedge \top \rightsquigarrow H$;
— $\top \wedge H \rightsquigarrow H$;
— $H \vee \bot \rightsquigarrow H$;
— $\bot \vee H \rightsquigarrow H$;
— $\neg\top \rightsquigarrow \bot$;
— $\bot \wedge H \rightsquigarrow \bot$;
— $H \wedge \bot \rightsquigarrow \bot$;
— $\neg\bot \rightsquigarrow \top$;
— $\top \vee H \rightsquigarrow \top$;
— $H \vee \top \rightsquigarrow \top$.

Propriété : Soit $n \geqslant 2$ et H_n la formule $H_n = (a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \cdots \wedge (a_n \vee b_n)$ avec $\mathscr{P}_n = \{a_1, b_1, a_2, b_2, \ldots, a_n, b_n\}$. Alors, par application de l'algorithme précédent on obtient

$$\bigvee_{P \in \wp([\![1,n]\!])} \left(\bigwedge_{j=1}^{n} \begin{cases} a_j \text{ si } j \in P \\ b_j \text{ sinon} \end{cases} \right).$$

À faire :

Preuve (par récurrence) *:*

□

Remarque :
Qu'en est-il du problème Sat ? Le problème est-il simplifié pour les fnd ou les fnc ?

Oui, pour les fnd, le problème se simplifie. On considère, par exemple, la formule

$$\begin{array}{ll} & (\ell_{11} \wedge \ell_{12} \wedge \cdots \wedge \ell_{1,n_1}) \\ \vee & (\ell_{21} \wedge \ell_{22} \wedge \cdots \wedge \ell_{2,n_2}) \\ \vdots & \vdots \\ \vee & (\ell_{m,1} \wedge \ell_{m,2} \cdots \ell_{m,n_m}). \end{array}$$

On procède en suivant l'algorithme suivant : (À faire : Mettre l'algorithme à part) Pour i fixé, je lis la ligne i, puis je fabrique un environnement ρ.

Par exemple, pour $(p \wedge \neg q \wedge r \wedge \neg p) \vee (q \wedge r \wedge \neg q) \vee (p \wedge r)$, on a $\rho = (p \mapsto \boldsymbol{V}, r \mapsto \boldsymbol{V})$.

On en conclut que Sat peut être résolu en temps linéaire dans le cas d'une forme normale disjonctive. Le problème est de construire cette fnd.

Remarque :
Après s'être intéressé au problème Sat, on s'intéresse au problème Valide.

Par exemple, on considère la formule $(p \vee q \vee \neg r \vee \neg p) \wedge (p \vee \neg r \vee p \vee r) \wedge (q \vee r)$. On peut construire $\rho = (q \mapsto \boldsymbol{F}, r \mapsto \boldsymbol{F})$ est tel que $[\![H]\!]^\rho = \boldsymbol{F}$.

Si on ne peut pas construire un tel environnement propositionnel, la formule vérifie le problème Valide.

On en conclut que Valide peut être résolu en temps linéaire dans le cas d'une forme normale conjonctive. Le problème est de construire cette fnc.

0.6 Algorithme de Quine

L'objectif de cet algorithme est de résoudre le problème Sat. On commence par poser quelques lemmes, puis on donne l'algorithme.

Remarque :
Une forme normale peut être vue comme un ensemble d'ensembles de littéraux (c'est la représentation que nous allons utiliser en OCamL).

Exemple :

L'ensemble $\{\{p,q\},\{p,r\},\varnothing\}$, a pour formule sous FNC associée $(p\vee\neg q)\wedge(q\vee r)\wedge\bot$.

L'ensemble $\{\{p,\neg q\},\{q,r\},\varnothing\}$ a pour formule sous FND associée $(p\wedge\neg q)\vee(q\wedge r)\vee\top$.

L'ensemble $\varnothing$ a pour formule sous FNC associée $\top$.

L'ensemble $\varnothing$ a pour formule sous FND associée $\bot$.

Lemme : Pour toute formule H, pour tout variable propositionnelle p et pour tout environnement propositionnel ρ, tel que $\rho(p)=\boldsymbol{V}$, alors

$$\Big[\!\Big[H[p\mapsto\top]\Big]\!\Big]^{\rho}=[\![H]\!]^{\rho}.$$

Preuve (par induction sur les formules) :

— Si $H=p$, avec $p\in\mathcal{P}$, alors

$$\Big[\!\Big[H[p\mapsto\top]\Big]\!\Big]^{\rho}=[\![\top]\!]^{\rho}=\boldsymbol{V}=[\![H]\!]^{\rho}.$$

— Si $H=\top$, alors $H[p\mapsto\top]=H$.

— Si $H=\neg H_1$ tel que $\Big[\!\Big[H_1[p\mapsto\top]\Big]\!\Big]^{\rho}=[\![H_1]\!]^{\rho}$, alors,

$$\begin{aligned}\Big[\!\Big[H[p\mapsto\top]\Big]\!\Big]^{\rho}&=\Big[\!\Big[\neg(H_1[p\mapsto\top])\Big]\!\Big]^{\rho}\\&=\overline{\Big[\!\Big[H_1[p\mapsto\top]\Big]\!\Big]^{\rho}}\\&=\overline{[\![H_1]\!]^{\rho}}\\&=[\![\neg H_1]\!]^{\rho}\\&=[\![H]\!]^{\rho}.\end{aligned}$$

— Si $H=H_1\wedge H_2$ avec H_1 et H_2 vérifiant l'hypothèse d'induction, alors

$$\begin{aligned}\Big[\!\Big[H[p\mapsto\top]\Big]\!\Big]^{\rho}&=\Big[\!\Big[(H_1[p\mapsto\top])\wedge(H_2[p\mapsto\top])\Big]\!\Big]^{\rho}\\&=\Big[\!\Big[H_1[p\mapsto\top]\Big]\!\Big]^{\rho}\cdot\Big[\!\Big[H_2[p\mapsto\top]\Big]\!\Big]^{\rho}\\&=[\![H_1]\!]^{\rho}\cdot[\![H_2]\!]^{\rho}\\&=[\![H_1\wedge H_2]\!]^{\rho}.\end{aligned}$$

□

Remarque :
Le résultat reste vrai en remplaçant $\boldsymbol{V}$ par $\boldsymbol{F}$ et $\top$ par $\bot$.

Lemme : Pour toute formule H, et pour toute variable propositionnelle p, H est satisfiable si, et seulement si $H[p\mapsto\top]$ est satisfiable ou $H[p\mapsto\bot]$ est satisfiable.

Preuve " $\Longrightarrow$ " Soit $H \in \mathcal{F}$ une formule satisfiable. Il existe donc un environnement propositionnel ρ défini sur $\text{vars}(H)$ tel que $[\![H]\!]^\rho = \boldsymbol{V}$.

- Si $\rho(p) = \boldsymbol{V}$, alors $\left[\!\left[H[p \mapsto \top]\right]\!\right]^\rho = \boldsymbol{V}$, d'après le lemme précédent. Donc, $H[p \mapsto \top]$ est satisfiable.
- Sinon, $\rho(p) = \boldsymbol{F}$ et donc, d'après le lemme précédent, $\left[\!\left[H[p \mapsto \bot]\right]\!\right]^\rho = \boldsymbol{V}$. La formule $H[p \mapsto \bot]$ est satisfiable.

" $\Longleftarrow$ " Supposons $H[p \mapsto \top]$ satisfiable. Il existe donc un environnement propositionnel μ défini sur $\text{vars}(H[p \mapsto \top])$ tel que $[\![H[p \mapsto \top]]\!]^\mu = \boldsymbol{V}$. Or, $\text{vars}(H[p \mapsto \top]) = \text{vars}(H) \setminus \{p\}$. On peut donc étendre μ en

$$\rho : \text{vars}(H) \longrightarrow \mathbb{B}$$
$$x \longmapsto \begin{cases} \mu(x) & \text{si } x \neq p \\ \boldsymbol{V} & \text{sinon.} \end{cases}$$

Ainsi, $[\![H]\!]^\mu = [\![H[p \mapsto \top]]\!]^\rho = [\![H]\!]^\rho$ car $p \notin \text{vars}(H[p \mapsto \top])$.

□

Lemme : Une formule sans variables est

- satisfiable si et seulement si elle est équivalente à $\top$;
- insatisfiable si et seulement si elle est équivalente à $\bot$;

Preuve :
Soit H une telle formule.

$$H \equiv \top \iff \forall \rho \in \mathbb{B}^\varnothing,\ [\![H]\!]^\rho = [\![\top]\!]^\rho$$
$$\iff [\![H]\!]^{()} = [\![\top]\!]^{()}$$

Si H est satisfiable, il existe un environnement propositionnel $\rho \in \mathbb{B}^\varnothing$ tel que $[\![H]\!]^\rho = \boldsymbol{V} = [\![H]\!]^{()}$. On conclut que $H \equiv \top$. De même si H est insatisfiable. □

```
type formule =
  | Top | Bot
  | Var   of string
  | Not   of formule
  | And   of formule * formule
  | Or    of formule * formule
  | Imply of formule * formule
  | Equiv of formule * formule

let substitution (f: formule) (x: string) (g: string): formule =
  ⟨code déjà fait en TP⟩

exception Found of string (* arret de la boucle *)
let get_var (f: formule): string option =
  let rec aux (f: formule): unit =
    match f with
    | Top | Bot       -> ()
    | Var(y)          -> raise(Found(y))
    | Not(f1)         -> aux f1
    | And(f1, f2)
    | Or(f1, f2)
    | Imply(f1, f2)
    | Equiv(f1, f2)   -> (aux f1; aux f2)
  in try
    aux f;
    None
  with
  | Found(y) -> Some(y)

let rec test_equiv (f: formule): bool option =
  match f with
  | Var(_) -> None
```

```
33    | Top     -> Some(true)
34    | Bot     -> Some(false)
35    | Not(h) ->
36              begin
37                match test_equiv h with
38                | None     -> None
39                | Some(b) -> Some(not b)
40              end
41    | And(h1, h2) ->
42              begin
43                match test_equiv h1, test_equiv h2 with
44                | Some(false), _ | _, Some(false) -> Some(false)
45                | Some(true), Some(true)          -> Some(true)
46                | _                               -> None
47              end
48    | Or(h1, h2) ->
49              begin
50                match test_equiv h1, test_equiv h2 with
51                | Some(true), _ | _, Some(true) -> Some(true)
52                | Some(false), Some(false)      -> Some(true)
53                | _                            -> None
54              end
55    | Imply(h1, h2) ->
56              begin
57                match test_equiv h1, test_equiv h2 with
58                | Some(false), _
59                | Some(true), Some(true)       -> Some(true)
60                | Some(true), Some(false)      -> Some(false)
61                | _                            -> None
62              end
63    | Equiv(h1, h2) ->
64              begin
65                match test_equiv h1, test_equiv h2 with
66                | Some(true), Some(true)
67                | Some(false), Some(false)     -> Some(true)
68                | Some(true), Some(false)
69                | Some(false), Some(true)      -> Some(false)
70                | _                            -> None
71              end
72
73  let rec quine (f: formule): bool =
74    match get_var f, test_equiv f with
75    | _, Some(b)    -> b
76    | None, _       -> failswith "cas impossible"
77    | Some(p), None -> ?
```

Code 0.1 – Algorithme de Quine *version zéro*

> Remarque :
> Dans la suite, on s'intéresse aux formules sous forme cnf. Une cnf (ou même une dnf) peut être représentée au moyen d'un ensemble d'ensemble de littéraux. Ces ensembles sont finis. Par exemple, la formule $(p \vee \neg q) \wedge (r \vee p \vee p)$ est représenté par
>
> $$\{\{p, \neg q\}, \{r, p\}\}.$$

Algorithme 0.1 Algorithme *Assume*

Entrée G une cnf, p une variable propositionnelle, et $b \in \mathbb{B}$.
Sortie Une cnf équivalente à $G[p \mapsto b]$.

```
1 : Soit ℓ_V le littéral p si b = V, ¬p sinon.
2 : Soit ℓ_F le littéral ¬p si b = V, p sinon.
3 : pour C ∈ G faire
4 :     si ℓ_V ∈ C alors
5 :         On retire C de G.
6 :     sinon
7 :         si ℓ_F ∈ C alors
8 :             On retire ℓ_F de C.
```

On peut donc donner l'algorithme de Quine final :

Algorithme 0.2 Algorithme de Quine

Entrée Une cnf G

si $G = \varnothing$ **alors**
  retourner Oui
sinon
  si $\varnothing \in G$ **alors**
    retourner Non
  sinon si $\exists \{\ell\} \in G$ **alors**
    si $\ell = p$, avec $p \in \mathrm{vars}(G)$ **alors**
      Quine($\mathit{Assume}(G, p, \boldsymbol{V})$)
    sinon si $\ell = \neg p$, avec $p \in \mathrm{vars}(G)$ **alors**
      Quine($\mathit{Assume}(G, p, \boldsymbol{F})$)
  sinon
    $p \leftarrow h(G)$
    On essaie Quine($\mathit{Assume}(G, p, \boldsymbol{V})$)
    On essaie Quine($\mathit{Assume}(G, p, \boldsymbol{F})$)

0.7 Synthèse du chapitre

— Formules —

On définit l'ensemble des formules $\mathcal{F}$ par induction nommé avec les règles

- $\neg\big|^1$;
- $\leftrightarrow\big|^2$;
- $\wedge\big|^2$;
- $\top\big|^0$;
- $\vee\big|^2$;
- $\bot\big|^0$;
- $\rightarrow\big|^2$;
- $V\big|^0_{\mathcal{P}}$.

On définit inductivement $\text{taille}(F)$, pour $F \in \mathcal{F}$, la taille de cette formule, *i.e.* le nombre d'opérateurs dans cette formule. On définit également l'ensemble des variables $\text{vars}(F)$ d'une formule $F \in \mathcal{F}$.

Une *substitution* est une fonction de $\mathcal{P}$ dans $\mathcal{F}$, où elle est l'identité partout, sauf en nombre fini de variables, alors nommés *clés* de cette substitution. On note $F[\sigma]$ l'application d'une substitution σ à une formule $F \in \mathcal{F}$. On définit la composée de deux substitutions $\sigma \cdot \sigma'$, comme $\sigma \cdot \sigma' : p \mapsto (p[\sigma])[\sigma']$, cela ne correspond pas à la définition mathématique d'une composition de fonctions.

— Fonctions booléennes —

Un *environnement propositionnel* est une fonction de $\mathcal{P}$ dans $\mathbb{B}$. Une *fonction booléenne* est une fonction de $\mathbb{B}^{\mathcal{P}}$ dans $\mathbb{B}$. L'ensemble $\mathbb{F}$ est l'ensemble des fonctions booléennes.

On définit inductivement l'*interprétation* d'une formule $F \in \mathcal{F}$ dans un environment ρ. On note ce booléen $[\![F]\!]^\rho$. On note également $[\![F]\!]$ l'application $\rho \mapsto [\![F]\!]^\rho$.

— Liens sémantiques —

On note $G \equiv H$ si, et seulement si $[\![G]\!] = [\![H]\!]$. On note $G \models H$ dès lors que, pour $\rho \in \mathbb{B}^{\mathcal{P}}$, si $[\![G]\!]^\rho = \boldsymbol{V}$, alors $[\![H]\!]^\rho = \boldsymbol{V}$. On étend cette définition pour un ensemble Γ de formules G. On a $G \equiv H$ si, et seulement si $G \models H$ et $H \models G$.

Une formule *valide* ou *tautologique* est une formule dont l'interprétation vaut toujours $\boldsymbol{V}$, peu importe l'environnement propositionnel. Une formule satisfiable est une formule dont l'interprétation vaut $\boldsymbol{V}$, pour un certain environnement propositionnel. Si $\rho \in \mathbb{B}^{\mathcal{P}}$ vérifie $[\![F]\!]^\rho = \boldsymbol{V}$, on dit que ρ est un *modèle* de F.

CHAPITRE

1

LANGAGES RÉGULIERS ET AUTOMATES

Sommaire

1.1 Motivation **40**
1.1.1 1ère motivation 40
1.1.2 2nde motivation 40
1.2 Mots et langages, rappels **41**
1.3 Langage régulier **42**
1.3.1 Opérations sur les langages 42
1.3.2 Expressions régulières 44
1.4 Automates finis (sans ε-transitions) **46**
1.4.1 Définitions 46
1.4.2 Transformations en automates équivalents 49
1.5 Automates finis avec ε-transitions **52**
1.5.1 Cloture par concaténation 54
1.5.2 Cloture par étoile 55
1.5.3 Cloture par union 55
1.6 Théorème de Kleene **57**
1.6.1 Langages locaux 58
1.6.2 Expressions régulières linéaires 62
1.6.3 Automates locaux 64
1.6.4 Algorithme de Berry-Sethi : les langages réguliers sont reconnaissables 67
1.6.5 Les langages reconnaissables sont réguliers 68
1.7 La classe des langages réguliers **72**
1.7.1 Limite de la classe/Lemme de l'étoile 73
Annexe 1.A Comment prouver la correction d'un programme ? **75**
Annexe 1.B Hors-programme **76**

1.1 Motivation

1.1.1 1ère motivation

Il y a une grande différence entre les mathématiques et l'informatique : la gestion de l'infini. On n'a pas de mémoire infinie sur un ordinateur. Par exemple, pour représenter π ou $\sqrt{2}$, on ne peut pas stocker un nombre infini de décimales. Ce n'est pas une question de base, ces nombres ont aussi des décimales infinies dans une base 2.

Par exemple, si on ne veut utiliser $\sqrt{2}$ seulement pour plus tard le mettre au carré. On peut définir une structure en C comme celle qui suit

```c
typedef struct {
  int carre;
  bool sign;
};
```

Code 1.1 – $\sqrt{2}$ sous forme de structure

On a pu décrire $\sqrt{2}$ comme cela car il y a une certaine régularité dans ce nombre.

On définit des relations entre ces objets. Dans ce chapitre, on va commencer par étudier autre chose : les mots. Les mots sont utilisés, tout d'abord, pour entrer une liste de lettres mais aussi le programme lui-même. En effet, il y a une liste infinie de code possibles en C.

1.1.2 2nde motivation

Les ordinateurs sont complexes ; il peut être dans une multitude d'états. On représente une succession de tâches (le symbole ● représente une tâche) :

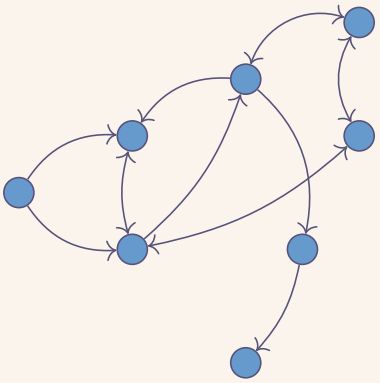


Figure 1.1 – États d'un ordinateur

Par exemple, pour une boucle infinie, on a un cycle dans le graphe ci-dessus. Ou, s'il atteint un certain nœud, on a un *bug*.

Mais, pour résoudre ce problème, on peut forcer le nombre d'état d'un ordinateur ; par exemple, dire qu'un ordinateur a 17 états.

On décide donc de représenter mathématiquement un ordinateur afin de pouvoir faire des preuves avec. Et, c'est l'objet de ce chapitre.

1.2 Mots et langages, rappels

Définition : On appelle *alphabet* un ensemble fini, d'éléments qu'on appelle *lettres*.

Définition : On appelle une *mot* sur Σ (où Σ est un alphabet) une suite finie de lettres de Σ.

La *longueur* d'un mot est le nombre de lettres, comptées avec leurs multiplicité. On la note $|w|$ pour un mot w. Si $|w| = n \in \mathbb{N}^\star$, on indexe les lettres de w pour $(w_i)_{i\in[\![1,n]\!]}$ et on écrit alors

$$w = w_1 w_2 w_3 \dots w_n.$$

Il existe un unique mot de longueur 0 appelé *mot vide*, on le note ε.

Définition : Si Σ est un alphabet, on note
- Σ^n les mots de longueur n ;
- $\Sigma^\star$ les mots de longueurs positives ou nulle ;
- Σ^+ les mots de longueurs strictement positives.

Remarque :

$$\Sigma^\star = \bigcup_{n\in\mathbb{N}} \Sigma^n \qquad \Sigma^+ = \bigcup_{n\in\mathbb{N}^\star} \Sigma^n.$$

Définition : Soit Σ un alphabet. Soit x et y deux mots de $\Sigma^\star$. Notons

$$x = x_1 x_2 x_3 \dots x_n$$
$$y = y_1 y_2 y_3 \dots y_n.$$

On définit alors *concaténation* notée $x \cdot y$ l'opération définie comme

$$x \cdot y = x_1 x_2 x_3 \dots x_n y_1 y_2 \dots y_n.$$

Remarque :
- $\cdot$ est une opération interne ;
- ε est neutre pour $\cdot$: $\forall n \in \Sigma^\star,\ \varepsilon \cdot x = x \cdot \varepsilon = x$.
- $\cdot$ est associatif.

(On dit que c'est un monoïde.)

Remarque :
L'opération $\cdot$ n'est pas commutative.

Définition : Soit x et y deux mots sur l'alphabet Σ. On dit que
- x est un *préfixe* de y si $\exists v \in \Sigma^\star,\ y = x \cdot v$;
- x est un *suffixe* de y si $\exists v \in \Sigma^\star,\ y = v \cdot x$;
- x est un *facteur* de y si $\exists (v, w) \in \left(\Sigma^\star\right)^2,\ y = v \cdot x \cdot w$.

Exemple : — a est facteur de aa ;
— ε est un facteur de a.

Définition : On dit que x est un *sous-mot* de y si x est une suite extraite de y. Par exemple

$$a \not b\, a\, a \not b\, a \qquad \longrightarrow \qquad a\, a\, a\, a.$$

Définition : Un *langage* est un ensemble de mots. C'est donc un élément de $\wp(\Sigma^\star)$.

Remarque (⚠) :
$\varnothing \neq \{\varepsilon\}$.

1.3 Langage régulier

1.3.1 Opérations sur les langages

Remarque :
Les langages sont des ensembles. On peut donc leurs appliquer des opérations ensemblistes.

Définition : Soient $L_1, L_2 \in \wp(\Sigma^\star)$ deux langages. On définit la *concaténation* de deux langages, notée $L_1 \cdot L_2$:

$$L_1 \cdot L_2 = \{u \cdot v \mid u \in L_1,\ v \in L_2\}.$$

Remarque : — L'opération $\cdot$ (langages) a $\{\varepsilon\}$ pour neutre.
— L'opération $\cdot$ (langages) a $\varnothing$ pour élément absorbant :

$$\forall L \in \wp(\Sigma^\star),\ L \cdot \varnothing = \varnothing \cdot L = \varnothing.$$

— L'opération $\cdot$ est distributive : soient K, L, M trois langages ; on a

$$K \cdot (L \cup M) = (K \cdot L) \cup (K \cdot M);$$
$$(L \cup M) \cdot K = (L \cdot K) \cup (M \cdot K).$$

Définition : Étant donné un langage L, on définit par récurrence :
- $L^0 = \{\varepsilon\}$;
- $L^{n+1} = L^n \cdot L = L \cdot L^n$. [1]

On note alors

$$L^\star = \bigcup_{n\in\mathbb{N}} L^n \qquad \text{et} \qquad L^+ = \bigcup_{n\in\mathbb{N}} L^n.$$

Remarque :
On a $\Sigma^\star = \Sigma^{\star\star}$. On notera donc $\Sigma^\star$ dans tous les cas.

Remarque :
Si $\varepsilon \in L$, alors $L^\star = L^+$. En effet, $L^\star = L^+ \cup \{\varepsilon\}$.

Remarque :
On nomme l'application $L \mapsto L^\star$ l'*étoile de Kleene*.

Remarque :
Avec L et K deux alphabets, on a

$$|L \cdot K| \leqslant |L|\,|K|.$$

En effet, avec $K = \{a, aa\}$, on a $K^2 = \{aa, aaa, aaaa\}$.

Exemple :
Avec $L = \{a, ab\}$, on a
- $L^1 = L$;
- $L^2 = \{aa,\ aab,\ aba\,abab\}$;
- $L^\star \supseteq \{\varepsilon,\ a,\ ab,\ aa, \ldots, aaaaaaa\ldots\}$.

Définition : Soit Σ un alphabet. On appelle *ensemble des langages réguliers*, noté LR. Le plus petit ensemble tel que
- $\varnothing \subset \mathrm{LR}$;
- Si $a \in \Sigma$, alors $a \in \mathrm{LR}$;
- Si $L_1 \in \mathrm{LR}$ et $L_2 \in \mathrm{LR}$, on a $L_1 \cup L_2 \in \mathrm{LR}$;
- Si $L_1 \in \mathrm{LR}$ et $L_2 \in \mathrm{LR}$, on a $L_1 \cdot L_2 \in \mathrm{LR}$;
- Si $L \in \mathrm{LR}$, alors $L^\star \in \mathrm{LR}$.

Exemple : — Soit $\Sigma = \{t, o\}$. Est-ce que $\{toto\} \in \mathrm{LR}$? On sait déjà que $\{t\}$ et $\{o\}$ sont déjà des langages réguliers. Or,

$$\{toto\} = \{t\} \cdot \{o\} \cdot \{t\} \cdot \{o\}.$$

1. La deuxième égalité est assurée par l'associativité de l'opération $\cdot$.

Et donc $\{toto\} \in \mathrm{LR}$.

— On a $\{\varepsilon\} = \varnothing^\star \in \mathrm{LR}$.

— On a

$$\left\{ \begin{array}{cccc} x_{11} & x_{12} & \cdots & x_{1,n_1} \\ x_{21} & x_{22} & \cdots & x_{2,n_2} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n_m} \end{array} \right\} \in \mathrm{LR}$$

car $\{x_{i,1}, x_{i,2}, \ldots, x_{i,n_i}\} \in \mathrm{LR}$ et c'est stable par union finie.

— Avec $\Sigma = \{a, b\}$, on a

$$L = \{\varepsilon,\ a,\ aa,\ aaa, \ldots\} = \underbrace{\Big(\overbrace{\{a\}}^{\in \mathrm{LR}}\Big)^\star}_{\in \mathrm{LR}}.$$

— $\Sigma^\star \in \mathrm{LR}$.

— Contre-exemple : $\{a^n \mid a \text{ premier}\} \notin \mathrm{LR}$.

1.3.2 Expressions régulières

On représente $\varnothing$ l'ensemble vide, _|_ l'union de deux expressions régulières, _ · _ la concaténation de deux expression régulières, et, _⋆ l'étoile de Klein pour les expressions régulières.

On cherche à représenter informatiquement ces expressions à l'aide d'une règle de construction nommée.

Définition : Étant donné un alphabet Σ, on définit $\mathrm{Reg}(\Sigma)$ défini par induction nommée à partir des règles :

— $\varnothing\big|^0$;
— $\star\big|^1$;
— $|\,\big|^2$;
— $L\big|^1_\Sigma$;
— $\cdot\,\big|^2$;
— $\varepsilon\big|^0$.

Ces règles peuvent être définies en OCamL (douteux) de la façon suivante :

```
1 type regex =
2   | ∅
3   | |  of regex * regex
4   | ·  of regex * regex
5   | ⋆  of regex
6   | L of char
7   | ε
```

Code 1.2 – Règles des expressions régulières en OCamL

On peut donc écrire

$$((\varnothing^\star) \mid (a \cdot b)) \quad \longrightarrow \quad |(\star(\varnothing()), \cdot(L(a), L(b))).$$

A-t-on $|(L(a), L(b)) =^? |(L(b), L(a))$? Non, sinon on risque d'avoir une boucle infinie au moment de l'évaluation de cette expression.

On peut simplifier la notation : au lieu d'écrire $|(\star(\varnothing()), \cdot(L(a), L(b)))$, on note $\varnothing^\star \mid (a \cdot b)$.

Définition : On définit

$$\begin{aligned}\mathscr{L} : \mathrm{Reg}(\Sigma) &\longrightarrow \wp(\Sigma^\star)\\ \varnothing &\longmapsto \varnothing\\ a &\longmapsto \{a\}\\ \varepsilon &\longmapsto \{\varepsilon\}\\ e_1 \cdot e_2 &\longmapsto \mathscr{L}(e_1)\cdot\mathscr{L}(e_2)\\ e_1 \mid e_2 &\longmapsto \mathscr{L}(e_1)\cup\mathscr{L}(e_2)\\ e^\star &\longmapsto \mathscr{L}(e)^\star\end{aligned}$$

Exemple :
Deux expressions régulières peuvent donner le même langage : on a $\mathscr{L}(\varnothing) = \varnothing$ mais également $\mathscr{L}(\varnothing \mid \varnothing) = \mathscr{L}(\varnothing)\cup\mathscr{L}(\varnothing) = \varnothing\cup\varnothing = \varnothing$. De même, $\mathscr{L}\big(a \mid (b\cdot b^\star)\big) = \{a, b, bb, bbb, \ldots\} = \mathscr{L}\big((bb^\star) \mid a\big)$.

Définition : On définit sur $\mathrm{Reg}(\Sigma)$ la fonction "vars" définie comme

$$\begin{aligned}\mathrm{vars} : \mathrm{Reg}(\Sigma) &\longrightarrow \wp(\Sigma)\\ \varnothing &\longmapsto \varnothing\\ \varepsilon &\longmapsto \varnothing\\ a \in \Sigma &\longmapsto \{a\}\\ e_1 \cdot e_2 &\longmapsto \mathrm{vars}(e_1)\cup\mathrm{vars}(e_2)\\ e_1 \mid e_2 &\longmapsto \mathrm{vars}(e_1)\cup\mathrm{vars}(e_2)\\ e^\star &\longmapsto \mathrm{vars}(e).\end{aligned}$$

Propriété : Un langage L est régulier si et seulement s'il existe $e \in \mathrm{Reg}(\Sigma)$ telle que $\mathscr{L}(e) = L$.

Preuve : "$\Longleftarrow$" Montrons que, pour toute expression régulière $e \in \mathrm{Reg}(\Sigma)$, P_e : "le langage $\mathscr{L}(e)$ est régulier." On procède par induction.

- $P_\varnothing : \mathscr{L}(\varnothing) = \varnothing \in \mathrm{LR}$;
- $P_\varepsilon : \mathscr{L}(\varepsilon) = \{\varepsilon\} = \varnothing^\star \in \mathrm{LR}$;
- Soit $a \in \Sigma$. Montrons $P_a : \mathscr{L}(a) = \{a\} \in \mathrm{LR}$;
- Soient e_1 et e_2 deux expressions régulières telles que P_{e_1} et P_{e_2} soient vrais. Montrons $P_{e_1\cdot e_2} : \mathscr{L}(e_1\cdot e_2) = \underset{\in \mathrm{LR}}{\mathscr{L}(e_1)}\cdot\underset{\in \mathrm{LR}}{\mathscr{L}(e_2)} \in \mathrm{LR}$
- Soient e_1 et e_2 deux expressions régulières telles que P_{e_1} et P_{e_2} soient vrais. Montrons $P_{e_1\mid e_2} : \mathscr{L}(e_1 \mid e_2) = \underset{\in \mathrm{LR}}{\mathscr{L}(e_1)}\cup\underset{\in \mathrm{LR}}{\mathscr{L}(e_2)} \in \mathrm{LR}$

"$\Longrightarrow$" Montrons que, pour tout langage régulier L, il existe une expression régulière e de l'alphabet Σ telle que $\mathscr{L}(e) = L$. Soit X l'ensemble des langages L tels qu'il existent une expression régulière e de l'alphabet Σ telle que $\mathscr{L}(e) = L$. On a

$$X \supseteq \{\underset{\mathscr{L}(\varnothing)}{\varnothing}\} \cup \{\{\underset{\mathscr{L}(a)}{a}\} \mid a \in \Sigma\}.$$

De plus, si deux langages L_1 et L_2 sont dans X, alors il existent e_1 et e_2 deux expressions régulières de Σ telle que $\mathscr{L}(e_1) = L_1$ et $\mathscr{L}(e_2) = L_2$. Or, $\mathscr{L}(e_1 \mid e_2) = \mathscr{L}(e_1) \cup \mathscr{L}(e_2) = L_1 \cup L_2$ et donc $L_1 \cup L_2$.
De même pour $L_1 \cdot L_2$.
Si un langage L est dans X, alors il existe une expression régulière e d'un alphabet Σ telle que $\mathscr{L}(e) = L$, alors $\mathscr{L}(e^\star) = L^\star \in X$.
X contient les langages $\varnothing$ et $\{a\}$ (avec $a \in \Sigma$) et X est stable par $\cup$, $\cdot$ et $\star$. Or, LR est défini comme le plus petit ensemble vérifiant les propriétés et donc $\mathrm{LR} \subseteq X$.

□

Remarque (Notation) :
Les notations $\mathrm{Reg}(\Sigma)$ et $\mathrm{Regexp}(\Sigma)$ sont équivalentes.

1.4 Automates finis (sans ε-transitions)

On considère l'automate représenté par les états suivants. L'entrée est représentée par la flèche sans nœud de départ et la sortie par celle sans nœud d'arrivée.

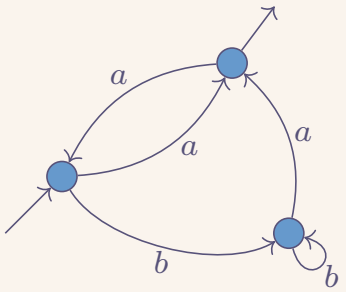


Figure 1.2 – Exemple d'automate

On représente une séquence d'état par un mot comme $aaba$ (qui correspond à une séquence valide) ou $bbab$ (qui n'est pas valide).

1.4.1 Définitions

Définition : Un *automate fini* (sans ε-transition) est un quintuplet $(Q, \Sigma, I, F, \delta)$ où
- Q est un ensemble d'états ;
- Σ est son alphabet de travail ;
- $I \subseteq Q$ est l'ensemble des états initiaux ;
- $F \subseteq Q$ est l'ensemble des états finaux.
- $\delta \subseteq Q \times \Sigma \times Q$.

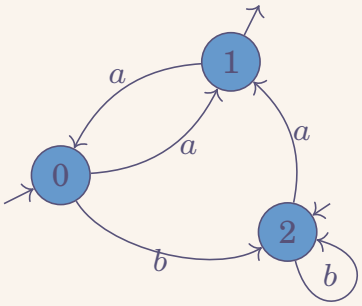


FIGURE 1.3 – Exemple d'automate (2)

EXEMPLE :
L'automate ci-dessus est donc représenté mathématiquement par

$$\Big(\underbrace{\{0,1,2\}}_{Q},\underbrace{\{a,b\}}_{\Sigma},\underbrace{\{0,2\}}_{I},\underbrace{\{1\}}_{F},\underbrace{\{(0,a,1),(0,b,2),(1,a,0),(2,a,1),(2,b,2)\}}_{\delta}\Big).$$

Définition : On dit d'une suite $(q_0, q_1, q_2, \dots, q_n) \in Q^{n+1}$ qu'elle est *une suite de transitions de l'automate* $\mathcal{A} = (Q, \Sigma, I, F, \delta)$ dès lors qu'il existe $(a_1, a_2, \dots, a_n) \in \Sigma^n$ tels que, pour out $i \in [\![1, n]\!]$, $(q_{i-1}, a_i, q_i) \in \delta$. On note parfois une telle suite de transition par

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} q_3 \to \cdots \to q_n - 1 \xrightarrow{a_n} q_n.$$

EXEMPLE :
$1 \xrightarrow{a} 0 \xrightarrow{b} 2 \xrightarrow{b} 2$ est une suite de transitions de l'automate ci-avant.

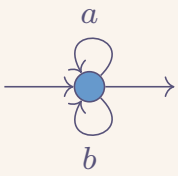
Définition : On dit qu'une suite $(q_0, q_1, \dots, q_n)$ est une *exécution* dans l'automate $\mathcal{A} = (Q, \Sigma, I, F, \delta)$ si c'est une suite de transition de A telle que $q_0 \in I$. On dit également qu'elle est *acceptante* si $q_n \in F$.

Lorsque $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \to \cdots \to q_{n-1} \xrightarrow{a_n} q_n$ est une suite de transitions d'un automate $\mathcal{A}$, on dit que le mot $a_1 a_2 \dots a_n$ est l'*étiquette* de cette transition.

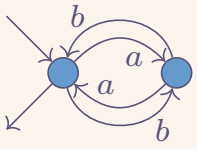
Définition (Langage reconnu par un automate) **:** On dit qu'un mot $w \in \Sigma^\star$ est *reconnu* par un automate $\mathcal{A} = (Q, \Sigma, I, F, \delta)$ s'il est l'étiquette d'une exécution acceptante de $\mathcal{A}$. On note alors $\mathcal{L}(\mathcal{A})$ l'ensemble des mots reconnus par l'automate $\mathcal{A}$.

Définition : On dit d'un langage L qu'il est *reconnaissable* s'il existe un automate $\mathcal{A}$ tel que $\mathcal{L}(\mathcal{A}) = L$. On note $\mathrm{Rec}(\Sigma)$ l'ensemble des mots reconnaissables.

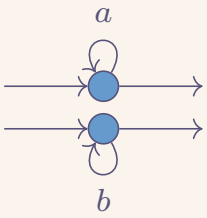
Exemple :
Avec $\Sigma = \{a, b\}$, on cherche $\mathcal{A}$ tel que
— $\mathcal{L}(\mathcal{A}) = \Sigma^\star$
— $\mathcal{L}(\mathcal{A}) = (\Sigma^2)^\star$



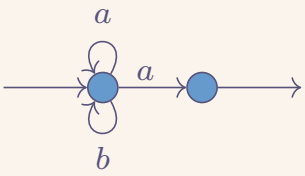
(a) Automate minimal pour $\mathcal{L}(\mathcal{A}) = \Sigma^\star$



(b) Automate minimal pour $\mathcal{L}(\mathcal{A}) = (\Sigma^2)^\star$



(c) Automate minimal pour $\mathcal{L}(\mathcal{A}) = \{a\}^\star \cup \{b\}^\star$



(d) Automate minimal pour $\mathcal{L}(\mathcal{A}) = \Sigma^\star \cdot \{a\}$

Figure 1.4 – Automates minimaux pour différentes valeurs de $\mathcal{L}(\mathcal{A})$

Définition (Automate déterministe) **:** On dit d'un automate $\mathcal{A} = (Q, \Sigma, I, F, \delta)$ qu'il est *déterministe* si
1. $|I| = 1$;
2. $\forall (q, q_1, q_2) \in Q^3$, $\forall a \in \Sigma$, $(q, a, q_1) \in \delta$ et $(q, a, q_2) \in \delta \implies q_1 = q_2$;

Remarque :
(2) est équivalent à

$$\forall (q, a) \in Q \times \Sigma,\ \big|\{q' \in Q \mid (q, a, q') \in \delta\}\big| \leqslant 1.$$

Définition (Automate complet) **:** On dit d'un automate $\mathcal{A} = (Q, \Sigma, I, F, \delta)$ qu'il est *complet* si

$$\forall (q, a) \in Q \times \Sigma,\ \exists q' \in Q,\ (q, a, q') \in \delta.$$

Exemple :
Les automates ci-avant sont
(a) complet et déterministe ;
(b) complet et déterministe ;
(c) non complet et non déterministe ;
(d) non complet et non déterministe.

1.4.2 Transformations en automates équivalents

On peut représenter le langage utilisé par l'automate ci-dessous avec une expression régulière : $a^\star \cdot (a \mid bab)$. L'arbre ci-dessous n'est pas déterministe. On cherche à le rendre déterministe : pour cela, on trace un arbre contenant les nœuds accédés en fonctions de l'expression lue.

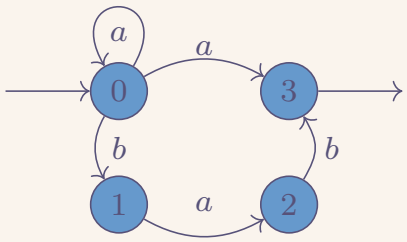


FIGURE 1.5 – Automate non déterministe ayant pour expression régulière $a^\star \cdot (a \mid bab)$

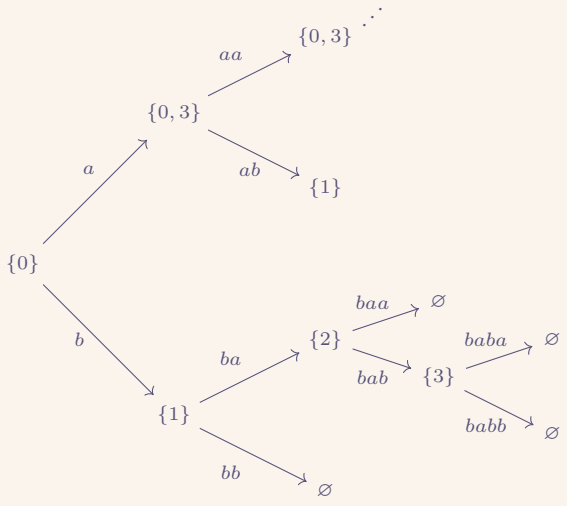


FIGURE 1.6 – Nœuds possibles par rapport à l'expression lue

À l'aide de cet arbre, on peut trouver un automate déterministe équivalent à l'automate précédent.

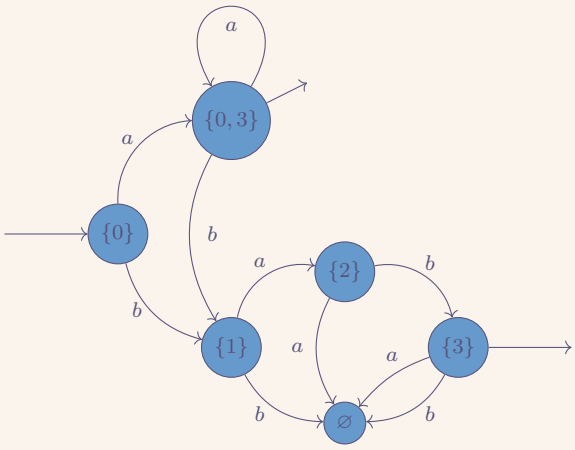


FIGURE 1.7 – Automate déterministe ayant pour expression régulière $a^\star \cdot (a \mid bab)$

Définition : On dit de deux automates $\mathcal{A}$ et $\mathcal{A}'$ qu'ils sont *équivalents* si $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

Théorème : Pour tout automate $\mathcal{A}$, il existe un automate déterministe $\mathcal{A}'$ tel que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

Preuve :
Soit $\mathcal{A} = (\mathbb{Q}, \Sigma, I, F, \delta)$ un automate. On pose $\Sigma' = \Sigma$, $\mathbb{Q}' = \wp(\mathbb{Q})$, $I' = \{I\}$, $F = \{Q \in \wp(\mathbb{Q}) \mid Q \cap F \neq \varnothing\}$ et

$$\delta' = \left\{(Q, a, Q') \in \mathbb{Q}' \times \Sigma \times \mathbb{Q}' \;\middle|\; Q' = \left\{q' \in \mathbb{Q} \mid \exists q \in Q,\ (q, a, q') \in \delta\right\}\right\}.$$

On pose alors l'automate $\mathcal{A}' = (\mathbb{Q}', \Sigma', I', F', \delta')$. Montrons que $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$. On procède par double-inclusion.

"$\subseteq$" Soit $w \in \mathcal{L}(\mathcal{A})$. Il existe donc une exécution acceptante $I \ni q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} q_2 \to \cdots \to q_{n-1} \xrightarrow{w_n} q_n \in F$ telle que $w = w_1 w_2 \dots w_n$. On pose $Q_0 = I$, et, pour tout entier $i \in [\![1, n]\!]$,

$$Q_i = \left\{q' \in \mathbb{Q} \mid \exists q \in Q_{i-1},\ (q, w_i, q') \in \delta\right\}.$$

Remarquons que, pour tout entier $i \in [\![1, n]\!]$, on a $(Q_{i-1}, w_i, Q_i) \in \delta'$. On a donc $I' \ni Q_0 \xrightarrow{w_1} Q_1 \to \cdots \to Q_{n-1} \xrightarrow{w_n} Q_n$ est une exécution de $\mathcal{A}'$. Montrons que, pour tout $i \in [\![0, n]\!]$, on a $q_i \in Q_i$ par récurrence finie.

- $q_0 \in I = Q_0$.
- Soit $p < n$ tel que $q_p \in Q_p$ alors q_{p+1} est tel que $(q_p, w_{p+1}, q_{p+1}) \in \delta$ et q_{p+1} est tel qu'il existe $q \in Q_p$ tel que $(q, w_{p+1}, q_{p+1}) \in \delta$. On en déduit $q_{p+1} \in Q_{p+1}$.

On a donc $q_n \in Q_n$ et $q_n \in F$ donc $Q_n \cap F \neq \varnothing$ et donc $Q_n \in F'$. L'exécution $Q_0 \xrightarrow{w_1} Q_1 \to \cdots \to Q_{n-1} \xrightarrow{w_n} Q_n$ est donc acceptante dans $\mathcal{A}'$ et donc $w = w_1 \dots w_n \in \mathcal{L}(\mathcal{A}')$.

"$\supseteq$" Soit $w \in \mathcal{L}(\mathcal{A}')$. Soit donc $I' \ni Q_0 \xrightarrow{w_1} Q_1 \to \cdots \to Q_{n-1} \xrightarrow{w_n} Q_n \in F'$ une exécution acceptante de w dans $\mathcal{A}'$. $Q_n \cap F \neq \varnothing$. Soit donc $q_n \in Q_n \cap F$. Soit $q_{n-1} \in Q_{n-1}$ tel que $(q_{n-1}, w_n, q_n) \in \delta$ (par définition de $(Q_{n-1}, w_n, Q_n) \in \delta'$). "De proche en proche," il existe $q_0, q_1, \dots, q_{n-2}$ tels que, pour tout $i \in [\![1, n]\!]$, $(q_{i-1}, w_i, q_i) \in \delta$ et $q_i \in Q_i$. Or, $q_0 \in Q_0 \in I' = \{I\}$ donc $Q_0 = I$ et donc $q_0 \in I$. On rappelle que $q_n \in F$. On en déduit donc que $q_0 \xrightarrow{w_1} q_1 \to \cdots \to q_{n-1} \xrightarrow{w_n} q_n$ une exécution acceptante da ns $\mathcal{A}$ et donc $w \in \mathcal{L}(\mathcal{A}')$.

□

Pour comprendre la construction de l'automate dans la preuve, on fait un exemple. On consi-

dère l'automate non-déterministe ci-dessous.

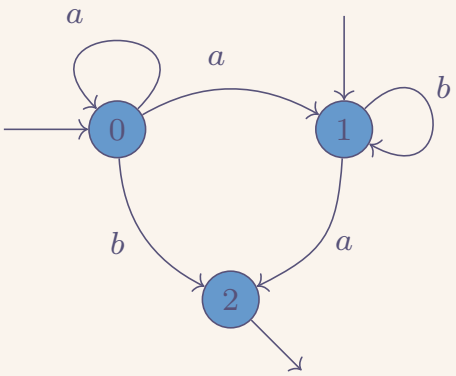


FIGURE 1.8 – Automate non déterministe

On construit la *table de transition* :

	a	b
$\varnothing$	$\varnothing$	$\varnothing$
$\{0\}$	$\{0,1\}$	$\{2\}$
$\{1\}$		

TABLE 1.1 – Table de transition de l'automate ci-avant

À faire : Finir la table de transition

REMARQUE :
L'automate $\mathcal{A}'$ construit dans le théorème précédent est complet mais son nombre de nœud suit une exponentielle.

Propriété : Soit $\mathcal{A}$ un automate fini à n états. Il existe un automate $\mathcal{A}'$ ayant $n+1$ états tel que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$ avec $\mathcal{A}'$ complet.

Preuve :
Soit $\mathcal{A} = (\mathbb{Q}, \Sigma, I, F, \delta)$ un automate à n états. Soit $P \notin \mathbb{Q}$. On pose $\Sigma' = \Sigma$, $\mathbb{Q}' = \mathbb{Q} \cup \{P\}$, $I' = I$, $F' = F$ et

$$\delta' = \delta \cup \left\{ (q, \ell, P) \in \mathbb{Q}' \times \Sigma \times \{P\} \;\middle|\; \forall q' \in \mathbb{Q}',\ (q, \ell, q') \notin \delta \right\}.$$

Montrons que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. À faire : Preuve à faire

□

Définition : Soit $\mathcal{A} = (\mathbb{Q}, \Sigma, I, F, \delta)$ un automate. On dit d'un état $q \in \mathbb{Q}$ qu'il est

- *accessible* s'il existe une exécution $I \ni q_0 \xrightarrow{w_1} q_1 \to \cdots \to q_{n-1} \xrightarrow{w_n} q_n = q$.
- *co-accessibles*'il existe une suite de transitions $q \xrightarrow{w_1} q_1 \to \cdots \to q_{n-1} \xrightarrow{w_n} q_n \in F$.

Dans l'automate ci-dessous, l'état 0 n'est pas accessible et l'état 2 n'est pas co-accessible.

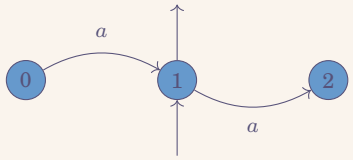


FIGURE 1.9 – Non-exemples d'états accessibles et co-accessibles

Définition : On dit d'un automate $\mathcal{A}$ qu'il est *émondé* dès lors que chaque état est accessible et co-accessible.

Propriété : Soit $\mathcal{A}$ un automate. Il existe $\mathcal{A}'$ un automate émondé tel que $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$.

Preuve :
Soit $\mathcal{A} = (\mathbb{Q}, \Sigma, I, F, \delta)$. On pose $\Sigma' = \Sigma$, $\mathbb{Q}' = \{q \in \mathbb{Q} \mid q \text{ accessible ou co-accessible }\}$, $I' = I \cap R'$, $F' = F \cap R'$ et

$$\delta = \{(q, \ell, q') \in \mathbb{Q}' \times \Sigma \times \mathbb{Q}' \mid (q, \ell, q') \in \delta\} = (\mathbb{Q}', \Sigma, \mathbb{Q}') \cap \delta.$$

Montrons que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. On procède par double inclusion. On vérifie aisément que $\mathcal{L}(A') \subseteq \mathcal{L}(\mathcal{A})$. On montre maintenant $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$. Soit $w \in \mathcal{L}(\mathcal{A}')$. Soit $q_0, \dots, q_n$ tels que $q_0 \xrightarrow{w_1} q_1 \to \cdots \to q_{n-1} \xrightarrow{w_n} q_n$ est une exécution acceptante. Or, pour tout $i \in [\![0, n]\!]$, q_i est accessible et co-accessible donc $q_i \in \mathbb{Q}'$. De plus, $q_0 \in I'$ et $q_n \in F'$. De plus, pour tout $i \in [\![0, n-1]\!]$, $(q_i, w_{i+1}, q_{i+1}) \in \delta'$. Donc, $q_0 \xrightarrow{w_1} q_1 \to \cdots \to q_{n-1} \xrightarrow{w_n} q_n$ est une exécution acceptante de $\mathcal{A}'$. □

Parfois, on veut pouvoir "sauter" d'un état à un autre dans un automate. On utilise pour cela des ε-transitions.

1.5 Automates finis avec ε-transitions

Définition : On dit d'un automate sur l'alphabet $\Sigma \cup \{\varepsilon\}$ que c'est un *automate avec ε-transition*.

EXEMPLE :
L'automate ci-dessous est un automate avec ε-transitions.

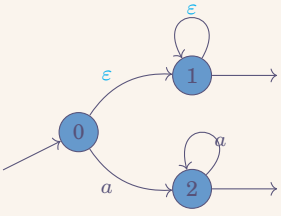


FIGURE 1.10 – Exemple d'automate avec ε-transition

Définition : Soit $w \in (\Sigma \cup \{\varepsilon\})^\star$. On définit alors $\tilde{w}$ le mot obtenu en supprimant les occurrences de ε dans w.

Exemple :
Avec $\Sigma = \{a, b\}$ et $w = ab\varepsilon aa\varepsilon\varepsilon a$, on a $\tilde{w} = abaaa$.

On a également $\tilde{\varepsilon} = \varepsilon$; pour deux mots w_1 et w_2, on a $\widetilde{w_1 \cdot w_2} = \tilde{w}_1 \cdot \tilde{w}_2$; on a également $\tilde{a} = a$ pour $a \in \Sigma$ et $\tilde{\varepsilon} = \varepsilon$.

Définition : Soit $\mathcal{A}$ un automate avec ε-transition. On pose $\tilde{\mathcal{L}}(\mathcal{A})$ est le langage de l'automate sur l'alphabet $\Sigma \cup \{\varepsilon\}$. On appelle *langage* de $\mathcal{A}$, l'ensemble À faire : retrouver la formule.

Exemple :
On peut trouver un automate reconnaissant la concaténation des langages des deux automates ci-dessous.

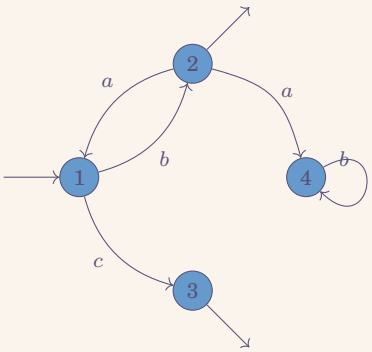


Figure 1.11 – Automate reconnaissant le langage $(ba)^\star \cdot (c \mid a(ba)^\star)$

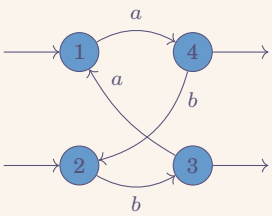


Figure 1.12 – Automate reconnaissant le langage $(a \mid baa)(bbaa)^\star \mid (b \mid abb)(aabb)^\star$

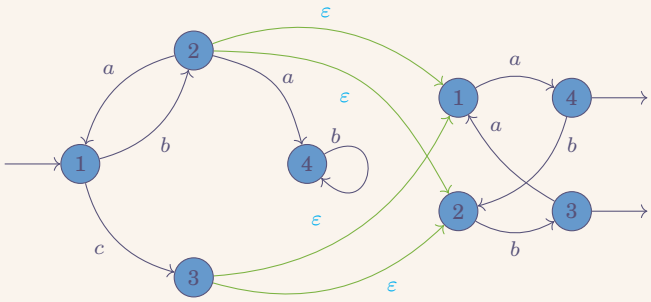


FIGURE 1.13 – Automate reconnaissant la concaténation des deux précédents

1.5.1 Cloture par concaténation

Propriété : Soient $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ et $\mathcal{A}' = (\Sigma', Q', I', F', \delta')$ deux automates avec $Q \cap Q' = \varnothing$. Alors $\mathcal{L}(\mathcal{A}) \cdot \mathcal{L}(\mathcal{A}')$ est un langage reconnaissable. Il est d'ailleurs reconnu par l'automate $\mathcal{A}^{\cdot} = (\Sigma^{\cdot}, Q^{\cdot}, I^{\cdot}, F^{\cdot}, \delta^{\cdot})$ défini avec $\Sigma^{\cdot} = \Sigma \cup \Sigma'$, $Q^{\cdot} Q \cup Q'$, $I^{\cdot} = I$, $F^{\cdot} = F'$ et

$$\delta^{\cdot} = \{(q, \varepsilon, q) \mid q \in F,\ q' \in I'\}.$$

Preuve :
Montrons que $\mathcal{L}(\mathcal{A}^{\cdot}) = \mathcal{L}(\mathcal{A}) \cdot \mathcal{L}(\mathcal{A}')$. On procède par double inclusion.

"$\subseteq$" Soit $w \in \mathcal{L}(\mathcal{A}^{\cdot})$ et soit

$$Q \supseteq I = I^{\cdot} \ni q_0 \xrightarrow{u_1} q_1 \xrightarrow{u_2} q_2 \to \cdots \to q_{n-1} \xrightarrow{u_n} q_n \in F^{\cdot} = F' \subseteq Q$$

une exécution acceptante de $\mathcal{A}^{\cdot}$ telle que $\tilde{u} = w$. On pose $i_0 = \min\{i \in [\![1, n]\!] \mid q_i \in Q\}$. On a alors, $\forall i \in [\![1, i_0 - 1]\!]$, $q_i \in Q$. Montrons que $\forall i \in [\![i_0, n]\!]$, $q_i \in Q'$ par récurrence finie. On a $q_{i_0} \in Q'$. De plus, si $q_i \in Q'$ et $(q_i, u_{i+1}, q_{i+1}) \in \delta^{\cdot}$ donc $q_{i+1} \in Q'$. Inspectons $(Q \ni q_{i_0 - 1}, u_{i_0}, q_{i_0} \in Q') \in \delta^{\cdot}$. On sait que $(q_{i_0-1}, u_{i_0}, q_{i_0}) \notin \delta$ car $q_{i_0} \in Q'$; de même, $(q_{i_0-1}, u_{i_0}, q_{i_0}) \notin \delta'$ car $q_{i_0 - 1} \in Q$ donc $(q_{i_0-1}, u_{i_0}, q_{i_0}) \in \{(q, \varepsilon, q') \mid q \in F n q' \in I'\}$. On a donc que $q_{i_0 - 1} \in F$ et $q_{i_0} \in I$. Ainsi

$$I \ni \underbrace{q_0 \xrightarrow{u_1} q_1 \to \cdots \to \overset{F}{\ni} q_i}_{} \xrightarrow{\varepsilon} \underbrace{\overset{I'}{\ni} q_{i_0} \xrightarrow{u_{i_0}} \to \cdots \to q_n}_{} \in F$$

est une exécution acceptante de $\mathcal{A}$ d'étiquette $\widetilde{u_1 \dots u_{i_0 - 1}}$.

est une exécution acceptante de $\mathcal{A}'$ d'étiquette $\widetilde{u_{i_0+1} \dots u_n}$.

donc $\widetilde{u_{|[\![1, i_0 - 1]\!]}} \in \mathcal{L}(\mathcal{A})$ et $\widetilde{u_{|[\![i_0+1, n]\!]}} \in \mathcal{L}(\mathcal{A}')$.

$$\begin{aligned} w = \tilde{u} &= u_{|[\![1, i_0 - 1]\!]} \widetilde{u_{i_0} \cdot u}_{|[\![i_0+1, n]\!]} \\ &= \widetilde{u_{|[\![1, i_0+1]\!]}} \cdot \widetilde{u_{|[\![i_0+1, n]\!]}} \end{aligned}$$

"$\supseteq$" Montrons que $\mathcal{L}(\mathcal{A}) \cdot \mathcal{L}(\mathcal{A}') \subseteq \mathcal{L}(\mathcal{A}^{\star})$. Soit $w \in \mathcal{L}(\mathcal{A}) \cdot \mathcal{L}(\mathcal{A}^{\cdot})$, il existe donc

$$I \ni q_0 \xrightarrow{u_1} q_1 \to \cdots \to q_n \in F$$

une exécution acceptante dans $\mathcal{A}$ d'étiquette $\widetilde{u_1 \dots u_n}$. Il existe également

$$I' \ni q_{n+1} \xrightarrow{u_{n+2}} q_{n+1} \to \cdots \to q_m \in F$$

une exécution acceptante dans $\mathcal{A}'$ d'étiquette $\widetilde{u_{n+2} \dots u_m}$. Or, $\delta \subseteq \delta^{\cdot}$ et $\delta' \subseteq \delta^{\cdot}$ donc $\forall i \in [\![1, n]\!]$, $(q_{i-1}, u_i, q_i) \in \delta^{\cdot}$ et $\forall i \in [\![n+2, m]\!]$, $(q_{i-1}, u_i, q_i) \in \delta^{\cdot}$. Or, $q_n \in F$ et $q_{n+1} \in I'$ donc $(q_n, \varepsilon, q_{n+1}) \in \delta^{\cdot}$. Finalement À faire : recopier.

□

1.5.2 Cloture par étoile

Propriété : Soit $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ un automate fini. Alors $\mathcal{L}(\mathcal{A})^\star$ est un automate reconnaissable, il est de plus reconnu par l'automate $\mathcal{A}_\star = (\Sigma_\star, Q_\star, I_\star, F_\star, \delta_\star)$ défini avec $\Sigma_\star = \Sigma$, $Q_\star = Q \cup \{V\}$ où $V \notin Q$. À faire : recopier ici...

Exemple :

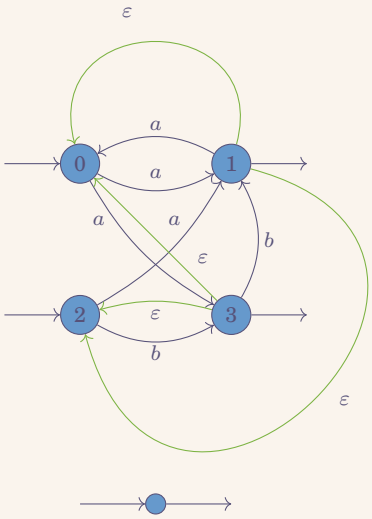


Figure 1.14 – Automate reconnaissant $\mathcal{L}(\mathcal{A})^\star$

1.5.3 Cloture par union

Propriété : Soit $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ et $\mathcal{A}' = (\Sigma', Q', I', F', \delta')$ deux automates avec $Q \cap Q' = \varnothing$. Alors, $\mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{A}')$ est un langage reconnaissable. Il est, de plus, reconnu par $\mathcal{A}^\cup = (\Sigma^\cup, Q^\cup, I^\cup, F^\cup, \delta^\cup)$ avec $\Sigma^\cup = \Sigma \cup \Sigma'$, $Q^\cup = Q \cup Q'$, $I^\cup = I \cup I'$, $F^\cup = F \cup F'$ et $\delta^\cup = \delta \cup \delta'$.

Preuve :
Montrons que $\mathcal{L}(\mathcal{A}^\cup) \subseteq \mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{A}')$. Supposons, sans perte de généralité que les automates $\mathcal{A}$ et $\mathcal{A}'$ sont sans ε-transitions. Soit $w \in \mathcal{L}(\mathcal{A}^\cup)$. Il existe une exécution acceptante

$$I^\cup \ni q_0 \xrightarrow{w_1} q_1 \to \cdots \to q_{n-1} \xrightarrow{w_n} q_n \in F^\cup$$

avec $w = w_0 \dots w_n$.

Montrons que, en supposant $q_0 \in I$ sans perte de généralité, $\forall i \in [\![1, n]\!]$, $q_i \in Q$ de proche en proche.

On a donc $q_n \in Q \cap F^{\cup} = F$ et on a alors, pour tout $i \in [\![1, n]\!]$, $(q_{i-1}, w_i, q_i) \in \delta^{\cup}$. Or, $q_{i-1} \in Q$ et $(q_{i-1}, w_i, q_i) \in \delta'$ donc $(q_{i-1}, w_i, q_i) \in \delta$.

Finalement, $q_0 \xrightarrow{w_1} q_1 \to \cdots \to q_n$ est un exécution acceptante de $\mathscr{A}$ donc $w \in \mathscr{L}(\mathscr{A})$. □

Remarque :
Pour tout $a \in \Sigma$, $\{a\}$ est reconnaissable : par exemple,

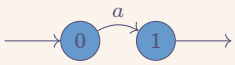


Figure 1.15 – Automate reconnaissant $\{a\}$ avec $a \in \Sigma$

Remarque :
$\varnothing$ est reconnaissable : par exemple,

Figure 1.16 – Automate reconnaissant $\varnothing$

Propriété : De ce qui précède, on en déduit que l'ensemble des langages reconnaissables par automates avec ε-transition est au moins l'ensemble des langages réguliers.

Théorème : Si $\mathscr{A}$ est un automate avec ε-transitions, alors il existe un automate $\mathscr{A}'$ sans ε-transition tel que $\mathscr{L}(\mathscr{A}) = \mathscr{L}(\mathscr{A}')$.

Exemple :

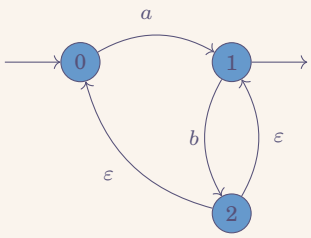


Figure 1.17 – Automate avec ε-transition

L'automate avec ε-transition ci-dessus peut être transformé en automate sans ε-transition comme celui ci-dessous.

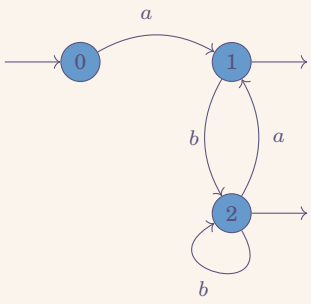


FIGURE 1.18 – Automate sans ε-transition

Propriété : Soit $\mathcal{A} = (\Sigma, \mathbb{Q}, I, F, \delta)$ un automate avec ε-transitions. Soit $q_r \in \mathbb{Q}$ un état de l'automate. Alors, l'automate $\mathcal{A}' = (\Sigma', \mathbb{Q}', I', F', \delta')$ défini par $\Sigma' = \Sigma$, $\mathbb{Q}' = \mathbb{Q}$, $I' = I$,

$$F' = F \cup \begin{cases} \{q \in \mathbb{Q} \mid (q, \varepsilon, q_r) \in \delta\} & \text{si } q_r \in F \\ \varnothing & \text{sinon,} \end{cases}$$

$$\begin{aligned} \delta' = (\delta \setminus \{(q, \varepsilon, q_r) \in \delta \mid q \in \mathbb{Q}\}) \\ \cup \{(q, a, q') \mid (q, \varepsilon, q_r) \in \delta \text{ et} (q_r, a, q') \in \delta \text{ et } a \in \Sigma\} \\ \cup \{(q, \varepsilon, q') \mid (q, \varepsilon, q_r) \in \delta \text{ et } (q_r, \varepsilon, q') \in \delta \text{ et } q_r \neq q' \in \mathbb{Q}\}, \end{aligned}$$

est tel que

- il n'y a pas d'ε-transitions entrant en q_r ;
- $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$;
- si $q \in \mathbb{Q}$ n'a pas d'ε-transition entrante dans $\mathcal{A}$, il n'en a pas dans $\mathcal{A}'$.

Algorithme 1.1 Suppression des ε-transitions

Entrée un automate $\mathcal{A} = (\Sigma, \mathbb{Q}, I, F, \delta)$
Sortie un automate équivalent à $\mathcal{A}$ sans ε-transitions

$\delta' \leftarrow \delta$
$F' \leftarrow F$
$\mathbb{Q}' \leftarrow \mathbb{Q}$
tant que il existe $q \in \mathbb{Q}'$ avec une ε-transition entrante dans δ' **faire**
  $(\Sigma', \mathbb{Q}', I', F', \delta') \leftarrow$ résultat de la proposition précédente avec $q_R = q$
retourner $(\Sigma', \mathbb{Q}', I', F', \delta')$

On a donc démontré que tout langage régulier peut être reconnu par un automate.

EXEMPLE :
On veut, par exemple, reconnaître le langage $(a \cdot b)^\star \cdot (a \mid b)$. À faire : Faire les automates

1.6 Théorème de KLEENE

On s'intéresse à un autre ensemble de langages, les *langages locaux*.

1.6.1 Langages locaux

Définitions, propriétés

Définition (lettre préfixe, lettre suffixe, facteur de taille 2, non facteur) **:** Soit L un langage. On note l'ensemble $P(L)$ des lettres préfixes défini comme

$$\begin{aligned} P(L) &= \{\ell \in \Sigma \mid \exists w \in \ell, \exists v \in \Sigma^\star, w = \ell \cdot v\} \\ &= \{\ell \in \Sigma,\ \{\ell\} \cdot \Sigma^\star \cap L = \varnothing\}. \end{aligned}$$

On note l'ensemble $S(L)$ des lettres suffixes défini comme

$$S(L) = \{w_{|w|} \mid w \in L\} = \{\ell \in \Sigma \mid \Sigma^\star \cdot \{\ell\} \cap L \neq \varnothing\}.$$

On note l'ensemble $F(L)$ des facteurs de taille 2 défini comme

$$F(L) = \{\ell_1 \cdot \ell_2 \in \Sigma^2 \mid \Sigma^\star \cdot \{\ell_1, \ell_2\} \cdot \Sigma^\star \cap L \neq \varnothing\}.$$

On note l'ensemble $N(L)$ des non-facteurs défini comme

$$N(L) = \Sigma^2 \setminus F(L).$$

On défini également l'ensemble

$$\Lambda(L) = L \cap \{\varepsilon\}.$$

Définition : Soit L un langage. On définit le *langage local engendré* par L comme étant

$$\rho(L) = \Lambda(L) \cup \Big(P(L) \cdot \Sigma^\star \cap \Sigma^\star \cdot S(L)\Big) \setminus \Sigma^\star N(L) \Sigma^\star.$$

Exemple :
Avec $L = \{aab, \varepsilon\}$, on a $P(L) = \{a\}$, $S(L) = \{b\}$, $F(L) = \{aa, ab\}$, $N(L) = \{ba, bb\}$, $\Lambda(L) = \{\varepsilon\}$. Et donc, on en déduit que

$$\rho(L) = \{\varepsilon\} \cup \{ab\} \cup \{aab\} \cup \cdots \{\varepsilon\} \cup \{a^n \cdot b \mid n \in \mathbb{N}^\star\}.$$

Définition : Un langage est dit *local* s'il est son propre langage engendré i.e. $\rho(L) = L$.

Propriété : Soit L un langage. Alors, $\rho(L) \supseteq L$.

Preuve :
Soit $w \in L$. Montrons que $w \in \rho(L)$.

- Si $w = \varepsilon$, alors $\Lambda(L) = L \cap \{\varepsilon\} = \{\varepsilon\}$ donc $w \in \rho(L)$.
- Sinon, notons $w = w_1 w_2 \dots w_n$. On doit montrer que $w_1 \in P(L)$, $w_n \in S(L)$, et $\forall i \in [\![1, n-1]\!]$, $w_i w_{i+1} \in F(L)$. Par définition de ces ensembles, c'est vrai.

□

Propriété : Soit L de la forme

$$\Lambda \cup (P\Sigma^\star \cap \Sigma^\star S) \setminus (\Sigma^\star N \Sigma^\star)$$

avec $\Lambda \subseteq \{\varepsilon\}$, $P \subseteq \Sigma$, $S \subseteq \Sigma$, et $N \subseteq \Sigma^2$. Alors $\rho(L) = L$.

Preuve :
On a $L \subseteq \rho(L)$. Montrons donc $\rho(L) \subseteq L$.

- Montrons que $\Lambda(L) \subseteq L$.
  - Si $\Lambda(L) = \varnothing$, alors OK.
  - Sinon, $\Lambda(L) = \{\varepsilon\} = L \cap \{\varepsilon\}$ donc $\varepsilon \in L$ et donc $\varepsilon \in \Lambda$ parce que ce n'est possible.
- Montrons que $P(L) \subseteq P$. Soit $\ell \in P(L)$. Soient $v \in \Sigma^\star$, et $w \in L$ tels que $w = \ell v$. On a donc $w \notin \Lambda$, donc $w \in (P\Sigma^\star \cap \Sigma^\star S)$ et donc $\ell v = w \in P\Sigma^\star$ et donc $\ell \in P$.
- De même, $S(L) \subseteq L$
- À faire à la maison : $N \subseteq N(L)$ (ou $F(L) \subseteq F$)

□

Corollaire : On a $\rho^2 = \rho$.

À faire : Figure ensembles langages locaux, réguliers, ...

Preuve :
$\rho(\varnothing) = \varnothing$ et $\rho(\Sigma^\star) = \Sigma^\star$. □

Remarque :
Un langage L est local si et seulement s'il existe $S \subseteq \Sigma$, $P \subseteq \Sigma$, $N \subseteq \Sigma^2$ tel que

$$L \setminus \{\varepsilon\} = (P\Sigma^\star \cap \Sigma^\star S) \setminus \Sigma^\star N \Sigma^\star.$$

Exemple :
Le langage $L = \{a\}$ est local avec $S = \{a\}$, $P = \{a\}$, $F = \varnothing$ et $\Lambda = \varnothing$.

Le langage $L = \{a, ab\}$ est local avec $S = \{b, a\}$, $P = \{a\}$, $F = \{ab\}$ et $\Lambda = \varnothing$.

Le langage $L = (ab)^\star$ est local avec $S = \{b\}$, $P = \{a\}$, $F = \{ab, ba\}$. Soit $w \in \rho(L)$. Si $w = \varepsilon$, alors OK. Sinon, $w = abw_1$ et $w_1 \in \rho(L)$. Par récurrence, on montre que le langage est local.

Le langage $L = a \cdot (ab)^\star$ n'est pas local.

Stabilité

Intersection

Propriété : Si L_1 et L_2 sont deux langages locaux, alors $L_1 \cap L_2$ est un langage local.

Preuve :
Soit $L_1 = \Lambda_1 \cup (P_1\Sigma^\star \cap \Sigma^\star S_1) \setminus (\Sigma^\star N_1 \Sigma^\star)$, et $L_2 = \Lambda_2 \cup (P_2\Sigma^\star \cap \Sigma^\star S_2) \setminus (\Sigma^\star N_2 \Sigma^\star)$. On pose

$F_1 = \Sigma^2 \setminus N_1$ et $F_2 = \Sigma^2 \setminus N_2$. On pose alors $\Lambda_\cap = \Lambda_1 \cap \Lambda_2$; $P_\cap = P_1 \cap P_2$; $S_\cap = S_1 \cap S_2$; $F_\cap = F_1 \cap F_2$; $N_\cap = \Sigma^2 \setminus F_\cap$. On a

$$L_1 \cap L_2 = \Lambda_\cap \cup \left(P_\cap \Sigma^\star \cap \Sigma^\star S_\cap\right) \setminus \Sigma^\star N_\cap \Sigma^\star.$$

En effet,

$$\begin{aligned} L_1 \cap L_2 &= (\Lambda_1 \cap \Lambda_2) \\ &\quad \cap (\Lambda_1 \cap (P_2\Sigma^\star \cap \Sigma^\star S_2) \setminus \Sigma^\star N_2 \Sigma^\star) \\ &\quad \cap (((P_1\Sigma^\star \cap \Sigma^\star S_1) \setminus \Sigma^\star N_1 \Sigma^\star) \cap \Lambda_2) \\ &\quad \cap (((P_1\Sigma^\star \cap \Sigma^\star S_1) \setminus \Sigma^\star N_1 \Sigma^\star) \cap (P_2\Sigma^\star \cap (P_2\Sigma^\star \cap \Sigma^\star S_2) \setminus \Sigma^\star N_2 \Sigma^\star) \\ &= (\Lambda_1 \cap \Lambda_2)((P_1 \cap P_2)\Sigma^\star \cap \Sigma^\star(S_1 \cap S_2)) \setminus \Sigma^\star(N_1 \cap N_2)\Sigma^\star \end{aligned}$$

□

Union

CONTRE-EXEMPLE :
Avec $L_1 = ab$ et $L_2 = ba$, on a $\Lambda_1 = \Lambda_2 = \varnothing$, $P_1 = \{a\}$, $P_2 = \{b\}$, $S_1 = \{b\}$, $S_2 = \{b\}$, $F_1 = \{ab\}$ et $F_2 = \{ba\}$. Le langage $L_1 \cup L_2 = \{ab, ba\}$ n'est pas local : en effet, on a $\Lambda = \varnothing$, $P = \{a, b\}$, $S = \{a, b\}$, et $F = ab, ba$. Le mot aba est donc dans le langage local engendré.

On doit donc ajouter une contrainte afin d'éviter ce type de contre-exemples. L'intersection des alphabets est vide.

Propriété : Soient L_1 un langage local sur un alphabet Σ_1 et L_2 un langage local sur un alphabet Σ_2 avec $\Sigma_1 \cap \Sigma_2 = \varnothing$. Alors $L_1 \cup L_2$ est local.

Preuve :
Soient Λ_1, S_1, P_1, N_1, F_1 tels que L_1 soit défini par $(\Lambda_1, S_1, P_1, N_1, F_1)$. De même, soient Λ_2, S_2, P_2, N_2, F_2 tels que L_2 soit défini par $(\Lambda_2, S_2, P_2, N_2, F_2)$. Construisons alors $\Lambda_\cup = \Lambda_1 \cup \Lambda_2$, $P_\cup = P_1 \cup P_2$, $S_\cup = S_1 \cup S_2$, $F_\cup = F_1 \cup F_2$ et $N_\cup = (\Sigma_1 \cup \Sigma_2)^2 \setminus F_\cup$. On note $\Sigma = \Sigma_1 \cup \Sigma_2$. Montrons alors que

$$L_1 \cup L_2 = \underbrace{\Lambda_\cup \cup \left(P_\cup \Sigma^\star \cap \Sigma^\star S_\cup\right) \setminus \left(\Sigma^\star N_\cup \Sigma^\star\right)}_{L_\cup}.$$

On procède par double-inclusion.

"$\subseteq$" Soit $w \in L_1 \cup L_2$.

Cas 1 $w = \varepsilon$, alors $\Lambda_1 = \{\varepsilon\}$ ou $\Lambda_2 = \{\varepsilon\}$ donc $L_1 \cup L_2 = \{\varepsilon\}$ et donc $w \in L_\cup$.

Cas 2 $w \neq \varepsilon$. On pose $w = w_1 \dots w_n$. Sans perte de généralité, on suppose $w \in L_1$ et $w \notin L_2$. D'où $w_1 \in P_1$ et $w_n \in S_1$. Et, pour $i \in [\![1, n-1]\!]$, $w_i w_{i+1} \in F_1$ donc $w_1 \in P_1 \cup P_2$, $w_n \in S_1 \cup S_2$ et $\forall i \in [\![1, n-1]\!]$, $w_i w_{i+1} \in F_1 \cup F_2$. D'où $w \in L_\cup$.

"$\supseteq$" Cas 1 $w = \varepsilon$ alors $w \in \Lambda_\cup = L_1 \cup L_2$ donc $w \in \Lambda_1$ ou $w \in \Lambda_2$ donc $w \in L_1$ ou $w \in L_2$.

Cas 2 $w \neq \varepsilon$. On pose $w = w_1 w_2 \dots w_n$ avec $w_1 \in P_\cup$, $w_n \in S_\cup$ et $\forall i \in [\![1, n-1]\!]$, $w_i w_{i+1} \in F_\cup$. Alors, sans perte de généralité, on suppose $w_1 \in \Sigma_1$. Montrons par récurrence que $\forall p \in [\![1, n]\!]$, $w_p \in \Sigma_1$.

- On sait que $w_1 \in \Sigma_1$ par hypothèse.
- On suppose que $w_p \in \Sigma_1$ avec $p < n$. Alors, $w_p w_{p+1} \in F_\cup = F_1 \cup F_2$. Or, $F_2 \subseteq (\Sigma_2)^2$ et $w_p \in \Sigma_1$ avec $\Sigma_1 \cap \Sigma_2 = \varnothing$ donc $w_{p+1} \in \Sigma_1$.

On conclut par récurrence que $\forall i \in [\![1, n]\!]$, $w_i \in \Sigma_i$. Or, $w_n \in S_1 \cup S_2$ et $S_2 \cap \Sigma_1 = \varnothing$ donc $w_n \in S_1$. De plus, pour $i \in [\![1, n-1]\!]$, $w_i w_{i+1} \in F_1 \cup F_2$ donc $w_i w_{i+1} \in F_1$ et donc $w \in L_1$.

□

Concaténation

Contre-exemple :
Avec $L_1 = \{ab\}$ et $L_2 = \{ab\}$, deux langages locaux, alors $L_1 \cdot L_2 = \{abba\}$ n'est pas local. En effet, $P = \{a\}$, $S = \{a\}$, $F = \{ab, bb, ba\}$; or $aba \notin L_1 \cdot L_2$.

Propriété : Soient L_1 un langage local sur un alphabet Σ_1 et L_2 un langage local sur un alphabet Σ_2, avec $\Sigma_1 \cap \Sigma_2 = \varnothing$. Alors $L_1 \cdot L_2$ est un langage local.

Preuve :
Soient $\Lambda_1, S_1, P_1, N_1, F_1$ définissant L_1 et soient $\Lambda_2, S_2, P_2, N_2, F_2$ définissant L_2. Construisons $\Lambda_\bullet = \Lambda_1 \cap \Lambda_2$, $P_\bullet = P_1 \cup \Lambda_1 \cdot P_2$, $S_\bullet = S_2 \cup \Lambda_2 \cdot S_1$, $F_\bullet = F_1 \cup F_2 \cup S_1 \cdot P_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$. Montrons que

$$L_1 \cdot L_2 = \underbrace{\Lambda_\bullet \cup (P_\bullet \Sigma^\star \cap \Sigma^\star S_\bullet) \setminus \Sigma^\star N_\bullet \Sigma^\star}_{L^\bullet}.$$

On procède par double inclusion.

"$\subseteq$" Soit $w \in L_1 \cdot L_2$.

- Si $w = \varepsilon$, alors $\varepsilon \in L_1$ et $\varepsilon \in L_2$ donc $w = \varepsilon \in \Lambda_\bullet \subseteq L^\bullet$.
- Sinon, $w = u \cdot v$ avec $u \in L_1$ et $v \in L_2$. On sait que $|u| > 0$ ou $|v| > 0$.
  - Si $u \neq \varepsilon$, alors $u = u_1 \dots u_p$ avec $p \geqslant 1$. On sait que $u_1 \in P_1$, $u_p \in S_1$ et, pour $i \in [\![1, p-1]\!]$, $u_1 u_{i+1} \in F_1$.

  Sous-cas 1 Si $v = \varepsilon$, alors $\Lambda_2 = \{\varepsilon\}$, et donc $S_1 \subseteq S_\bullet$. Or, $P_1 \subseteq P_\bullet$ et $F_1 \subseteq F_\bullet$. On en déduit que $w = u \in L^\bullet$.

  Sous-cas 2 $v \neq \varepsilon$, alors $v = v_1 \dots v_q$ avec $v_1 \in P_2$, $v_q \in S_2$ et, pour $i \in [\![1, nq-1]\!]$, $v_i v_{i+1} \in F_2$. Or, $u_p v_1 \in S_1 \cdot P_2$ et donc $w = u \cdot v \in L^\bullet$.
  - Si $u = \varepsilon$, on procède de la même manière.

"$\supseteq$" Soit $w \in L^\bullet$.

- Si $w = \varepsilon$, alors $\varepsilon \in \Lambda_\bullet$ et donc $\varepsilon \in L_1$ et $\varepsilon \in L_2$. D'où $\varepsilon \in L_1 \cdot L_2$.
- Sinon, on pose $w = w_1 \dots w_n$.

Sous-cas 1 Si $\{i \in [\![1, n]\!] \mid w_i \in \Sigma_2\} = \varnothing$, on a donc $\forall i \in [\![1, n]\!]$, $w_i \in \Sigma_1$. De plus, $w_1 \in P_\bullet$, $w_n \in S_\bullet$ et, pour $i \in [\![1, n-1]\!]$, $w_i w_{i+1} \in F_\bullet$. Or, $P_\bullet \cap \Sigma_1 = P_1$, $S_\bullet \cap \Sigma_1 = S_1$, $\Lambda_2 = \{\varepsilon\}$ et $\forall i \in [\![1, n-1]\!]$ $w_i w_{i+1} \in F_1$. On en déduit que $w = w_1 \dots w_n \cdot \varepsilon \in L_1 \cdot L_2$.

Sous-cas 2 Si $M = \{i \in [\![1, n]\!] \mid w_i \in \Sigma_2\} \neq \varnothing$. Soit $i_0 = \min(M)$. D'où

$$w = \underbrace{w_1 \dots w_{i_0 - 1}}_{\in \Sigma_1} \cdot \underbrace{w_{i_0}}_{\in \Sigma_2} \dots w_n.$$

Si, $\ell_1, \ell_2 \in F_\bullet$, et $\ell_1 \in \Sigma_2$, alors $\ell_2 \in \Sigma_2$. De proche en proche, on en déduit que $\forall i \in [\![i_0, n]\!]$, $w_i \in \Sigma_2$.

- Si $i_0 = 1$, alors $w_1 \in \Sigma_2$, $w_1 \in P_\bullet$, $w_1 \in P_2$, $w_1 \in P_2$, $\Lambda_1 = \{\varepsilon\}$, et $w_n \in S_\bullet \cap \Sigma_2$, et $w_n \in S_2$. De plus, pour $i \in [\![1, n-1]\!]$, $w_i w_{i+1} \in F_\bullet \cap (\Sigma_2)^2$ donc $w_i w_{i+1} \in F_2$. D'où $w = \varepsilon \cdot w_1 \dots w_n \in L_1 \cdot L_2$.
- Si $i_0 > 1$, alors $w_1 \in \Sigma_1 \cap P^\bullet = P_1$, $w_n \in \Sigma_2 \cap S^\bullet = S_2$, et $\forall i \in [\![1, i_0 - 2]\!]$, $w_i w_{i+1} \in F^\bullet \cap (\Sigma_1)^2 = F_1$. D'où $w_{i_0 - 1} w_{i_0} \in F^\bullet \cap \Sigma_1 \Sigma_2 = S_1 P_2$ donc $w_{i_0 - 1} \in S_1$ et $w_{i_0} \in P_2$ donc $w_1 \dots w_{i_0 - 1} \in L_1$. Finalement, $\forall i \in [\![i_0, n-1]\!]$, $w_i w_{i_1 \in F_\bullet \cap (\Sigma)^2}$ donc À faire : Finir la preuve.

□

Étoile

Propriété : Soit L un langage local, alors $L^\star$ est un langage local.

Preuve :
Soient Λ, P, S, F et N définissant L. Alors $\Lambda_\star = \{\varepsilon\}$, $P_\star = P$, $S_\star = S$ et $F_\star = F \cup S \cdot P$. À faire : preuve à faire à la maison. □

Exemple :
Avec $L = \{a, b\}$, un langage local, on a $\Lambda_\star = \{\varepsilon\}$, $P_\star = S_\star = \{a, b\}$ (car $P = \{a, b\} = S$), et $S_\star = \{ab, ba, aa, bb\}$.

1.6.2 Expressions régulières linéaires

Définition : Un expression régulière est dite *linéaire* si chacune de ses lettres apparaît une fois au plus dans l'expression.

Exemple :

Oui	Non
a	aa
$a\|b$	$a\|ba$

Table 1.2 – Exemples et non-exemples d'expressions régulières linéaires

Exemple :
On définit une fonction booléenne permettant de vérifier si une expression régulière est linéaire :

$$\text{linéaire} : \left(\begin{array}{rcl} \varnothing & \mapsto & \top \\ \varepsilon & \mapsto & \top \\ a \in \Sigma & \mapsto & \top \\ e_1 \cdot e_2 & \mapsto & (\text{vars}(e_1) \cap \text{vars}(e_2) = \varnothing) \text{ et linéaire}(e_1) \text{ et linéaire}(e_2) \\ e_1 \mid e_2 & \mapsto & (\text{vars}(e_1) \cap \text{vars}(e_2) = \varnothing) \text{ et linéaire}(e_1) \text{ et linéaire}(e_2) \\ e_1{}^\star & \mapsto & \text{linéaire}(e_1) \end{array} \right).$$

Propriété : Le langage d'une expression régulière est local.

Preuve :
Il nous suffit de montrer le résultat sur le cas de base.

$\mathscr{L}(\varnothing) : \varnothing$ qui correspond à $\Lambda = \varnothing$, $S = \varnothing$, $P = \varnothing$, $F = \varnothing$.

$\mathscr{L}(\varepsilon) = \{\varepsilon\}$ qui correspond à $\Lambda = \{\varepsilon\}$ et $S = P = F = \varnothing$.

$\mathscr{L}(a) = \{a\}$ qui correspond à $\Lambda = \varnothing$, $S = \{a\}$, $P = \{a\}$ et $F = \varnothing$. □

Remarque :
Les grandeurs Λ, P, S et F sont de plus définies individuellement par la table suivante.

e	Λ	P	S	F
$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
ε	$\{\varepsilon\}$	$\varnothing$	$\varnothing$	$\varnothing$
a	$\varnothing$	$\{a\}$	$\{a\}$	$\varnothing$
$e_1{}^\star$	$\{\varepsilon\}$	$P(e_1)$	$S(e_1)$	$F(e_1) \cup S(e_1) \cdot P(e_1)$
$e_1 \cdot e_2$	$\Lambda(e_1) \cap \Lambda(e_1)$	$P(e_1) \cup \Lambda(e_2) \cdot P(e_2)$	$S(e_2) \cup \Lambda(e_2) \cdot S(e_1)$	$F(e_1) \cup F(e_2) \cup S(e_1) \cdot P(e_2)$
$e_1 \mid e_2$	$\Lambda(e_1) \cup \Lambda(e_2)$	$P(e_1) \cup P(e_2)$	$S(e_1) \cup S(e_2)$	$F(e_1) \cup F(e_2)$

TABLE 1.3 – Construction de Λ, P, S et F dans différents cas

REMARQUE (Notation) :
Si Σ_1 et Σ_2 sont deux alphabets et $\varphi : \Sigma_1 \to \Sigma_2$, alors on note $\tilde{\varphi}$ l'extension de φ aux mots de $\Sigma_1^\star$:

$$\tilde{\varphi}(w_1 \dots w_n) = \varphi(w_1) \dots \varphi(w_n)$$

et, de plus, on note

$$\tilde{\varphi}(L) = \{\tilde{\varphi}(w) \mid w \in L\}.$$

REMARQUE :
On a $\tilde{\varphi}(L \cup M) = \tilde{\varphi}(L \cup M) = \tilde{\varphi}L \cup \tilde{\varphi}(M)$.

Propriété :

$$\tilde{\varphi}(L \cdot M) = \tilde{\varphi}(L) \cdot \tilde{\varphi}(M)$$

Preuve :

$$\begin{aligned} w \in \tilde{\varphi}(L \cdot M) &\iff \exists u \in L \cdot M,\ w = \tilde{\varphi}(u) \\ &\iff \exists (v,t) \in L \times M,\ w = \tilde{\varphi}(v \cdot t) \\ &\iff \exists (v,t) \in L \times M,\ w = \tilde{\varphi}(v) \cdot \tilde{\varphi}(t) \\ &\iff w \in \tilde{\varphi}(L) \cdot \tilde{\varphi}(M). \end{aligned}$$

□

Définition : Soient $e \in \mathrm{Reg}(\Sigma_1)$, $\varphi : \Sigma_1 \to \Sigma_2$. On définit alors inductivement e_φ comme étant

$$\varnothing_\varphi = \varnothing \qquad a_\varphi = \varphi(a) \text{ si } a \in \Sigma_1 \qquad (e_1 \mid e_2)_\varphi = (e_1)_\varphi \mid (e_2)_\varphi$$
$$\varepsilon_\varphi = \varepsilon \qquad (e_1 \cdot e_2)_\varphi = (e_1)_\varphi \cdot (e_2)_\varphi \qquad (e_1^\star)_\varphi = ((e_1)_\varphi)^\star.$$

Propriété : Si $\varphi : \Sigma_1 \to \Sigma_2$ et $e \in \mathrm{Reg}(\Sigma_1)$, alors

$$\mathscr{L}(e_\varphi) = \tilde{\varphi}(\mathscr{L}(e)).$$

Preuve (par incuction sur $e \in \mathrm{Reg}(\Sigma_1)$) : cas $\varnothing$ $\mathscr{L}(\varnothing_\varphi) = \mathscr{L}(\varnothing) = \varnothing = \tilde{\varphi}(\varnothing) = \tilde{\varphi}(\mathscr{L}(\varnothing))$
cas ε $\mathscr{L}(\varepsilon_\varphi) = \mathscr{L}(\varepsilon) = \{\varepsilon\} = \tilde{\varphi}($ À faire : recopier ici
cas $e_1 \cdot e_2$ $\mathscr{L}((e_1 \cdot e_2)_\varphi) = \mathscr{L}((e_1)_\varphi \cdot (e_2)_\varphi)) = \mathscr{L}((e_1)_\varphi) \cdot \mathscr{L}((e_2)_\varphi) = \tilde{\varphi}(\mathscr{L}(e_1)) \cdot \tilde{\varphi}(\mathscr{L}(e_2)) = \tilde{\varphi}(\mathscr{L}(e_1) \cdot \mathscr{L}(e_2)) = \tilde{\varphi}(\mathscr{L}(e_1 \cdot e_2))$.
De même pour les autres cas

□

Propriété : Soit $e \in \mathrm{Reg}(\Sigma_1)$. Il existe $f \in \mathrm{Reg}(\Sigma)$ et $\varphi : \Sigma \to \Sigma_1$ tel que f est linéaire et $e = f_\varphi$.

Preuve :
Il suffit de numéroter les lettres (c.f. exemple ci-dessous). □

Exemple :
Avec $e = c^\star((a \cdot a) \mid \varepsilon) \cdot ((a \mid c \mid \varepsilon)^\star))^\star \cdot b \cdot a \cdot a^\star$, on a

$$f = c_1^\star((a_1 \cdot a_2) \mid \varepsilon) \cdot (b_1((a_3 \mid c_2 \mid \varepsilon)^\star))^\star \cdot b_2 \cdot a_4 \cdot a_5$$

et

$$\varphi : \begin{pmatrix} a_1 & \mapsto & a \\ a_2 & \mapsto & a \\ a_3 & \mapsto & a \\ a_4 & \mapsto & a \\ a_5 & \mapsto & a \\ b_1 & \mapsto & b \\ b_2 & \mapsto & b \\ c_1 & \mapsto & c \\ c_2 & \mapsto & c \end{pmatrix}.$$

1.6.3 Automates locaux

Définition (automate local, local standard) **:** Un automate $\mathscr{A} = (\Sigma, \mathbb{Q}, I, F, \delta)$ est dit *local* dès lors que pour out $\forall (q_1, q_2, \ell, q_3, q_4) \in \mathbb{Q} \times \mathbb{Q} \times \Sigma \times \mathbb{Q} \times \mathbb{Q}$,

$$(q_1, \ell, q_3) \in \delta \quad \text{et} \quad (q_2, \ell, q_4) \in \delta \qquad \Longrightarrow \qquad q_3 = q_4.$$

L'automate $\mathscr{A}$ est dit, de plus, *standard* lorsque $\mathrm{Card}(I) = 1$ et qu'il n'existe pas de transitions entrante en l'unique état initial q_0.

Propriété : Un langage est local si et seulement s'il est reconnu par un automate local standard.

Exemple :

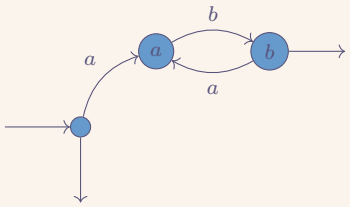


FIGURE 1.19 – Automate local reconnaissant le langage $(ab)^\star$

Preuve " $\Longrightarrow$ " Soit L un langage local. Soit (Λ, S, P, F, N) tels que

$$L = \Lambda \cup (P\Sigma^\star \cap \Sigma^\star) \setminus (\Sigma^\star N \Sigma^\star).$$

Soit alors l'automate $\mathbb{Q} = \Sigma \cup \{\varepsilon\}$, $I = \{\varepsilon\}$, $F_{\mathcal{A}} = S \cup \Lambda$, et

$$\begin{aligned}\delta = \quad & \{(q, \ell, q') \in \mathbb{Q} \times \Sigma \times \mathbb{Q} \mid qq' \in F \text{ et } q' = \ell\} \\ & \cup \{(\varepsilon, \ell, q) \in \mathbb{Q} \times \Sigma \times \mathbb{Q} \mid \ell = q \text{ et } q \in P\}.\end{aligned}$$

On pose $\mathcal{A} = (\Sigma, \mathbb{Q}, I, F_{\mathcal{A}}, \delta)$. Montrons que $\mathcal{L}(\mathcal{A}) = L$.

"$\subseteq$" Soit $w \in \mathcal{L}(\mathcal{A})$. Soit donc

$$q_1 \xrightarrow{w_1} q_2 \to \cdots \to q_{n-1} \xrightarrow{w_n} q_n$$

une exécution acceptante dans $\mathcal{A}$. Montrons que $w_1 \dots w_n \in L$.

Cas 1 $w = \varepsilon$ et $n = 0$. Ainsi $q_0 = q_n = \varepsilon$ et $F \cap I = \varnothing$. Or $F_{\mathcal{A}} = S \cup \Lambda$, et donc $\Lambda = \{\varepsilon\}$, d'où $\varepsilon \in L$.

Cas 2 $w \neq \varepsilon$. On sait que $w_1 \in P$; en effet, $(\varepsilon, w_1, q_1) \in \delta$ donc $w_1 = q_1 \in P$. De même, $(q_{n-1}, w_n, q_n) \in \delta$, d'où $S \ni w_n = q_n$. De plus, $\forall i \in [\![1, n-1]\!]$, $(q_{i-1}, w_i, q_i) \in \delta$ et $(q_i, w_{i+1}, q_{i+1}) \in \delta$. Ainsi, $w_i = q_i$ et $w_{i+1} = q_{i+1}$ avec $q_i q_{i+1} \in F$, d'où $w_i w_{i+1} \in F$. Donc $w \in L$.

"$\supseteq$" Soit $w = w_1 \dots w_n \in L$.

Cas 1 $w = \varepsilon$. On a $\Lambda = \{\varepsilon\}$, et donc ε est final (ou initial). On en déduit que $\varepsilon \in \mathcal{L}(\mathcal{A})$.

Cas 2 $w \neq \varepsilon$. Montrons, par récurrence finie sur $p \leqslant n$, qu'il existe une exécution

$$q_0 \xrightarrow{w_1} q_1 \to \cdots \to q_{n-1} \xrightarrow{w_p} q_p$$

dans $\mathcal{A}$.

— Avec $p = 1$, on a $w_1 \in P$ donc $(\varepsilon, w_1, w_1) \in \delta$. Ainsi, $\varepsilon \xrightarrow{w_1} w_1$ est une exécution dans $\mathcal{A}$.

— Supposons construit $\varepsilon \xrightarrow{w_1} q_1 \to \cdots \xrightarrow{w_p} q_p = w_p$ avec $p < n$. Or, $w_p w_{p+1} \in F$ donc $(w_p, w_{p+1}, w_{p+1}) \in \delta$. Ainsi,

$$\varepsilon \xrightarrow{w_1} q_1 \to \cdots \xrightarrow{w_p} q_p \xrightarrow{w_{p+1}} w_{p+1}$$

est une exécution acceptante de $\mathcal{A}$.

De proche en proche, on a

$$\varepsilon \xrightarrow{w_1} q_1 \to \cdots \to q_{n-1} \xrightarrow{w_n} w_n$$

une exécution dans $\mathcal{A}$. Or, $w_n \in S = F_{\mathcal{A}}$ et donc l'exécution est acceptante dans $\mathcal{A}$, et $w \in \mathcal{L}(\mathcal{A})$.

" $\Longleftarrow$ " Soit $\mathcal{A} = (\Sigma, \mathbb{Q}, I, F_{\mathcal{A}}, \delta)$ un automate localement standard. Montrons que $\mathcal{L}(\mathcal{A})$ est local. Il suffit de montrer que $\rho(\mathcal{L}(\mathcal{A})) = \mathcal{L}(\mathcal{A})$. Or $\mathcal{L}(\mathcal{A}) \subseteq \rho(\mathcal{L}(\mathcal{A}))$. On montre donc $\rho(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{A})$.
Soit $w \in \rho(\mathcal{L}(\mathcal{A}))$. Ainsi,

$$w \in \Lambda(\mathcal{L}(\mathcal{A})) \cup \Big(P(\mathcal{L}(\mathcal{A}))\Sigma^\star \cap \Sigma^\star S(\mathcal{L}(\mathcal{A}))\Big) \setminus \Big(\Sigma^\star N(\mathcal{L}(\mathcal{A}))\Sigma^\star\Big).$$

Montrons que $w \in \mathcal{L}(\mathcal{A})$.

— Si $w \in \Lambda(\mathcal{L}(\mathcal{A}))$, alors $w = \varepsilon$. Or, $\Lambda(\mathcal{L}(\mathcal{A})) = \mathcal{L}(\mathcal{A}) \cap \{\varepsilon\}$. Ainsi $w \in \mathcal{L}(\mathcal{A})$.

— Sinon, $w = w_1 \dots w_n$ avec $w_1 \in P(\mathcal{L}(\mathcal{A}))$, donc il existe $u \in \Sigma^\star$ tel que $w_1 \cdot u \in \mathcal{L}(\mathcal{A})$. Il existe donc une exécution acceptante

$$I \ni q_0 \xrightarrow{w_1} q_1 - - \overset{u}{\to} q_s \in F_{\mathcal{A}}.$$

Il existe donc une exécution $q_0 \xrightarrow{w_1} q_1$.

Supposons construit $q_1 \xrightarrow{w_1} q_1 \to \cdots \xrightarrow{w_p} q_p$ avec $p < n$. Or, $w_p w_{p+1} \in F(\mathscr{L}(\mathscr{A}))$, donc il existe $w \in \Sigma^\star$ et $y \in \Sigma^\star$ tels que $x \cdot w_p \cdot w_{p+1} \cdot y \in \mathscr{L}(\mathscr{A})$. Il existe donc une exécution acceptante

$$r_0 \overset{x}{\dashrightarrow} r_{p-1} \xrightarrow{w_p} r_p \xrightarrow{w_{p+1}} r_{p+1} \overset{y}{\dashrightarrow} r_s.$$

Or, par localité de l'automate, $q_p = r_p$. Il existe donc q_{p+1} $(= r_{p+1})$ tel que $(q_p, w_{p+1}, q_{p+1}) \in \delta$. On a donc une exécution

$$q_0 \xrightarrow{w_1} q_1 \to \cdots \to q_p \xrightarrow{w_{p+1}} q_{p+1}.$$

De proche en proche, il existe une exécution

$$q_0 \xrightarrow{w_1} q_1 \to \cdots \to q_n.$$

Or, $w_n \in S(\mathscr{L}(\mathscr{A}))$, il existe donc $v \in \Sigma^\star$ tel que $v \cdot w_n \in \mathscr{L}(\mathscr{A})$, donc il existe un exécution acceptante

$$I \ni r_0 \overset{v}{\dashrightarrow} r_{s-1} \xrightarrow{w_n} r_s \in F_{\mathscr{A}}.$$

Par localité, $r_s = q_n \in F_{\mathscr{A}}$.
Donc $\rho(\mathscr{L}(\mathscr{A})) \subseteq \mathscr{L}(\mathscr{A})$ et donc $\rho(\mathscr{L}(\mathscr{A})) = \mathscr{L}(\mathscr{A})$. On en déduit que $\mathscr{L}(\mathscr{A})$ est local.

□

Exemple :
Dans le langage local $(ab)^\star \mid c^\star$, on a $\Lambda = \{\varepsilon\}$, $S = \{c, b\}$, $P = \{a, c\}$ et $F = \{ab, ba, cc\}$. L'automate local reconnaissant ce langage est celui ci-dessous.

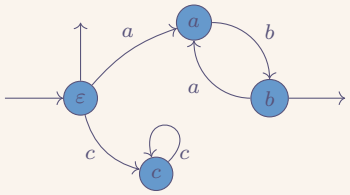


Figure 1.20 – Automate local reconnaissant $(ab)^\star \mid c^\star$

Propriété : Soit $\mathscr{A} = (\Sigma, \mathbb{Q}, I, F, \delta)$ un automate et $\varphi : \Sigma \to \Sigma_1$. On pose

$$\delta' = \{(a, \varphi(\ell), q') \mid (q, \ell, q') \in \delta\}.$$

On pose $\mathscr{A}' = (\Sigma, \mathbb{Q}, I, F, \delta')$. On a $\mathscr{L}(\mathscr{A}') = \tilde{\varphi}(\mathscr{L}(\mathscr{A}))$.

Exercice :
Montrons que $\mathrm{LR} \subsetneq \wp(\Sigma^\star)$ i.e. il existe des langages non reconnaissables.

$\mathbb{R}$ n'est pas dénombrable. On écrit un nombre réel comme une suite infinie

$$0{,}10110010101\ldots1010110\ldots$$

On pose $\Sigma = \{a\}$, on crée le langage L, associé au nombre ci-dessus comme l'ensemble contenant a, aaa, $aaaa$,…

Remarque (Notation) :
On note A_φ, l'automate $(\varphi(E), \mathbb{Q}, I, F, \delta')$ où $\delta' = \{(q, \varphi(\ell), q') \mid (q, \ell, q') \in \delta\}$.

1.6.4 Algorithme de Berry-Sethi : les langages réguliers sont reconnaissables

Exemple :
On considère l'expression régulière $aab(a \mid b)^\star$. On numérote les lettres : $a_1a_2b_1(a_3 \mid b_2)^\star$, avec

$$\varphi : \begin{pmatrix} a_1 & \mapsto & a \\ a_2 & \mapsto & a \\ a_3 & \mapsto & a \\ b_1 & \mapsto & b \\ b_2 & \mapsto & b \end{pmatrix}.$$

	Λ	S	P	F
a_1	$\varnothing$	a_1	a_1	$\varnothing$
a_2	$\varnothing$	a_2	a_2	$\varnothing$
$a_1 \cdot a_2$	$\varnothing$	a_2	a_1	a_1a_2
b_1	$\varnothing$	b_1	b_1	$\varnothing$
$a_1a_2b_1$	$\varnothing$	b_1	a_1	a_1a_2, a_2b_1
a_3	$\varnothing$	a_3	a_3	$\varnothing$
b_2	$\varnothing$	b_2	b_2	$\varnothing$
$a_3 \mid b_2$	$\varnothing$	a_3, b_2	a_3, b_2	$\varnothing$
$(a_3 \mid b_2)^\star$	ε	a_3, b_2	a_3, b_2	$a_3b_2, b_2a_3, a_3a_3, b_2b_2$
$a_1a_2b_1(a_3 \mid b_2)^\star$	$\varnothing$	a_3, b_2ab_1	a_1	$a_3b_2, b_2a_3, a_3a_3, b_2b_2, a_1a_2, a_2b_1, b_1a_3, b_1b_2$

Table 1.4 – Λ, S, P et F pour les différents mots reconnus

On crée donc l'automate ci-dessous.

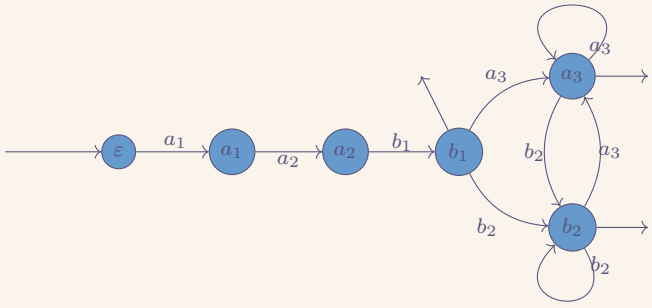


Figure 1.21 – Automate déduit de la table 1.4

On applique la fonction φ a tous les états et transitions pour obtenir l'automate ci-dessous. Cet algorithme reconnaît le langage $aab(a \mid b)^\star$.

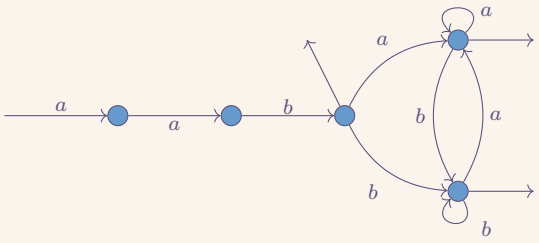


FIGURE 1.22 – Application de φ à l'automate de la figure 1.21

Théorème : Tout langage régulier est reconnaissable. De plus, on a un algorithme qui calcule un automate le reconnaissant, à partir de sa représentation sous forme d'expression régulière.

Algorithme (BERRY-SETHI) : Entrée : Une expression régulière e
Sortie : Un automate reconnaissant $\mathscr{L}(e)$

1. On linéarise e en f avec une fonction φ telle que $f_\varphi = e$.
2. On calcule inductivement $\Lambda(f)$, $S(f)$, $P(f)$, et $F(f)$.
3. On fabrique $\mathscr{A} = (\Sigma, \mathbb{Q}, I, F, \delta)$ un automate reconnaissant $\mathscr{L}(f)$.
4. On retourne $\mathscr{A}_\varphi$.

À faire : refaire la mise en page pour les algorithmes

1.6.5 Les langages reconnaissables sont réguliers

On fait le « sens inverse » : à partir d'un automate, comment en déduire le langage reconnu par cet automate ?

L'idée est de supprimer les états un à un. Premièrement, on rassemble les états initiaux en les reliant à un état i, et de même, on relie les états finaux à f. Pour une suite d'états, on concatène les lettres reconnus sur chaque transition :

FIGURE 1.23 – Succession d'états

De même, lors de « branches » en parallèles, on les concatène avec un |. En appliquant cet algorithme à l'automate précédent, on a

$$(aab) \cdot \Big((\varepsilon \mid aa^\star) \mid (b \mid aa^\star b) \cdot (b \mid aa^\star b)^\star (aa^\star \mid \varepsilon) \Big).$$

Définition : Un *automate généralisé* est un quintuplet $(\Sigma, \mathbb{Q}, I, F, \delta)$ où

— Σ est un alphabet ;
— $\mathbb{Q}$ est un ensemble fini ;
— $I \subseteq \mathbb{Q}$;
— $F \subseteq \mathbb{Q}$;

— $\delta \subseteq \mathbb{Q} \times \mathrm{Reg}(\Sigma) \times \mathbb{Q}$, avec

$$\forall r \in \mathrm{Reg}(\Sigma),\ \forall (q, q') \in \mathbb{Q}^2,\ \mathrm{Card}(\{(q, r, q') \in \delta\}) \leqslant 1.$$

Définition (Langage reconnu par un automate généralisé) **:** Soit $(\Sigma, \mathbb{Q}, I, F, \delta)$ un automate généralisé. On dit qu'un mot w est reconnu par l'automate s'il existe une suite

$$q_0 \xrightarrow{r_1} q_1 \xrightarrow{r_2} q_2 \to \cdots \to q_{n-1} \xrightarrow{r_n} q_n$$

et $(u_i)_{i \in [\![1,n]\!]}$ tels que $\forall i \in [\![1, n]\!]$, $u_i \in \mathscr{L}(r_i)$ et $w = u_1 \cdot u_2 \cdot \ldots \cdot u_n$.

Définition : Un automate généralisé $(\Sigma, \mathbb{Q}, I, F, \delta)$ est dit « bien détouré [2] » si $I = \{i\}$ et $F = \{f\}$, avec $i \neq f$, tels que i n'a pas de transitions entrantes et f n'a pas de transitions sortantes.

Lemme : Tout automate généralisé est équivalent à un automate généralisé « bien détouré. » En effet, soit $\mathscr{A} = (\Sigma, \mathbb{Q}, I, F, \delta)$ un automate généralisé. Soit $i \notin \mathbb{Q}$ et $f \notin \mathbb{Q}$. On pose $\Sigma' = \Sigma$, $I' = \{i\}$, $F' = \{f\}$, $\mathbb{Q}' = \mathbb{Q} \cup \{i, f\}$ et

$$\delta' = \delta \cup \{(i, \varepsilon, q) \mid q \in I\} \cup \{(q, \varepsilon, f) \mid q \in F\}.$$

Alors, l'automate $\mathscr{A}' = (\Sigma', \mathbb{Q}', I', F', \delta')$ est équivalent à $\mathscr{A}$ et « bien détouré. »

Lemme : Soit $\mathscr{A} = (\Sigma, \mathbb{Q}, I, F, \delta)$ un automate généralisé « bien détouré » tel que $|\mathbb{Q}| \geqslant 3$. Alors il existe un automate généralisé « bien détouré » $\mathscr{A}' = (\Sigma, \mathbb{Q}', I, F, \delta')$ avec $\mathbb{Q}' \subsetneq \mathbb{Q}$ et $\mathscr{L}(\mathscr{A}) = \mathscr{L}(\mathscr{A}')$.

Preuve :
Étant donné qu'il existe au plus une transition entre chaque pair d'état $(q, q') \in \mathbb{Q}^2$, il est possible de le représenter au moyen d'une fonction de transition

$$T : \mathbb{Q} \times \mathbb{Q} \longrightarrow \mathrm{Reg}(\Sigma).$$

À faire : Recopier la def de T Soit $q \in \mathbb{Q} \setminus \{i, f\}$. Soit alors T' défini, pour $(q_a, q_b) \in \mathbb{Q} \setminus \{q\}$, par

$$T'(q_a, q_b) = T(q_a, q_b) \mid T(q_a, q) \cdot T(q, q)^\star \cdot T(q, q_b).$$

On considère l'automate $\mathbb{Q}' = \mathbb{Q} \setminus \{q\}$ et δ' construit à partir de T'. □

Exemple :
On considère l'automate ci-dessous.

2. Cette notation n'est pas officielle.

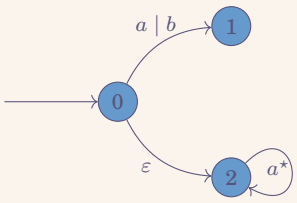


FIGURE 1.24 – Automate exemple

La fonction T peut être représentée dans la table ci-dessous.

	0	1	2
0	$\varnothing$	$a \mid b$	ε
1	$\varnothing$	$\varnothing$	$\varnothing$
2	$\varnothing$	$\varnothing$	$a^\star$

TABLE 1.5 – Fonction T équivalente à l'automate de la figure 1.24

EXEMPLE :
On applique l'algorithme à l'automate suivant.

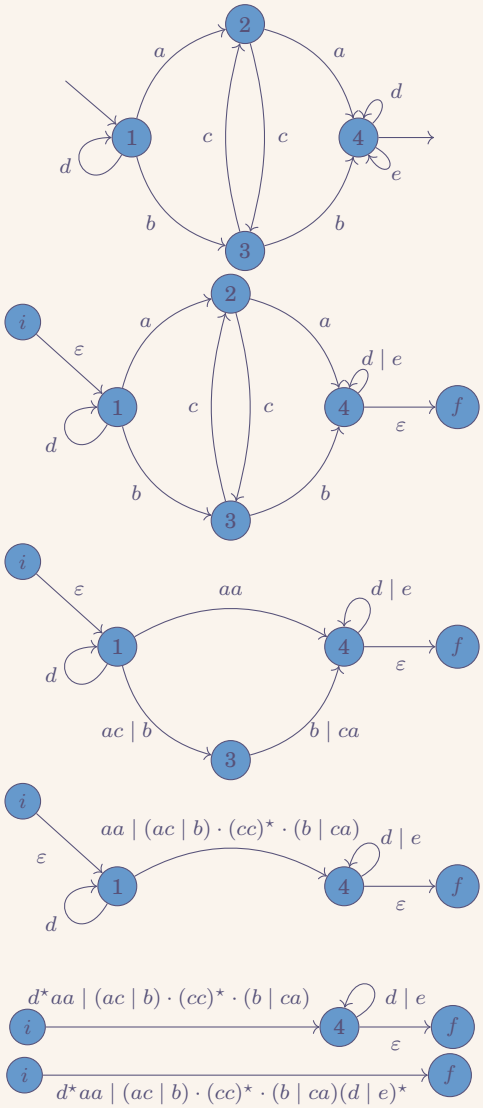


FIGURE 1.25 – Application de l'algorithme à un exemple

On a donc que le langage de l'automate initial est

$$\mathcal{L}(d^\star(aa) \mid (ac \mid b)(cc)^\star(b \mid ca)(d \mid e)^\star).$$

Théorème : Un langage reconnaissable est régulier.

Preuve :
On itère le lemme précédent depuis un automate généralisé $\mathcal{A}$ jusqu'à obtention d'un automate comme celui ci-dessous.

FIGURE 1.26 – Automate résultat de l'application du lemme

On a alors $\mathscr{L}(\mathscr{A}) = \mathscr{L}(r)$. □

Théorème (KLEENE) : Un langage est régulier si et seulement s'il est reconnaissable. Et, on a donné un algorithme effectuant ce calcul dans les deux sens.

1.7 La classe des langages réguliers

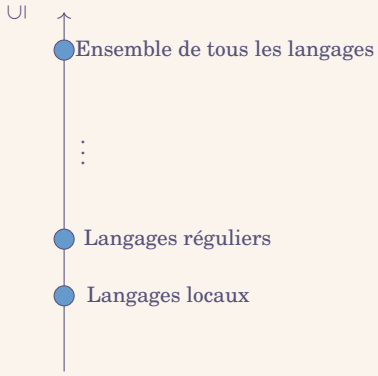


FIGURE 1.27 – Ensembles de langages

Propriété : La classe des langages réguliers/reconnaissables est stable par passage au complémentaire.

Preuve :
Soit $L \in \mathrm{LR}$. Soit $\mathscr{A} = (\Sigma, \mathbb{Q}, I, F, \delta)$ un automate reconnaissant le langage L. Soit $\mathscr{A}' = (\Sigma, \mathbb{Q}', I', F', \delta')$ un automate déterministe et complet équivalent à $\mathscr{A}$. Soit $\mathscr{A}'' = (\Sigma, \mathbb{Q}', I', \mathbb{Q}' \setminus F', \delta')$. Alors (à prouver à la maison) $\mathscr{L}(\mathscr{A}'') = \Sigma^\star \setminus \mathscr{L}(\mathscr{A}) = \Sigma^\star \setminus L$ et donc $\Sigma^\star \setminus L$ est reconnaissable/régulier. □

Corollaire : On a la stabilité par intersection. En effet,

$$L \cap L' = \left(L^{\mathrm{c}} \cup (L')^{\mathrm{c}}\right)^{\mathrm{c}}$$

où L^{c} est le complémentaire de L.

Corollaire : Si L et L' sont deux langages réguliers (quelconques), alors $L \setminus L'$ est un langage régulier. En effet,

$$L \setminus L' = L \cap (L')^{\mathrm{c}}.$$

Corollaire : Si L et L' sont deux langages réguliers. Alors $L \triangle L'$ [3] est un langage régulier. En effet

$$L \triangle L' \overset{(\text{def})}{=} (L \cup L') \setminus (L \cap L').$$

1.7.1 Limite de la classe/Lemme de l'étoile

Théorème (Lemme de l'étoile) **:** Soit L un langage reconnu par un automate à n états. Pour tout mot $u \in L$ de longueur supérieure ou égale à n, il existe trois mots x, y et z tels que

$$u = x \cdot y \cdot z, \qquad |x \cdot y| \leqslant n, \qquad y \neq \varepsilon, \qquad \text{et} \qquad \forall p \in \mathbb{N},\ x \cdot y^p \cdot z \in L.$$

Preuve :
Soit L un langage reconnu par un automate $\mathcal{A} = (\Sigma, \mathbb{Q}, I, F, \delta)$ à n états. Soit u un mot d'un alphabet Σ de longueur supérieure ou égale à n ($u \in \Sigma^{\geqslant n}$) tel que $u \in L$. Alors, il existe un exécution acceptante

$$q_0 \xrightarrow{u_1} q_1 \xrightarrow{u_2} q_2 \to \cdots \to q_{m-1} \xrightarrow{u_m} q_m$$

avec $m \geqslant n$. Par principe des tiroirs, l'ensemble $\{(i,j) \in [\![0,m]\!]^2 \mid i < j \text{ et } q_i = q_j\}$ est non vide. Et donc $A = \{j \in [\![0,m]\!] \mid \exists i \in [\![0, j-1]\!],\ q_i = q_j\}$ est non vide. Soit alors $j_0 = \min A$ bien défini. Alors, par définition de A, il existe $i_0 \in [\![0, j_0 - 1]\!]$ tel que $q_{i_0} = q_{j_0}$ et $j_0 \leqslant n$. On pose donc

$$\underbrace{q_0 \xrightarrow{u_1} q_1 \xrightarrow{u_2} \cdots \xrightarrow{u_{i_0}}}_{x} q_{i_0} \underbrace{\xrightarrow{u_{i_0+1}} q_{i_0+1} \to \cdots \xrightarrow{u_{j_0}}}_{y} q_{j_0} \underbrace{\xrightarrow{u_{j_0+1}} q_{j_0+1} \to \cdots \xrightarrow{u_m}}_{z} q_m \ :$$

$x = u_1 u_2 \dots u_{i_0}$, $y = u_{i_0+1} \dots u_{j_0}$ et $z_{j_0+1} \dots u_m$. On a donc $y \neq \varepsilon$: en effet $i_0 < j_0$. Également, on a $|x \cdot y| = j_0 \leqslant n$ et $u = x \cdot y \cdot z$. Montrons alors que $\forall p \in \mathbb{N},\ x \cdot y^p \cdot z \in L$. La suite de transitions

$$q_0 \xrightarrow{u_1} q_1 \to \cdots \xrightarrow{u_{i_0}} q_{i_0} \xrightarrow{u_{j_0+1}} q_{j_0+1} \to \cdots \xrightarrow{u_m} q_m$$

est une exécution acceptante donc $x \cdot z \in L$. De proche en proche, on en déduit que $x \cdot y^p \cdot z \in L$ pour tout $p \in \mathbb{N}$. □

Corollaire : Il y a des langages non réguliers/reconnaissables.

Preuve :
Soit $L = \{a^n \cdot b^n \mid n \in \mathbb{N}\}$. Montrons que L n'est pas régulier par l'absurde. Supposons L reconnaissable par un automate $\mathcal{A}$ à n états, et soit $u = a^n \cdot b^n$. Alors $|u| \geqslant n$. D'où, d'après le lemme de l'étoile, il existe un triplet $(x,y,z) \in (\Sigma^\star)^3$ tel que $y \neq \varepsilon$, $u = x \cdot y \cdot z$, $|x \cdot y| \leqslant n$ et $x \cdot y^\star \cdot z \subseteq L$ $(\star)$. Il existe donc $p \in [\![1,n]\!]$ tel que $y = a^p$. De même, il existe $q \in [\![0, n-p]\!]$ tel que $x = a^q$ et $z = a^{n-p-q} \cdot b^n$. Donc, d'après $(\star)$, $x \cdot y \cdot y \cdot z \in L$ et donc $a^q \cdot a^p \cdot a^p \cdot a^{n-p-q} \cdot b^n \in L$, d'où $a^{n+p} \cdot b^n \in L$. Or, comme $p \neq 0$, $n + p \neq n$: une contradiction. □

Exercice :
On considère le langage $L_2 = \{w \in \Sigma^\star \mid |w|_a = |w|_b\}$. *Le langage L_2 est-il régulier ?*
La même démonstration fonction en remplaçant L par L_2. Mais, nous allons procéder autrement, par l'absurde : on suppose L_2 régulier. Or, on sait que, d'après la preuve précédente, $L = L_2 \cap a^\star \cdot b^\star$, et $a^\star \cdot b^\star$ est régulier. D'où L régulier, ce qui est absurde.

Exercice :
On considère le langage $L = \{w \in \Sigma^\star \mid |w|_a \equiv |w|_b\ [3]\}$. *Le langage L est-il régulier ?*

3. $\triangle$ est la différence symétrique

Oui, l'automate de la figure suivante reconnait le langage L (les états représentent la différence $|w|_a - |w|_b \bmod 3$).

Montrons à présent qu'un automate à moins de trois états n'est pas possible : si $\delta^\star(i_0, a^x) = \delta^\star(i_0, a^y)$ avec $[\![0,2]\!] \ni x < y \in [\![0,2]\!]$, alors pour tout $z \in \mathbb{N}$, $\delta^\star(i_0, a^{x+z}) = \delta^\star(i_0, a^{y+z})$. On pose $z = 3 - y$. Alors

$$\delta^\star(i_0, a^{\underbrace{x+3-y}_{F}}) = \delta^\star(i_0, a^{\underbrace{3}_{F}}).$$

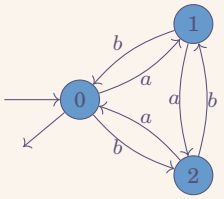


Figure 1.28 – Automate reconnaissant le langage $\{w \in \Sigma^\star \mid |w|_a \equiv |w|_b \ [3]\}$

Exercice :
Soit $\Sigma = \{0, 1, \text{'('}, \text{')'}, \text{'\{'}, \text{'\}'}, \text{','}\}$. On écrit en OCaml la fonction `to_string` définie telle que si (`affiche` $\mathcal{A}$) et (`affiche` $\mathcal{A}'$) donnent le même affichage, alors $\mathcal{A} = \mathcal{A}'$.

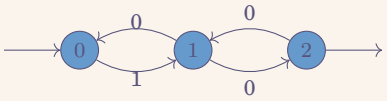


Figure 1.29 – Codage d'un automate par une chaîne de caractères

Par exemple, on représente l'automate ci-dessus par

"$(\{0,1,10\},\{0\},\{10\},\{(0,0,1),(1,0,10),(10,0,1),(1,1,0)\})$."

```
1 let affiche (Q,I,F,δ) =
```

Code 1.3 – Fonction `affiche` affichant un automate

À faire : Recopier le code

Exercice :
Supposons que tout langage est reconnaissable. Soit $L = \{w \in \Sigma^\star \mid \exists \mathcal{A}, w \leftarrow$ `affiche` $\mathcal{A}$ et $w \notin \mathcal{L}(\mathcal{A})\}$. Soit B un automate tel que $L = \mathcal{L}(B)$. Soit $w \in$ `affiche` B. Si $w \in L$, alors il existe un automate tel que $w =$ `affiche` $\mathcal{A}$ et $w \notin \mathcal{L}(\mathcal{A})$. D'où $\mathcal{A} = B$ par injectivité et donc $w \notin \mathcal{L}(B) = L$, ce qui est absurde. Sinon, si $w \notin L$, alors $w =$ `affiche` B avec $w \notin \mathcal{L}(B)$ et $w \in L$, ce qui est absurde.

Annexe 1.A Comment prouver la correction d'un programme ?

Avec $\Sigma = \{a, b\}$. Comment montrer qu'un mot a au moins un a et un nombre pair de b.

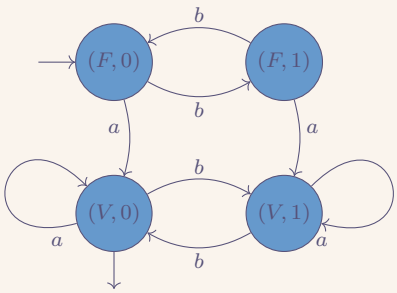


FIGURE 1.30 – Automate reconnaissant les mots valides

On veut montrer que

P_w : « $\forall w \in \Sigma^\star$, $\forall q \in \mathbb{Q}$, (il existe une exécution par w menant à q) $\iff$ w satisfait I_q »

où

$$I_{\underset{\mathbb{B}}{(v,} \ \underset{\{0,1\}}{r)}} \quad : \quad (|w|_a \geqslant 1 \iff v) \text{ et } (r = |w|_b \bmod 2).$$

On le montre par récurrence sur la longueur de w :

" $\Longrightarrow$ " — Pour $w = \varepsilon$, alors montrons que $\forall q \in \mathbb{Q}$, il existe un exécution menant à q étiquetée par w (noté $\xrightarrow[\mathcal{A}]{w} q$) si et seulement si w satisfait I_q.

- $\xrightarrow[\mathcal{A}]{\varepsilon} (\boldsymbol{F}, 0)$ est vrai, de plus ε satisfait $I_{(\boldsymbol{F},0)}$;
- sinon si $q \neq (\boldsymbol{F}, 0)$, alors $\xrightarrow[\mathcal{A}]{\varepsilon} q$ est fausse, de plus ε ne satisfait pas I_q.

— Supposons maintenant P_w vrai pour tout mot w de taille n. Soit $w = w_1 \dots w_n w_{n+1}$ un mot de taille $n+1$. Notons $\underline{w} = w_1 \dots w_n$. Montrons que P_w est vrai. Soit $q \in \mathbb{Q}$. Supposons $\xrightarrow[\mathcal{A}]{w} q$.

- Si $q = (\boldsymbol{F}, 0)$ et $w_{n+1} = b$. On a donc $\xrightarrow[\mathcal{A}]{w} (\boldsymbol{F}, 1)$, et, par hypothèse de récurrence, $\underline{w}$ satisfait. Donc $|\underline{w}|_a = 0$ et $|\underline{w}|_b \equiv 1$ [2] donc $|w|_a = 0$ et $|w|_b \equiv 0$ [2] donc w satisfait $I_{(\boldsymbol{F},0)}$.
- De même pour les autres cas.

" $\Longleftarrow$ " Réciproquement, supposons que w satisfait I_q.

— Si $w = (\boldsymbol{V}, 0)$ et $w_{n+1} = a$. Alors,

- si $|\underline{w}|_a = 0$, alors $\underline{w}$ satisfait $I_{(\boldsymbol{F},0)}$. Par hypothèse de récurrence, on a donc $\xrightarrow[\mathcal{A}]{\underline{w}} (\boldsymbol{F}, 0)$ et donc $\xrightarrow[\mathcal{A}]{w} (\boldsymbol{V}, 0)$.
- si $|\underline{w}|_b \geqslant 1$, alors $\underline{w}$ satisfait $I_{(\boldsymbol{V},0)}$ donc $\xrightarrow[\mathcal{A}]{\underline{w}} (\boldsymbol{V}, 0)$ et donc $\xrightarrow[\mathcal{A}]{w} (\boldsymbol{V}, 0)$.

— De même pour les autres cas.

On a donc bien

$$\forall w \in \Sigma^\star, \forall q \in \mathbb{Q}, \ \xrightarrow[\mathcal{A}]{w} q \iff w \text{ satisfait } I_q.$$

Finalement,

$$\begin{aligned}\mathscr{L}(\mathscr{A}) &= \{w \in \Sigma^\star \mid \exists f \in F,\ \xrightarrow[\mathscr{A}]{w} f\}\\ &= \{w \in \Sigma^\star \mid \xrightarrow[\mathscr{A}]{w} (\boldsymbol{V}, 0)\}\\ &= \{w \in \Sigma^\star \mid w \text{ satisfait } I_{(\boldsymbol{V},0)}\}\\ &= \{w \in \Sigma^\star \mid |w|_a \geqslant 1 \text{ et } |w|_b \equiv 0\ [2]\}\end{aligned}$$

Annexe 1.B Hors-programme

Définition : On appelle monoïde un ensemble M muni d'une loi "$\cdot$" interne associative admettant un élément neutre 1_M.

Définition : Étant donné deux monoïdes M et N, on appelle morphisme de monoïdes une fonction $\mu : M \to N$ telle que

1. $\mu(1_M) = 1_N$;
2. $\mu(x \cdot_M y) = \mu(x) \cdot_N \mu(y)$.

Exemple :
$|\cdot| : (\Sigma^\star, \cdot) \to (\mathbb{N}, +)$ est un morphisme de monoïdes.

Définition : Un langage L est dit reconnu par un monoïde M, un morphisme $\mu : \Sigma^\star \to M$ et un ensemble $P \subseteq M$ si $L = \mu^{-1}(P)$.

Exemple :
L'ensemble $\{a^{n^3} \mid n \in \mathbb{N}\}$ est reconnu par le morphisme $|\cdot|$ et l'ensemble $P = \{n^3 \mid n \in \mathbb{N}\}$.

Théorème : Un langage est régulier si et seulement s'il est reconnu par un monoïde fini.

Exemple :
L'ensemble $\{a^{2n} \mid n \in \mathbb{N}\}$ est un langage régulier. En effet, on a $M = \mathbb{Z}/2\mathbb{Z}$, $P = \{0\}$ et

$$\begin{aligned}\mu : \Sigma^\star &\longrightarrow \mathbb{Z}/2\mathbb{Z}\\ w &\longmapsto |w| \bmod 2.\end{aligned}$$

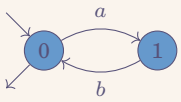


FIGURE 1.31 – Automate reconnaissant $\mu^{-1}(P) = L$

Preuve : “$\Longrightarrow$” Soit $L \in \wp(\Sigma^\star)$ reconnu par un monoïde M fini, un morphisme μ et un ensemble P : $L = \mu^{-1}(P)$. Posons $\mathcal{A} = (\Sigma', \mathbb{Q}, I, F, \delta)$ avec

$$\Sigma' = \Sigma \qquad \mathbb{Q} = M \qquad I = \{1_M\} \qquad F = P$$
$$\delta = \{(q, \ell, q') \in \mathbb{Q} \times \Sigma \times \mathbb{Q} \mid q \cdot \mu(\ell) = q'\}.$$

Montrons que $\mathcal{L}(\mathcal{A}) = L$. Soit $w \in \mathcal{L}(\mathcal{A})$. Il existe une exécution acceptante

$$1_M = q_0 \xrightarrow{w_1} q_1 \to \cdots \xrightarrow{w_n} q_n \in P.$$

Or, $\mu(w_1 \dots w_n) = \prod_{i=1}^n \mu(w_i) = q_0 \prod_{i=1}^n \mu(w_i) = q_0\mu(w_1)\cdot\prod_{i=1}^n \mu(w_i) = q_1 \prod_{i=1}^n \mu(w_i) = q_n \in P$.

□

CHAPITRE

2

ALGORITHMES PROBABILISTES

Sommaire

2.1 Introduction **79**
2.2 Algorithme de Monte-Carlo **83**
2.3 Algorithme de type Las-Vegas **84**
Annexe 2.A Hors-programme **89**

2.1 Introduction

Dans ce chapitre, on s'intéresse aux algorithmes probabilistes de deux types : Monte-Carlo et Las-Vegas. L'idée est de donner une définition plus mathématique d'un « algorithme probabiliste » et de l'influence de l'aléatoire.

Définition : Un algorithme *déterministe* est un algorithme tel que pour chaque entrée I de l'algorithme, l'exécution de l'algorithme produit toujours exactement la même suite d'états.

Remarque :
Un algorithme déterministe produit donc toujours les même sorties sur les mêmes entrées.

Définition : Un algorithme *probabiliste* est un algorithme opérant sur un ensemble $\mathscr{E}$, tel que la suite d'états obtenus par exécution de l'algorithme sur une entrée $e \in \mathscr{E}$ est une variable aléatoire.

Remarque :
Avec cette définition, un algorithme déterministe est un algorithme probabiliste.

Exemple :
On considère le problème suivant :

$$\text{Problème Tri} : \begin{cases} \text{Entrée} & : \text{un tableau } T \text{ de taille } n \\ \text{Sortie} & : T \text{ trié.} \end{cases}$$

Une réponse à ce problème est l'algorithme nommé Bozosort décrit ci-dessous. On le nomme aussi « tri aléatoire. »

Algorithme 2.1 Bozosort

Entrée T un tableau
1 : **tant que** T non trié **faire**
2 : $\quad i \leftarrow \mathcal{U}(\llbracket 1, n-1 \rrbracket)$
3 : $\quad j \leftarrow \mathcal{U}(\llbracket 1, n-1 \rrbracket)$
4 : $\quad$ Échanger i et j dans le tableau T

On étudie l'algorithme ci-dessus : il est trivialement partiellement correct (i.e. s'il est correct). En effet, par négation de la condition de boucle, on a T trié.

Le temps d'exécution de l'algorithme est difficile à estimer. L'algorithme peut ne pas terminer.

Exemple :
On considère à présent le problème ci-dessous : approximer π. L'algorithme tire des points au hasard dans un carré unité, et regarde si le point est dans le disque unité.

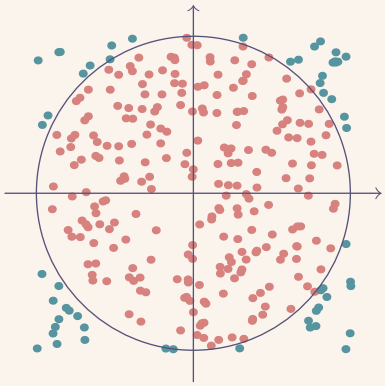


Figure 2.1 – Algorithme de Monte-Carlo pour approximer π

On compte le nombre de points dans le disque, et ceux en dehors. Avec un grand nombre de points, on approxime le ratio de l'aire du disque et de l'aire du carré. Puis, on calcule

$$4 \times \left(\frac{\#\bullet}{\#\bullet + \#\bullet} \right) \approx \pi.$$

Le temps d'exécution dépend uniquement des paramètres de précision de l'algorithme, pas des tirages. Par contre, la qualité de la réponse dépend des choix aléatoires.

Définition (Algorithme de type Las Vegas) **:** Étant donné un problème P, un algo-

rithme probabiliste répondant au problème P est dit de type Las Vegas dès lors que, s'il se termine, c'est en donnant une réponse correcte.

Définition (Algorithme de type Monte-Carlo) **:** Étant donné un problème P, un algorithme probabiliste répondant au problème P est dit de type Monte-Carlo dès lors que son temps d'exécution dépend uniquement de son entrée. L'algorithme peut cependant répondre de manière erronée au problème P avec une « certaine » probabilité.

Remarque :
Dans le cas d'un problème de décision (la réponse de l'algorithme est oui ou non), un algorithme de type Monte-Carlo est dit

— « à erreur unilatérale » s'il existe une des réponses (oui ou non) r telle que, si l'algorithme répond r, alors il a raison (r est la réponse au problème) ;

— « à erreur bilatérale » si pour chaque réponse l'algorithme se trompe avec une probabilité non nulle.

Exemple :
On considère le problème : étant donné un tableau $T \in \{0,1\}^n$ tel que T contient p fois la valeur '0', avec $0 < p < n$, on cherche si, pour $i \in [\![1, n-1]\!]$, $T[i] = 1$.

Une réponse à ce problème est un algorithme de type Monte-Carlo, comme celui ci-dessous.

Algorithme 2.2 Algorithme de Monte-Carlo pour répondre au problème

Entrée $k \in \mathbb{N}$ et T un tableau
```
1 : i ← 0
2 : pour j ∈ [[1, k]] faire
3 :     i ← 𝒰([[1, n − 1]])
4 :     si T[i] = 1 alors
5 :         retourner i
6 : retourner i
```

Mais, on peut également donner un algorithme de Las-Vegas répondant aussi au même problème.

Algorithme 2.3 Algorithme de Las-Vegas pour répondre au problème

Entrée T un tableau
```
1 : i ← 𝒰([[1, n − 1]])
2 : tant que T[i] ≠ 1 faire
3 :     i ← 𝒰([[0, n − 1]])
4 : retourner i
```

Étudions l'algorithme de Las-Vegas : la correction partielle est validée. Étudions la terminaison : fixons T un tableau de taille n contenant p occurrences de '0' avec $0 < p < n$. Notons $(X_\ell)_{\ell \in \mathcal{D}}$ la suite des variables aléatoires donnant la valeur produite par le ℓième appel à $\mathcal{U}([\![0,1]\!])$. Remarquons que $\mathcal{D}$ est une variable aléatoire : en effet c'est $[\![0, N]\!]$ pour un certain $N \in \mathbb{N}$ si l'algorithme se termine ; sinon, on a $\mathcal{D} = \mathbb{N}$. Notons A_q l'événement « l'algorithme s'arrête après q appels au générateur $\mathcal{U}([\![0, n-1]\!])$. On a donc

$$A_q = \text{“}T[X_0] = 0 \wedge T[X_1] = 0 \wedge \cdots \wedge T[X_{q-2}] = 0 \wedge T[X_{q-1}] = 1\text{”}.$$

D'où en passant aux probabilités, on a

$$P(A_q) = P(\text{“}T[X_0] = 0 \wedge T[X_1] = 0 \wedge \cdots \wedge T[X_{q-2}] = 0 \wedge T[X_{q-1}] = 1\text{”})$$

et, par indépendance, on a donc

$$P(A_q) = P(T[X_{q-1} = 1) \times \Big(\prod_{j=0}^{q-2} P(T[X_j] = 0)\Big).$$

Or, $\forall j \in [\![0, q-2]\!]$, $P(T[X_j] = 0) = \frac{p}{n}$ par uniformité, et, de plus, $P(T[X_{q-1}] = 1) = \frac{n-p}{n}$ par uniformité. On pose $\rho = \frac{p}{n}$, et donc $P(A_q) = \rho^{q-1}(1-\rho)$. Notons N l'événement « l'algorithme ne se termine pas » et calculons

$$\begin{aligned}
P(N) &= 1 - P(\bar{N}) \\
&= 1 - P\Big(\bigvee_{q\in\mathbb{N}^\star} A_q\Big) \\
&= 1 - \sum_{q\in\mathbb{N}^\star} P(A_q) \\
&= 1 - \sum_{q\in\mathbb{N}^\star} \rho^{q-1}(1-\rho) \\
&= 1 - \frac{1-\rho}{1-\rho} \\
&= 0
\end{aligned}$$

Soit $\mathcal{T}$ la variable aléatoire indiquant le temps d'arrêt de l'algorithme (en nombre d'itérations). On calcule l'espérance de $\mathcal{T}$:

$$\begin{aligned}
\mathrm{E}(\mathcal{T}) &= \sum_{t=1}^{+\infty} t \times P(\mathcal{T} = t) \\
&= \sum_{t=1}^{+\infty} t \times \rho^{t-1}(1-\rho) \\
&= (1-\rho)\sum_{t=1}^{+\infty} t \times \rho^{t-1} \\
&= (1-\rho)\sum_{t=1}^{+\infty}\sum_{k=0}^{t-1} \rho^t \\
&= (1-\rho)\sum_{k=0}^{+\infty}\sum_{t=k+1}^{+\infty} \rho^{t-1} \\
&= \cdots\cdots \text{ [1]} \\
&= \frac{1}{1-\rho}
\end{aligned}$$

Étudions maintenant l'algorithme de Monte-Carlo. Il se termine trivialement, et la probabilité d'erreur est $\underbrace{\rho \times \cdots \times \rho}_{k}$. Par exemple, pour $\rho = \frac{1}{2}$, et $k = 80$, la probabilité d'erreur est de $\frac{1}{2^{80}}$.

2.2 Algorithme de Monte-Carlo

On considère le problème : « étant donné trois matrices A, B, C de $\mathcal{M}_n(\mathbb{Z}/2\mathbb{Z})$, a-t-on $A \cdot B = C$? »

Un algorithme trivial serait de calculer $A \cdot B$ et on vérifie, point à point, que $A \cdot B = C$. La complexité cet algorithme est en $\Theta(n^3)$ à cause du produit matriciel.

Un algorithme de Monte-Carlo serait le suivant.

Algorithme 2.4 Algorithme de Monte-Carlo répondant au problème

Entrée A, B, C trois matrices et $k \in \mathbb{N}$
1 : **pour** $j \in [\![1, k]\!]$ **faire**
2 : $\quad r \leftarrow \mathcal{U}\big((\mathbb{Z}/2\mathbb{Z})^n\big)$ $\quad \triangleright\ n$
3 : $\quad r_1 \leftarrow B \cdot r$ $\quad \triangleright\ n^2$
4 : $\quad r_2 \leftarrow A \cdot r_1$ $\quad \triangleright\ n^2$
5 : $\quad r_3 \leftarrow C \cdot r$ $\quad \triangleright\ n^2$
6 : $\quad$ **si** $r_3 \neq r_2$ **alors**
7 : $\quad\quad$ **retourner** Non
8 : **retourner** Oui

Dans le pire cas, la complexité est en $k \times n^2$. On cherche la probabilité d'erreur de cet algorithme. Pour cela, on utilise le lemme suivant.

Lemme : Si $D \neq 0$, et $r \sim \mathcal{U}\big((\mathbb{Z}/2\mathbb{Z})^n\big)$, alors $P(D \cdot r = 0) \leqslant \frac{1}{2}$.

Preuve :
Si $D \in \mathcal{M}_n(\mathbb{Z}/2\mathbb{Z}) \setminus \{0\}$, alors il existe i et j tels que $D_{i,j} \neq 0$. Si $D \cdot r = 0$, on a

$$\sum_{k=1}^{n} D_{i,k} r_k = 0 \quad \text{et donc} \quad r_k = -\sum_{\substack{k=1 \\ k \neq j}}^{n} D_{i,k} r_k.$$

Donc, si $r_j \neq \sum_{\substack{k=1 \\ k \neq j}}^{n} D_{ik} r_k$, alors $P(D \cdot r \neq 0) \geqslant P\Big(r_j \neq \sum_{\substack{k=1 \\ k \neq j}}^{n} D_{i,k} r_k\Big)$. On note E_0 l'événement « $r_j = 0$ et $\sum_{\substack{k=1 \\ j \neq j}}^{n} D_{i,k} r_k = 1$ » et E_1 l'événement « $r_j = 1$ et $\sum_{\substack{k=1 \\ j \neq j}}^{n} D_{i,k} r_k = 0$. » D'où

$$P\Big(r_j \neq \sum_{\substack{k=1 \\ j \neq j}}^{n} D_{i,k} r_k\Big) = P(E_0 \vee E_1).$$

Par incompatibilité, on a $P(E_0 \vee E_1) = P(E_0) + P(E_1)$, d'où

$$\forall a \in \{0,1\}, \quad P(E_a) = P\Big(r_j = a \wedge \sum_{\substack{k=1 \\ j \neq j}}^{n} D_{i,k} r_k = 1 - a\Big)$$

et, par indépendance,

$$\forall a \in \{0,1\} \quad P(E_a) = P(r_j = a) \cdot P\Big(\sum_{\substack{k=1 \\ j \neq j}}^{n} D_{i,k} r_k = 1 - a\Big) = \frac{1}{2} P(\ldots).$$

1. à faire

D'où

$$P\Big(r_j \neq \sum_{\substack{k=1\\j\neq j}} D_{i,k} r_k\Big) = \frac{1}{2}\Big[P\Big(\sum_{\substack{k=1\\j\neq j}}^{n} D_{i,k} r_k = 1\Big) + P\Big(\sum_{\substack{k=1\\j\neq j}}^{n} D_{i,k} r_k = 0\Big)\Big]$$

et, par incompatibilité,

$$P\Big(r_j \neq \sum_{\substack{k=1\\j\neq j}} D_{i,k} r_k\Big) = \frac{1}{2}\frac{1}{2} P\Big(\sum_{\substack{k=1\\j\neq j}} D_{i,k} r_k \in \{0,1\}\Big) = \frac{1}{2}.$$

□

D'où, l'algorithme ci-dessous est tel que sa probabilité d'échec est de $\frac{1}{2^k}$. Or, l'algorithme a une complexité de $\mathcal{O}(k\,n^2)$.

2.3 Algorithme de type Las-Vegas

On étudie le tri rapide. On considère les fonctions "Partitionner," puis "Tri Rapide."

Algorithme 2.5 Fonction "Partitionner" utilisée dans le tri rapide

Entrée T le tableau à trier, g, d et p trois entiers (bornes du tableau)
Sortie un entier J et le sous-tableau $T[g..d]$ est modifié en $\bar{T}$ de sorte que $\bar{T}^{2}[J] = \underline{T}[p]$, et $\forall i \in [\![g, J-1]\!], \bar{T}[i] \leqslant \bar{T}[J]$, et $\forall i \in [\![J+1, d]\!], \bar{T}[i] \geqslant \bar{T}[J]$, et $\forall i \in [\![0, n-1]\!] \setminus [\![g, d]\!]$, $\bar{T}[i] = \underline{T}[i]$, et $\bar{T}$ est une permutation de $\underline{T}$.

```
1 : Échanger(T, J, d)
2 : J ← g
3 : I ← g
4 : tant que I < d faire
5 :     si T[I] > T[d] alors        ▷ Cas "T[I] > pivot"
6 :         I ← I + 1
7 :     sinon                       ▷ Cas "T[I] ⩽ pivot"
8 :         Échanger(T, I, J)
9 :         J ← J + 1
10 :        I ← I + 1
11 : Échanger(T, J, d)
12 : retourner J
```

Remarque :
On admet que $\bar{T}$ est une permutation de $\underline{T}$. On admet également que, $\forall i \in [\![0, n-1]\!] \setminus [\![g, d]\!]$, $\bar{T}[i] = \underline{T}[i]$.

Lemme : "Partitionner" est correct.

Preuve :
On considère

$$(\mathcal{J}) : \begin{cases} \forall k \in [\![g, I]\!]\,,\ T[k] \leqslant T[d] & (1) \\ \forall k \in [\![J, I-1]\!]\,,\ T[k] > T[d] & (2)\ . \\ g \leqslant J \leqslant I \leqslant d & (3) \end{cases}$$

— Montrons que $\mathcal{J}$ est vrai initialement : à l'initialisation, $I = g$ et $J = g$, donc les trois propriétés $(\mathcal{J})$ sont trivialement vraies.
— Montrons que l'invariant $(\mathcal{J})$ se propage. Notons $\underline{I}$, $\underline{J}$, et $\underline{T}$ les valeurs de I, J et T avant itération

2. La notation $\bar{T}$ représente le tableau T après l'algorithme, et la notation $\underline{T}$ représente le tableau T avant l'algorithme.

de boucle. Notons également $\bar{I}$, $\bar{J}$ et $\bar{T}$ les valeurs de I, J et T après cette même itération de boucle. Supposons que $\underline{I}$, $\underline{J}$ et $\underline{T}$ vérifient $(\mathcal{I})$, et la condition de boucle. Montrons que $\bar{I}$, $\bar{J}$ et $\bar{T}$ vérifient $(\mathcal{I})$. On a donc, d'après $(\mathcal{I})$,

$$\begin{cases} \forall k \in [\![g, \underline{J}-1]\!]\,,\ \underline{T}[k] \leqslant \underline{T}[d] \\ \forall k \in [\![\underline{T}, \underline{I}-1]\!]\,,\ \underline{T}[k] > \underline{T}[d] \\ g \leqslant \underline{J} \leqslant \underline{I} \leqslant d. \end{cases}$$

Mais aussi, d'après la condition de boucle, $\underline{I} \leqslant d$. Mais également, d'après le programme,

- si $\underline{T}[\underline{I}] > \underline{T}[d]$, alors $\bar{I} = \underline{I} + 1$, $\bar{T} = \underline{T}$, et $\bar{J} = \underline{J}$;
- sinon si $\underline{T}[\underline{I}] \leqslant \underline{T}[d]$, et donc $\bar{J} = \underline{J} + 1$, $\bar{I} = \underline{I} + 1$, $\forall k \in [\![0n, -1]\!] \setminus \{\underline{I}, \underline{J}\}$, $\bar{T}[k] = \underline{T}[k]$, et $\bar{T}[\bar{I}] = \underline{T}[\underline{J}]$, et $\bar{T}[\underline{J}] = \underline{T}[\underline{I}]$.

Cas 1 $\underline{T}[\underline{I}] > \underline{T}[d]$, alors

(3) $g \leqslant \underline{J} = \bar{J} \leqslant \underline{I} < \bar{I} \leqslant d$.

(1) Soit $k \in [\![g, \bar{J}-1]\!]$, on a donc $k \in [\![g, \underline{J}-1]\!]$, et donc $\bar{T}[k] = \underline{T}[k] \leqslant \underline{T}[d] = \bar{T}[d]$.

(2) Soit $k \in [\![\bar{J}, \bar{I}-1]\!]$,

- si $k \in [\![\underline{J}, \underline{I}-1]\!]$, alors $\bar{T}[k]$ă $= \underline{T}[k] > \underline{T}[d] = \bar{T}[d]$.
- si $k = \bar{I} - 1 = \underline{I}$, par condition `if`, alors $\bar{T}[\underline{I}] = \underline{T}[\underline{I}] > \underline{T}[d] = \bar{T}[d]$.

Cas 2 $\underline{T}[\underline{I}] \leqslant \underline{T}[d]$

(3) On a $\underline{J} \leqslant \underline{I}$, donc $\underline{J} + 1 \leqslant \underline{I} + 1$, d'où $g \leqslant \underline{J} + 1 = \bar{J} \leqslant \bar{I} \leqslant d$.

(1) Soit $k \in [\![g, \bar{J}-1]\!]$, donc

- si $k \in [\![g, \underline{J}-1]\!]$, alors $\bar{T}[k] = \underline{T}[k] \leqslant \underline{T}[d] = \bar{T}[d]$.
- si $k = \bar{J} - 1 = \underline{J}$, alors $\bar{T}[k] = \bar{T}[\underline{J}] = \underline{T}[\underline{I}] \leqslant \underline{T}[d] = \bar{T}[d]$.

(2) Soit $k \in [\![\bar{J}, \bar{I}-1]\!]$, alors

- si $k \in [\![\bar{J}, \underline{J}-1]\!]$, alors, comme $\bar{J} \geqslant \underline{J}$, et donc $\bar{T}[k] = \underline{T}[k] > \underline{T}[d] = \bar{T}[d]$.
- si $k = \bar{I} - 1 = \underline{I}$, et donc $\bar{T}[k] = \bar{T}[\underline{I}] = \bar{T}[\underline{J}] > \underline{T}[d] = \bar{T}[d]$.

Ainsi, $(\mathcal{I})$ est un invariant, et donc, en sortie de boucle, I, J et T sont tels que

$$\left.\begin{array}{l} \forall k \in [\![g, J-1]\!]\,,\ T[k] \leqslant T[d] \\ \forall k \in [\![J, I-1]\!]\,,\ T[k] > T[d] \\ g \leqslant J \leqslant I \leqslant d \end{array}\right\} \quad \text{et} \quad I \geqslant d,$$

la négation de la condition de boucle. On a donc $I = d$, et donc en fin de programme,

$$\forall k \in [\![g, J-1]\!]\,,\ T[k] \leqslant T[J] \quad \text{et} \quad \forall k \in [\![J+1, I]\!]\,,\ T[k] > T[J].$$

□

Algorithme 2.6 Tri rapide

Entrée T un tableau, g et d les bornes de ce tableau

```
1 : si d > g alors
2 :     p ← ChoixPivot(T, g, d)
3 :     J ← Partition(T, g, d, p)
4 :     TriRapide(T, g, J − 1)
5 :     TriRapide(T, J + 1, d)
```

La fonction "Tri(T)" est donc définie comme TriRapide($T, 0, n-1$) si T est un tableau de taille n.

Étudions rapidement l'influence du choix du pivot.

Cas 1 On définit "ChoixPivot$(T, g, d) = g$." Ainsi

À faire : Figure

Figure 2.2 – Arbre des appels récursifs de "TriRapide" avec le pivot à gauche

Ainsi, la complexité de cet algorithme, avec ce choix de pivot, est en $(n-1) + (n-2) + (n-3) + \cdots + 2 = \Theta(n^2)$.

Cas 2 On définit maintenant le choix du pivot comme l'indice de la médiane.

À faire : Figure

Figure 2.3 – Arbre des appels récursifs de "TriRapide" avec le pivot à la médiane

Rédigeons-le rigoureusement : soit $C_n = \max_{T \text{ tableau de taille } n} C(T)$. Posons $(u_p)_{p\in\mathbb{N}} = (C_{2^p})_{p\in\mathbb{N}}$. D'après l'algorithme de "TriRapide," on a

$$\begin{aligned} u_{p+1} &= 2^{p+1} - 1 + u_p + u_p \\ &= 2^{p+1} - 1 + 2u_p \\ &= (2^{p+1} - 1) + 2(2^p - 1) + 2^2 u_{p-1} \\ &= 2^{p+1} - 1 + 2^{p+1} - 2 + 2^2 u_{p-1} \\ &= 2^{p+1} - 1 + 2^{p+1} - 2 + 2^2(2^{p-1} - 1 + 2u_{p-2}) \\ &= 2^{p+1} - 1 + 2^{p+1} - 2 + 2^{p+1} - 2^2 + 2^3 u_{p-2}. \end{aligned}$$

On a donc $u_0 = 1$ et $u_p = p \times 2^p - (2^p - 1)$. Or, la suite $(c_n)_{n\in\mathbb{N}}$ est croissante. Or,

$$\forall n \in \mathbb{N},\ 2^{\lfloor \log_2 n \rfloor} \leqslant n \leqslant 2^{\lfloor \log_2 n \rfloor + 1}$$

donc

$$u_{\lfloor \log_2 n \rfloor} \leqslant C_n \leqslant u_{\lfloor \log_2 n \rfloor + 1}.$$

D'où

$$c_n \leqslant (\lfloor \log_2 n \rfloor + 1) \times 2^{\lfloor \log_2 n \rfloor + 1} - 2^{\lfloor \log_2 n \rfloor - 1}.$$

Et donc, on en déduit que $c_n = \Theta\big(n \log_2(n)\big)$.

Remarque (Notations) :
On fixe un tableau T de taille n. De plus, on suppose dans toute la suite, que $T \in \mathfrak{S}_n$.[a] On note alors $X_g^d[T]$ la variable aléatoire indiquant le nombres de comparaisons effectuées par l'algorithme TriRapide(T, g, d), dès lors que $T(\llbracket g, d \rrbracket) \subseteq \llbracket g, d \rrbracket$.

On note de plus, $\mathrm{E}\big[X_g^d[T]\big]$ l'espérance de cette variable aléatoire.

a. T est une permutation de n éléments. Ici, $\mathfrak{S}_n$ représente l'ensemble des permutations de $\llbracket 1, n \rrbracket$.

Théorème : Le nombre moyen de comparaisons effectuées par l'algorithme de tri rapide pour une entrée T de taille n est équivalent à $2n \ln n$. Autrement dit,

$$\mathrm{E}\big[X_0^{n-1}[T]\big] \sim 2n \ln n.$$

Preuve :
- Lorsque $d \leqslant g$, alors $X_g^d[T] = 0$.
- Lorsque $g < d$,
  - dans l'éventualité d'un choix de pivot d'indice $p \in \llbracket g, d \rrbracket$, le nombre de comparaisons est alors

$$\underbrace{d - g - 1}_{\text{coût de Partition}(T,\, g,\, d,\, p)} + \overbrace{X_g^{T[p]-1}\left[T^{g,d,p}\right]}^{\text{coût du } 1^{\text{er}} \text{ appel récursif}} + \underbrace{X_{T[p]+1}^{d}\left[T^{g,d,p}\right]}_{\text{coût du } 2^{\text{nd}} \text{ appel récursif}}$$

où $T^{g,d,p}$ est le tableau T après appel à Partition(T, g, d, p). D'où

$$\begin{aligned}
\mathrm{E}\left[X_g^d[T]\right] &= \sum_{j=g}^{d} \mathrm{E}\left[X_g^d[T]\Big|_{p=j}\right] \cdot P(p=j) \\
&= \frac{1}{d-g+1} \sum_{j=g}^{d} \mathrm{E}\left[X_g^d[T]\Big|_{p=j}\right] \\
&= \frac{1}{d-g+1} \sum_{j=g}^{d} \Big(d-g-1+\mathrm{E}\left[X_g^{T[j]-1}\left[T^{g,d,j}\right]\right] \\
&\qquad\qquad + \mathrm{E}\left[X_{T[j]+1}^{d}\left[T^{g,d,j}\right]\right]\Big) \\
&= (d-g-1) + \frac{1}{d-g+1} \sum_{k=g}^{d} \Big(\mathrm{E}\left[X_g^{k-1}\left[T^{g,d,T^{-1}[k]}\right]\right] \\
&\qquad\qquad + \mathrm{E}\left[X_{k+1}^{d}\left[T^{g,d,T^{-1}[k]}\right]\right]\Big)
\end{aligned}$$

car $T : [\![g, d]\!] \to [\![g, d]\!]$ est une bijection.
Soit donc la suite $(c_\ell)_{\ell\in\mathbb{Z}}$ définie par

$$\begin{cases} \forall \ell \in \mathbb{Z}^-, & c_\ell = 0 \\ \forall \ell \in \mathbb{N}^\star, & c_\ell = \mathrm{E}\left[X_0^\ell[\mathrm{id}]\right]. \end{cases}$$

On a alors

$$\begin{aligned}
\forall \ell \in \mathbb{N}^\star, \quad c_\ell &= (\ell-1) + \frac{1}{\ell-1} \sum_{k=0}^{\ell} (c_{k-1} - c_{\ell-k-1}) \\
c_\ell &= (\ell-1) + \frac{2}{\ell+1} \sum_{k=1}^{\ell-1} c_k \\
(\ell+1)c_\ell &= (\ell+1)(\ell-1) + 2\sum_{k=1}^{\ell-1} c_k
\end{aligned}$$

D'où, $\ell c_\ell = \ell(\ell-2) + 2\sum_{k=1}^{\ell-2} c_k$, et donc

$$(\ell+1)c_\ell - \ell c_{\ell-1} = \ell^2 - 1 - \ell^2 + 2\ell + 2c_{\ell-1}.$$

On en déduit donc que

$$(\ell+1)c_\ell - (\ell+2)c_{\ell-1} = 2\ell - 1$$

et donc

$$\frac{c_\ell}{\ell+2} - \frac{c_{\ell-1}}{\ell+1} = \frac{2\ell-1}{(\ell+1)(\ell+2)}.$$

Soit alors $(u_\ell)_{\ell\in\mathbb{N}} = \left(c_\ell/(\ell+2)\right)_{\ell\in\mathbb{N}}$, et $u_0 = 0$. Alors

$$u_\ell = \sum_{k=1}^{n} (u_k - u_{k-1}) = \sum_{k=1}^{n} \frac{2k-1}{(k+1)(k+2)}$$

or $\frac{2k-1}{(k+1)(k+2)} \sim \frac{2}{k}$, et $\sum_{k\geqslant 1} \frac{2}{k}$ diverge donc $u_\ell \sim \sum_{k=1}^{\ell} \frac{2}{k} \sim 2\ln\ell$. On en déduit donc que $c_\ell \sim 2\ell\ln\ell$.

□

Dans la preuve précédente, on a utilisé le lemme suivant.

Lemme : Soit $(g,d) \in \mathbb{N}^2$ et soit $T \in \mathfrak{S}_n$ une permutation telle que $T(\llbracket g,d \rrbracket) \subseteq \llbracket g,d \rrbracket$.

$$\mathrm{E}\Big[X_g^d[T]\Big] = \mathrm{E}\Big[X_0^{d-g}[\mathrm{id}]\Big].$$

Preuve (par récurrence forte sur $d-g=\ell \in \mathbb{N}$) :
— Soient $(g,d) \in \mathbb{N}^2$ tel que $d-g=0$. Soit $T \in \mathfrak{S}_n$ telle que $T(\llbracket g,d \rrbracket) \subseteq \llbracket g,d \rrbracket$. On a bien $X_g^d[T] = 0 = X_0^{d-g}[\mathrm{id}]$.
— On remarque, par hypothèse de récurrence,

$$\mathrm{E}\Big[X_g^{k-1}\Big[\overbrace{T^{g,d,T^{-1}[k]}}^{k-1-g<d-g}\Big]\Big] = \mathrm{E}\Big[X_0^{k-1-g}[\mathrm{id}]\Big]$$

et

$$\mathrm{E}\Big[X_{k-1}^{d}\Big[T^{g,d,T^{-1}[k]}\Big]\Big] = \mathrm{E}\Big[X_0^{d-k-1}[\mathrm{id}]\Big].$$

On a alors

$$\begin{aligned}
&\mathrm{E}\Big[X_g^d[T]\Big] \\
&= (d-g-1) + \frac{1}{d-g+1}\sum_{k=g}^{d}\Big(\mathrm{E}\Big[X_g^{k-1}\Big[T^{g,d,T^{-1}[k]}\Big]\Big] + \mathrm{E}\Big[X_{k-1}^{d}\Big[T^{g,d,T^{-1}[k]}\Big]\Big]\Big) \\
&= (d-g-1) + \frac{1}{d-g-1}\sum_{k=g}^{}\Big(\mathrm{E}\Big[X_0^{k-1-g}[\mathrm{id}]\Big] + \mathrm{E}\Big[X_0^{d-k-1}[\mathrm{id}]\Big]\Big).
\end{aligned}$$

Ceci est vrai pour tout $T \in \mathfrak{S}_n$ telle que $T(\llbracket g,d \rrbracket) \subseteq \llbracket g,d \rrbracket$, donc

$$\begin{aligned}
\mathrm{E}[X_g^d[\mathrm{id}]] &= (d-g-1) + \frac{1}{d-g-1}\sum_{k=g}^{}\Big(\mathrm{E}\Big[X_0^{k-1-g}[\mathrm{id}]\Big] + \mathrm{E}\Big[X_0^{d-k-1}[\mathrm{id}]\Big]\Big) \\
&= \mathrm{E}[X_g^d[T]]
\end{aligned}$$

□

Annexe 2.A Hors-programme

CHAPITRE

3

APPRENTISSAGE

Sommaire

3.1 Motivation **91**
3.2 Vocabulaire **91**
3.3 Apprentissage supervisé **92**
3.3.1 k plus proches voisins 93
3.3.2 Arbres k-dimensionnels 94
3.3.3 Algorithme ID3 95
3.4 Apprentissage non supervisé **101**
3.4.1 Algorithme HAC, classification hiérarchique ascendant 101
3.4.2 k-moyenne 102

3.1 Motivation

L'INTELLIGENCE ARTIFICIELLE est vue comme un « objet magique » mais ce n'est pas le cas : c'est ce que nous allons étudier dans ce chapitre. Il existe plusieurs méthodes permettant l'apprentissage : descente de gradient, k-plus proches voisins, …

La base de donnée la plus utilisée est MNIST : elle contient 60 000 images de 28×28 pixels représentant un chiffre, et le chiffre correspondant. L'idée de l'apprentissage est de « deviner » le chiffre dessiné en connaissant l'image.

3.2 Vocabulaire

Définition : On appelle *signature de données* un n-uplet de paires nom, ensemble ; on le typographie

$$(\text{nom}_1 : S_1, \text{nom}_2 : S_2, \ldots, \text{nom}_n : S_n).$$

Exemple : 1. $\mathbb{S}_1 = (\text{titre} : \text{string}, \text{longueur} : \mathbb{N}, \text{date} : \mathbb{N})$,
2. $\mathbb{S}_2 = (x : \mathbb{R}, y : \mathbb{R})$,
3. $\mathbb{S}_3 = (R : [\![0, 255]\!], G : [\![0, 255]\!], B : [\![0, 255]\!])$.

Définition : Étant donné une signature de données $\mathbb{S} = (\text{nom}_1 : S_1, \ldots, \text{nom}_n : S_n)$, on appelle *donnée* un vecteur

$$\bar{v} = (v_1, v_2, \ldots, v_n) \in S_1 \times S_2 \times \cdots \times S_n.$$

Exemple : 1. (“2001, a space odyssey”, 139, 1968) est une donnée/un vecteur de signature $\mathbb{S}_1$.
2. $(\pi, \sqrt{2})$ est une donnée/un vecteur de signature $\mathbb{S}_2$.

Définition : Étant donné une signature de données $\mathbb{S}$, on appelle *jeu de données* un ensemble fini de vecteurs de signature $\mathbb{S}$.

Définition : Étant donnée une signature de données $\mathbb{S}$ et un ensemble de classes $\mathscr{C}$, on appelle *jeu de données classifié* la donnée
— d'un jeu de données S,
— d'une fonction $f : S \to \mathscr{C}$ de classification.

3.3 Apprentissage supervisé

L'objectif de cette section est de construire des fonctions de classification, à partir d'un jeu de données classifié.

Définition : Étant donné une signature de données $\mathbb{S}$, et un ensemble de classes $\mathscr{C}$, on appelle *fonction de classification* une fonction des données de signature $\mathbb{S}$ dans $\mathscr{C}$.

Remarque :
On discutera de la « qualité » d'une fonction de classification en fonction de ses résultats sur les données d'un jeu de données et sur des exemples de tests.

3.3.1 k plus proches voisins

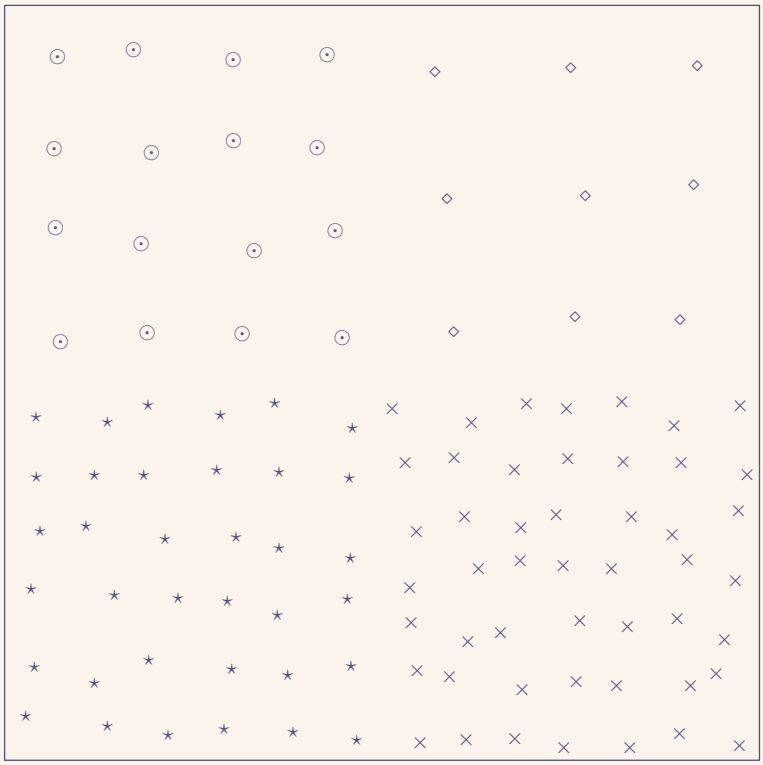


FIGURE 3.1 – Représentation de l'algorithme des k plus proches voisins

Algorithme 3.1 k-NN (k *nearest neighbors*)

Entrée Un jeu de données classifié (S, c), un vecteur d'entrée $\vec{v}$
1 : On trie S par distance croissante de v en d_1, d_2, …, d_k, d_{k+1}, …
2 : Soit D un dictionnaire de $\mathscr{C}$ vers $\mathbb{N}$ initialisé à 0 [1]
3 : **pour** $j \in [\![1, k]\!]$ **faire**
4 : $\quad D\big[c(d_j)\big] \leftarrow D\big[c(d_j)\big] + 1$
5 : **retourner** $\arg\max_{d \in \mathscr{C}} D[d]$

REMARQUE :
On doit avoir $k \leqslant n$, et l'espace doit être muni d'une distance. Les résultats de l'algorithme dépendent fortement du jeu de données, du paramètre k et de la distance choisie.

Matrice de confusion

Définition : On appelle *matrice de confusion* d'un algorithme de prédiction $\mathscr{A}$ sur un jeu de données classifié (T, c), la matrice

$$\Big(\mathrm{Card}\{t \in T \mid \mathscr{A}(t) = i \text{ et } c(t) = j\}\Big)_{(i,j)\in\mathscr{C}^2}.$$

1. où toutes les valeurs sont initialisées à 0, pas un dictionnaire vide

Dans le cas particulier d'une classification $(\boldsymbol{V}, \boldsymbol{F})$, on nomme

$\mathcal{A}$ \ vrai	$\boldsymbol{F}$	$\boldsymbol{V}$
$\boldsymbol{F}$	vrai négatif	faux négatif
$\boldsymbol{V}$	faux positif	vrai positif

TABLE 3.1 – Matrice de confusion dans le cas d'une classification en $\boldsymbol{V}$ et $\boldsymbol{F}$

Comment améliorer la performance de l'algorithme des k plus proches voisins ? En dimension 1, on peut utiliser une dichotomie. Mais, dans des dimensions plus grandes, l'ordre lexicographique, et l'ordre produit ne fonctionnent pas. Mais, on peut appliquer une "dichotomie" en changeant de dimension. Par exemple, en deux dimension, on obtient la dichotomie ci-dessous.

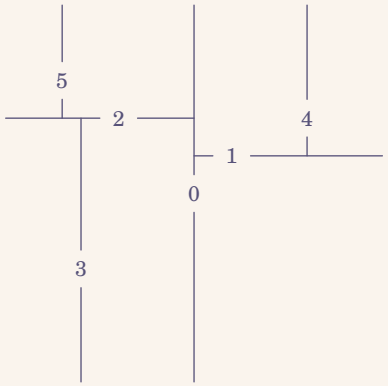


FIGURE 3.2 – Représentation de la "dichotomie" en dimension 2

Pour représenter cette structure de données, on utilise un arbre binaire comme montré ci-dessous. Cet arbre est appelé un arbre k-dimensionnels.

FIGURE 3.3 – Arbre 2-dimensionnel représentant la "dichotomie" précédente

3.3.2 Arbres k-dimensionnels

REMARQUE (Notations) :
Étant donné un jeu de données S, on note pour $v \in S$,

$$S^{\leqslant_i v} = \{u \in S \mid u_i \leqslant v_i\} \qquad \text{et} \qquad S^{>_i v} = \{u \in S \mid u_i > v_i\}.$$

Algorithme 3.2 "F" : Fabrication d'un arbre k-dimensionnel

Entrée $\mathscr{V}$ un jeu de données et $i \in [\![0, n-1]\!]$, où n est la dimension des données

si $\mathscr{V} = \varnothing$ **alors**
  retourner Vide
sinon
  On cherche $v \in \mathscr{V}$ tel que v_i est la médiane de $\{u_i \mid u \in \mathscr{V}\}$
  retourner Nœud$\Big((v,i), \mathrm{F}\big((\mathscr{V} \setminus \{v\})^{\leqslant_i v}, i+1 \bmod n\big), \mathrm{F}\big((\mathscr{V} \setminus \{v\})^{>_i v}, i+1 \bmod n\big)\Big)$

Algorithme 3.3 "R" : Recherche du point le plus proche

Entrée Un arbre k-dimensionnel et un vecteur v

si T est vide **alors**
  retourner None
sinon
  Nœud$\big((u,i), G, D\big) \leftarrow T$
  si $u_i \leqslant v_i$ **alors**
    $W \leftarrow \mathrm{R}(D, v)$
    si $W =$ None **alors**
      $W' \leftarrow \mathrm{R}(G, v)$
      si $W' = \varnothing$ **alors**
        retourner Some(u)
      sinon
        Some$(z) \leftarrow W'$
        retourner le plus proche de v entre u et z
    sinon
      Some$(w) \leftarrow W$
      si $v_i - u_i \leqslant d(w, v)$ **alors** ▷ $d(w,v)$ *représente la distance entre* w *et* v
        $W' \leftarrow \mathrm{R}(G, v)$
        si $W' =$ None **alors**
          retourner Some(plus proche de v entre u et w)
        sinon
          Some$(z) \leftarrow w$
          retourner Some(plus proche de v entre u, z, et w)
      sinon
        retourner W
  sinon
    ⟨comme le cas précédent⟩

3.3.3 Algorithme ID3

L'algorithme des k plus proches voisins (sans arbres k-dimensionnels) n'a pas de *phase d'apprentissage* : en effet, les données ne sont pas réorganisées. Mais, par exemple, pour l'utilisation des arbres k-dimensionnels, les données sont réorganisées dans un arbre.

Ce qu'on aimerai avoir, c'est des *bordures* entre les différentes classes. Par exemple, dans l'exemple précédent, on aimerai avoir les différentes zones ci-dessous.

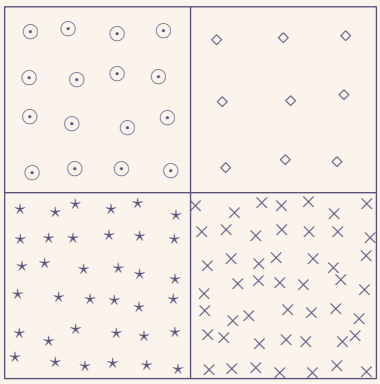


FIGURE 3.4 – Représentation de *bordures* entre les différentes classes

De ces zones, on peut construire un algorithme qui classifie les données, que l'on représente sous forme d'arbre. Ce type d'arbre est un *arbre de décision*. Dans l'exemple précédent, on peut donc créer l'arbre ci-dessous. [2]

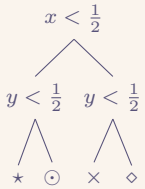


FIGURE 3.5 – Arbre de décision pour la classification

Dans le reste de cette section, on s'intéresse uniquement à des données de $\mathbb{B}^n$ (une liste de n booléens) pour un certain $n \in \mathbb{N}$.

On considère l'exemple dont les données ci-dessous.

Transport	Moteur	Rails	Sous-terre	$\geqslant 320$ km/h	Train ?
A380	✓	✗	✗	✓	✗
TGV	✓	✓	✗	✓	✓
Métro	✓	✓	✓	✗	✓
Wagonnet	✗	✓	✓	✗	✗
Draisine	✗	✓	✗	✗	✗
Tram	✓	✓	✗	✗	✓

TABLE 3.2 – Exemple de données

Entropie

Définition : Étant donné un variable aléatoire finie X à valeurs dans E. On note $p_X : E \to [0,1]$ sa loi de probabilité :

$$\forall x \in E, \quad p_X(x) = P(X = x).$$

2. Dans cet arbre, si un nœud représente une condition, son fils gauche représente si cette condition est vraie, et le fils droit sinon.

On définit l'*entropie* $\mathrm{H}(X)$ de cette variable aléatoire comme

$$\mathrm{H}(X) = -\sum_{x\in E} p_X(x)\ \ln\big(p_X(x)\big).$$

On prolonge $p_X(x)\ln(p_X(x))$ par continuité à la valeur 0 lorsque $p_X(x) = 0$.

Exemple :
On considère la variable aléatoire X à valeur dans $\{\bullet, \bullet\}$ telle que $P(X = \bullet) = 1$ et $P(X = \bullet) = 0$. On a

$$\mathrm{H}(X) = -0\ln 0 - 1\ln 1 = 0.$$

Exemple :
On considère la variable aléatoire X à valeur dans $\{\bullet, \bullet\}$ telle que $P(X = \bullet) = p$ et $P(X = \bullet) = 1 - p$, avec $p \in [0, 1]$. On a alors

$$\mathrm{H}(X) = -p\ln p - (1-p)\ln(1-p)$$

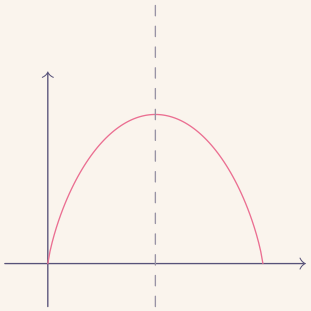


Figure 3.6 – Représentation graphique de $\mathrm{H}(X)$ en fonction de p

Lemme : Si $(p_i)_{i\in[\![1,n]\!]} \in\ ]0,1]^n$ sont tels que $\sum_{i=1}^n p_i = 1$. Soit $(q_i)_{i\in[\![1,n]\!]}$ tels que $\forall i,\ q_i = \frac{1}{n}$. On a alors

$$-\sum_{i=1}^{n} p_i \ln p_i \leqslant -\sum_{i=1}^{n} p_i \ln(q_i).$$

Preuve :
On a

$$\sum_{i=1}^{n} p_i \ln(q_i) - \sum_{i=1}^{n} p_i \ln(p_i) = \sum_{i=1}^{n} p_i \ln\left(\frac{q_i}{p_i}\right) \leqslant \ln\left(\sum_{i=1}^{n} p_i \frac{q_i}{p_i}\right) = 0.$$

□

Propriété : Soit $n = |E|$. L'entropie d'une variable aléatoire à valeurs dans E est maximale lorsque

$$\forall e \in E,\ P(X = e) = \frac{1}{n}.$$

Preuve :
On conclut, d'après le lemme précédent. □

Définition : Étant donné un jeu de données classifiés (S, c) où $c : \mathbb{S} \to E$, on appelle *entropie* de ce jeu de données l'entropie de la variable aléatoire $c(Y)$ où $Y \sim \mathcal{U}(S)$. On a donc

$$\mathrm{H}\big((S,c)\big) = -\sum_{e\in E} \frac{\operatorname{Card} c^{-1}(\{e\})}{\operatorname{Card} S} \ln\left(\frac{\operatorname{Card} c^{-1}(\{e\})}{\operatorname{Card} S}\right).$$

Définition : Étant donné un jeu de données une partition $\{S_1, S_2, \dots, S_p\}$ d'un jeu de données classifié (S, c), l'*entropie* de cette partition est la moyenne (pondérée par les cardinaux et renormalisée) :

$$\mathrm{H}\big((\{S_1,\dots,S_p\},c)\big) = \sum_{i=1}^{p} \frac{\operatorname{Card} S_i}{\operatorname{Card} S}\, \mathrm{H}\big((S_i,c)\big).$$

L'entropie de $\{w, a, t, d, r, m\}$ est $\mathrm{H} = -\frac{3}{6}\ln\left(\frac{3}{6}\right) - \frac{3}{6}\ln\left(\frac{3}{6}\right) = \ln 2 \simeq 0{,}69$. Mais, avec le découpage de l'arbre de décision ci-dessous, on obtient l'entropie

$$\begin{aligned}\mathrm{H} &= \frac{2}{6}\mathrm{H}(\{w;d\}) + \frac{4}{6}\mathrm{H}(\{a,t,r,m\})\\ &= \frac{2}{6}\times 0 + \frac{4}{6}\times\left(-\frac{1}{4}\ln\left(\frac{1}{4}\right) - \frac{3}{4}\ln\left(\frac{3}{4}\right)\right)\\ &\simeq 0{,}37.\end{aligned}$$

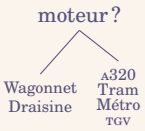


FIGURE 3.7 – Arbre de décision possible se basant sur le moteur

Avec un autre arbre (comme celui ci-dessous), on obtient une entropie différente :

$$\begin{aligned}\mathrm{H} &= \frac{1}{6}\mathrm{H}(\{a\}) + \frac{5}{6}\mathrm{H}(\{w,d,t,r,m\})\\ &= \frac{5}{6}\left(-\frac{2}{5}\ln\left(\frac{2}{5}\right) - \frac{3}{5}\ln\left(\frac{3}{5}\right)\right)\\ &\simeq 0{,}56.\end{aligned}$$

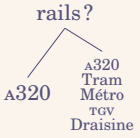


FIGURE 3.8 – Arbre de décision possible se basant sur les rails

Pour le sous-terrain, on a $\mathrm{H} = \ln 2$.

FIGURE 3.9 – Arbre de décision possible se basant sur sous-terrain

À faire : Vérifier

À faire : Autre cas

Ainsi, on choisit de commencer avec la condition "moteur" car l'entropie est la plus faible avec cette condition. Ainsi, l'arbre de décision ressemble à celui ci-dessous.

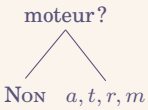


FIGURE 3.10 – Arbre de décision partiel

On réitère avec les autres conditions. L'entropie en se basant sur la vitesse est $\frac{1}{2} \ln 2 \simeq 0{,}34$. En effet, l'arbre de décision possible ressemble à celui ci-dessous.

FIGURE 3.11 – Arbre de décision possible se basant sur le moteur puis la vitesse

Mais, en se basant sur sous-terrain, on obtient une entropie de $\frac{3}{4}\left(-\frac{2}{3}\ln\left(\frac{2}{3}\right) - \frac{1}{3}\ln\left(\frac{1}{3}\right)\right) \simeq 0{,}48$.

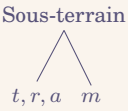


FIGURE 3.12 – Arbre de décision possible se basant sur le moteur puis sous-terrain

Et, en se basant sur les rails, on obtient une entropie de 0.

FIGURE 3.13 – Arbre de décision possible se basant sur le moteur puis les rails

On en déduit que l'arbre final de décision est celui ci-dessous.

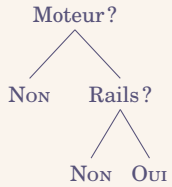


Figure 3.14 – Arbre de décision final pour la classification de trains

Les données que l'on a utilisées sont pour l'apprentissage. On teste notre arbre de décision sur les données ci-dessous.

Nom	Moteur	Rail	Sous-terre	Vitesse	Résultat de l'algorithme
Bus	✓	✗	✗	✗	✗
ter	✓	✓	✗	✗	✓
Cheval	✗	✗	✗	✗	✗
Ascenseur spatial	✓	✓	✗	✗	✓

Table 3.3 – Test de l'arbre de décision créé

Attention : il ne faut pas faire du *sur-apprentissage*, comme montré sur la figure ci-dessus, où la modélisation correspond trop aux données utilisées pour la classification.

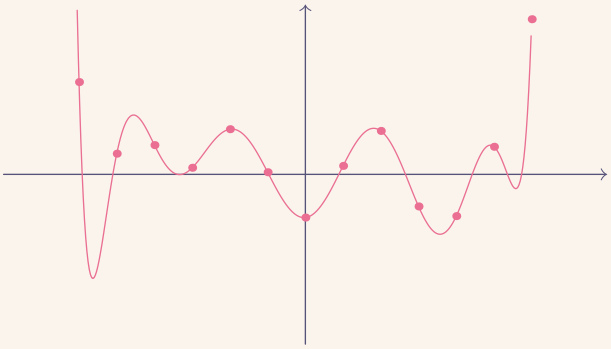


Figure 3.15 – Représentation graphique du problème de *sur-apprentissage*

Aussi, il faut faire attention aux critères : par exemple, lors de la classification de photos de chats et de chiens, les photos de chiens sont en général prises en extérieur et l'algorithme id3 aurait donc pu choisir de baser sa décision sur l'emplacement de la photo, même si elle n'importe pas dans la différenciation chat/chiens.

Autre exemple : on considère la table de données ci-dessous. Utilisons l'algorithme id3 sur ces données, et trouvons l'arbre de décision.

A	B	C	D	Classification
✓	✗	✗	✓	•
✓	✓	✗	✓	•
✓	✗	✓	✓	•
✓	✓	✓	✓	•
✗	✗	✓	✓	•
✗	✓	✓	✗	•
✗	✗	✗	✗	•
✗	✓	✗	✓	•

TABLE 3.4 – Table de données d'exemple

Pour les conditions A, B et C, on a $\mathrm{H} \simeq 0{,}63$ et, pour D, on a $\mathrm{H} = 0{,}65$. Comme on prend le 1er dans l'ordre lexicographique, on choisit la condition A. De même, on construit l'arbre ci-dessous.

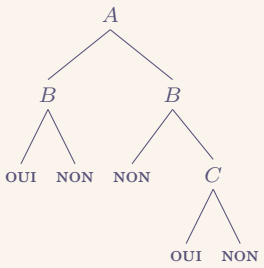


FIGURE 3.16 – Arbre de décision pour la table de données précédente

Le nom de ID3 vient de *iterative dichotomizer*.

3.4 Apprentissage non supervisé

On rappelle que l'*apprentissage non supervisé* utilise des données non classifiées. L'objectif est de les classifier. Plus rigoureusement, étant donné un jeu de données $\mathcal{D}$, on veut partitionner $\mathcal{D}$ en $\{S_1, S_2, \dots, S_n\}$ où les S_i sont regroupés par « similitude. »

> REMARQUE :
> Dans le reste de cette section, $\mathcal{D} \subseteq \mathbb{R}^p$, avec $p \in \mathbb{N}^\star$, et la notion de similitude est donnée par la distance euclidienne.

3.4.1 Algorithme HAC, classification hiérarchique ascendant

Définition : Étant donné deux sous-ensembles A et B de $\mathcal{D}$ disjoints et non-vides, on définit différentes *mesures de dissimilarité* :

- $D(A,B) = \min\{d(a,b) \mid a \in A, b \in B\}$;
- $D(A,B) = \max\{d(a,b) \mid a \in A, b \in B\}$;
- $D(A,B) = \frac{1}{|A|\,|B|} \times \sum_{(a,b)\in A\times B} d(a,b)$,

où $d(a,b)$ représente la distance entre a et b.

Algorithme 3.4 Algorithme HAC

$P \leftarrow \{\{x\} \mid x \in \mathcal{D}\}$
tant que $|P| \geqslant 2$ et ?[3] **faire**
  Soit $A, B \in P$ minimisant $D(A, B)$ avec $A \neq B$
  $P \leftarrow (P \setminus \{A, B\}) \cup \{A \cup B\}$
retourner P

3.4.2 k-moyenne

Au préalable, on fixe le nombre de classes k que l'on souhaite obtenir après exécution de l'algorithme.

Propriété : On considère la fonction

$$\begin{aligned} f : \mathbb{R}^p &\longrightarrow \mathbb{R} \\ H &\longmapsto \sum_{v \in \mathcal{D}} \|H - v\|_2^2. \end{aligned}$$

Cette fonction atteint son minimum en le barycentre G.

Preuve :
Soit G le barycentre de $\mathcal{D}$. Soit $H \in \mathbb{R}^p$. Montrons que $f(H) > f(G)$. On calcule

$$\begin{aligned} f(H) &= f(H - G + G) \\ &= \sum_{v \in \mathcal{D}} \Big\langle (H - G + G) - v \,\Big|\, (H - G + G) - v \Big\rangle \\ &= \sum_{v \in \mathcal{D}} \Big(\langle G - v \mid G - v \rangle + 2\langle H - G \mid H - v \rangle + \langle H - G \mid H - G \rangle \Big) \\ &= f(G) + \underbrace{|\mathcal{D}| \cdot \|H - G\|_2^2}_{\geqslant 0} + \underbrace{2\Big\langle H - G \,\Big|\, \sum_{v \in \mathcal{D}} (G - v) \Big\rangle}_{= 0 \text{ par définition du barycentre}} \end{aligned}$$

Donc, si $H \neq G$, on a bien $f(H) > f(G)$. □

Dans la suite de cette sous-section, on s'intéresse à des partitionnement particuliers : étant donné une famille $(m_i)_{i \in [\![1,k]\!]}$ de vecteurs de $\mathbb{R}^p$, et un entier $j \in [\![1, k]\!]$, on définit

$$\mathcal{C}_j^m = \Big\{ v \in \mathcal{D} \;\Big|\; \operatorname*{arg\,min}_{i \in [\![1,k]\!]} \|v - m_i\|_2 = j \Big\}.$$

Remarque :
En cas d'égalité ($\|v - m_{i_1}\|_2 = \|v - m_{i_2}\|_2$), le "arg min" retourne le plus petit indice : $\min(i_1, i_2)$.

Étant donné une famille $(m_i)_{i \in [\![1,k]\!]}$ de vecteurs de $\mathbb{R}^p$, on a donc un partitionnement $\{\mathcal{C}_j^m \mid j \in [\![1, k]\!]\}$.

3. où, pour ?, on peut choisir (1) un critère sur le nombre de classes (2) un critère sur la valeur de la plus petite dissimilarité

Définition : Étant donné une famille $(m_i)_{i\in[\![1,k]\!]}$ de vecteurs de $\mathbb{R}^p$, on définit

$$L(m) = \sum_{k=1}^{n}\Big(\sum_{v\in\mathscr{C}_i^m} \|v-m_i\|_2^2\Big).$$

Pour minimiser localement la fonction L, on aimerait que les $(m_i)_{i\in[\![1,k]\!]}$ soient les barycentres des $(\mathscr{C}_i^m)_{i\in[\![1,k]\!]}$. Mais, la définition des $(\mathscr{C}_i^m)$ dépend de la position des (m_i). Ainsi, en itérant ce procédé, en modifiant les (m_i) pour être les barycentres des $(\mathscr{C}_i^m)$ respectifs, puis en modifiant les $(\mathscr{C}_i^m)$, la famille $(m_i)_{i\in[\![1,k]\!]}$ converge vers les barycentres des classes.

Algorithme 3.5 k-moyenne

Entrée $\mathscr{D}$ et k
1 : $(m_i)_{i\in[\![1,k]\!]} \leftarrow k$ vecteurs de $\mathscr{D}$
2 : stable $\leftarrow \boldsymbol{F}$
3 : **tant que** ¬stable **faire**
4 : $\quad C \leftarrow (\mathscr{C}_i^m)_{i\in[\![1,k]\!]}$
5 : $\quad m' \leftarrow \big(\text{barycentre}(\mathscr{C}_i)\big)_{i\in[\![1,k]\!]}$
6 : $\quad$ stable $\leftarrow m \overset{?}{=} m'$
7 : $\quad m \leftarrow m'$
8 : **retourner** C

CHAPITRE

4

CALCULABILITÉ, DÉCIDABILITÉ, COMPLEXITÉ

Sommaire

4.1 Remarques mathématiques **105**
4.2 Problèmes **106**
4.3 Décidabilité **107**
4.3.1 Modèles de calcul 108
4.3.2 Décidabilité 109
4.3.3 Langages et problèmes de décision 110
4.3.4 Sérialisation 111
4.3.5 Machine universelle 112
4.3.6 Théorème de l'Arrêt 113
4.3.7 Réduction 114
4.4 Classe **P** et **NP** **115**
4.4.1 Complexité d'une machine 115
4.4.2 Classe **P** 116
4.4.3 Classe **NP** 117
4.4.4 **NP**-difficile 118

Au chapitre 1, on s'est intéressé aux langages réguliers, et aux automates qui est un modèle de calcul relativement simple. Dans ce chapitre, on s'intéresse à un modèle plus puissant : un *ordinateur*, on va notamment re-définir la notion de *problème*. Le choix classique de définition d'un ordinateur n'est pas celui qui a été choisi au programme : un programme OCaml.

4.1 Remarques mathématiques

Définition : Étant donné une relation $\mathcal{R}$ sur $\mathscr{E} \times \mathscr{F}$, on dit que $\mathcal{R}$ est

— *totale à gauche* dès lors que $\forall e \in \mathscr{E},\ \exists f \in \mathscr{F},\ (e,f) \in \mathcal{R}$;

— *déterministe* dès lors que $\forall e \in \mathscr{E},\ \forall (f,f') \in \mathscr{F}^2$, si $(e,f) \in \mathcal{R}$ et $(e,f') \in \mathcal{R}$, alors $f = f'$.

Définition : On appelle *fonction totale* de $\mathscr{E}$ dans $\mathscr{F}$ une relation sur $\mathscr{E}\times\mathscr{F}$ déterministe et totale à gauche.

Définition : On appelle *fonction partielle* de $\mathscr{E}$ dans $\mathscr{F}$ une relation sur $\mathscr{E}\times\mathscr{F}$ déterministe. On note alors

$$\mathrm{def}(f) = \{x \in \mathscr{E} \mid \exists y \in \mathscr{F},\ (x,y) \in f\}.$$

Remarque :
Soit f une fonction partielle de $\mathscr{E}$ dans $\mathscr{F}$ tel que $\square \notin \mathscr{F}$, alors on peut completer f en une fonction totale de $\mathscr{E}$ dans $\mathscr{F}\cup\{\square\}$ de la manière suivante :

$$\begin{aligned} f : \mathscr{E} &\longrightarrow \mathscr{F}\cup\{\square\} \\ x &\longmapsto \begin{cases} f(x) & \text{si } x \in \mathrm{def}(f) \\ \square & \text{sinon.} \end{cases} \end{aligned}$$

4.2 Problèmes

Définition : Étant donnés un ensemble d'entrée $\mathscr{E}$, un ensemble de sortie $\mathcal{S}$, on appelle *problème* sur $\mathscr{E}\times\mathcal{S}$ une relation $\mathscr{R}$ [1] sur $\mathscr{E}\times\mathcal{S}$ totale à gauche.

Exemple :
Le problème

$$\textsc{Prime} : \begin{cases} \textbf{Entrée} & : n \in \mathbb{N} \\ \textbf{Sortie} & : n \text{ est-il premier ?} \end{cases}$$

est donc défini comme

$$\textsc{Prime} = \{(0,\boldsymbol{F}),(1,\boldsymbol{F}),(2,\boldsymbol{V}),(3,\boldsymbol{V}),(4,\boldsymbol{F}),(5,\boldsymbol{V}),\dots\} \subseteq \mathbb{N}\times\mathbb{B}.$$

Exemple :
Le problème

$$\textsc{Find}_0 : \begin{cases} \textbf{Entrée} & : \text{un tableau } T \text{ contenant au moins un } 0 \\ \textbf{Sortie} & : i \in \mathbb{N} \text{ tel que } T[i] = 0 \end{cases}$$

est donc défini comme

$$\textsc{Find}_0 = \Big\{([|0,1,1|],0),([|0,1,0|],0),([|0,1,0|],2),\dots\Big\}.$$

Remarque :
De la même manière qu'en mathématiques, on n'écrit pas $\{(0,1),(1,2),(2,3),\dots\}$ mais

1. C'est l'ensemble des liens entrées/sorties

$x \mapsto x+1$, en informatique, on préfère la notation sous forme de problème.

Remarque :
On précisera, en cas d'ambigüité la représentation choisie pour les entrées.

Exemple :
Les deux problèmes

$$\begin{cases} \textbf{Entrée} & : n \text{ sous forme décomposée en facteurs premiers} \\ \textbf{Sortie} & : n \text{ est-il premier ?} \end{cases}$$

et

$$\begin{cases} \textbf{Entrée} & : n \text{ en base 2} \\ \textbf{Sortie} & : n \text{ est-il premier ?} \end{cases}$$

sont différents.

Remarque :
Lorsque ce n'est pas précisé, dans la suite du cours, les entiers sont représentés en base 2.

Exemple :
Dans le problème

$$\begin{cases} \textbf{Entrée} & : L_1 \text{ et } L_2 \text{ deux langages réguliers} \\ \textbf{Sortie} & : L_1 = L_2, \end{cases}$$

on doit préciser la représentation de L_1 et L_2.

Définition : On appelle *problème de décision* un problème à valeurs dans $\mathbb{B}$ déterministe.

Exemple :
Par exemple, le problème Prime est un problème de décision.

Remarque (Notation) :
Lorsque Q est un problème, on note $\mathscr{E}_Q$ son espace d'entrée, et $\mathscr{S}_Q$ son espace de sortie. De plus, si Q est un problème de décision, on note $Q^+ = \{e \in \mathscr{E}_Q \mid (e, \boldsymbol{V}) \in Q\}$ et $Q^- = \{e \in \mathscr{E}_Q \mid (e, \boldsymbol{F}) \in Q\}$. $\{Q^+, Q^-\}$ est une partition de $\mathscr{E}_Q$.

4.3 Décidabilité

La définition d'un algorithme comme une suite finie d'instruction élémentaire, nous montre que l'ensemble d'algorithmes est dénombrable.[2] Mais, l'ensemble de problèmes est indénom-

2. en bijection avec $\mathbb{N}$

brable. En effet, pour $x \in [0,1[$, on définit le problème

$$\textsc{Bit}_x : \begin{cases} \textbf{Entrée} & : n \in \mathbb{N} \\ \textbf{Sortie} & : \text{le } n\text{-ième bit de } x. \end{cases}$$

Et, comme $[0,1[$ n'est pas dénombrable, l'ensemble des problèmes ne l'est pas non plus.

4.3.1 Modèles de calcul

Définition : On appellera *modèle de calcul* la donnée d'un ensemble de machines
- qu'il est possible d'exécuter sur des entrées ;
- qui peuvent ou pas retourner une réponse.

Exemple :
Les automates déterministes qu'il est possible d'exécution sur une entrée $w \in \Sigma^\star$, obtenant ainsi un booléen $b \in \mathbb{B}$, est un modèle de calcul.

Dans ce chapitre, notre modèle de calcul sera l'ensemble des fonction OCaml ayant pour type `string` $\to$ `string` qu'il est possible d'exécuter, qui peuvent donner un réponse ou non (boucle infinie, erreur).

Remarque :
On se place dans un monde d'exécution idéal : *mémoire infinie*.

Exemple :
On reprend le problème Prime. On code, sous forme de fonction OCaml, la machine ci-dessous. Elle répond au problème Prime.

```
let est_premier (s: string) : string =
  let n = int_of_string s in
  if n <= 1 then "false"
  else
    let i = ref 2 in
    let i_sq = ref 4 in
    let compose = ref false in
    while not !compose && !i_sq <= n do
      if n mod !i = 0 then compose := true
      else i_sq := !i_sq + 2 * !i + 1;
        i := !i + 1;
    done;
    string_of_bool(not !compose)
```

Code 4.1 – Machine décidant le problème Prime

Remarque (Notation) :
Dans la suite, lorsque $\mathscr{M}$ est une machine, et $w \in$ `string`, on notera
- $w \xrightarrow[\mathscr{M}]{} w'$ si l'exécution de $\mathscr{M}$ sur w conduit à $w' \in$ `string`.
- $w \xrightarrow[\mathscr{M}]{} \circlearrowleft$ si l'exécution de $\mathscr{M}$ sur w conduit à une erreur.

Dans la suite de ce chapitre, on fixe $\Sigma =$ `char` et donc $\Sigma^\star =$ `string`.

Remarque :
On pourra, dans la suite, généraliser la signature de nos machines à un type $\mathscr{E} \to \mathscr{F}$ dès lors qu'on exhibe une fonction de sérialisation $\varphi : \mathscr{E} \to \Sigma^\star$ inversible (sur son espace image) injective : avec $\varphi_{\mathscr{E}} : \mathscr{E} \to \Sigma^\star$ et $\varphi_{\mathscr{F}} : \mathscr{F} \to \Sigma^\star$, on a

```
let ma_super_fonction (e: E) : F = ...

let ma_fonction (s: string) : string =
  φ_F(ma_super_fonction(φ_E^-1(s)))
```

Code 4.2 – Généralisation des machines ayant pour entrée un ensemble $\mathscr{E}$ et sortie $\mathscr{F}$

4.3.2 Décidabilité

Définition : Une fonction partielle $f : \mathscr{E} \to \mathcal{S}$ est dite *calculée* par une machine $\mathscr{M}$ dès lors que

$$\forall e \in \mathrm{def}(f), \quad e \underset{\mathscr{M}}{\longrightarrow} f(e).$$

On dit alors d'une telle fonction qu'elle est *calculable*.

Remarque :
Cette définition ne spécifie aucunement le comportement de $\mathscr{M}$ sur une entrée $e \notin \mathrm{def}(f)$.

Exemple :
Considérons la fonction partielle $\sqrt{\ } : \mathbb{Z} \to \mathbb{Z}$ telle que $\mathrm{def}(\sqrt{\ }) = \{p^2 \mid p \in \mathbb{N}\}$, et $\sqrt{x}$ est l'unique $y \in \mathbb{N}$ tel que $y^2 = x$.

```
let sqrt (n: int): int =
  if m < 0 then -1
  else
    begin
      let i = ref 0 in
      let i_sq = ref 0 in
      while !i_sq <> n do
        i_sq := !i_sq + 2 * !i + 1;
        i := !i + 1;
      done;
      !i
    end
```

Code 4.3 – Machine calculant la fonction $\sqrt{\ }$

— Pour $\mathtt{n} < 0$, on a $\mathtt{n} \underset{\mathscr{M}}{\longrightarrow} -1$.
— Pour $\mathtt{n} \geqslant 0$ qui n'est pas un carré, $\mathtt{n} \underset{\mathscr{M}}{\longrightarrow} \circlearrowleft$.
— Pour $\mathtt{n} \geqslant 0$ qui est un carré, $\mathtt{n} \underset{\mathscr{M}}{\longrightarrow} \sqrt{\mathtt{n}}$.

Ainsi, $\sqrt{\ }$ est calculable.

Définition : Étant donné qu'un problème de décision Q est un cas particulier de fonc-

tion totale $\mathscr{E}_Q \to \mathbb{B}$, on dit que Q est *décidé* par une machine $\mathscr{M}$ dès lors que

$$\forall e \in \mathscr{E}_Q, \qquad \Big(e \in Q^+ \iff e \underset{\mathscr{M}}{\longrightarrow} \boldsymbol{V} \quad \text{et} \quad e \in Q^- \iff e \underset{\mathscr{M}}{\longrightarrow} \boldsymbol{F}\Big).$$

On dit alors que ce problème Q est *décidable*.

Exemple :
On considère le problème

$$\textsc{Existe}_0 : \begin{cases} \textbf{Entrée} & : \text{un tableau } T \\ \textbf{Sortie} & : \exists i \in \mathbb{N},\ T[i] = 0 ?. \end{cases}$$

La machine ci-dessous décide le problème $\textsc{Existe}_0$.

```
exception OK

let existe (t: int array) : bool =
  try
    for i = 0 to (Array.length t) - 1 do
      if t.(i) = 0 then
        raise OK
    done;
    false
  with
  | OK -> true
```

Code 4.4 – Machine décide le problème $\textsc{Existe}_0$

4.3.3 Langages et problèmes de décision

Les notions de langages et problèmes de décision d'entrée $\Sigma^\star$ coïncident. En effet, à un langage $L \subseteq \Sigma^\star$, on associe le problème de décision

$$\textsc{Appartient}_L : \begin{cases} \textbf{Entrée} & : w \in \Sigma^\star \\ \textbf{Sortie} & : w \in L\ ?. \end{cases}$$

On a alors $(\textsc{Appartient}_L)^+ = L$. Réciproquement, à un problème de décision Q d'entrées $\Sigma^\star$, on associe le langage Q^+.

Définition : Un langage L est dit *décidable* lorsque le problème $\textsc{Appartient}_L$ est décidable.

Définition : Étant donné une machine $\mathscr{M}$ de type `string` $\to$ `bool`, on appelle *langage* de $\mathscr{M}$, que l'on note $\mathscr{L}(\mathscr{M})$, l'ensemble

$$\{w \in \Sigma^\star \mid w \underset{\mathscr{M}}{\longrightarrow} \boldsymbol{V}\}.$$

Remarque :
$\mathscr{L}(\mathscr{M})$ n'est pas le complémentaire de $\{w \in \Sigma^\star \mid w \underset{\mathscr{M}}{\longrightarrow} \boldsymbol{F}\}$. En effet, il peut exister $w \in \Sigma^\star$ tel que $w \underset{\mathscr{M}}{\longrightarrow} \circlearrowleft$.

Propriété : Un langage L est décidable si, et seulement si L est le langage d'une machine $\mathcal{M}$ telle que $\forall w \in \Sigma^\star$, $w \xrightarrow[\mathcal{M}]{} \boldsymbol{V}$ ou $w \xrightarrow[\mathcal{M}]{} \boldsymbol{F}$.

Propriété : Tout langage régulier est décidable.

Preuve :
Soit L un langage régulier. Montrons que L est décidable. On utilise alors la fonction OCaml suivante de reconnaissance d'un mot dans un automate (que l'on a déjà codé en TP). □

Propriété (stabilité des langages décidables) **:** Un langage décidable est stable par

1. union ;
2. intersection ;
3. complémentaire.

Preuve : 1. Soient L_1 et L_2 deux langages décidables. Montrons que $L_1 \cup L_2$ est décidable. Soit $\texttt{decide}_1 : \texttt{string} \to \texttt{bool}$ la fonction décidant le langage L_1.[3] Soit $\texttt{decide}_2 : \texttt{string} \to \texttt{bool}$ la fonction décidant du langage L_2. On construit alors la fonction

```
let decide (w : string) : bool =
  (decide₁ w) || (decide₂ w)
```

Code 4.5 – Fonction OCamlreconnaissant l'union de deux langages décidables

□

Remarque :
$\varnothing$ est décidable (par `fun s -> false`); $\Sigma^\star$ est décidable (par `fun s -> true`).

4.3.4 Sérialisation

Définition : Étant donné un type OCaml `t`, on appelle *sérialisation calculable* de ce type `t`, la donnée d'une fonction `f` OCaml de type $\texttt{t} \to \texttt{string}$ qui soit telle que

- pour tout e : `t`, (`f` e) est bien parenthésée ;
- `f` est injective ;
- la réciproque de `f` (définie sur $\mathrm{Im}(\texttt{f})$) est définissable en OCaml.

Exemple :
Le type `int` est sérialisable par la fonction `string_of_int`.

Propriété : Soit $\texttt{t}_\texttt{a}$ et $\texttt{t}_\texttt{b}$ deux types OCaml sérialisables, alors le type $\texttt{t}_\texttt{a} \star \texttt{t}_\texttt{b}$ est sérialisable.

3. *i.e.* $\forall w \in \Sigma^\star, w \in L_1 \iff \texttt{decide}_1(w) = \texttt{true}$

Preuve :
Soit φ_a et φ_b les fonctions de sérialisation des types t_a et t_b. On définit alors la fonction

```
let φ ((a,b): t_a * t_b) =
  "(" ^ (φ_a a) ^ "),(" ^ (φ_b b) ^ ")"
```

Code 4.6 – Fonction OCaml sérialisant le produit cartésien de deux types sérialisables

On remarque que
- φ est à valeur dans les chaînes de caractères bien parenthésées ;
- φ est injective (par identification de parenthèses et injectivité de φ_a et φ_b) ;
- la réciproque de φ est décidable en OCaml (preuve à faire en OCaml).

□

Remarque :
Une programme OCaml est trivialement sérialisable : c'est déjà une chaîne de caractères.

Exemple :
La sérialisation de la fonction

```
let rec fact (n : int) : int =
  if n = 0 then 1
  else n * (fact (n-1))
```

Code 4.7 – Fonction factorielle en OCaml

est la chaîne de caractère

```
"let␣rec␣fact␣(n␣:␣int)␣:␣int␣=
␣␣if␣n␣=␣0␣then␣1
␣␣else␣n␣*␣(fact␣(n-1))".
```

4.3.5 Machine universelle

Soit l'ensemble $\mathcal{O}$ des chaînes de caractères qui sont des sérialisations de programme OCaml valide.

Exemple :
La sérialisation de la fonction `fact` trouvée précédemment est un élément de $\mathcal{O}$. Mais, "`let`" $\notin \mathcal{O}$.

Définition : Soit la fonction `interprète` : $\mathcal{O} \times \Sigma^\star \to \Sigma^\star$ définie par

$$\texttt{interprète}(\mathcal{M}, w) = \begin{cases} w' \text{ tel que } w \underset{\mathcal{M}}{\longrightarrow} w' \\ \text{non défini sinon.} \end{cases}$$

Théorème : La fonction `interprète` est calculable. On appelle *machine universelle* un programme OCaml la calculant.

Preuve :
On considère `utop` ou `WinCaml`. □

De même, considérons le problème suivant

$$\begin{cases}\textbf{Entrée} & : M \in \mathcal{C}, w \in \Sigma^\star, n \in \mathbb{N} \\ \textbf{Sortie} & : M \text{ se termine-t-elle sur } w \text{ en moins de } n \text{ étapes élémentaires ?}\end{cases}$$

Ce problème est décidable.

4.3.6 Théorème de l'Arrêt

Dans cette sous-section, on considère le problème

$$\textsc{Arrêt} : \begin{cases}\textbf{Entrée} & : M \in \mathcal{C}, w \in \Sigma^\star \\ \textbf{Sortie} & : M \text{ s'arrête-t-elle sur } w.\end{cases}^{4}$$

Théorème : Le problème Arrêt est indécidable.

Preuve :
Par l'absurde : supposons Arrêt décidable. Soit `arret : string → bool` prenant en argument une chaîne de caractères qui est la sérialisation d'un couple M code d'une machine ($M \subseteq \Sigma^\star$), et $w \in \Sigma^\star$, et retournant `true` si et seulement si M s'arrête sur l'entrée w. Si l'entrée n'est pas une sérialisation convenable, on retourne `false`. On crée le programme suivant

```
let paradoxe (w : string) : bool =
  if arret (serialise_couple w w) then
    (while true do () done; true)
  else false
```

Code 4.8 – Programme paradoxe prouvant que le problème Arrêt est indécidable

où `serialise_couple` est une fonction sérialisant un couple avec l'algorithme trouvé précédemment. Soit $S_{\texttt{paradoxe}}$ la sérialisation de la fonction `paradoxe`.

Quid de (`paradoxe` $S_{\texttt{paradoxe}}$) : soit $c =$ (`serialise_couple` $S_{\texttt{paradoxe}}$ $S_{\texttt{paradoxe}}$). La chaîne c est donc la sérialisation d'un couple dont la première composante est le code de la fonction `paradoxe`. De plus, `arret` se termine sur toute entrée.

- Si (`arret` c) $=$ `false`, alors on va dans la branche `else` et l'exécution de `paradoxe` sur $S_{\texttt{paradoxe}}$ termine. Mais, comme (`arret` c) $=$ `false`, `paradoxe` ne se termine pas sur $S_{\texttt{paradoxe}}$.
- Si (`arret` c) $=$ `true`, alors on va dans la branche `then`, et donc (`paradoxe` $S_{\texttt{paradoxe}}$) ne se termine pas. Mais, comme (`arret` c) $=$ `true`, alors (`paradoxe` $S_{\texttt{paradoxe}}$) se termine.

On en conclut que le problème Arrêt est indécidable. □

Corollaire : Il existe des problèmes indécidables.

4. *i.e.* $\exists ? w' \in \Sigma^\star,\ w \underset{M}{\longrightarrow} w'$

4.3.7 Réduction

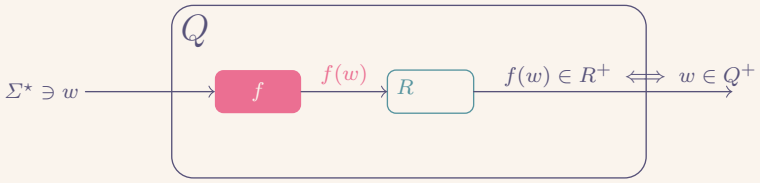


FIGURE 4.1 – Structure d'un sous-problème

Définition : Soit Q et R deux problèmes de décision. On dit que Q *se réduit au problème* R s'il existe $f : \mathscr{C}_Q \to \mathscr{C}_R$ totale et calculable, telle que

$$w \in Q^+ \iff f(w) \in R^+.$$

On note alors $Q \preccurlyeq R$.

Propriété : Si $Q \preccurlyeq R$, et que R est décidable, alors Q est décidable.

Preuve :
Soit $\texttt{decide}_R : \texttt{string} \to \texttt{bool}$ décidant R *i.e.* $(\texttt{decide}_R\ w) = \texttt{true} \iff w \in R^+$. Soit $f : \mathscr{C}_Q \to \mathscr{C}_R$ totale calculable, telle que $w \in Q^+ \iff f(w) \in R^+$. On doit coder la fonction suivante.

```
let decide_Q (w: string) : bool =
  (decide_R (f w))
```

CODE 4.9 – Fonction décidant un sous-problème

La fonction $\texttt{decide}_Q$ décide bien le problème Q. En effet,

$$\begin{aligned}(\texttt{decide}_Q\ \texttt{w}) = \texttt{true} &\iff (\texttt{decide}_R\ (f\ \texttt{w})) \\ &\iff (f\ \texttt{w}) \in R^+ \\ &\iff w \in Q^+.\end{aligned}$$

□

Corollaire : Si $Q \preccurlyeq R$, et Q non décidable, alors R non décidable.

EXEMPLE :
On considère le problème

$$\text{NonVide} : \begin{cases}\textbf{Entrée} & : M \in \mathcal{O} \\ \textbf{Sortie} & : \mathscr{L}(M) \neq \varnothing\ ?\ .\end{cases}$$

Le problème NonVide est indécidable.

EXEMPLE :
Les problèmes

$$\begin{cases}\textbf{Entrée} & : M_1 \text{ et } M_2 \text{ deux machines} \\ \textbf{Sortie} & : \mathscr{L}(M_1) \cup \mathscr{L}(M_2) = \Sigma^\star\end{cases}$$

et

$$\begin{cases} \textbf{Entrée} & : M_1 \text{ et } M_2 \text{ deux machines} \\ \textbf{Sortie} & : \mathscr{L}(M_1) \cap \mathscr{L}(M_2) = \varnothing \end{cases}$$

sont indécidables.

Propriété : La relation $\preccurlyeq$ est un *pré-ordre* :
- $\preccurlyeq$ est réflective ;
- $\preccurlyeq$ est transitive.

Preuve :
Soit Q un problème de décision.
- $Q \preccurlyeq Q$ par la fonction identité, qui est totale et calculable.
- Soient Q, R et S trois problèmes de décision tels que $Q \preccurlyeq R$ et $R \preccurlyeq S$. Soit donc f_1 la réduction de Q à R, et f_2 la réduction de R à S. Soit $f = f_2 \circ f_1 : \mathscr{E}_Q \to \mathscr{E}_S$. La fonction f est totale comme composée de fonctions totales, f est calculable comme composée de fonctions calculables. De plus,

$$\begin{aligned} \forall e \in \mathscr{E}_Q, \qquad f(e) \in S^+ &\iff f_2(f_1(e)) \in S^+ \\ &\iff f_1(e) \in R^+ \\ &\iff e \in Q^+ \end{aligned}$$

□

4.4 Classe P et NP

Pour répondre à un problème, on peut le résoudre par des algorithmes plus ou moins rapides. Mais, l'objectif de cette section est de montrer que certains problèmes ne peuvent se résoudre que par des algorithmes lents, et que l'on ne peut pas faire mieux.

Définition : Le modèle de calcul impose une représentation des entrées par chaînes de caractères. Cela induit donc une notion de *taille d'entrée*, qui est la longueur de la chaîne de caractères.

4.4.1 Complexité d'une machine

Définition : Étant donné une machine $\mathscr{M}$ et une entrée $w \in \Sigma^\star$, on note $C^{\mathscr{M}}(w)$ le nombre d'opérations élémentaires effectuées lors de l'appel de $\mathscr{M}$ sur w. Lorsque $w \xrightarrow[\mathscr{M}]{} \circlearrowleft$, on définit $C^{\mathscr{M}} = +\infty$.

Pour $n \in \mathbb{N}$, on définit alors

$$C_n^{\mathscr{M}} = \max\{C^{\mathscr{M}}(w) \mid w \in \Sigma^n\}.$$

Remarque :
On a, $\forall n \in \mathbb{N}$, $C_n^{\mathscr{M}} \in \bar{\mathbb{N}} = \mathbb{N} \cup \{+\infty\}$.

Définition : Soit $f : \mathbb{N} \to \mathbb{N}$ une fonction totale et calculable. On note $\textsc{Time}(f)$ l'ensemble des machines $\mathscr{M}$ telles que
— $\mathscr{M}$ s'arrête sur toute entrée ;
— $(C_n^{\mathscr{M}})_{n\in\mathbb{N}} = \mathcal{O}((f(n))_{n\in\mathbb{N}})$.

4.4.2 Classe P

Définition : On dit d'une machine $\mathscr{M}$ qu'elle est de *complexité polynômiale* dès lors qu'il existe $k \in \mathbb{N}$ tel que $\mathscr{M} \in \textsc{Time}(n^k)$.

Définition : On dit d'une fonction (partielle ou non), qu'elle est *calculable en temps polynômial* dès lors qu'il existe une machine $\mathscr{M}$ de complexité polynômiale la calculant.

Exemple : — l'identité ($n \mapsto n$)
— la fonction successeur ($n \mapsto n+1$)

Propriété : La composée de deux fonctions totales calculables en temps polynômial est une fonction totale calculable en temps polynômial.

Preuve :
Soient f_1 et f_2 deux telles fonctions. Soit donc `calcule`$_1$: `string` $\to$ `string` et `calcule`$_2$: `string` $\to$ `string` calculant respectivement f_1 et f_2. On pose la fonction ci-dessous

```
let calcul s = calcule1 (calcule2 s)
```

Code 4.10 – Machine calculant la composée en temps polynômial

dont le nombre d'opérations élémentaires est

$$C(\mathtt{s}) = \underbrace{C^2(\mathtt{s})}_{\text{calcul de } f_2(\mathtt{s})} + \overbrace{C^1(f_2(\mathtt{s}))}^{\text{calcul de } f_1(\mathtt{s})} + \underbrace{1}_{\text{composée}}.$$

Or, la fabrication d'une chaîne de caractères de taille n nécessite au moins n opérations élémentaires. Soit $p_1 \in \mathbb{N}$ et $p_2 \in \mathbb{N}$ tels que la complexité de `calcule`$_1$ est $\mathcal{O}(n^{p_1})$ et la complexité de `calcule`$_2$ est $\mathcal{O}(n^{p_2})$. On a donc, pour tout $w \in \Sigma^\star$ avec $|w| = n$, que $|f_2(w)| = \mathcal{O}(n^{p_2})$ Par composition des $\mathcal{O}$, on a que

$$C(\mathtt{s}) = \mathcal{O}(|\mathtt{s}|^{p_1 \cdot p_2}).$$

□

Remarque :
Dans la preuve précédente, l'ordre du polynôme change. En effet, la composée de deux programmes en $\mathcal{O}(n^2)$ est un $\mathcal{O}(n^4)$. L'espace des fonctions calculables en $\mathcal{O}(n^p)$, pour un p fixé, n'est pas stable par composition.

Définition (Classe P) : On dit qu'un problème est dans **P** dès lors qu'il est décidable en temps polynômial.

Propriété (Stabilité de la classe **P**) **:** La classe **P** est stable par
— union ; — intersection ; — complémentaire.

Preuve :
à faire □

4.4.3 Classe NP

Définition (Classe **NP**) **:** On dit qu'un problème de décision Q est dans **NP** si et seulement si
— il existe un polynôme A ;
— un problème $\textsc{Verif} \in \mathbf{P}$;
— un ensemble $\mathscr{C}$ (certificats),
tels que

$$\forall w \in \Sigma^\star, \quad \Big(w \in Q^+ \iff \exists u \in \mathscr{C},\ |u| \leqslant A(|w|) \text{ et } (w,u) \in \textsc{Verif}^+\Big).$$

La classe **NP** est la classe de problèmes tels qu'ils sont vérifiables en temps polynômial. Par exemple, si on a une formule logique, trouver un environnement propositionnel est coûteux en temps, mais vérifier la solution est très simple. Nous verrons ce résultat plus tard dans cette sous-section.

Le "**NP**" ne vient pas de **N**on **P**olynômial, mais vient de **N**on-déterministe **P**olynômial.

Propriété :

$$\mathbf{P} \subseteq \mathbf{NP}.$$

Preuve :
À faire : Modifier la définition de Verif avec la nouvelle définition d'un problème **NP** Soit Q un problème de décision dans **P**. On pose $\mathscr{C} = \mathscr{C}_Q$, $A(X) = X$, et $\textsc{Verif} = Q$. En effet, pour tout $w \in \Sigma^\star$,

$$w \in Q^+ \iff \exists u = w,\ |u| \leqslant A(|w|) \text{ et } u \in \textsc{Verif}^+.$$

□

Propriété :

$$\textsc{Sat} \in \mathbf{NP}.$$

Rappel :
On rappelle la définition du problème Sat.

$$\textsc{Sat} : \begin{cases} \textbf{Entrée} & : \text{Une formule } G \\ \textbf{Sortie} & : \text{Existe-t-il } \rho \in \mathbb{B}^{\text{vars}(G)} \text{ tel que } [\![G]\!]^\rho = V \,? \end{cases}$$

Preuve :

Soit alors le problème suivant.

$$\textsc{VerifSat} : \begin{cases} \textbf{Entrée} & : \begin{pmatrix} \text{Une formule } G, \\ \text{un environnement propositionnel } \rho \in \mathbb{B}^{\text{vars}(G)}, \end{pmatrix} \\ \textbf{Sortie} & : [\![G]\!]^\rho =^? \boldsymbol{V} \end{cases}$$

En TP, on a codé une solution polynômial à VerifSat donc VerifSat $\in$ **P**. On définit l'ensemble $\mathscr{C}$ des certificats comme

$$\mathscr{C} = \{(G, \rho) \mid G \in \mathscr{F} \text{ et } \rho \in \mathbb{B}^{\text{vars}(G)}\}.$$

On a alors

$$G \in \textsc{Sat}^+ \iff \exists \rho \in \mathbb{B}^{\text{vars}(G)},\ [\![G]\!]^\rho = \boldsymbol{V}.$$

Il suffit alors de choisir $A(X) = 2X$. □

4.4.4 NP-difficile

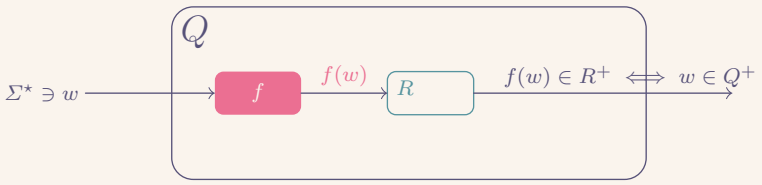


FIGURE 4.2 – Structure d'un sous-problème

Définition (Réduction polynômiale) **:** Soit Q et R deux problèmes de décision, on appelle *réduction polynômiale* de Q à R la donnée d'une fonction f totale, calculable en temps polynômial de $\mathscr{C}_Q$ dans $\mathscr{C}_R$ telle que

$$\forall w \in \mathscr{C}_Q, \quad w \in Q^+ \iff f(w) \in R^+.$$

On note alors $Q \preccurlyeq_{\text{p}} R$.

Propriété : La relation $\preccurlyeq_{\text{p}}$ est transitive et reflective : c'est un pré-ordre.

Preuve :
La réflectivité est assurée car id est totale et calculable en temps polynômial. La transitivité est assurée par les propriétés précédentes (composée de deux fonctions polynômiales ?). □

Propriété : Si $R \preccurlyeq_{\text{p}} Q$, et $R \in$ **P**, alors $Q \in$ **P**. □

Définition (NP-difficile) **:** Un problème Q est **NP**-difficile si

$$\forall R \in \textbf{NP},\ R \preccurlyeq_{\text{p}} Q.$$

Propriété : Si Q est **NP**-difficile, et $Q \preccurlyeq_{\text{p}} R$, alors R est **NP**-difficile.

Preuve :
Soit $S \in \mathbf{NP}$, donc $S \preccurlyeq_{\mathrm{p}} Q$. De plus, $Q \preccurlyeq_{\mathrm{p}} R$, donc $S \preccurlyeq_{\mathrm{p}} R$, par transitivité. Ceci étant vrai pour tout $S \in \mathbf{NP}$, on en déduit que R est **NP**-difficile. □

On admet le théorème suivant.

Théorème (Cook-Levin) : Le problème Sat est **NP**-difficile.

Preuve :
Admis □

Définition (n-fnc) : Soit $n \in \mathbb{N}$. Une formule G est sous forme n-fnc dès lors que G est sous forme fnc et chaque clause de G contient au plus n littéraux.

On parle aussi de forme cnf traduction anglaise de fnc. De même, on parle de forme n-cnf au lieu de n-fnc.

Exemple :
La formule $(p \vee q) \wedge r$ est une 2-cnf. La formule $(p \vee q \vee p) \wedge (r \vee p \vee q \vee q)$ est une 4-cnf.

Définition : On définit le problème ci-dessous.

$$n\text{-cnf-Sat} : \begin{cases} \textbf{Entrée} & : G \text{ une } n\text{-cnf} \\ \textbf{Sortie} & : \text{Existe-t-il } \rho \text{ tel que } [\![G]\!]^{\rho} = \boldsymbol{V} \,? \end{cases}$$

Propriété : Soit 3sat = 3-cnf-Sat. Le problème 3sat est **NP**-difficile.

Preuve (par réduction de Sat à 3sat) *:*
Soit G une formule sur $\mathbb{Q}$, un ensemble de variables propositionnelles. Pour toute sous-formule H, on note
- x_H une variable propositionnelle,
- K_H une formule définie par
  - si $H = \top$, $H = \bot$ ou $H = p \in \mathbb{Q}$, alors $K_H = H$,
  - si $H = \neg H_1$, alors $K_H = \neg x_{H_1}$,
  - si $H = H_1 \odot H_2$, avec $\odot \in \{\rightarrow, \vee, \wedge, \leftrightarrow\}$, alors $K_H = x_{H_1} \odot x_{H_2}$.

Définissons alors la formule

$$K = \bigwedge_{H \text{ sous-formule de } G} (x_H \leftrightarrow K_H).$$

On note aussi, si $\mathbb{Q} \subseteq \mathbb{Q}'$ deux ensembles de variables propositionnelles, et $\rho \in \mathbb{B}^{\mathbb{Q}}$, on note $\rho' \sqsupseteq \rho$ dès lors que $\mathrm{def}(\rho') = \mathbb{Q}'$ et $\forall x \in \mathbb{Q}, \rho(x) = \rho'(x)$. On pose $\mathbb{Q}' = \mathbb{Q} \cup \{x_H \mid H \text{ sous-formule de } G\}$. On considère à présent le lemme suivant.

Lemme : Soit $\rho \in \mathbb{B}^{\mathbb{Q}}$. Il existe $\rho' \in \mathbb{B}^{\mathbb{Q}'}$ tel que $\rho' \sqsupseteq \rho$ et $[\![K]\!]^{\rho} = \boldsymbol{V}$.

Prouvons ce lemme.

Preuve :
On définit

$$\rho'(x) = \begin{cases} \rho(x) & \text{si } x \in \mathbb{Q} \\ [\![H]\!]^{\rho} & \text{si } x = x_H. \end{cases}$$

Soit alors H une sous-formule de G.

— Si $H = \top$, alors $\rho'(x_H) = [\![H]\!]^{\rho} = [\![x_H]\!]^{\rho'}$, et $[\![K_H]\!]^{\rho'} = [\![H]\!]^{\rho'} = [\![\top]\!]^{\rho'} = [\![H]\!]^{\rho}$. Ainsi, $[\![x_H \leftrightarrow K_H]\!]^{\rho'} = \boldsymbol{V}$.

— Si $H = \neg H_1$, alors $[\![x_H]\!]^{\rho'} = [\![H]\!]^{\rho}$, et

$$[\![K_H]\!]^{\rho'} = [\![\neg x_{H_1}]\!]^{\rho'} = \overline{[\![x_{H_1}]\!]^{\rho'}} = \overline{[\![H_1]\!]^{\rho}} = [\![H]\!]^{\rho}.$$

On en déduit que $[\![x_H \leftrightarrow K_H]\!]^{\rho'} = \boldsymbol{V}$.

— Si $H = H_1 \wedge H_2$, alors $[\![x_H]\!]^{\rho'} = [\![H]\!]^{\rho}$, et

$$\begin{aligned} [\![K_H]\!]^{\rho'} &= [\![x_{H_1} \wedge x_{H_2}]\!]^{\rho'} \\ &= [\![x_{H_1}]\!]^{\rho'} \cdot [\![x_{H_2}]\!]^{\rho'} \\ &= [\![H_1]\!]^{\rho} \cdot [\![H_2]\!]^{\rho} \\ &= [\![H_1 \wedge H_2]\!]^{\rho} \\ &= [\![H]\!]^{\rho} \end{aligned}$$

— De même pour les autres cas...

On a donc

$$\left[\!\!\left[\bigwedge_{H \text{ sous-formule de } G} x_H \leftrightarrow K_H \right]\!\!\right]^{\rho'} = \bigcdot_{H \text{ sous-formule de } G} [\![x_H \leftrightarrow K_H]\!]^{\rho'} = \boldsymbol{V}$$

et donc $[\![K]\!]^{\rho'} = \boldsymbol{V}$. □

Lemme : Pour tout environnement propositionnel $\rho \in \mathbb{B}^{\mathbb{Q}}$, et pour tout $\rho' \in \mathbb{B}^{\mathbb{Q}'}$. Si $[\![K]\!]^{\rho'} = \boldsymbol{V}$, alors $\rho(x_G) = [\![G]\!]^{\rho}$.

Preuve :
Soient $\rho \in \mathbb{B}^{\mathbb{Q}}$ et $\rho' \in \mathbb{B}^{\mathbb{Q}'}$ deux environnements propositionnels. *Montrons, par induction sur les sous-formules de G, pour toute sous-formule H de G, que $\rho'(x_H) = [\![H]\!]^\rho$.*

— Si $H = \top$, alors, comme $[\![K]\!]^{\rho'} = \boldsymbol{V}$, on a $[\![x_H \leftrightarrow K_H]\!]^{\rho'} = \boldsymbol{V}$, donc $\rho'(x_H) = [\![H]\!]^{\rho'} = [\![H]\!]^\rho$.

— Si $H = \neg H_1$, avec $\rho'(x_{H_1}) = [\![H_1]\!]^\rho$ par hypothèse d'induction, alors $[\![K]\!]^{\rho'} = \boldsymbol{V}$ donc $[\![x_H \leftrightarrow K_H]\!]^{\rho'} = \boldsymbol{V}$, d'où

$$\begin{aligned}\rho'(x_H) &= [\![K_H]\!]^{\rho'}\\ &= [\![\neg x_{H_1}]\!]^{\rho'}\\ &= \overline{[\![x_{H_1}]\!]^{\rho'}}\\ &= \overline{[\![H_1]\!]^{\rho}}\\ &= [\![\neg H_1]\!]^{\rho}\\ &= [\![H]\!]^{\rho}\end{aligned}$$

— Si $H = H_1 \wedge H_2$, avec $\rho'(x_{H_1}) = [\![H_1]\!]^\rho$, et $\rho'(x_{H_2}) = [\![H_2]\!]^\rho$ par hypothèse d'induction, alors $[\![K]\!]^\rho = \boldsymbol{V}$ donc $[\![x_H \leftrightarrow K_H]\!]^{\rho'} = \boldsymbol{V}$ d'où

$$\begin{aligned}\rho'(x_H) &= [\![K_H]\!]^{\rho'}\\ &= [\![x_{H_1} \wedge x_{H_2}]\!]^{\rho'}\\ &= [\![x_{H_1}]\!]^{\rho'} \cdot [\![x_{H_2}]\!]^{\rho'}\\ &= [\![H_1]\!]^{\rho} \cdot [\![H_2]\!]^{\rho}\\ &= [\![H_1 \wedge H_2]\!]^{\rho}\\ &= [\![H]\!]^{\rho}\end{aligned}$$

— De même pour les autres cas…

□

À l'instance G du problème Sat, on associe donc la formule $K \wedge x_G$.

" $\Longrightarrow$ " Soit $G \in$ Sat$^+$, soit alors ρ tel que $[\![G]\!]^\rho = \boldsymbol{V}$, alors il existe, d'après le premier lemme, un environnement ρ' tel que $[\![K]\!]^{\rho'} = \boldsymbol{V}$. Le second lemme nous donne alors $[\![x_G]\!]^{\rho'} = [\![G]\!]^\rho = \boldsymbol{V}$ donc $[\![K \wedge x_G]\!]^{\rho'}$ d'où $[\![K \wedge x_G]\!] \in$ Sat$^+$.

" $\Longleftarrow$ " Si $K \wedge x_G \in$ Sat$^+$, il existe donc ρ' tel que $[\![K \wedge x_G]\!]^{\rho'} = \boldsymbol{V}$ donc $[\![K]\!]^{\rho'} = \boldsymbol{V}$ et $\rho'(x_G) = \boldsymbol{V}$. Soit alors ρ la restriction de ρ' à $\mathbb{Q}$. Ainsi, d'après le second lemme, $[\![G]\!]^\rho = \rho'(x_G) = \boldsymbol{V}$.

□

Remarque :
Pour tout $n \geqslant 3$, le problème n-cnf-sat est **NP**-difficile. Ceci peut être prouvé par réduction avec la fonction identité à 3Sat.

Définition : On dit qu'un problème est **NP**-complet s'il est dans la classe **NP** et dans la classe **NP**-difficile :

$$\textbf{NP}\text{-complet} = \textbf{NP}\text{-difficile} \cap \textbf{NP}.$$

Remarque :
Le problème Sat est **NP**-complet.

CHAPITRE

5

TROIS EXEMPLES D'ALGORITHMES DE GRAPHES

Sommaire

5.1 Composantes fortement connexes (CFC) **123**
5.1.1 Rappels 125
5.1.2 Rangement particulier de graphe 127
5.1.3 Graphe transposé et CFC 128
5.1.4 Calcul de tri préfixe 130
5.1.5 Algorithme de Kosaraju 131
5.1.6 Applications 131
5.2 Arbres couvrants de poids minimum **132**
5.3 Couplage dans un graphe biparti **137**
Annexe 5.A Remarques supplémentaires **140**

Dans ce chapitre, on s'intéresse aux graphes. Nous rappellerons les notions et algorithmes de graphes vus l'année dernière. On considère 3 exemples d'algorithmes de graphes.

Par exemple, l'année dernière, nous avons vu comment décomposer un graphe non-orienté en composantes connexes : on choisit un sommet au hasard, puis on parcours les voisins de ce sommets, et on répète. Mais, dans un graphe orienté, la notion de « composante connexe » n'est plus la même dans un graphe orienté. C'est l'algorithme décrit dans la section 1.

5.1 Composantes fortement connexes (CFC)

Dans la suite de cette section, $G = (S, A)$ est un graphe orienté.

Définition : On dit que $v \in S$ est *accessible* depuis $u \in S$ s'il existe un chemin de u à v, que l'on note $u \xrightarrow{\star} v$. De même, on dit que $v \in S$ est *co-accessible* depuis $u \in S$ s'il existe un chemin de v à u, que l'on note $v \xrightarrow{\star} u$.

Définition ($u \sim_G v$) : On note $u \sim_G v$ si $u \xrightarrow{\star} v$ et $v \xrightarrow{\star} u$.

Remarque :
La relation $\sim_G$ est une relation d'équivalence.

Digression L'année dernière, dans un graphe non orienté, on peut noter $\sim$ la relation d'équivalence induite par, si $\{u, v\} \in A$, alors $u \sim v$. Dans ce cas, le même chemin permet d'aller de u à v, puis de v à u, et ce chemin est le même. Mais, dans un graphe orienté, si $u \sim_G v$, les chemins de u à v puis de v à u ne sont pas forcément les mêmes.

Définition (cfc) : On appelle *composantes fortement connexes* (cfc) d'un graphe G, les classes d'équivalences de la relation $\sim_G$.

Définition : On dit d'un graphe ayant une unique composante fortement connexe qu'il est *fortement connexe*.

Exemple :
Le graphe ci-dessous a deux composantes fortement connexes, il n'est donc pas fortement connexe.

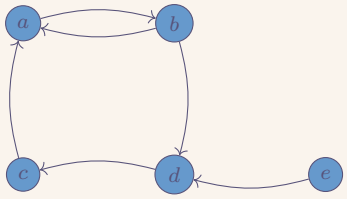


Figure 5.1 – Graphe non fortement connexe

Définition : On appelle *ensemble fortement connexe* un ensemble $V \subseteq S$ tel que G_V est fortement connexe où G_V est le *graphe induit* par V : $G_V = (V, A \cap V^2)$.

Exemple :
Avec le graphe précédent, l'ensemble $\{a, b\}$ est un ensemble fortement connexe.

Lemme : Si W est une composante fortement connexe de G, alors W est un ensemble fortement connexe.

Preuve :
Soit W une composante fortement connexe. On doit montrer que W est un ensemble fortement connexe, *i.e.* le graphe G_W est fortement connexe, *i.e.* pour tout couple de sommets $(u, v) \in W^2$, $u \sim_{G_W} v$. Étant

donné que W est une composante connexe, $u \sim_G v$, donc il existe $(n, m) \in \mathbb{N}^2$ et deux chemins

$$u \to v_1 \to v_2 \to \cdots \to v_n = v \quad \text{et} \quad v \to u_1 \to u_2 \to \cdots \to u_m = u.$$

Soit alors $i \in [\![1, n]\!]$. On a donc deux chemins

$$v \to u_1 \to u_2 \to \cdots \to u_m = u \to v_1 \to \cdots \to v_i$$

et

$$v_i \to v_{i+1} \to \cdots \to v_n = v.$$

Ainsi, $v \sim_G v_i$ et $v_i \in W$. De même, pour tout $j \in [\![1, m]\!]$, $u_j \in W$. Ainsi, $u \sim_{G_w} v$ (en utilisant les mêmes chemins). Donc, G_W est fortement connexe puisque toute paire de sommets est équivalent) □

Propriété : Les composantes fortement connexes sont les ensembles fortement connexes qui sont maximaux pour l'inclusion.

Preuve : " $\Longleftarrow$ " Soit V un ensemble fortement connexe maximal pour l'inclusion.
Remarque : on a $V \neq \varnothing$. En effet, un singleton est un ensemble fortement connexe, et donc V ne sera pas maximal pour l'inclusion.
Soit $(u, v) \in V^2$. Par définition de fortement connexe, $u \sim_{G_V} v$ donc $u \sim_G v$. Soit alors W la classe des éléments de $V \subseteq W$. L'ensemble W est fortement connexe (d'après le lemme précédent). Ainsi, $W = V$, W est donc une composante fortement connexe.

" $\Longrightarrow$ " Soit V une composante fortement connexe. L'ensemble V est fortement connexe, d'après le lemme précédent. Soit $W \supseteq V$ tel que W est fortement connexe. Montrons que $V = W$.

- Si $W \setminus V = \varnothing$, alors OK.
- Si $W \setminus V \neq \varnothing$, alors soit $z \in W \setminus V$. L'ensemble W est fortement connexe. Soit $u \in V$. On a $u \sim_{G_W} z$, donc $u \sim_G z$, donc $z \in V$ ce qui est absurde.

□

Définition : On appelle *graphe réduit* de G le graphe orienté $\hat{G} = (\hat{S}, \hat{A})$ où

$$\hat{S} = \{\bar{x} \mid x \in S\} \quad \text{et} \quad \hat{A} = \big\{(\bar{x}, \bar{y}) \mid (x, y) \in A \text{ et } \bar{x} \neq \bar{y}\big\}.$$

Exemple :
À faire : Figure

Remarque :
- $\hat{G}$ est acyclique.
- Pour tout couple $(\bar{x}, \bar{y}) \in \hat{S}^2$, si $\bar{x} \xrightarrow{\star}_{\hat{G}} \bar{y}$, alors $\forall u \in \bar{x}, \forall v \in \bar{y}, x \xrightarrow{\star}_G v$.

5.1.1 Rappels

Définition : La *bordure* d'un ensemble de sommets $V \subseteq S$, noté $\mathcal{B}(V)$, est l'ensemble des successeurs de V non dans V :

$$\mathcal{B}(V) = \{s \in S \setminus V \mid \exists u \in V,\ (u, s) \in A\}.$$

Définition (parcours) **:** Un *parcours* est une permutation des sommets $(L_1, L_2, \dots, L_n)$ telle que, pour $i \in [\![1, n-1]\!]$,

$$L_i \in \mathscr{B}(\{L_1, \dots, L_{i-1}\}) \quad \text{ou} \quad \mathscr{B}(\{L_1, \dots, L_{i-1}\}) = \varnothing.$$

On dit d'un L_i avec $i \in [\![1, n]\!]$ tel que $\mathscr{B}(\{L_1, \dots, L_{i-1}\}) = \varnothing$, que c'est un *point de régénération* du parcours.

Lemme : Si $V \subseteq S$ est tel que $\mathscr{B}(V) = \varnothing$, il n'existe aucun chemin d'un sommet de V à un chemin de $S \setminus V$.

Preuve (par l'absurde) *:*
Soit un chemin $V \ni u_0 \to u_1 \to \cdots \to u_k \in S \setminus V$ avec $k \in \mathbb{N}$. Soit $I = \{i \in [\![0, k]\!] \mid u_i \in S \setminus V\}$. L'ensemble I est non vide, car $u_k \in S \setminus V$, et $I \subseteq N$. Il admet donc un plus petit élément ; nommons le $i_0 \in [\![0, k]\!]$. On a $i_0 \neq 0$ car $u_0 \in V$. Ainsi, $u_{i_0-1} \in V$, $u_{i_0} \in S \setminus V$, et $(u_{i_0-1}, u_{i_0}) \in A$. Donc, $u_{i_0} \in \mathscr{B}(V)$, ce qui est absurde. □

Définition : Soit $(L_1, L_2, \dots, L_n)$ un parcours de G. On note K le nombre de ses points de régénération. On note $(r_k)_{k \in [\![1,k]\!]}$ l'extractrice des points de régénération. En notant de plus $r_{K+1} = n + 1$. Le *partitionnement associé au parcours* est alors

$$\Big\{\{L_{r_i}, L_{r_i+1}, \dots, L_{r_{i+1}-1}\} \,\Big|\, i \in [\![1, K]\!]\Big\}..$$

Exemple :
Si $n = 10$ et les points de régénération sont d'indices 1, 4, 7 et 8, alors le partitionnement associé est donc

$$\Big\{\{1, 2, 3\}, \{4, 5, 6\}, \{7\}, \{8, 9, 10\}\Big\}.$$

Définition : Un partitionnement P_1 est un *raffinement* d'un partitionnement P_2 dès lors que

$$\forall C_1 \in P_1,\ \exists C_2 \in P_2,\ C_1 \subseteq C_2.$$

Propriété : Les composantes fortement connexes sont un raffinement des partitionnement des parcours.

Preuve :
Soient u et v deux sommets de la même composante fortement connexes. Supposons que u et v ne sont pas dans la même partie du partitionnement pour un parcours $(L_1, \dots, L_n)$ du graphe. Soit i_u et i_v tels que $u = L_{i_u}$ et $v = L_{i_v}$. Sans perdre en généralité, on peut supposer $i_u \leqslant i_v$. Il existe alors $i_0 \in]\!]i_u, i_v]\!]$ tels que L_{i_0} est un point de régénération. Alors,

$$\mathscr{B}(\{L_1, L_2, \dots, L_{i_0-1}\}) = \varnothing.$$

D'après le lemme précédent, il n'existe pas de chemin de u à v, ce qui est absurde car u et v sont dans la même composante fortement connexe. □

5.1.2 Rangement particulier de graphe

Définition (Sommet ouvert) **:** Soit $(L_1, \dots, L_n)$ un parcours de G. Pour $k \in [\![1, n]\!]$ et $i \in [\![1, k[\![$, on dit que L_i est *ouvert* à l'étape k si

$$\mathrm{Succ}(L_i) \not\subseteq \{L_j \mid j \in [\![1, k[\![\}$$

où $\mathrm{Succ}(L_i)$ est l'ensemble des successeurs de L_i.

Cette définition nous permet de définir les parcours en largeur et en profondeur.

Définition (parcours en largeur) **:** Soit $(L_1, \dots, L_n)$ un parcours. Il est dit *en largeur* si chaque sommet du parcours qui n'est pas un point de régénération est un successeur du premier sommet ouvert à cette étape :

$$\forall k \in [\![2, n]\!], \qquad \mathscr{B}(\{L_j \mid j \in [\![1, k[\![\} = \varnothing \quad \text{ou} \quad L_k \in \mathrm{Succ}(L_{i_0})$$

avec $i_0 = \min\{i \in [\![1, k[\![\mid L_i \text{ ouvert à l'étape } k\}$.

Définition (parcours en profondeur) **:** Soit $(L_1, \dots, L_n)$ un parcours. Il est dit *en largeur* si chaque sommet du parcours qui n'est pas un point de régénération est un successeur du dernier sommet ouvert à cette étape :

$$\forall k \in [\![2, n]\!], \qquad \mathscr{B}(\{L_j \mid j \in [\![1, k[\![\}) = \varnothing \quad \text{ou} \quad L_k \in \mathrm{Succ}(L_{i_0})$$

avec $i_0 = \max\{i \in [\![1, k[\![\mid L_i \text{ ouvert à l'étape } k\}$.

Exemple :
Dans le graphe ci-dessous, un parcours en largeur est

$$\underline{a} \to c \to b \to d \to f \rightsquigarrow \underline{e}.$$

Les sommets soulignées sont les points de régénération. On peut remarquer qu'il n'y a pas unicité du parcours en largeur, on aurait pu commencer le parcours par $\underline{a} \to b \to c \to \cdots$, et ce parcours est aussi un parcours en largeur.

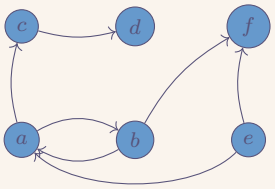


Figure 5.2 – Exemple de graphe orienté – parcours en largeur

Sur le même graphe, un parcours en profondeur est

$$\underline{a} \to b \to f \to c \to d \rightsquigarrow \underline{e}.$$

De même, il n'y a pas unicité du parcours en profondeur.

Définition (tri topologique) **:** Soit $(T_i)_{i\in[\![1,n]\!]}$ une permutation des sommets. On dit que T est un *tri topologique* si

$$\forall (i,j)\in[\![1,n]\!]^2, \quad \text{si } (T_i,T_j)\in A \text{ alors } i\leqslant j.$$

Exemple :
Un tri topologique du graphe ci-dessous est la permutation indiquée dans le graphe.

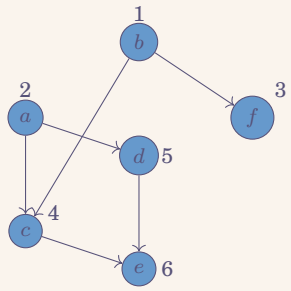


Figure 5.3 – Tri topologique d'un graphe

Définition : Soit une permutation de sommets $(T_i)_{i\in[\![1,n]\!]}$. On appelle *rang* de $u\in S$ dans T le plus petit indice dans T d'un élément accessible ($u \xrightarrow{\star} v$) et co-accessible ($v \xrightarrow{\star} u$) depuis u. On définit

$$\mathrm{rang}_T(u) = \min\{i\in[\![1,n]\!] \mid T_i \sim_G u\}.$$

Définition : Étant donné une permutation $(T_i)_{i\in[\![1,n]\!]}$ des sommets de G, on définit la relation

$$\preccurlyeq_T = \{(u,v)\in S^2 \mid \mathrm{rang}(u)\leqslant \mathrm{rang}(v)\}.$$

On dit alors que T est un *tri préfixe* dès lors que, pour tout $(u,v)\in S^2$, si $u \xrightarrow{\star} v$ alors $u\preccurlyeq_T v$.

Remarque :
Étant donné une permutation T, pour tout couple de sommets (u,v),

$$u\sim_G v \iff (u\preccurlyeq_T v \text{ et } v\preccurlyeq_T u).$$

5.1.3 Graphe transposé et CFC

Définition : Étant donné un graphe $G=(S,A)$, on appelle *graphe transposé* de G, que l'on note $G^\top$, le graphe

$$G^\top = \Big(S, \{(y,x)\in S^2 \mid (x,y)\in A\}\Big).$$

Propriété : Soit T un tri préfixe de G. Soit L un parcours de $G^\top$ utilisant l'ordre des points de régénération induit par T. Alors, la partition associée à L est la décomposition en composantes forment connexes.

Preuve :
Soit T un tri préfixe du graphe G. Soit L un parcours de $G^\top$. Montrons que le partitionnement associé à L est la décomposition en composantes fortement connexes. Il suffit de montrer que, si u et v sont dans la même partition du parcours L de $G^\top$: $u \sim_G v$.

Remarque : les composantes fortement connexes de G et $G^\top$ sont les mêmes.

Soient u et v deux sommets dans la même partition de L. C'est donc qu'il existe un point de régénération L_{r_k} tel que $L_{r_k} \xrightarrow{\star}_{G^\top} u$ et $L_{r_k} \xrightarrow{\star}_{G^\top} v$. Ainsi, $L_{r_k} \xleftarrow{\star}_{G} u$ et $L_{r_k} \xleftarrow{\star}_{G} v$. Alors, $\mathrm{rang}_T(u) \leqslant \mathrm{rang}_T(L_{r_k})$.

Supposons que $L_{r_k} \not\xrightarrow{\star}_{G} u$. Alors, $\mathrm{rang}_T(u) \neq \mathrm{rang}_T(L_{r_k})$. D'où, il existe $w \sim_G u$ apparaissant avant (strictement) L_{r_k} dans T. Ce qui est absurde car on aurait dû visiter w avant.

On en déduit que $L_{r_k} \xrightarrow{\star}_{G} u$, et $v \xrightarrow{\star}_{G} u$, De même pour v, on a $L_{r_k} \xrightarrow{\star}_{G} v$ et $u \xrightarrow{\star}_{G} v$. Finalement, on a

$$u \sim_G v.$$

□

Exemple :

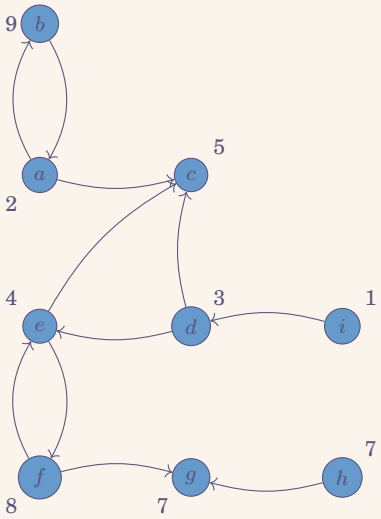


Figure 5.4 – Exemple de tri préfixe

Un tri préfixe dans le graphe ci-dessous est $g \leftarrow f \leftarrow c \leftarrow e \leftarrow d \rightsquigarrow i \rightsquigarrow b \leftarrow a \rightsquigarrow h$. Il s'agit d'un parcours en profondeur.

5.1.4 Calcul de tri préfixe

On peut donc donner un algorithme calculant un tri préfixe. On donne cet algorithme en impératif, même si la version récursive est plus simple.

Algorithme 5.1 Calcul d'un tri préfixe

Entrée Un graphe $G = (S, A)$.
Sortie Un tri préfixe des sommets de G.

```
1 : Procédure ExploreDescendants(s, Visités, Res)
2 :     Entrée Un graphe G = (S, A), Res, Visités, s ∈ S.
3 :     Sortie Modifie Res et Visités de sorte que Res soit un parcours préfixe de Visités et des
        sommets accessibles depuis s.
4 :     todo ← pileVide
5 :     empiler((s, Succ(s)), todo)
6 :     Visités ← {s} ∪ Visités
7 :     tant que todo ≠ pileVide faire
8 :         (x, ℓ) ← depiler(todo)
9 :         si ℓ = ( ) alors
10 :            Res ← x · Res
11 :        sinon
12 :            t · ℓ' ← ℓ        ▷ on sépare la tête t du reste ℓ' de la pile ℓ.
13 :            empiler((x, ℓ'), todo)
14 :            si t ∉ Visités alors
15 :                Visités ← {t} ∪ Visités
16 :                empiler((t, Succ(t)), todo)

17 : Visités ← ∅
18 : Res ← ( )
19 : tant que S \ Visités ≠ ∅ faire
20 :     s ← un sommet de S \ Visités
21 :     ExploreDescendants(s, Visités, Res)
22 : retourner Res
```

On montre la correction de cet algorithme. On cherche des invariants *intéressants*, que l'on ne prouvera pas. Pour la boucle "tant que," dans la procédure ExploreDescendants, on choisit les invariants

1. pour tout couple de sommets $(u, v) \in S^2$, si $\mathscr{C}\ell(u) \subseteq \text{Res}$ et que $u \xrightarrow{\star} v$, alors $\mathscr{C}\ell(v) \subseteq \text{Res}$ et $\text{rand}_{\text{Res}}(u) \leqslant \text{rang}_{\text{Res}}(v)$,

2. $K(\text{todo}) \cup \text{Res} = \text{Visités}$, où où $K(\text{todo})$ est l'ensemble des premières composantes des couples de todo,

3. les clés de todo, du fond de la pile au sommet forment un chemin,

4. si u est un élément de Visités, et v est un descendant de u,

   — ou bien $v \in \text{Res}$,

   — ou bien $v \in K(\text{todo})$

   — ou bien v est un descendant d'un élément d'une liste adjointe à un élément $w \in K(\text{todo})$ tel que $u \xrightarrow{\star} w$.

On admet que ces 4 propriétés sont invariantes. À la fin, $\forall x \in \text{Res}$, $\mathscr{C}\ell(x) \subseteq \text{Res}$, et dnc l'invariant 1 assure alors que nous avons un tri préfixe.

5.1.5 Algorithme de Kosaraju

Algorithme 5.2 Algorithme de Kosaraju

Entrée Un graphe $G = (S, A)$
Sortie Les composantes fortement connexes de G
1 : On calcule un tri préfixe de G.
2 : On parcours $G^\top$ en utilisant l'ordre T comme points de régénération.
3 : On retourne le plus petit partitionnement associé au parcours.

5.1.6 Applications

Théorème : 2-CNF-SAT $\in \mathbf{P}$.

Exemple :
On considère la formule $H = (x \vee \neg y) \wedge (\neg y \vee z) \wedge (y \vee \neg z) \wedge (y \vee z)$. Elle est équivalente à

$$\begin{aligned} H' = \quad & (\neg x \to \neg y) \\ & \wedge (y \to x) \\ & \wedge (y \to z) \\ & \wedge (\neg z \to \neg y) \\ & \wedge (z \to y) \\ & \wedge (\neg y \to \neg z) \\ & \wedge (\neg y \to z) \\ & \wedge (\neg z \to y) \end{aligned}$$

En replaçant les $\to$ par des arrêtes dans un un graphe, on obtient celui représenté ci-dessous.

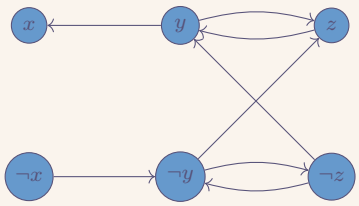


Figure 5.5 – Représentation d'une formule 2-CNF-SAT par un graphe

Preuve :
Soit $H \in$ 2CNF. On pose

$$H = (\ell_{1,1} \vee \ell_{1,2}) \wedge \ldots \wedge (\ell_{n,1} \vee \ell_{n,2}).$$

Dans la suite, on note $\ell^c_{i,j}$ le littéral opposé à $\ell_{i,j}$. À la formule H, nous associons le graphe G_H défini comme suit :

$$S_H = \{(\ell_{i,j}) \mid i \in [\![1, n]\!], j \in \{1, 2\}\} \cup \{(\ell^c_{i,j}) \mid i \in [\![1, n]\!], j \in \{1, 2\}\},$$
$$A_H = \{(\ell^c_{i,1}, \ell_{i,2} \mid i \in [\![1, n]\!]\} \cup \{(\ell^c_{i,2}, \ell_{i,1}) \mid i \in [\![1, n]\!]\}.$$

Lemme : Si ρ est un modèle de H et $u \xrightarrow{\star} v$ tel que $[\![u]\!]^\rho = \boldsymbol{V}$, alors $[\![v]\!]^\rho = \boldsymbol{V}$.

Preuve :
Soit ρ un modèle de H. Montrons par récurrence P_n : "si $u \xrightarrow{\star} v$ par un chemin de longueur n et $[\![u]\!]^\rho = \boldsymbol{V}$, alors $[\![v]\!]^\rho = \boldsymbol{V}$."

— P_0 : $u = v$, donc OK.

— P_{n+1} : supposons P_n vraie pour $n \in \mathbb{N}$. Soient u et v tels que

$$u \to u_1 \to u_2 \to \cdots \to u_n \to u_{n+1} = v$$

et $[\![u_n]\!]^\rho = \boldsymbol{V}$. D'après P_n, $[\![u_n]\!]^\rho = \boldsymbol{V}$. Or, $(u_n, u_{n+1}) \in A_H$. C'est donc que $u_n^c \vee u_{n+1} \in H$. Or, $[\![u_n]\!]^\rho = \boldsymbol{V}$, donc $[\![u_n^c]\!]^\rho = \boldsymbol{F}$. Or, $[\![u_n \vee v]\!]^\rho = \boldsymbol{V}$ et donc $[\![v]\!]^\rho = \boldsymbol{V}$.

On conclut par récurrence. □

Propriété : *H est satisfiable si, et seulement si aucune variable et sa négation ne se trouvent dans la même CFC de G_H.*

Preuve " $\Longrightarrow$ " Par contraposée, soit x et $\neg x$ se trouvant dans la même CFC de G_H. On procède par l'absurde. Soit ρ un environnement propositionnel tel que $[\![H]\!]^\rho = \boldsymbol{V}$.

— Si $\rho(x) = \boldsymbol{V}$, alors $[\![x]\!]^\rho = \boldsymbol{V}$. Or, $x \xrightarrow{\star} \neg x$. Alors, d'après le lemme, $[\![\neg x]\!]^\rho = \boldsymbol{V}$ et donc $\rho(x) = \boldsymbol{F}$, absurde.

— Si $\rho(x) = \boldsymbol{F}$, alors $[\![\neg x]\!]^\rho$. Or, $\neg x \xrightarrow{\star} x$ et donc, d'après le lemme, $[\![x]\!]^\rho = \boldsymbol{V}$ d'où $\rho(x) = \boldsymbol{V}$, absurde.

Il n'existe donc pas un tel ρ.

" $\Longleftarrow$ " Si G_H est telle qu'aucune variable et sa négation soient dans la même CFC. Soit $x \in \mathbb{Q}$ une variable propositionnelle. Soit $C_1, \ldots, C_p$ les composantes fortement connexes du graphe, triées par ordre topologique *i.e.* pour $(i, j) \in [\![1, p]\!]^2$, si $j > i$, alors il n'y a pas de chemin d'un élément de C_j vers un élément de C_i. Regardons alors où sont rangés x et $\neg x$. Si $x \in C_i$ et $\neg x \in C_j$ avec $i < j$, on définit alors $\rho(x) = \boldsymbol{F}$. Sinon, on définit $\rho(x) = \boldsymbol{V}$. Montrons alors que ρ est un modèle de H : $[\![H]\!]^\rho = \boldsymbol{V}$. Par l'absurde, soit $\ell_{i,1} \vee \ell_{i,2}$ une 2-clause de H telle que $[\![\ell_{i,1} \vee \ell_{i,2}]\!]^\rho = \boldsymbol{F}$, donc $[\![\ell_{i,1}]\!]^\rho = \boldsymbol{F}$ et $[\![\ell_{i,2}]\!]^\rho = \boldsymbol{F}$. Alors, $(\ell_{i,2}^c, \ell_{i,1})$ est une arrête de G_H et, $(\ell_{i,1}^c, \ell_{i,2})$ est une arrête de G_H. Ainsi, en notant a l'indice de la composante $\ell_{i,2}^c$, b l'indice de $\ell_{i,1}$, c l'indice de $\ell_{i,1}^c$ et d l'indice de $\ell_{i,2}$, on a $a \leqslant b$, $b < c$ par définition de ρ, $c \leqslant d$ et $d < a$ par définition de ρ, d'où

$$a \leqslant b < c \leqslant d < a,$$

ce qui est absurde. On a donc pour toute 2-clause $\ell_{i,1} \vee \ell_{i,2}$ de H, $[\![\ell_{i,1} \vee \ell_{i,2}]\!]^\rho = \boldsymbol{V}$ et donc $[\![H]\!]^\rho = \boldsymbol{V}$.

□

Algorithme 5.3 Solution au problème 2CNFSAT

Entrée H une 2-CNF
Sortie ρ un modèle de H ou None si H n'est pas satisfiable

```
1 : On construit G_H
2 : On construit les CFC C_1, ..., C_p de G_H (dans un ordre topologique)
3 : si il existe x et i ∈ [1, p] tel que x ∈ C_i et ¬x ∈ C_i alors
4 :     retourner None
5 : sinon
6 :     retourner ρ défini comme
```

$$\rho : \mathbb{Q} \longrightarrow \mathbb{B}$$
$$x \longmapsto \begin{cases} \boldsymbol{F} & \text{si } i < j \\ \boldsymbol{V} & \text{sinon} \end{cases}$$

où $x \in C_i$ et $\neg x \in C_j$.

□

5.2 Arbres couvrants de poids minimum

Exemple :
On considère le graphe ci-dessous.

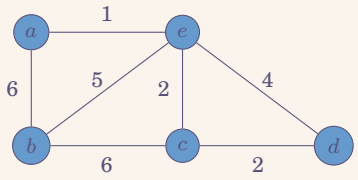


Figure 5.6 – Arbre pondéré

On cherche à « supprimer » des arrêtes de ce graphe afin d'avoir un poids total minimum, tout en conservant la connexité du graphe. Une structure assurant cette condition est un arbre.

Pour résoudre ce problème, on part du graphe vide, et on ajoute les arrêtes les moins coûteuses en premier.

Définition (Arbre) **:** Soit $G = (S, A)$ un graphe non-orienté. On dit que G est un *arbre* si G est connexe et acyclique.

Définition (Arbre couvrant) **:** Étant donné un graphe non orienté pondéré par poids positifs $G = (S, A, c)$,[1] on dit de $G' = (S', A')$ que c'est un *arbre couvrant* de G si $S' = S$ et $A' \subseteq A$, et G' est un arbre.

Définition (Arbre couvrant de poids minimum) **:** Étant donné un graphe non orienté pondéré $G = (S, A, c)$ et un arbre couvrant $T = (S', A')$, on appelle *poids* de l'arbre T la valeur $\sum_{a \in A'} c(a)$.

Si G est connexe, il admet au moins un arbre couvrant, on peut définir l'*arbre couvrant de poids minimum* (acpm).

On définir alors le problème

$$\text{ACPM}^{2} \begin{cases} \textbf{Entrée} & : G = (S, A, c) \text{ connexe} \\ \textbf{Sortie} & : \text{le poids de l'arbre couvrant de poids minimum.} \end{cases}$$

1. on dit que c est la fonction de pondération de ce graphe
2. Arbre Couvrant de Poids Minimum

Algorithme 5.4 Algorithme de Kruskal

Entrée $G = (S, A, c)$ un graphe connexe
Sortie Un arbre couvrant de poids minimum

$B \leftarrow \varnothing$
$U \leftarrow \varnothing$
tant que il existe u et v tels que $u \not\sim_B v$ **faire**
  Soit $\{x, y\} \in A \setminus U$ de poids minimal
  si $x \sim_B y$ **alors**
    $U \leftarrow \{\{x, y\}\} \cup U$
  sinon
    $U \leftarrow \{\{x, y\}\} \cup U$
    $B \leftarrow \{\{x, y\}\} \cup B$
retourner $T = (S, B)$

Propriété : L'algorithme de Kruskal est correct.

Preuve :
1. Il existe un arbre couvrant de poids minimum utilisant les arrêtes de B ;
2. $B \subseteq U \subseteq A$;
3. $\forall \{u, v\} \in U, u \sim_B v$.

Ces trois propriétés sont invariantes.

Initialement $B = \varnothing = U$, donc ok.

Propagation Soient $\underline{B}$ et $\underline{U}$ (resp. $\bar{B}, \bar{U}$) les valeurs de B et U avant (resp. après) une itération de boucle. Supposons que $\underline{B}$ et $\underline{U}$ satisfont les propriétés 1, 2 et 3. Montrons que $\bar{B}$ et $\bar{U}$ les satisfont aussi.

2. On a $\{x, y\} \in A$ et $\underline{B} \subseteq \underline{U} \subseteq A$, donc
$$\bar{B} \subseteq \underline{B} \cup \{\{x, y\}\} \subseteq \underline{U} \cup \{\{x, y\}\} \subseteq A.$$
3. Soit $\{u, v\} \in \bar{U}$.
   — Si $\{u, v\} \in \underline{U}$, alors de 3, $u \sim_{\underline{B}} v$. Or, $\underline{B} \subseteq \bar{B}$ et donc $u \sim_{\bar{B}} v$.
   — Sinon, $\{u, v\} = \{x, y\}$, alors $x = u$ et $v = y$.
      — Sous-cas 1 : $\bar{B} = \underline{B} \cup \{\{x, y\}\}$, alors $x \sim_{\bar{B}} y$.
      — Sous-cas 2 : $\bar{B} = \underline{B}$, alors par condition du **si**, $x \sim_{\underline{B}} y$ et donc $x \sim_{\bar{B}} y$.
1. Soit $\mathcal{T}$ un acpm contenant $\underline{B}$.
   — Cas 1 : $\bar{B} = \underline{B}$, ok
   — Cas 2 : $\bar{B} = \underline{B} \cup \{\{x, y\}\}$.
      — Sous-cas 1 : $\{x, y\} \in \mathcal{T}$, alors $\mathcal{T}$ est un acpm qui contient $\bar{B}$.
      — Sous-cas 2 : $\{x, y\} \notin \mathcal{T}$, $\mathcal{T}$ est un arbre couvrant, donc il contient une chaîne de x à y :
$$\overset{x}{\overset{\|}{\{x_0, x_1\}}}, \{x_1, x_2\}, \ldots, \{x_{n-1}, \overset{\|}{x_n}\}.\ \overset{y}{\|}$$
Or, $\forall i \in [\![1, n-1]\!]$, $x_i \sim_{\underline{B}} x_{i+1}$. Par transitivité, on a donc $x = x_0 \sim_{\underline{B}} x_n = y$, ce qui n'est pas le cas. Il existe donc $i_0 \in [\![0, n-1]\!]$, tel que $x_{i_0} \not\sim_{\underline{B}} x_{i_0+1}$ et donc $\{x_{i_0}, x_{i_0+1}\} \notin \underline{U}$. D'où, d'après 3, on a $\{x_{i_0}, x_{i_0+1}\} \notin \underline{B}$ Considérons alors $\mathcal{T}' = (\mathcal{T} \setminus \{\{x_{i_0}, x_{i_0+1}\}\}) \cup \{\{x, y\}\}$. Montrons que $\mathcal{T}'$ est un acpm contenant B, en commençant par montrer que c'est un arbre couvrant. L'arbre $\mathcal{T}'$ a $n-1$ arrêtes (autant que $\mathcal{T}$). Montrons que $\mathcal{T}'$ est connexe. Soit $(a, b) \in S^2$. $\mathcal{T}$ est connexe, soit donc une chaîne
$$C : \quad a = u_0, u_1, \ldots, u_n = b$$
de $\mathcal{T}$. Si la chaîne C n'utilise pas l'arrête $\{x_{i_0}, x_{i_0+1}\}$, alors C est une chaîne de $\mathcal{T}'$. Sinon, on pose μ et τ tels que
$$\underbrace{a, \ldots, x_{i_0}}_{\mu}, \underbrace{x_{i_0+1}, \ldots, b}_{\tau}.$$

Soit alors la chaîne

$$\overbrace{a, \dots, x_{i_0}}^{\mu}, x_{i_0-1}, x_{i_0-2}, \dots, x_0 = x,$$
$$\underbrace{b, \dots, x_{i_0+1}}_{\tau}, x_{i_0+2}, \dots, x_{n-1}, x_n = y$$

qui est dans $\mathcal{T}'$. Montrons que le poids est minimum. Notons $P(\mathcal{T})$ le poids de l'arbre. On a donc

$$P(\mathcal{T}') = P(\mathcal{T}) + c(\{x, y\}) - c(\{x_{i_0}, x_{i_0+1}\}).$$

Par choix glouton, $(\{x_{i_0}, x_{i_0+1} \notin U\}), c(\{x, y\}) \leqslant c(\{x_{i_0}, x_{i_0+1}\})$ donc $P(\mathcal{T}') \leqslant P(\mathcal{T})$, et $\mathcal{T}$ étant de poids min, $P(\mathcal{T}') = P(\mathcal{T})$ et $\mathcal{T}'$ est un ACPM contenant B.

Les invariants le sont. □

À la fin, B induit un graphe connexe et B est contenu dans un ACPM, c'en est donc un.

Une structure pour la gestion des partitions : UnionFind.

Définition (Type de données abstrait UnionFind) **:** On définit le type de données abstrait UnionFind comme contenant

- un type `t` de partitions ;
- un type `elem` des éléments manipulés par les partitions ;
- `initialise_partition` : `elem list` $\to$ `t` retournant le partitionnement dans lequel chaque élément est seul dans sa classe ;
- `find` : (`t * elem`) $\to$ `elem` retournant un représentant de la classe de l'élément. Si deux éléments x et y sont dans la même classe, dans le partitionnement p, alors `find`$(p, x) =$ `find`(p, y) ;
- `union` : (`t*elem*elem`) $\to$ `t` retourne le partitionnement dans lequel on a fusionné les classes des arguments.

EXEMPLE :
On réalise le *pseudo-code* ci-dessous.

- $p \leftarrow$ `initialise_partition`$([1, 2, 3, 4, 5]) \rightsquigarrow \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$
- `find`$(p, 1) = 1$
- `union`$(p, 1, 3) \rightsquigarrow \{\{1, 3\}, \{2\}, \{4\}, \{5\}\}$
- `find`$(p, 1) =$ `find`$(p, 3)$

On implémente ce type abstrait en OCAML.

REMARQUE (Niveau zéro – listes de liste) :

```
type 'a t = 'a list list

let initialise_partition (l: 'a list): 'a t =
  List.map (fun x -> [ x ] ) l

let rec find (p: 'a t) (x: 'a): 'a =
  match p with
  | classe :: classes ->
      if List.mem x classe then List.hd classe
      else find classes x
  | [] -> raise Not_Found

let est_equiv (p: 'a t) (x: 'a) (y: 'a): bool =
  (find p x) = (find p y)
```

```
15
16 let rec extrait_liste (x: 'a) (p: 'a t): 'a list * 'a p =
17   match p with
18   | classe :: classes ->
19       if List.mem x classe then (classe, classes)
20       else
21         let cl, cls' = extrait_liste x classes in
22         (cl, classe :: cls')
23   | [] -> raise Not_Found
24
25 let union (p: 'a t) (x: 'a) (y: 'a): 'a t =
26   if est_equiv p x y then p
27   else
28     let cx, p' = extrait_liste x p in
29     let cy, p'' = extrait_liste y p' in
30     (cx @ cy) :: p''
```

Code 5.1 – Implémentation du type UnionFind en OCaml

Remarque (Niveau un – tableau de classes) :
Dans la case du tableau, on inscrit le numéro de sa classe. Pour `find`, on prend le premier ayant la même classe. Pour `union`, on re-numérote vers un numéro commun. Par exemple,

0	1	0	0	1	2
0	1	2	3	4	5

$\longleftrightarrow \quad \{\{0,2,3\},\{1,4\},\{5\}\}.$

Remarque (Niveau deux – tableau de représentants) :
Dans les cases du tableau, on écrit le représentant de la classe de i. Pour `find`, on lit la case. Pour `union`, on re-numérote vers un numéro commun. Par exemple,

2	4	2	2	4	5
0	1	2	3	4	5

$\longleftrightarrow \quad \{\{0,2,3\},\{1,4\},\{5\}\}.$

Remarque (Niveau trois – arbres) :
Pour `union`$(0,1)$, on cherche le représentant de 0 (2) puis celui de 1 (4). On fait pointer 4 vers 2. Pour la suite de l'implémentation, *c.f.* DM3.

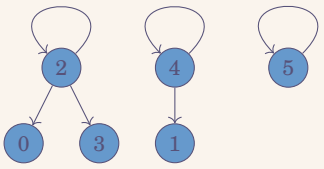


Figure 5.7 – Représentation par des arbres

Avec cette nouvelle structure, on peut maintenant revenir sur l'algorithme de Kruskal.

Algorithme 5.5 Algorithme de Kruskal – version 2

Entrée Un graphe $G = (S, A, c)$ un graphe non orienté, pondéré
Sortie Un acpm

```
1 : Soit (e_i)_{i∈⟦1,m⟧} un tri des arrêtes par coût croissant
2 : f ← 0        ▷ Nombre d'union effectuées
3 : p ← initialise_partition(S)
4 : I ← 0
5 : B ← ∅
6 : tant que f < n − 1 faire
7 :     {x, y} ← e_I
8 :     si find(p, x) ≠ find(p, y) alors
9 :         p ← union(p, x, y)
10 :        B ← B ∪ {{x, y}}
11 :        f ← f + 1
12 :    I ← I + 1
13 : retourner (S, B)
```

Étude de complexité. Notons $C^n_{\texttt{find}}$ un majorant du coût de `find` sur une structure contenant n éléments, notons $C^n_{\texttt{union}}$ un majorant du coût de `union` sur une structure contenant n éléments, et notons $C^n_{\texttt{init}}$ un majorant du coût de `init` sur une structure contenant n éléments. La complexité de cet algorithme est de

$$\mathcal{O}\big(C^n_{\texttt{init}} + 2m\, C^n_{\texttt{find}} + n\, C^n_{\texttt{union}} + m \log_2 m\big).$$

5.3 Couplage dans un graphe biparti

Définition (Couplage) **:** On appelle *couplage* d'un graphe non orienté $G = (S, A)$, la donnée d'un sous-ensemble $C \subseteq A$ tel que

$$\forall \{x, y\}, \{x', y'\} \in C, \qquad \{x, y\} \cap \{x', y'\} \neq \varnothing \implies \{x, y\} = \{x', y'\}.$$

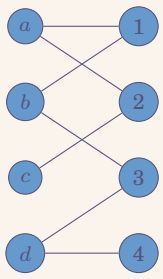


Figure 5.8 – Exemple de couplage

Exemple :
On réutilise l'exemple ci-dessous dans toute la section. L'ensemble $C = \{\{a, 2\}, \{b, 3\}\}$ est un couplage. Mais, l'ensemble $C' = \{\{a, 1\}, \{a, 2\}\}$ n'en est pas un.

Définition : Un couplage est dit *maximal* s'il est maximal pour l'inclusion ($\subseteq$). Un couplage est dit *maximum* si son cardinal est maximal.

Exemple :
Dans l'exemple précédent,
- le couplage $C = \{\{a, 2\}, \{b, 3\}\}$ n'est ni maximal, ni maximum ;
- le couplage $C' = \{\{a, 2\}, \{b, 3\}, \{d, 4\}\}$ est maximal mais pas maximum ;
- le couplage $C'' = \{\{a, 1\}, \{b, 3\}, \{c, 2\}, \{d, 4\}\}$ est maximum.

Remarque :
Dans toute la suite, on ne considère que des graphes bipartis.

Définition : Étant donné un graphe biparti $G = (S, A)$ et un couplage C, un sommet x est dit *libre* dès lors que

$$\forall \{y, z\} \in C,\ x \notin \{y, z\}.$$

Une chaîne élémentaire[3] $(c_0, c_1, \ldots, c_{2p+1})$ est dit *augmentante* si
- c_0 et c_{2n+1} sont libres ;
- $\forall i \in [\![0, p]\!]$, $\{c_{2i}, c_{2i+1}\} \in A \setminus C$;
- $\forall i \in [\![0, p-1]\!]$, $\{c_{2i+1}, c_{2i+2}\} \in C$.

Exemple :

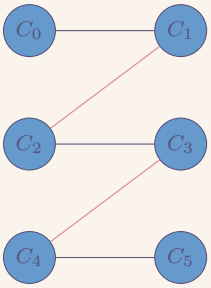


Figure 5.9 – Chaîne augmentante

Exemple :
Dans l'exemple de cette section, $(d, 4)$ et $(c, 2, a, 1)$ sont deux chaînes augmentantes.

Propriété : Étant donné un graphe biparti $G = (S, A)$ avec $S = S_1 \uplus S_2$ (partitionnement du graphe biparti), un couplage C est maximum si, et seulement s'il n'admet pas de chaînes augmentantes.

Preuve " $\Longrightarrow$ " Soit C un couplage admettant une chaîne augmentante. Montrons que C n'est pas maximum. Soit la chaîne augmentante[4]

$$c_0 \to c_1 \Rightarrow c_2 \to c_3 \Rightarrow c_4 \to \cdots \to c_{2p-1} \Rightarrow c_{2p} \to c_{2p+1}.$$

3. *i.e.* une chaîne sans boucles.

On considère alors le couplage

$$C' = \Big(C \setminus \{\{c_{2i+1}, c_{2i+2}\} \mid i \in [\![0, p-1]\!]\}\Big) \cup \{\{c_{2i}, c_{2i+1}\} \mid i \in [\![0, p]\!]\}.$$

On transforme donc la chaîne en

$$c_0 \Rightarrow c_1 \to c_2 \Rightarrow c_3 \to \cdots \to c_{2p-1} \to c_{2p} \Rightarrow c_{2p+1}.$$

C'est bien un couplage, et $\mathrm{Card}(C') = \mathrm{Card}\, C + 1$. C n'est donc pas un couplage maximum.

" $\Longleftarrow$ " Soit C un couplage non maximum. Montrons que C admet une chaîne augmentante. Soit M un couplage maximum, et $D = C \triangle M = (C \setminus M) \cup (M \setminus C)$. On a $\mathrm{Card}\, C < \mathrm{Card}\, M$ et $\mathrm{Card}(C \setminus M) < \mathrm{Card}(M \setminus C)$. On remarque que, si $c_0 \to c_1 \to c_2 \to \cdots \to c_{p-1} \to c_p$ est une chaîne de D, (si $c_0 \to c_1 \in C \setminus M$ et $c_1 \to c_2 \in C \setminus M$ donc c_1 est dans deux arrêtes distinctes d'un couplage C, ce qui est absurde ; de même pour les autres arrêtes). Ainsi, 2 arrêtes consécutives ne sont pas dans la même composante de l'union $(C \setminus M) \uplus (M \setminus C)$. Considérons la relation d'équivalence $\sim$ sur D définie par $\{x, y\} \sim \{z, t\} \overset{\text{def.}}{\iff}$ il existe une chaîne de D utilisant l'arrête $\{x, y\}$ et l'arrête $\{z, t\}$. Soit le partitionnement $D_1, \ldots, D_q$ de D par $\sim$. Par inégalité de cardinal, il existe un D_i tel que

$$\mathrm{Card}\{e \in D_i \mid e \in C\} < \mathrm{Card}\{e \in D_i \mid e \in M\}.$$

L'ensemble D_i contient alors une chaîne augmentante.

□

Algorithme 5.6 ChaîneAugmentante : Trouver une chaîne augmentante dans un graphe biparti $G = (S, A)$ muni d'un couplage C partant d'un sommet $s \in S$

```
1 : Procédure Augmente(x, chaîne)
2 :     pour y ∈ Succ(x) \ chaîne faire
3 :         si y est libre dans C alors
4 :             retourner Some(chaîne ⊎ (y))
5 :         sinon
6 :             Soit z tel que {y, z} ∈ C.
7 :             r ← Augmente(z, chaîne ⊎ (y, z))
8 :             si r ≠ None alors
9 :                 retourner r
10 :        retourner None
11 : si s est libre dans C alors
12 :     retourner Augmente(s, (s))
13 : sinon
14 :     retourner None
```

Remarque :
Si un sommet n'est pas libre dans le couplage C, il n'est pas libre dans les couplage obtenus par inversion de chaîne depuis C.

Algorithme 5.7 Calcul d'un couplage maximum

```
Entrée G = (S, A) un graphe biparti, avec S = S1 ⊎ S2
1 : C ← ∅
2 : Done ← ∅
3 : tant que ∃x ∈ S1 \ Done faire
4 :     Soit un tel x.
5 :     r ← ChaîneAugmentante(G, C, x)
6 :     si r ≠ None alors
7 :         Some(a) ← r
8 :         On inverse la chaîne a dans C.
9 :     Done ← {x} ∪ Done
10 : retourner C
```

4. On représente $\Rightarrow$ pour les arrêtes dans le couplage C.

Annexe 5.A Remarques supplémentaires

Le parcours d'un graphe $G = (S, A)$ a une complexité en $\mathcal{O}(|S| + |A|)$.

CHAPITRE

6

PREUVES

Sommaire

6.0 Motivation **141**
6.0.1 Tables de vérité 141
6.0.2 Équations 142
6.0.3 Raisonnement mathématiques 142
6.1 La déduction naturelle en logique propositionnelle **142**
6.1.1 Séquents 142
6.1.2 Preuves 143
6.1.3 Déduction naturelle 145
6.2 La logique du premier ordre **148**
6.2.1 Syntaxe de la logique du premier ordre 148
6.2.2 Substitution 151
6.2.3 Extension au premier ordre de la déduction naturelle 153
6.2.4 Règles dérivées 155
6.2.5 Sémantique 155
6.3 Synthèse du chapitre **159**

L'objectif de ce chapitre, sera de "critiquer" le travail en logique fait précédement, puis d'apporter une solution à ce problème ; on finira par un peu de HORS-PROGRAMME.

6.0 Motivation

6.0.1 Tables de vérité

Pour l'instant, pour montrer $\Gamma \models G$ ou $G \equiv H$, nous devons encore utiliser une table de vérité. Par exemple, montrons

$$\underbrace{(p \to q) \wedge (q \to r)}_{G} \models \overbrace{p \to r}^{H}.$$

On réalise la table de vérité ci-dessous.

p	q	r	$p \to q$	$q \to r$	G	H	
$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{V}$	✓
$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{V}$	$\boldsymbol{V}$	✓
$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{V}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{V}$	✓

À faire : Finir table de vérité

TABLE 6.1 – Table de vérité pour montrer $(p \to q) \wedge (q \to r) \models p \to r$

6.0.2 Équations

Supposons $[\![G]\!]^\rho = \boldsymbol{V}$. Montrons que $[\![H]\!]^\rho = \boldsymbol{V}$. On a

$$\begin{aligned} \boldsymbol{V} &= [\![G]\!]^\rho \\ &= [\![(p \to q) \wedge (q \to r)]\!]^\rho \\ &\vdots \\ &= \overline{[\![p]\!]^\rho} \cdot \overline{[\![q]\!]^\rho} + \overline{[\![p]\!]^\rho} \cdot \overline{[\![q]\!]^\rho} \cdot [\![r]\!]^\rho + [\![q]\!]^\rho \cdot [\![r]\!]^\rho \end{aligned}$$

et

$$\begin{aligned} [\![H]\!]^\rho &= [\![p \to r]\!]^\rho \\ &\vdots \\ &= \overline{[\![p]\!]^\rho} \cdot \overline{[\![q]\!]^\rho} + \overline{[\![p]\!]^\rho} \cdot \overline{[\![q]\!]^\rho} \\ &\quad + [\![r]\!]^\rho \cdot \overline{[\![q]\!]^\rho} \cdot \left([\![p]\!]^\rho + \overline{[\![p]\!]^\rho} \right) \\ &\quad + [\![r]\!]^\rho + ? \end{aligned}$$

À faire : finir le calcul

6.0.3 Raisonnement mathématiques

Supposons $(p \to q) \wedge (q \to r)$. Montrons que $p \to r$.

↪ Supposons donc p. Montrons r
- ↪ Montrons q.
  - ↪ Montrons p, qui est une hypothèse.
  - ↪ Montrons $p \to q$, qui est aussi une hypothèse.
- — Montrons $q \to r$, ce qui est vrai par hypothèse.

On reconnaît un arbre.

6.1 La déduction naturelle en logique propositionnelle

6.1.1 Séquents

Objectifs de preuves.

EXEMPLE :
Montrons $P \wedge Q$ est vrai.
↪ Montrons P.

↪ Montrons Q.

Hypothèses courantes.

Exemple :
Montrons que $(n \in 4\mathbb{N} \to n \in 2\mathbb{N}) \wedge (n \in 4\mathbb{N}+3 \to n \in 2\mathbb{N}+1)$.

↪ Montrons $n \in 4\mathbb{N} \to 2\mathbb{N}$.
↪ Supposons $n \in 4\mathbb{N}$ (local). Montrons $n \in 2\mathbb{N}$.
↪ Montrons $n \in 4\mathbb{N}+3 \to n \in 2\mathbb{N}+1$.
↪ Supposons $n \in 4\mathbb{N}+3$. Montrons $n \in 2\mathbb{N}+1$.

Définition (Séquent) **:** Un *séquent* est la donnée
— d'un ensemble d'hypothèses Γ ;
— d'un objectif G.
On le typographie $\Gamma \vdash G$.

Exemple :
Montrons $\varnothing \vdash (n \in 4\mathbb{N} \to n \in 2\mathbb{N}) \wedge (n \in 4\mathbb{N}+3 \to n \in 2\mathbb{N}+1)$
↪ $\varnothing \vdash (n \in 4\mathbb{N} \to n \in 2\mathbb{N})$
↪ $\{n \in 4\mathbb{N}\} \vdash n \in 2\mathbb{N}$
↪ $\varnothing \vdash (n \in 4\mathbb{N}+3 \to n \in 2\mathbb{N}+1)$
↪ $\{n \in 4\mathbb{N}+3\} \to n \in 2\mathbb{N}+1$
On typographie cette preuve sous forme d'un arbre.ăÀ faire : Arbre à faire

6.1.2 Preuves

Définition : On appelle *règle de construction de preuves* une règle de la forme :

$$\frac{\Gamma_1 \vdash \varphi_1 \quad \Gamma_2 \vdash \varphi_2 \quad \Gamma_3 \vdash \varphi_3 \quad \cdots \quad \Gamma_n \vdash \varphi_n}{\Gamma \vdash \varphi}\ \text{nom}.$$

On appelle $\Gamma_1 \vdash \varphi_1, \Gamma_2 \vdash \varphi_2, \Gamma_3 \vdash \varphi_3, \ldots, \Gamma_n \vdash \varphi_n$ les *prémisses*, et $\Gamma \vdash \varphi$ la *conclusion*.

Si $n = 0$, on dit que c'est une *règle de base*.

Remarque (Notation) :
Γ est un ensemble. Alors, l'ensemble $\Gamma \cup \{\psi\}$ est noté Γ, ψ.

Exemple :
Un *axiome* est de la forme

$$\frac{}{\Gamma, \varphi \vdash \varphi}\ \text{Ax}.$$

Une preuve de la forme, appelée *introduction du* ET,

$$\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\ \wedge\text{i}$$

permet de prouver un ET. Il correspond au raisonnement mathématique suivant : supposons Γ ; montrons $\varphi \wedge \psi$;

$\hookrightarrow$ montrons φ ;

$\hookrightarrow$ montrons ψ.

Définition (Arbre de preuve) **:** On appelle *arbre de preuve* un arbre étiqueté par des séquents, et dont les liens père–fils sont des liens autorisés par les règles du système de preuves. Un *système de preuves* étant un ensemble de règles.

EXEMPLE (Système Jouet) :

$$\frac{}{\Gamma, \varphi \vdash \varphi}\ \text{Ax} \quad \frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\ \wedge\text{i} \quad \frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}\ \vee\text{i,g} \quad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}\ \vee\text{i,d}.$$

EXEMPLE :
Avec le système précédent,

$$\dfrac{\dfrac{\dfrac{}{\{P,Q,R\} \vdash P}\ \text{Ax}}{\{P,Q,R\} \vdash P \vee Q}\ \vee\text{i,g} \qquad \dfrac{\dfrac{}{\{P,Q,R\} \vdash Q}\ \text{Ax}}{\{P,Q,R\} \vdash Q \vee \neg R}\ \vee\text{i,g}}{\{P,Q,R\} \vdash (P \vee Q) \wedge (Q \vee \neg R)}\ \wedge\text{i}}{\{P,Q,R\} \vdash (P \wedge X) \vee \big((P \vee Q) \wedge (Q \vee \neg R)\big)}\ \vee\text{i,d}.$$

Définition (Être prouvable) **:** On dit d'un séquent $\Gamma \vdash G$ qu'il est *prouvable* dans un système de preuve dès lors qu'il existe une preuve dont la racine est étiquetée par $\Gamma \vdash G$.

RAPPEL (objectifs) :
On veut trouver d'autres moyens de montrer $F \models G$. On veut que, si $F \vdash G$, alors $F \models G$ (correction). Mais, on veut aussi que, si $F \models G$, alors $G \vdash F$ (complétude). On veut aussi qu'il existe un algorithme qui vérifie $F \models G$ (décidabilité).

Définition (Correction) **:** On dit d'un système de preuve qu'il est *correct* dès lors que : pour tout Γ, pour tout G, si $\Gamma \vdash G$ admet une preuve, alors $\Gamma \models G$.

EXEMPLE : 1. On pose la règle "Menteur" définie comme

$$\frac{}{\Gamma \vdash \bot}\ \text{Menteur}.$$

Ce système de preuve n'est pas correct car $\{\top\} \not\models \bot$. Or,

$$\frac{}{\{\top\} \vdash \bot}\ \text{Menteur}.$$

2. Le système jouet est correct. Montrons cela par induction sur la preuve de $\Gamma \vdash G$.
   — Si la preuve de $\Gamma \vdash G$ est de la forme

$$\frac{}{\Gamma', G \vdash G}\ \text{Ax}.$$

   Montrons que $\Gamma' \cup \{G\} \models \{G\}$. Soit donc ρ un modèle de $\Gamma \cup \{G\}$. Alors $\forall \varphi \in \Gamma \cup \{G\}$, $[\![\varphi]\!]^\rho = \boldsymbol{V}$. Montrons que ρ est un modèle de G. On pose $\varphi = G$; on a donc $[\![G]\!]^\rho = \boldsymbol{V}$.
   — Si la preuve de $\Gamma \vdash G$ est de la forme

$$\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\ \wedge\text{i}.$$

   On appelle π_1 la branche gauche de l'arbre, et π_2 la branche de droite. Par hypothèse d'induction sur π_1, $\Gamma \vdash \varphi$ admet une preuve (qui est une sous-preuve), donc $\Gamma \models \varphi$. De même, $\Gamma \models \psi$ avec la branche π_2. On en déduit que $\Gamma \models \varphi \wedge \psi$ d'après les résultats du chapitre 0.
   — De même pour les autres cas.

Définition (Complétude) **:** Un système de preuves est *complet* dès lors que, si $\Gamma \models G$, alors il existe un preuve de $\Gamma \vdash G$.

Exemple :
Le système de preuve ayant pour règle, pour tout Γ, et tout G,

$$\frac{}{\Gamma \vdash G}\ \text{OP}$$

est complet mais pas correct.

Exemple :
Le système de preuve ayant pour règle, pour tout Γ, et tout G tel que $\Gamma \models G$,

$$\frac{}{\Gamma \vdash G}$$

est complet et correct.

6.1.3 Déduction naturelle

On définit les différentes règles d'introduction et d'élimination suivantes.

Symbole	Règle d'introduction	Règle d'élimination
$\top$	$\dfrac{}{\Gamma \vdash \top}\ \top\text{i}$	
$\bot$		$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash G}\ \bot\text{e}$
$\neg$	$\dfrac{\Gamma, G \vdash \bot}{\Gamma \vdash \neg G}\ \neg\text{i}$	$\dfrac{\Gamma \vdash G \quad \Gamma \vdash \neg G}{\Gamma \vdash \bot}\ \neg\text{e}$
$\to$	$\dfrac{\Gamma, G \vdash H}{\Gamma \vdash G \to H}\ \to\text{i}$	$\dfrac{\Gamma \vdash H \to G \quad \Gamma \vdash H}{\Gamma \vdash G}\ \to\text{e}$
$\wedge$	$\dfrac{\Gamma \vdash G \quad \Gamma \vdash H}{\Gamma \vdash G \wedge H}\ \wedge\text{i}$	$\dfrac{\Gamma \vdash G \wedge H}{\Gamma \vdash G}\ \wedge\text{e,g} \qquad \dfrac{\Gamma \vdash G \wedge H}{\Gamma \vdash H}\ \wedge\text{e,d}$
$\vee$	$\dfrac{\Gamma \vdash G}{\Gamma \vdash G \vee H}\ \vee\text{i,g} \qquad \dfrac{\Gamma \vdash H}{\Gamma \vdash G \vee H}\ \vee\text{i,d}$	$\dfrac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash G \quad \Gamma, B \vdash G}{\Gamma \vdash G}\ \vee\text{e}$
	$\dfrac{}{\Gamma, \varphi \vdash \varphi}\ \text{Ax}$	

Table 6.2 – Règles d'introduction et d'élimination

À ce stade, nous avons définis le système de preuves que l'on appellera *déduction naturelle intuitionniste*. Dans le chapitre 0, on a donné une notion de vérité. On a maintenant donné une notion de preuve. On souhaite maintenant montrer le séquent $\varnothing \vdash p \vee \neg p$, nommé *tiers exclu* : la variable p est, soit vrai, soit fausse. Avec le système de preuve actuel, on ne peut pas le montrer. Mais, on a bien $\varnothing \models p \vee \neg p$, car pour tout environment propositionnel ρ, $[\![p \vee \neg p]\!]^\rho = \boldsymbol{V}$. D'où la remarque suivante.

Remarque :
Ce système de preuve n'est pas complet vis à vis de la sémantique de la logique propositionnelle : on ne peut pas prouver le séquent $\varnothing \vdash p \vee \neg p$ malgré son caractère tautologique.

Exemple :
De plus, montrons le résultat : il existe deux irrationnels x et y tels que x^y soit rationnel. On considère le réel $\sqrt{2}^{\sqrt{2}}$. S'il est rationnel, la preuve est terminée. S'il ne l'est pas, notons $x = \sqrt{2}^{\sqrt{2}}$, et on remarque que $x^{\sqrt{2}} = 2$. Dans cette preuve, on utilise le tiers exclu : x est soit rationnel, soit irrationnel.

Ainsi, la *déduction naturelle classique* est le système de preuve obtenue en ajoutant la règle suivante :

$$\frac{}{\Gamma \vdash G \vee \neg G}\ \text{TE}.$$

La déduction naturelle classique est un système de preuve complet.

Exemple (preuve en déduction naturelle classique) :
Montrons le séquent $\varnothing \vdash \neg\neg p \to p$. Une preuve de ce séquent n'était pas possible en déduction naturelle intuitionniste, mais elle est possible en déduction naturelle classique

à l'aide de la règle du tiers exclu TE.

$$\dfrac{\dfrac{}{\neg\neg p \vdash p \vee \neg p}\,\text{TE} \qquad \dfrac{}{\neg\neg p, p \vdash p}\,\text{Ax} \qquad \dfrac{\dfrac{\dfrac{}{\neg\neg p, \neg p \vdash \neg p}\,\text{Ax} \qquad \dfrac{}{\neg\neg p, \neg p \vdash p}\,\text{Ax}}{\neg\neg p, \neg p \vdash \bot}\,\neg\text{e}}{\neg\neg p, \neg p \vdash p}\,\bot\text{e}}{\dfrac{\neg\neg p \vdash p}{\varnothing \vdash \neg\neg p \to p}\,\to\text{i}}\,\vee\text{e}$$

Exemple (preuve en dédution naturelle intuitionniste) :
On montre maintenant l'implication inverse. S'il existe une preuve en déduction naturelle intuitionniste, elle reste valide dans le système de preuve de la déduction naturelle classique.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p, \neg p \vdash p}\,\text{Ax} \qquad \dfrac{}{p, \neg p \vdash \neg p}\,\text{Ax}}{p, \neg p \vdash \bot}\,\neg\text{e}}{p \vdash \neg\neg p}\,\neg\text{i}}{\vdash p \to (\neg\neg p)}\,\to\text{i}$$

Exemple :
On prouve le séquent introductif $\varnothing \vdash \big((p \to q) \wedge (q \to r)\big) \to (p \to r)$, à l'aide le la déduction naturelle intuitionniste. On nomme Γ l'ensemble $p, (p \to q) \wedge (q \to r)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash (p \to q) \wedge (q \to r)}\,\text{Ax}}{\Gamma \vdash q \to r}\,\wedge\text{e,d} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash (p \to q) \wedge (q \to r)}\,\text{Ax}}{\Gamma \vdash p \to q}\,\wedge\text{e,g} \qquad \dfrac{}{\Gamma \vdash p}\,\text{Ax}}{\Gamma \vdash q}\,\to\text{e}}{p, (p \to q) \wedge (q \to r) \vdash r}\,\to\text{i}}{(p \to q) \wedge (q \to r) \vdash p \to r}\,\to\text{i}}{\varnothing \vdash \big((p \to q) \wedge (q \to r)\big) \to (p \to r)}\,\to\text{i}\,.$$

Exemple :
Toujours en déduction naturelle intuitionniste, on prouve le séquent

$$\varnothing \vdash (p \to q) \to (\neg q \to \neg p).$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p, \neg q, p \to q \vdash \neg q}\,\text{Ax} \qquad \dfrac{\dfrac{}{p, \neg q, p \to q \vdash p \to q}\,\text{Ax} \qquad \dfrac{}{p, \neg q, p \to q \vdash p}\,\text{Ax}}{p, \neg q, p \to q \vdash q}\,\to\text{e}}{p, \neg q, p \to q \vdash \bot}\,\neg\text{e}}{\neg q, p \to q \vdash \neg p}\,\neg\text{i}}{p \to q \vdash \neg q \to \neg p}\,\to\text{i}}{\vdash (p \to q) \to (\neg q \to \neg p)}\,\to\text{i}\,.$$

Théorème : La déduction naturelle classique (respectivement intuitionniste) est correcte.

Preuve :
Par induction (longue) sur l'arbre de preuves (*c.f.* plus haut). □

Corollaire : Pour prouver $\Gamma \models G$, il suffit de construire un arbre de preuve de $\Gamma \vdash G$.

Remarque :
On aurait pu définir la déduction naturelle classique en ajoutant une des deux règles suivantes plutôt que le tiers exclus :

$$\frac{\Gamma \vdash \neg\neg G}{\Gamma \vdash G}\ \neg\neg\text{e} \qquad \frac{\Gamma, \neg G \vdash \bot}{\Gamma \vdash G}\ \text{Abs}^{1}.$$

Exercice :
Refaire les preuves du séquent $\varnothing \vdash \neg\neg p \to p$ avec les règles $\neg\neg$e, et Abs.

6.2 La logique du premier ordre

On veut rajouter à la déduction naturelle des quantificateurs, tels que $\forall$ ou $\exists$. On considère la formule

$$G = \forall x,\ \Big(((x > 0) \wedge (\exists y,\ x = y + 1)) \vee (x = 0)\Big).$$

Cette formule peut être représentée sous forme d'arbre syntaxique, comme celui ci-dessous.

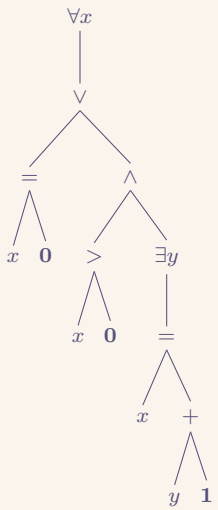


Figure 6.1 – Arbre syntaxique de la formule $G = \forall x,\ \Big(((x > 0) \wedge (\exists y,\ x = y + 1)) \vee (x = 0)\Big)$

6.2.1 Syntaxe de la logique du premier ordre

Définition : On appelle *signature du premier ordre* la donnée de deux ensembles $\mathcal{S}$ et $\mathcal{P}$.[2] Ces symboles viennent avec une notion d'arité

$$\mathfrak{a} : \mathcal{S} \cup \mathcal{P} \longrightarrow \mathbb{N}.$$

On appelle l'ensemble des *constantes* la sous-partie des éléments c de $\mathcal{S}$ telle que $\mathfrak{a}(c) = 0$.

1. Abs correspond à absurde

Les autres symboles, non constantes, sont appelés *fonctions*. On appelle $\mathcal{P}$ l'ensemble des prédicats. On a toujours $\mathcal{S} \cap \mathcal{P} = \varnothing$.

Définition : Étant donné un ensemble $\mathcal{S}$ de symboles de fonctions et de constantes, et un ensemble $\mathcal{V}$ de variables, on définit inductivement l'ensemble des *termes* sur $\mathcal{S}$ et $\mathcal{V}$, typographié $\mathcal{T}(\mathcal{S},\mathcal{V})$, par

— $\mathcal{V} \subseteq \mathcal{T}(\mathcal{S},\mathcal{V})$;

— si $f \in \mathcal{S}$, et $t_1, t_2, \ldots, t_{\mathfrak{a}(f)} \in \big(\mathcal{T}(\mathcal{S},\mathcal{V})\big)^{\mathfrak{a}(f)}$, alors $f\big(t_1, t_2, \ldots, f_{\mathfrak{a}(f)}\big) \in \mathcal{T}(\mathcal{S},\mathcal{V})$.

Exemple :
Si $\mathcal{S} = \{+, -, \mathbf{0}\}$, avec $\mathfrak{a}(+) = 2$, $\mathfrak{a}(-) = 1$ [3] et $\mathfrak{a}(\mathbf{0}) = 0$; si $\mathcal{V} \supseteq \{x, y, z\}$, alors

— $+(x, y)$

— $-(x)$

— $\mathbf{0}$

— $+\Big(x, -\big(+(z, \mathbf{0})\big)\Big)$

sont des termes.

Définition (Logique du premier ordre) **:** Étant donné une signature du premier ordre $(\mathcal{S},\mathcal{P})$, et un ensemble $\mathcal{V}$ de variables, on définit l'ensemble des formules des *formules de la logique du premier ordre* typographié $\mathcal{F}(\mathcal{S},\mathcal{P},\mathcal{V})$, par induction

— si $P \in \mathcal{P}$, et $t_1, \ldots, t_{\mathfrak{a}(P)} \in \big(\mathcal{T}(\mathcal{S},\mathcal{V})\big)^{\mathfrak{a}(P)}$, alors $P\big(t_1, \ldots, t_{\mathfrak{a}(P)}\big) \in \mathcal{F}(\mathcal{S},\mathcal{P},\mathcal{V})$;

— $\bot, \top \in \mathcal{F}(\mathcal{S},\mathcal{P},\mathcal{V})$;

— si $(G, H) \in \mathcal{F}(\mathcal{S},\mathcal{P},\mathcal{V})^2$, alors

$$G \wedge H \in \mathcal{F}(\mathcal{S},\mathcal{P},\mathcal{V}), \quad G \to H \in \mathcal{F}(\mathcal{S},\mathcal{P},\mathcal{V}), \quad \neg G \in \mathcal{F}(\mathcal{S},\mathcal{P},\mathcal{V}) ;$$
$$G \vee H \in \mathcal{F}(\mathcal{S},\mathcal{P},\mathcal{V}), \quad G \leftrightarrow H \in \mathcal{F}(\mathcal{S},\mathcal{P},\mathcal{V}),$$

— Si $x \in \mathcal{V}$ et $G \in \mathcal{F}(\mathcal{S},\mathcal{P},\mathcal{V})$, alors

$$(\forall x,\ G) \in \mathcal{F}(\mathcal{S},\mathcal{P},\mathcal{V}) \qquad (\exists x,\ G) \in \mathcal{F}(\mathcal{S},\mathcal{P},\mathcal{V}).$$

On note un symbole $+$ avec son arité $\mathfrak{a}(+) = 2$ comme $+(2)$.

Exemple :
En choisissant $\mathcal{P} = \{>(2), =(2)\}$, $\mathcal{S} = \{+(2), \mathbf{0}(0), \mathbf{1}(0)\}$ et $\mathcal{V} \supseteq \{x, y\}$, on peut alors construire la formule de l'exemple précédent :

$$G = \forall x,\ \Big(\big((x > 0) \wedge (\exists y,\ x = y + 1)\big) \vee (x = 0)\Big).$$

Codons le en OCaml, comme montré ci-dessous.

```
type symbole_arite = string * int

type signature = {
  symbole_terme: symbole_arite list;
```

2. $\mathcal{S}$ est l'ensemble des symboles utilisés pour construire des termes ; $\mathcal{P}$ est l'ensemble des symboles utilisés pour passer du monde des termes pour passer au monde des formules.
3. il s'agit du '$-$' dans l'expression $-x$.

```
  symbole_predicat: symbole_arite list
}

type var = string

type terme =
  | V of var
  | T of symbole_arite * (terme list)

(* Quelques exemples *)

let ex0 = T(("0", 0), []);
let ex1 = T(("1", 0), []);
let ex2 =
  T(("+", 2), [
    V("x"),
    T(("-", 1), [
      T(("+", 2), [
        V("z"),
        T(("0", 0), [])
      ])
    ])
  ])

(* Definissons la logique du 1er ordre *)

type po_logique =
  | Pred    of symbole_arite * (terme list)
  | Top | Bottom
  | And     of po_logique * po_logique
  | Or      of po_logique * po_logique
  | Imp     of po_logique * po_logique
  | Equiv   of po_logique * po_logique
  | Not     of po_logique
  | Forall  of var * po_logique
  | Exists  of var * po_logique
```

Code 6.1 – Définition des formules de premier ordre en OCaml

On définit, dans la suite de cette section, l'introduction et l'élimination de $\forall$ et $\exists$. Mais, nous devons réaliser des *substitutions*, et c'est ce que nous allons faire dans le reste de cette sous-section.

Définition : On définit vars inductivement sur $\mathcal{T}(\mathcal{S},\mathcal{V})$ par :

— si $x \in \mathcal{V}$, $\mathrm{vars}(x) = \{x\}$;

— $\mathrm{vars}\big(f(t_1,\dots,t_n)\big) = \bigcup_{i=1}^{n} \mathrm{vars}(t_i)$.

Définition : On définit inductivement deux fonctions

$$\mathsf{FV} : \mathcal{F}(\mathcal{S},\mathcal{P},\mathcal{V}) \longrightarrow \wp(\mathcal{V})\,[4] \qquad \mathsf{BV} : \mathcal{F}(\mathcal{S},\mathcal{P},\mathcal{V}) \longrightarrow \wp(\mathcal{V})\,[5]$$

par

— $\mathsf{FV}(\top) = \varnothing$,

— $\mathsf{FV}(\bot) = \varnothing$,

— $\mathsf{FV}(G \odot H) = \mathsf{FV}(G) \cup \mathsf{FV}(H)$ avec $\odot \in \{\wedge, \vee, \to, \leftrightarrow\}$,

— $\mathsf{FV}\big(P(t_1,\dots,t_n)\big) = \bigcup_{i=1}^{n} \mathrm{vars}(t_i)$,

— $\mathsf{FV}(\neg G) = \mathsf{FV}(G)$,

— $\mathsf{FV}(\forall x,\, G) = \mathsf{FV}(G) \setminus \{x\}$,

— $\mathsf{FV}(\exists x,\, G) = \mathsf{FV}(G) \setminus \{x\}$,

et

- $\mathsf{BV}(\top) = \varnothing$,
- $\mathsf{BV}(\bot) = \varnothing$,
- $\mathsf{BV}(P(t_1, \dots, t_n)) = \varnothing$,
- $\mathsf{BV}(\neg G) = \mathsf{BV}(G)$,
- $\mathsf{BV}(G \odot H) = \mathsf{BV}(G) \cup \mathsf{BV}(H)$ avec $\odot \in \{\wedge, \vee, \to, \leftrightarrow\}$,
- $\mathsf{BV}(\forall x,\ G) = \mathsf{BV}(G) \cup \{x\}$,
- $\mathsf{BV}(\exists x,\ G) = \mathsf{BV}(G) \cup \{x\}$,

Exemple :
À faire : Inclure exemple. $\mathsf{BV}(F) = \{x, y\}$ et $\mathsf{FV}(F) = \{y, z\}$.

Définition (α-renommage) **:** On appelle *α-renommage* l'opération consistant à renommer les occurrences liées des variables dans une formule.

Exemple :
On considère la formule

$$(\forall x,\ P(X)) \wedge (\forall x,\ \forall y,\ Q(x, x + y)).$$

Elle a pour α-renommage les formules

- $(\forall x,\ P(x)) \wedge (\forall x,\ \forall y,\ Q(x, x + y))$;
- $(\forall z,\ P(z)) \wedge (\forall x,\ \forall y,\ Q(x, x + y))$;
- $(\forall z,\ P(z)) \wedge (\forall z,\ \forall y,\ Q(z, z + y))$;
- $(\forall z,\ P(z)) \wedge (\forall x,\ \forall z,\ Q(x, x + z))$;
- ~~$(\forall z,\ P(z)) \wedge (\forall y,\ \forall y,\ Q(y, y + y))$~~.

6.2.2 Substitution

Définition (Substitution) **:** Une *substitution* est une fonction de $\mathcal{V} \longrightarrow \mathcal{T}(\mathcal{S}, \mathcal{V})$ qui est l'identité partout, sauf sur un nombre fini de variables que l'on appelle *clé* de cette substitution.

Exemple :
On considère la substitution

$$\sigma : \mathcal{V} \longrightarrow \mathcal{T}(\mathcal{S}, \mathcal{V})$$
$$x \longmapsto \begin{cases} y + y & \text{si } x = y \\ x & \text{sinon.} \end{cases}$$

L'ensemble des clés de cette substitution sont $\{y\}$; en effet, $\sigma(y) = y + y$, et $\sigma(z) = z$.

Définition (Application d'une substitution à un terme) **:** Étant donné une substitution σ, on définit inductivement la fonction

$$\cdot[\sigma] : \mathcal{T}(\Sigma, \mathcal{V}) \longrightarrow \mathcal{T}(\mathcal{S}, \mathcal{V})$$
$$t \longmapsto t[\sigma]$$

5. FV : *free variable*, variable libre
5. BV : *bound variable*, variable liée

par
- $x[\sigma] = \sigma(x)$ avec $x \in \mathscr{V}$;
- $\big(f(t_1, \ldots, t_n)\big)[\sigma] = f\big(t_1[\sigma], \ldots, t_n[\sigma]\big)$.

Définition (Application d'une substitution à une formule) **:** Étant donné une substitution σ, on définit inductivement l'application de la substitution σ à une formule par

- $\top[\sigma] = \top$;
- $\bot[\sigma] = \bot$;
- $P(t_1, \ldots, t_n)[\sigma] = P\big(t_1[\sigma], \ldots, t_n[\sigma]\big)$;
- $(G \odot H)[\sigma] = G[\sigma] \odot H[\sigma]$ avec $\odot \in \{\vee, \wedge, \to, \leftrightarrow\}$;
- $(\neg G)[\sigma] = \neg\big(G[\sigma]\big)$;
- $(\forall x,\ G)[\sigma] = \forall x,\ G\big[\sigma[x \mapsto x]\big]$
- $(\exists x,\ G)[\sigma] = \exists x,\ G\big[\sigma[x \mapsto x]\big]$

⚠ On s'assurera que les variables apparaissent dans l'espace image de la substitution σ n'intersecte pas avec les variables liées de G lors du calcul de $G[\sigma]$. Ce peut-être assuré au moyen du α-renommage.

Exemple :
On considère la formule $P(x, y) \wedge \big(\forall x,\ Q(x, y)\big)$. On applique la substitution $\sigma : (x \mapsto x + y,\ y \mapsto 0)$.

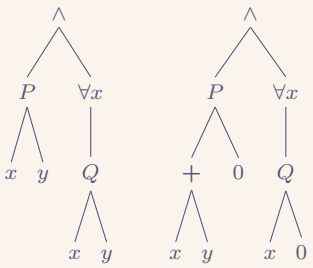
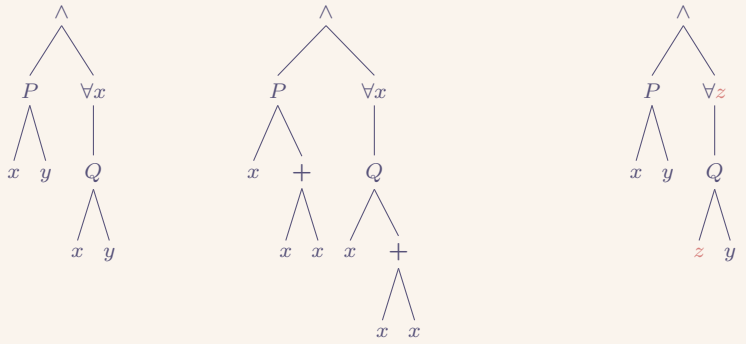


Figure 6.2 – Arbre de syntaxe de la formule $P(x, y) \wedge \big(\forall x,\ Q(x, y)\big)$, application de la substitution $\sigma = (x \mapsto x + y,\ y \mapsto 0)$

Exemple :
On considère la même formule, et la substitution $\sigma : (y \mapsto x + x)$.



formule originale substitution sans α-renommage substitution après α-renommage

FIGURE 6.3 – Arbre de syntaxe de la formule $P(x,y) \wedge (\forall x,\ Q(x,y))$, application de la substitution $\sigma = (y \mapsto x + x)$ directement et avec α-renommage

6.2.3 Extension au premier ordre de la déduction naturelle

On ajoute les règles suivantes.

SYMBOLE	RÈGLE D'INTRODUCTION	RÈGLE D'ÉLIMINATION
$\forall$	$\dfrac{\Gamma \vdash G}{\Gamma \vdash \forall x,\ G}\ \forall\text{i}$ $x \notin \mathsf{FV}(\Gamma)$	$\dfrac{\Gamma \vdash \forall x,\ G}{\Gamma \vdash G\big[(x \mapsto t)\big]}\ \forall\text{e}$ $\mathrm{vars}(t) \cap \mathsf{BV}(G) = \varnothing$
$\exists$	$\dfrac{\Gamma \vdash G\big[(x \mapsto t)\big]}{\Gamma \vdash \exists x,\ G}\ \exists\text{i}$	$\dfrac{\Gamma \vdash \exists x,\ H \qquad \Gamma, H \vdash G}{\Gamma \vdash G}\ \exists\text{e}$ $x \notin \mathsf{FV}(\Gamma) \cup \mathsf{FV}(G)$

TABLE 6.3 – Extension au premier ordre de la déduction naturelle

EXEMPLE :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x,\ P(\mathbf{0},x) \vdash \forall x,\ P(\mathbf{0},x)}\ \text{Ax}}{\forall x,\ P(\mathbf{0},x) \vdash P(\mathbf{0},\mathbf{1})}\ \forall\text{e}}{\forall x,\ P(\mathbf{0},x) \vdash \exists y,\ P(y,\mathbf{1})}\ \exists\text{i}}{\vdash (\forall x,\ P(\mathbf{0},x)) \to (\exists y,\ P(y,\mathbf{1}))}\ \to\text{i}$$

EXEMPLE :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x,\ P(x) \vdash \forall x,\ P(x)}\ \text{Ax}}{\forall x,\ P(x) \vdash P(x)}\ \forall\text{e}}{\forall x,\ P(x) \vdash \exists x,\ P(x)}\ \exists\text{i}}{\vdash (\forall x,\ P(x)) \to (\exists x,\ P(x))}\ \to\text{i}$$

Exemple :
On pose $F_1 = \exists x,\ P(x)$ et $F_2 = \forall x,\ \forall y,\ \big(P(x) \to Q(y)\big)$.

$$\dfrac{\dfrac{}{F_1, F_2 \vdash \exists x,\ P(x)}\ \text{Ax} \qquad \dfrac{\dfrac{\dfrac{}{F_1, F_2, P(x) \vdash \exists y,\ P(x) \to Q(y) \vdash \exists y,\ P(x) \to Q(y)}\ \text{Ax} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, P(x) \to Q(y), P(x) \vdash P(x) \to Q(y)}\ \text{Ax} \qquad \dfrac{}{\Gamma, P(x) \vdash P(x)}\ \text{Ax}}{F_1, F_2, P(x), (\exists y\ P(x) \to Q(y)), P(x) \to Q(y) \vdash Q(y)}\ \to\text{e}}{F_1, F_2, P(x), (\exists y\ P(x) \to Q(y)), P(x) \to Q(y) \vdash \exists y,\ Q(y)}\ \exists\text{i}}{F_1, F_2, P(x), (\exists y\ P(x) \to Q(y)) \vdash \exists z,\ Q(z)}\ \exists\text{e}}{F_1, F_2, P(x) \vdash (\exists y\ P(x) \to Q(y)) \to (\exists z,\ Q(z))}\ \to\text{i} \qquad \dfrac{\dfrac{}{F_1, F_2, P(x) \vdash \forall x,\ \exists y,\ P(x) \to Q(y)}\ \text{Ax}}{F_1, F_2, P(x) \vdash \exists y,\ P(x) \to Q(y)}\ \forall\text{e}}{F_1, F_2, P(x) \vdash \exists z,\ Q(z)}\ \to\text{e}}{\exists x,\ P(x); \forall x,\ \exists y,\ \big(P(x) \to Q(y)\big) \vdash \exists z,\ Q(z)}\ \exists\text{e}$$

Exemple (Paradoxe du buveur) :
On pose $\varphi = \exists x,\ \neg B(x)$.

$$\dfrac{\dfrac{}{\varnothing \vdash \varphi \vee \neg\varphi}\ \text{TE} \qquad \dfrac{\dfrac{}{\varphi \vdash \varphi}\ \text{Ax} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\varphi, \neg B(x), B(x) \vdash B(x)}\ \text{Ax} \qquad \dfrac{}{\varphi, \neg B(x), B(x) \vdash \neg B(x)}\ \text{Ax}}{\varphi, \neg B(x), B(x) \vdash \bot}\ \bot\text{e}}{\varphi, \neg B(x), B(x) \vdash \forall y,\ B(y)}\ \to\text{i}}{\varphi, \neg B(x) \vdash B(x) \to \forall y,\ B(y)}\ \exists\text{i}}{\varphi \vdash \exists x,\ \big(B(x) \to \forall y,\ B(y)\big)}\ \exists\text{e} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(\exists x,\ \neg B(x)), B(x), \neg B(y) \vdash \neg(\exists x,\ \neg B(x))}\ \text{Ax} \qquad \dfrac{\dfrac{}{\neg(\exists x,\ \neg B(x)), B(x), \neg B(y) \vdash \neg B(y)}\ \text{Ax}}{\neg(\exists x,\ \neg B(x)), B(x), \neg B(y) \vdash \exists x,\ \neg B(x)}\ \exists\text{i}}{\neg(\exists x,\ \neg B(x)), B(x), \neg B(y) \vdash \bot}\ \neg\text{e}}{\neg(\exists x,\ \neg B(x)), B(x) \vdash B(y)}\ \text{Abs}}{\neg\varphi, B(x) \vdash \forall y,\ B(y)}\ \forall\text{i}}{\neg\varphi \vdash B(x) \to \forall y,\ B(y)}\ \to\text{i}}{\neg\varphi \vdash \exists x,\ \big(B(x) \to \forall y,\ B(y)\big)}\ \exists\text{i}}{\varnothing \vdash \exists x,\ \big(B(x) \to \forall y,\ B(y)\big)}\ \vee\text{e}$$

Théorème : L'ajout des quatre règles précédentes à la déduction naturelle, intuitionniste ou classique, maintient sa correction. □

Remarque (Hors-Programme) :
L'ajout de ces règles maintient également sa complétude vis-à-vis de la logique classique.

6.2.4 Règles dérivées

On définit de manière informelle la notion de *règle dérivée* comme des règles que l'on peut obtenir comme combinaison des règles déjà existantes.

Exemple :
On considère la règle nommée TE′ définie comme

$$\frac{\Gamma, H \vdash G \qquad \Gamma, \neg H \vdash G}{\Gamma \vdash G}\ \text{TE}'.$$

Elle se dérive des règles TE et $\vee$e :

$$\dfrac{\dfrac{}{\Gamma \vdash H \vee \neg H}\ \text{Ax} \qquad \Gamma, H \vdash G \qquad \Gamma, \neg H \vdash G}{\Gamma \vdash G}\ \vee\text{e}.$$

Remarque :
Si $\Gamma \vdash G$ est prouvable, et si $\Gamma \subseteq \Gamma'$, alors $\Gamma' \vdash G$ est prouvable. On ajoute donc parfois une règle dit d'*affaiblissement*, définie comme

$$\frac{\Gamma' \vdash G}{\Gamma \vdash G}\ \text{Aff} \qquad \Gamma' \subseteq \Gamma.$$

6.2.5 Sémantique

On considère la formule défini par l'arbre de syntaxe suivant. On a $\mathcal{P} = \{P(1), Q(1)\}$, $\mathcal{S} = \{\oplus(2), \tilde{0}(0), \tilde{1}(0), \ominus(3)\}$ et $\mathcal{V} \supseteq \{x, y, z\}$.

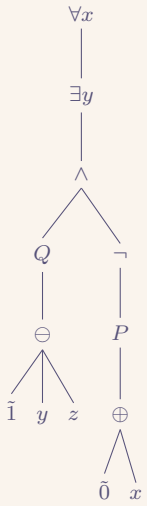


FIGURE 6.4 – Arbre de syntaxe exemple

Pour interpréter une formule de la logique du premier ordre, on doit définir le "monde" des variables, leur valeur, la valeur d'une constante et d'une fonction, la valeur des prédicats, et le sens des quantificateurs.

Définition (Domaine) **:** On appelle *domaine d'interprétation* des termes un ensemble non vide $\mathbf{M}$.

EXEMPLE :
On peut choisir $\mathbf{M} = \mathbb{R}$, ou $\mathbf{M} = \mathbb{N}$, $\mathbf{M} = \mathbb{Z}$, $\mathbf{M} = \mathbb{B}$ ou $\mathbf{M} = \mathcal{T}(\mathcal{S}, \mathcal{V})$.

Dans toute la suite de cette section, on fixe les ensembles $\mathcal{S}$, $\mathcal{V}$ et $\mathcal{P}$.

Définition (Environnement de variables) **:** On appelle *environnement de variable* sur $V \subseteq \mathcal{V}$ une fonction

$$\mu : V \longrightarrow \mathbf{M}.$$

EXEMPLE :
On a par exemple $\mu = (x \mapsto 3)$.

Définition (Structure d'interprétation) **:** On appelle *structure d'interprétation* la donnée de

- un domaine $\mathbf{M}$;
- une fonction $f^M : \mathbf{M}^{\mathfrak{a}(f)} \longrightarrow \mathbf{M}$ pour chaque symbole $f \in \mathcal{S}$;
- une fonction $P^M : \mathbf{M}^{\mathfrak{a}(P)} \longrightarrow \mathbb{B}$ pour chaque symbole $P \in \mathcal{P}$.

On typographie une telle structure M.

Exemple :
Si $\mathcal{S} = \{\oplus(2), \ominus(2)\}$, $\mathcal{P} = \{\lessdot(2), \eqslantless(2)\}$, et $\mathbf{M} = \mathbb{N}$, alors on définit les fonctions

$$\oplus^M : \mathbb{N}^2 \longrightarrow \mathbb{N} \qquad (m,n) \longmapsto m+n,$$

$$\lessdot^M : \mathbb{N}^2 \longrightarrow \mathbb{B} \qquad (n,m) \longmapsto \begin{cases} \boldsymbol{V} & \text{si } n < m \\ \boldsymbol{F} & \text{sinon,} \end{cases}$$

et

$$\ominus^M : \mathbb{N}^2 \longrightarrow \mathbb{N} \qquad (n,m) \longmapsto \begin{cases} n-m & \text{si } n \geqslant m \\ 0 & \text{sinon,} \end{cases}$$

$$\eqslantless^M : \mathbb{N}^2 \longrightarrow \mathbb{B} \qquad (n,m) \longmapsto \begin{cases} \boldsymbol{V} & \text{si } n = m \\ \boldsymbol{F} & \text{sinon.} \end{cases}$$

Définition : On définit la fonction `eval` prenant en argument
- un terme t,
- une structure d'interprétation,
- un environnement sur au moins les variables de t,

et s'évaluant dans $\mathbf{M}$, telle que $\texttt{eval}(x, M, \mu) = \mu(x)$ avec $x \in \mathcal{V}$, et que

$$\begin{aligned} &\texttt{eval}\big(f(t_1, t_2, \ldots, t_{\mathfrak{a}(f)}), M, \mu\big) \\ =&f^M\big(\texttt{eval}(t_1, M, \mu), \texttt{eval}(t_2, M, \mu), \ldots, \texttt{eval}(t_{\mathfrak{a}(f)}, M, \mu)\big) \end{aligned}$$

Exemple :
Avec la structure précédente, on a

$$\begin{aligned} \texttt{eval}\Big(\oplus\big(x, \ominus(x,y)\big), M, (x \mapsto 1, y \mapsto 2)\Big) &= \oplus^M\Big(\mu(x), \ominus^M\big(\mu(x), \mu(y)\big)\Big) \\ &= 1 + \ominus^M(1,2) \\ &= 1 + 0 = 1 \end{aligned}$$

Définition (Interprétation des formules) **:** On définit inductivement $[\![\cdot]\!]^{M,\mu}$ comme
- $[\![\top]\!]^{M,\mu} = \boldsymbol{V}$;
- $[\![\bot]\!]^{M,\mu} = \boldsymbol{F}$;
- $[\![\neg G]\!]^{M,\mu} = \overline{[\![G]\!]^{M,\mu}}$;
- $[\![G \wedge H]\!]^{M,\mu} = [\![G]\!]^{M,\mu} \cdot [\![H]\!]^{M,\mu}$;
- $[\![G \vee H]\!]^{M,\mu} = [\![G]\!]^{M,\mu} + [\![H]\!]^{M,\mu}$;
- $[\![G \to H]\!]^{M,\mu} = \overline{[\![G]\!]^{M,\mu}} + [\![H]\!]^{M,\mu}$;
- $[\![G \leftrightarrow H]\!]^{M,\mu} = \Big(\overline{[\![G]\!]^{M,\mu}} + [\![H]\!]^{M,\mu}\Big) \cdot \Big(\overline{[\![H]\!]^{M,\mu}} + [\![G]\!]^{M,\mu}\Big)$;
- $[\![P(t_1, \ldots, t_{\mathfrak{a}(P)})]\!]^{M,\mu} = P^M\big(\texttt{eval}(t_1, M, \mu), \ldots, \texttt{eval}(t_{\mathfrak{a}(P)}, M, \mu)\big)$;
- $[\![\exists x,\, G]\!]^{M,\mu} = \underset{v_x \in \mathbf{M}}{+} [\![G]\!]^{M, \mu[x \mapsto v_x]}$;
- $[\![\forall x,\, G]\!]^{M,\mu} = \underset{v_x \in \mathbf{M}}{\bullet} [\![G]\!]^{M, \mu[x \mapsto v_x]}$,

où on définit $+\,\mathcal{B} = \boldsymbol{V} \iff \boldsymbol{V} \in \mathcal{B}$, et $\bullet\, \mathcal{B} = \boldsymbol{F} \iff \boldsymbol{F} \in \mathcal{B}$ avec $\mathcal{B} \subseteq \{\boldsymbol{V}, \boldsymbol{F}\}$.

Exemple :
On pose $\mathcal{S} = \{\oplus(2), Z(0)\}$, $\mathcal{P} = \{\leqslant(2), \ominus(2)\}$, et

$$G = \forall x,\ \Big(\exists y,\ \Big(\exists z, \ominus(x, \oplus(y, z)) \wedge \neg\ominus(z, Z)\Big)\Big).$$

On considère la structure M définie comme donnée de $\mathbf{M} = \mathbb{N}$, $\oplus^M = +$, $Z^M = 0$, $\ominus^M = \leqslant$, et $\leqslant^M =$ " $=$ ". Ainsi,

$$[\![G]\!]^{M,(\,)} = \bullet_{v_x \in \mathbb{N}} \Big(+_{v_y \in \mathbb{N}} \Big(+_{v_z \in \mathbb{N}} \mathbb{1}_{v_x = v_y + v_z} \cdot \overline{\mathbb{1}_{v_z = 0}} \Big) \Big),$$

où l'on définit $\mathbb{1}_G$ comme $\boldsymbol{V}$ si G est vrai, et $\boldsymbol{F}$ sinon. Pour $v_x = 0$, alors, pour tous $v_y \in \mathbb{N}$ et $v_z \in \mathbb{N}$, on a $\mathbb{1}_{v_x = v_y + v_z} \cdot \overline{\mathbb{1}_{v_z = 0}} = \boldsymbol{F}$. Ainsi, $[\![G]\!]^{M,\mu} = \boldsymbol{F}$. À faire : cas où $\mathbf{M} = \mathbb{Z}$

Définition : Une formule G de la logique du premier ordre est dite *satisfiable* dès lors qu'il existe une structure M, et un environnement de variables μ tel que $[\![G]\!]^{M,\mu} = \boldsymbol{V}$.

Une structure M est dit *modèle* de G dès lors que, pour tout environnement de variables μ, on a $[\![G]\!]^{M,\mu} = \boldsymbol{V}$.

Une formule G de la logique du premier ordre est dite *valide* dès lors que pour toute structure M, et tout environnement de variables μ, on a $[\![G]\!]^{M,\mu} = \boldsymbol{V}$.

Étant donné deux formules G et H, on dit que H est *conséquence sémantique* de G dès lors que, pour toute structure M et environnement de variables μ, si $[\![G]\!]^{M,\mu} = \boldsymbol{V}$, alors $[\![H]\!]^{M,\mu} = \boldsymbol{V}$. On le note $G \models H$.

On dit que deux formules G et H sont *équivalentes* dès lors que, $G \models H$ et $H \models G$. On le note $G \equiv H$.

Remarque :
— Une formule est dit *close* dès lors que $\mathsf{FV}(H) = \varnothing$.
— Une formule de ma forme $P(t_1, t_2, \ldots, t_{\mathfrak{a}(P)})$ est appelée *formule atomique* ou *prédicat atomique*.
— Si $\mathsf{FV}(G) = \{x_1, \ldots, x_n\}$, la formule $\forall x_1, \ldots, \forall x_n,\ G$ est appelée *cloture universelle* de G. La formule $\exists x_1, \ldots, \exists x_n,\ G$ est appelée *cloture existentielle* de G.

6.3 Synthèse du chapitre

Symbole	Règle d'introduction	Règle d'élimination
$\top$	$\dfrac{}{\Gamma \vdash \top}\ \top\text{i}$	
$\bot$		$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash G}\ \bot\text{e}$
$\neg$	$\dfrac{\Gamma, G \vdash \bot}{\Gamma \vdash \neg G}\ \neg\text{i}$	$\dfrac{\Gamma \vdash G \quad \Gamma \vdash \neg G}{\Gamma \vdash \bot}\ \neg\text{e}$
$\to$	$\dfrac{\Gamma, G \vdash H}{\Gamma \vdash G \to H}\ \to\text{i}$	$\dfrac{\Gamma \vdash H \to G \quad \Gamma \vdash H}{\Gamma \vdash G}\ \to\text{e}$
$\wedge$	$\dfrac{\Gamma \vdash G \quad \Gamma \vdash H}{\Gamma \vdash G \wedge H}\ \wedge\text{i}$	$\dfrac{\Gamma \vdash G \wedge H}{\Gamma \vdash G}\ \wedge\text{e,g} \qquad \dfrac{\Gamma \vdash G \wedge H}{\Gamma \vdash H}\ \wedge\text{e,d}$
$\vee$	$\dfrac{\Gamma \vdash G}{\Gamma \vdash G \vee H}\ \vee\text{i,g} \qquad \dfrac{\Gamma \vdash H}{\Gamma \vdash G \vee H}\ \vee\text{i,d}$	$\dfrac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash G \quad \Gamma, B \vdash G}{\Gamma \vdash G}\ \vee\text{e}$
	$\dfrac{}{\Gamma, \varphi \vdash \varphi}\ \text{Ax}$	

Table 6.4 – Règles d'introduction et d'élimination

$\dfrac{}{\Gamma \vdash G \vee \neg G}\ \text{TE}$	$\dfrac{\Gamma \vdash \neg\neg G}{\Gamma \vdash G}\ \neg\neg\text{e}$	$\dfrac{\Gamma, \neg G \vdash \bot}{\Gamma \vdash G}\ \text{Abs}$

Table 6.5 – Déduction naturelle classique

Symbole	Règle d'introduction	Règle d'élimination
$\forall$	$\dfrac{\Gamma \vdash G}{\Gamma \vdash \forall x,\ G}\ \forall\text{i}$ $x \notin \mathsf{FV}(\Gamma)$	$\dfrac{\Gamma \vdash \forall x,\ G}{\Gamma \vdash G\big[(x \mapsto t)\big]}\ \forall\text{e}$ $\mathrm{vars}(t) \cap \mathsf{BV}(G) = \varnothing$
$\exists$	$\dfrac{\Gamma \vdash G\big[(x \mapsto t)\big]}{\Gamma \vdash \exists x,\ G}\ \exists\text{i}$	$\dfrac{\Gamma \vdash \exists x,\ H \quad \Gamma, H \vdash G}{\Gamma \vdash G}\ \exists\text{e}$ $x \notin \mathsf{FV}(\Gamma) \cup \mathsf{FV}(G)$

Table 6.6 – Extension au premier ordre de la déduction naturelle

CHAPITRE

7

TENTATIVE DE RÉPONSE À LA **NP**-COMPLÉTUDE

Sommaire

7.0 Motivation **161**
7.1 Problèmes d'optimisation **161**
7.2 Algorithmes d'approximations **163**
7.3 *Branch and Bound* — Séparation et évaluation **167**
Annexe 7.A Programmation dynamique **169**

7.0 Motivation

On considère le problème **NP**-complet du *voyageur de commerce* : étant donné un graphe pondéré G quel est le tour de longueur [1] minimale *i.e.* quelle est la permutation de sommets telle que la longueur totale est minimale.

On se ramène à un problème de décision : étant donné une constante $K \in \mathbb{R}$, existe-t-il un chemin de longueur inférieure à K.

Un algorithme glouton, allant d'un sommet à son voisin le plus proche, ne permet pas de résoudre ce problème en complexité polynômiale.

On ne cherche plus le « tour optimal » mais on cherche une solution proche : on veut trouver une constante ρ telle que, quelque soit l'entrée, le chemin obtenu est de longueur inférieure à ρ fois la longueur optimale.

7.1 Problèmes d'optimisation

Dans un premier temps, on s'intéresse à un problème où l'on cherche à minimiser quelque chose. On réalise la transformation réalisée dans la partie précédente : étant donné un seuil K, on est ce que la valeur est inférieure à K. On se ramène donc à un problème de décision.

1. poids des arrêtes total

Définition : Soit $Q \subseteq \mathscr{E} \times \mathcal{S}$ un problème. Soit $\mathrm{opt} \in \{\min, \max\}$. On dit que Q est un *problème d'*opt*imisation* (*i.e.* problème de minimisation, maximisation), si pour toute entrée $e \in \mathscr{E}$, il existe

- un ensemble $\mathrm{sol}(e)$ de solutions,
- une fonction $c_e : \mathrm{sol}(e) \to \mathbb{R}^+$,

tels que $c_e^\star = \mathrm{opt}\{c_e(s) \mid s \in \mathrm{sol}(e)\}$ est bien défini, et

$$\forall s \in \mathrm{sol}(e), \qquad (e, s) \in Q \implies c_e(s) = c_e^\star.$$

On nomme :

- $\mathrm{sol}(e)$ l'*ensemble des solutions pour l'entrée* e,
- c_e *la fonction objectif*,
- $c_e^\star$ la *valeur* opt*imale* (minimale ou maximale),
- pour une solution $s \in \mathrm{sol}(e)$, $c_e(s)$ est appelée la *valeur de la solution*.

On appelle *solution* opt*imale* une solution de valeur optimale.

Exemple :
On considère le problème

$$\begin{cases} \textbf{Entrée} & : G = (S, A) \text{ un graphe orienté fortement connexe, } s \in S, \text{ et } p \in S \\ \textbf{Sortie} & : \text{un plus court chemin (en nombre d'arcs) de } s \text{ à } p \text{ dans } G. \end{cases}$$

Soit l'entrée ci-dessous.

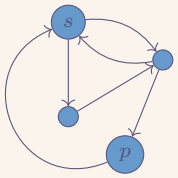


Figure 7.1 – Entrée du problème du plus court chemin

L'ensemble $\mathrm{sol}(e)$ est l'ensemble (infini) des chemins de s à p, et $c_e(\gamma) = |\gamma|$ (la longueur du chemin γ). On vérifie qu'il existe un chemin de s à p, donc $\{c_e(s) \mid s \in \mathrm{sol}(e)\}$ est une partie de $\mathbb{N}$ non vide, elle admet donc un minimum.

Définition : Le *problème de décision associé* à un problème d'optimisation est le problème obtenu en ajoutant une constante aux entrées et en demandant en sortie s'il est possible de dépasser cette constante.

Exemple :
Étant donné le problème d'optimisation

$$Q_{\mathrm{O}} : \begin{cases} \textbf{Entrée} & : e \in \mathscr{E}_{Q_{\mathrm{O}}} \\ \textbf{Sortie} & : \arg\mathrm{opt}_{s \in \mathrm{sol}(e)}\, c_e(s), \end{cases}$$

on définit le problème de décision associé

$$Q : \begin{cases} \textbf{Entrée} & : e \in \mathscr{C}_{Q_O},\ K \in \mathbb{R}^+ \\ \textbf{Sortie} & : \text{existe-t-il } s \in \mathrm{sol}(e) \text{ tel que } c_e(s) \bowtie K \end{cases}$$

avec $\bowtie = \leqslant$ si $\mathrm{opt} = \min$ et $\bowtie = \geqslant$ si $\mathrm{opt} = \max$.

Exemple :
Avec l'exemple précédent (plus court chemin), le problème de décision associé est

$$\begin{cases} \textbf{Entrée} & : G = (S, A) \text{ un graphe orienté fortement connexe, } (p, s) \in S^2, \text{ et } K \in \mathbb{R}^+ \\ \textbf{Sortie} & : \text{Existe-t-il un chemin de } s \text{ à } p \text{ dans } G \text{ de longueur inférieure ou égale à } K \,? \end{cases}$$

Exemple :
On considère le problème Knapsack de décision défini comme

$$\begin{cases} \textbf{Entrée} & : \text{Un entier } n \in \mathbb{N},\ (p_1, \dots, p_n) \in (\mathbb{N}^\star)^n,\ (v_1, \dots, v_n) \in (\mathbb{N}^\star)^n,\ P \in \mathbb{N} \text{ et } K \in \mathbb{N} \\ \textbf{Sortie} & : \text{Existe-t-il } I \subseteq [\![1, n]\!] \text{ telle que } \sum_{i \in I} p_i \leqslant P \text{ et } \sum_{i \in I} v_i \geqslant K \,? \end{cases}$$

Le problème d'optimisation associé est Knapsack$_O$ défini comme

$$\begin{cases} \textbf{Entrée} & : \text{Un entier } n \in \mathbb{N},\ (p_1, \dots, p_n) \in (\mathbb{N}^\star)^n,\ (v_1, \dots, v_n) \in (\mathbb{N}^\star)^n, \text{ et } P \in \mathbb{N} \\ \textbf{Sortie} & : I \subseteq [\![1, n]\!] \text{ tel que } \sum_{i \in I} p_i \leqslant P \text{ et maximisant } \sum_{i \in I} v_i. \end{cases}$$

Remarque :
Soit Q_O un problème d'optimisation et Q le problème de décision associé. Étant donné un algorithme $\mathscr{A}_O$ pour Q_O, on fabrique l'algorithme $\mathscr{A}$ suivant résolvant Q.

Algorithme 7.1 Solution à un problème de seuil

Entrée e une entrée de Q_O et K un seuil

1 : **retourner** $c_e(\mathscr{A}_O) \overset{?}{\bowtie} K$ ▷ *où* $\bowtie$ *est* $\geqslant$ *pour si* opt *est* max, *et* $\leqslant$ *si* opt *est* min

Ainsi, le problème Q_O est plus difficile à résoudre que le problème Q de décision associé. Alors, lorsque le problème de décision Q associé à un problème d'optimisation S_O est **NP**-difficile, c'est mal engagé.

7.2 Algorithmes d'approximations

Remarque (Vocabulaire) :
On fixe dans la suite un problème d'optimisation Q, on note $\mathrm{OPT}(e)$ la valeur optimale pour une entrée e.

Définition (Algorithme d'approximation pour un problème de maximisation) **:** On dit d'un algorithme $\mathscr{A} : \mathscr{C}_Q \to \mathbb{R}^+$ qu'il approxime un problème Q de maximisation avec un ratio d'approximation $\rho < 1$ dès lors que

$$\forall e \in \mathscr{C}_Q, \quad \mathscr{A}(e) \geqslant \rho \cdot \mathrm{OPT}(e).$$

On dit alors que l'algorithme $\mathscr{A}$ est une *ρ-approximation* (standard).

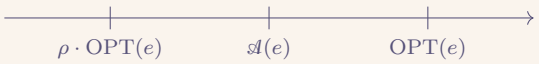


FIGURE 7.2 – Algorithme d'approximation pour un problème de maximisation

Définition (Algorithme d'approximation pour un problème de minimisation) **:** On dit d'un algorithme $\mathscr{A} : \mathscr{C}_Q \to \mathbb{R}^+$ qu'il approxime un problème Q de minimisation avec un ratio d'approximation $\rho > 1$ dès lors que

$$\forall e \in \mathscr{C}_Q, \quad \mathscr{A}(e) \leqslant \rho \cdot \mathrm{OPT}(e).$$

On dit alors que l'algorithme $\mathscr{A}$ est une *ρ-approximation* (standard).

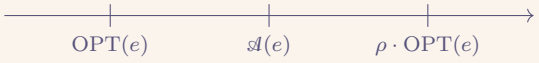


FIGURE 7.3 – Algorithme d'approximation pour un problème de minimisation

Dans la définition suivante, on suppose connu une fonction $\mathrm{Pire}(e)$ donnant la pire valeur de solution pour une entrée e.

Définition : On dit qu'un algorithme $\mathscr{A} : \mathscr{C}_Q \to \mathbb{R}$ est une *ρ-approximation différentielle* dès lors que

$$\frac{|\mathscr{A}(e) - \mathrm{Pire}(e)|}{|\mathrm{Pire}(e) - \mathrm{OPT}(e)|} \geqslant \rho.$$

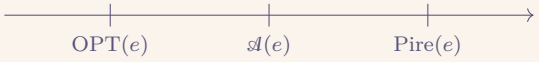


FIGURE 7.4 – ρ-approximation différentielle

REMARQUE :
Dans le cadre d'un problème de minimisation, on ne calcule pas $\mathrm{OPT}(e)$ en général. On minore $\mathrm{OPT}(e)$, et alors

$$\frac{\mathscr{A}(e)}{\mathrm{OPT}(e)} \leqslant \underbrace{\frac{\mathscr{A}(e)}{B}}_{\rho}.$$

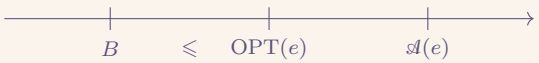


FIGURE 7.5 – Calcul de $\mathrm{OPT}(e)$

EXEMPLE :
Soit $C \in \mathbb{N}^\star$. On rappelle le problème STABLE, restreint à un graphe de degré maximal

C :

$$\textsc{Stable} : \begin{cases} \textbf{Entrée} & : G = (S, A) \text{ une graphe tel que } 0 \neq \Delta(G) \leqslant C \\ \textbf{Sortie} & : \text{ un stable de } G \text{ de cardinal maximal} \end{cases}$$

où $X \subseteq S$ est un stable si pour tout $(u, v) \in X^2$, $\{u, v\} \notin A$, et $\Delta(G) = \max_{v \in S} \deg_G(v)$ est le degré du graphe.

Algorithme 7.2 Algorithme glouton de recherche de stables

Entrée $G = (S, A)$ un graphe
1 : $S' \leftarrow \varnothing$
2 : **tant que** $S \neq \varnothing$ **faire**
3 : $\quad v^\star = \arg\min_{v \in S} \deg_G(v)$ ▷ *les degrés sont modifiés à chaque itération*
4 : $\quad S' \leftarrow S' \cup \{v^\star\}$
5 : $\quad S \leftarrow S \setminus \big(\{v^\star\} \cup \text{voisins}(v^\star)\big)$
6 : $\quad A \leftarrow$ restriction de A à S
7 : **retourner** S'

Cet algorithme n'est pas correct, la figure ci-après en est un contre-exemple.

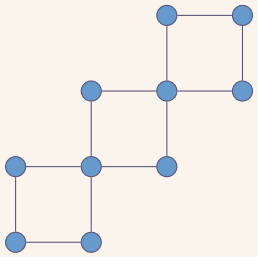


Figure 7.6 – Contre-exemple à l'algorithme 7.2

Propriété : L'algorithme 7.2 est une $\frac{1}{\Delta(G)}$-approximation.

Preuve :
Soit S' la réponse de l'algorithme. Soit $S^\star$ la solution optimale. On a, par terminaison de l'algorithme

$$\forall v \in S \setminus S, \exists v' \in S',\ \{v, v'\} \in A$$

car l'algorithme s'arrête. En particulier, $\forall v^\star \in S^\star \setminus S',\ \exists v' \in S',\ \{v, v'\} \in A$. Or, $S^\star$ est stable donc, si $v^\star \in S^\star$ et $v' \in S'$ tels que $\{v^\star, v' \in A\}$, alors $v' \notin S^\star$. D'où,

$$\forall v^\star \in S^\star \setminus S',\ \exists v' \in S' \setminus S^\star,\ \{v^\star, v'\} \in A.$$

Par définition de degré d'un graphe, on a $|S^\star \setminus S'| \leqslant \Delta(G)\,|S' \setminus S^\star|$ donc

$$\begin{aligned} |S^\star| &= |S^\star \cap S'| + |S^\star \setminus S'| \\ &\leqslant |S^\star \cap S'| + \Delta(G)\,|S' \setminus S^\star| \\ &\leqslant \Delta(G)\,|S^\star \cap S'| + \Delta(G)\,|S' \setminus S^\star| \\ &\leqslant \Delta(G)\,|S'|. \end{aligned}$$

□

Remarque :
Cette preuve ne fait pas d'hypothèses sur le résultat de l'algorithme. Tout algorithme répondant au problème est une $\frac{1}{\Delta(G)}$-approximation.

Exemple :
On appelle *couverture par sommets* d'un graphe $G = (S, A)$ la donnée d'un ensemble $X \subseteq S$ tel que

$$\forall \{u, v\} \in A, \quad u \in X \text{ ou } v \in X.$$

Exemple :
L'ensemble ● est une couverture par sommets du graphe ci-dessous.

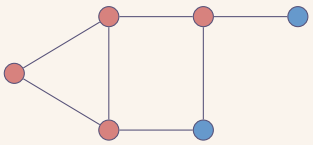


Figure 7.7 – Exemple de couverture par sommets

On considère le problème

$$\begin{cases} \textbf{Entrée} & : G = (S, A) \text{ un graphe} \\ \textbf{Sortie} & : \text{une couverture de cardinal maximal.} \end{cases}$$

Ce problème peut-être résolu à l'aide du calcul de couplages maximal.

Algorithme 7.3 Calcul d'un couplage maximal (CouplageMaximal)

Entrée $G = (S, A)$ un graphe
Sortie C un couplage maximal, *non nécessairement maximum*

```
C ← ∅
tant que ∃{u, v} ∈ A, u libre dans C et v libre dans C faire
    Soit {u, v} une telle arrête.
    C ← C ∪ {{u, v}}
retourner C
```

L'algorithme retourne un couplage maximal d'après la négation de la condition de boucle. On répond donc au problème avec l'algorithme ci-dessous.

Algorithme 7.4 Approximation de couverture par sommets

Entrée $G = (S, A)$ un graphe
Sortie Une couverture par sommets

```
C ← CouplaxeMaximal(G)
retourner {u ∈ S | ∃v ∈ S, {u, v} ∈ C}.
```

L'algorithme retourne une couverture : soit X la valeur retournée pour une entrée $G = (S, A)$. Soit $\{u, v\} \in A$. Si $u \notin X$ et $v \notin X$, alors le couplage C calculé pour l'algorithme n'est pas maximal, on peut y ajouter $\{u, v\}$. Montrons que l'algorithme $\mathscr{A}$ est une 2-approximation du problème « couverture par sommets. »

$$\forall G \in \mathscr{C}, \quad \mathscr{A}(G) \leqslant 2\,\mathrm{OPT}(G).$$

Soit $G \in \mathscr{C}$. Soit X la couverture par sommets calculé par $\mathscr{A}$ sur G, et C le couplage calculé par cet algorithme. Soit $X^\star$ la couverture par sommets optimale. Soit donc

$$\varphi : \quad C \longrightarrow X^\star$$
$$\{u, v\} \longmapsto \begin{cases} u & \text{si } u \in X^\star \\ v & \text{si } v \in X^\star \end{cases}$$

Soit $(c_1, c_2) \in C^2$, tels que $\varphi(c_1) = \varphi(c_2)$, alors c_1 et c_2 partagent un sommet, ce qui est absurde (*c.f.* définition de couplage). Donc φ est injective, d'où $|C| \leqslant |X^\star|$. Or, $\mathscr{A}(X) = |X| = 2|C| \leqslant 2\,|X^\star| \leqslant 2\,\mathrm{OPT}(X)$.

7.3 *Branch and Bound* — Séparation et évaluation

Branch and Bound n'est pas un algorithme, mais une famille d'algorithmes, similairement au algorithmes diviser pour régner. Ces algorithmes répondent à des problèmes de maximisation. Les algorithmes *Branch and Bound* sont des algorithmes enrichit de trois fonctions :

- une fonction `branch` de branchement, *i.e.* découpage en sous-problèmes,
- une fonction `valeur` donnant un résultat, pas forcément optimal, *i.e.* elle associe une solution partielle à une solution,
- une fonction `bound` donnant un majorant de la solution optimale, complétant cette solution partielle.

Avec les deux dernières fonctions, on borne la valeur de la solution optimale.

Exemple (PL et PLNE) :
c.f. DM4.

Exemple :
On considère le problème Knapsack :

$$\begin{cases}\textbf{Entrée} & : n \in \mathbb{N},\ (v_i)_{i\in[\![1,n]\!]} \in (\mathbb{N}^\star)^n,\ (w_i)_{i\in[\![1,n]\!]} \in (\mathbb{N}^\star)^n,\ \text{et } P \in \mathbb{N}^\star \\ \textbf{Sortie} & : \text{une allocation } I \text{ maximale d'objets de somme de poids } (w_i)_{i\in I} \text{ inférieure ou égale à } P.\end{cases}$$

Dans toute la suite de l'exemple, les $(v_i)_{i\in[\![1,n]\!]}$ et $(w_i)_{i\in[\![1,n]\!]}$ sont triés par v_i/w_i décroissants.

Tentative 1. Algorithme glouton.

Algorithme 7.5 Algorithme glouton $\mathscr{G}^{\mathbb{N}}$ répondant au problème Knapsack

```
1 : I ← ∅
2 : S ← 0
3 : pour i ∈ [[1, n]] faire
4 :     si S + w_i ⩽ P alors
5 :         I ← I ∪ {i}
6 :         S ← S + w_i
7 : retourner I
```

Cet algorithme ne donne pas toujours une solution optimale, voici un contre-exemple : $P = 3$, $(v_i) = (1, 2)$ et $(w_i) = (1, 3)$. L'algorithme renvoie $\mathscr{G}^{\mathbb{N}}(E_1) = 1$ mais la solution optimale $\mathrm{OPT}(E_1) = 2$, pour l'entrée E_1. On peut compléter une solution partielle à l'aide de cet algorithme, on a donc défini la fonction `valeur`.

Tentative 2. On résout le problème associé dans $\mathbb{R}$, définit ci-dessous :

$$\text{Knapsack}_{\mathbb{R}} \begin{cases}\textbf{Entrée} & : n \in \mathbb{N},\ (v_i)_{i\in[\![1,n]\!]} \in (\mathbb{N}^\star)^n,\ (w_i)_{i\in[\![1,n]\!]} \in (\mathbb{N}^\star)^n,\ \text{et } P \in \mathbb{N}^\star \\ \textbf{Sortie} & : \arg\max_{x\in S} \sum_{i=1}^n x_i v_i\end{cases}$$

où $S = \left\{(x_i)_{i\in[\![1,n]\!]} \in [0,1]^n \;\middle|\; \sum_{i=1}^n x_i w_i \leqslant P\right\}$. On résout ce problème à l'aide d'un algorithme glouton.

Algorithme 7.6 Algorithme glouton $\mathcal{G}^{\mathbb{R}}$ répondant au problème $\textsc{Knapsack}_{\mathbb{R}}$

```
S ← 0
i ← 1
x ← (0)_{i∈⟦1,n⟧}
tant que i ⩽ n et S + w_i < P faire
    S ← S + w_i
    x_i ← 1
    i ← i + 1
si i ⩽ n alors
    x_i ← (P − S)/2
    S ← P
retourner x
```

En notant $\mathrm{OPT}(e)$ une solution optimale à Knapsack, et $\mathrm{OPT}^{\mathbb{R}}(e)$ une solution optimale à $\textsc{Knapsack}_{\mathbb{R}}$, on a $\forall e \in \mathscr{E}, \quad \mathrm{OPT}(e) \leqslant \mathrm{OPT}^{\mathbb{R}}(e)$. De plus, à faire à la maison, le glouton $\mathcal{G}^{\mathbb{R}}$ donne la solution optimale : on a $\mathrm{OPT}^{\mathbb{R}}(e) = \mathcal{G}^{\mathbb{R}}(e)$.

On branche sur la partie fractionnaire. On considère l'entrée définie dans la table ci-après, avec $P = 20$.

i	1	2	3	4	5	6
v_i	13	16	19	24	3	5
w_i	6	8	10	14	2	5
$\approx v_i/w_i$	2,2	2	1,9	1,7	1,5	1

Table 7.1 – Entrée du problème Knapsack

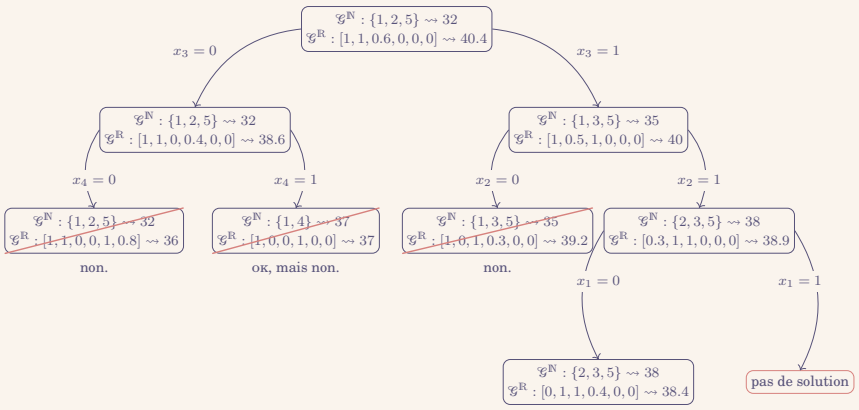


Figure 7.8 – Stratégie *branch and bound* appliquée au problème Knapsack

Annexe 7.A Programmation dynamique

On rappelle le problème KNAPSACK :

$$\begin{cases}\textbf{Entrée} & : n \in \mathbb{N},\ w \in (\mathbb{N}^\star)^n,\ v \in (\mathbb{N}^\star)^n,\ P \in \mathbb{N} \\ \textbf{Sortie} & : \max_{x\in\{0,1\}^n} \{\, \langle x, v\rangle \mid \langle x, w\rangle \leqslant P\}.\end{cases}$$

On pose

$$\textsc{sad}(n, w, v, P) = \max_{x\in\{0,1\}^n} \{\, \langle x, v\rangle \mid \langle x, w\rangle \leqslant P\},$$

et

$$\mathrm{sol}(n, w, v, P) = \{\langle x, v\rangle \mid \langle x, w\rangle \leqslant P,\ x \in \{0,1\}^n\}.$$

Lorsque $y \in \mathbb{R}^n$, avec $y = (y_1, \dots, y_n)$, on note $\mathbb{R}^{n-1} \ni \tilde{y} = (y_2, y_3, \dots, 0)$. Ainsi, si $n > 0$,

$$\begin{aligned}\mathrm{sol}(n, w, v, P) &= \{\langle x, v\rangle \mid \langle x, w\rangle \leqslant P,\ x \in \{0,1\}^n \text{ et } x_1 = 0\} \\ &\quad \cup \{\langle x, v\rangle \mid \langle x, w\rangle \leqslant P,\ x \in \{0,1\}^n \text{ et } x_1 = 1\} \\ &= \{\langle \tilde{x}, \tilde{v}\rangle \mid \langle \tilde{x}, \tilde{w}\rangle \leqslant P,\ xu \in \{0,1\}^n \text{ et } x_1 = 0\} \\ &\quad \cup \{v_1 + \langle \tilde{x}, \tilde{v}\rangle \mid \langle \tilde{x}, \tilde{w}\rangle \leqslant P - w_1,\ x \in \{0,1\}^n \text{ et } x_1 = 1\} \\ &= \{\langle y, \tilde{v}\rangle \mid \langle y, \tilde{w}\rangle \leqslant P \text{ et } y \in \{0,1\}^{n-1}\} \\ &\quad \cup \{v_1 + \langle y, \tilde{v}\rangle \mid \langle y, \tilde{w}\rangle \leqslant P - w_1 \text{ et } y \in \{0,1\}^{n-1}\}\end{aligned}$$

D'où, par passage au max, si $n > 0$,

$$\begin{aligned}\textsc{sad}(n, w, v, P) = \max(&\\ &\max\{\langle y \mid \tilde{v}\rangle \mid \langle y, \tilde{w}\rangle \leqslant P \text{ et } y \in \{0,1\}^{n-1}\} \\ &v_1 + \max\{\langle y \mid \tilde{v}\rangle \mid \langle y, \tilde{w}\rangle \leqslant P - w_1 \text{ et } y \in \{0,1\}^{n-1}\} \\ ) &= \max\big(\textsc{sad}(n-1, \tilde{w}, \tilde{v}, P), v_1 + \textsc{sad}(n-1, \tilde{w}, \tilde{v}, P - w_1)\big).\end{aligned}$$

Si $n = 0$, alors $\textsc{sad}(0, v, w, P) = 0$.

> REMARQUE :
> Si on le code *tel quel*, il y aura $\mathcal{O}(2^n)$ appels récursifs. Mais, on a $(n+1)(P+1)$ sous-problèmes.

Notons alors, pour n, v, w, P fixés, $(s_{i,j})_{\substack{i\in[\![1,n]\!]\\ j\in[\![0,P]\!]}}$ tel que

$$s_{i,j} = \textsc{sad}\Big(n-i, v_{|[\![i+1,n]\!]}, w_{|[\![i+1,n]\!]}, j\Big).$$

On a alors $\textsc{sad}(n, v, w, P) = s_{0,P}$. Ainsi, pour $j \in [\![0, P]\!]$, $s_{n,j} = 0$; pour $i \in [\![0, n]\!]$, $s_{i,0} = 0$;

$$s_{i,j} = \max(s_{i+1,j}, v_{i+1} + s_{i+1,j-w_{i+1}});$$

et, si $w_{i+1} > j$, alors $s_{i,j} = s_{i+1,j}$.

La complexité de remplissage de la matrice est en $\mathcal{O}(n\,P)$ en temps et en espace. On n'a pas prouvé **P** = **NP**, la taille de l'entrée est

- pour un entier n : $\log_2(n)$,
- pour un tableau de n entiers : $n\log_2(n)$,
- pour un tableau de n entiers : $n\log_2(n)$,
- pour un entier P : $\log_2(P)$.

Vis à vis de la taille de l'entrée, la complexité de remplissage est exponentielle.

CHAPITRE

8

JEUX

Sommaire

8.1 Jeux sur un graphe **171**
8.2 Résolution par heuristique **177**

Dans ce chapitre, on s'intéresse à la théorie des jeux. L'objectifs est de modéliser des interactions (économie, coopération, ...). On s'intéressera, par contre, à une petite partie de la théorie des jeux : les jeux d'accessibilité. Par exemple, les échecs et le go sont deux jeux d'accessibilité, mais on s'intéressera à des jeux beaucoup plus simples. On cherchera les stratégies gagnantes pour ces jeux, des heuristiques. On raccrochera ce chapitre avec la théorème des graphes.

Par exemple, on peut coder un programme répondant, *plus ou moins correctement*, au jeu puissance 4. [Démonstration d'un jeu de puissance 4.]

Dans un premier temps, on s'intéresse à un jeu simple. On a 13 allumettes :

| | | | | | | | | | | | | |.

On peut retirer une, deux ou trois allumettes. Le joueur retirant la dernière allumette perd. On peut représenter le jeu par un graphe, et *jouer* au jeu est se déplacer dans ce graphe. Ce graphe est *biparti*.

8.1 Jeux sur un graphe

Définition : On appelle *arène* la donnée
- d'un graphe biparti orienté $G = (V, E)$, avec $V = V_A \cup V_B$ la séparation en deux sommets de ce graphes bipartis ;
- d'un sommet initial s.

Définition : Un état du jeu (un sommet de l'arène) est dit *terminal* lorsqu'il n'a pas de successeurs. On note, dans la suite, $V_A^\star$ et $V_B^\star$ les états non terminaux (notation non officielle).

Remarque :
Les états V_A sont les états où c'est à Alice de jouer. Les états V_B sont les états où c'est à Bob de jouer.

Exemple :
On reprend l'exemple du jeu des allumettes. La figure ci-dessus représente les états du jeu. Les états (i) représentent les états de Alice (*i.e.* des éléments de V_A), les états $\boxed{j}$ représentent les états de Bob (*i.e.* des éléments de V_B). Les états doublement encadrés sont terminaux, les autres sont non terminaux.

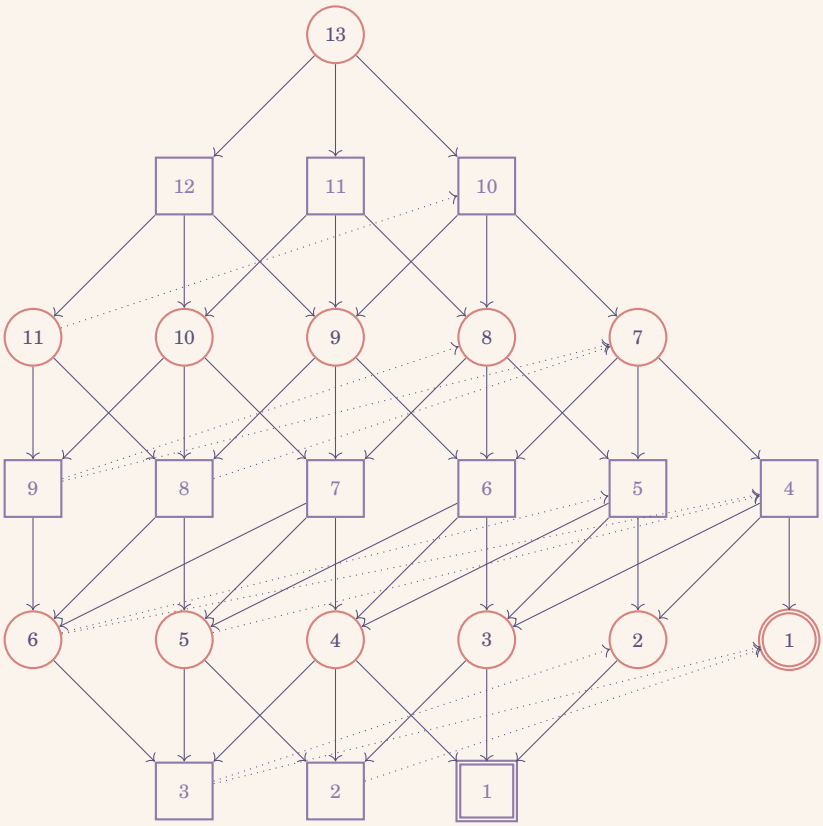


Figure 8.1 – États du jeu des allumettes

Définition : Un *jeu d'accessibilité* est déterminé par une arène $G = (V_A \cup V_B, E)$ de sommet initial s, et deux ensembles $\mathcal{O}_A$ et $\mathcal{O}_B$ de sommets terminaux tels que

- $\mathcal{O}_A \cap \mathcal{O}_B = \varnothing$,
- $\mathcal{O}_A \subseteq V_A \cup V_B$,
- et $\mathcal{O}_B \subseteq V_A \cup V_B$.

L'ensemble $\mathcal{O}_A$ représente les sommets « *gagnants* » pour Alice. L'ensemble $\mathcal{O}_B$ représente les sommets « *gagnants* » pour Bob.

Remarque :
Ils ne forment pas nécessairement une partition des sommets terminaux.

Exemple :
Dans la figure 8.1, l'état gagnant pour Alice est ⓵ ; celui de Bob est [1].

Définition : Une *partie depuis un sommet* s dans un jeu d'accessibilité est une suite (finie ou infinie) de sommets formant un chemin dans l'arène, partant de s, telle que, si la partie est finie, le dernier sommet est terminal.

On dit d'une partie
- qu'elle est *gagnée par Alice* si elle est finie et le dernier sommet est dans $\mathcal{O}_A$;
- qu'elle est *gagnée par Bob* si elle est finie et le dernier sommet est dans $\mathcal{O}_B$;
- qu'elle est nulle sinon.

Exemple :
À faire : Ajouter exemple ?

Définition : On appelle *stratégie pour Alice* une fonction $f : V_A^\star \to V_B$ telle que, pour tout état $s_A \in V_A^\star$, on a $(s_A, f(s_A)) \in E$.

Soit $N \in \mathbb{N} \cup \{+\infty\}$ la longueur d'une partie $(s_1, s_2, \dots, s_n, \dots, s_N ?)$, cette partie est dite *jouée selon une stratégie* f si

$$\forall i \in [\![1, N+1[\![, \quad s_i \in V_A^\star \implies s_{i+1} = f(s_i).$$

Une stratégie pour Alice est dit *gagnante depuis un état* s dès lors que toute partie depuis s jouée selon f est gagnée par Alice. Plus généralement, une stratégie est dit *gagnante* si elle est gagnante depuis l'état initial.

On appelle *stratégie pour Bob* une fonction $f : V_B^\star \to V_B$ telle que, pour tout état $s_B \in V_B^\star$, on a $(s_B, f(s_B)) \in E$.

Soit $N \in \mathbb{N} \cup \{+\infty\}$ la longueur d'une partie $(s_1, s_2, \dots, s_n, \dots, s_N ?)$, cette partie est dite *jouée selon une stratégie* f si

$$\forall i \in [\![1, N+1[\![, \quad s_i \in V_B^\star \implies s_{i+1} = f(s_i).$$

Une stratégie pour Bob est dit *gagnante depuis un état* s dès lors que toute partie depuis s jouée selon f est gagnée par Bob.

Exemple :
La stratégie $\Longrightarrow$ est gagnante pour Bob.

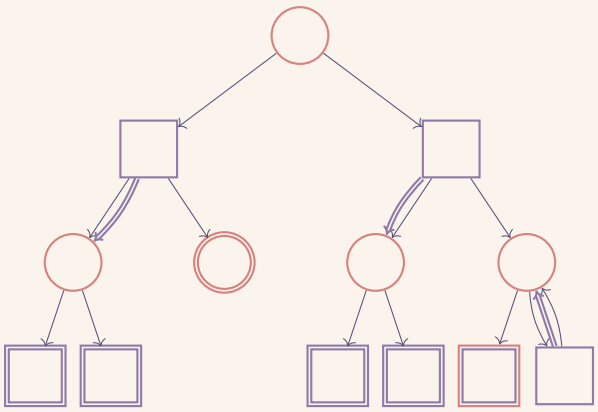


FIGURE 8.2 – Stratégie gagnante pour Bob

Interlude : représentation machine des jeux. On représente le graphe de manière implicite.

```
type joueur
type etat

val joueur : etat -> move
val possibilite_moves : etat -> etat list
val init : etat
val est_gagant : etat -> joueur option
```

CODE 8.1 – Représentation machine des jeux, types

```
type joueur = Alice | Bob
type etat = { allu : int; joueur : joueur }

let autre = function
| Alice -> Bob
| Bob -> Alice

let joueur (e: etat) = e.joueur
let init = { allu = 13; joueur = Alice }
let est_gagnant (e: etat) =
  if e.allu = 1 then Some(autre e.joueur)
  else None
let possible_moves (e: etat) =
  (if e.allu > 3 then
    [{ allu = e.allu - 3; joueur = autre e.joueur }] else [])
  @ (if e.allu > 2 then
    [{ allu = e.allu - 2; joueur = autre e.joueur }] else [])
  @ (if e.allu > 1 then
    [{ allu = e.allu - 1; joueur = autre e.joueur }] else [])
```

CODE 8.2 – Représentation machine du jeux des allumettes

Avec une représentation implicite du graphe, on ne stocke pas l'entièreté du graphe directement mais on ne récupère que les successeurs d'un sommet en particulier.

Par exemple, pour écrire un moteur de recherche, on utilise aussi une représentation implicite pour le graphe *internet* ; on ne peut pas *juste* télécharger l'entièreté du graphe.

Attracteur.

Remarque :
On suppose que les deux joueurs (Alice et Bob) jouent intelligemment. On se met à la place d'Alice. Bob joue le "mieux" pour lui, et le "pire" pour nous. Nous jouerons le "mieux" pour nous. Ainsi, Bob jouera vers un état de valeur minimale, nous jouerons vers un état de valeur maximale.

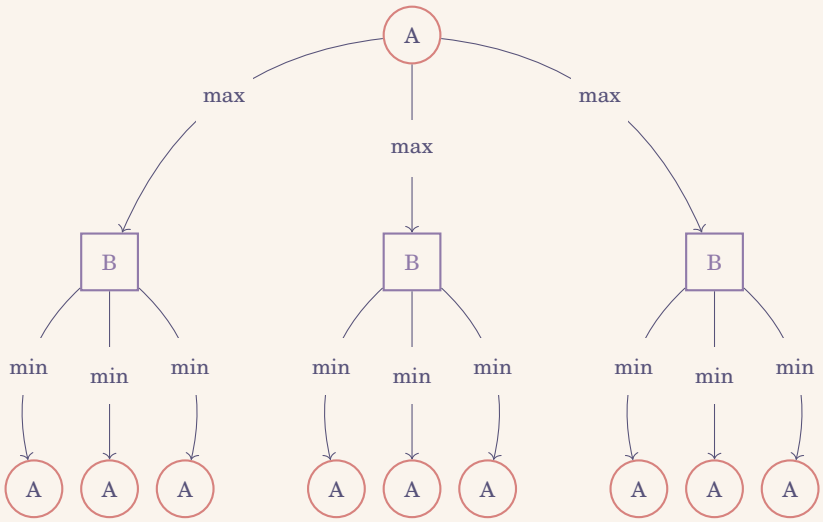


Figure 8.3 – Stratégie de minimisation pour Bob, et de maximisation pour Alice

Définition : On définit la suite d'ensembles $(\mathscr{A}_i)_{i\in\mathbb{N}}$ par $\mathscr{A}_0 = \mathscr{O}_\mathrm{A}$, et, pour tout $n \in \mathbb{N}$,

$$\begin{aligned}\mathscr{A}_{n+1} = \mathscr{A}_n \cup \{s_\mathrm{B} \in V_\mathrm{B}^\star \mid \forall s_\mathrm{A} \in \mathrm{succ}(s_\mathrm{B}),\ s_\mathrm{A} \in \mathscr{A}_n\} \\ \cup \{s_\mathrm{A} \in V_\mathrm{A}^\star \mid \exists s_\mathrm{B} \in \mathrm{succ}(s_\mathrm{A}),\ s_\mathrm{B} \in \mathscr{A}_n\}.\end{aligned}$$

On a appelle alors *attracteur d'Alice* les états $\mathscr{A} = \bigcup_{n\in\mathbb{N}} \mathscr{A}_n$.

Remarque :
La suite des $(\mathscr{A}_n)_{n\in\mathbb{N}}$ est croissante pour l'inclusion $\subseteq$, et ultimement stationnaire car le graphe est fini :

$$\exists N \in \mathbb{N},\ \forall n \geqslant N, \qquad \mathscr{A}_n = \mathscr{A}_N.$$

De même, on définit la suite d'ensembles $(\mathscr{B}_n)_{n\in\mathbb{N}}$ et l'ensemble $\mathscr{B}$ des attracteurs de Bob, en permutant les deux joueurs dans la définition précédente.

Exemple :
À faire : Ajouter exemple, *c.f. cahier de prépa* Si Bob joue intelligemment, il est garanti de gagner.

Remarque :
On remarque que les ensembles $\bigcup_{n\in\mathbb{N}} \mathscr{A}_n$ et $\bigcup_{n\in\mathbb{N}} \mathscr{B}_n$ ne forment pas une partition de l'ensemble des états : l'intersection est bien vide, mais l'union de ces deux ensembles ne

couvre pas l'ensemble des états.

Définition (Stratégie induite par les attracteurs) **:** Soit $\mathscr{A}$ l'ensemble des attracteurs d'Alice. On définit la stratégie suivante

$$f : V_{\mathrm{A}}^\star \longrightarrow V_{\mathrm{B}}$$
$$s_{\mathrm{A}} \longmapsto \begin{cases} s_{\mathrm{B}} \in \mathrm{succ}(s_{\mathrm{A}}) \cap \mathscr{A} & \text{si } s_{\mathrm{A}} \in \mathscr{A} \\ s_{\mathrm{B}} \in \mathrm{succ}(s_{\mathrm{A}}) \text{ quelconque} & \text{sinon.} \end{cases}$$

Cette stratégie est nommée *stratégie induite par les attracteurs*.

Définition : Soit $s \in \mathscr{A}$ un attracteur. On appelle *rang* de s, notée $\mathrm{rg}(s)$ l'entier

$$\mathrm{rg}(s) = \min\{n \in \mathbb{N} \mid s \in \mathscr{A}_n\}.$$

Lemme : Pour tout entier n, pour tout état $s \in \mathscr{A}_n$, un des trois cas est vrai :

- $s \in \mathcal{O}_{\mathrm{A}}$;
- $n > 0$, $s \in V_{\mathrm{A}}^\star$ et il existe $s_{\mathrm{B}} \in \mathrm{succ}(s)$ tel que $s_{\mathrm{B}} \in \mathscr{A}_{n-1}$;
- $n > 0$, $s \in V_{\mathrm{B}}^\star$ et pour tout $s_{\mathrm{A}} \in \mathrm{succ}(s)$, $s_{\mathrm{A}} \in \mathscr{A}_{n-1}$.

Preuve :
Par récurrence.

- Si $n = 0$, alors $\mathscr{A}_0 = \mathcal{O}_{\mathrm{A}}$.
- Supposons la propriété vraie au rang n. Soit $s \in \mathscr{A}_{n+1}$. Alors,
  - ou bien $s \in \mathscr{A}_n$, et on conclut par hypothèse de récurrence.
  - ou bien $s \in V_{\mathrm{A}}^\star$, et il existe $s_{\mathrm{B}} \in \mathrm{succ}(s)$ tel que $s_{\mathrm{B}} \in \mathscr{A}_n$, alors OK ;
  - ou bien $s \in B_{\mathrm{B}}^\star$, et pour tout $s_{\mathrm{A}} \in \mathrm{succ}(s)$, $s_{\mathrm{A}} \in \mathscr{A}_n$, alors OK.

□

Propriété : On a $\mathscr{A} \cap \mathscr{B} = \varnothing$.

Preuve :
Montrons par récurrence que, pour tout entier n, $\mathscr{A}_n \cap \mathscr{B}_n = \varnothing$. À faire : finir cette partie de la preuve.
Soit $N \in \mathbb{N}$ tel que $\mathscr{A}_n = \mathscr{A}_N$, pour tout $n \geqslant N$. Soit $M \in \mathbb{N}$ tel que $\mathscr{B}_n = \mathscr{B}_M$, pour tout $n \geqslant M$. Ainsi, $\mathscr{A} = \mathscr{A}_N$ et $\mathscr{B} = \mathscr{B}_M$. On pose $K = \max(M, N)$, et on a donc

$$\mathscr{A} \cap \mathscr{B} = \mathscr{A}_K \cap \mathscr{B}_K = \varnothing.$$

□

Propriété : Si f est une stratégie induite par les attracteurs d'Alice $\mathscr{A}$, et que $s \in \mathscr{A}$, alors toute partie jouée selon f depuis s est gagnante pour Alice.

Preuve :
Soit $(s_n)_{n \in [\![1, N+1[\![}$, avec $N \in \mathbb{N} \cup \{+\infty\}$, une partie jouée selon f depuis s.

- Montrons $\forall n \in [\![1, N+1[\![$, $s_n \in \mathscr{A}$. Par récurrence. On a $s_1 = s \in \mathscr{A}$, d'où l'initialisation. Si la propriété est vraie au rang $n \in [\![1, N[\![$ (ainsi s_{n+1} existe),

- si $s_n \in V_A^\star$, alors $s_{n+1} = f(s_n) \in \text{succ}(s_n) \cap \mathcal{A}$ donc $s_{n+1} \in \mathcal{A}$,
- si $s_n \in V_B^\star$, on a $s_n \in \mathcal{A}$ et $s_{n+1} \in \text{succ}(s_n)$, d'où $s_{n+1} \in \mathcal{A}$.

Concluant ainsi la récurrence.

— Montrons que $N \neq +\infty$. En effet, montrons que, $\forall n \in [\![1, N[\![, \text{rg}(s_{n+1}) < \text{rg}(s_n)$. L'état $s_n \in \mathcal{A}$, son rang est bien défini. De plus, $s_n \in \mathcal{A}_r$, où $r = \text{rg}(s_n)$. Or, $s_n \notin \mathcal{O}_A$. D'après le lemme, $s_{n+1} \in \mathcal{A}_{r-1}$, donc $\text{rg}(s_{n+1}) \leqslant r - 1 < r$. Ainsi, $N \neq +\infty$ par existence du variant, comme l'ensemble ordonné $(\mathbb{N}, \leqslant)$ est bien fondé.

Ainsi, $(s_n)_{n \in [\![1, N+1[\![}$ est une partie finie, et $\forall n \in [\![1, N]\!]$, $s_n \in \mathcal{A}$. L'état s_N est terminal, et donc $s_N \in \mathcal{O}_A$, d'où le résultat. □

Propriété : Soit s un sommet admettant une stratégie gagnante pour Alice. Alors, s est un attracteur : $s \in \mathcal{A}$.

Preuve :
Montrons par récurrence forte : « si s est un sommet admettant une stratégie gagnante f et que toute partie jouée depuis s selon f est de longueur au plus n, alors $s \in \mathcal{A}_n$. »

— Si le sommet s admet une stratégie gagnante, et toute partie depuis s est de longueur 0, alors $s \in \mathcal{O}_A = \mathcal{A}_0$.

— Supposons, pour tout $k \in [\![1, n]\!]$, l'hypothèse de récurrence. Montrons la propriété pour $n+1$. Soit s admettant une stratégie gagnante f et tel que toute partie jouée depuis s selon f est de longueur au plus $n+1$.

- Si $s \in V_B$, alors
  - si s est terminal, $s \in \mathcal{O}_A \subseteq \mathcal{A}$.
  - sinon, soit $s_1 \in \text{succ}(s)$. Soit γ une partie jouée selon f depuis s_1. γ est une partie jouée selon f depuis s, elle est donc gagnante et de longueur au plus $n+1$. Alors γ est gagnante et de longueur au plus n donc s_1 vérifie les prémisses de l'hypothèse de récurrence, et donc $s_1 \in \mathcal{A}_n$. Ceci étant vrai pour tout $s_1 \in \text{succ}(s)$, on a $s \in \mathcal{A}_{n+1}$.
- Si $s \in V_A$, alors
  - si s est terminal, alors $s \in \mathcal{O}_A \subseteq \mathcal{A}$.
  - sinon, soit $s_1 = f(s)$, alors s_1 admet une stratégie gagnante (f), et toute partie jouée depuis s_1 est de longueur, au plus, n. L'état s_1 vérifie donc les prémisses l'hypothèse de récurrence. Ainsi $s_1 \in \mathcal{A}_n$, et donc s a un successeur dans $\mathcal{A}_n$. On en déduit $s \in \mathcal{A}_{n+1}$.

Soit alors s admettant une stratégie gagnante f. Ainsi, toute partie jouée depuis s selon f est finie. On peut donc conclure grâce à la démonstration par récurrence ci-avant. □

En pratique, la détermination des attracteurs nécessite une quantité bien trop importante de calculs. On souhaite revenir à une idée développée précédemment : réaliser un min-max. Mais, pour cela, il est nécessaire de donner une notion de « valeur » a chaque état du jeu. On définit donc une *fonction d'heuristique* qui, à un état, associé sa valeur. Le choix de cette fonction reste, cependant, très subjectif.

8.2 Résolution par heuristique

On suppose définie une *fonction d'heuristique* h de la forme :

$$h : \{A, B\} \times \underbrace{V}_{\text{états}} \to \overbrace{\mathbb{Z} \cup \{+\infty, -\infty\}}^{\bar{\mathbb{Z}}}.$$

(joueur au-dessus de $\{A, B\}$)

On représente informatiquement l'ensemble $\bar{\mathbb{Z}}$ par le type OCaml `int_bar` défini ci-dessous

```
1  type int_bar =
2    | PInt
3    | NInt
4    | F of int
```

Code 8.3 – Type `int_bar` représentant un élément l'ensemble $\bar{\mathbb{Z}}$

On peut donc définir l'algorithme MinMax.

Algorithme 8.1 Algorithme MinMax

Entrée Un état e, un seuil de profondeur d, le joueur j
Sortie Un score

```
si succ(e) = ∅ alors
    si e ∈ 𝒪_j alors
        retourner +∞
    sinon si e ∈ 𝒪_autre(j) alors
        retourner −∞
    sinon
        retourner 0
sinon si d = 0 alors
    retourner h(j, e)
sinon
    si j = joueur(e) alors
        retourner max{MinMax(e′, d − 1, j) | e′ ∈ succ(e)}
    sinon
        retourner min {MinMax(e′, d − 1, j) | e′ ∈ succ(e)}
```

Exemple :
c.f. cahier-de-prépa On applique l'algorithme MinMax pour Alice.

Exemple (td 13. Exercice 4) :

On peut améliorer l'algorithme MinMax avec « l'élagage α, β. »

Algorithme 8.2 Algorithme MinMax avec élagage α, β

Entrée Un état e, un seuil de profondeur d, le joueur j, et $\alpha, \beta \in \bar{\mathbb{Z}}$
Sortie Un score dans $[\alpha, \beta]$

```
si succ(e) = ∅ alors
    si e ∈ 𝒪_j alors
        retourner β
    sinon si e ∈ 𝒪_autre(j) alors
        retourner α
    sinon
        si β ⩽ 0 alors retourner β
        sinon si α ⩾ 0 alors retourner α
        sinon retourner 0
sinon si d = 0 alors
    si h(j, e) ⩾ β alors retourner β
    sinon si h(j, e) ⩽ α alors retourner α
    sinon retourner h(j, e)
sinon
    si joueur(e) = j alors
        v ← α
        pour e′ ∈ succ(e) faire
            v ← max(v, MinMaxÉlagué(e′, d − 1, j, v, β))
            si v ⩾ β alors
                retourner β
        retourner v
    sinon
        u ← β
        pour e′ ∈ succ(e) faire
            u ← min(u, MinMaxÉlagué(e′, d − 1, j, α, u))
            si u ⩽ α alors
                retourner α
        retourner u
```

CHAPITRE

9

GRAMMAIRES NON CONTEXTUELLES

Sommaire

9.1 Définition, vocabulaire, propriétés **182**
9.1.1 Grammaires non contextuelles 182
9.1.2 Dérivation 183
9.1.3 Preuves par induction 184
9.1.4 Définitions équivalentes 185
9.2 La hiérarchie de Chomsky **188**
9.2.1 Avec les langages réguliers 188
9.2.2 Lien avec les langages décidables 190

Ce chapitre se rattache à la branche de l'informatique des langages formels. On l'a étudié au chapitre 1 avec les automates, et au chapitre 4 avec les machines. Pour le moment, nous avons 5 classes de langages : (1) les langages finis, (2) les langages locaux, (3) les langages réguliers, (4) les langages décidables en temps polynomial, (5) les langages décidables. L'objectif de ce chapitre se situe entre les points (3) et (4).

Intéressons-nous à un langage particulier, le langage des programmes OCaml avec une syntaxe valide. Ce langage est il régulier ? Non. On peut considérer l'expression

$$e_n = \text{« } \underbrace{(\cdots(}_{n}0\underbrace{)\cdots)}_{n} \text{ »}$$

qui est une expression OCaml valide. Par application du lemme de l'étoile, il n'est pas reconnaissable par un automate à N états en considérant e_{N+1}.

L'ensemble $\mathscr{B}$ des mots bien parenthésés est le plus petit ensemble tel que

— $\varepsilon \in \mathscr{B}$

— si $u \in \mathscr{B}$ et $v \in \mathscr{B}$, alors $u \cdot v \in \mathscr{B}$

— si $u \in \mathscr{B}$, alors $(\cdot\, u \cdot\,) \in \mathscr{B}$.

Une telle définition de langage est appelée une grammaire. Un autre exemple de langage est la grammaire de la langue française :

— phrase : sujet + verbe + complément,

— complément : COD + complément,

— complément : CCL + complément,

— complément : ε.

Avec cette définition [1], on peut reconnaître des phrases simples comme

« $\underbrace{\text{Matthieu}}_{\text{sujet}}$ $\underbrace{\text{aime}}_{\text{verbe}}$ $\underbrace{\text{les trains}}_{\text{complément}}$. »

9.1 Définition, vocabulaire, propriétés

9.1.1 Grammaires non contextuelles

Définition : On se munit d'un alphabet Σ qu'on appelle *terminaux*. On se munit d'un ensemble de symboles $\mathcal{V}$ qu'on appelle *non-terminaux*. On suppose $\mathcal{V} \cap \Sigma$.

Exemple :
Dans la suite, on pose $\Sigma = \{(,)\}$ et $\mathcal{V} = \{\mathrm{B}\}$.

Définition : On appelle *règle de production* la donnée

— d'un symbole $V \in \mathcal{V}$,

— d'un mot $w_1 w_2 \dots w_n$ sur l'alphabet $\mathcal{V} \cup \Sigma$,

que l'on note $V \to w_1 w_2 \dots w_n$.

Exemple :
L'ensemble $\mathcal{B}$ est décrit par les règles

— $\mathrm{B} \to \varepsilon$,

— $\mathrm{B} \to \mathrm{BB}$,

— $\mathrm{B} \to (\mathrm{B})$.

Définition (Grammaire non contextuelle) **:** Une *grammaire non contextuelle* est la donnée de

— un alphabet de non-terminaux $\mathcal{V}$,

— un alphabet de terminaux Σ, [2]

— un ensemble fini de règles de production P,

— un symbole initial $S \in \mathcal{V}$,

que l'on note $(\mathcal{V}, \Sigma, P, S)$.

Exemple :
On note dans la suite

$$\mathcal{G} = \big(\{\mathrm{B}\}, \{(,)\}, \{\mathrm{B} \to \varepsilon, \mathrm{B} \to \mathrm{BB}, \mathrm{B} \to (\mathrm{B})\}, \mathrm{B}\big).$$

1. On néglige les règles manquantes à cette définition de la grammaire française.
2. avec "terminaux/non-terminaux" vient l'implication que $\mathcal{V} \cap \Sigma = \varnothing$

Interlude. Avec cette définition, on espère pouvoir appliquer ces règles pour définir des mots valides. Par exemple,

$$\mathrm{B} \overset{?}{\rightsquigarrow} \mathrm{BB} \overset{?}{\rightsquigarrow} (\mathrm{B})\mathrm{B} \overset{?}{\rightsquigarrow} (\mathrm{B})(\mathrm{B}) \overset{?}{\rightsquigarrow} (\mathrm{BB})(\mathrm{B}) \overset{?}{\rightsquigarrow} (\mathrm{BB})() \overset{?}{\rightsquigarrow} (\mathrm{B})() \overset{?}{\rightsquigarrow} ((\mathrm{B}))() \overset{?}{\rightsquigarrow} (())().$$

C'est l'objectif de la sous-section suivante.

9.1.2 Dérivation

Définition (Dérivation immédiate) **:** Soit $\mathcal{G} = (\mathcal{V}, \Sigma, P, S)$ une grammaire. Soient u et v deux mots de $(\Sigma \cup \mathcal{V})^\star$. On dit que v *dérive immédiatement* de u dans la grammaire $\mathcal{G}$, que l'on note $u \Rightarrow v$, si

— il existe x et y deux mots de $(\Sigma \cup \mathcal{V})^\star$

— il existe $(V \to w_1 w_2 \dots w_n) \in P$

tels que $u = x \cdot V \cdot y$ et $v = x \cdot w_1 w_2 \dots w_n \cdot y$.

Exemple :
Ainsi, $u \Rightarrow v$.

Lemme (de composition) **:** Soit $\mathcal{G} = (\mathcal{V}, \Sigma, P, S)$ une grammaire. Soient $u, v \in (\Sigma \cup \mathcal{V})^\star$ tels que $u \Rightarrow v$. Soit $w \in (\Sigma \cup \mathcal{V})^\star$. On a $u \cdot w \Rightarrow v \cdot w$ et $w \cdot u \Rightarrow w \cdot v$.

Preuve :
À faire à la maison. □

On propose trois définitions différentes mais équivalentes pour la dérivation $\overset{\star}{\Rightarrow}$.

Définition : Étant donné une grammaire $\mathcal{G} = (\mathcal{V}, \Sigma, P, S)$, on étend $\Rightarrow$ en sa cloture réflexive et transitive, que l'on note $\overset{\star}{\Rightarrow}$, appelé *dérivation*.

Définition : On définit $\overset{\star}{\Rightarrow}$ comme $u \overset{\star}{\Rightarrow} v \overset{\text{def.}}{\iff} \exists n \in \mathbb{N},\ \exists (u_0, u_1, \dots, u_n) \in ((\mathcal{V} \cup \Sigma)^\star)^{n+1}$ tels que $u_0 = u$, $u_n = v$ et $\forall i \in [\![0, n-1]\!],\ u_i \Rightarrow u_{i+1}$.

Définition : Soit la suite $(\Rightarrow^n)_{n \in \mathbb{N}}$ définie inductivement par

— $u \Rightarrow^0 v \overset{\text{def.}}{\iff} u = v$,

— $u \Rightarrow^{n+1} v \overset{\text{def.}}{\iff} u \Rightarrow^n v$ ou $\exists w \in (\Sigma \cup \mathcal{V})^\star,\ u \Rightarrow w$ et $w \Rightarrow^n v$.

On pose alors $\overset{\star}{\Rightarrow} = \bigcup_{n \in \mathbb{N}} \Rightarrow^n$.

Lemme (de composition*) **:** Soit $\mathcal{G} = (\mathcal{V}, \Sigma, P, S)$ une grammaire, et soient $u, v \in (\Sigma \cup \mathcal{V})^\star$ tels que $u \Rightarrow^p v$, pour $p \in \mathbb{N}$. Et, soient $(w, t) \in (\Sigma \cup \mathcal{V})^2$ tels que $w \Rightarrow^q t$, pour $q \in \mathbb{N}$. Alors, $uw \Rightarrow^{p+q} vt$.

Preuve :
Soit $u_0u_1\dots u_p$ tels que $u_0 = u$, $u_p = v$ et $\forall i \in [\![0, p-1]\!]$, $u_i \Rightarrow u_{i+1}$. Soit $w_0w_1\dots w_n$ tels que $w_0 = w$, $w_q = t$ et $\forall i \in [\![0, q-1]\!]$, $w_i \Rightarrow w_{i+1}$. Soit alors la dérivation

$$uw = u_0w \Rightarrow u_1w \Rightarrow u_2w \Rightarrow \cdots \Rightarrow u_pw \Rightarrow u_pw_1 \Rightarrow u_pw_2 \Rightarrow \cdots \Rightarrow u_pw_q = vt.$$

On a donc $uw \Rightarrow^{p+q} vt$. □

Définition : Soit $\mathcal{G} = (\mathcal{V}, \Sigma, P, S)$ une grammaire. Son langage est défini par

$$\mathcal{L}(\mathcal{G}) = \{u \in \Sigma^\star \mid S \overset{\star}{\Rightarrow} u\}.$$

Exemple :
Dans l'exemple précédent, on a

$$\mathcal{L}(\mathcal{G}) = \mathcal{B}.$$

Interlude : grammaires contextuelles. Dans une grammaire contextuelle, une règle de production peut-être de la forme $b\mathrm{B} \to a\mathrm{B}a\mathrm{B}$, où $\Sigma = \{a, b\}$. Ces règles dépendent du contexte, et non juste des symboles.

Lemme (de décomposition) **:** Étant donnée une grammaire non contextuelle $(\mathcal{V}, \Sigma, P, S)$, soit $w = w_1\dots w_n$ un mot de n lettres tel qu'il existe $p \in \mathbb{N}$ et $v \in (\Sigma \cup \mathcal{V})^\star$ tel que $w \Rightarrow^p v$ alors il existe $\tilde{v}_1, \tilde{v}_2, \dots, \tilde{v}_n \in (\Sigma \cup \mathcal{V})^\star$ et $p_1, \dots, p_n \in \mathbb{N}$ tels que $w_1 \Rightarrow^{p_1} \tilde{v}_1$, $w_2 \Rightarrow^{p_2} \tilde{v}_2$, …, $w_n \Rightarrow^{p_n} \tilde{v}_n$, et $v = \tilde{v}_1 \cdots \tilde{v}_n$, et $\sum_{i=1}^n p_i = p$.

Preuve :
On procède par récurrence sur $p \in \mathbb{N}$.

- **cas de base ($p = 0$).** On a $w_1w_2\dots w_n \Rightarrow^0 v$, d'où $v = w_1w_2\dots w_n$. On choisit donc $\tilde{v}_i = w_i$ et $p_i = 0$ pour tout $i \in [\![1, n]\!]$. On a alors, pour tout $i \in [\![1, n]\!]$, $w_i \Rightarrow^0 \tilde{v}_i$, et $v = \tilde{v}_1\dots\tilde{v}_n$ et $\sum_{i=1}^n p_i = 0 = p$.
- **hérédité.** Soit $w \Rightarrow w_1\dots w_{q-1}u_1u_2\dots u_rw_{q+1}\dots w_n \Rightarrow^{p-1} v$, où $(w_q \to u_1\dots u_r) \in P$. Soit donc, par hypothèse de récurrence, $\hat{v}_1, \hat{v}_2, \dots, \hat{v}_{n-1+r}$ tels que
  - $\forall i \in [\![1, q_1]\!]$, $w_i \Rightarrow^{s_i} \hat{v}_i$,
  - $\forall i \in [\![1, r]\!]$, $u_i \Rightarrow^{s_{i+q-1}} \hat{v}_{i+q-1}$,
  - $\forall i \in [\![q+1, n]\!]$, $w_i \Rightarrow^{s_{i+r-1}} \hat{v}_{i+r-1}$,
  - et $\sum_{i=1}^{n-1+r} s_i = p - 1$.

  On pose alors, pour tout $i \in [\![1, q-1]\!]$, $\tilde{v}_i = \hat{v}_i$, $\tilde{v}_q = \hat{v}_q\hat{v}_{q+1}\cdots\hat{v}_{q+r-1}$, et pour tout $i \in [\![q+1, n]\!]$, $\tilde{v}_i = \hat{v}_{i+r-1}$. On pose aussi, pour $i \in [\![1, q-1]\!]$, $p_i = s_i$, puis $p_q = 1 + \sum_{i=1}^r s_{i+q-1}$, et, pour $i \in [\![q+1, n]\!]$, $p_i = s_{i+r-1}$.
  On a clairement $v = \tilde{v}_1\tilde{v}_2\cdots\tilde{v}_n$. De plus, $\sum_{i=1}^n p_i = \left(\sum_{i=1}^{n+r-1} s_i\right) + 1 = (p-1) + 1 = p$.
  Et surtout, pour $i \in [\![1, q-1]\!]$, $w_i \Rightarrow^{s_i} \hat{v}_i$ donc $w_i \Rightarrow^{p_i} \tilde{v}_i$. Ainsi,

$$w_q \Rightarrow u_1\dots u_r \Rightarrow^{\sum_{i=1}^r s_{i+q-1}} \underbrace{\hat{v}_q\dots\hat{v}_{q+r-1}}_{\tilde{v}_q},$$

  donc $w_q \Rightarrow^{p_q} \tilde{v}_q$ et, pour $i \in [\![q+1, n]\!]$, $w_i \Rightarrow^{s_{i+r-1}} \hat{v}_{r+r-1}$ et $w_i \Rightarrow^{p_i} \tilde{v}_i$.

□

9.1.3 Preuves par induction

Propriété (principe d'induction) **:** Étant donnée une grammaire $\mathcal{G} = (\mathcal{V}, \Sigma, P, S)$, et un non-terminal $V \in \mathcal{V}$, on note localement $V_\downarrow = \{w \in \Sigma^\star \mid V \overset{\star}{\Rightarrow} w\}$.[3] Soit alors un ensemble de propriétés $\mathcal{P}_V$ pour $V \in \mathcal{V}$, tel que, $\forall (V \to w_1 w_2 \dots w_n) \in P$, $\forall (\hat{w}_1, \hat{w}_2, \dots, \hat{w}_m) \in (\Sigma^\star)^m$

$$\left(\forall i \in [\![1, m]\!], \begin{cases} w_i \in \Sigma \implies \hat{w}_i = w_i \\ w_i \in \mathcal{V} \implies \hat{w}_i \text{ vérifie } \mathcal{P}_{w_i} \end{cases} \right) \implies \hat{w}_i \dots \hat{w}_n \text{ vérifie } \mathcal{P}_V,$$

alors pour tout $V \in \mathcal{V}$, $V_\downarrow$ vérifie $\mathcal{P}_V$.

Exemple :
On considère $\mathcal{G}$ la grammaire

$$\mathcal{G} = \big(\{\mathrm{P}, \mathrm{I}, \mathrm{X}\}, \{a\}, \{\mathrm{P} \to \varepsilon, \mathrm{I} \to a, \mathrm{P} \to a\mathrm{P}a, \mathrm{I} \to a\mathrm{I}a, \mathrm{X} \to \mathrm{IPI}\}, \mathrm{X}\big).$$

On pose les propriétés $\mathcal{P}_\mathrm{P}(w)$: « $|w|$ est pair » ; $\mathcal{P}_\mathrm{I}(w)$: « $|w|$ est impair » ; et, $\mathcal{P}_\mathrm{X}(w)$: « $|w|$ est pair. »

- **cas $\mathrm{P} \to \varepsilon$.** Le mot ε est de taille pair donc $\mathcal{P}_\mathrm{P}(\varepsilon)$ est vrai.
- **cas $\mathrm{I} \to a$.** $|a|$ est impair donc $\mathcal{P}_\mathrm{I}(a)$ est vrai.
- **cas $\mathrm{P} \to a\mathrm{P}a$.** Soit donc $w = aw'a$ avec w' vérifiant $\mathcal{P}_\mathrm{P}$. Alors, $|w| = 2 + |w'|$ qui est pair.
- **cas $\mathrm{I} \to a\mathrm{I}a$.** Soit donc $w = aw'a$ avec w' vérifiant $\mathcal{P}_\mathrm{I}$. Alors, $|w| = 2 + |w'|$ qui est impair.
- **cas $\mathrm{X} \to \mathrm{IPI}$.** Soit donc $w = xyz$, avec x et z vérifiant $\mathcal{P}_\mathrm{I}$, et y vérifiant $\mathcal{P}_\mathrm{P}$. Alors, $|w| = |x| + |y| + |z|$, qui est pair.

9.1.4 Définitions équivalentes

Comment représenter une dérivation en machine ? On considère les règles de production $\mathrm{X} \to \mathrm{XX}$ et $\mathrm{X} \to a$. On peut, par exemple, représenter une dérivation par un ensemble de possibilités :

$$\mathrm{X} \Rightarrow \mathrm{XX} \Rightarrow \{a\mathrm{X}, \mathrm{X}a, \mathrm{XXX}\} \Rightarrow \cdots.$$

Mais, avec une telle définition, il y a explosions du nombre de possibilités.

Définition (Dérivation immédiatement gauche (resp. droite)) **:** Étant donnée une grammaire $\mathcal{G} = (\mathcal{V}, \Sigma, P, I)$, et étant donnés deux mots u et v de $(\Sigma \cup \mathcal{V})^\star$, on dit que u *dérive immédiatement à gauche* (resp. droite) de u dès lors qu'il existe $x \in \Sigma^\star$, $y \in (\Sigma \cup \mathcal{V})^\star$ (resp. $x \in (\Sigma \cup \mathcal{V})^\star$ et $y \in \Sigma^\star$), et $(V \to w_1 \dots w_n) \in P$ tels que $u = xVy$ et $v = x \cdot w_1 w_2 \dots w_n \cdot y$. On note alors $u \Rightarrow_\mathrm{g} v$ (resp. $u \Rightarrow_\mathrm{d} v$). On définit, de la même manière que pour la dérivation simple, $\overset{\star}{\Rightarrow}_\mathrm{g}$ et $\overset{\star}{\Rightarrow}_\mathrm{d}$.

Exemple :
On considère une grammaire ayant pour règles de production $\mathrm{X} \to \mathrm{XX}$ et $\mathrm{X} \to a$. A-t-on

- $\mathrm{X} \overset{\star}{\Rightarrow}_\mathrm{g} a\mathrm{X}$? ✓ ($\mathrm{X} \Rightarrow_\mathrm{g} \mathrm{XX} \Rightarrow_\mathrm{g} a\mathrm{X}$)
- $\mathrm{X} \overset{\star}{\Rightarrow}_\mathrm{g} \mathrm{X}a$? ✗

3. Avec cette définition, $\mathscr{L}(\mathcal{G}) = S_\downarrow$.

Définition (Arbre de dérivation) **:** Étant donnée une grammaire $\mathcal{G} = (\mathcal{V}, \Sigma, P, S)$, on appelle *arbre de dérivation* un arbre dont les nœuds sont étiquetés par des éléments de $\{\varepsilon\} \cup \mathcal{V} \cup \Sigma$ et tel que

- tout nœud interne (ayant des fils) est étiquetés par un élément de $\mathcal{V}$,
- la racine est étiquetée par S,
- tout nœud interne ayant une étiquette V et ayant des fils $T_1, \dots, T_n$ où $n \neq 0$ dont les racines sont étiquetées par $w_1, \dots, w_n$ avec $(V \to w_1 \dots w_n) \in P$.
- tout nœud interne d'étiquette V ayant pour unique fils l'arbre feuille réduit à ε et tel que $(V \to \varepsilon) \in P$.

Dans la suite du chapitre, on fixe $\mathcal{G} = (\mathcal{V}, \Sigma, I, P)$ une grammaire.

Exemple :

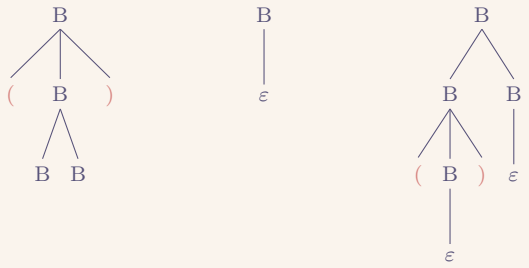


Figure 9.1 – Exemple d'arbres de dérivation

Définition (« production » d'un arbre) **:** On définit inductivement la fonction prod de l'ensemble des arbres de la grammaire $\mathcal{G}$ vers $(\Sigma \cup \mathcal{V})^\star$, comme

- $\text{prod}\big(\text{Leaf}(x)\big) = x$,
- $\text{prod}\big(\text{Node}(\_, [T_1, \dots, T_n])\big) = \text{prod}(T_1) \cdot \text{prod}(T_2) \cdot \dots \cdot \text{prod}(T_n)$.

Exemple :
Dans les arbres de dérivation exemples précédents, la fonction prod retourne (BB), ε et ( ).

Remarque (Notation) :
On dit qu'un arbre de dérivation T est « *clos* » lorsque $\text{prod}(T) \in \Sigma^\star$.

Exemple :
Dans les exemples précédents, le premier arbre n'est pas « clos » mais les deux suivants le sont.

Propriété : Étant donné $w \in \Sigma^\star$, il est équivalent de dire que

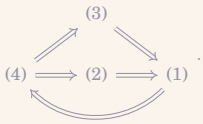
(1) $w \in \mathcal{L}(\mathcal{G})$,

(2) $S \overset{\star}{\Rightarrow}_{\mathrm{g}} w$

(3) $S \overset{\star}{\Rightarrow}_{\mathrm{d}} w$

(4) il existe un arbre de dérivation T dont w est le produit.

Preuve :
On procède la démonstration dans l'ordre suivant.



— **(4) $\Longrightarrow$ (2).** Soit la propriété $\mathcal{P}(T)$: « si $w \in \Sigma^\star$ admet T comme arbre de dérivation (w est le produit de T) avec T un arbre de dérivation généralisé, enraciné en $V \in \mathcal{V}$, alors $V \overset{\star}{\Rightarrow}_{\mathrm{g}} w$. » Montrons cette propriété par induction.
  — Si $T = \mathrm{Leaf}(x)$ avec $x \in \Sigma \cup \{\varepsilon\}$, alors ok par un arbre de dérivation.
  — Si $\mathrm{T} = \mathrm{Node}(V, [T_1, \dots, T_n])$, alors, pout $i \in [\![1, n]\!]$, raisonnons par disjonction.
    — Si T_i a une racine d'étiquette $w_i \in \Sigma$, alors $\mathrm{prod}(T_i) = w_i$, donc $w_i \overset{\star}{\Rightarrow}_{\mathrm{g}} \mathrm{prod}(T_i)$
    — Si T_i a une racine d'étiquette $w_i \in \mathcal{V}$, alors, par hypothèse d'induction sur T_i, on a $w_i \overset{\star}{\Rightarrow}_{\mathrm{g}} \mathrm{prod}(T_i)$.

  Ainsi, par concaténation, $\mathrm{prod}(T) = \mathrm{prod}(T_1) \cdot \ldots \cdot \mathrm{prod}(T_n)$. Or, $(V \to w_1 \dots w_n) \in P$. Alors, considérons la dérivation

$$\begin{aligned} V &\Rightarrow_{\mathrm{g}} w_1 w_2 \dots w_n \\ &\overset{\star}{\Rightarrow}_{\mathrm{g}} \mathrm{prod}(T_1) \cdot w_2 \dots w_n \\ &\overset{\star}{\Rightarrow}_{\mathrm{g}} \mathrm{prod}(T_1) \cdot \mathrm{prod}(T_2) \\ &\overset{\star}{\Rightarrow}_{\mathrm{g}} \cdots \overset{\star}{\Rightarrow}_{\mathrm{g}} \mathrm{prod}(T_1) \cdot \mathrm{prod}(T_2) \cdot \ldots \cdot \mathrm{prod}(T_n) \end{aligned}$$

— **(2) $\Longrightarrow$ (1).** Vrai car $\Rightarrow_{\mathrm{g}}$ est « inclus dans » $\Rightarrow$ (c'est un cas particulier).

— **(1) $\Longrightarrow$ (4).** Soit la propriété $\mathcal{P}_n$ « $\forall V \in \mathcal{V}, \forall w \in \Sigma^\star$, si $V \Rightarrow^n w$, alors w admet un arbre de dérivation généralisé enraciné en V. » Montrons le par induction.
  — Si $n = 0$, absurde.
  — Si $V \Rightarrow^n w$ avec $n \geqslant 1$, donc $V \Rightarrow w_1 w_2 \dots w \Rightarrow^{n-1} w$ donc $(V \to w_1 w_2 \dots w_n) \in P$, alors, par lemme de décompositions, il existe $\tilde{w}_1, \dots, \tilde{w}_p$ tels que $w = \tilde{w}_1 \cdot \ldots \cdot \tilde{w}_p$ et $\forall i \in [\![1, p]\!]$, $w_i \Rightarrow^{p_i} \tilde{w}_i$ avec $\sum_{i=1}^{p} p_i = n - 1$. D'où, $\forall i \in [\![1, p]\!]$, $p_i < n$. Par hypothèse d'induction, il existe, pour tout $i \in [\![1, p]\!]$, un arbre T_i produisant $\tilde{w}_i$ enraciné en w_i. Soit alors $T = \mathrm{Node}(w, [T_1, \dots, T_p])$. Ainsi, $\mathrm{prod}(T) = \tilde{w}_1 \cdot \ldots \cdot \tilde{w}_p = w$. Dans l'éventualité où l'un des p_i aurait le mauvais goût d'être nul, on fabrique, à la place, l'arbre $T_i = \mathrm{Leaf}(w_i)$.

□

Définition : Une grammaire est dite *ambigüe* s'il existe un mot w de son langage admettant au moins deux arbres de dérivations.

Exemple :
On considère la grammaire de non-terminal initial B ayant pour règles de production $\mathrm{F} \to 0 \mid 1 \mid \cdots \mid 9$ et $\mathrm{B} \to \mathrm{B} + \mathrm{B} \mid \mathrm{B} - \mathrm{B} \mid \mathrm{F}$. Le mot « $1 - 1 + 9$ » admet les deux arbres de dérivations ci-dessous.

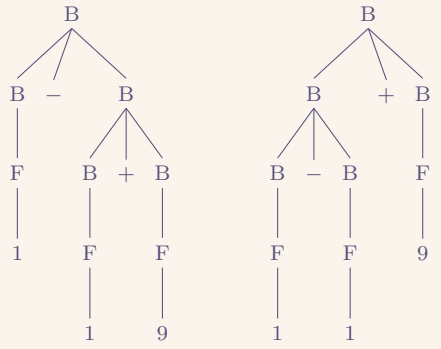


FIGURE 9.2 – Arbres de dérivations de « $1 - 1 + 9$ »

Interlude : langages algébriques. Les langages reconnus par des grammaires non-contextuelles sont des solutions d'équations polynômiales, où la multiplication correspond à la concaténation et l'addition correspond à l'union. D'où le terme langage *algébrique*.

Définition : Deux grammaires $\mathcal{G}_1$ et $\mathcal{G}_2$ sont dites *faiblement équivalentes* dès lors que $\mathcal{L}(\mathcal{G}_1) = \mathcal{L}(\mathcal{G}_2)$.

EXEMPLE :
On considère la grammaire $\mathcal{G}_1$ de règle de production $\mathrm{S} \to a\mathrm{S} \mid \varepsilon$, et la grammaire $\mathcal{G}_2$ de règle de production $\mathrm{S} \to \mathrm{SS} \mid a \mid \varepsilon$. Les deux grammaires $\mathcal{G}_1$ et $\mathcal{G}_2$ sont faiblement équivalentes. En effet, $\mathcal{L}(\mathcal{G}_1) = \mathcal{L}(a^\star) = \mathcal{L}(\mathcal{G}_2)$. Mais, elles ne sont pas fortement équivalentes. [4]

9.2 La hiérarchie de CHOMSKY

Le terme « *hiérarchie de CHOMSKY* » n'est pas au programme. Dans cette partie, on traite des inclusions avec les autres familles des langages au programme.

9.2.1 Avec les langages réguliers

Propriété : Soient $\mathcal{G}_1$ et $\mathcal{G}_2$ des grammaires non contextuelles. Alors,

1. $\mathcal{L}(\mathcal{G}_1) \cup \mathcal{L}(\mathcal{G}_2)$ est reconnu par une grammaire non-contextuelle ;
2. $\mathcal{L}(\mathcal{G}_1) \cdot \mathcal{L}(\mathcal{G}_2)$ est reconnu par une grammaire non-contextuelle ;
3. $(\mathcal{L}(\mathcal{G}_1))^\star$ est reconnu par une grammaire non-contextuelle.

Preuve :
Soient $\mathcal{G}_1 = (\mathcal{V}_1, \Sigma, P_1, S_1)$ et $\mathcal{G}_2 = (\mathcal{V}_2, \Sigma, P_2, S_2)$. On peut supposer, quitte à renommer les non-terminaux, que $\mathcal{V}_1 \cap \mathcal{V}_2 = \varnothing$.

1. Soit alors $S \notin \mathcal{V}_1 \cup \mathcal{V}_2$. On pose $\mathcal{G} = (\mathcal{V}_1 \cup \mathcal{V}_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \to S_1 \mid S_2\}, S)$. Soit

4. Deux grammaires sont fortement équivalentes si elles produisent les mêmes arbres de dérivation.

$w \in \Sigma^\star$.

$$\begin{aligned} w \in \mathscr{L}(\mathscr{G}) &\iff S \overset{\star}{\Rightarrow}_{\mathscr{G}} w \\ &\iff (S \Rightarrow_{\mathscr{G}} S_1 \overset{\star}{\Rightarrow}_{\mathscr{G}} w) \text{ ou } (S \Rightarrow_{\mathscr{G}} S_2 \overset{\star}{\Rightarrow}_{\mathscr{G}} w) \\ &\iff (S_1 \overset{\star}{\Rightarrow}_{\mathscr{G}} w) \text{ ou } (S_2 \overset{\star}{\Rightarrow}_{\mathscr{G}} w) \\ &\iff (S_1 \overset{\star}{\Rightarrow}_{\mathscr{G}_1} w) \text{ ou } (S_2 \overset{\star}{\Rightarrow}_{\mathscr{G}_2} w) \\ &\iff w \in \mathscr{L}(\mathscr{G}_1) \cup \mathscr{L}(\mathscr{G}_2) \end{aligned}$$

2. Soit alors $S \notin \mathscr{V}_1 \cup \mathscr{V}_2$. On pose $\mathscr{G} = (\mathscr{V}_1 \cup \mathscr{V}_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \to S_1 \cdot S_2\}, S)$. Soit $w \in \Sigma^\star$.

$$\begin{aligned} w \in \mathscr{L}(\mathscr{G}) &\iff S \overset{\star}{\Rightarrow}_{\mathscr{G}} w \\ &\iff S \Rightarrow S_1 \cdot S_2 \overset{\star}{\Rightarrow}_{\mathscr{G}} w \\ &\iff \exists (u,v) \in (\Sigma^\star)^2,\ S_1 \overset{\star}{\Rightarrow}_{\mathscr{G}} u \text{ et } S_2 \overset{\star}{\Rightarrow}_{\mathscr{G}} v \text{ et } w = u \cdot v \\ &\iff \exists (u,v) \in (\Sigma^\star)^2,\ S_1 \overset{\star}{\Rightarrow}_{\mathscr{G}_1} u \text{ et } S_2 \overset{\star}{\Rightarrow}_{\mathscr{G}_2} v \text{ et } w = u \cdot v \\ &\iff \exists (u,v) \in (\Sigma^\star)^2,\ u \in \mathscr{L}(\mathscr{G}_1) \text{ et } v \in \mathscr{L}(\mathscr{G}_2) \text{ et } w = u \cdot v \\ &\iff w \in \mathscr{L}(\mathscr{G}_1) \cdot \mathscr{L}(\mathscr{G}_2) \end{aligned}$$

3. Soit alors $S \notin \mathscr{V}_1 \cup \mathscr{V}_2$. On pose $\mathscr{G} = (\mathscr{V}_1 \cup \{S\}, \Sigma, P_1 \cup \{S \to S \cdot S_1 \mid \varepsilon\}, S)$. Soit $w \in \Sigma^\star$.

$$\begin{aligned} w \in \mathscr{L}(\mathscr{G}) &\iff S \overset{\star}{\Rightarrow}_{\mathscr{G}} w \\ &\iff S \overset{\star}{\Rightarrow}_{\mathscr{G}, gw} \\ &\iff S \Rightarrow \overbrace{S \cdot S_1 \Rightarrow SS_1S_1 \Rightarrow \cdots \Rightarrow \underbrace{S_1S_1\cdots S_1}_{n}}^{n} \overset{\star}{\Rightarrow}_{\mathscr{G}} w^\star \\ &\iff \exists n \in \mathbb{N},\ \exists (\tilde{w}_1, \ldots, \tilde{w}_n),\ (\forall i \in [\![1,n]\!],\ S_1 \overset{\star}{\Rightarrow}_{\mathscr{G}} \tilde{w}_i) \text{ et } w = \tilde{w}_1 \cdots \tilde{w}_n \\ &\iff \exists n \in \mathbb{N},\ \exists (\tilde{w}_1, \ldots, \tilde{w}_n),\ (\forall i \in [\![1,n]\!],\ S_1 \overset{\star}{\Rightarrow}_{\mathscr{G}_1} \tilde{w}_i) \text{ et } w = \tilde{w}_1 \cdots \tilde{w}_n \\ &\iff \exists n \in \mathbb{N},\ \exists (\tilde{w}_1, \ldots, \tilde{w}_n),\ (\forall i \in [\![1,n]\!],\ \tilde{w}_i \in \mathscr{L}(\mathscr{G}_1)) \text{ et } w = \tilde{w}_1 \cdots \tilde{w}_n \\ &\iff w \in (\mathscr{L}(\mathscr{G}_1))^\star \end{aligned}$$

□

Théorème : Tout langage régulier est reconnu par une grammaire non contextuelle.

Preuve :
On a déjà montré que les grammaires non-contextuelles sont stables par union, concaténation et passage à l'étoile. Il ne reste qu'à montrer le résultat sur les cas de base. On a

- $\{\varepsilon\} = \mathscr{L}((\{S\}, \Sigma, \{S \to \varepsilon\}, S))$,
- $\varnothing = \mathscr{L}((\{S\}, \Sigma, \varnothing, S))$, (▷ voir TD 14.)
- $\{\ell\} = \mathscr{L}((\{S\}, \Sigma, \{S \to \ell\}, S))$.

Il suffit alors de conclure par induction. □

Remarque :
L'inclusion précédente est stricte. En effet le langage $\{a^n \cdot b^n \mid n \in \mathbb{N}\}$ est reconnu par une grammaire non contextuelle mais n'est pas un langage régulier.

Remarque (Digression) :

« Tout langage est-il le langage d'une grammaire non contextuelle ? »

On procède par un argument de taille d'ensembles. Soit $\mathcal{G} = (\mathcal{V}, \Sigma, P, I)$ une grammaire. On considère $\Sigma = \{0, 1\}$. On pose $|\mathcal{G}| = |\Sigma| + |\mathcal{V}| + \sum_{(V \to w_1 \dots w_n) \in P} (n + 1) + 1$. On considère $\mathbb{G}_n = \{\mathcal{G} \mid |\mathcal{G}| = n\}$. L'ensemble $\mathbb{G} = \bigcup_{n \in \mathbb{N}} \mathbb{G}_n$ est dénombrable comme union dénombrable d'ensembles finis. Montrons qu'il existe une bijection entre $\wp(\{0,1\}^\star)$ et $[0, 1[$. À tout $x \in [0, 1[$, on peut poser $x = 0, x_1 x_2 \dots x_n \dots$. D'où,

$$[0,1[\xleftrightarrow[\text{encodage binaire}]{\text{bijection}} (\mathbb{N} \to \{0,1\}) \xleftrightarrow[\mathbb{1}]{\text{bijection}} \wp(\mathbb{N}) \xleftrightarrow[\text{encodage binaire}]{\text{bijection}} \wp(\{0,1\}^\star).$$

9.2.2 Lien avec les langages décidables

Propriété : Les langages des grammaires non contextuelles sont des langages décidables. □

La preuve de cette propriété est dans le TD 15. (On peut montrer, par programmation dynamique, que l'appartenance d'un mot à une grammaire non contextuelle est calculable en temps polynômial, en $\mathcal{O}(n^3)$.)

EXEMPLE :
▷ TD 15, exercice 1.

EXEMPLE :
On considère l'alphabet $\Sigma = \{C, LP, RP\}$ et les règles de production $\mathrm{S} \to \mathrm{TS} \mid C$ et $\mathrm{T} \to LP \cdot \mathrm{S} \cdot RP$. Codons un programme reconnaissant la grammaire définie par ces règles. On représente Σ par le type `token` où C correspond à `C`, LP à `LP` et RP à `RP`.

```
type token = C | LP | RP
type word = token list

type non_term = S | T

(* closed derivation tree *)
type cdt =
  | Node of non_term * cdt list
  | Leaf of token option


exception Non

let rec parse_s (w: word): cdt * word =
  match w with
  | C ::  w' -> (Node(S, [Leaf(Some C)]), w')
  | _        ->
      let (g, w')  = parse_t w  in
      let (d, w'') = parse_s w' in
      (Node (S, [g; d]), w'')
and parse_t (w: word): cdt * word =
  match w with
  | LP :: w' -> begin
      let (u, w'') = parse_s w' in
      match w'' with
      | RP :: w''' -> (Node (T,
          [Leaf (Some LP); u; Leaf (Some RP)]), w''')
      | _          -> raise Non
    end
  | _ -> raise Non
```

Code 9.1 – Programme reconnaissant une grammaire $\mathcal{G}$

CHAPITRE

10

CONCURRENCE

Sommaire

10.1 Motivation **193**
10.2 *Mutex* **196**
10.2.1 À deux fils d'exécutions 196
10.2.2 À N fils d'exécutions 197
10.3 Sémaphore **198**

10.1 Motivation

En place au centre de la classe 40 bonbons. On en distribue un chacun. Si, par exemple, chacun choisit un bonbon et, au *top* départ, prennent celui choisi. Il est probable que plusieurs choisissent le même. Comme gérer lorsque plusieurs essaient d'accéder à la mémoire ?

Deuxièmement, sur l'ordinateur, plusieurs applications tournent en même temps. Pour le moment, on considérait qu'un seul programme était exécuté, mais, le PC ne s'arrête pas pendant l'exécution du programme.

On s'intéresse à la notion de « processus » qui représente une tâche à réaliser. On ne peut pas assigner un processus à une unité de calcul, mais on peut « allumer » et « éteindre » un processus. Le programme allumant et éteignant les processus est « l'ordonnanceur. » Il doit aussi s'occuper de la mémoire du processus (chaque processus à sa mémoire séparée).

On s'intéresse, dans ce chapitre, à des programmes qui « partent du même » : un programme peut créer un « fil d'exécution » (en anglais, *thread*). Le programme peut gérer les fils d'exécution qu'il a créé, et éventuellement les arrêter. Les fils d'exécutions partagent la mémoire du programme qui les a créé.

En C, une tâche est représenté par une fonction de type `void* tache(void* arg)`. Le type `void*` est l'équivalent du type `'a` : on peut le *cast* à un autre type (comme `char*`).

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#include "common.h"
#include "common_thread.h"

```

```c
void* tache(void* arg) {
  printf("%s\n", (char*) arg);
  return NULL;
}

int main() {
  pthread_t p1, p2;

  printf("main: begin\n");

  pthread_create(&p1, NULL, tache, "A");
  pthread_create(&p2, NULL, tache, "B");

  pthread_join(p1, NULL);
  pthread_join(p2, NULL);

  printf("main: end\n");

  return 0;
}
```

Code 10.1 – Création de *thread*s en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#include "common.h"
#include "common_thread.h"

int max = 10;
volatile int counter = 0;

void* tache(void* arg) {
  char* letter = arg;
  int i;

  printf("%s begin [addr of i: %p] \n", letter, &i);

  for(i = 0; i < max; i++) {
    counter = counter + 1;
  }

  printf("%s : done\n", letter);
  return NULL;
}

int main() {
  pthread_t p1, p2;

  printf("main: begin\n");

  pthread_create(&p1, NULL, tache, "A");
  pthread_create(&p2, NULL, tache, "B");

  pthread_join(p1, NULL);
  pthread_join(p2, NULL);

  printf("main: end\n");

  return 0;
}
```

Code 10.2 – Mémoire dans les *thread*s en C

Dans les *thread*s, les variables locales (comme `i`) sont séparées en mémoire. Mais, la variable `counter` est modifiée, mais elle ne correspond pas forcément à $2 \times$ `max`. En effet, si `p1` et `p2` essaient d'exécuter au même moment de réaliser l'opération `counter = counter + 1`, ils peuvent récupérer deux valeurs identiques de `counter`, ajouter 1, puis réassigner `counter`. Ils « se marchent sur les pieds. »

Parmi les opérations, on distingue certaines dénommées « atomiques » qui ne peuvent pas

être séparées. L'opération `i++` n'est pas atomique, mais la lecture et l'écriture mémoire le sont.

Définition : On dit d'une variable qu'elle est *atomique* lorsque l'ordonnanceur ne l'interrompt pas.

Exemple :
L'opération `counter = counter + 1` exécutée en série peut être représentée comme ci-dessous. Avec `counter` valant 40, cette exécution donne 42.

Exécution du fil A	Exécution du fil B
(1) $\text{reg}_1 \leftarrow$ `counter`	(4) $\text{reg}_2 \leftarrow$ `counter`
(2) $\text{reg}_1{+}{+}$	(5) $\text{reg}_2{+}{+}$
(3) `counter` $\leftarrow \text{reg}_1$	(6) `counter` $\leftarrow \text{reg}_2$

Mais, avec l'exécution en simultanée, la valeur de `counter` sera 41.

Exécution du fil A	Exécution du fil B
(1) $\text{reg}_1 \leftarrow$ `counter`	(2) $\text{reg}_2 \leftarrow$ `counter`
(3) $\text{reg}_1{+}{+}$	(5) $\text{reg}_2{+}{+}$
(4) `counter` $\leftarrow \text{reg}_1$	(6) `counter` $\leftarrow \text{reg}_2$

Il y a *entrelacement* des deux fils d'exécution.

Remarque (Problèmes de la programmation concurrentielle) :
- Problème d'accès en mémoire,
- Problème du rendez-vous,[a]
- Problème du producteur-consommateur,[b]
- Problème de l'entreblocage,[c]
- Problème famine, du dîner des philosophes.[d]

a. Lorsque deux programmes terminent, ils doivent s'attendre pour donner leurs valeurs.
b. Certains programmes doivent ralentir ou accélérer.
c. *c.f.* exemple ci-après.
d. Les philosophes mangent autour d'une table, et mangent du riz avec des baguettes. Ils décident de n'acheter qu'une seule baguette par personne. Un philosophe peut, ou penser, ou manger. Mais, pour manger, ils ont besoin de deux baguettes. S'ils ne mangent pas, ils meurent.

Exemple (Problème de l'entreblocage) :

Fil A	Fil B	Fil C
RDV(C)	RDV(A)	RDV(B)
RDV(B)	RDV(C)	RDV(A)

Table 10.1 – Problème de l'entreblocage

Comment résoudre le problème des deux incrementations ? Il suffit de « mettre un verrou. » Le premier fil d'exécution « s'enferme » avec l'expression `count++`, le second fil d'exécution attend que l'autre sorte pour pouvoir entrer et s'enfermer à son tour.

10.2 *Mutex*

Le mot *mutex* vient de *mutual exception* (exclusion mutuelle).

Définition :

Le type de données abstrait verrou fournit

- un type `t` : le type des verrous,
- une fonction `lock : t → unit` qui ferme le verrou,
- une fonction `unlock : t → unit` qui ouvre le verrou,
- une fonction `create : ( ) → t` qui crée un verrou,

```
lock(v);
/* section critique */
unlock(v);
```

de sorte que, lorsque plusieurs fils d'exécution exécutent de manière concurrent l'algorithme ci-dessous, on ait

1. au plus un seul fil d'exécution dans la section critique (exclusion mutuelle) ;
2. un fil d'exécution en attente (ayant appelé la fonction `lock`) n'empêche pas d'autres fils d'exécutions d'accéder à la section critique ;
3. un fil d'exécution ayant fini `lock` aura accès à un moment à la section critique.

10.2.1 À deux fils d'exécutions

On se restreint, pour simplifier, dans le cas où l'on n'a que deux fils d'exécutions.

Algorithme 10.1 Tentative 1 d'implémentation du type verrou

```
1 : Soit Dedans un tableau de taille 2 initialisé à F.
2 : Procédure Lock(i)        ▷ i ∈ {0,1} est l'identifiant du fil
3 :     o ← 1 − i            ▷ identifiant de l'autre fil d'exécution
4 :     tant que Dedans[o] faire
5 :         rien
6 :     Dedans[i] ← V
7 : Procédure Unlock(i)
8 :     Dedans[i] ← F
```

Mais, cet tentative ne résout pas le problème. En effet, l'exécution où l'ordonnanceur exécute l'algorithme jusqu'à la fin de tant que pour le fil 1, puis passe au fil 2, est un contre-exemple à la tentative 1 d'implémentation du type verrou. ▷ propriété d'exclusion mutuelle

Algorithme 10.2 Tentative 2 d'implémentation du type verrou

```
1 : Soit Want un tableau de taille 2 initialisé à F.
2 : Procédure Lock(i)        ▷ i ∈ {0,1} est l'identifiant du fil
3 :     o ← 1 − i            ▷ identifiant de l'autre fil d'exécution
4 :     Want[i] ← V
5 :     tant que Want[o] faire
6 :         rien
7 : Procédure Unlock(i)
8 :     Want[i] ← F
```

Cette tentative ne résout pas non plus le programme : si l'ordonnanceur exécute le fil 1 jusqu'à avant la boucle tant que, puis passe au fil 2, les deux fils sont bloqués. ▷ propriété de non-entreblocage

Algorithme 10.3 Tentative 3 d'implémentation du type verrou

```
1 : turn ← 0        ▷ premier fil d'exécution
2 : Procédure Lock(i)
3 :     o ← 1 − i
4 :     tant que turn = o faire
5 :         rien
6 : Procédure Unlock(i)
7 :     o ← 1 − i
8 :     turn ← o
```

Cette tentative impose une alternance 0 puis 1 puis 0… Mais, si l'un des deux fils termine, alors l'autre est bloqué.

Dans la suite, on suppose qu'un fil ne meure pas dans la section critique, ou lors de l'appel de Lock(i), ou lors de l'appel de Unlock(i).

On donne donc la tentative finale, ci-dessous, qui réussi à combler toutes les hypothèses du type verrou.

Algorithme 10.4 Tentative 4 d'implémentation du type verrou – Algorithme de Peterson

```
1 : turn ← 0        ▷ premier fil d'exécution
2 : Soit Want un tableau de taille 2 initialisé à F.
3 : Procédure Lock(i)
4 :     o ← 1 − i
5 :     Want[i] ← V
6 :     turn ← o
7 :     tant que turn = o et Want[o] faire
8 :         rien
9 : Procédure Unlock(i)
10 :    Want[i] ← F
```

On vérifie que les trois propriétés du type verrou sont vérifiées.

1. **Exclusion mutuelle.** Par l'absurde, supposons que les deux fils d'exécutions sont dans la section critique. Ainsi, $\text{Want}[0] = \text{Want}[1] = \boldsymbol{V}$. Le fil d'exécution 0 nous donne que $\text{Want}[1] = \boldsymbol{F}$ ou $\text{turn} \neq 1$. Le fil d'exécution 1 nous donne que $\text{Want}[0] = \boldsymbol{F}$ ou $\text{turn} \neq 0$. On en déduit que $\text{turn} \neq 0$ et $\text{turn} \neq 1$. Or, $\text{turn} \in \{0, 1\}$ (on peut le montrer par un rapide invariant). D'où l'absurdité.

2. **Non-interblocage.** Si les deux conditions de boucles sont vraies, alors $\text{turn} = 0$ et $\text{turn} = 1$, ce qui est absurde.

3. **Résilience à la mort de l'autre fil d'exécution.**[1] Supposons que le fil d'exécution n'exécute plus Lock(1) (mais il a exécuté Unlock(1) avant de partir). Alors, $\text{Want}[1] = \boldsymbol{F}$, et ce pour toujours. Donc, le fil 0 n'est pas bloqué.

10.2.2 À N fils d'exécutions

On propose de l'algorithme de la boulangerie. On se donne un « ticket » qui donne l'ordre de passage. En voulant accéder à la boulangerie, on prend un ticket (1 plus la valeur maximale des tickets), et on attend son tour Pour attendre son tour, on attend que chacune des personnes n'ai un ticket inférieur au sien. Ceci donne l'algorithme suivant.

1. *i.e.* accès peu importe si l'autre est encore en vie ou non.

Algorithme 10.5 Tentative 1 d'implémentation du type verrou à N fils

```
1 : Ticket est un tableau de n entiers initialisés à 0.
2 : Procédure Lock(Ticket, i)
3 :     Ticket[i] ← 1 + max{Ticket[j] | j ∈ ⟦0, n − 1⟧}
4 :     pour j ∈ ⟦0, n − 1⟧ faire
5 :         tant que Ticket[i] ≠ 0 et Ticket[j] < Ticket[i] faire
6 :             rien
7 : Procédure Unlock(Ticket, i)
8 :     Ticket[i] ← 0
```

Mais, cet algorithme peuvent prendre un ticket pendant le calcul du max. On utilise une variable EnCalcul qui dit lorsque le calcul du max est en cours.

Algorithme 10.6 Tentative 2 d'implémentation du type verrou à N fils

```
1 : Ticket est un tableau de n entiers initialisés à 0.
2 : EnCalcul est un tableau de n booléens initialisés à F.
3 : Procédure Lock(Ticket, i)
4 :     EnCalcul[i] ← V
5 :     Ticket[i] ← 1 + max{Ticket[j] | j ∈ ⟦0, n − 1⟧}
6 :     EnCalcul[i] ← F
7 :     pour j ∈ ⟦0, n − 1⟧ faire
8 :         tant que EnCalcul[j] faire
9 :             rien
10 :        tant que Ticket[i] ≠ 0 et Ticket[j] < Ticket[i] faire
11 :            rien
12 : Procédure Unlock(Ticket, i)
13 :    Ticket[i] ← 0
```

Mais, cet algorithme laisse avoir deux personnes ayant un ticket de même valeur. On peut départager les deux personnes avec les identifiants.

Algorithme 10.7 Tentative 3 d'implémentation du type verrou à N fils – algorithme de la boulangerie

```
1 : Ticket est un tableau de n entiers initialisés à 0.
2 : EnCalcul est un tableau de n booléens initialisés à F.
3 : Procédure Lock(Ticket, i)
4 :     EnCalcul[i] ← V
5 :     Ticket[i] ← 1 + max{Ticket[j] | j ∈ ⟦0, n − 1⟧}
6 :     EnCalcul[i] ← F
7 :     pour j ∈ ⟦0, n − 1⟧ faire
8 :         tant que EnCalcul[j] faire
9 :             rien
10 :        tant que Ticket[j] ≠ 0 et [Ticket[j] < Ticket[i] ou (Ticket[j] = Ticket[i] et j < i)] faire
                                    ordre lexicographique
11 :            rien
12 : Procédure Unlock(Ticket, i)
13 :    Ticket[i] ← 0
```

Cet algorithme assure l'exclusion mutuelle. On peut penser que ce problème peut créer le problème de famine. Mais, non, la comparaison entre identifiants n'a lieu qu'en cas d'égalité de ticket, donc lorsque deux personnes arrivent en même temps à la boulangerie.

10.3 Sémaphore

Un sémaphore est utilisé pour assurer une propriété moins forte que le *mutex* : on assure qu'il n'y a pas « trop » de personnes dans la zone critique. On fixe un nombre maximal de fils

qui sont dans la zone critique et on évite un « flot » de personnes ininterrompu dans la zone critique. Par exemple, en TP, on n'a qu'un nombre limité d'ordinateurs. À l'entrée de la salle, on met à disposition les ordinateurs et chacun dépose le sien lorsqu'il a fini de l'utiliser.

Définition : Le type de données abstrait sémaphore fournit

- le type `t` des sémaphores,
- une fonction d'acquisition du sémaphore `acquire` : $\texttt{t} \to ()$,
- une fonction de libération du sémaphore `release` : $\texttt{t} \to ()$,
- une fonction de création/d'initialisation du sémaphore `make` : $\mathbb{N} \to \texttt{t}$,

tels que

- lors d'une tentative d'acquisition du sémaphore : si le compteur du sémaphore est nul, alors le fil d'exécution courant est mis en attente ; sinon, le compteur est décrémenté et le fil d'exécution peut continuer ;
- lors de la libération du sémaphore : si un fil d'exécution est en attente, on le laisse continuer son exécution ; sinon, on incrémente le compteur.

EXEMPLE :
On considère deux « types » de personnes. Les *producteurs* qui peuvent écrire une donnée. Les *consommateurs* qui récupère une donnée (qui consomme une donnée). En tant que métaphore, on considère que les données sont des pommes. Le producteur ne doit pas produire trop de pommes, le consommateur doit avoir des pommes à consommer. On considère, à présent, l'algorithme ci-dessous.

Algorithme 10.8 Utilisation de la structure SÉMAPHORE dans une problématique producteur/consommateur

```
1 : Procédure Producteur
2 :     acquire plein
3 :     lock()
4 :     Produit une pomme.
5 :     unlock()
6 :     release vide

7 : Procédure Consommateur
8 :     acquire vide
9 :     lock()
10 :    Mange une pomme.
11 :    unlock()
12 :    release plein

13 : vide ← make 0
14 : plein ← make nb_pommes
15 : pour i ∈ ⟦1, n⟧ faire
16 :     Soit un nouveau consommateur.   ▷ i.e. un fil d'exécution qui exécute la procédure Consommateur
17 : pour j ∈ ⟦1, m⟧ faire
18 :     Soit un nouveau producteur.   ▷ i.e. un fil d'exécution qui exécute la procédure Producteur
```

Partie II

Travaux Dirigés

TRAVAUX DIRIGÉS

1

ORDRE & INDUCTION

Sommaire

TD 1.1 Listes, listes, listes ! **203**
TD 1.2 Ensembles définis inductivement **204**
TD 1.3 Arbres, Arbres, Arbres ! **204**
TD 1.4 Ordre sur *powerset* **205**
TD 1.5 Ordres bien fondés en vrac **206**
TD 1.6 Définition inductive des mots et ordre préfixe **206**
TD 1.7 $\mathcal{N}$ **206**
TD 1.8 Résultats manquants du cours **207**

TD 1.1 Listes, listes, listes !

1. On a $\forall \ell \in \mathcal{L},\ @([\,], \ell) = \ell$; $\forall \ell_1, \ell \in \mathcal{L},\ @\big(::(x, \ell_1), \ell\big) = ::\big(x, @(\ell_1, \ell)\big)$.

2. On fait une induction. Comme dans l'énoncé, on passe le '@' en infixe. Notons P_ℓ : "ℓ @ $[\,] = \ell$".

   Montrons $P_{[\,]}$: on sait que $[\,]$ @ $[\,] = [\,]$ par définition de @.

   On suppose P_ℓ est vraie pour une certaine liste $\ell \in \mathcal{L}$. Montrons que, $\forall x \in \mathbb{N}$, $P_{::(x,\ell)}$ vrai. Soit $x \in \mathbb{N}$.

   $$\big(::(x, \ell)\big) \text{ @ } [\,] \overset{\text{(def)}}{=} ::(x, \ell \text{ @ } [\,]) \overset{(P_\ell)}{=} ::(x, \ell).$$

3. Notons P_{ℓ_1} : "$\forall \ell_2, \ell_3 \in \mathcal{L},\ (\ell_1 \text{ @ } \ell_2) \text{ @ } \ell_3 = \ell_1 \text{ @ } (\ell_2 \text{ @ } \ell_3)$", où $\ell_1 \in \mathcal{L}$ est une liste.

   Soient $\ell_2, \ell_3 \in \mathcal{L}$ deux listes. On a, par définition de @, $([\,] \text{ @ } \ell_2) \text{ @ } \ell_3 = \ell_2 \text{ @ } \ell_3$ et $[\,] \text{ @ } (\ell_2 \text{ @ } \ell_3) = \ell_2 \text{ @ } \ell_3$.

   Soit $\ell_1 \in \mathcal{L}$ une liste telle que P_{ℓ_1}. Soient $\ell_2, \ell_2 \in \mathcal{L}$ deux listes. Soit $x \in \mathbb{N}$. Montrons

que $P(::(x,\ell_1))$:

$$\begin{aligned}(::(x,\ell_1) \mathbin{@} \ell_2) \mathbin{@} \ell_3 &= ::(x, \ell_1 \mathbin{@} \ell_2) \mathbin{@} \ell_3 \\ &= ::\big(x, (\ell_1 \mathbin{@} \ell_2) \mathbin{@} \ell_3\big) \\ &\overset{(H)}{=} ::\big(x, \ell_1 \mathbin{@} (\ell_2 \mathbin{@} \ell_3)\big) \\ &= ::(x,\ell_1) \mathbin{@} (\ell_2 \mathbin{@} \ell_3).\end{aligned}$$

4. Notons P_{ℓ_1} : "$\forall \ell_2 \in \mathscr{L}$, $\texttt{rev}(\ell_1 \mathbin{@} \ell_2) = \texttt{rev}(\ell_2) \mathbin{@} \texttt{rev}(\ell_1)$. Soit $\ell_2 \in \mathscr{L}$.
On a $\texttt{rev}([\,] \mathbin{@} \ell_2) = \texttt{rev}(\ell_2) = \texttt{rev}(\ell_2) \mathbin{@} \texttt{rev}([\,])$.
On suppose P_{ℓ_1} vraie pour une certaine liste $\ell_1 \in \mathscr{L}$. Soit $x \in \mathbb{N}$.

$$\begin{aligned}\texttt{rev}(::x,\ell_1) \mathbin{@} \ell_2) &= \texttt{rev}(::(x, \ell_1 \mathbin{@} \ell_2) \\ &= \texttt{rev}(\ell_1 \mathbin{@} \ell_2) \mathbin{@} ::(x,[\,]) \\ &= \big(\texttt{rev}(\ell_2) \mathbin{@} \texttt{rev}(\ell_1)\big) \mathbin{@} ::(x,[\,]) \\ &= \texttt{rev}(\ell_2) \mathbin{@} \big(\texttt{rev}(\ell_1) \mathbin{@} ::(x,[\,])\big) \\ &= \texttt{rev}(\ell_2) \mathbin{@} \texttt{rev}\big(::(x,\ell_1)\big)\end{aligned}$$

5. Notons, pour toute liste $\ell \in \mathscr{L}$, P_ℓ : "$\texttt{rev}\big(\texttt{rev}(\ell)\big) = \ell$".
Montrons que $P_{[\,]}$ est vraie : $\texttt{rev}(\texttt{rev}([\,])) = \texttt{rev}([\,]) = [\,]$.
Soit une liste $\ell \in \mathscr{L}$ telle que P_ℓ soit vraie. Soit $n \in \mathbb{N}$. Montrons que $P_{::(x,\ell)}$ vraie :

$$\begin{aligned}\texttt{rev}(\texttt{rev}(::(x,\ell))) &= \texttt{rev}(\texttt{rev}(\ell) \mathbin{@} ::(x,[\,])) \\ &= \texttt{rev}(::(x,[\,])) \mathbin{@} ::(x,\ell) \mathbin{@} \ell \\ &= [\,] \mathbin{@} ::(x,[\,]) \mathbin{@} \ell \\ &= ::(x,[\,]) \mathbin{@} \ell \\ &= ::(x,\ell).\end{aligned}$$

TD 1.2 Ensembles définis inductivement

La correction est disponible sur *cahier-de-prepa*.

TD 1.3 Arbres, Arbres, Arbres !

1. On pose $R = \left\{ V\big|^0, N\big|^2_{\mathbb{N}} \right\}$. Ainsi, par induction nommée, on crée l'ensemble $\mathscr{A}$ des arbres.
2. On pose

$$\begin{aligned}h : \mathscr{A} &\longrightarrow \mathbb{N} \cup \{-1\} \\ V &\longmapsto -1 \\ N(x,f_1,f_2) &\longmapsto 1 + \max\big(h(f_1), h(f_2)\big)\end{aligned}$$

et

$$\begin{aligned}t : \mathscr{A} &\longrightarrow \mathbb{N} \\ V &\longmapsto 0 \\ N(x,f_1,f_2) &\longmapsto 1 + t(f_1) + t(f_2)\end{aligned}$$

3. On rappelle les relations taille/hauteur (vues l'année dernière) :

$$h(a) + 1 \leqslant t(a) \leqslant 2^{h(a)+1} - 1.$$

Soit, pour tout arbre $a \in \mathcal{A}$, P_a la propriété ci-dessus. Montrons que P_a est vraie pour tout arbre $a \in \mathcal{A}$ par induction.
Montrons que P_V vraie : on a $h(V)+1 = 1-1 = 0$, $t(V) = 0$ et $2^{h(V)+1} - 1 = 1-1 = 0$ d'où $h(V) + 1 \leqslant t(V) \leqslant 2^{h(V)+1} - 1$.
Supposons P_g vraie et P_d vraie pour deux arbres $g, d \in \mathcal{A}$. Soit $x \in \mathbb{N}$. Montrons que $P_{N(x,g,d)}$ est vraie :

$$\begin{aligned} h\big(N(x,g,d)\big) - 1 &= 1 + \max\big(h(g), h(d)\big) + 1 \\ &\leqslant \max(t(g) - 1, t(d) - 1) + 2 \\ &\leqslant \max\big(t(g), t(d)\big) + 1 \\ &\leqslant t(g) + t(d) + 1 \\ &= t\big(N(x,g,d)\big) \end{aligned}$$

et

$$\begin{aligned} t\big(N(x,g,d)\big) &= t(g) + t(d) + 1 \\ &\leqslant 2^{h(g)+1} + 2^{h(d)+1} - 1 \\ &\leqslant 2 \times 2^{\max(h(g),h(d))+1} - 1 \\ &\leqslant 2^{\max(h(g),h(d))+2} - 1 \\ &\leqslant 2^{h(N(x,g,d))+1} - 1. \end{aligned}$$

4. Je pense qu'il y a une erreur d'énoncé : les arbres crées sont de la forme

où □ représente un nœud. Il ne sont pas de la forme "peigne."

TD 1.4 Ordre sur *powerset*

1. Soient A et B deux parties d'un ensemble ordonné $(S, \preccurlyeq)$. Si $A = B$, alors $A \dot{\preccurlyeq} B$ et donc A et B sont comparables. Si $A \neq B$, alors $A \triangle B \neq \varnothing$, et donc $A \triangle B$ admet un plus petit élément m. Par définition de $\triangle$, on a $m \in A$ (et donc $A \dot{\succcurlyeq} B$) ou $m \in B$ (et donc $A \dot{\preccurlyeq} B$). On en déduit que A et B sont comparables. La relation $\dot{\preccurlyeq}$ est donc totale.
2. On a
$$\varnothing \dot{\preccurlyeq} \{2\} \dot{\preccurlyeq} \{1\} \dot{\preccurlyeq} \{1,2\} \dot{\preccurlyeq} \{0\} \dot{\preccurlyeq} \{0,2\} \dot{\preccurlyeq} \{0,1\} \dot{\preccurlyeq} \{0,1,2\}.$$
3. Non, l'ordre $(\wp(S), \dot{\preccurlyeq})$ n'est pas forcément bien fondé. Par exemple, on pose $(S, \preccurlyeq) = (\mathbb{N}, \leqslant)$. Toute partie non vide de $\mathbb{N}$ admet bien un plus petit élément. Mais, la suite $(u_n)_{n\in\mathbb{N}} = (\{n\})_{n\in\mathbb{N}}$ est strictement décroissante : en effet, pour $n \in \mathbb{N}$, on a $u_n \triangle u_{n+1} = \{n, n+1\}$ qui admet pour élément minimal $n \in A$, d'où $u_n \dot{\succcurlyeq} u_{n+1}$.

TD 1.5 Ordres bien fondés en vrac

1. Non, l'ensemble $(\mathbb{N}, \sqsubseteq)$ n'est pas un ensemble ordonné. En effet, en posant $n = 4$ et $m = 8$, on a $\forall i \in \mathbb{N}$, $\frac{n}{2^i} \pmod 2 = 0$, et $\forall i \in \mathbb{N}$, $\frac{m}{2^i} \pmod 2 = 0$. Ainsi, on a $n \sqsubseteq m$, et $m \sqsubseteq n$, mais comme $n \neq m$, la relation "$\sqsubseteq$" n'est pas anti-symétrique, ce n'est donc pas une relation d'ordre (et donc encore moins un ordre bien fondé).
2. Non, l'ensemble $(\Sigma^\star, \sqsubseteq)$ n'est pas un ensemble ordonné. En effet, en posant $\Sigma = \{a, b\}$, et $u = aa$ et $v = ab$ deux mots de Σ, on a $|u| = |v|$ et donc $u \sqsubseteq v$ et $u \sqsupseteq v$ mais comme $u \neq v$, la relation "$\sqsubseteq$" n'est pas anti-symétrique, ce n'est donc pas une relation d'ordre (et donc encore moins un ordre bien fondé).
3. Oui, l'ensemble $(\Sigma^\star, \sqsubseteq)$ est un ensemble ordonné, et cet ordre est total. En effet, soit u, v et w trois mots. On a bien $u \sqsubseteq u$ (avec $\phi = \mathrm{id}_{[\![0,|u|-1]\!]}$). Également, si $u \sqsubseteq v$ et $v \sqsubseteq u$, alors $|u| = |v|$, et par stricte croissance de ϕ, on a bien $u = v$. Aussi, si $u \sqsubseteq v$ et $v \sqsubseteq w$, alors soit ϕ l'extractrice de la suite v de u, et soit φ l'extractrice de la suite w de v. Alors, la fonction $\phi \circ \varphi$ est strictement croissante, et $\forall i \in [\![|u| - 1]\!]$, $u_i = v_{\phi(i)} = w_{\phi(\varphi(i))}$, et donc $u \sqsubseteq w$. Ainsi, la relation "$\sqsubseteq$" est une relation d'ordre.
   Montrons à présent que l'ordre est bien fondé. Soit L une partie non vide de $\Sigma^\star$. Soit $x \in L$. Si $\varepsilon \in L$, alors $x \sqsupseteq \varepsilon$.
4. Non, l'ensemble $(\wp(E), \sqsubseteq)$ n'est pas un ensemble ordonné. En effet, on pose $E = \mathbb{N}$. Soit A une partie finie de E. On sait que son plus grand élément existe, et on le note m. Par définition du maximum, $\forall y \in A$, $y \preccurlyeq m$. Et donc $A \not\sqsupseteq A$. La relation "$\sqsubseteq$" n'est donc pas une relation d'ordre (et donc encore moins un ordre bien fondé).

TD 1.6 Définition inductive des mots et ordre préfixe

1. On pose $X_0 = \{\varepsilon\}$, et pour $n \in \mathbb{N}$,
$$X_{n+1} = X_n \cup \Big(\bigcup_{a \in \Sigma} \{a \cdot w \mid w \in X_n\} \Big).$$
   Ainsi, on définit par induction l'ensemble des mots $\Sigma^\star$.
2. Soient u et v deux mots. Montrons $u \preccurlyeq_1 v \iff u \preccurlyeq_2 v$.

" $\Longrightarrow$ " Supposons $u \preccurlyeq_1 v$. Soit $w \in \Sigma^\star$ tel que $v = uw$. Par définition de $\preccurlyeq_2$, on a bien $\varepsilon \preccurlyeq_2 w$. On décompose u en $u = u_1 \cdot u_2 \cdot \ldots \cdot u_n \cdot \varepsilon$ (avec la définition de mot de la question précédente). D'où, toujours par définition de $\preccurlyeq_2$, on a $u_n \cdot \varepsilon \preccurlyeq_2 u_n \cdot w$, puis $u_{n-1} \cdot u_n \cdot \varepsilon \preccurlyeq_2 u_{n-1} \cdot u_n \cdot w$. En itérant ce procédé, on obtient
$$\underbrace{u_1 \cdot u_2 \cdot \ldots \cdot u_n \cdot \varepsilon}_{u} \preccurlyeq_2 \overbrace{\underbrace{u_1 \cdot u_2 \cdot \ldots \cdot u_n}_{u} \cdot w}^{v}.$$
Et donc $u \preccurlyeq_2 v$.

" $\Longleftarrow$ " Supposons à présent que $u \preccurlyeq_2 v$. On pose $u = u_1 u_2 \ldots u_n$, et $v = v_1 v_2 \ldots v_m$. Par définition de $\preccurlyeq_2$, on a $u_1 \cdot (u_2 \ldots u_n) \preccurlyeq_2 u_1 \cdot (v_2 \ldots v_m)$. Puis, toujours par définition de $\preccurlyeq_2$, on a $u_1 \cdot u_2 \cdot (u_3 \ldots u_n) \preccurlyeq u_1 \cdot u_2 \cdot (v_3 \ldots v_n)$. En itérant ce procédé, on a $u \cdot \varepsilon \preccurlyeq u \cdot (v_n \ldots v_m)$. On pose $w = v_n \ldots v_m$, et on a bien $v = uw$. D'où $v \succcurlyeq_1 u$.

TD 1.7 $\mathcal{N}$

1. On définit par induction la fonction suivante
$$\begin{aligned} \oplus : \mathcal{N}^2 &\longrightarrow \mathcal{N} \\ (\boldsymbol{S}(x), y) &\longmapsto \oplus(x, \boldsymbol{S}(y)) \\ (\boldsymbol{0}, x) &\longmapsto x. \end{aligned}$$
2. Soit $(x, y) \in \mathcal{N}^2$.

— Si $f(x) = 0$, alors $\oplus(x, y) = y$ et donc $f(\oplus(x, y)) = f(y) = f(x) + f(y)$.

— Si $f(x) \geqslant 1$, alors $x = \boldsymbol{S}(z)$ avec $z \in \mathcal{N}$. Ainsi, $\oplus(x, y) = \oplus(z, \boldsymbol{S}(y))$. Or, $f(z) = f(x) - 1 \leqslant f(x)$. Et donc, par définition de $\oplus$ puis par hypothèse d'induction, on a $f(\oplus(x, y)) = f(\oplus(z, \boldsymbol{S}(y)) = f(z) + f(\boldsymbol{S}(y))$. On en déduit que $f(\oplus(x, y)) = f(x) - 1 + f(y) + 1 = f(x) + f(y)$.

Par induction, on a bien $\forall (x, y) \in \mathcal{N}^2,\ f\big(\oplus(x, y)\big) = f(x) + f(y)$.

3. On définit par induction la fonction suivante

$$\begin{aligned} \otimes : \mathcal{N}^2 &\longrightarrow \mathcal{N} \\ (\boldsymbol{S}(x), y) &\longmapsto \oplus\big(y, \otimes(x, y)\big) \\ (\mathbf{0}, y) &\longmapsto \mathbf{0}. \end{aligned}$$

4. Soit $(x, y) \in \mathcal{N}^2$.

— Si $f(x) = 0$, alors $\otimes(x, y) = \mathbf{0}$, et donc $f(\otimes(x, y)) = 0 = f(x) \times f(y)$.

— Si $f(x) \geqslant 1$, alors $x = \boldsymbol{S}(z)$ avec $z \in \mathcal{N}$. Ainsi, par définition de $\otimes$, on a $\otimes(x, y) = \oplus(y, \otimes(z, y))$. Or, par hypothèse d'induction, $f(\otimes(z, y)) = f(z) \times f(y)$ (car $f(z) < f(x)$), et donc $f(\otimes(x, y)) = f(y) + f(\otimes(z, y)) = f(y) + f(z) \times f(y) = f(y) \times (1 + f(z)) = f(y) \times f(x)$.

Par induction, on a bien $\forall (x, y) \in \mathcal{N}^2,\ f\big(\otimes(x, y)\big) = f(x) \times f(y)$.

5. On définit par induction la fonction suivante

$$\begin{aligned} \textcircled{!} : \mathcal{N} &\longrightarrow \mathcal{N} \\ \mathbf{0} &\longmapsto \boldsymbol{S}(\mathbf{0}) \\ \boldsymbol{S}(x) &\longmapsto \otimes\big(\boldsymbol{S}(x), \textcircled{!}(x)\big). \end{aligned}$$

6. Soit $x \in \mathcal{N}$.

— Si $f(x) = 0$, alors $\textcircled{!}(x) = \boldsymbol{S}(\mathbf{0})$ par définition, et donc $f(\textcircled{!}(x)) = 1 = 0! = f(x)!$.

— Si $f(x) \geqslant 1$, alors $x = \boldsymbol{S}(z)$ avec $z \in \mathcal{N}$. Ainsi, par définition de $\textcircled{!}$, on a $\textcircled{!}(x) = \otimes(x, \textcircled{!}(z))$, et donc, par hypothèse de récurrence, $f(\textcircled{!}(x)) = f(x) \times f(\textcircled{!}(z)) = f(x) \times \big(f(z)!\big)$. Or, comme $f(z) = f(x) - 1$, on a donc $f(\textcircled{!}(x)) = f(x) \times \big(f(x) - 1\big)! = f(x)!$.

Par induction, on a bien $\forall x \in \mathcal{N},\ f\big(\textcircled{!}(x)\big) = f(x)!$.

TD 1.8 Résultats manquants du cours

TRAVAUX DIRIGÉS

2 LOGIQUE PROPOSITIONNELLE

Sommaire

TD 2.1 Logique avec If **209**
TD 2.1.1 Représentatibilité des fonctions booléennes par formules de $\mathscr{F}_{\texttt{if}}$. 209
TD 2.2 Définitions de cours : syntaxe **210**
TD 2.3 Formules duales **212**
TD 2.4 Conséquence sémantique **212**
TD 2.5 Axiomatisation algèbre de Boole **213**
TD 2.6 Exercice 6 : Barre de Scheffer **213**
TD 2.7 Énigmes en logique propositionnelle **213**
TD 2.7.1 Fraternité 213
TD 2.7.2 Alice au pays des merveilles 214
TD 2.7.3 Socrate et Cerbère 214
TD 2.8 Compléments de cours, en vrac **214**

TD 2.1 Logique avec `If`

TD 2.1.1 Représentatibilité des fonctions booléennes par formules de $\mathscr{F}_{\texttt{if}}$

1. On pose $G = \texttt{if}\ p\ \texttt{then}\ \top\ \texttt{else}\ (\underbrace{\texttt{if}\ q\ \texttt{then}\ \top\ \texttt{else}\ r}_{A})$, et on a

$$\begin{aligned}
[\![G]\!]^\rho &= [\![p]\!]^\rho \cdot [\![\top]\!]^\rho + \overline{[\![p]\!]^\rho} \cdot [\![A]\!]^\rho \\
&= \rho(p) + \overline{\rho(p)} \cdot \left([\![q]\!]^\rho \cdot [\![\top]\!]^\rho + \overline{[\![q]\!]^\rho} \cdot [\![r]\!]^\rho \right) \\
&= \rho(p) + \overline{\rho(p)} \cdot \left(\rho(q) + \overline{\rho(q)} \cdot \rho(r) \right) \\
&= \rho(p) + \overline{\rho(p)} \cdot \rho(q) + \overline{\rho(p)} \cdot \overline{\rho(q)} \cdot \rho(r) \\
&= \rho(p) + \rho(q) + \rho(r).
\end{aligned}$$

2.

$$[\![\texttt{if}\ C\ \texttt{then}\ G\ \texttt{else}\ H]\!]^\rho = \begin{cases} [\![G]\!]^\rho & \text{si } [\![C]\!]^\rho = \boldsymbol{V} \\ [\![H]\!]^\rho & \text{if } [\![C]\!]^\rho = \boldsymbol{F} \end{cases}.$$

3. Soit $G \in \mathbb{F}$.

   Cas 1 Soit $\mathcal{P} = \{p\}$

   — Sous-cas 1 : $f : \rho \mapsto \boldsymbol{V}$ est associée à $\top$.

   — Sous-cas 2 : la fonction dont la table de vérité est ci-dessous est associée à p.

p	f
$\boldsymbol{F}$	$\boldsymbol{F}$
$\boldsymbol{V}$	$\boldsymbol{V}$

   — Sous-cas 3 : la fonction dont la table de vérité est ci-dessous est associée à $\bar{p}$.

p	f
$\boldsymbol{F}$	$\boldsymbol{V}$
$\boldsymbol{V}$	$\boldsymbol{F}$

   — Sous-cas 4 : $f : \rho \mapsto \boldsymbol{F}$ est associée à $\bot$.

   Cas 2 Soit $\mathcal{P} = \{p_1, \dots, p_n\}$ et on pose

   $$P_r : \text{“}\forall f : \mathbb{B}^{\{p_1,\dots,p_r\}} \to \mathbb{B},\ \exists G \in \mathcal{F}_{\texttt{if}},\ [\![G]\!] = f.\text{”}$$

   Soit $r \in [\![2, n]\!]$ et f une fonction booléenne définie sur $\mathbb{B}^{\{p_1,\dots,p_r\}}$ à valeurs dans $\mathbb{B}$. Soit

   $$\begin{aligned} g : \mathbb{B}^{\{p_1,\dots,p_{r-1}\}} &\longrightarrow \mathbb{B} \\ \rho' &\longmapsto f\big(\rho' \uplus (p_r \mapsto \boldsymbol{V})\big). \end{aligned}$$

   où $\uplus$ est défini comme dans l'exemple $(p \mapsto \boldsymbol{V}, q \mapsto \boldsymbol{F}) \uplus (r \mapsto \boldsymbol{V}) = (p \mapsto \boldsymbol{V}, q \mapsto \boldsymbol{F}, r \mapsto \boldsymbol{V})$. Soit alors G par hypothèse de récurrence tel que $[\![G]\!] = g$. Soit

   $$\begin{aligned} h : \mathbb{B}^{\{p_1,\dots,p_{r-1}\}} &\longrightarrow \mathbb{B} \\ \rho' &\longmapsto f\big(\rho' \uplus (p_r \mapsto \boldsymbol{F})\big). \end{aligned}$$

   Soit alors H par hypothèse de récurrence tel que $[\![H]\!] = h$.
   On pose alors $A = \texttt{if } p_r \texttt{ then } G \texttt{ else } H$. Montrons que $[\![A]\!] = f$. Soit $\rho \in \mathbb{B}^{\{p_1,\dots,p_r\}}$.

   — Si $\rho(p_r) = \boldsymbol{V}$ alors

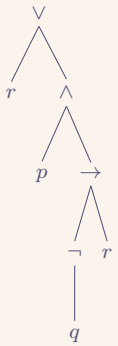
   $$\begin{aligned} [\![A]\!]^\rho &= [\![G]\!]^\rho \\ &= [\![G]\!]^{\rho_{|\{p_1,\dots,p_{r-1}\}}} \\ &= g\left(\rho_{|\{p_1,\dots,p_{r-1}\}}\right) \\ &= f\left(\rho_{|\{p_1,\dots,p_{r-1}\}} \uplus (p_r \mapsto \boldsymbol{V})\right) \\ &= f(\rho) \end{aligned}$$

   — Si $\rho(p_r) = \boldsymbol{F}$, alors

   $$\begin{aligned} [\![A]\!]^\rho &= [\![H]\!]^\rho \\ &= [\![H]\!]^{\rho_{|\{p_1,\dots,p_{r-1}\}}} \\ &= h\left(\rho_{|\{p_1,\dots,p_{r-1}\}}\right) \\ &= f\left(\rho_{|\{p_1,\dots,p_{r-1}\}} \uplus (p_r \mapsto \boldsymbol{F})\right) \\ &= f(\rho) \end{aligned}$$

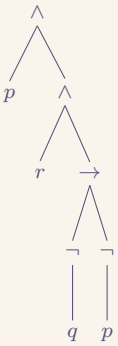
TD 2.2 Définitions de cours : syntaxe

1. On considère la formule $H_1 = r \vee (p \wedge (\neg q \to r))$. Son arbre syntaxique est



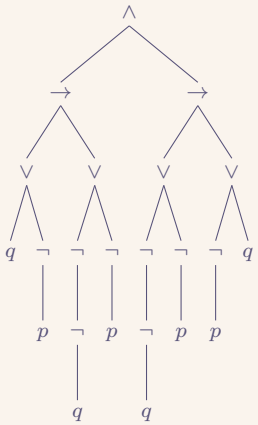
Ses sous-formules sont $r \vee (p \wedge (\neg q \to r))$, r, $p \wedge (\neg q \to r)$, p, $\neg q \to r$, $\neg q$, et q. Ses variables sont p, q et r.

2. On considère la formule $H_2 = p \wedge (r \wedge (\neg q \to \neg p))$. Son arbre syntaxique est



Ses sous-formules sont $p \wedge (r \wedge (\neg q \to \neg p))$, p, $r \wedge (\neg q \to \neg p)$, r, $\neg q \to \neg p$, $\neg q$, q, et $\neg p$. Ses variables sont p, q et r.

3. On considère la formule $H_3 = ((q \vee \neg p) \to (\neg\neg q \vee \neg p)) \wedge ((\neg\neg q \vee \neg p) \to (\neg p \vee q))$. Son arbre syntaxique est



Ses sous-formules sont $((q \vee \neg p) \to (\neg\neg q \vee \neg p)) \wedge ((\neg\neg q \vee \neg p) \to (\neg p \vee q))$, $(q \vee \neg p) \to (\neg\neg q \vee \neg p)$, $(\neg\neg q \vee \neg p) \to (\neg p \vee q)$, $q \vee \neg p$, $\neg\neg q \vee \neg p$, $\neg p \vee q$, q, $\neg p$, $\neg\neg q$, p, et $\neg q$. Ses variables sont p et q.

TD 2.3 Formules duales

1. On définit par induction $(\cdot)^\star$ comme

   — $\top^\star = \bot$; — $(G \vee H)^\star = G^\star \wedge H^\star$; — $(\neg G)^\star = \neg G^\star$;
   — $\bot^\star = \top$; — $(G \wedge H)^\star = G^\star \vee H^\star$; — $p^\star = p$.

2. Soit $\rho \in \mathbb{B}^{\mathscr{P}}$. Montrons, par induction, $P(H)$: " $[\![H^\star]\!]^\rho = [\![\neg H]\!]^{\bar\rho}$ " où $\bar\rho : p \mapsto \overline{\rho(p)}$.
   — On a $[\![\bot^\star]\!]^\rho = [\![\top]\!]^\rho = \boldsymbol{V}$, et $[\![\neg\bot]\!]^{\bar\rho} = [\![\top]\!]^{\bar\rho} = \boldsymbol{V}$, d'où $P(\bot)$.
   — On a $[\![\top^\star]\!]^\rho = [\![\bot]\!]^\rho = \boldsymbol{F}$, et $[\![\neg\top]\!]^{\bar\rho} = [\![\bot]\!]^{\bar\rho} = \boldsymbol{F}$, d'où $P(\top)$.
   — Soit $p \in \mathscr{P}$. On a $[\![p^\star]\!]^\rho = [\![p]\!]^\rho = \rho(p)$, et $[\![\neg p]\!]^{\bar\rho} = \overline{[\![p]\!]^{\bar\rho}} = \overline{\bar\rho(p)} = \overline{\overline{\rho(p)}} = \rho(p)$, d'où $P(p)$.

   Soient F et G deux formules.
   — On a

$$\begin{aligned}[\![(F \wedge G)^\star]\!]^\rho &= [\![F^\star \vee G^\star]\!]^\rho \\ &= [\![F^\star]\!]^\rho + [\![G^\star]\!]^\rho \\ &= [\![\neg F]\!]^{\bar\rho} + [\![\neg G]\!]^{\bar\rho} \\ &= [\![\neg F \vee \neg G]\!]^{\bar\rho} \\ &= [\![\neg(F \wedge G)]\!]^{\bar\rho}\end{aligned}$$

   d'où $P(F \wedge G)$.
   — On a

$$\begin{aligned}[\![(F \vee G)^\star]\!]^\rho &= [\![F^\star \wedge G^\star]\!]^\rho \\ &= [\![F^\star]\!]^\rho \cdot [\![G^\star]\!]^\rho \\ &= [\![\neg F]\!]^{\bar\rho} \cdot [\![\neg G]\!]^{\bar\rho} \\ &= [\![\neg F \wedge \neg G]\!]^{\bar\rho} \\ &= [\![\neg(F \vee G)]\!]^{\bar\rho}\end{aligned}$$

   d'où $P(F \vee G)$.
   — On a

$$[\![(\neg F)^\star]\!]^\rho = [\![\neg(F^\star)]\!]^\rho = \overline{[\![F^\star]\!]^\rho} = \overline{[\![\neg F]\!]^{\bar\rho}} = [\![\neg(\neg F)]\!]^{\bar\rho}.$$

   d'où $P(\neg F)$.

   Par induction, on en conclut que $P(F)$ est vraie pour toute formule F.
3. Soit G une formule valide. Alors, par définition, $G \equiv \top$. Or, d'après la question précédente, $G^\star \equiv (\top)^\star = \bot$. Ainsi, $G^\star$ n'est pas satisfiable.

TD 2.4 Conséquence sémantique

1. Soit $\rho \in \mathbb{B}^{\mathscr{P}}$. On suppose $[\![A \vee B]\!]^\rho = \boldsymbol{V}$, $[\![A \to C]\!]^\rho = \boldsymbol{V}$ et $[\![B \to C]\!]^\rho = \boldsymbol{V}$.
   — Si $[\![A]\!]^\rho = \boldsymbol{V}$, alors, comme $[\![A \to C]\!]^\rho = \boldsymbol{V}$, $[\![C]\!]^\rho = \boldsymbol{V}$.
   — Si $[\![B]\!]^\rho = \boldsymbol{V}$, alors, comme $[\![B \to C]\!]^\rho = \boldsymbol{V}$, $[\![C]\!]^\rho = \boldsymbol{V}$.

   D'où $\{A \vee B, A \to C, B \to C\} \models C$.
2. Soit $\rho \in \mathbb{B}^{\mathscr{P}}$. On suppose $[\![A \to B]\!]^\rho = \boldsymbol{V}$. Si $[\![B]\!]^\rho = \boldsymbol{F}$ (i.e. $[\![\neg B]\!]^\rho = \boldsymbol{V}$), alors $[\![A]\!]^\rho = \boldsymbol{F}$, par implication. Et donc, $[\![\neg A]\!]^\rho = \boldsymbol{V}$. On a donc bien $[\![\neg B \to \neg A]\!]^\rho$. On en déduit que $A \to B \models \neg B \to \neg A$.

3. oui
4. non
5. oui
6. non
7. oui
8. non
9. oui
10. non

TD 2.5 Axiomatisation algèbre de Boole

1. On pose, pour $t \in \mathbb{T}$, $P(t)$: "$t \simeq 0$ ou $t \simeq 1$," et on démontre cette propriété par induction.
   — On a bien $P(\mathbf{0})$ et $P(\mathbf{1})$.
   — Soient $t_1, t_2 \in \mathbb{T}$.
   (a) Si $t_1 \simeq \mathbf{1}$, alors $\bar{t_1} \simeq \bar{\mathbf{1}} \simeq \bar{\bar{\mathbf{0}}} \simeq \mathbf{0}$; si $t_1 \simeq \mathbf{0}$, alors $\bar{t_1} \simeq \bar{\mathbf{0}} \simeq \mathbf{1}$.
   (b) Si $t_1 \simeq \mathbf{0}$, alors $t_1 \cdot t_2 \simeq \mathbf{0} \cdot t_2 \simeq \mathbf{0}$; si $t_1 \simeq \mathbf{1}$, alors $t_1 \cdot t_2 \simeq \mathbf{1} \cdot t_2 \simeq t_2$ (qui est équivalent à $\mathbf{0}$ ou $\mathbf{1}$).
   (c) Si $t_1 \simeq \mathbf{1}$, alors $t_1 + t_2 \simeq \mathbf{1} + t_2 \simeq \mathbf{1}$; si $t_1 \simeq \mathbf{0}$, alors $t_1 + t_2 \simeq \mathbf{0} + t_2 \simeq t_2$ (qui est équivalent à $\mathbf{0}$ ou $\mathbf{1}$)
2. En reprenant les relations trouvées dans la question précédente, on construit les tables ci-dessous.

t_1	t_2	$t_1 \cdot t_2$
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$
$\mathbf{0}$	$\mathbf{1}$	$\mathbf{0}$
$\mathbf{1}$	$\mathbf{0}$	$\mathbf{0}$
$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$

t_1	t_2	$t_1 + t_2$
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$
$\mathbf{0}$	$\mathbf{1}$	$\mathbf{1}$
$\mathbf{1}$	$\mathbf{0}$	$\mathbf{1}$
$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$

t_1	$\bar{t_1}$
$\mathbf{0}$	$\mathbf{1}$
$\mathbf{1}$	$\mathbf{0}$

TD 2.6 Exercice 6 : Barre de Scheffer

1. — $\neg H_1 \equiv H_1 \barwedge H_1$;
   — $H_1 \wedge H_2 \equiv \neg(H_1 \barwedge H_2) \equiv (H_1 \barwedge H_1) \barwedge (H_2 \barwedge H_2)$;
   — $H_1 \vee H_2 \equiv \neg H_1 \barwedge \neg H_2 \equiv (H_1 \barwedge H_1) \barwedge (H_2 \barwedge H_2)$;
   — $H_1 \to H_2 \equiv H_1 \vee (\neg H_2) \equiv (H_1 \barwedge H_1) \barwedge H_2$;
   — $H_1 \leftrightarrow H_2 \equiv (H_1 \to H_2) \wedge (H_2 \to H_1)$.
2. — $\neg H_1 \equiv H_1 \barvee H_1$;
   — $H_1 \vee H_2 \equiv \neg(H_1 \barvee H_2) \equiv (H_1 \barvee H_2) \barvee (H_1 \barvee H_2)$;
   — $H_1 \wedge H_2 \equiv (\neg H_1) \barvee (\neg H_2) \equiv (H_1 \barvee H_1) \barvee (H_2 \barvee H_2)$;
   — $H_1 \to H_2 \equiv H_1 \vee (\neg H_2) \equiv \neg(H_1 \barvee (\neg H_2)) \equiv (H_1 \barvee (H_2 \barvee H_2)) \barvee (H_1 \barvee (H_2 \barvee H_2))$;
   — $H_1 \leftrightarrow H_2 \equiv (H_1 \to H_2) \wedge (H_2 \to H_1)$.

TD 2.7 Énigmes en logique propositionnelle

TD 2.7.1 Fraternité

1. Ou les deux mentent, ou les deux disent la vérité. D'où $(A_1 \wedge C_1) \vee (\neg A_1 \wedge \neg C_1)$.
2. On a $A_1 = G \vee R$, et $C_1 = \neg G$.
3. On développe l'expression trouvée dans la question 1. :

$$\begin{aligned}(A_1 \wedge C_1) \vee (\neg A_1 \wedge \neg C_1) &\equiv \big((G \vee R) \wedge \neg G\big) \vee \big(\neg(G \vee R) \wedge \neg\neg G\big)\\ &\equiv (G \wedge \neg G \vee R \wedge \neg G) \vee \big((\neg G \wedge \neg R) \wedge G\big)\\ &\equiv (\bot \vee R \wedge \neg G) \vee (\neg G \wedge G \wedge \neg R)\\ &\equiv (R \wedge \neg G) \vee (\bot \wedge \neg R)\\ &\equiv (R \wedge \neg G) \vee \bot\\ &\equiv R \wedge \neg G.\end{aligned}$$

La cérémonie se tiendra donc dans le réfectoire et non dans le gymnase.

4. $H = (A_2 \wedge B_2 \wedge C_2) \vee (\neg A_2 \wedge \neg B_2 \wedge \neg C_2)$.
5. $A_2 = E_1 \vee E_3$, $B_2 = E_2 \to \neg E_3$, $C_2 = E_1 \wedge \neg E_2$.
6.

E_1	E_2	E_3	A_2	B_2	C_2	H
F	***F***	***F***	***F***	***V***	***F***	***F***
F	***F***	***V***	***F***	***V***	***F***	***F***
F	***V***	***F***	***F***	***V***	***F***	***F***
F	***V***	***V***	***F***	***F***	***F***	***V***
V	***F***	***F***	***F***	***V***	***V***	***F***
V	***F***	***V***	***V***	***V***	***V***	***V***
V	***V***	***F***	***F***	***V***	***F***	***F***
V	***V***	***V***	***V***	***F***	***F***	***F***

   L'escalier 3 conduit bien à l'intronisation.
7. Si les trois mentent, alors on peut aussi prendre l'escalier 2.

TD 2.7.2 Alice au pays des merveilles

1. $I_R = \bar{J} \wedge B$, $I_J = \bar{R} \to \bar{B}$, et $I_B = B \wedge (\bar{R} \vee \bar{J})$.
2. Oui, la situation où le flacon jaune contient le poison ($\bar{J}$) satisfait ces formules.
3. Oui, on a $I_B \models I_R$.
4. Oui, les instructions sur le flacon rouge et le bleu sont fausses. En effet, le flacon jaune ne contient pas de poison, et il n'y a pas "au moins l'un des deux autres flacons contenant du poison."
5. Oui, comme vu précédemment, la configuration où le poison est seulement dans le flacon jaune est valide. Si le flacon rouge contient du poison ou le flacon bleu contient du poison, on a une contradiction avec l'une des instructions. La configuration trouvée précédemment est la seule valide.
6. À cette condition, deux autres configurations sont possibles : le poison est dans les flacons jaune et rouge, ou le poison est dans les flacons rouge et bleu.

TD 2.7.3 Socrate et Cerbère

1. On a $H = (I_1 \wedge I_2 \wedge I_3) \vee (\neg I_1 \wedge \neg I_2 \wedge \neg I_3)$.
2. On a $I_1 = C_1 \wedge C_3$, $I_2 = C_2 \to \bar{C}_3$, et $I_3 = C_1 \wedge \bar{C}_2$.
3.

C_1	C_2	C_3	I_1	I_2	I_3	H
F	***F***	***F***	***F***	***V***	***F***	***F***
V	***F***	***F***	***F***	***V***	***V***	***F***
F	***V***	***F***	***F***	***V***	***F***	***F***
V	***V***	***F***	***F***	***V***	***F***	***F***
F	***F***	***V***	***F***	***V***	***F***	***F***
V	***F***	***V***	***V***	***V***	***V***	***V***
F	***V***	***V***	***F***	***F***	***F***	***V***
V	***V***	***V***	***V***	***F***	***F***	***F***

   Socrate doit suivre le couloir 3.
4. En supposant que Cerbère ait menti, on suppose que les trois têtes ont menti, et donc que I_1, I_2 et I_3 sont fausses. On aurait aussi pu choisir le couloir 2.

TD 2.8 Compléments de cours, en vrac

1. *Montrer que $\models$ est une relation d'ordre sur $\mathscr{F}$.*
   — Soit F une formule. On sait que $F \models F$. En effet, pour tout $\rho \in \mathbb{B}^{\mathscr{P}}$, si $[\![F]\!]^\rho = \boldsymbol{V}$, alors $[\![F]\!]^\rho = \boldsymbol{V}$. La relation $\models$ est donc reflective.

— Soient F et G deux formules. On suppose $F \models G$ et $G \models H$. Montrons que $F \equiv G$. On reprend la démonstration du cours : soit $\rho \in \mathbb{B}^{\mathscr{P}}$. On suppose $[\![G]\!]^\rho = \boldsymbol{V}$, alors $[\![F]\!] = \boldsymbol{V}$ car $G \models H$; et donc $[\![G]\!]^\rho = [\![F]\!]^\rho$. On suppose à présent que $[\![G]\!]^\rho = \boldsymbol{F}$, alors, par contraposée, $[\![F]\!]^\rho = \boldsymbol{F}$; on a donc $[\![G]\!]^\rho = [\![F]\!]^\rho$. On en déduit que $[\![G]\!] = [\![F]\!]$, i.e. $G \equiv F$. La relation est donc *quasi*–anti-symétrique.

— Soient F, G et H trois formules. On suppose $F \models G$ et $G \models H$. Soit $\rho \in \mathbb{B}^{\mathscr{P}}$. Si $[\![F]\!]^\rho = \boldsymbol{V}$, alors $[\![G]\!]^\rho = \boldsymbol{V}$. Or, comme $G \models H$, si $[\![G]\!]^\rho = \boldsymbol{V}$, alors $[\![H]\!]^\rho = \boldsymbol{V}$. D'où $[\![F]\!]^\rho = \boldsymbol{V} \implies [\![H]\!]^\rho = \boldsymbol{V}$. On a donc $F \models H$. La relation $\models$ est donc transitive.

2. Soit $A \in \mathscr{F}$, soit $\rho \in \mathbb{B}^{\mathscr{P}}$, et soit $\sigma \in \mathscr{P}^{\mathscr{F}}$. On pose, pour $p \in \mathscr{P}$, $\tau(p) = [\![\sigma(p)]\!]^\rho$. *Montrons que* $[\![A[\sigma]]\!]^\rho = [\![A]\!]^\tau$.

TRAVAUX DIRIGÉS

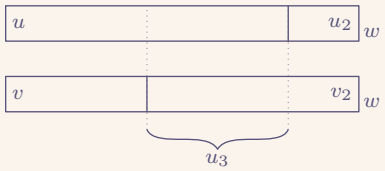
3 LANGAGES ET EXPRESSIONS RÉGULIÈRES

Sommaire

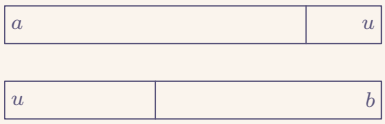
TD 3.1 Propriétés sur les mots 217
TD 3.2 Construction d'automates 218
TD 3.3 Déterminisation 1 219
TD 3.4 Déterminisation 2 220
TD 3.5 Une équivalence sur les mots 220
TD 3.6 Langages 220
TD 3.7 Propriétés sur les opérations régulières 220
TD 3.8 Habitants d'expressions régulières 222
TD 3.9 Regexp Crossword 222
TD 3.10 Description d'automates au moyen d'expression régulières 222
TD 3.11 Vocabulaire des automates 222
TD 3.12 Complétion d'automate 223
TD 3.13 Exercice supplémentaire 1 223

TD 3.1 Propriétés sur les mots

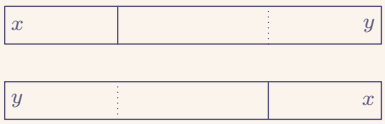
1. Soit $u_2, v_2 \in \Sigma^\star$ tels que $w = uu_2$ et $w = vv_2$. Si $|u_2| = |v_2|$, alors $u = v = v\varepsilon$ donc v est préfixe de u. Si $|u_2| < |v_2|$, u_2 est suffixe de v_2. Soit $u_3 \in \Sigma^\star$ tel que $v_2 = u_2u_3$. Ainsi, $w = vu_3u_2 = uu_2$, d'où $u = vu_3$. On en déduit que v est un préfixe de u. Similairement, si $|u_2| > |v_2|$, par symétrie du problème, en inversant u et v, puis u_2 et v_2, on se trouve bien dans le cas précédent. Ainsi, on a bien u est un préfixe de v.



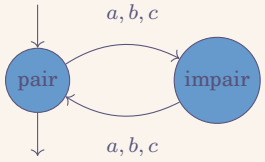
2. Soit $u = u_1 \dots u_n$ avec, pour tout $i \in [\![1, n]\!]$, $u_i \in \Sigma$. Or, $au = ub$ donc $au_1 \dots u_n = u_1 \dots u_n b$ donc, pour tout $i \in [\![1, n-1]\!]$, $u_i = u_{i+1}$. Or, $u_1 = a$. De proche en proche, on a $\forall i \in [\![1, n]\!]\ ,\ u_i = a$. Or, $u_n = b$ et donc $a = b$. On en déduit également que $u \in a^\star$.

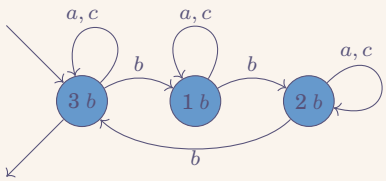


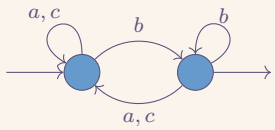
3. La suite de la correction de cet exercice est disponible sur *cahier-de-prepa*.

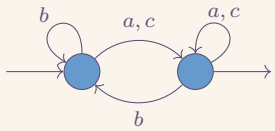


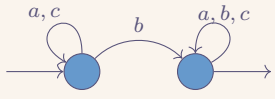
TD 3.2 Construction d'automates

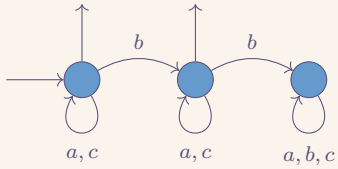
1.


2.


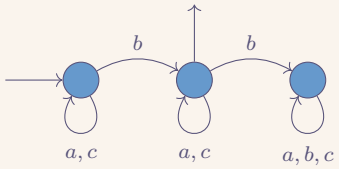
3.


4.


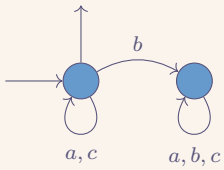
5.


6.


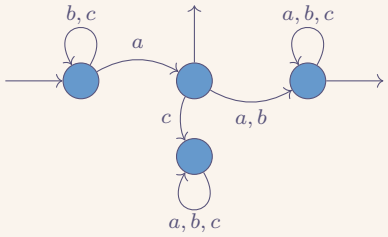
7.



8.

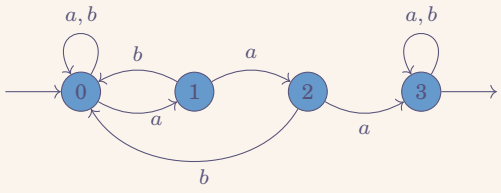


9.

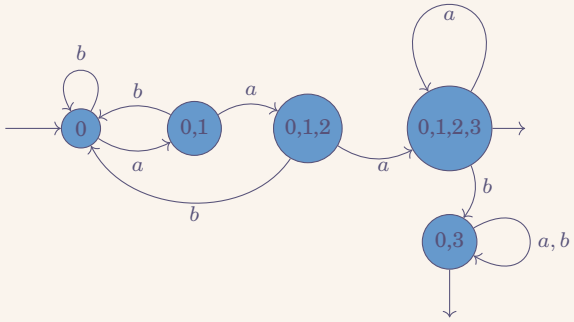


TD 3.3 Déterminisation 1

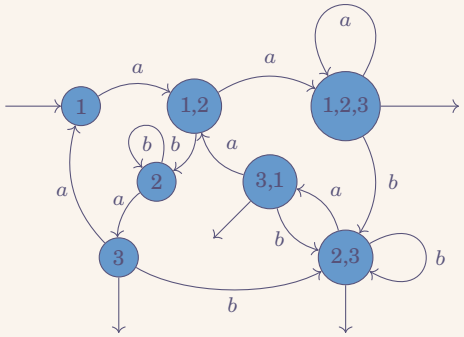
1.

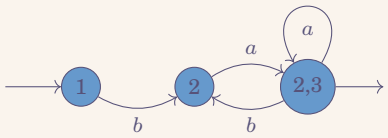


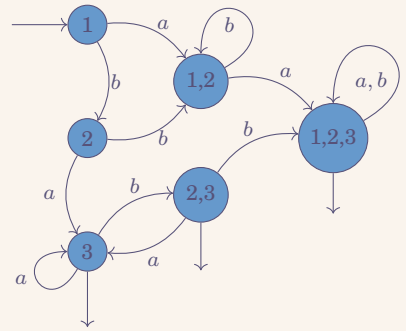
2.



TD 3.4 Déterminisation 2

1.


2.


3.


TD 3.5 Une équivalence sur les mots

La correction de cet exercice est disponible sur *cahier-de-prepa*.

TD 3.6 Langages

La correction de cet exercice est disponible sur *cahier-de-prepa*.

TD 3.7 Propriétés sur les opérations régulières

1. On a
$$\varnothing^\star = \{\varepsilon\} \; ; \qquad \varnothing \cdot A = \varnothing \; ; \qquad \{\varepsilon\} \cdot A = A.$$

2. *(1)* On procède par double-inclusion.

   "$\subseteq$" Soit $w \in (A \cdot B) \cdot C$. On pose $w = u \cdot v$ avec $u \in A \cdot B$ et $v \in C$. On pose ensuite $u = x \cdot y$ avec $x \in A$ et $y \in B$. Or, comme l'opération "$\cdot$," pour les mots, est associative, on a bien $w = (x \cdot y) \cdot v = x \cdot (y \cdot v)$, et donc $w \in A \cdot (B \cdot C)$.

"$\supseteq$" Soit $w \in A \cdot (B \cdot C)$. On pose $w = u \cdot v$ avec $u \in A$ et $v \in B \cdot C$. On pose ensuite $v = x \cdot y$ avec $x \in B$ et $y \in C$. Or, comme l'opération "$\cdot$," pour les mots, est associative, alors $w = u \cdot (x \cdot y) = (u \cdot x) \cdot y$ et donc $w \in A \cdot (B \cdot C)$.

(2) On suppose $A \subseteq B$. On a donc $B = A \cup (B \setminus A)$, et par définition $A^\star = \bigcup_{n\in\mathbb{N}} A^n$, et $B^\star = \bigcup_{n\in\mathbb{N}} B^n$. Montrons par récurrence, pour $n \in \mathbb{N}$, $P(n)$: "$A^n \subseteq B^n$."

— On a $A^0 = \{\varepsilon\} \subseteq B^0 = \{\varepsilon\}$ d'où $P(0)$.

— Soit $n \in \mathbb{N}$ tel que $A^n \subseteq B^n$. On a $A^{n+1} = A^n \cdot A$ et $B^{n+1} = B^n \cdot B$. Or, comme $A^n \subseteq B^n$ et $A \subseteq B$, et que "$\cdot$"est croissant (dans l'inclusion), on en déduit que $A^{n+1} \subseteq B^{n+1}$. D'où $P(n+1)$.

(3) On procède par double-inclusion.

"$\supseteq$" On a $A^\star = (A^\star)^1 \subseteq \bigcup_{n\in\mathbb{N}} (A^\star)^n = (A^\star)^\star$.

"$\subseteq$" Soit $w \in (A^\star)^\star$. On pose donc $w = u_1 \dots u_n$ avec, pour tout $i \in [\![1, n]\!]$, $u_i \in A^\star$. On pose également, pour tout $i \in [\![1, n]\!]$, $u_i = v_{i,1} \dots v_{i,m_i}$ où, pour tout $j \in [\![1, m_i]\!]$, $v_{i,j} \in A$. D'où, $w = v_{11} \dots v_{1,m_1} v_{21} \dots v_{2,m_2} \dots v_{n,m_n} \in A^\star$. On en déduit que $(A^\star)^\star \subseteq A^\star$.

(4) On procède par double-inclusion.

"$\subseteq$" On a $\{\varepsilon\} \subseteq A^\star$ et donc $A^\star = A^\star \cdot \{\varepsilon\} \subseteq A^\star \cdot A^\star$. D'où $A^\star \subseteq A^\star \cdot A^\star$.

"$\supseteq$" Soit $w \in A^\star \cdot A^\star$. On décompose ce mot : soient $u_1, u_2 \in A^\star$ tels que $w = u_1 \cdot u_2$. On pose $n = |u_1|$, et $m = |u_2|$. On décompose également ces deux mots : soient $(w_1, w_2, \dots, w_n) \in A^n$ et $(w_{n+1}, w_{n+2}, \dots, w_{n+m}) \in A^m$ tels que $u_1 = w_1 \cdot w_2 \cdot \dots \cdot w_n$ et $u_2 = w_{n+1} \cdot w_{n+2} \cdot \dots \cdot w_{n+m}$. Ainsi,

$$w = w_1 \cdot w_2 \cdot \dots \cdot w_n \cdot w_{n+1} \cdot \dots \cdot w_{n+m} \in A^\star.$$

D'où $A^\star \cdot A^\star \subseteq A^\star$.

(5) On procède par double-inclusion.

"$\subseteq$" Soit $w \in A \cup B$.

— Si $w \in A$, alors $w \in A^\star$, et donc $w = w \cdot \varepsilon \in A^\star \cdot B^\star$.

— Si $w \in B$, alors $w \in B^\star$, et donc $w = \varepsilon \cdot w \in A^\star \cdot B^\star$.

On a donc bien $A \cup B \subseteq A^\star \cdot B^\star$, et par croissance de l'étoile, on a bien $(A \cup B)^\star \subseteq (A^\star \cdot B^\star)^\star$.

"$\supseteq$" Soit $w \in (A^\star \cdot B^\star)^\star$. On pose

$$\begin{aligned} w = \; & u_{11} \dots u_{1,n_1} v_{11} \dots v_{1,m_1} \\ & \cdot u_{21} \dots u_{2,n_2} v_{21} \dots v_{2,n_2} \\ & \vdots \\ & \cdot u_{p,1} \dots u_{p,n_p} v_{p,1} \dots v_{p,m_p} \end{aligned}$$

où, $u_{i,j} \in A$ et $v_{i,j} \in B$. On a donc $w \in (A \cup B)^\star$.

(6) On procède par double-inclusion.

"$\subseteq$" Soit $w \in A \cdot (B \cup C)$. On pose $w = u \cdot v$ avec $u \in A$ et $v \in B \cup C$.

— Si $v \in B$, alors $w = u \cdot v \in A \cdot B$ et donc $w \in (A \cdot B) \cup (A \cdot C)$.

— Si $v \in C$, alors $w = u \cdot v \in A \cdot C$ et donc $w \in (A \cdot B) \cup (A \cdot C)$.

On a bien montré $A \cdot (B \cup C) \subseteq (A \cdot B) \cup (A \cdot C)$.

"$\supseteq$" Soit $w \in (A \cdot B) \cup (A \cdot C)$.

— Si $w \in A \cdot B$, on pose alors $w = u \cdot v$ avec $u \in A$ et $v \in B \subseteq B \cup C$. Ainsi, on a bien $w = u \cdot v \in A \cdot (B \cup C)$.

— Si $w \in A \cdot C$, on pose alors $w = u \cdot v$ avec $u \in A$ et $v \in C \subseteq B \cup C$. Ainsi, on a bien $w = u \cdot v \in A \cdot (B \cup C)$.

On a bien montré $A \cdot (B \cup C) \supseteq (A \cdot B) \cup (A \cdot C)$.

3. *(1)* Soit $A = \{a\}$ et $B = \{b\}$ avec $a \neq b$. On sait que $abab \in (A \cdot B)^\star$. Or, $abab \notin A^\star \cdot B^\star$ donc $L_1 \not\subseteq L_2$. De plus, $a \in A^\star \cdot B^\star$ et $a \notin (A \cdot B)^\star$ donc $L_2 \not\subseteq L_1$. Il n'y a aucune relation entre L_1 et L_2.

(2) On sait que $(A \cdot B)^\star \subseteq (A^\star \cdot B^\star)^\star$ (car $A \cdot B \subseteq A^\star \cdot B^\star$ et par croissance de l'étoile). Mais, $(A \cdot B)^\star \not\supseteq (A^\star \cdot B^\star)^\star$. En effet, avec $A = \{a\}$ et $B = \{b\}$ où $a \neq b$, on a $ba \in (A^\star \cdot B^\star)^\star$ (d'après la question précédente) mais $ba \notin (A \cdot B)^\star$. On a donc seulement $L_1 \subseteq L_2$.

(3) On a $L_1 \subseteq L_2$. En effet, $A \cap B \subseteq B$ donc $(A \cap B)^\star \subseteq B^\star$ par croissance l'étoile. De même, $A \cap B \subseteq A$ donc $(A \cap B)^\star \subseteq A^\star$. D'où $(A \cap B)^\star \subseteq A^\star \cap B^\star$. Mais, $L_1 \not\supseteq L_2$. En effet, avec $A = \{a\}$ et $B = \{aa\}$, on a $A \cap B = \varnothing$ et donc $L_1 = (A \cap B)^\star = \{\varepsilon\}$, mais, $L_2 = A^\star \cap B^\star = B^\star$ (car $A^\star \subseteq B^\star$), et donc $L_2 \not\subseteq L_1$.

(4) Comme $A^\star \subseteq (A \cup B^\star)$ et $B^\star \subseteq (A \cup B)^\star$, alors $A^\star \cup B^\star \subseteq (A \cup B)^\star$. Mais, $A^\star \cup B^\star \not\supseteq (A \cup B)^\star$. En effet, si $A = \{a\}$ et $B = \{b\}$ où $a \neq b$, alors on a $ba \in (A \cup B)^\star$ mais $ba \notin A^\star \cup B^\star$. On a donc seulement $L_1 \subseteq L_2$.

(5) On a $L_1 \subseteq L_2$. En effet, soit $w \in A \cdot (B \cap C)$. On pose $w = u \cdot v$ avec $u \in A$ et $v \in B \cap C$. Comme $v \in B$, alors $w = u \cdot v \in A \cdot B$. De même, comme $v \in C$, alors $w = u \cdot v \in A \cdot C$. On a donc bien $w \in (A \cdot B) \cap (A \cdot C)$. D'où $L_1 \subseteq L_2$. Mais, $L_1 \not\supseteq L_2$. En effet, avec $A = \{a, aa\}$, $B = \{b\}$ et $C = \{ab\}$ où $a \neq b$, on a $aab \notin B \cap C = \varnothing$ mais, $aab \in A \cdot B$ et $aab \in A \cdot C$, donc $aab \in L_2$. On a donc seulement $L_1 \subseteq L_2$.

(6) On a, d'après la question 2.

$$L_1 = (A^\star \cup B)^\star = \big((A^\star)^\star \cdot B^\star\big)^\star = (A^\star \cdot B^\star)^\star = (A \cup B)^\star = L_2.$$

TD 3.8 Habitants d'expressions régulières

1. *(1)* Les mots de taille 1, 2, 3 et 4 de $\big((ab)^\star \mid a\big)^\star$ sont a, aa, ab, aaa, $aaaa$, $abab$, aba, $abaa$, aab, $aaab$ et $aaba$.

   (2) On sait, tout d'abord, que l'expression régulière $\big(a \cdot \big((b \cdot b)^\star \mid (a \cdot \varnothing)\big) \cdot b\big) \mid \varepsilon$ est équivalente à $\big(a \cdot (bb)^\star \cdot b\big)$. Les mots de taille 1, 2, 3 et 4 sont donc $abbb$ et ab.

2. *(1)* Les mots de taille 1, 2 et 3 de $(a \mid b)^\star \cdot (a \mid c)^\star$ sont a, b, c, aa, bb, cc, ab, ac, bc, ba, ca, aaa, bbb, ccc, aac, aab, bbc, bba, cca, aba, abc, baa, aca, acc, abb, bca, bcc, cac, bab et caa.

   (2) Les mots de taille 1, 2 et 3 de $(a \cdot b)^\star \mid (a \cdot c)^\star$ sont ab et ac.

TD 3.9 Regexp Crossword

`https://regexcrossword.com/`

TD 3.10 Description d'automates au moyen d'expression régulières

1. $(a \mid b)^\star \cdot a \cdot b \cdot b \cdot a \cdot (a \mid b)^\star$;
2. $a^\star \cdot a \cdot b^\star$;
3. $\big(a \cdot (ab)^\star\big) \mid \big(a \cdot a \cdot (ba)^\star\big)$;
4. $(aa) \cdot (aa)^\star$.

TD 3.11 Vocabulaire des automates

On représente, ci-dessous, l'automate $\mathcal{A}$ décrit dans l'énoncé.

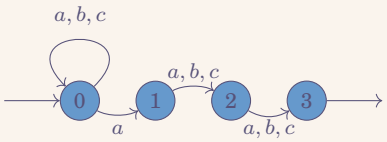


FIGURE TD 3.1 – Automate décrit dans l'énoncé de l'exercice 8

1. Cet automate n'est pas complet : à l'état 0, la lecture d'un a peut conduire à l'état 0 ou bien à l'état 1.
2. Le mot $baba$ est reconnu par $\mathcal{A}$ mais pas le mot $cabcb$.
3. L'automate reconnaît les mots dont la $3^{\underline{\text{ème}}}$ lettre du mot, en partant de la fin, est un a.

TD 3.12 Complétion d'automate

1. Non, cet automate n'est pas complet. Par exemple, la lecture d'un b à l'état 1 est impossible.
2. Cet automate reconnaît le langage $L = \mathcal{L}\big(a \cdot b \cdot (a \mid b)^\star\big)$.
3.

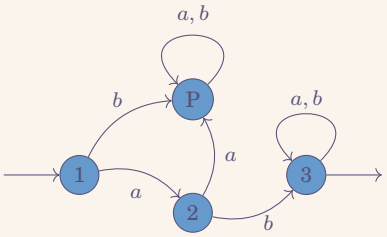


FIGURE TD 3.2 – Automate complet équivalent à $\mathcal{A}$

TD 3.13 Exercice supplémentaire 1

1. *Montrer que l'ensemble des langages reconnaissables est stable par complémentaire.*
2. *Montrer que l'ensemble des langages reconnaissables est stable par intersection.*

1. Soient $\mathcal{A} = (\Sigma, \mathbb{Q}, I, F, \delta)$ et $\mathcal{A}' = (\Sigma, \mathbb{Q}', I', F', \delta')$ deux automates déterministes complets, tels que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. Alors
$$\mathcal{L}\big(\Sigma, \mathbb{Q}', I', \mathbb{Q}' \setminus F', \delta'\big) = \Sigma^\star \setminus \mathcal{L}(\mathcal{A}).$$
2. On utilise les lois de DE MORGAN en passant au complémentaire les deux automates, puis l'union (que l'on a vu en cours), et on repasse au complémentaire.

TRAVAUX DIRIGÉS

4

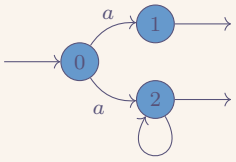
LANGAGES ET EXPRESSIONS RÉGULIÈRES (2)

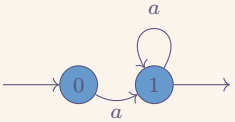
Sommaire

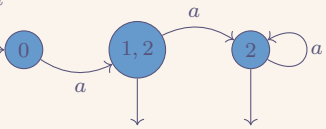
TD 4.1 Déterminisation de taille exponentielle **225**
TD 4.2 Suppression des ε-transitions **226**
TD 4.3 Déterminisation d'automates avec ε-transitions **226**
TD 4.4 Automates pour le calcul de modulo **227**
TD 4.5 Automates pour le calcul de l'addition en binaire **227**
TD 4.5.1 Nombres de même tailles 227

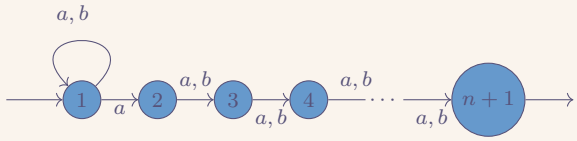
TD 4.1 Déterminisation de taille exponentielle

1. En notant n le nombre d'états de $\mathcal{A}$, alors le nombre d'états de $\det(\mathcal{A})$ est, au plus, 2^n. En effet, les états sont des éléments de $\wp(Q)$ et $|\wp(Q)| = 2^n$.

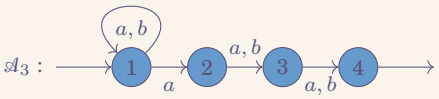
2. $\mathcal{A}$: 

$\mathcal{B}$: 

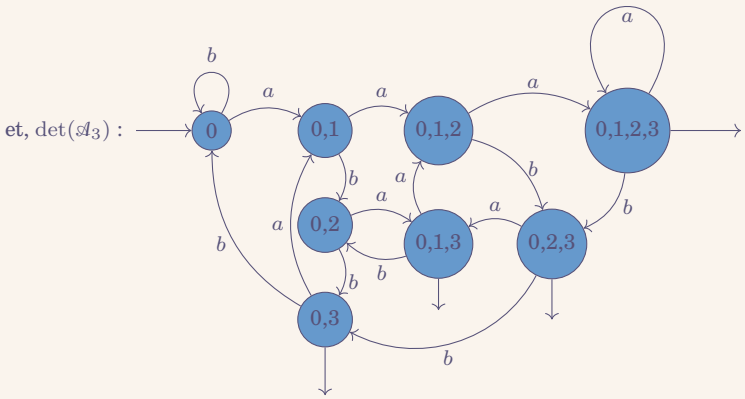
mais $\det(\mathcal{A})$: 

3. $\mathcal{A}_n$: 

4. $\mathcal{A}_3$:



et, $\det(\mathcal{A}_3)$:

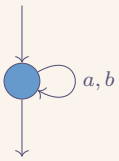


5. Soit $i_0 = \max\{k \in [\![1, n]\!] \mid u_k \neq v_k\}$. Soit $m \in \Sigma^{i_0}$ tel que $u \cdot m \in L_n$ mais $v \cdot m \notin L_n$. Or, $\delta^\star(i, u \cdot m) = \delta^\star(\delta^\star(i, u), m)$ et $\delta^\star(i, v \cdot m) = \delta^\star(\delta^\star(i, v), m)$. D'où $\delta^\star(i, u \cdot m) \in F$ et $\delta^\star(i, v \cdot m) \notin F$. Ce qui est absurde.
6. Ainsi, l'application

$$\begin{aligned} f : \Sigma^\star &\longrightarrow Q \\ u &\longmapsto \delta^\star(i, u) \end{aligned}$$

est injective. D'où, $\mathcal{D}_n = |Q| \geqslant |\Sigma^\star| = 2^n$.
7. D'où, d'après les questions 1 et 6, on en déduit que le nombre d'états utilisés pour la déterminisation de $\mathcal{A}_n$ est de $\mathcal{D}_n \geqslant 2^n$.

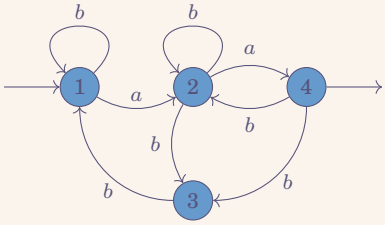
TD 4.2 Suppression des ε-transitions



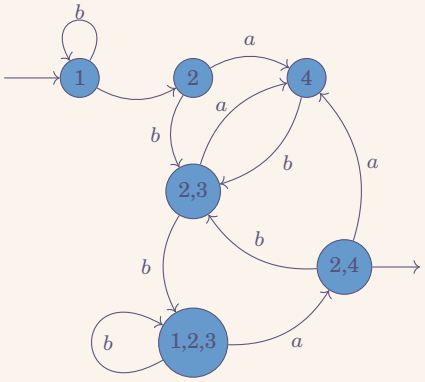
TD 4.3 Déterminisation d'automates avec ε-transitions

Pour les deux automates, on commence par supprimer les ε-transitions, puis on le détermine.

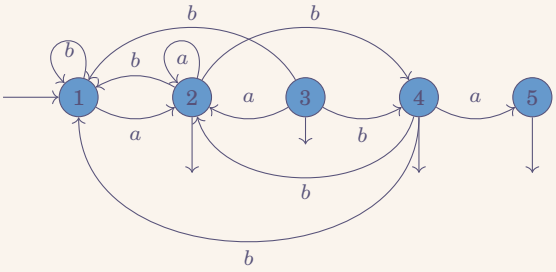
1. L'automate équivalent sans ε-transitions est le suivant.



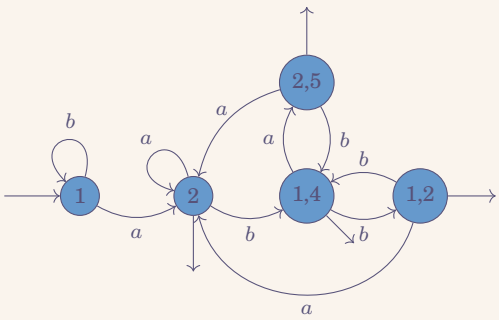
Une fois déterminisé, on obtient l'automate ci-dessous.



2. L'automate équivalent, sans ε-transitions, est le suivant.



Une fois déterminisé, on obtient l'automate ci-dessous.

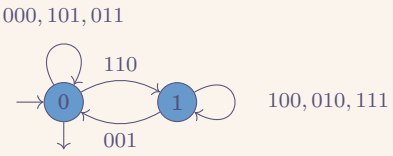


TD 4.4 Automates pour le calcul de modulo

TD 4.5 Automates pour le calcul de l'addition en binaire

TD 4.5.1 Nombres de même tailles

Q. 1



Q. 2 Pour $r \in \{0,1\}$, il existe une exécution dans $\mathcal{A}$ étiquetée par

$$(u_0,v_0,w_0)(u_1,v_1,w_1)\dots(u_{n-1},v_{n-1},w_{n-1})$$

menant à r si et seulement si

$$\overline{u_0\dots u_{n-1}}^2+\overline{v_0\dots v_{n-1}}^2=\overline{w_0\dots w_{n-1}}^2+r\,2^n,$$

ce qui est équivalent à si et seulement si

$$\overline{u_0\dots u_{n-1}0}^2+\overline{v_0\dots v_{n-1}0}^2=\overline{w_0\dots w_{n-1}r}^2.$$

Q. 3 Prouvons-le par récurrence.

— Pour $n=0$, il existe une exécution dans $\mathcal{A}$ étiquetée par ε menant à $r=0$ si et seulement si $\overline{\varepsilon}^2+\overline{\varepsilon}^2=0=\overline{\varepsilon}^2+0\times 2^0$. De même, il existe une exécution dans $\mathcal{A}$ étiquetée par ε menant à $r=1$ si et seulement si $\overline{\varepsilon}^2+\overline{\varepsilon}^2=0=1=\overline{\varepsilon}^2+1\times 2^0$.

TRAVAUX DIRIGÉS

5

LANGAGES ET EXPRESSIONS RÉGULIÈRES (3)

Sommaire

TD 5.1 Exercice 4 229
TD 5.2 Exercice 5 230
TD 5.3 Exercice 6 : Langages reconnaissables ou non 230

TD 5.1 Exercice 4

Q. 1

Algorithme : *Entrée* : Un automate $\mathcal{A}$;
Sortie : $\mathcal{L}(\mathcal{A}) = \varnothing$;
On fait un parcours en largeur depuis les états initiaux et on regarde si on atteint un état final.

Algorithme (Nathan F.) **:** *Entrée* : Deux automates $\mathcal{A}$ et $\mathcal{B}$
Sortie : $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$; Soit $\mathcal{C}$ l'automate reconnaissant $\mathcal{L}(\mathcal{A}) \triangle \mathcal{L}(\mathcal{B})$. On retourne $\mathcal{L}(\mathcal{C}) \stackrel{?}{=} \varnothing$ à l'aide de l'algorithme précédent.

Autre possibilité, on procède par double inclusion :

Algorithme ($\subseteq$) **:** *Entrée* : Deux automates $\mathcal{A}$ et $\mathcal{B}$
Sortie : $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$; On retourne $\mathcal{A} \setminus \mathcal{B} \stackrel{?}{=} \varnothing$.

Q. 2 L'algorithme reconnaissant $\mathscr{L}(\mathscr{A}) \triangle \mathscr{L}(\mathscr{B})$ doit être déterminisé, sa complexité est donc au moins de 2^n.

TD 5.2 Exercice 5

1. On a $e = a(ab \mid b^\star) \mid a$, $f = a_1(a_2b_1 \mid b_2^\star) \mid a_3$et $f_\varphi = e$ où

$$\varphi : \begin{pmatrix} \forall i,\ a_i & \longmapsto & a \\ \forall i,\ b_i & \longmapsto & b \end{pmatrix}.$$

D'où

	Λ	P	S	F
a_1	$\varnothing$	a_1	a_1	$\varnothing$
a_2	$\varnothing$	a_2	a_2	$\varnothing$
$b_2^\star$	ε	b_2	b_2	b_1b_2
$a_2b_1 \mid b_2^\star$	ε	a_2, b_2	b_1, b_2	a_2b_1, b_2b_2
a_3	$\varnothing$	a_3	a_3	$\varnothing$
$a_1(a_2b_1 \mid b_2^\star)$	$\varnothing$	a_1	b_1, b_2, a_1	$a_1a_2, a_1b_2, a_2b_1, b_2b_2$
f	$\varnothing$	a_1, a_3	b_1, b_2, a_3, a_1	$a_1a_2, a_1b_2, a_3b_1, b_2b_2$

Automate à faire…

2. On pose $e = (\varepsilon \mid a)^\star \cdot ab \cdot (a \mid b)^\star$ et $f = (\varepsilon \mid a_1)^\star \cdot a_2b_1 \cdot (a_3 \mid b_2)^\star$ et

$$\varphi : \begin{pmatrix} \forall i,\ a_i & \longmapsto & a \\ \forall i,\ b_i & \longmapsto & b \end{pmatrix}$$

d'où $f_\varphi = e$.

TD 5.3 Exercice 6 : Langages reconnaissables ou non

Q. 7 *Le carré d'un langage est le langage* $L_2 = \{u \cdot u \mid u \in L\}$. *Si* L *est reconnaissable,* L_2 *est-il nécessairement reconnaissable ?*

Avec $\Sigma = \{a, b\}$, soit $L = \mathscr{L}(a^\star \cdot b^\star)$. On a donc $L_2 = \{a^n \cdot b^m \cdot a^n \cdot b^m \mid (n, m) \in \mathbb{N}^2\}$. Supposons L_2 reconnaissable. Soit $\mathscr{A}$ un automate à n états reconnaissant L_2. On pose $u = a^{2n} \cdot b^n \cdot a^{2n} \cdot b^n \in L_2$. D'après le lemme de l'étoile, il existe $(x, y, z) \in (\Sigma^\star)^3$ tel que $u = x \cdot y \cdot z$, $|xy| \leqslant n$, $\mathscr{L}(x \cdot y^\star \cdot z) \subseteq L_2$, et $y \neq \varepsilon$. Ainsi, il existe $m \in [\![1, n]\!]$ et $p \in [\![1, n]\!]$ tels que $y = a^m$, $x = a^p$ et $z = a^{2n-m-p} \cdot b^n \cdot a^{2n} \cdot b^n$. Et alors, $x \cdot y^2 \cdot z = a^p \cdot a^{2m} \cdot a^{n-m-p} \cdot b^n \cdot a^{2n} \cdot b^n = a^{2n+m} \cdot b^n \cdot a^{2n} \cdot b^n \notin L_2$.

Q. 5 *Le langage* $L_5 = \{a^{n^3} \mid n \in \mathbb{N}\}$ *est-il reconnaissable ?* Soit $\mathscr{A}$ un automate à N états, et soit $u = a^{N^3}$. D'après le lemme de l'étoile, il existe $(x, y, z) \in (\Sigma^\star)^3$ tel que $u = x \cdot y \cdot z$, $|xy| \leqslant N$, $\mathscr{L}(x \cdot y^\star \cdot z) \subseteq L_5$ et $y \neq \varepsilon$. D'où $x \cdot y^0 \cdot z \in L$, et donc $a^{N^3-i} \in L$, avec $i \leqslant N$. Or, $\forall k \in \mathbb{N},\ N^3 - i \neq k^3$, ce qui est absurde.

TRAVAUX DIRIGÉS

6 ALGORITHMES PROBABILISTES

Sommaire

TD 6.1 Exercice 1 : Vérification d'égalité polynomiale **231**
TD 6.2 Test de primalité probabiliste **232**
TD 6.2.1 Résultats mathématiques 232
TD 6.2.2 Algorithme 232
TD 6.2.3 Implémentation 232
TD 6.3 Exercice 3 : Échantillonnage **233**

TD 6.1 Exercice 1 : Vérification d'égalité polynomiale

1. Étant donnés deux tableaux représentant deux polynômes, on peut calculer leurs produit en concaténant ce tableau. La complexité du produit de polynômes avec cet algorithme est en $\mathcal{O}(nm)$ où n est le degré du 1er polynôme, et m est le degré du second. En effet, *dans le pire des cas*, tous les polynômes représentant les deux polynômes sont des monômes, or, la concaténation étant en $\mathcal{O}(nm)$ (pour un tableau de taille n et un de taille m). D'où la complexité en $\mathcal{O}(nm)$.
2. Afin d'évaluer ces polynômes, on utilise l'algorithme de Horner, qui est en $\mathcal{O}(n)$, donc en temps linéaire.
3. En développant ces polynômes, la complexité serait en $\mathcal{O}(n^3)$. En effet, la multiplication de deux polynômes de degrés n a une complexité en $\mathcal{O}(n^2)$. D'où la complexité en $\mathcal{O}(n^3)$ pour la multiplication de deux polynômes ayant chacun un degré n.
4. Un polynôme de degré n a, au plus, n racines. D'où, le polynôme $P - Q$, a au plus n racines (où $n = \max(\deg P, \deg Q)$). Ainsi, s'il a $n + 1$ racines, c'est alors le polynôme nul, et donc $P = Q$.

Algorithme TD 6.1 Algorithme déterministe pout tester l'égalité polynomiale en $\mathcal{O}(n^2)$

Entrée : $P = (P_i)_{i \in [\![1,m]\!]}$ et $Q = (Q_j)_{j \in [\![1,p]\!]}$ deux polynômes
$n \leftarrow \deg P$
pour $i \in [\![0, n]\!]$ **faire**
  si $P(i) \neq Q(i)$ **alors** ▷ *Avec l'algorithme de Horner, évaluation en* $\mathcal{O}(n)$
    retourner Non
retourner Oui

5.

Algorithme TD 6.2 Algorithme probabiliste pout tester l'égalité polynomiale en $\mathcal{O}(n)$

Entrée $P = (P_i)_{i\in[\![1,n]\!]}$ et $Q = (Q_j)_{j\in[\![1,n]\!]}$ deux polynômes, et $k \in \mathbb{N}$ un entier

```
x ← 𝒰([[1, k × n]])
si P(x) ≠ Q(x) alors
    retourner Non
retourner Oui
```

Soit X la variable aléatoire de $\mathcal{U}([\![1, k \times n]\!])$. L'événement "$P \neq Q$ mais l'algorithme retourne Oui" arrive si $X \in \{j \in [\![1, kn]\!] \mid P(j) = Q(j)\} = A$. Or $|A| \leqslant n$, et $A \subseteq [\![1, kn]\!]$. Ainsi, l'événement a une probabilité de $\frac{1}{k}$.

TD 6.2 Test de primalité probabiliste

TD 6.2.1 Résultats mathématiques

1. — Élément neutre : soit $x \in G_n$, d'où $x \cdot 1 = 1 \times x \bmod n = x \bmod n$, et donc $1 \in G_n$ est l'élément neutre de G_n.
   — Associativité : par associativité de $\times$, et par le fait que "mod" soit une congruence, on en conclut que $\cdot$ est associative.
   — Soient $x, y \in G_n$. Ainsi, $x \cdot y = x \times y \bmod n$. Or, $x \times y \wedge n = 1$, et donc $x \cdot y \wedge n = 1$.
   — Soit $x \in G_n$, donc $x \wedge n = 1$. D'où, d'après le théorème de Bézout, il existe u et $v \in \mathbb{Z}$ deux entiers tels que $u \times x + v \times n = 1$. D'où $1 \bmod n = u \times x + v \times n \bmod n$ et donc $1 = u \times x \bmod n$. Ainsi $x^{-1} = u \in G_n$, car $u \neq 0$.
2. On sait que $1 \in E_n$. Soit $y \in E_n$, d'où $y^{n-1} \equiv 1\ [n]$, i.e. $y \times (y^{n-2}) \equiv 1\ [n]$, donc $y^{n-2} \in E_n$ est l'inverse de y. Soient x et $y \in E_n$. On a $(x \cdot y^{-1})^{n-1} \equiv x^{n-1} \cdot y^{n-1}\ [n] \equiv 1\ [n]$. D'où $x \cdot y^{-1} \in E_n$. Ainsi, E_n est un sous-groupe de $(G_n, \cdot)$.
3. Soit n composé. Il existe $a \in [\![1, n-1]\!]$ tel que $a^{n-1} \not\equiv 1\ [n]$, et donc $E_n \subsetneq G_n$. Or, *le cardinal d'un sous-groupe divise le cardinal du groupe*, et donc $|E_n| \mid |G_n| \leqslant n-1$, donc $|E_n| \leqslant \frac{n-1}{2}$.

TD 6.2.2 Algorithme

4.

Algorithme TD 6.3 Algorithme Monte-Carlo testant la primalité d'un nombre en $\mathcal{O}(k\,(\ln k)^3$

Entrée $n \in \mathbb{N}$ et $k \in \mathbb{N}$ deux entiers.

```
pour j ∈ [[1, k]] faire
    a ← 𝒰([[1, n − 1]])
    si a^{n−1} mod n ≠ 1 alors
        retourner Non
retourner Oui
```

En effet, si $|E_n| \leqslant \frac{n-1}{2}$, donc si $a \sim \mathcal{U}([\![1, n-1]\!])$, d'où $P(a \in E_n) \leqslant \frac{1}{2}$. La probabilité que l'algorithme échoue est inférieure à $\frac{1}{2^k}$.

TD 6.2.3 Implémentation

Indications Pour calculer $a^b \bmod c$, on décompose b en base 2 : $b = \sum_{i=1}^{p} b_i 2^i$, et donc

$$a^b \bmod c = \left(\prod_{i=1}^{p} a^{b_i 2^i}\right) \bmod c = \prod_{i=0}^{p} \left(a^{b_i 2^i} \bmod c\right).$$

Et, $p \sim \log_2(n)$.

TD 6.3 Exercice 3 : Échantillonnage

Q. 1

Algorithme TD 6.4 Échantillonnage naïf

Entrée T un tableau à n éléments, et $k \in \mathbb{N}$ avec $k \leqslant n$
1 : $T \leftarrow$ Mélanger(T)
2 : $R \leftarrow T[0..k]$
3 : **retourner** R

Q. 2 Un invariant de boucle est « $\forall p \in [\![0, I-1]\!]\,,\ P(T[p] \in \mathsf{Res}) = \frac{k}{I}$ et $\forall p \in [\![I, n]\!]\,,\ T[p] \notin \mathsf{Res}$ »

Q. 3 Notons $\underline{I}$ et $\underline{\mathsf{Res}}$ l'état des variables avant un tour de boucle ; et, $\overline{I}$ et $\overline{\mathsf{Res}}$ l'état des variables après un tour de boucle.

— Pour $k = I$, on a
  1. $\forall p \in [\![0, k-1]\!]$, $P(T[p] \in \mathsf{Res}) = 1$,
  2. $\forall p \in [\![k, n-1]\!]$, $T[p] \notin \mathsf{Res}$,
  3. $I \leqslant n$.

— Supposons $\underline{I}$, et $\underline{\mathsf{Res}}$ vérifiant l'invariant et la condition de boucle. Alors, on a
  1. $\forall p \in [\![0, \underline{I}-1]\!]$, $P(T[p] \in \underline{\mathsf{Res}}) = \frac{k}{\underline{I}}$,
  2. $\forall p \in [\![\underline{I}, n-1]\!]$, $T[p] \notin \underline{\mathsf{Res}}$,
  3. $\underline{I} < n$, la condition de boucle.

  Soit $j \in [\![0, \underline{I}]\!]$. On a $\overline{I} = \underline{I} + 1$.

  Cas 1 $j < k$, et donc $\overline{\mathsf{Res}}(j) = T[\underline{I}]$, et $\forall \ell \neq j$, $\overline{\mathsf{Res}}[\ell] = \underline{\mathsf{Res}}[\ell]$.

  Cas 2 $j \geqslant k$, et donc $\forall \ell$, $\overline{\mathsf{Res}}[\ell] = \underline{\mathsf{Res}}[\ell]$.

  1. Soit $p \in [\![0, \underline{I}]\!]$. Montrons $P(T[p] \notin \overline{\mathsf{Res}}) = \frac{k}{\overline{I}}$. Si $p < \underline{I}$, alors

$$\begin{aligned} P(T[p] \in \overline{\mathsf{Res}}) &= P\big(T[p] \in \underline{\mathsf{Res}} \cap j \neq p\big) \\ &= \frac{k}{\underline{I}} \times \frac{\underline{I}}{\underline{I}+1} \\ &= \frac{k}{\overline{I}}. \end{aligned}$$

  Si $P = \underline{I}$, alors d'après 2. $T[p] \notin \underline{\mathsf{Res}}$, donc $P(T[p] \in \overline{\mathsf{Res}}) = P(j < k) = \frac{k}{\overline{I}+1}$.

TRAVAUX DIRIGÉS

7

DÉCIDABILITÉ, CALCULABILITÉ

Sommaire

TD 7.1 Quelques problèmes décidables 235
TD 7.2 Sérialisation de types énumérés 236
TD 7.3 Stabilité de la classe des langages décidables 236
TD 7.4 Non monotonie du caractère décidable des langages 237
TD 7.5 Réduction 237
TD 7.6 Amélioration de code 239
TD 7.7 Théorème de Rice 240
TD 7.8 Un changement de modèle de calcul 240

TD 7.1 Quelques problèmes décidables

1. Soit $f : \mathbb{R} \to \mathbb{R}$.
   - Si f admet un zéro, on pose $\mathcal{M} =$ `fun s` $\to$ `true`.
   - Si f n'admet pas un zéro, on pose $\mathcal{M} =$ `fun s` $\to$ `false`.

   Alors, $\mathcal{M}$ décide $\textsc{Zero}_f$.
2. Soit $\mathcal{M}$ une machine, et soit $w \in \Sigma^\star$.
   - Si $\mathcal{M}$ se termine sur l'entrée w, alors on pose $\mathcal{M}' =$ `fun s` $\to$ `true`.
   - Si $\mathcal{M}$ ne se termine pas sur l'entrée w, alors on pose $\mathcal{M}' =$ `fun s` $\to$ `false`.

   Alors, $\mathcal{M}'$ décide $\textsc{Arrêt}_{\mathcal{M},w}$.
3. Le problème est trivialement vrai. En effet, soit $M \in \mathbb{O}$, de la forme

```
let m (s: string): string =
  ⟨code⟩
```

   On crée la machine $\mathcal{N}$ ci-dessous.

```
let n (s: string): string =
  if true then
    ⟨code⟩
  else
    ⟨code⟩
```

   On a `m` $\neq$ `n`, mais $\mathscr{L}$(`m`) $= \mathscr{L}$(`n`), donc le problème est vrai sur toute entrée et la fonction `fun s` $\to$ `true` répond au problème.

TD 7.2 Sérialisation de types énumérés

1. Pour sérialiser une liste, on utilise la fonction `serialise_couple` définie dans le cours.

```
let rec serialise_liste (t: 'a list) (serialise: 'a ->
    ↪ string): string =
  match s with
  | [] -> ""
  | x :: q -> serialise_couple (serialise x) (
      ↪ serialise_liste t serialise)
```

Code TD 7.1 – Sérialisation de listes

2. Soit $(\varphi_{i,j})_{\substack{i\in[\![1,m]\!]\\ j\in[\![1,n_i]\!]}}$ sérialisant le type $\mathtt{t}_{i,j}$. Soit

$$\varphi_i : (x_1, \ldots, x_{n_i}) \longmapsto \text{“(”} \hat{} \, \varphi_{i,1}(x_1) \hat{} \text{“)”}, \text{(”} \hat{} \ldots \hat{} \, \varphi_{i,n_i}(x_{n_i}) \hat{} \text{“)”}.$$

```
let rec serialise_enum (e : enumeration) : string =
  let rec aux (i: int) (j: int)
  match e with
  | C_1(e_{1,1},...,e_{1,n_1}) ->
    serialise_couple (1, φ_1(e_1,...,e_{n_1}))
  | ...
  | C_m(e_{m,1},...,e_{m,n_m}) ->
    serialise_couple (m, φ_m(e_1,...,e_{n_m}))
```

Code TD 7.2 – Serialisation de types énumérés

TD 7.3 Stabilité de la classe des langages décidables

1. Soit L un langage fini. On pose donc $L = \{w_1, \ldots, w_n\}$. On code donc la fonction ci-dessous.

```
let decide_L (w: string): bool =
  match w with
  | w_1 -> true
  | ...
  | w_n -> true
  | _ -> false
```

Code TD 7.3 – Fonction décidant d'un langage fini

Autre preuve : tout langage fini est régulier, et tout langage régulier est reconnaissable, et est donc décidable.

2. Soient L_1 et L_2 deux langages décidables. Soit $\mathtt{decide}_{L_1}$ et $\mathtt{decide}_{L_2}$ décidant respectivement L_1 et L_2. On code la fonction $\mathtt{decide}_{L_1 \cdot L_2}$:

```
let decide_{L_1·L_2} (w: string) : bool =
  let m = String.length w in
  let rep = ref false in
  for i = 0 to n - 1 do
    if decide_{L_1} w|_{[[0,i]]} && decide_{L_2} w|_{[[i+1,n-1]]} then
      rep := true
  done;
  !rep
```

Code TD 7.4 – Fonction décidant d'une concaténation de langages décidables

3.

4. Tout langage singleton est décidable. Soit $L_{\text{Arrêt}}$ l'ensemble

$$L_{\text{Arrêt}} = \{\mathtt{serialise\_couple}\ M\ w \mid M \text{ s'arrête sur } w\}.$$

Le langage $L_{\text{Arrêt}}$ est indécidable. Mais, $L_{\text{Arrêt}} = \bigcup_{x \in L_{\text{Arrêt}}} \{x\}$, est une union dénombrable.

TD 7.4 Non monotonie du caractère décidable des langages

1. *Montrer qu'il existe trois langages A, B et C tels que $A \subseteq B \subseteq C$, que A soit décidable, B indécidable et C décidable.*
2. *Montrer qu'il existe A, B et C trois langages tels que $A \subseteq B \subseteq C$, que A soit indécidable, B décidable, et C indécidable.*

1. On pose $A = \varnothing$, et $C = \Sigma^\star$, et $B = \{(M, w) \mid M \text{ s'arrête sur } w\}$.
2. Soit $w \in \Sigma^\star$. On pose

$$L_0 = \{\texttt{serialise\_couple}(w, M) \mid M \in L_{\textsc{ArrêtUniv}}{}^1\}.$$

Montrons que le langage L_0 est indécidable. Soit $M \in \mathbb{O}$ une entrée du problème ArrêtUniv.

$$\begin{aligned}\texttt{serialise\_couple}(w, M) \in L_0 &\iff M \text{ s'arrête sur toutes ses entrées}\\ &\iff M \in \textsc{ArrêtUniv}^+.\end{aligned}$$

En effet, la fonction f de cette réduction est définie comme ci-dessous.

```
let f (M : string) : string =
  serialise_couple w M
```

Par réduction de ArrêtUniv à $\textsc{Appartient}_{L_0}$, le problème $\textsc{Appartient}_{L_0}$ est indécidable. On pose ensuite

$$L_1 = \{\texttt{serialise\_couple}(M, w) \mid w \in \Sigma^\star, M \in L_{\textsc{ArrêtUniv}}\}.$$

Soit $M \in \mathbb{O}$ une entrée du problème ArrêtUniv. La fonction g de cette réduction est définie comme ci-dessous.

```
let g (M: string) : string =
  serialise_couple M "a"
```

$$\begin{aligned}g(M) \in L_1 &\iff \begin{cases}\texttt{"a"} \in \Sigma^\star\\ M \in \textsc{ArrêtUniv}^+\end{cases}\\ &\iff M \in \textsc{ArrêtUniv}^+\end{aligned}$$

On pose $w =$ "`fun` $s \to$ `true`." On a $L_0 \subseteq L \subseteq L_2$ où

$$L = \{\texttt{serialise\_couple}(\text{“}\texttt{fun } s \to \texttt{true}\text{”}, w) \mid w \in \Sigma^\star\}.$$

Le langage L est décidable. En effet, le code ci-dessous décide du problème $\textsc{Appartient}_L$.

```
let decide_L (w: string) : bool =
  let m = String.length w in
  w.[n - 1] = ')' && w_|[0,15]| = "(fun␣s␣->␣true)("
```

TD 7.5 Réduction

1. Soit $\texttt{m}, w$ les entrées du problème de l'Arrêt. On fabrique la sérialisation de machine

$$\text{“}\texttt{fun s} \to \texttt{execute } \{\{\texttt{m}\}\}\ \{\{w\}\}\text{.”}$$

Notons $M'_{\texttt{m},w}$ cette machine. On a

$$\begin{aligned}M'_{\texttt{m},w} \in \textsc{ArrêtUniv}^+ &\iff \forall x \in \Sigma^\star, \texttt{ execute m } w \text{ se termine}\\ &\iff \texttt{execute m } w \text{ se termine}\\ &\iff (\texttt{m}, w) \in \textsc{Arrêt}^+\end{aligned}$$

Ainsi, on crée la réduction.

```
let reduction (s: string): string =
  let (m, w) = deserialise_couple s in
  "fun x -> execute " ^ m ^ " " ^ w
```

CODE TD 7.5 – Réduction de ARRÊTUNIV au problème de l'ARRÊT

Ainsi, ARRÊTUNIV est indécidable par réduction.

2. Réduisons le problème ARRÊT à ARRÊTSIMULT. Soit $\mathtt{m}, w$ les données du problème de l'ARRÊT. Posons les machines $M_{\mathtt{m},w} =$ "$\mathtt{fun}\ x \to \mathtt{execute}\ \{\{\mathtt{m}\}\}\ \{\{w\}\}$", et N : "$\mathtt{fun}\ x\ \to 3$." On a

$$\begin{aligned}
&\mathtt{serialise_couple}(M_{\mathtt{m},w}, N) \in \text{ARRÊT}^+ \\
\iff &(\forall x \in \Sigma^\star, \quad \mathtt{execute}\ M_{\mathtt{m},w}\ x\ \text{termine} \iff \mathtt{execute}\ N\ x\ \text{termine}) \\
\iff &(\mathtt{execute}\ \{\{\mathtt{m}\}\}\ \{\{w\}\}\ \text{termine} \iff \boldsymbol{V}) \\
\iff &\mathtt{execute}\ \{\{\mathtt{m}\}\}\ \{\{w\}\}\ \text{termine} \\
\iff &(\mathtt{m}, w) \in \text{ARRÊT}^+.
\end{aligned}$$

Autre possibilité : réduisons ARRÊTUNIV à ARRÊTSIMULT. On pose N : "$\mathtt{fun}\ x \to 2$," et on a

$$\begin{aligned}
&\mathtt{serialise_couple}(M, N) \in \text{ARRÊTSIMULT}^+ \\
\iff &(\forall x \in \Sigma^\star, \quad \mathtt{execute}\ M\ x\ \text{termine} \iff \mathtt{execute}\ N\ x\ \text{termine}) \\
\iff &\forall x \in \Sigma^\star, \ \mathtt{execute}\ M\ x\ \text{termine} \\
\iff &M \in \text{ARRÊTUNIV}^+.
\end{aligned}$$

3. Réduisons ARRÊT à RÉGULIER. Soit $(\mathtt{m}, w)$ une entrée du problème de l'ARRÊT.

```
let reconnait_an_bn (w: string): bool =
  let n = String.length w in
  if n mod 2 = 1 then false
  else begin
      let ok = ref true in
      for i = 1 to n / 2 do
        if w.[i] = 'a' then ok := false
      done;
      for i = (n / 2) + 1 to n do
        if w.[i] = 'b' then ok := false
      done;
      !ok
    end
```

CODE TD 7.6 – Machine reconnaissant le langage $\{a^n \cdot b^n \mid n \in \mathbb{N}\}$

On considère la fonction de réduction ci-dessous.

```
let f (e: string): string =
  let (m, w) = deserialise_couple e in
  "fun s -> execute {{m}} {{w}}; reconnait_an_bn {{w}}"
```

En notant $(\mathtt{m}, \mathtt{w}) = \mathtt{deserialise\_couple(e)}$,

$$(f\ \mathtt{e}) \in \text{RÉGULIER}^+ \iff \text{"}\mathtt{fun}\ s \to \mathtt{execute}\{\{\mathtt{m}\}\}\{\{\mathtt{w}\}\}\text{"}.$$

Or,

- si $w \underset{\mathtt{m}}{\to} \circlearrowleft$, alors $\forall s \in \Sigma^\star$, $s \underset{\mathtt{m}}{\to} \circlearrowleft$, donc $\mathscr{L}(\mathtt{m}) = \varnothing$.
- si $w \underset{\mathtt{m}}{\to} w'$, avec $w' \in \Sigma^\star$, alors
  - si $s \underset{\mathtt{m}}{\to} \mathtt{true} \iff s \in \{a^n \cdot b^n \mid n \in \mathbb{N}\}$,
  - si $s \underset{\mathtt{m}}{\to} \mathtt{false} \iff s \notin \{a^n \cdot b^n \mid n \in \mathbb{N}\}$,

  Et donc, $\mathscr{L}(\mathtt{m}) = \{a^n \cdot b^n \mid n \in \mathbb{N}\}$.

On a donc

$$\begin{aligned}(f\ \mathtt{e}) \in \textsc{Régulier}^+ &\iff w \underset{\mathtt{m}}{\to} \circlearrowleft \\ &\iff (\mathtt{m}, w) \in \textsc{Arrêt}^- \\ &\iff (\mathtt{m}, w) \in \textsc{CoArrêt}^+\end{aligned}$$

où le problème CoArrêt est défini comme

$$\textsc{CoArrêt} : \begin{cases}\textbf{Entrée} & : (\mathtt{m}, w) \\ \textbf{Sortie} & : \text{A-t-on } w \underset{\mathtt{m}}{\to} \circlearrowleft \ ?\end{cases}.$$

On a $\textsc{CoArrêt}^- = \textsc{Arrêt}^+$, et $\textsc{CoArrêt}^+ = \textsc{Arrêt}^-$. Le problème CoArrêt est indécidable par stabilité des problèmes décidables par complémentaire. On conclut par réduction de CoArrêt à Régulier.

4. Réduisons Arrêt à $\textsc{Arrêt}_w$. Soit (w, M) une entrée du problème Arrêt. Fabriquons la machine M_w : "`fun` $s \to$ `execute` $M\ w$." On a

$$\begin{aligned}M_w \in {\textsc{Arrêt}_w}^+ &\iff (\forall x \in \Sigma^\star,\ \texttt{execute}\ M\ w \text{ se termine}) \\ &\iff \texttt{execute}\ M\ w \text{ se termine} \\ &\iff (M, w) \in \textsc{Arrêt}^+\end{aligned}$$

Par réduction, le problème $\textsc{Arrêt}_w$ est indécidable.

5. Réduisons ArrêtUniv à ArrêtExiste. Soit M une entrée du problème ArrêtExiste.

TD 7.6 Amélioration de code

1.

```
let mort (s: string): string =
  let f (a: int) = a in
  "sortie"
```

Code TD 7.7 – Fonction morte

2. On réalise une réduction du problème ArrêtUniv au problème CodeMort. Soit $\mathscr{M}$ une entrée du problème ArrêtUniv. Fabriquons l'entrée M du problème CodeMort, comme montré ci-dessous.

```
let M (s: string): string =
  execute 𝓜 s;
  let f (a: string) = a in
  f x
```

Code TD 7.8 – Réduction de ArrêtUniv à CodeMort

Ainsi, avec cette construction de la machine M, on définition u comme `execute` $\mathscr{M}$ `s`, on définit d comme `let f (a: string) = a in`, et on définit v comme `f x`. Alors,

$$\begin{aligned}M \in \textsc{CodeMort}^+ &\iff \mathscr{L}(u \cdot d \cdot v) = \mathscr{L}(u \cdot v) \\ &\iff \forall \mathtt{s} \in \Sigma^\star,\ \text{la ligne 4 est atteinte} \\ &\iff \forall \mathtt{s} \in \Sigma^\star,\ \texttt{execute}\ \mathscr{M}\ \mathtt{s} \text{ se termine} \\ &\iff \forall \mathtt{s} \in \Sigma^\star,\ \mathscr{M} \text{ se termine sur l'entrée } \mathtt{s} \\ &\iff \mathscr{M} \in \textsc{ArrêtUniv}^+.\end{aligned}$$

Or, comme ArrêtUniv est indécidable, CodeMort aussi.

3.

```
let f (s: string): string =
  let x = ref 3 in
  let y = int_of_string s in
  string_of_int (!x + y)

let f (s: string): string =
  let y = int_of_string s in
  string_of_int (3 + y)
```

CODE TD 7.9 – Variable constante

4. Soit M une entrée du problème CODEMORT. Fabriquons l'entrée $(M', \mathscr{V})$ du problème P de détection de variables constantes. Soit F l'ensemble des "fonctions locales" de M.[2] Au début de la machine M', on définit, pour chaque fonction locale $\texttt{f} \in F$, la variable $\texttt{x}_\texttt{f}$ comme `let` $\texttt{x}_\texttt{f}$ `= ref 0 in`. On pose $\mathscr{V}$ l'ensemble de ces nouvelles variables :

$$\mathscr{V} = \{\texttt{x}_\texttt{f} \mid \texttt{f} \in F\}.$$

Puis, on transforme chaque définition de fonction locale $\texttt{f} \in F$ en

```
let f ⟨arguments de f⟩ =
  incr x_f ;
  ⟨code original de f⟩
```

Ainsi,

$$\begin{aligned}(M', \mathscr{V}) \in P^+ &\iff \exists \texttt{x}_\texttt{f} \in \mathscr{V}, \text{ la variable } \texttt{x}_\texttt{f} \text{ n'est pas modifiée} \\ &\iff \exists \texttt{f} \in F, \text{ la fonction } \texttt{f} \text{ n'est pas appelée} \\ &\iff \exists \texttt{f} \in F, \ \texttt{f} \text{ est une fonction morte} \\ &\iff M \in \text{CODEMORT}^+.\end{aligned}$$

Or, comme le problème CODEMORT est indécidable, le problème de détection de variables constantes l'est aussi.

TD 7.7 Théorème de Rice

TD 7.8 Un changement de modèle de calcul

1. Pour toute machine M, on considère la *super-machine* $\mathscr{M}_M$ suivante :

```
let arret (n_V: int) (x: string): bool =
  let (M,w) = deserialise_couple x in
  decide_ArrêtÉtapes M w n_V
```

CODE TD 7.10 – *Super-machine* résolvant le problème de l'ARRÊT sur une machine classique

où la fonction $\texttt{decide}_{\text{ArrêtÉtapes}}$ décide du problème

$$\text{ArrêtÉtapes} : \begin{cases}\textbf{Entrée} & : M \in \mathbb{O},\ w \in \Sigma^\star,\ n \in \mathbb{N} \\ \textbf{Sortie} & : M \text{ se termine-t-elle sur } w \text{ en moins de } n \text{ étapes élémentaires ?}\end{cases}.$$

D'après le cours, ce problème est décidable. Ainsi, pour tout mot $w \in \Sigma^\star$ et toute machine M,

$$\begin{aligned}w \xrightarrow[\mathscr{M}_M]{} \boldsymbol{V} &\iff \exists n \in \mathbb{N},\ \texttt{arret}\ n\ (\texttt{serialise\_couple}\ M\ w) = \texttt{true} \\ &\iff \exists n \in \mathbb{N},\ M \text{ s'arrête en moins de } n \text{ étapes sur } w \\ &\iff M \text{ s'arrête sur l'entrée } w \\ &\iff (M, w) \in \text{ARRÊT}^+.\end{aligned}$$

2. S'il y a plusieurs fonctions ayant le même nom, on les numérote de telle sorte que leurs noms soient différents.

2. Par l'absurde, supposons le problème de l'arrêt pour les *super-machines* décidable. Soit `arret` une fonction décidant de ce problème. On considère la machine suivante.

```
let paradoxe (n: int) (w: string): bool =
  if arret n (serialise_couple w w) then
    (while true do () done; true)
  else false
```

CODE TD 7.11 – Programme paradoxe proubant que le problème de l'arrêt des *super-machines* est indécidable

Soit $S_{\texttt{paradoxe}}$ la sérialisation de la fonction `paradoxe` ci-dessous. Analysons l'exécution de (`paradoxe` n $S_{\texttt{paradoxe}}$). Soit $c =$ (`serialise_couple` $S_{\texttt{paradoxe}}$ $S_{\texttt{paradoxe}}$).

TRAVAUX DIRIGÉS

8

CLASSE **P**, CLASSE **NP**

Sommaire

TD 8.1 Problèmes de partitions **243**
TD 8.2 Optimisation linéaire en nombres entiers **244**
TD 8.3 Clique, Stable et Couv.Sommets **244**
TD 8.4 . . . . . **245**

TD 8.1 Problèmes de partitions

1. On définit les quatre problèmes comme

   SubsetSum : $\begin{cases}\textbf{Entrée} & : n \in \mathbb{N},\ (w_1,\dots,w_n) \in \mathbb{N}^n \text{ et } W \in \mathbb{N} \\ \textbf{Sortie} & : \text{Existe-t-il une partie } I \in \wp(\llbracket 1,n \rrbracket) \text{ telle que } \sum_{i\in I} w_i = W\,?\end{cases}$

   Partition : $\begin{cases}\textbf{Entrée} & : n \in \mathbb{N} \text{ et } (w_1,\dots,w_n) \in \mathbb{N}^n \\ \textbf{Sortie} & : \text{Existe-t-il } I \in \wp(\llbracket 1,n \rrbracket) \text{ telle que } \sum_{i\in I} w_i = \sum_{i\in \llbracket 1,n\rrbracket\setminus I} w_i\,?\end{cases}$

   Knapsack : $\begin{cases}\textbf{Entrée} & : n \in \mathbb{N},\ (x_1,\dots,x_n) \in \mathbb{N}^n,\ (v_1,v_2,\dots,v_n) \in \mathbb{N}^n,\ P \in \mathbb{N} \text{ et } K \in \mathbb{N} \\ \textbf{Sortie} & : \text{Existe-t-il } I \in \wp(\llbracket 1,n \rrbracket) \text{ telle que } \sum_{i\in I} x_i \leqslant P \text{ et } \sum_{i\in I} v_i \geqslant K\,?\end{cases}$

   BinPacking : $\begin{cases}\textbf{Entrée} & : n \in \mathbb{N},\ (t_1,\dots,t_n) \in \mathbb{N}^n,\ C \in \mathbb{N} \text{ et } K \in \mathbb{N} \\ \textbf{Sortie} & : \text{Existe-t-il } k \in \mathbb{N} \text{ et } (I_1,\dots,I_k) \in \wp(\llbracket 1,n \rrbracket)^k \text{ } k \text{ parties de } \llbracket 1,n \rrbracket \\ & \quad \text{telles que } \bigcup_{i\in\llbracket 1,k\rrbracket} I_i = \llbracket 1,n \rrbracket, \forall i \neq j,\ I_i \cap I_j = \varnothing,\ k \leqslant K \text{ et} \\ & \quad \forall i \in \llbracket 1,k \rrbracket,\ \sum_{j\in I_i} t_j \leqslant C\,?\end{cases}$

2. (a) Soit (Ω, W) une entrée de SubsetSum. On fabrique l'entrée (Ω,Ω,W,W) du problème Knapsack. On pose $\Omega = (w_1,\dots,w_n)$ et on a

$$\begin{aligned}(\Omega,W) \in \text{SubsetSum}^+ &\iff \exists A \in \wp(\Omega),\ \sum_{w\in A} w = W \\ &\iff \exists A \in \wp(\Omega),\ \sum_{w\in A} w \geqslant W \text{ et } \sum_{w\in A} w \leqslant W \\ &\iff (\Omega,\Omega,W,W) \in \text{Knapsack}^+.\end{aligned}$$

   Cette réduction est polynômiale.

(b) Soit Ω une entrée de PARTITION. On pose $\Omega = (w_1, \dots, w_n)$. Fabriquons l'entrée $(\Omega, S/2)$ du problème SUBSETSUM, où $S = \sum_{i=1}^{n} w_i$. On suppose $S \equiv 0\ [2]$. On a

$$\begin{aligned}
\Omega \in \text{PARTITION} &\iff \exists I \subseteq [\![1,n]\!]\,,\ \sum_{i\in I} w_i = \sum_{i\in[\![1,n]\!]\setminus I} w_i \\
&\iff \exists I \subseteq [\![1,n]\!],\ \sum_{i\in I} w_i = \frac{S}{2} \\
&\iff (\Omega, S/2) \in \text{SUBSETSUM}^+
\end{aligned}$$

TD 8.2 Optimisation linéaire en nombres entiers

1. Montrons SYSLIN $\preccurlyeq_{\text{p}}$ SYSLININEG. Soient n, m, A et b les entrées du problème SYSLIN. Fabriquons, en temps polynômial, les entrées n', m', A', b' du problème SYSLININEG : on choisit $n' = 2n$, $m' = m$, $A' = \left(\begin{smallmatrix} A \\ -A \end{smallmatrix}\right)$ et $b' = \left(\begin{smallmatrix} b \\ -b \end{smallmatrix}\right)$. Ainsi,

$$\begin{aligned}
(n', m', A', b') \in \text{SYSLININEG}^+ &\iff \exists X,\ A'X \leqslant b' \\
&\iff \exists X,\ AX \leqslant b \text{ et } -AX \leqslant -b \\
&\iff \exists X,\ AX = b \\
&\iff (n, m, A, b) \in \text{SYSLIN}^+
\end{aligned}$$

TD 8.3 CLIQUE, STABLE et COUV.SOMMETS

1. Soit $G = (S, A)$ un graphe. On pose $G' = (S, A')$ où $A' = \big\{\{x, y\} \in \mathscr{C}_2(S) \big| \{x, y\} \notin A\big\}$.[1]
   Prouvons la réduction de CLIQUE à STABLE. Soit (G, K) une entrée du problème CLIQUE. Fabriquons l'entrée (G', K) de STABLE, comme défini précédemment. Montrons que $(G, K) \in \text{CLIQUE}^+ \iff (G', K) \in \text{STABLE}^+$. On a

$$\begin{aligned}
(G, K) \in \text{CLIQUE}^+ &\iff \exists S_1 \subseteq S_2 \text{ avec } |S_1| \geqslant K,\ \forall x \neq y \in S_1,\ \{x, y\} \in A \\
&\iff \exists S_1 \subseteq S_2 \text{ avec } |S_1| \geqslant K,\ \forall x \neq y,\ \{x, y\} \notin A' \\
&\iff (G', K) \in \text{STABLE}^+.
\end{aligned}$$

   La réduction est calculable en temps polynômiale. On a donc CLIQUE $\preccurlyeq_{\text{p}}$ STABLE.

   La réduction de STABLE à CLIQUE est la même. Elle est également en temps polynômiale.

2. Montrons la réduction de COUVSOMMETS à CLIQUE. Soit (G, K) une entrée du problème COUVSOMMETS. Fabriquons $(G', n - K)$ une entrée du problème STABLE, où $G' = (S, A')$ comme défini à la question précédente, et $n = |S|$. On a

$$\begin{aligned}
&(G, K) \in \text{COUVSOMMETS}^+ \\
\iff &\exists S_1 \subseteq S \text{ avec } |S_1| \leqslant K,\ \forall \{x, y\} \in A,\ x \in S_1 \text{ ou } x \in S_2 \\
\iff &\exists S_1 \subseteq S \text{ avec } |S_1| \leqslant K,\ \forall x \neq y \in S \setminus S_1,\ \{x, y\} \notin A \\
\iff &\exists S_1 \subseteq S \text{ avec } |S \setminus S_1| \leqslant |S| - K,\ \forall x \neq y \in S \setminus S_1,\ \{x, y\} \in A' \\
\iff &\exists S_2 \subseteq S \text{ avec } |S_2| \geqslant |S| - K,\ \forall x \neq y \in S_2,\ \{x, y\} \in A' \\
\iff &(G', |S| - K) \in \text{STABLE}^+.
\end{aligned}$$

3. On représente le graphe G pour l'entrée $\{x \vee x \vee y, \neg x \vee \neg y \vee \neg y, x \vee y \vee y\}$.

1. On note $\mathscr{C}_p(E)$ l'ensemble des parties de E de cardinal p.

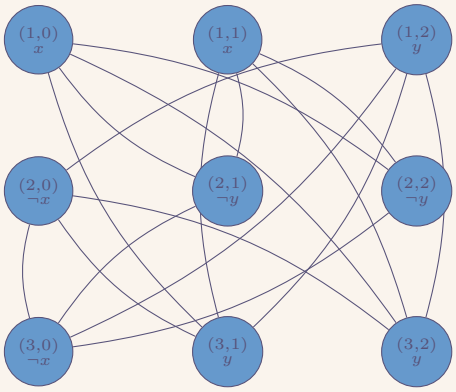


FIGURE TD 8.1 – Représentation du graphe G pour l'entrée $\{x \vee x \vee y, \neg x \vee \neg y \vee \neg y, x \vee y \vee y\}$

4. Montrons la réduction de 3SAT à CLIQUE. Soit $H = \{c_1, \dots, c_m\}$ une instance de 3SAT. Construisons le graphe G comme proposé dans l'énoncé. On construit alors l'entrée de CLIQUE (G, m). Soit $(G, m) \in \text{CLIQUE}^+$. C'est donc qu'il existe une clique de G de taille m ; nommons la C. Deux sommets $(i, \_)$ et $(j, \_)$ ne sont pas reliés dans G si $i = j$. Ainsi,

$$\forall i \in [\![1, m]\!]\,,\ \exists! j \in [\![0, 2]\!]\,,\ (i, j) \in C.$$

Soient p et $\neg p$ deux littéraux. Si $p \in C$, alors $\neg p \notin C$. Si $\neg p \in C$, alors $p \notin C$. On construit alors l'environnement propositionnel

$$\rho : \mathbb{Q} \longrightarrow \mathbb{B}$$
$$p \longmapsto \begin{cases} \boldsymbol{V} & \text{si } p \in C \\ \boldsymbol{F} & \text{sinon.} \end{cases}$$

Pour $i \in [\![1, m]\!]$, soit $j \in [\![0, 2]\!]$, tel que $(i, j) \in C$, on a donc $[\![\ell_{i,j}]\!]^\rho = \boldsymbol{V}$. On en déduit que $\forall i \in [\![1, m]\!]\,,\ [\![c_i]\!]^\rho = \boldsymbol{V}$. On en déduit que $[\![H]\!]^\rho = \boldsymbol{V}$, *i.e.* $H \in \text{3SAT}^+$.
Réciproquement, supposons $H \in \text{3SAT}^+$. Alors, soit $\rho \in \mathbb{B}^{\mathbb{Q}}$ tel que $[\![H]\!]^\rho = \boldsymbol{V}$. On a

$$\forall i \in [\![1, m]\!]\,,\ \exists j \in [\![0, 2]\!]\,,\ [\![\ell_{i,j}]\!]^\rho = \boldsymbol{V}.$$

Notons $\varphi(i)$ un tel j. On fabrique alors l'ensemble de m sommets $C = \{(i, \varphi(i)) \mid i \in [\![1, m]\!]\}$. Soient $(i, \varphi(i))$ et $(j, \varphi(j))$ deux éléments de C. Si $i \neq j$, alors $\{(i, \varphi(i)), (j, \varphi(j))\} \in A$, car $\left[\!\left[\ell_{i,\varphi(i)}\right]\!\right]^\rho = \boldsymbol{V}$ et $\left[\!\left[\ell_{j,\varphi(j)}\right]\!\right]^\rho = \boldsymbol{V}$. On en déduit que C est une clique de taille m. Ainsi, $(G, m) \in \text{CLIQUE}^+$.

5. Comme CLIQUE est **NP**-difficile, alors STABLE et COUVSOMMETS sont **NP**-difficile. Montrons que CLIQUE est un problème **NP**. Le programme CLIQUE est vérifiable en temps polynômiale : ...

TD 8.4

TRAVAUX DIRIGÉS

9 ALGORITHMIQUE DES GRAPHES

Sommaire

TD 9.1 Arbres et forêts **247**
TD 9.2 Tri topologique **247**
TD 9.3 Détection de graphe biparti **248**
TD 9.4 Exploration du graphe $\mathfrak{S}_n$ **248**
TD 9.5 Parcours selon le miroir d'un tri préfixe **248**

TD 9.1 Arbres et forêts

TD 9.2 Tri topologique

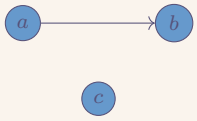


FIGURE TD 9.1 – Exemple de graphe

1. Dans le graphe ci-dessus, $a \to c \to b$ est un tri topologique mais pas un parcours.
2. Dans le même graphe, $b \to a \to c$ est un parcours mais pas un tri topologique.
3. Supposons que L_1 possède un prédécesseur, on le note L_i où $i > 1$. Ainsi, $(L_i, L_1) \in A$ et donc $i < 1$, ce qui est absurde. De même pour le dernier.
4. Il existe un tri topologique si, et seulement si le graphe est acyclique.

" $\Longrightarrow$ " Soit $L_1, \ldots, L_n$ un tri topologique. Montrons que le graphe est acyclique. Par l'absurde, on suppose le graphe non acyclique : il existe $(i, j) \in [\![1, n]\!]^2$ avec $i \neq j$ tels que $T_i \to \cdots \to T_j$ et $T_j \to \cdots \to T_i$ soient deux chemins valides. Ainsi, comme le tri est topologique et par récurrence, $i \leqslant j$ et $j \leqslant i$ et donc $i = j$, ce qui est absurde car i et j sont supposés différents. Le graphe est donc acyclique.

" $\Longleftarrow$ " Soit G un graphe tel que tous les sommets possèdent une arrête entrante. On suppose par l'absurde ce graphe acyclique. Soit x_0 un sommet du graphe. On

construit par récurrence $x_0, x_1, \ldots, x_n, x_{n+1}, \ldots$ les successeurs successifs. Il y a un nombre fini de sommets donc deux sommets sont identiques. Donc, il y a nécessairement un cycle, ce qui est absurde.

Algorithme TD 9.1 Génération d'un tri topologique d'un graphe acyclique

Entrée $G = (S, A)$ un graphe acyclique
Sortie Res un tri topologique.

```
1 : Res ← [ ]
2 : tant que G ≠ ∅ faire
3 :     Soit x un sommet de G sans prédécesseur
4 :     G ← (S \ {x}, A ∩ (S \ {x})²)
5 :     Res ← Res · [x]
6 : retourner Res
```

5.

Algorithme TD 9.2 Génération d'un tri topologique d'un graphe

Entrée $G = (S, A)$ un graphe
Sortie Res un tri topologique, ou un cycle

```
1 : Res ← [ ]
2 : tant que G ≠ ∅ faire
3 :     si il existe x sans prédécesseurs alors
4 :         Soit x un sommet de G sans prédécesseur
5 :         G ← (S \ {x}, A ∩ (S \ {x})²)
6 :         Res ← Res · [x]
7 :     sinon
8 :         Soit x ∈ S
9 :         Soit x ← x_1 ← x_2 ← ··· ← x_i la suite des prédécesseurs
10 :        retourner x_i, x_{i+1}, ..., x_i, un cycle
11 : retourner Res
```

6. On utilise la représentation par liste d'adjacence, et on stocke le nombre de prédécesseurs que l'on décroit à chaque choix de sommet.
7. On essaie de trouver un tri topologique, et on voit si l'on trouve un cycle.

TD 9.3 Détection de graphe biparti

TD 9.4 Exploration du graphe $\mathfrak{S}_n$

TD 9.5 Parcours selon le miroir d'un tri préfixe

TRAVAUX DIRIGÉS

10

PREUVES EN LOGIQUE PROPOSITIONNELLE

Sommaire

TD 10.1 Premiers arbres de preuves **250**
TD 10.2 Divers arbres de preuves **250**
TD 10.3 Lois de DE MORGAN **251**
TD 10.4 Distributivités entre $\wedge$ et $\vee$ **251**
TD 10.5 Implications **252**
TD 10.6 Implications (partie 1 : simplifications) **253**
TD 10.7 Implications (partie 2 : transformations) **254**

TD 10.1 Premiers arbres de preuves

1.
$$\dfrac{\dfrac{}{p \vdash p}\,\text{Ax}}{\varnothing \vdash p \to p}\,{\to}\text{i}.$$

2.
$$\dfrac{\dfrac{}{p, \neg p \vdash p}\,\text{Ax} \qquad \dfrac{}{p, \neg p \vdash \neg p}\,\text{Ax}}{p, \neg p \vdash \bot}\,\neg\text{e}.$$

3.
$$\dfrac{\dfrac{}{p, q \vdash p}\,\text{Ax} \qquad \dfrac{}{p, q \vdash q}\,\text{Ax}}{p, q \vdash p \wedge q}\,\wedge\text{i}.$$

4.
$$\dfrac{\dfrac{\dfrac{}{p \wedge q \vdash p \wedge q}\,\text{Ax}}{p \wedge q \vdash q}\,\wedge\text{e,g} \qquad \dfrac{\dfrac{}{p \wedge q \vdash p \wedge q}\,\text{Ax}}{p \wedge q \vdash p}\,\wedge\text{e,d}}{p \wedge q \vdash q \wedge p}\,\wedge\text{i}.$$

5.
$$\dfrac{\dfrac{}{p \vee q \vdash p \vee q}\,\text{Ax} \qquad \dfrac{\dfrac{}{p \vee q, p \vdash p}\,\text{Ax}}{p \vee q, p \vdash q \vee p}\,\vee\text{i,d} \qquad \dfrac{\dfrac{}{p \vee q, q \vdash q}\,\text{Ax}}{p \vee q, q \vdash q \vee p}\,\vee\text{i,g}}{p \vee q \vdash q \vee p}\,\vee\text{e}.$$

6.
$$\dfrac{\dfrac{\dfrac{\dfrac{}{p \wedge \neg p \vdash p \wedge \neg p}\,\text{Ax}}{p \wedge \neg p \vdash p}\,\wedge\text{e,g} \qquad \dfrac{\dfrac{}{p \wedge \neg p \vdash p \wedge \neg p}\,\text{Ax}}{p \wedge \neg p \vdash \neg p}\,\wedge\text{e,d}}{p \wedge q \vdash \bot}\,\neg\text{e}}{\vdash \neg(p \wedge \neg p)}\,\neg\text{i}.$$

TD 10.2 Divers arbres de preuves

1.
$$\dfrac{\dfrac{}{p \vee (p \wedge q) \vdash p \vee (p \wedge q)}\,\text{Ax} \qquad \dfrac{}{p \vee (p \wedge q), p \vdash p}\,\text{Ax} \qquad \dfrac{\dfrac{}{p \vee (p \wedge q), p \vee q \vdash p \wedge q}\,\text{Ax}}{p \vee (p \wedge q), p \vee q \vdash p}\,\wedge\text{e,g}}{p \vee (p \wedge q) \vdash p}\,\vee\text{e}.$$

2.
$$\dfrac{\dfrac{\dfrac{}{p \wedge q, r \wedge s \vdash p \wedge q}\,\text{Ax}}{p \wedge q, r \wedge s \vdash p}\,\wedge\text{e,g} \qquad \dfrac{\dfrac{}{p \wedge q, r \wedge s \vdash r \wedge s}\,\text{Ax}}{p \wedge q, r \wedge s \vdash r}\,\wedge\text{e,d}}{p \wedge q, r \wedge s \vdash p \wedge r}\,\wedge\text{i}.$$

3.
$$\dfrac{\dfrac{}{p, q \wedge r \vdash p}\,\text{Ax} \qquad \dfrac{\dfrac{}{p, q \wedge r \vdash q \wedge r}\,\text{Ax}}{p, q \wedge r \vdash q}\,\wedge\text{e,g}}{p, q \wedge r \vdash p \wedge q}\,\wedge\text{i}.$$

4.
$$\dfrac{\dfrac{\dfrac{}{p, \neg p \vdash p}\,\text{Ax} \qquad \dfrac{}{p, \neg p \vdash \neg p}\,\text{Ax}}{p, \neg p \vdash \bot}\,\neg\text{e}}{p \vdash \neg\neg p}\,\neg\text{i}.$$

5.
$$\dfrac{\dfrac{\dfrac{\dfrac{}{p, \neg\neg\neg p, \neg p \vdash p}\,\text{Ax} \qquad \dfrac{}{p, \neg\neg\neg p, \neg p \vdash \neg p}\,\text{Ax}}{\neg\neg\neg p, p, \neg p \vdash \bot}\,\neg\text{e}}{\neg\neg\neg p, p \vdash \neg\neg p}\,\neg\text{i} \qquad \dfrac{}{\neg\neg\neg p, p \vdash \neg\neg\neg p}\,\text{Ax}}{\neg\neg\neg p \vdash \bot}\,\neg\text{e}}{\neg\neg\neg p \vdash \neg p}\,\neg\text{i}.$$

TD 10.3 Lois de De Morgan

1.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(p\vee q),p\vdash p}\,\mathrm{Ax}}{\neg(p\vee q),p\vdash p\vee q}\,\vee\mathrm{i,g} \quad \dfrac{}{\neg(p\vee q),p\vdash\neg(p\vee q)}\,\mathrm{Ax}}{\dfrac{\neg(p\vee q),p\vdash\bot}{\neg(p\vee q)\vdash\neg p}\,\neg\mathrm{i}}\,\neg\mathrm{e} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\neg(p\vee q),q\vdash q}\,\mathrm{Ax}}{\neg(p\vee q),q\vdash p\vee q}\,\vee\mathrm{i,d} \quad \dfrac{}{\neg(p\vee q),q\vdash\neg(p\vee q)}\,\mathrm{Ax}}{\dfrac{\neg(p\vee q),q\vdash\bot}{\neg(p\vee q)\vdash\neg q}\,\neg\mathrm{i}}\,\neg\mathrm{e}}{\neg(p\vee q)\vdash\neg p\wedge\neg q}\,\wedge\mathrm{i}$$

2.

$$\dfrac{\dfrac{}{\neg p\vee\neg q,p\vee q\vdash p\vee q}\,\mathrm{Ax} \quad \dfrac{\dfrac{\dfrac{}{\neg p\wedge\neg q,p\vee q,p\vdash p}\,\mathrm{Ax} \quad \dfrac{\dfrac{}{\neg p\wedge\neg q,p\vee q,p\vdash\neg p\wedge\neg q}\,\mathrm{Ax}}{\neg p\wedge\neg q,p\vee q,p\vdash\neg p}\,\wedge\mathrm{e,d}}{\neg p\wedge\neg q,p\vee q,p\vdash\bot}\,\neg\mathrm{e} \quad \dfrac{\dfrac{\dfrac{}{\neg p\wedge\neg q,p\vee q,q\vdash q}\,\mathrm{Ax} \quad \dfrac{\dfrac{}{\neg p\wedge\neg q,p\vee q,q\vdash\neg q\wedge\neg q}\,\mathrm{Ax}}{\neg p\wedge\neg q,p\vee q,q\vdash\neg q}\,\wedge\mathrm{e,d}}{\neg p\wedge\neg q,p\vee q,q\vdash\bot}\,\neg\mathrm{e}}{\neg p\wedge\neg q,p\vee q\vdash\bot}\,\vee\mathrm{e}}{\dfrac{\neg p\wedge\neg q\vdash\bot}{\neg p\wedge\neg q\vdash\neg(p\vee q)}\,\neg\mathrm{i}}$$

TD 10.4 Distributivités entre $\wedge$ et $\vee$

1.

$$\dfrac{\dfrac{\dfrac{}{p\wedge(q\vee r)\vdash p\wedge(q\vee r)}\,\mathrm{Ax}}{p\wedge(q\vee r)\vdash q\vee r}\,\wedge\mathrm{e,d} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{q,p\wedge(q\vee r)\vdash p\wedge(q\vee r)}\,\mathrm{Ax}}{p,p\wedge(q\vee r)\vdash p}\,\wedge\mathrm{e,g} \quad \dfrac{}{p,q\wedge(q\vee r)\vdash q}\,\mathrm{Ax}}{q,p\wedge(q\vee r)\vdash p\vee q}\,\wedge\mathrm{i}}{q,p\wedge(q\vee r)\vdash(q\vee r)\wedge(p\wedge r)}\,\vee\mathrm{i,g} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{r,p\wedge(q\vee r)\vdash p\wedge(q\vee r)}\,\mathrm{Ax}}{r,p\wedge(q\vee r)\vdash p}\,\wedge\mathrm{e,g} \quad \dfrac{}{r,q\wedge(q\vee r)\vdash r}\,\mathrm{Ax}}{r,p\wedge(q\vee r)\vdash p\wedge r}\,\wedge\mathrm{i}}{r,p\wedge(q\vee r)\vdash(p\wedge q)\vee(p\wedge r)}\,\vee\mathrm{i,d}}{p\wedge(q\vee r)\vdash(p\wedge q)\vee(p\wedge r)}\,\vee\mathrm{e}$$

2. On nomme $\varphi = (p \wedge q) \vee (p \wedge r)$.

$$\dfrac{\dfrac{\dfrac{}{\varphi \vdash (p \wedge q) \vee (p \wedge r)}\,\text{Ax} \quad \dfrac{\dfrac{}{\varphi, p \wedge q \vdash p \wedge q}\,\text{Ax}}{\varphi, p \wedge q \vdash p}\,\wedge\text{e,g} \quad \dfrac{\dfrac{}{\varphi, p \wedge r \vdash p \wedge r}\,\text{Ax}}{\varphi, p \wedge r \vdash p}\,\wedge\text{e,g}}{\varphi \vdash p}\,\vee\text{e} \quad \dfrac{\dfrac{\dfrac{}{\varphi \vdash (p \wedge q) \vee (p \wedge r)}\,\text{Ax} \quad \dfrac{\dfrac{}{\varphi, p \wedge q \vdash p \wedge q}\,\text{Ax}}{\varphi, p \wedge q \vdash q}\,\wedge\text{e,d}}{\varphi, p \wedge q \vdash q \vee r}\,\vee\text{i,g} \quad \dfrac{\dfrac{}{\varphi, p \wedge r \vdash p \wedge r}\,\text{Ax}}{\varphi, p \wedge r \vdash r}\,\wedge\text{e,d}}{\varphi, p \wedge r \vdash q \vee r}\,\vee\text{i,d}}{\varphi \vdash q \vee r}\,\vee\text{e}}{(p \wedge q) \vee (p \wedge r) \vdash p \wedge (q \vee r)}\,\wedge\text{i}.$$

TD 10.5 Implications

1. $$\dfrac{\dfrac{}{p, q \vdash q}\,\text{Ax}}{q \vdash p \to q}\,\to\text{i}$$

2. $$\dfrac{\dfrac{\dfrac{}{p, p \wedge q \vdash p \wedge q}\,\text{Ax}}{p, p \wedge q \vdash q}\,\wedge\text{e,d}}{p \wedge q \vdash p \to q}\,\to\text{i}$$

3. $$\dfrac{\dfrac{\dfrac{}{p, p \to q \vdash p}\,\text{Ax} \quad \dfrac{\dfrac{}{p, p \to q \vdash p \to q}\,\text{Ax} \quad \dfrac{}{p, p \to q \vdash p}\,\text{Ax}}{p, p \to q \vdash q}\,\to\text{e}}{p, p \to q \vdash p \wedge q}\,\wedge\text{i}}{p \to q \vdash p \to (p \wedge q)}\,\to\text{i}$$

4. $$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg q, p \to q, p \vdash p \to q}\,\text{Ax} \quad \dfrac{}{\neg q, p \to q, p \vdash p}\,\text{Ax}}{\neg q, p \to q, p \vdash q}\,\to\text{e} \quad \dfrac{}{\neg q, p \to q, p \vdash \neg q}\,\text{Ax}}{\neg q, p \to q, p \vdash \bot}\,\neg\text{e}}{\neg q, p \to q \vdash \neg p}\,\neg\text{i}}{p \to q \vdash \neg q \to \neg p}\,\to\text{i}$$

5. $$\dfrac{\dfrac{\dfrac{}{p \wedge q, p \to r \vdash p \wedge q}\,\text{Ax}}{p \wedge q, p \to r \vdash p}\,\wedge\text{e,g} \quad \dfrac{}{p \wedge q, p \to r \vdash p \to r}\,\text{Ax}}{p \wedge q, p \to r \vdash r}\,\to\text{e}}{p \to r \vdash (p \wedge q) \to r}\,\to\text{i}$$

6. $$\dfrac{\dfrac{\dfrac{}{p, q \vdash p}\,\text{Ax}}{p \vdash q \to p}\,\to\text{i}}{\vdash p \to (q \to p)}\,\to\text{i}$$

7. $$\dfrac{\dfrac{}{p \to q, p \vdash p \to q}\,\text{Ax} \quad \dfrac{}{p \to q, p \vdash p}\,\text{Ax}}{p, p \to q \vdash q}\,\to\text{e}}{p \vdash (p \to q) \to q}\,\to\text{i}$$

TD 10.6 Implications (partie 1 : simplifications)

1.
$$\dfrac{\dfrac{}{p, p \to \neg p \vdash p}\,\text{Ax} \qquad \dfrac{\dfrac{}{p, p \to \neg p \vdash p \to \neg p}\,\text{Ax} \qquad \dfrac{}{p, p \to \neg p \vdash p}\,\text{Ax}}{p, p \to \neg p \vdash \neg p}\,\to\text{e}}{\dfrac{p, p \to \neg p \vdash \bot}{p \to \neg p \vdash \neg p}\,\neg\text{i}}\,\neg\text{e}$$

2.
$$\dfrac{\dfrac{\dfrac{}{p, p \to q, \neg q \vdash p \to q}\,\text{Ax} \qquad \dfrac{}{p, p \to q, \neg q \vdash p}\,\text{Ax}}{p, p \to q, \neg q \vdash q}\,\to\text{e} \qquad \dfrac{}{p, p \to q, \neg q \vdash \neg q}\,\text{Ax}}{\dfrac{p, p \to q, \neg q \vdash \bot}{p \to q, \neg q \vdash \neg p}\,\neg\text{i}}\,\neg\text{e}$$

3.
$$\dfrac{\dfrac{}{p \to q, p \vee q}\,\text{Ax} \qquad \dfrac{\dfrac{}{p \to q, p \vee q, p \vdash p \to q}\,\text{Ax} \qquad \dfrac{}{p \to q, p \vee q, p \vdash p}\,\text{Ax}}{p \to q, p \vee q, p \vdash q}\,\to\text{e} \qquad \dfrac{}{p \to q, p \vee q, q \vdash q}\,\text{Ax}}{p \to q, p \vee q \vdash q}\,\vee\text{e}$$

4.
$$\dfrac{\dfrac{\dfrac{}{p, p \to q, p \to \neg q \vdash p \to}\,\text{Ax} \qquad \dfrac{}{p, p \to q, p \to \neg q \vdash p}\,\text{Ax}}{p, p \to q, p \to \neg q \vdash q}\,\to\text{e} \qquad \dfrac{\dfrac{}{p, p \to q, p \to \neg q \vdash p \to \neg q}\,\text{Ax} \qquad \dfrac{}{p, p \to q, p \to \neg q \vdash p}\,\text{Ax}}{p, p \to q, p \to \neg q \vdash \neg q}\,\to\text{e}}{\dfrac{p, p \to q, p \to \neg q \vdash \bot}{p \to q, p \to \neg q \vdash \neg p}\,\neg\text{i}}\,\neg\text{e}$$

5. On pose $\Gamma = p, p \to (q \vee r), \neg q, \neg r, q \vee r$.

$$\dfrac{\dfrac{\dfrac{}{p, p \to (q \vee r), \neg q, \neg r \vdash p \to (q \vee r)}\,\text{Ax} \qquad \dfrac{}{p, p \to (q \vee r), \neg q, \neg r \vdash p}\,\text{Ax}}{p, p \to (q \vee r), \neg q, \neg r \vdash q \vee r}\,\to\text{e} \qquad \dfrac{\dfrac{}{\Gamma \vdash q \vee r}\,\text{Ax} \qquad \dfrac{\dfrac{}{\Gamma, q \vdash q}\,\text{Ax} \qquad \dfrac{}{\Gamma, q \vdash \neg q}\,\text{Ax}}{\Gamma, q \vdash \bot}\,\neg\text{e} \qquad \dfrac{\dfrac{}{\Gamma, r \vdash r}\,\text{Ax} \qquad \dfrac{}{\Gamma, r \vdash \neg r}\,\text{Ax}}{\Gamma, r \vdash \bot}\,\neg\text{e}}{p, p \to (q \vee r), \neg q, \neg r, q \vee r \vdash \bot}\,\vee\text{e}}{\dfrac{p, p \to (q \vee r), \neg q, \neg r \vdash \neg(q \vee r)}{p, p \to (q \vee r), \neg q, \neg r \vdash \bot}\,\neg\text{i}}\,\neg\text{e}$$
$$\dfrac{p, p \to (q \vee r), \neg q, \neg r \vdash \bot}{p \to (q \vee r), \neg q, \neg r \vdash \neg p}\,\neg\text{i}$$

6. On pose $\Gamma = p \to (q \to r), p, \neg r, q$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash p \to (q \to r)}\,\text{Ax} \quad \dfrac{}{\Gamma \vdash p}\,\text{Ax}}{\Gamma \vdash q \to r}\,{\to}\text{e} \quad \dfrac{}{\Gamma \vdash q}\,\text{Ax}}{\Gamma \vdash r}\,{\to}\text{e} \quad \dfrac{}{\Gamma \vdash \neg r}\,\text{Ax}}{\dfrac{\Gamma \vdash \bot}{p \to (q \to r), p \neg r \vdash \neg q}\,\neg\text{i}}\,\neg\text{e}$$

7. On pose $\Gamma = p \to (q \to r), p, \neg r, q$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash p \to (q \to r)}\,\text{Ax} \quad \dfrac{}{\Gamma \vdash p}\,\text{Ax}}{\Gamma \vdash q \to r}\,{\to}\text{e} \quad \dfrac{}{\Gamma \vdash q}\,\text{Ax}}{\Gamma \vdash r}\,{\to}\text{e} \quad \dfrac{}{\Gamma \vdash \neg r}\,\text{Ax}}{\dfrac{\Gamma \vdash \bot}{p \to (q \to r), p, \neg r \vdash \neg q}\,{\to}\text{i}}\,\neg\text{e}$$

8. On pose $\Gamma = q, p \to (q \to r), q \to p$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash p \to (q \to r)}\,\text{Ax} \quad \dfrac{\dfrac{}{\Gamma \vdash q \to p}\,\text{Ax} \quad \dfrac{}{\Gamma \vdash q}\,\text{Ax}}{\Gamma \vdash p}\,{\to}\text{e}}{\Gamma \vdash q \to r}\,{\to}\text{e} \quad \dfrac{}{\Gamma \vdash q}\,\text{Ax}}{\Gamma \vdash r}\,{\to}\text{e}}{p \to (q \to r), q \to p, \vdash q \to r}\,{\to}\text{i}$$

9.

$$\dfrac{\dfrac{\dfrac{}{p \to (p \to q), p \vdash p \to (p \to q)}\,\text{Ax} \quad \dfrac{}{p \to (p \to q), p \vdash p}\,\text{Ax}}{p \to (p \to q), p \vdash p \to q}\,{\to}\text{e} \quad \dfrac{}{p \to (p \to q), p \vdash p}\,\text{Ax}}{p \to (p \to q), p \vdash q}\,{\to}\text{e}.$$

TD 10.7 Implications (partie 2 : transformations)

1. *c.f.* Exercice 5, question 4.

2.

$$\dfrac{\dfrac{\dfrac{}{p \to q, p \wedge r \vdash p \to q}\,\text{Ax} \quad \dfrac{\dfrac{\dfrac{}{p \to q, p \wedge r \vdash p \wedge r}\,\text{Ax}}{p \to q, p \wedge r \vdash p}\,\wedge\text{e,g}}{p \to q, p \wedge r \vdash q}\,\to\text{e} \quad \dfrac{\dfrac{}{p \to q, p \wedge r \vdash p \wedge r}\,\text{Ax}}{p \to q, p \wedge r \vdash r}\,\wedge\text{e,g}}{p \to q, p \wedge r \vdash q \wedge r}\,\wedge\text{i}}{p \to q \vdash (p \wedge q) \to (q \wedge r)}\,\to\text{i}$$

3.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(p \wedge r) \to (q \wedge r), r, p \vdash (p \wedge r) \to (q \wedge r)}\,\text{Ax} \quad \dfrac{\dfrac{}{(p \wedge r) \to (q \wedge r), r, p \vdash p}\,\text{Ax} \quad \dfrac{}{(p \wedge r) \to (q \wedge r), r, p \vdash r}\,\text{Ax}}{(p \wedge r) \to (q \wedge r), r, p \vdash p \wedge r}\,\wedge\text{i}}{(p \wedge r) \to (q \wedge r), r, p \vdash q \wedge r}\,\to\text{e}}{(p \wedge r) \to (q \wedge r), r, p \vdash q}\,\vee\text{e,g}}{(p \wedge r) \to (q \wedge r), r \vdash p \wedge q}\,\to\text{i}$$

.

4.

$$\dfrac{\dfrac{\dfrac{}{p \to q, p \vee r \vdash p \vee r}\,\text{Ax} \quad \dfrac{\dfrac{\dfrac{}{p \to q, p \vee r, p \vdash p \to q}\,\text{Ax} \quad \dfrac{}{p \to q, p \vee r, p \vdash p}\,\text{Ax}}{p \to q, p \vee r, p \vdash q}\,\to\text{e}}{p \to q, p \vee r, p \vdash q \vee r}\,\vee\text{i,g} \quad \dfrac{\dfrac{}{p \to q, p \vee r, r \vdash r}\,\text{Ax}}{p \to q, p \vee r, r \vdash q \vee r}\,\vee\text{i,d}}{p \to q, p \vee r \vdash q \vee r}\,\vee\text{e}}{p \to q \vdash (p \vee r) \to (q \vee r)}\,\to\text{i}$$

5. On pose $\Gamma = p, q, (p \wedge q) \to r, \neg r$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash (p \wedge q) \to r}\,\text{Ax} \quad \dfrac{\dfrac{}{\Gamma \vdash p}\,\text{Ax} \quad \dfrac{}{\Gamma \vdash q}\,\text{Ax}}{\Gamma \vdash p \wedge q}\,\wedge\text{i}}{\Gamma \vdash r}\,\to\text{e} \quad \dfrac{}{\Gamma \vdash \neg r}\,\text{Ax}}{\Gamma \vdash \bot}\,\neg\text{e}}{\neg r, p, (p \wedge q) \to r \vdash \neg q}\,\neg\text{i}}{(p \wedge q) \to r, \neg r, \vdash p \to \neg q}\,\to\text{i}$$

6. On pose $\Gamma =$

TRAVAUX DIRIGÉS

11

PREUVES

Sommaire

TD 11.1 Formalisation **257**
TD 11.2 Variables libres et liées, clôture universelle **257**
TD 11.3 Substitution **259**
TD 11.4 Quelques arbres de preuves **260**
TD 11.5 Distributivité des quantificateurs **260**
TD 11.6 Semi-distributivité des quantificateurs **261**
TD 11.7 Logique classique du premier ordre **261**

TD 11.1 Formalisation

1. (1) $\forall e,\ (\mathsf{etudiant}(e) \to \mathsf{raison}(e))$;
   (2) $\forall e,\ (\mathsf{raison}(e) \to \mathsf{humain}(e))$;
   (3) $\forall e,\ \mathsf{raison}(e) \to \neg\mathsf{elephant}(e)$;
   (4) $\forall e,\ ((\mathsf{animal}(e) \wedge \neg\mathsf{chien}(e)) \to \forall \ell,\ (\mathsf{logicien}(\ell) \to \mathsf{gentil\_avec}(e, \ell))$
   (5) $\forall h,\ \exists e,\ \mathsf{elephant}(e) \to \mathsf{cherche}(h, e)$;
   (6) $\forall e,\ [(\exists a,\ \mathsf{aime}(x, y)) \wedge (\exists a,\ \neg\mathsf{aime}(e, a))]$;
2. $\forall x,\ \mathsf{entier}(x) \to [\exists y,\ \mathsf{entier}(y) \to (\mathsf{successeur}(x, y) \to (\forall z,\ \mathsf{entier}(z) \to \neg\mathsf{inf}(z, x) \to \mathsf{inf}(y, z)))]$.

TD 11.2 Variables libres et liées, clôture universelle

1.

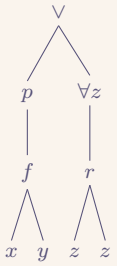


FIGURE TD 11.1 – Arbre de syntaxe de la formule $p\big(f(x,y)\big) \vee \forall z,\ r(z,z)$

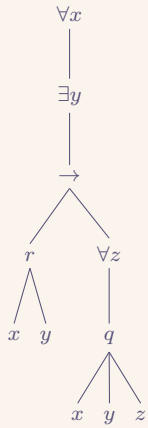


FIGURE TD 11.2 – Arbre de syntaxe de la formule $\forall x,\ \exists y,\ \big(r(x,y) \to \forall z,\ q(x,y,z)\big)$

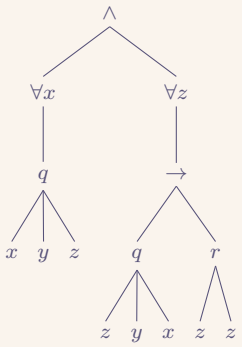


FIGURE TD 11.3 – Arbre de syntaxe de la formule $\forall x,\ (x,y,z) \wedge \forall z,\ \big(q(z,y,x) \to r(z,z)\big)$

2. Pour la formule (F_1), l'ensemble des constantes est $\mathcal{S}_0 = \varnothing$, l'ensemble des fonctions est $\mathcal{S} = \big\{f(2)\big\}$. Pour la formule (F_2), l'ensemble des constantes est $\mathcal{S}_0 = \varnothing$, l'ensemble des fonctions est $\mathcal{S} = \varnothing$. Pour la formule (F_3), l'ensemble des constantes est $\mathcal{S}_0 = \varnothing$, l'ensemble des fonctions est $\mathcal{S} = \varnothing$.
3. Pour la formule (F_1), l'ensemble des symboles de prédicats est $\mathcal{P} = \big\{p(1), r(2)\big\}$. Pour la formule (F_2), l'ensemble des symboles de prédicats est $\mathcal{P} = \big\{r(2), q(3)\big\}$. Pour la formule (F_3), l'ensemble des symboles de prédicats est $\mathcal{P} = \big\{q(3), r(2)\big\}$.
4. Pour la formule (F_1), $\mathsf{FV} = \{x, y\}$ et $\mathsf{BF} = \{z\}$. Pour la formule (F_2), $\mathsf{FV} = \varnothing$ et $\mathsf{BF} = \{x, y, z\}$. Pour la formule (F_3), $\mathsf{FV} = \{x, y, z\}$ et $\mathsf{BF} = \{x, z\}$.

TD 11.3 Substitution

1.

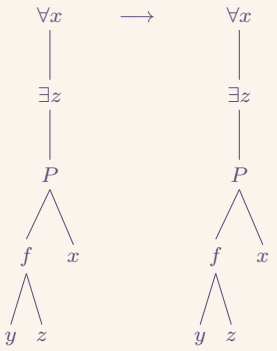


FIGURE TD 11.4 – Calcul de la substitution $F[x \mapsto f(y,z)]$, pour la formule 1

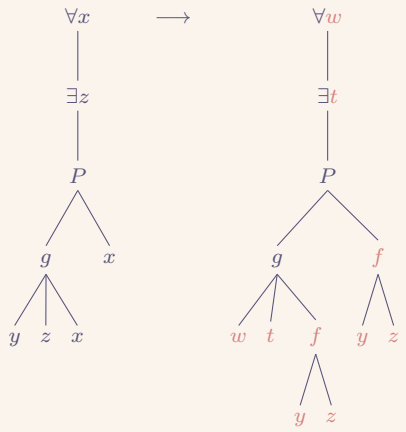


FIGURE TD 11.5 – Calcul de la substitution $F[x \mapsto f(y,z)]$, pour la formule 2

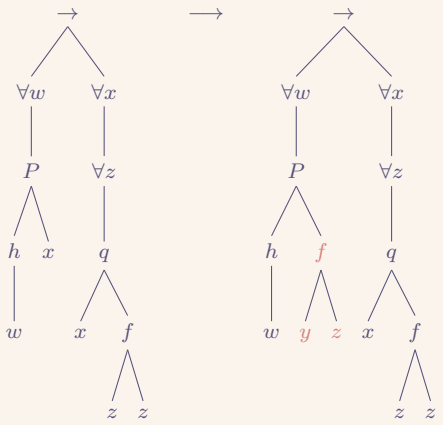


FIGURE TD 11.6 – Calcul de la substitution $F[x \mapsto f(y,z)]$, pour la formule 3

TD 11.4 Quelques arbres de preuves

1. *c.f.* exemple chapitre 6.
2. $$\dfrac{\dfrac{}{\exists t,\ \forall x,\ \varphi(t,x) \vdash \exists t,\ \forall x,\ \varphi(t,x)}\ \text{Ax} \qquad \dfrac{\dfrac{\dfrac{}{\exists t,\ \forall x,\ \varphi(t,x); \forall x,\ \varphi(t,x) \vdash \forall x,\ \varphi(t,x)}\ \text{Ax}}{\exists t,\ \forall x,\ \varphi(t,x); \forall x,\ \varphi(t,x) \vdash \varphi(t,x)}\ \forall\text{e}}{\exists t,\ \forall x,\ \varphi(t,x); \forall x,\ \varphi(t,x) \vdash \forall y,\ \exists x,\ \varphi(x,y)}\ \exists\text{i}}{\exists t,\ \forall x,\ \varphi(t,x) \vdash \forall y,\ \exists x,\ \varphi(x,y)}\ \exists\text{e}.$$
3. $$\dfrac{\dfrac{\dfrac{}{\forall x,\ \varphi; \exists x,\ \neg\varphi \vdash \exists x,\ \neg\varphi}\ \text{Ax} \qquad \dfrac{\dfrac{}{\forall x,\ \varphi; \exists x,\ \neg\varphi; \neg\varphi \vdash \forall x,\ \varphi}\ \text{Ax}}{\forall x,\ \varphi; \exists x,\ \neg\varphi; \neg\varphi \vdash \varphi}\ \forall\text{e} \qquad \dfrac{}{\forall x,\ \varphi; \exists x,\ \neg\varphi; \neg\varphi \vdash \neg\varphi}\ \text{Ax}}{\forall x,\ \varphi; \exists x,\ \neg\varphi; \neg\varphi \vdash \bot}\ \neg\text{e}}{\forall x,\ \varphi; \exists x,\ \neg\varphi \vdash \bot}\ \exists\text{e}}{\forall x,\ \varphi \vdash \neg(\exists x,\ \neg\varphi)}\ \neg\text{i}.$$
4. $$\dfrac{\dfrac{\dfrac{}{\exists x,\ \neg\varphi; \forall x,\ \varphi \vdash \exists x,\ \neg\varphi}\ \text{Ax} \qquad \dfrac{\dfrac{}{\exists x,\ \neg\varphi; \forall x,\ \varphi; \neg\varphi \vdash \neg\varphi}\ \text{Ax} \qquad \dfrac{\dfrac{}{\exists x,\ \neg\varphi; \forall x,\ \varphi; \neg\varphi \vdash \forall x,\ \varphi}\ \text{Ax}}{\exists x,\ \neg\varphi; \forall x,\ \varphi; \neg\varphi \vdash \varphi}\ \forall\text{e}}{\exists x,\ \neg\varphi; \forall x,\ \varphi; \neg\varphi \vdash \bot}\ \neg\text{e}}{\exists x,\ \neg\varphi; \forall x,\ \varphi \vdash \bot}\ \exists\text{e}}{\exists x,\ \neg\varphi \vdash \neg(\forall x,\ \varphi)}\ \neg\text{i}.$$

TD 11.5 Distributivité des quantificateurs

1. $$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x,\ (\varphi \wedge \psi) \vdash \forall x,\ (\varphi \wedge \psi)}\ \text{Ax}}{\forall x,\ (\varphi \wedge \psi) \vdash \varphi \wedge \psi}\ \forall\text{e}}{\forall x,\ (\varphi \wedge \psi) \vdash \varphi}\ \wedge\text{e,g}}{\forall x,\ (\varphi \wedge \psi) \vdash \forall x,\ \varphi}\ \forall\text{i} \qquad \dfrac{\vdots}{\forall x,\ (\varphi \wedge \psi) \vdash \forall x,\ \psi}}{\forall x,\ (\varphi \wedge \psi) \vdash (\forall x,\ \varphi) \wedge (\forall x,\ \psi)}.$$

TD 11.6 Semi-distributivité des quantificateurs

TD 11.7 Logique classique du premier ordre

1.

$$\dfrac{\dfrac{}{\neg(\exists x,\ \neg\varphi),\neg\varphi\vdash\neg(\exists x,\ \neg\varphi)}\ \text{Ax}\qquad\dfrac{\dfrac{}{\neg(\exists x,\ \neg\varphi),\neg\varphi\vdash\neg\varphi}\ \text{Ax}}{\neg(\exists x,\ \neg\varphi),\neg\varphi\vdash\exists x,\ \neg\varphi}\ \exists\text{i}}{\dfrac{\dfrac{\neg(\exists x,\ \neg\varphi),\neg\varphi\vdash\bot}{\neg(\exists x,\ \neg\varphi)\vdash\varphi}\ \text{Abs}}{\neg(\exists x,\ \neg\varphi)\vdash\forall x,\ \varphi}\ \forall\text{i}}\ \neg\text{e}.$$

TRAVAUX DIRIGÉS

12

ALGORITHMES D'APPROXIMATION

Sommaire

TD 12.1 Un problème proche de KNAPSACK **263**
TD 12.2 Le problème BINPACKING **264**
TD 12.3 Le problème VOYAGEURCOMMERCE **265**

TD 12.1 Un problème proche de KNAPSACK

1. $$\text{SOMMEMAX} : \begin{cases} \textbf{Entrée} & : \text{Un entier } n \in \mathbb{N}, \text{ une suite finie } (a_i)_{i\in[\![1,n]\!]} \in \mathbb{N}^n, \\ & \quad \text{un entier } B \in \mathbb{N} \text{ et un seuil } K \in \mathbb{N}, \\ \textbf{Sortie} & : \text{Existe-t-il } I \subseteq [\![1,n]\!] \text{ avec } K \geqslant \sum_{i\in I} a_i \geqslant B \,? \end{cases}$$

2. On a SOMMEMAX $\preccurlyeq_{\text{p}}$ KNAPSACK, mais on ne peut pas faire une réduction.

3. Soit $(w_1, \dots, w_n)$ les entrées du problème PARTITION. On pose $K = B = \sum_{i=1}^{n} a_i$. On construit l'entrée $(n, (2w_i)_{i\in[\![1,n]\!]}, B, K)$ de SOMMEMAX.

$$\begin{aligned} (n, (2w_i)_{i\in[\![1,n]\!]}, K, B) \in \text{SOMMEMAX}^+ &\iff \exists I \subseteq [\![1,n]\!],\ B \leqslant \textstyle\sum_{i\in I} 2a_i \leqslant K \\ &\iff \exists I \subseteq [\![1,n]\!],\ \textstyle\sum_{i=1}^{n} a_i \leqslant \sum_{i\in I} 2a_i \leqslant \sum_{i=1}^{n} a_i \\ &\iff \exists I \subseteq [\![1,n]\!],\ \textstyle\sum_{i\in I} 2a_i = \sum_{i=1}^{n} a_i \\ &\iff \exists I \subseteq [\![1,n]\!],\ \textstyle\sum_{i\in I} a_i = \sum_{i\in[\![1,n]\!]\setminus I} a_i \\ &\iff (w_1, \dots, w_n) \in \text{PARTITION}^+ \end{aligned}$$

Or, comme PARTITION est **NP**-difficile (*c.f.* TD 8), on en déduit que SOMMEMAX est **NP**-difficile.

4.

Algorithme TD 12.1 Algorithme glouton pour résoudre le problème SommeMax$_\mathcal{O}$ en $\mathcal{O}(n)$

```
somme ← 0
pour i ∈ [[1, n]] faire
    si somme + a_i ⩽ B alors
        somme ← somme + a_i
retourner somme
```

5. L'algorithme n'est pas optimal : avec l'entrée $(1, B)$, l'algorithme renvoie 1, mais la valeur optimale est B. Cet algorithme n'est pas une ρ-approximation, pour $\rho \in \mathbb{R}$.
6.

Algorithme TD 12.2 Algorithme glouton pour résoudre le problème SommeMax$_\mathcal{O}$ en $\mathcal{O}(n \ln n)$

```
On trie, par ordre décroissant, les entrées (a_1, ..., a_n) avec un tri rapide.
somme ← 0
pour i ∈ [[1, n]] faire
    si somme + a_i ⩽ B alors
        somme ← somme + a_i
retourner somme
```

7. Soit e une entrée. Soit S la solution, *non nécessairement optimale*, de l'algorithme. Soit $S^\star$ la solution optimale. On a $S^\star \leqslant B$.
   - Si $S \geqslant \frac{B}{2}$, alors $\frac{S}{S^\star} \geqslant \frac{B/2}{B} = \frac{1}{2}$.
   - Si $S < \frac{B}{2}$, alors, en supposant que $(a_1, \dots, a_n)$ est trié par ordre décroissant, on a, pour $i \in [\![1, n]\!]$, $a_i > B$ ou $a_i < \frac{B}{2}$.
     - Si $\forall i \in [\![1, n]\!]$, $a_i > B$, alors $S = 0 = S^\star$.
     - Soit $i = \arg\min_{i \in [\![1,n]\!]}(a_i < B/2)$. On pose I_{algo} les valeurs de i telles que a_i ait été ajoutée à somme. On pose $i' = \max I_{\text{algo}}$.
       Si $i' < n$, alors par l'algorithme, l'objet d'indice n ne rentre pas dans le sac. Or, la valeur du sac est inférieure stricte à $B/2$, et $a_n < B/2$, ce qui est absurde (l'objet rentre dans le sac).
       Ainsi, $i' = n$, et l'algorithme a mis dans le sac tous les objets de poids inférieurs ou égal à $B/2$, donc tous les objets ont un poids inférieur ou égal à B. Donc $S^\star = S$.
8.

Algorithme TD 12.3 Algorithme glouton pour résoudre le problème SommeMax$_\mathcal{O}$ en $\mathcal{O}(n)$

Entrée $(a_i)_{i \in [\![1,n]\!]}$ une suite finie

```
somme ← 0
maxi ← 0
pour i ∈ [[1, n]] faire
    si a_i ⩾ B/2 et a_i ⩽ B alors
        maxi ← a_i
    sinon si a_i + somme ⩽ B alors
        somme ← somme + a_i
retourner max(somme, a_i)
```

TD 12.2 Le problème BinPacking

1.

$$\text{BinPacking}_\mathcal{O} : \begin{cases} \textbf{Entrée} & : \text{un entier } n \in \mathbb{N}, \text{ une suite finie } (t_i)_{i \in [\![1,n]\!]} \in \mathbb{N}^n \text{ et } C \in \mathbb{N}^\star \\ \textbf{Sortie} & : \max\left\{K \in \mathbb{N} \mid \exists \varphi : [\![1, n]\!] \to [\![1, K]\!],\ \forall i \in [\![1, K]\!],\ \sum_{j \in \varphi^{-1}(\{i\})} t_k \leqslant C\right\} \end{cases}$$

2. On réduit Partition à BinPacking. On construit les entrées à l'aide de l'algorithme ci-dessous.

Algorithme TD 12.4 Réduction polynômiale de Partition à BinPacking

```
1 : B ← ∑_{i=1}^{n} t_i
2 : si B ≡ 0 [2] et B ≠ 0 alors
3 :     retourner (n, T, B/2, 2)
4 : sinon
5 :     retourner (3, (1, 1, 1), 2, 1)
```

Montrons que BinPacking $\in$ **NP**. On choisit pour ensemble de certificats l'ensemble des fonctions $\varphi : [\![1, n]\!] \to [\![1, K]\!]$, pour $(n, K) \in \mathbb{N}^2$. On devine un certificat φ, on vérifie que $\forall i \in [\![1, K]\!]$, $\sum_{j \in \varphi^{-1}(\{i\})} t_j \leqslant C$ en temps polynômiale.

3.

4. On considère l'entrée $(3, (3, 3, 3), 3)$ de l'algorithme. La solution optimale est 3, et c'est la réponse donnée par l'algorithme. On considère l'entrée $(3, (2, 3, 1), 3)$ de l'algorithme. La solution optimale est 2, mais l'algorithme renvoie 3.

5. Pour une instance (n, t, C), un minorant est $\left\lceil \frac{V}{C} \right\rceil$ avec $V = \sum_{i=1}^{n} t_i$, un majorant est n.

TD 12.3 Le problème VoyageurCommerce

1. Soit $\mathscr{C} = \{a_1, a_2\} - \cdots - \{a_n, a_{n+1}\}$ et $a_{n+1} = a_1$, un tour où $\bigcup_{i \in [\![1,n]\!]} \{a_i\} = S$. Soit $\{a, b\}$ une arrête. Posons $\mathscr{C}' = \mathscr{C} \setminus \{\{a, b\}\}$, et $H = (S', \mathscr{C}')$, où $S' = S$. Montrons que H est un arbre couvrant.
   - Montrons que H est connexe. Soit un couple sommets $(x, y) \in S^2$ tel que $x \neq y$. $\mathscr{C}$ est un tour, donc en re-numérotant
   $$x - b_1 - b_2 - \cdots - b_{n-1} - x.$$
   Soit $i \in [\![1, n-1]\!]$ tel que $b_i = y$. Soit $j \in [\![1, n]\!]$ tel que $\{b_j, b_{j+1}\} = \{a, b\}$.
   Cas 1 Si $j < i$, alors $y - b_{i+1} - \cdots - b_{n-1} - x$ est un chemin.
   Cas 2 Si $j > i$, alors $x - b_1 - \cdots - b_{i-1} - y$ est un chemin.
   Donc H est connexe.
   - On a $\#\mathscr{C}' = n - 1$ et H est connexe, donc acyclique.
   - Enfin $S' = S$, donc H est couvrant.
2. Soit $T^\star$ le tour de coût minimal. On note $c(T)$ le coût d'un tour T. On a $c(T^\star) \geqslant c(H)$ où H est l'arbre défini dans la question précédente (par suppression d'arête), d'où $c(T^\star) = c(H^\star)$ où $H^\star$ est un arbre couvrant de poids minimal (par définition de poids minimal).
3. Non, avec le graphe ci-dessous est un contre-exemple.

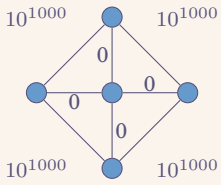


Figure TD 12.1 – Contre exemple

4. On obtient l'arbre couvrant de poids minimal suivant.

FIGURE TD 12.2 – Arbre couvrant de poids minimal

Un parcours en profondeur est

$$a \to e \to c \to d \to h \rightsquigarrow g \to b \to f.$$

TRAVAUX DIRIGÉS

13

JEUX

Sommaire

TD 13.1 Attracteurs pour le jeu de la soustraction généralisé **267**
TD 13.2 Chomp **268**
TD 13.3 Hex **268**
TD 13.4 Calcul de MinMax **268**
TD 13.5 Calcul de MinMax avec mémoïsation **268**

TD 13.1 Attracteurs pour le jeu de la soustraction généralisé

1. On considère une partie nulle.

   — Le nombre d'allumettes, à chaque tours, décroît strictement dans $(\mathbb{N}, \leqslant)$. Or, $(\mathbb{N}, \leqslant)$ est bien fondé, donc la partie se termine.

   — Il existe donc un joueur j qui prend la dernière allumette. Ce joueur perd et l'autre gagne, donc la partie est non nulle.

2. Montrons que

$$\mathscr{A} = \{(\mathrm{A}, i) \mid i \in [\![1, n]\!],\ i \not\equiv 1\ [k+1]\} \cup \{(\mathrm{B}, i) \mid i \in [\![1, n]\!],\ i \equiv 1\ [k+1]\}$$
$$\mathscr{B} = \{(\mathrm{B}, i) \mid i \in [\![1, n]\!],\ i \not\equiv 1\ [k+1]\} \cup \{(\mathrm{A}, i) \mid i \in [\![1, n]\!],\ i \equiv 1\ [k+1]\}$$

On pose, pour tout $p \in \mathbb{N}$,

TD 13.2 Chomp

TD 13.3 Hex

TD 13.4 Calcul de MinMax

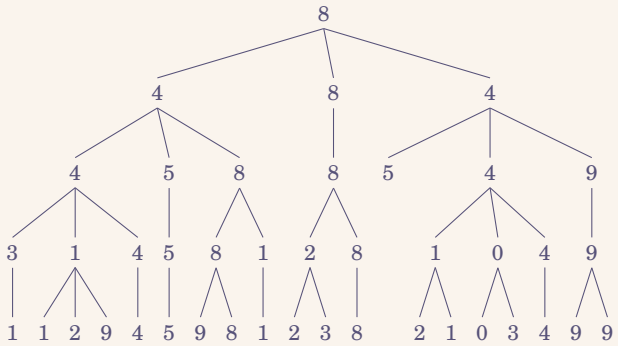


FIGURE TD 13.1 – Arbre remplit avec l'algorithme MinMax

1.

TD 13.5 Calcul de MinMax avec mémoïsation

TRAVAUX DIRIGÉS

14 GRAMMAIRES NON CONTEXTUELLES (1)

Sommaire

TD 14.1 Exemples de dérivations 269
TD 14.2 Arbres de dérivations 269
TD 14.3 Construction de grammaires 270
TD 14.4 Raisonner par induction sur une grammaire 271
TD 14.5 Ambigüité 271
TD 14.6 Langage de Dyck 271
TD 14.7 Listes OCaml 271
TD 14.8 Mots de Łukasiewicz 272

TD 14.1 Exemples de dérivations

1. Les dérivations (b), et (d) sont vraies.
2. Les dérivations (a), (c), et (d).
3. On a $\{ab, ba, aab\} \subseteq \mathscr{L}(\mathscr{G})$. En effet,
   - $\mathrm{S} \Rightarrow \mathrm{D} \Rightarrow a\mathrm{T}b \Rightarrow ab$,
   - $\mathrm{S} \Rightarrow \mathrm{D} \Rightarrow b\mathrm{T}a \Rightarrow ba$,
   - $\mathrm{S} \Rightarrow \mathrm{D} \Rightarrow a\mathrm{T}b \Rightarrow a\mathrm{X}b \Rightarrow aab$.
4. On a $\varepsilon \notin \mathscr{L}(\mathscr{G})$, $aa \notin \mathscr{L}(\mathscr{G})$ et $bb \notin \mathscr{L}(\mathscr{G})$.
5. Le langage $\mathscr{L}(\mathscr{G})$ est l'ensemble des mots non palindromes.

TD 14.2 Arbres de dérivations

1.

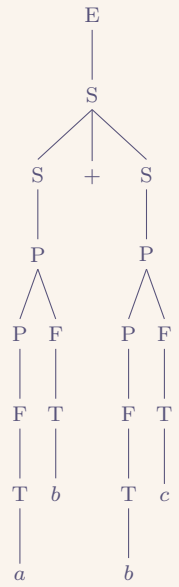


FIGURE TD 14.1 – Arbre de dérivation de $ab+bc$ dans la grammaire $\mathcal{G}$

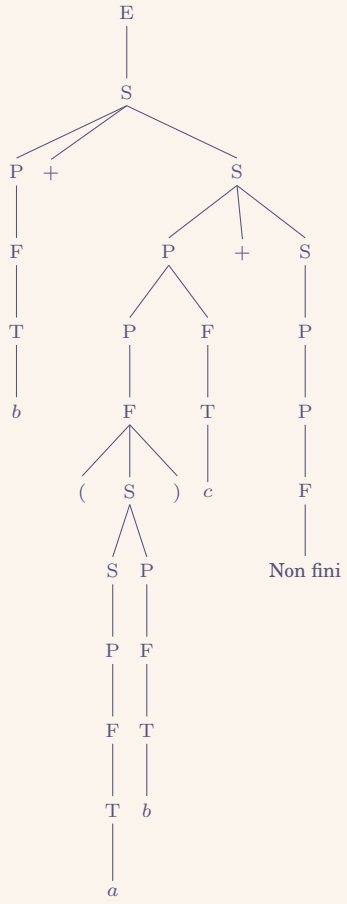


FIGURE TD 14.2

2.

TD 14.3 Construction de grammaires

1. — $\mathcal{G}_0 = (\Sigma, \{\mathrm{L}, \mathrm{A}, \mathrm{S}\}, \{\mathrm{S} \to \mathrm{LLLA}, \mathrm{L} \to a \mid b, \mathrm{A} \to \mathrm{LA} \mid \varepsilon\}, \mathrm{S})$,
   — $\mathcal{G}_1 = (\Sigma, \{\mathrm{L}, \mathrm{A}, \mathrm{S}\}, \{\mathrm{S} \to \mathrm{A}a\mathrm{A}a\mathrm{A}a\mathrm{A}, \mathrm{A} \to \mathrm{LA} \mid \varepsilon, \mathrm{L} \to a \mid b\}, \mathrm{S})$,
   — $\mathcal{G}_2 = (\Sigma, \{\mathrm{S}, \mathrm{V}\}, \{\mathrm{S} \to a\mathrm{X}a \mid b\mathrm{X}b, \mathrm{X} \to a\mathrm{X} \mid b\mathrm{X} \mid \varepsilon\}, \mathrm{S})$,
   — $\mathcal{G}_3(\Sigma, \{\mathrm{S}, \mathrm{X}\}, \{\mathrm{S} \to \mathrm{X}, \mathrm{X} \to \mathrm{S}\}, \mathrm{S})$.
2. — $\mathcal{G}_4 = (\Sigma, \{\mathrm{L}, \mathrm{S}, \mathrm{X}\}, \{\mathrm{S} \to \mathrm{LX}, \mathrm{L} \to a \mid b, \mathrm{X} \to \mathrm{LLX} \mid \varepsilon\}, \mathrm{S})$,
   — $\mathcal{G}_5 = (\Sigma, \{\mathrm{S}, \mathrm{X}\}, \{\mathrm{S} \to \mathrm{X}a\mathrm{X}, \mathrm{X} \to a\mathrm{X} \mid b\mathrm{X} \mid \varepsilon\}, \mathrm{S})$,
   — $\mathcal{G}_6 = (\Sigma, \{\mathrm{S}, \mathrm{X}\}, \{\mathrm{S} \to a\mathrm{S}a \mid b\mathrm{S}b \mid \mathrm{X}, \mathrm{X} \to \varepsilon \mid a \mid b\}, \mathrm{S})$,
   — *c.f.* exercice 1.
3. — $\mathcal{G}_8 = (\Sigma, \{\mathrm{S}\}, \{\mathrm{S} \to \varepsilon \mid a\mathrm{S}b\}, \mathrm{S})$,
   — $\mathcal{G}_9 = (\Sigma, \{\mathrm{S}, \mathrm{X}\}, \{\mathrm{S} \to a\mathrm{S}b \mid \mathrm{SX}, \mathrm{X} \to b\mathrm{X} \mid \varepsilon\}, \mathrm{S})$,
   — $\mathcal{G}_{10} = (\Sigma, \{\mathrm{S}\}, \{\mathrm{S} \to a\mathrm{S}bb \mid \mathrm{S}b \mid \varepsilon\}, \mathrm{S})$
4. — $\mathcal{G}_{11} = (\Sigma, \{\mathrm{S} \to a\mathrm{S}b \mid b\mathrm{S}a \mid a\mathrm{S} \mid \mathrm{SS} \mid \varepsilon\}, \mathrm{S})$,

TD 14.4 Raisonner par induction sur une grammaire

1. Montrons le par induction.
   - **Cas $\mathrm{S} \to a\mathrm{S}$.** Soit $w = aw'$ un mot, où ba n'est pas un sous-mot de w'. Alors, ba n'est pas un sous-mot de $w = aw'$.
   - **Cas $\mathrm{S} \to \mathrm{S}b$.** Soit $w = w'b$ un mot, où ba n'est pas un sous-mot de w'. Alors, ba n'est pas un sous-mot de $w = w'b$.
   - **Cas $S \to \varepsilon \mid a \mid b$.** Le mot ba n'est pas un sous-mot de a, ni de b, ni de ε.
2. Montrons, par double-inclusion, que $\mathscr{L}(\mathscr{G}) = \mathscr{L}(a^\star b^\star)$.

   "$\supseteq$" Soit $w \in \mathscr{L}(a^\star b^\star)$. Il existe m et n deux entiers tels que $w = a^n b^m$. On applique la dérivation

   $$\underbrace{\mathrm{S} \Rightarrow a\mathrm{S} \Rightarrow aa\mathrm{S} \Rightarrow \cdots \Rightarrow a^n\mathrm{S}}_{n \text{ fois}} \Rightarrow \underbrace{a^n\mathrm{S}b \Rightarrow a^n\mathrm{S}bb \Rightarrow \cdots \Rightarrow a^n\mathrm{S}b^m}_{m \text{ fois}} \Rightarrow a^n b^m.$$

   "$\subseteq$" D'après la question 1, on a ce sens de l'inclusion.

TD 14.5 Ambigüité

TD 14.6 Langage de Dyck

1. On suppose ce langage reconnaissable par un automate à n états. On considère le mot $w = ({}^n \cdot)^n$, donc $|w| \geqslant n$. Ainsi, il existe x, y et z trois mots tels que $w = xyz$, $|xy| \leqslant n$, $y \neq \varepsilon$ et $\forall p \in \mathbb{N}$, $xy^p z \in \mathscr{L}(\mathscr{G})$. Soit alors $p \in [\![1, n-1]\!]$ et $q \in [\![1, n-p]\!]$ tels que $x = ({}^p$, $y = ({}^q$ et $z = ({}^{n-q-p} \cdot)^n$. Ainsi, $xy \in \mathscr{L}(\mathscr{G})$, ce qui est absurde. On en déduit que $\mathscr{L}(\mathscr{G})$ n'est pas reconnaissable, il n'est donc pas régulier.
2. On pose $\mathscr{G} = (\Sigma, \{\mathrm{S}\}, \{\mathrm{S} \to (\mathrm{S}) \mid \mathrm{SS} \mid \varepsilon\}, \mathrm{S})$.
3. — On le montre par induction.
   - **Cas $\mathrm{S} \to \varepsilon$.** On a $|\varepsilon|_(= 0 = |\varepsilon|_)$.
   - **Cas $\mathrm{S} \to (\mathrm{S})$.** Soit $u \in \mathscr{L}(\mathscr{G})$ avec $|u|_(= |u|_) = n$. Ainsi, $|(u)|_(= |(u)|_) = n + 1$.
   - **Cas $\mathrm{S} \to \mathrm{SS}$.** Soient u et v deux mots de $\mathscr{L}(\mathscr{G})$ tels que $|u|_(= |u|_) = n$ et $|v|_(= |v|_) = m$. Alors, $|u \cdot v|_(= |v \cdot u|_) = n + m$.

   — Montrons par induction $\mathscr{P}_u$: « pour tout v préfixe de u, $|v|_(\geqslant |v|_)$. »
   - **Cas $\mathrm{S} \to \varepsilon$.** Le seul préfixe de ε est ε, et on a bien $|\varepsilon|_(= 0 \geqslant 0 = |\varepsilon|_)$.
   - **Cas $\mathrm{S} \to (\mathrm{S})$.** Soit u un mot de $\mathscr{L}(\mathscr{G})$ vérifiant $\mathscr{P}_u$. Soit v un préfixe de (u). On procède par induction sur v.
     - **Cas $v = \varepsilon$ ou (.** OK.
     - **Cas $(\tilde{u}$,** où $\tilde{u}$ est un préfixe de u. Par hypothèse d'induction, $|\tilde{u}|_(\geqslant |\tilde{u}|_)$ donc $|(\tilde{u}|_(= |(\tilde{u}|_)$.
     - **Cas (u).** Par hypothèse d'induction, $|u|_(\geqslant |u|_)$ donc $|(u)|_(\geqslant |(u)|_)$.
4. On note $\overline{w}^j = \left|w_{[\![0,j]\!]}\right|_(- \left|w_{[\![0,j]\!]}\right|_)$. Alors les deux conditions se traduisent par $\overline{w}^{|w|} = 0$ et $\forall i \in [\![0, |w|-1]\!]$, $\overline{w}^i \geqslant 0$.

TD 14.7 Listes OCaml

On pose $\mathscr{G} = (\Sigma, \{\mathrm{S}, \mathrm{L}, \mathrm{B}\}, \{\mathrm{S} \to \texttt{[L]} \mid \texttt{[]}, \mathrm{L} \to \mathrm{B}\texttt{;}\mathrm{L} \mid \mathrm{B}, \mathrm{B} \to \texttt{true} \mid \texttt{false}\}, \mathrm{S})$.

TD 14.8 Mots de Łukasiewicz

1. On a $\Sigma \cap \mathscr{L} = \{\square\}$, $\Sigma^2 \cap \mathscr{L} = \varnothing$ et $\Sigma^3 \cap \mathscr{L} = \{\bigcirc\square\square\}$.
2. Montrons que $\Sigma^{2N} \cap \mathscr{L} = \varnothing$, pour tout $N \in \mathbb{N}^\star$. (Dans le cas $N = 0$, le seul mot est ε, et il n'est pas dans $\mathscr{L}$.) Soit $w \in \Sigma^{2N} \cap \mathscr{L}$. On sait que $-1 = \overline{w}^{|w|} = \overline{w}^{|w|-1} + \mathsf{va}(w_{|w|}) \geqslant \mathsf{va}(w_{|w|})$, d'où $w_{|w|} = \square$, et $\overline{w}^{|w|-1} = 0$. On pose $w = u \cdot \square$, le mot u est donc de longueur impaire. Or, la somme $\sum_{k=1}^{|u|} \mathsf{va}(w_i)$ est une somme d'un nombre impaire de termes valant -1 ou 1, elle ne peut pas être nulle. Mais, comme $\overline{w}^{|w|-1} = 0$, donc elle est nulle, une contradiction. On en déduit que $\mathscr{L} \cap \Sigma^{2N} = \varnothing$. Par suite, on conclut que $\mathscr{L}$ ne contient pas de mots pairs.
3. On suppose $\mathscr{L}$ reconnaissable par un automate $\mathscr{A}$ à n états. On considère le mot $w = \bigcirc^n \cdot \square^n \cdot \square \in \mathscr{L}$. Alors, par application du lemme de l'étoile à l'automate $\mathscr{A}$ avec ce mot w, il existe donc x, y et z trois mots de $\Sigma^\star$ tels que $w = xyz$, $|xy| \leqslant n$, $y \neq \varepsilon$ et, pour tout $p \in \mathbb{N}$, $xy^pz \in \mathscr{L}$. D'où, $x = \bigcirc^k$, $y = \bigcirc^j$ et $z = \bigcirc^{n-k-j} \cdot \square^{n+1}$, où k et j sont des entiers. De plus, $j \neq 0$ car, sinon, $y = \varepsilon$. Alors, $xy^pz \in \mathscr{L}$, pour tout $p \in \mathbb{N}$. En particulier, pour $p = 0$, $xz \in \mathscr{L}$. Or, $xz = \bigcirc^{n-j} \cdot \square^{n+1} \notin \mathscr{L}$ car $\overline{xz}^{|xz|} = (n-j) - (n+1) = -j - 1 \neq -1$ car $j \neq 0$. On en déduit donc que que $\mathscr{L}$ n'est pas reconnaissable par un automate à n états. Ceci étant vrai pour tout n, alors $\mathscr{L}$ n'est pas un langage régulier.
4.

```
let est_luka (w: word): bool =
  let l = List.map va w in
  let rec aux (l: int list) (s: int): bool =
    match l with
    | []      -> s = -1
    | x :: q -> s >= 0 && aux q (s + x)
  in aux l 0
```

CODE TD 14.1 – Fonction `est_luka` testant si w est un mot de Łukasiewicz

5. Soit $w = u \cdot v \in \mathscr{L}$, où $u \neq \varepsilon$ est un préfixe strict de w. Alors, $0 \leqslant \overline{w}^{|u|} = \overline{u}^{|u|} \neq -1$ donc $u \notin \mathscr{L}$.
6. Soient u et v deux mots de $\mathscr{L}$. On pose $w = \bigcirc \cdot u \cdot v$. Montrons que $w \in \mathscr{L}$. On pose $n = |u|$, et $m = |v|$.
   - On a $\overline{w}^0 = 0 \geqslant 0$,
   - et $\overline{w}^1 = \mathsf{va}(\bigcirc) = 1 \geqslant 0$,
   - et, pour tout $i \in [\![1, n[\![$, $\overline{w}^{i+1} = \overline{u}^i + \mathsf{va}(\bigcirc) \geqslant \mathsf{va}(\bigcirc) \geqslant 0$,
   - et, pour tout $i \in [\![1, m[\![$, $\overline{w}^{i+n+1} = \overline{u}^n + \overline{v}^i + \mathsf{va}(\bigcirc) = \overline{v}^i \geqslant 0$,
   - et finalement, $\overline{w}^{n+m+1} = \overline{u}^n + \overline{v}^m + \mathsf{va}(\bigcirc) = -1 - 1 + 1 = -1$.

   On en conclut que $w \in \mathscr{L}$.
7. Soit $w \in \mathscr{L}$, un mot de taille n. Comme montré précédemment, $\overline{w}^{n-1} = 0$. Ainsi, l'ensemble $\{k \in [\![1, n]\!] \mid \overline{w}^i = 0\}$ est une partie de $\mathbb{N}$ non vide, elle admet donc un minimum que l'on notera k. De plus, on a montré que $w_0 = \bigcirc$. On en déduit que w peut être décomposé en
$$w = \bigcirc \cdot \underbrace{w_2 w_3 \dots w_k}_{u} \cdot \underbrace{w_{k+1} w_{k+2} \dots w_n}_{v} \cdot$$
   Montrons que u et v sont des mots de $\mathscr{L}$. D'une part, pour $i \in [\![1, k-2]\!]$, $0 < \overline{w}^{i+1} = \overline{u}^i + \mathsf{va}(\bigcirc)$, d'où $\overline{u}^i \geqslant 0$. De plus, $0 = \overline{w}^k = \overline{u}^{k-1} + \mathsf{va}(\bigcirc)$, d'où $\overline{u}^{k-1} = \overline{u}^{|u|} = -1$. Également, pour $i \in [\![1, n-k-1]\!]$, $0 \leqslant \overline{w}^{k+i} = \overline{w}^k + \overline{v}^i = \overline{v}^i$. Finalement, $-1 = \overline{w}^n = \overline{w}^k + \overline{v}^{n-k} = \overline{v}^{|v|}$. On en conclut que u et v sont bien des mots de $\mathscr{L}$.
   Montrons, à présent, l'unicité de la décomposition. On suppose qu'il existe u' et v' deux mots de $\mathscr{L}$ tels que $w = \bigcirc \cdot u' \cdot v'$. Nous savons, en particulier, que $\overline{w}^{1+|u'|} = 0$, afin de maintenir la condition $u \in \mathscr{L}$. Alors, $1 + |u'| \leqslant p$, et donc $|u'| \geqslant |u|$. Le mot u' est donc un préfixe de u. Au vu de la question 5., afin que u' soit un mot de $\mathscr{L}$, il est nécessaire que u' ne soit pas un préfixe strict de u. On en déduit que $u' = u$.
8. On utilise habilement la question précédente, et on pose
$$\mathscr{G} = (\{\mathrm{L}\}, \Sigma, \{\mathrm{L} \to \bigcirc\mathrm{LL} \mid \bigcirc\}, \mathrm{L}).$$
   La question précédente montre la non-ambigüité de la grammaire, et que $\mathscr{L}(\mathscr{G}) = \mathscr{L}$.
9.

TRAVAUX DIRIGÉS

15 GRAMMAIRES NON CONTEXTUELLES (2)

Sommaire

TD 15.1 Forme normale conjonctive **273**
TD 15.2 Réduction de grammaire et systèmes de conséquences **274**
TD 15.2.1 Digression OCaml 274
TD 15.3 Grammaires propres **274**
TD 15.4 Un lemme d'itération **274**
TD 15.5 Les langages réguliers sont non contextuels **274**
TD 15.5.1 Avec des automates 274
TD 15.5.2 Avec des expressions régulières 275

TD 15.1 Forme normale conjonctive

1. On pose la grammaire $\mathcal{G}$ de symbole initial U et ayant pour règles de production

$$\begin{aligned} \mathrm{U} &\to \mathrm{S} \mid \varepsilon, \\ \mathrm{S} &\to \mathrm{S\&X} \mid \mathrm{X}, \\ \mathrm{X} &\to \text{-}\mathrm{V} \mid \mathrm{V} \mid \mathrm{V}\mid\mathrm{X} \mid \text{-}\mathrm{V}\mid\mathrm{X}, \\ \mathrm{V} &\to \mathtt{p} \mid \mathtt{q} \mid \cdots . \end{aligned}$$

2.

```
'p' ⇝ la variable p
('p', true) ⇝ le littéral p
('p', false) ⇝ le littéral ¬p
[ ('p', true); ('q', true) ] ⇝ la clause p ∨ q
[[ ('p', true); ('q', true) ]] ⇝ la formule p ∨ q
[[('p', true)]; [('q', true)]]] ⇝ la formule p ∧ q
[[('p', true)]; [('q', true); ('p', false)]]] ⇝ la formule p ∧ (q ∨ ¬p)
```

Code TD 15.1 – Expressions OCaml

3.

```
let rec parse_f (s: string) (start: int) (len: int): fnc =
  if String.length s = 0 then []
  else parse_d s start len
and parse_d (s: string) (start: int) (len: int): fnc =
  let i = ref start in
  while s.[!i] != '&' && !i < len do incr i; done;
  if !i = len then [parse_c s start len]
  else (parse_c s start (!i - 1)) :: (parse_d s (!i+1) len)
and parse_c (s: string) (start: int) (len: int): cls =
  let i = ref start in
  while s.[!i] != '|' && !i < len do incr i; done;
  if !i = len then [parse_l s start len]
  else (parse_l s start (!i - 1)) :: (parse_c s (!i+1) len)
and parse_l (s: string) (start: int) (len: int): lit =
  if s.[start] = '-' then (parse_v s (start + 1) len, false
      ↪ )
  else (parse_v s start len, true)
and parse_v (s: string) (start: int) (len: int): char =
  let x = s.[start] in
  assert(Char.code x >= 97 && Char.code x <= 122);
  assert(len - start = 1);
  x
```

CODE TD 15.2 – Parsing des fonctions OCAML

TD 15.2 Réduction de grammaire et systèmes de conséquences

TD 15.2.1 Digression OCAML

TD 15.3 Grammaires propres

TD 15.4 Un lemme d'itération

TD 15.5 Les langages réguliers sont non contextuels

TD 15.5.1 Avec des automates

1. On pose P l'ensemble des règles de productions définies comme

$$\{X_q \to \ell X_{q'} \mid (q, \ell, q') \in \delta\} \cup \{X_q \to \varepsilon \mid q \in F\}.$$

Montrons par récurrence $\mathscr{P}(n)$:

« $\forall q \in \mathbb{Q}$, $\mathscr{L}_n(\mathscr{A}_q) = \{w \in \mathscr{L} \mid |w| = n\} = \{w \in \Sigma^n \mid X_q \overset{\star}{\Rightarrow} w\} = G_n(q)$. »

— Pour $n = 0$, soit $q \in Q$ et soit $w \in \Sigma^\star$. Si $w \in \mathscr{L}_n(\mathscr{A}_q)$, alors $w = \varepsilon$ et $q \in F$, d'où $(X_q \to \varepsilon) \in P$ donc $X_q \overset{\star}{\Rightarrow} w$ donc $w \in G_n(q)$. Réciproquement, si $w \in G_n(q)$, alors $X_q \overset{\star}{\Rightarrow} \varepsilon$ car il n'y a pas d'ε-transitions, donc $q \in F$ et donc $w = \varepsilon \in \mathscr{L}_n(\mathscr{A}_q)$.

— Soit $n \in \mathbb{N}$. On suppose $\mathscr{P}(n)$ vraie. Soit $q \in Q$ et soit $w \in \Sigma^{n+1}$. Si $w \in \mathscr{L}_{n+1}(\mathscr{A}_q)$, alors il existe une exécution acceptante

$$q \xrightarrow{w_1} q_1 \to \cdots \to q_n \xrightarrow{w_{n+1}} q_{n+1} \in F.$$

D'où, $w_2 \dots w_{n+1} \in \mathscr{L}_n(\mathscr{A}_{q_1}) = G_n(q_1)$ par hypothèse donc $X_{q_1} \overset{\star}{\Rightarrow} w_2 \dots w_{n+1}$. Or, $(q, w_1, q_1) \in \delta$ donc $(X_q \to w_n X_{q_1}) \in P$ et donc $X_q \Rightarrow w_1 X_{q_1} \overset{\star}{\Rightarrow} w$. D'où, $X_q \overset{\star}{\Rightarrow} w$ et donc $w \in G_{n+1}(q)$.

Réciproquement, si $w \in G_{n+1}(q)$ alors $X_q \overset{\star}{\Rightarrow} w$. Soit $w' \in Q$ tel que $X_q \Rightarrow w_1 X_{q'}$. Alors, $(q, w_1, q') \in \delta$. De plus, $|w_2 \dots w_n| = n$ et $X_{q'} \overset{\star}{\Rightarrow} w_2 \dots w_{n+1}$ donc $w_2 \dots w_{n+1} \in G_n(q') = \mathscr{L}_n(\mathscr{A}_{q'})$. Il existe donc une exécution acceptante

$$q' \xrightarrow{w_2} q_2 \to \cdots \to q_n \xrightarrow{w_{n+1}} q_{n+1} \in F.$$

Or, $(q, w_1, q') \in \delta$ d'où

$$q \xrightarrow{w_1} q' \xrightarrow{w_2} q_2 \to \cdots \to q_n \xrightarrow{w_{n+1}} q_{n+1} \in F$$

est une exécution acceptante de $\mathscr{A}_q$. On en déduit que $w \in \mathscr{L}_{n+1}(\mathscr{A}_q)$.

2. On en conclut que tout langage régulier est représentable par une grammaire non-contextuelle.

TD 15.5.2 Avec des expressions régulières

3. On pose P l'ensemble de règles de productions défini comme

$$\begin{aligned} P = \quad & \{X_r \to X_r X_{r'} \mid \varepsilon \text{ tel que } r = (r')^\star\} \\ & \cup \{X_r \to X_{r_1} \mid X_{r_2} \text{ tel que } r = r_1 \mid r_2\} \\ & \cup \{X_r \to X_{r_1} X_{r_2} \text{ tel que } r = r_1 \cdot r_2\} \\ & \cup \{X_r \to r \text{ tel que } r \in \Sigma\} \\ & \cup \{X_r \to \varepsilon \text{ tel que } r = \varepsilon\} \end{aligned}$$

TRAVAUX DIRIGÉS

16 CONCURRENCE

Sommaire

TD 16.1 Entrelacements 277
TD 16.2 Généralisation de l'algorithme de Peterson à N fils 277
TD 16.3 Parallélisation pour le produit de deux matrices 278
TD 16.4 Calcul du maximum par "diviser pour régner" 278
TD 16.5 Un très mauvais algorithme de tri 278

TD 16.1 Entrelacements

TD 16.2 Généralisation de l'algorithme de Peterson à N fils

1. *c.f.* cours
2. Les algorithmes 1 et 2 ne sont pas corrects. On donne deux exécutions posant problème.

Algorithme TD 16.1 Proposition 1

```
1 : Turn ← 0
2 : Want est un tableau de N booléens initialisé à F.
3 : Procédure Lock(i)
4 :     Want[i] ← V
5 :     Turn ← i + 1 mod N
6 :     tant que Turn ≠ i et Want[Turn] faire
7 :         rien
8 :
9 : Procédure Unlock(i)
10 :    Want ← F
```

Algorithme TD 16.2 Proposition 2

```
1 : Turn ← 0
2 : Want est un tableau de N booléens initialisé à F.
3 : Procédure Lock(i)
4 :     Want[i] ← V
5 :     Turn ← i + 1 mod N
6 :     tant que Turn ≠ i faire
7 :         si Want[Turn] alors
8 :
9 :             Turn ← Turn + 1 mod N
10 : Procédure Unlock(i)
11 :    Want ← F
```

TD 16.3 Parallélisation pour le produit de deux matrices

TD 16.4 Calcul du maximum par “diviser pour régner”

TD 16.5 Un très mauvais algorithme de tri

TRAVAUX DIRIGÉS

17 CONCURRENCE

Sommaire

TD 17.1 Addition binaire en parallèle **279**
TD 17.2 Producteurs–consommateurs en mémoire non bornée **279**
TD 17.2.1 Version allégée de la solution vue en cours 279
TD 17.3 Rendez-vous à l'aide de sémaphores **280**
TD 17.3.1 Deux fils d'exécutions se rencontrent une fois 280
TD 17.3.2 Plusieurs fils se rencontrent une fois 280
TD 17.3.3 Plusieurs fils d'exécution se rencontrent plusieurs fois 281

TD 17.1 Addition binaire en parallèle

TD 17.2 Producteurs–consommateurs en mémoire non bornée

TD 17.2.1 Version allégée de la solution vue en cours

1. C'est le sémaphore Plein qui a pour rôle d'éviter les problèmes de dépassement. Avec l'hypothèse d'une mémoire non bornée, on retire l'appel `acquire` Plein de la procédure PRODUCTION, l'appel `release` Plein de la procédure CONSOMMATION, et la création du sémaphore Plein de l'algorithme 1.

2. L'utilisation d'une telle variable sur plusieurs fils d'exécution nécessite un *mutex* afin que l'écriture en mémoire ne soit faite qu'un fil à la fois.

3.

TD 17.3 Rendez-vous à l'aide de sémaphores

TD 17.3.1 Deux fils d'exécutions se rencontrent une fois

1. On considère les fils P_1 et P_2. Dans le programme principal, on crée deux sémaphores $\mathcal{S}_1$ et $\mathcal{S}_2$, initialisés à 0.

Algorithme TD 17.1 Fil d'exécution P_1

```
1 : A_1
2 : release S_2
3 : acquire S_1
4 : B_1
```

Algorithme TD 17.2 Fil d'exécution P_2

```
1 : A_2
2 : release S_1
3 : acquire S_2
4 : B_2
```

Pour généraliser, on considère 3 sémaphores $\mathcal{S}_1$, $\mathcal{S}_2$ et $\mathcal{S}_3$ initialement à 0. On définit donc les fils P_1, P_2 et P_3 définis ci-dessous.

Algorithme TD 17.3 Fil d'exécution P_1

```
1 : A_1
2 : release S_2
3 : release S_3
4 : acquire S_1
5 : acquire S_1
6 : B_1
```

Algorithme TD 17.4 Fil d'exécution P_2

```
1 : A_2
2 : release S_1
3 : release S_3
4 : acquire S_2
5 : acquire S_2
6 : B_2
```

Algorithme TD 17.5 Fil d'exécution P_3

```
1 : A_3
2 : release S_1
3 : release S_2
4 : acquire S_3
5 : acquire S_3
6 : B_3
```

2. On perds l'indépendance des fils P_1 et P_2. Pour généraliser la solution, on considère les algorithmes ci-dessous.

Algorithme TD 17.6 Fil P_1

```
1 : A_1
```

Algorithme TD 17.8 Fil P_2

```
1 : A_2
```

Algorithme TD 17.10 Fil P_3

```
1 : A_3
```

Algorithme TD 17.7 Fil P'_1

```
1 : B_1
```

Algorithme TD 17.9 Fil P'_2

```
1 : B_2
```

Algorithme TD 17.11 Fil P'_3

```
1 : B_3
```

Algorithme TD 17.12 Programme principal

```
1 : lancer P_1, P_2 et P_3 en parallèle
2 : attendre la fin de P_1, P_2 et P_3
3 : lancer P'_1, P'_2 et P'_3 en parallèle
```

3. Non, les instructions B_2 et A_1 pourraient être exécutées en parallèle, mais ce n'est pas possible avec la subdivision des fils.

TD 17.3.2 Plusieurs fils se rencontrent une fois

4. On considère l'algorithme ci-dessous.

Algorithme TD 17.13 Implémentation de la structure barrière $\mathcal{b}$

```
1 : Procédure CréeBarrière
2 :     Soit un verrou 𝒱.
3 :     Soit i = 0.
4 :     Soit un sémaphore 𝒮 initialisé à 0.
5 :     retourner 𝒷 = (𝒱, 𝒮, i).
6 : Procédure AppelBarrière(b)
7 :     (𝒱, 𝒮, i) ← 𝒷
8 :     lock(𝒱)
9 :     i ← i + 1
10 :    si i = n alors
11 :        pour k ∈ ⟦1, n⟧ faire
12 :            release 𝒮
13 :    unlock(𝒱)
14 :    acquire 𝒮
```

TD 17.3.3 Plusieurs fils d'exécution se rencontrent plusieurs fois

5.

6. On considère l'algorithme suivant.

Algorithme TD 17.14 Implémentation de la structure barrière robuste $\mathcal{b}$

```
1 : Procédure CréeBarrière(n)
2 :     Soit un verrou 𝒱.
3 :     Soit i = 0, et soit nb = i.
4 :     Soit un sémaphore 𝒮 initialisé à 0.
5 :     retourner 𝒷 = (𝒱, 𝒮, i, nb).
6 : Procédure AppelBarrière(b)
7 :     (𝒱, 𝒮, i) ← 𝒷
8 :     lock(𝒱)
9 :     i ← i + 1
10 :    si i = n alors
11 :        pour k ∈ ⟦1, n⟧ faire
12 :            release 𝒮
13 :    unlock(𝒱)
14 :    acquire 𝒮
15 : Procédure JeNeViendraisPlus(𝒷)
16 :    (𝒱, 𝒮, i, nb) ← 𝒷
17 :    lock(𝒱)
18 :    nb ← nb − 1
19 :    si i = nb alors
20 :        pour k ∈ ⟦1, nb⟧ faire
21 :            release 𝒮
22 :        i ← 0
23 :    unlock(𝒱)
```

TRAVAUX DIRIGÉS

1 COMPLEXITÉ AMORTIE

Sommaire

TD BONUS 1.1 Complexité amortie **283**
TD BONUS 1.2 Incrementation compteur binaire **284**
TD BONUS 1.3 Tableaux dynamiques **284**
TD BONUS 1.4 Tableaux dynamiques, 2 . . . . . **284**

TD BONUS 1.1 Complexité amortie

1. (a) Soit $n \in \mathbb{N}$.

   — Si n est pair, $\displaystyle\sum_{i=0}^{n-1} c_i = \sum_{i=1}^{n/2} 1 + \sum_{i=1}^{n/2} 3 = 2n.$

   — Si n est impair,

$$\begin{aligned}\sum_{i=0}^{n-1} c_i &= \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor + 1} 1 + \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} 3 \\ &= \left\lfloor \frac{n}{2} \right\rfloor + 1 + 3 \left\lfloor \frac{n}{2} \right\rfloor \text{ car } n = 2 \left\lfloor \frac{n}{2} \right\rfloor + 1 \\ &= 4 \left\lfloor \frac{n}{2} \right\rfloor + 1 \\ &= n + 2 \left\lfloor \frac{n}{2} \right\rfloor \\ &= n + n - 1 \\ &= 2n - 1\end{aligned}$$

   (b) On pose h la fonction de potentiel définie comme

$$\begin{aligned} h : \mathbb{N} &\longrightarrow \mathbb{R}^+ \\ n &\longmapsto \begin{cases} 0 & \text{si } n \text{ est pair,} \\ 1 & \text{si } n \text{ est pair.} \end{cases}\end{aligned}$$

— Si n est pair, $\underline{C_o}(n) = C_o(n) + h(n+1) - h(n) = 1 + 1 = 2.$
— Si n est impair, $\underline{C_o}(n) = C_o(n) + h(n+1) - h(n) = 3 + 0 - 1 = 2.$

TD BONUS 1.2 Incrementation compteur binaire

TD BONUS 1.3 Tableaux dynamiques

3. On trouve une complexité amortie en n^2. À rédiger.
4. Au lieu de diviser quand $r < n/2$, mais quand $r < n/4$.

TD BONUS 1.4 Tableaux dynamiques, 2

TRAVAUX DIRIGÉS

2

DIVISER POUR RÉGNER

Sommaire

TD BONUS 2.1 Suites récurrentes de complexité **285**
TD BONUS 2.2 Multiplication d'entiers par algorithme de KARATSUBA **286**

TD BONUS 2.1 Suites récurrentes de complexité

(*a*) On considère la suite $u_0 = 1$ et $u_n = u_{n-1} + a$ pour $a > 0$. Montrons que $u_n = \Theta(n)$.

(*b*) On considère la suite $u_0 = 1$ et $u_n = u_{n-1} + an$ pour $a > 0$.

(*c*) On considère la suite $u_0 = 1$ et $u_n = au_{n-1} + b$ pour $a > 2$ et $b \in \mathbb{R}_+^\star$.

$$\begin{aligned}
u_n &= au_{n-1} + b \\
&= a(au_{n-2} + b) + b \\
&= a^2 u_{n-2} + ab + b \\
&= a^3 u_{n-3} + a^2 b + ab + b \\
&\vdots \\
&= a_n + \sum_{k=0}^{n-1} a^k b \\
&= a^n + b\frac{1-a^n}{1-a} \\
&= a^n \cdot \left(1 - \frac{b}{1-a}\right) + \frac{b}{1-a} \\
&= \Theta(a^n)
\end{aligned}$$

(*d*) On considère la suite $u_0 = 1$ et $u_n = u_{n/2} + b$ avec $b > 0$. Soit $(v_p)_{p\in\mathbb{N}}$ la suite définie par $v_p = u_{2^p}$. Donc

$$v_p = u_{2^p} = u_{2^p/2} + b = u_{2^{p-1}/2} + 2b = \cdots = v_0 + bp = 1 + (p+1)b.$$

La suite $(u_n)_{n\in\mathbb{N}}$ est croissante donc, pour $n \in \mathbb{N}$, Alors,

$$v_{\lfloor \log_2 n \rfloor} \leqslant u_n \leqslant v_{\lceil \log_2 n \rceil} \text{ d'où } b(\lfloor \log_2 n \rfloor + 1) + 1 \leqslant u_n \leqslant b$$
$$\text{d'où } b \log_2 n + 1 \leqslant u_n \leqslant b(\log_2 n + 2) + 1$$

On en déduit que $u_n = \Theta(\log_2 n)$.

(*e*) On considère la suite $u_0 = 1$ et $u_n = u_{n/2} + bn$ avec $b > 0$. On pose $(v_p)_{p\in\mathbb{N}}$ la suite définie par $v_p = u_{2^p}$.

$$\begin{aligned} v_p &= v_{p-1} + 2^p b \\ &= v_{p-2} + 2^{p-1} b + 2^p b \\ &= b \sum_{k=1}^{p} 2^k + v_0 \\ &= b \sum_{k=0}^{p} 2^k + u_1 \\ &= b \sum_{k=0}^{p} 2^k + 1 + b \\ &= b \sum_{k=0}^{p} 2^k + 1 \\ &= b(2^{p+1} - 1) + 1. \end{aligned}$$

D'où, par croissance de la suite $(u_n)_{n\in\mathbb{N}}$,

$$b(2^{\log_2 n} - 1) + 1 \leqslant u_n \leqslant b(2^{\log_2 n + 2} - 1) + 1 \quad \text{d'où} \quad bn - b + 1 \leqslant u_n \leqslant 4bn - b + 1.$$

Ainsi, $u_n = \Theta(n)$.

TD BONUS 2.2 Multiplication d'entiers par algorithme de KARATSUBA

1. On peut éventuellement rajouter des zéros à gauche jusqu'à avoir la même taille pour les deux nombres.
2. L'algorithme a une complexité en $\mathcal{O}(n^2)$.
3. On a $u = u_\mathrm{a} + 2^p \cdot u_\mathrm{b}$, où $p = |u|/2$. De même, $v = v_\mathrm{a} + 2^p \cdot v_\mathrm{b}$. On remarque que

   $$\mathrm{prod}(u, b) = \mathrm{prod}(u_\mathrm{a}, v_\mathrm{a}) + 2^p\big(\mathrm{prod}(u_\mathrm{a}, v_\mathrm{b}) + \mathrm{prod}(u_\mathrm{b}, v_\mathrm{a})\big) + 2^{2p} \cdot \mathrm{prod}(u_\mathrm{b}, v_\mathrm{b}).$$

   Avec $|u| = 2^p$, on note la complexité C_p. On a $C_p = 4C_{p-1} + K = 4(4C_{p-2} + K) + K = 4^p C_0 + K \sum_{i=0}^{p-1} 4^i$. On en déduit que $C_o = \Theta(4^p)$. Or, $p = \log_2 |u|$, d'où $4^{\log_2 |u|} = n^2$. L'algorithme est en $\mathcal{O}(n^2)$.
4. On remarque que $u_\mathrm{a}v_\mathrm{a} + u_\mathrm{b}v_\mathrm{b} - (u_\mathrm{a} - u_\mathrm{b})(v_\mathrm{a} - v_\mathrm{b}) = u_\mathrm{a}v_\mathrm{a} + u_\mathrm{b}v_\mathrm{b}$. Ainsi, $C_p = 3C_{p-1} + K$, d'où $C_p = \Theta(3^p)$. On en déduit que l'algorithme est en $\mathcal{O}(n^{\log_2 3})$.

TRAVAUX DIRIGÉS

3

INVARIANTS PLUS COMPLEXES

Partie III

Travaux Pratiques

TRAVAUX PRATIQUES

1 LOGIQUE PROPOSITIONNELLE

Sommaire

TP 1.1 Syntaxe et sémantique **291**
TP 1.1.1 Syntaxe 291
TP 1.1.2 Sémantique 292
TP 1.2 Construction d'une formule à partir d'une fonction **294**
TP 1.3 Application de règles de réécriture **294**

TP 1.1 Syntaxe et sémantique

TP 1.1.1 Syntaxe

1.

```
type formule =
  | Var   of string
  | And   of formule * formule
  | Or    of formule * formule
  | Imply of formule * formule
  | Equiv of formule * formule
  | Top
  | Bot
  | Not   of formule
```

Code TP 1.1 – Définition du type `formule` représentant les éléments de $\mathcal{F}$

2.

```
let rec print_formule (x: formule) : unit =
  match x with
  | Var(p) -> print_string p;
  | And(p,q) ->
      print_string "(";
      print_formule p;
      print_string "∧";
      print_formule q;
      print_string ")";
  | Or(p,q) ->
```

```
      print_string "(";
      print_formule p;
      print_string "∨";
      print_formule q;
      print_string ")";
  | Imply(p,q) ->
      print_string "(";
      print_formule p;
      print_string "→";
      print_formule q;
      print_string ")";
  | Equiv(p,q) ->
      print_string "(";
      print_formule p;
      print_string "↔";
      print_formule q;
      print_string ")";
  | Top -> print_string "⊤";
  | Bot -> print_string "⊥";
  | Not(p) ->
      print_string "(";
      print_string "¬";
      print_formule p;
      print_string ")"
```

CODE TP 1.2 – Affichage du type `formule`

3.

```
let rec vars (x: formule) : string set =
  match x with
  | Var(p) -> Set.add p Set.empty
  | And(p,q) | Or(p,q) | Imply(p,q) | Equiv(p,q) -> Set.
      ↪ union (vars p) (vars q)
  | Top | Bot -> Set.empty
  | Not(p) -> vars p
```

CODE TP 1.3 – Ensemble de variables d'une formule de type `formule`

TP 1.1.2 Sémantique

4.

```
type env_prop = (string * bool) list
```

CODE TP 1.4 – Définition du type `env_prop` représentant un environnement propositionnel

5.

```
let print_env_prop (e: env_prop): unit =
  let strings = List.map
    (fun (p, b) -> p ^ "↦" ^ (if b then "V" else "F"))
    e in
  let concatenated = List.fold_left (fun s p -> s ^ "," ^ p
      ↪ ) "" strings in
  print_string("{" ^ concatenated ^ "}")
```

CODE TP 1.5 – Affichage du type `env_prop`

6.

```
exception Missing_Env

let rec interprete (f: formule) (e: env_prop) : bool =
  match f with
  | Var(p) ->
      let rec aux e =
        match e with
        | [] -> raise Missing_Env
        | (v,t)::_ when t -> true
```

```
        | _::q -> aux q
      in aux e
  | And(p,q) -> (interprete p e) && (interprete q e)
  | Or(p,q) -> (interprete p e) || (interprete q e)
  | Imply(p,q) -> interprete (Or(q, Not(p))) e
  | Equiv(p,q) -> (interprete p e) = (interprete q e)
  | Top -> true
  | Bot -> false
  | Not(p) -> not (interprete p e)
```

Code TP 1.6 – Interprétation d'une formule

7.

```
let rec all_envs (vars: string list): env_prop list =
  match vars with
  | x :: q -> let envs = all_envs q in
    let aux1 e = (x, true) :: e in
    let aux2 e = (x, false) :: e in
    (List.map aux1 envs) @ (List.map aux2 envs)
  | [] -> []
```

Code TP 1.7 – Génération des environnements propositionnels

8.

```
let sat (f: formule): env_prop =
  let envs = all_envs (vars f) in
  let envs_valides = List.filter (interprete f) envs in
  match envs_valides with
  | [] -> raise Unsat
  | x::_ -> x
```

Code TP 1.8 – Résolution du problème Sat

9.

```
let est_valide (f: formule): bool =
  let envs = all_envs (vars f) in
  List.for_all (interprete f) envs
```

Code TP 1.9 – Résolution du problème Valide

10.

```
let est_cons_semantique (f: formule) (g: formule): bool =
  let envs = all_envs (vars f) in
  let envs_f_valides = List.filter (interprete f) envs in
  List.for_all (interprete g) envs_f_valides
```

Code TP 1.10 – Vérification de "conséquence sémantique"

11.

```
let equiv (f: formule) (g: formule): bool =
  let envs = all_envs (vars f) in
  let envs_f_valides = List.filter (interprete f) envs in
  let envs_g_valides = List.filter (interprete g) envs in
  envs_f_valides = envs_g_valides
```

Code TP 1.11 – Vérification de "équivalence"

12.

```

  let envs_g_valides = List.filter (interprete g) envs in
  envs_f_valides = envs_g_valides
```

Code TP 1.12 – Calcul de modèles d'une formule

```
let models (f: formule): env_prop set =
  let envs = all_envs (vars f) in
  Set.of_list (List.filter (interprete f) envs)
```

Code TP 1.13 – Calcul de modèles d'une formule

TP 1.2 Construction d'une formule à partir d'une fonction

1.

```
let formule_of_fct_bool (vars: string list) (f: fct_bool):
    ↪ formule =
  let envs = all_envs vars in
  let envs_valides = List.filter f envs in
  let rec gen_conj (rho: env_prop): formule =
    match rho with
    | [] -> Top
    | (p,b)::q -> if b then And(Var(p), gen_conj q)
                  else And(Not(Var(p)), gen_conj q)
  in
  let conjs = List.map gen_conj envs_valides in
  List.fold_left (fun x a -> Or(x, a)) Bot conjs
```

CODE TP 1.14 – Déterminaison d'une formule dont l'interprétation est `f` sous forme FND

2.

```
let formule_of_fct_bool2 (vars: string list) (f: fct_bool):
    ↪  formule =
  let f' rho = not (f rho) in
  let h = formule_of_fct_bool vars f' in
  let rec convert_not (h: formule) =
    match h with
    | Or(a, b) -> And(convert_not a, convert_not b)
    | And(a, b) -> Or(convert_not a, convert_not b)
    | Not(a) -> convert_not a
    | Var(p) -> Not(Var(p))
    | Imply(a,b) -> convert_not (Or(b, Not(a)))
    | Equiv(a,b) -> convert_not (And(Imply(a, b), Imply(b,
        ↪ a)))
    | Top -> Bot
    | Bot -> Top
  in convert_not h
```

CODE TP 1.15 – Déterminaison d'une formule dont l'interprétation est `f` sous forme FNC

TP 1.3 Application de règles de réécriture

TRAVAUX PRATIQUES

3

LANGAGES ET EXPRESSIONS RÉGULIÈRES (2)

TRAVAUX PRATIQUES

7

ALGORITHME DE KOSARAJU EN OCAML

Sommaire

TP 7.1 Gestion du graphe et du graphe transposé **297**
TP 7.2 Gestion des points de régénération dun parcours **297**

TP 7.1 Gestion du graphe et du graphe transposé

1.

```
type graphe = int list array
```

Code TP 7.1 – Type graphe

2.

```
let transpose (g: graphe): graphe =
let n = Array.length g in
let g' = Array.make n [] in
for i = 0 to n - 1 do
  List.iter (fun j -> g'.(j) <- i :: g'.(j)) g.(i)
done;
g'
```

Code TP 7.2 – Transposée d'un graphe

TP 7.2 Gestion des points de régénération dun parcours

3. Avec comme ordre de priorité (a, b, c, d, e, f, g), on a le parcours

$$a \to b \to c \to d \to e \to f \to g.$$

Avec l'ordre de priorité (f, e, g, c, d, b, a), on a le parcours

$$f \to e \rightsquigarrow g \to c \to d \to b \rightsquigarrow a.$$

4.

```
exception Plus_d_entiers

let cree_entiers (n: int): (int ref) * (unit -> unit) =
let elem = ref 0 in
let next () =
  if !elem < n then elem := !elem + 1
  else raise Plus_d_entiers
in (elem, next)
```

CODE TP 7.3 – Générateur du tableau $[\![0, n]\!]$

5.

```
let cree_enumerateur_tab (tab: int array): (int ref) * (
    ↪ unit -> unit) =
let i = ref 0 in
let elem = ref tab.(0) in
let next () =
  i := !i + 1;
  elem := tab.(!i)
in
(elem, next)
```

CODE TP 7.4 – Générateur d'un tableau `tab`

Partie IV

Annexes

ANNEXE

A

COMPLEXITÉ AMORTIE

Avec une fonction $\mathrm{PA}(n)$ ayant une complexité en $\mathcal{O}(f(n))$, on considère le problème ci-dessous.

```
for i = 0 to n - 1 do
  PA(i)
done
```

Cet algorithme a une complexité en $\mathcal{O}(nf(n))$. Mais, parfois, cette complexité est trop approximative : parfois, des sommes mathématiques se compensent :

$$\sum_{i=0}^{n-1} 2^i = 2^n \neq \cancel{\mathcal{O}}(n2^n).$$

On considère le problème ci-dessous.

```
for (int i = 0; i < n; i = i + 1) (
  // calcul qui ne coute pas cher
}
```

Le calcul `i = i + 1` est parfois plus coûteux que le calcul dans la boucle. Par exemple, un algorithme permettant de faire ce calcul est celui ci-dessous.

AlgorithmeAnnexe A.1 Calcul de $n+1$ avec un tableau de *bit*s

Entrée un entier n, représenté sous la forme d'un tableau de *bit* T (où les *bit*s de poids forts sont à droite

1 : $I \leftarrow \mathrm{len}(T) - 1$
2 : **tant que** $T[I] = 1$ **faire**
3 : $\quad T[I] \leftarrow 0$
4 : $\quad I \leftarrow I - 1$
5 : $T[I] \leftarrow 1$

Avec un tel algorithme, on a une complexité, dans le pire des cas, en $\Theta(\log_2 n)$. Ainsi, en modifiant le code, on peut avoir une complexité importante.

```
1  // n est une valeur donnee par l'utilisateur
2  for (int i = 0; i < 2^n; i = i + 1) (
3    // calcul qui ne coute pas cher
4  }
```

La complexité de cet algorithme, que l'on nommera $\mathcal{A}$ dans la suite, est en $\mathcal{O}(\mathtt{n}2^{\mathtt{n}})$, car le `i = i + 1` coûte, au pire des cas, `n`. En réalisant des mesures, et en graphant le temps de cet algorithme divisé par $2^{\mathtt{n}}$, on remarque que ce le ratio n'est pas une droite de coefficient directeur `n`, mais une constante (à partir d'un certain rang). On doit donc faire une étude plus précise de la complexité, et faire le calcul de la somme plus proprement. Une étude plus fine nous montre que l'algorithme est beaucoup plus long pour les entiers en plaisances de deux, mais les autres nombres, on n'a pas besoin d'autant de calcul. Faisons cette étude plus fine.

On nomme $\mathcal{T}$ l'ensemble des tableaux de taille n contenant des valeurs dans $\{0, 1\}$. On a,

$$\text{Coût}_{\mathcal{A}}(n) = \sum_{t \in \mathcal{T}} \text{Coût}_{\text{Incr}}(t).$$

Partitionnons $\mathcal{T}$:

$$\mathcal{T}_i = \left\{ \begin{array}{|c|c|c|c|c|c|} \hline \ldots & i+1 & i & i-1 & \ldots & 0 \\ \hline \ldots & \mathbf{0} & \mathbf{1} & \mathbf{1} & \ldots & \mathbf{1} \\ \hline \end{array} \in \mathcal{T} \right\}.$$

Si $t \in \mathcal{T}_i$, on sait que $\text{Coût}_{\text{Incr}}(t) = i + 1$. Ainsi,

$$\begin{aligned} \sum_{t \in \mathcal{T}} \text{Coût}_{\text{Incr}}(t) &= \sum_{i=0}^{n-1} \sum_{t \in \mathcal{T}_i} \text{Coût}_{\text{Incr}}(t) \\ &= \sum_{i=0}^{n-1} |\mathcal{T}_i| \cdot (i+1) \\ &= \sum_{i=0}^{n-1} 2^{n-1-i} \, (i+1) \\ &= 2^n \sum_{i=1}^{n} \frac{i}{2^i} \\ &= 2^n \times \mathcal{O}(1) \\ &= \mathcal{O}(2^n) \end{aligned}$$

ce qui explique les résultats trouvés précédemment. L'incrementation `i = i + 1` est donc en $\mathcal{O}(1)$, et non en $\mathcal{O}(\log_2 \mathtt{i})$.

Définition : Étant donnée une structure (des éléments d'un type de données abstrait, TDA), $\mathbb{F}$, munie d'opérations $\mathbb{O}$ opérant sur $\mathbb{F}$ munis de fonctions de coût

$$\forall o \in \mathbb{O}, \quad C_o : \mathbb{F} \to \mathbb{R}^+.$$

Étant donné un élément initial $f_0 \in \mathbb{F}$ et une suite d'opérations $(o_1, \ldots, o_n) \in \mathbb{O}^n$, cela conduit donc à une suite d'éléments

$$f_0 \overset{o_1}{\rightsquigarrow} f_1 \overset{o_2}{\rightsquigarrow} f_2 \rightsquigarrow \cdots \rightsquigarrow f_n.$$

On appelle *complexité* de cette séquence, notée $\tilde{c}$,

$$\tilde{C}\big((o_1, \ldots, o_n), f_0\big) = \sum_{i=0}^{n} C_{o_i}(f_{i-1}).$$

On appelle alors *complexité amortie* depuis $f_0 \in \mathbb{F}$ la suite

$$C_{\mathrm{A}}(f_0, n) = \frac{1}{n} \sup_{(o_1,\dots,o_n)\in \mathcal{O}^n} \tilde{C}\big((o_1,\dots,o_n), f_0\big).$$

Exemple (Tableaux dynamiques) :
On s'intéresse aux tableaux à longueur variable : on alloue un tableau de petite taille, et on alloue plus de mémoire au besoin. On a une structure de tableau dynamique :

AlgorithmeAnnexe A.2 Agrandit(T), fonction agrandissant le tableau t

```
1 : Soit taille' = f(len(T))        ▷ f reste à déterminer
2 : On alloue T' de taille taille'
3 : On recopie T dans T'
4 : T ← T'
```

Cet algorithme a une complexité $\mathrm{Coût}_{\textsc{Agrandit}}(n) = n + f(n)$. On suppose que le tableau T est rempli jusqu'à la r-ième case.

AlgorithmeAnnexe A.3 Ajout(T, x), ajout d'un élément dans le tableau

```
1 : si len(T) = r alors
2 :     Agrandit(T)
3 : T[r] ← x
4 : r ← r + 1
```

On choisit la fonction f.

Cas 1 On choisit $f(n) = n + 1$. Soit une suite de n opérations Ajout depuis un tableau de taille 1, où $r = 0$. Ainsi,

$$f_0 \overset{\textsc{Ajout}}{\rightsquigarrow} f_1 \overset{\textsc{Ajout}}{\rightsquigarrow} \cdots \rightsquigarrow f_i \rightsquigarrow \cdots \rightsquigarrow f_{n-1} \overset{\textsc{Ajout}}{\rightsquigarrow} f_n.$$

La complexité de cette suite d'opérations est

$$\tilde{C}\big((o_1,\dots,o_n), f_0\big) = n + 2 \cdot \frac{n(n+1)}{2},$$

d'où la complexité amortie est de $C_{\mathrm{A}}(f_0, n) = \Theta(n)$.

Cas 2 On choisit $f(n) = 2n$. On somme les complexités : $2n$ (clairement par dessin). Ainsi, $C_{\mathrm{A}}(f_0, n) = \Theta(1)$.

Méthode (du potentiel) **:** Considérons une fonction $h : \mathbb{F} \to \mathbb{R}^+$ dite *de potentiel* telle que $h(f_0) = 0$. Intéressons nous alors à $C_o(f) = C_o(f) + h(\bar{f}) - h(f)$, où $f \overset{o}{\rightsquigarrow} \bar{f}$. Soit alors

$$f_0 \overset{o_1}{\rightsquigarrow} f_1 \overset{o_1}{\rightsquigarrow} f_2 \rightsquigarrow \cdots \rightsquigarrow f_n$$

une suite d'opérations. Alors,

$$\sum_{i=1}^{n} \underline{C}_{o_i}(f_{i-1}) = \sum_{i=1}^{n} \Big(C_{o_i}(f_{i-1}) + h(f_i) - h(f_{i-1}) \Big)$$
$$= \Big(\sum_{i=1}^{n} C_{o_i}(f_{i-1}) \Big) + \underbrace{h(f_n) - h(f_0)}_{\geqslant 0}$$

par télescopage. Ainsi,

$$\sum_{i=1}^{n} C_{o_i}(f_{i-1}) \leqslant \sum_{i=1}^{n} \underline{C}_{o_i}(f_{i-1}).$$

Exemple :
On applique la méthode du potentiel au cas 2 de l'exemple ci-avant. On rappelle que $\mathbb{F}$ est l'ensemble des tableaux. On pose la fonction

$$h : \mathbb{F} \longrightarrow \mathbb{R}^+$$
$$(T, r) \longmapsto 6\left(r - \frac{\text{len}(T)}{2}\right)$$

Inspectons alors

$$\underline{C}_{\text{Ajout}}(T, r) = C_{\text{Ajout}}(\underline{T}, \underline{r}) + 6\bar{r} - 3\,\text{len}(\bar{T}) - 3\underline{r} + 3\,\text{len}(\underline{T}).$$

Si $\text{len}(\underline{T}) = \underline{r}$, alors $C_{\text{Ajout}}(\underline{T}, \underline{r}) = 3\,\text{len}(\underline{T})$ et $\text{len}(\bar{T}) = 2\,\text{len}(\underline{T})$ et $\bar{r} = \underline{r} + 1$. D'où,

$$\underline{C}_{\text{Ajout}}(\underline{T}, \underline{r}) = 3\,\text{len}(\underline{T}) + 6\underline{r} + 6 - 6\,\text{len}(\underline{T}) - 6\underline{r} + 3\,\text{len}(\underline{T}) = 6.$$

Sinon, $\text{len}(\underline{T}) > \underline{r}$, alors $\text{len}(\bar{T}) = \text{len}(\underline{T})$ et $\bar{r} = \underline{r} + 1$. Ainsi,

$$\underline{C}_{\text{Ajout}}(\underline{T}, \underline{r}) = 1 + 6(\underline{r} + 1) - 6\,\text{len}(\bar{T}) - 6\underline{r} + 6\,\text{len}(\underline{T}) = 7$$

D'où

$$\sum_{i=1}^{n} C_{o_i}(f_{i-1}) \leqslant \sum_{i=1}^{n} \underline{C}_{o_i}(f_{i-1}) \leqslant 7n.$$

Le coût amorti est en $\mathcal{O}(1)$.

Exemple (Méthode du Banquier) :
On encode une file avec deux piles. Au moment de défiler, on doit potentiellement transvaser une pile dans une autre. Avec la méthode du Banquier, on a l'*intuition* que le coût amorti est constant.

On pose $\mathbb{F}$ l'ensemble des couples de piles (p_1, p_2). On a

$$C_{\text{défiler}}\big((p_1, p_2)\big) = \begin{cases} \text{taille } p_1 + 1 & \text{si } p_2 \text{ est vide} \\ 1 & \text{sinon} \end{cases}, \text{ et } C_{\text{enfiler}}\big((p_1, p_2)\big) = 1.$$

Soit h la fonction de potentiel définie comme

$$h : \mathbb{F} \longrightarrow \mathbb{R}^+$$
$$(p_1, p_2) \longmapsto \text{taille } p_1$$

Étudions alors $\underline{C}_{\text{défiler}}\big((p_1, p_2)\big)$.

— Si p_2 est vide, alors

$$\begin{aligned} C_{\text{défiler}}\big((p_1,p_2)\big) &= C_{\text{défiler}}\big((p_1,p_2)\big) + h\big((\bar{p}_1,\bar{p}_2)\big) - h\big((p_1,p_2)\big) \\ &= \text{taille } p_1 + 1 + \overbrace{\text{taille } \bar{p}_1}^{=0} - \text{taille } p_1 \\ &= 1. \end{aligned}$$

— Si p_2 n'est pas vide, alors

$$C_{\text{défiler}}\big((p_1,p_2)\big) = 1 + \underbrace{\text{taille } \bar{p}_1}_{=\text{taille } p_1} - \text{taille } p_1.$$

D'où, pour $(p_1,p_2) \in \mathbb{F}$, $C_{\text{défiler}}\big((p_1,p_2)\big) \leqslant 1$. De plus,

$$\begin{aligned} C_{\text{enfiler}}\big((p_1,p_2)\big) &= C_{\text{enfiler}}\big((p_1,p_2)\big) + h\big((\bar{p}_1,\bar{p}_2)\big) - h\big((p_1,p_2)\big) \\ &= 1 + \text{taille } \bar{p}_1 - \text{taille } p_1 \\ &= 2 \end{aligned}$$

Finalement, pour toute séquence d'opérations $o_1, \ldots, o_n$ initialisée à la file vide f_0, on a

$$\begin{aligned} \frac{1}{n}\tilde{C}\big((o_1,\ldots,o_n), f_0\big) &= \frac{1}{n}\sum_{i=0}^{n} C_{o_i}(f_{i-1}) \\ &\leqslant \frac{1}{n}\sum_{i=1}^{n} C_{o_i}(f_{i-1}) \\ &\leqslant 2 \end{aligned}$$

D'où, un coût amorti constant.

ANNEXE

B

ALGORITHMES DIJKSTRA ET $A^\star$

On s'intéresse, dans cette annexe, à l'algorithme $A^\star$. Cette annexe se situe à l'intersection des chapitres sur les graphes, et sur les jeux. L'algorithme $A^\star$ est une modification de l'algorithme de Dijkstra. Dans cette annexe, on prouvera la correction de l'algorithme $A^\star$.

On se place dans le contexte d'exécution d'un algorithme de calcul de plus cours chemin utilisant un tableau de distances μ, et le manipulant en n'effectuant que des opérations Relâcher. Notons le graphe $G = (V, E)$, le sommet source s. Notons également $d(\cdot, \cdot)$ la distance induite par les arêtes du graphe G. De plus, on notera $c(\cdot, \cdot)$ les coûts (positifs, non nuls) d'une arête de G. Notons $\ell(\cdot)$ les rongeurs des chemins.

Lemme 1 :

$$\forall (u, v) \in E, \quad d(s, v) \leqslant d(s, u) + c(u, v).$$

Preuve :
Soit $(u, v) \in E$. Soit γ_u un plus court chemin de s à u. Alors, $\gamma_u \cdot v$ est un chemin de s à v :

$$\ell(\gamma_u \cdot v) = \ell(\gamma_u) + c(u, v) = d(s, u) + c(u, v) \geqslant d(s, v).$$

□

Lemme 2 : Pour tout sommet u, la valeur de $\mu[u]$ est décroissant à mesure que l'algorithme s'exécute.

Preuve :
Soit μ et $\bar{\mu}$ les valeurs de μ avant et après une opération Relâcher(x, y). Pour tout sommet $v \neq y$, $\bar{\mu}[v] = \underline{\mu}[v]$. De plus, par disjonction de cas,

— ou bien $\bar{\mu}[y] = \underline{\mu}[y]$, ok.
— ou bien $\bar{\mu}[y] = \underline{\mu}[x] + c(x, y)$ lorsque $\underline{\mu}[x] + c(x, y) \leqslant \underline{\mu}[y]$, donc $\bar{\mu}[y] \leqslant \underline{\mu}[y]$, ok.

□

Lemme 3 : Supposons que l'algorithme ait initialisé μ de la manière suivante :

$$\forall u \in V, \qquad \mu[u] = \begin{cases} +\infty & \text{si } u \neq s \\ 0 & \text{sinon.} \end{cases}$$

Alors, tout au long de l'exécution de l'algorithme, pour tout sommet u, $\mu[u] \geqslant d(s,u)$.

Preuve : **Initialement** La propriété est vraie par hypothèse.

Hérédité ăSupposons vrai jusqu'à un certain état $\underline{\mu}$, pour une opération Relâcher(x,y). Pour tout sommet $v \neq y$, $\underline{\mu}[v] = \bar{\mu}[v] \geqslant d(s,v)$. De plus, par disjonction de cas,

- si $\bar{\mu}[y] = \underline{\mu}[y] \geqslant d(s,y)$;
- sinon si $\underline{\mu}[y] = \underline{\mu}[x] + c(x,y) \geqslant d(s,x) + c(x,y) \geqslant d(s,y)$ par hypothèse de récurrence, puis par lemme 1.

□

Corollaire : Si « à un moment » $\mu[u] = d(s,u)$, alors « pour toujours après » $\mu[u] = d(s,u)$. □

Lemme 4 : Si $(s,\ldots,u,v)$ est un plus court chemin de s à v tel que $\underline{\mu}[u] = d(s,u)$ « à un certain moment de l'exécution de l'algorithme. » Notons $\bar{\mu}$ obtenu par Relâcher(u,v).

Preuve :
On a

$$\bar{\mu} = \begin{cases} \underline{\mu}[v] & \text{si } \underline{\mu}[v] \text{ă} < \underline{\mu}[u] + c(u,v) \\ \underline{\mu}[u] + c(u,v) & \text{sinon.} \end{cases}$$

Par disjonction de cas,

- si $\underline{\mu}[v] < \underline{\mu}[u] + c(u,v) = d(s,u) + c(u,v) = d(s,v)$, et donc, en utilisant le lemme 3, $\bar{\mu}[v] = \underline{\mu}[v] = d(s,v)$.
- sinon, $\bar{\mu}[v] = \underline{\mu}[u] + c(u,v) = d(u,v) + c(u,v) = d(s,v)$.

□

Lemme 5 : Soit $(s = x_0, x_1, x_2, \ldots, x_n)$ un plus court chemin. Si on effectue des opérations Relâcher(x_i, x_{i+1}) dans l'ordre $0 \to n-1$, possiblement entremêlés avec d'autres opérations Relâcher, alors pour tout $i \in [\![0,n]\!]$, $\mu_{\text{final}}[x_i] = d(s,x_i)$.

Preuve (par récurrence) *:*
- Initialement, $\mu[x_0] = d(s,x_0) = d(s,s)$.
- Et, pour tout les i inférieurs stricts, $\mu[x_i] = d(s,x_i)$, on conclut par le lemme 4.

□

(De ce lemme découle l'algorithme de Bellman-Ford.)

Corollaire : L'algorithme Dijkstra est correct.

Preuve :
Soit $t \in V$, un sommet du graphe. Soit $(s = x_0, x_1, \ldots, x_{p-1}, x_p = t)$ un plus court chemin de s à t. Montrons que $\mu_{\text{final}}[t] = d(s,t)$. En utilisant le lemme 5, il suffit de montrer que Dijkstra relâche les arêtes dans cet ordre. Supposons les sommets extraits todo dans l'ordre $x_0, \ldots, x_i$, pour $i \in [\![0, p-1]\!]$.

Par l'absurde, supposons que Dijkstra sorte x_k de todo pour $k \in [\![i+2, p]\!]$. « À ce moment là, » on a

$$d(s, x_k) \leqslant \mu[x_k] \leqslant \mu[x_{i+1}] \leqslant d(s, x_{i+1}),$$

d'après le lemme 5, ce qui est absurde ($k > i+1$). □

Corollaire : L'algorithme $A^\star$ est correct.

AlgorithmeAnnexe B.1 Algorithme $A^\star$ (partiel)

```
1 : Procédure Relâcher(u, v)
2 :     si μ[v] > μ[u] + c(u, v) alors
3 :         μ[v] ← μ[u] + c(u, v)
4 :         π[v] ← u
5 :         η[v] ← μ[v] + h(v)
```

Preuve :
Par l'absurde, supposons que non. Soit $t \in V$, un sommet du graphe, tel que $\mu_{\text{final}}[t] \neq d(s,t)$. Donc $d = \mu_{\text{final}}[t] > d(s,t) = d^\star$. Soit $(s = x_0, x_1, \ldots, x_{p-1}, x_p = t)$ un plus court chemin de s à t de longueur $d^\star$. L'algorithme commence par visiter $x_0 = s$ et on relâche les arêtes sortantes. Alors, $\mu[x_i] = d(s, x_1)$ et $\eta[x_1] = \mu[x_1] + \mu[x_1] + h(x_1) \leqslant d(s, x_1) + d(x_1, t) = d(s, t) = d^\star < d$ par hypothèse. « À ce state, » $\eta[t] = \mu[t] + h(t) \geqslant d + 0$. Ainsi, x_1 devrait être choisi avant t. À un tel moment, $\mu[x_1] = d(s, x_1)$, on relâche alors ses arêtes sortantes ; en particulier x_1 et x_2. Ceci assure alors que $\mu[x_2] = d(s, x_2)$, et $\eta[x_2] = \mu[x_2] + h(x_2) \leqslant d(snx_2) + d(x_2, t) \leqslant d(s, t) = d^\star < d$. « De proche en proche, » alors que l'on choisit x_{p-1} dans todo, on a $\mu[x_{p-1}] = d(s, x_{p-1})$. On relâche alors $\mu[x_p] = d(s, x_p) = d^\star$. Or, $d = \mu_{\text{final}}[t] \leqslant \mu_{\text{à ce moment}}[t]$. Absurde. □

Exemple (ré-entrée dans todo) :
Exécution de l'algorithme $A^\star$ sur l'entrée ci-dessus. La pile todo est vaut donc $\not{s}, \not{a}, \not{b}, \not{t}, \not{a}$.

ANNEXE

C

DIVISER POUR RÉGNER

On se place dans un contexte dans l'idée de coder des algorithmes avec la stratégie « diviser pour régner. » Par exemple, on considère un algorithme de calcul de maximum divisant le tableau en deux à chaque itération.[1] Le calcul usuel de maximum est en $\mathcal{O}(n)$. On peut espérer que cet algorithme divisant le tableau ait une complexité logarithmique. Calculons cette complexité.

Soit $(C_n)_{n\in\mathbb{N}}$ la suite donnant le nombre de comparaisons entre éléments du tableau[2] effectuées par l'algorithme `aux_max` sur un intervalle de taille $|j-i+1|=n$. On a $C_0=0$, $C_1=0$, et pour tout $n\geqslant 2$, $C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} + 1$.

FigureAnnexe C.1

Soit $(v_p)_{p\in\mathbb{N}}$ la suite définie par $v_p = C_{2^p}$. Ainsi, $v_0 = 0$ et

$$v_{p+1} = C_{2^{p+1}} = C_{\left\lfloor \frac{2^p+1}{2} \right\rfloor} + C_{\left\lceil \frac{2^p+1}{2} \right\rceil} + 1 = 2v_p + 1.$$

1. C'est le principe des tournois sportifs.
2. *i.e.* le nombre d'appels à la fonction `max`.

Calculons v_p :

$$\begin{aligned} v_p &= 2v_{p-1} + 1 \\ &= 2(2v_{p-2} + 1) + 1 \\ &= 2^2 v_{p-2} + 2^1 + 2^0 \\ &\vdots \\ &= 2^p v_0 + \sum_{i=1}^{p-1} 2^i \\ &= 2^p - 1 \end{aligned}$$

Ce résultat ne s'applique que pour les puissances de 2, mais, par un argument de croissance, on peut en déduire un résultat pour tout entier n. Montrons que la suite $(C_n)_{n\in\mathbb{N}}$ est croissante. En effet, par récurrence forte, soit $n \geqslant 2$.

— Si n est pair, alors $C_n = 2 \times C_{\frac{n}{2}} + 1$ et $C_{n-1} = C_{\frac{n-2}{2}} + C_{\frac{n-2}{2}+1} + 1 = C_{\frac{n}{2}-1} + C_{\frac{n}{2}} + 1$. Et, $C_{\frac{n}{2}-1} \leqslant C_{\frac{n}{2}}$ par hypothèse de récurrence. D'où, $C_n \geqslant C_{n+1}$.

— De même si n est impair.

Ainsi, soit $n \in \mathbb{N}^\star$. On pose alors $p = \lfloor \log_2 n \rfloor$ donc $p \leqslant \log_2 n$. Par croissance de C, on a $C_{2^p} \leqslant C_n \leqslant C_{2^{p+1}}$ donc $v_p \leqslant c_n \leqslant v_{p+1}$. Ainsi,

$$\begin{aligned} 2^p - 1 \leqslant C_n &\leqslant 2^{p+1} - 1 \\ 2^{\lfloor \log_2 n \rfloor} - 1 \leqslant C_n &\leqslant 2^{\lfloor \log_2 n \rfloor + 1} - 1 \\ 2^{\log_2(n)-1} - 1 \leqslant C_n &\leqslant 2^{\log_2(n)+1} \\ \frac{1}{2}n - 1 \leqslant C_n &\leqslant 2n - 1 \end{aligned}$$

Ceci étant vrai pour tout $n \in \mathbb{N}^\star$, on a $C_n = \Theta(n)$. On peut remarquer que cet algorithme a une complexité équivalente à un algorithme itératif. Mais, [3] la complexité de cet algorithme s'améliore si les calculs se font en parallèles.

Autre exemple, le tri fusion a une complexité en $C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} + n$, ce qui donne une complexité en $\mathcal{O}(n \log n)$.

3. et c'est tout l'intérêt pour les tournois sportifs

ANNEXE

D

LEMME D'ARDEN ET RETOUR SUR LE THÉORÈME DE KLEENE

Exemple (Lemme d'Arden) :
Soient K et L deux langages. Résoudre $X = K \cdot X \cup L$ pour X un langage. (On trouve $X = K^\star \cdot L$.) On suppose que $\varepsilon \notin K$. On procède par double-inclusion.

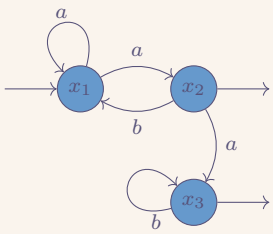
"$\supseteq$" Soit X un langage tel que $X = K \cdot X \cup L$. Montrons par récurrence « si w est un mot de X de taille n, alors $w \in K^\star \cdot L$.

- Si $n = 0$, alors $w \in L$ car $\varepsilon \notin K$. Ainsi, $w = \varepsilon \cdot w$ et $\varepsilon \in K^\star$. On en déduit que $w \in K^\star \cdot L$.
- Si $|w| = n$, alors
  - si $w \in L$, alors $w = \varepsilon \cdot w$ et donc $w \in K^\star L$.
  - si $w = v \cdot w'$ où $v \in K$ et $w' \in X$, alors $|w'| < |w|$. Ainsi, par hypothèse de récurrence, $w' \in K^\star \cdot L$. Ainsi, $v \cdot w' \in K^\star \cdot L$.

Ainsi, $X \subseteq K^\star \cdot L$.

"$\subseteq$" Soit $w \in K^\star \cdot L$. Il existe donc $n \in \mathbb{N}$, $(v_1, \ldots, v_n) \in K^n$ et $w' \in L$ tels que $w = v_1 \cdot \ldots \cdot v_n \cdot w'$. Alors, $w' \in X$ donc $v_n \cdot w' \in X$ donc … donc $v_1 v_2 \ldots v_n w' \in X$. Ainsi, $w \in X$.

Exemple :
On considère l'automate ci-dessous.



FigureAnnexe D.1 – Automate exemple ($\mathcal{A}$)

On pose $X_i = \mathscr{L}\big((\Sigma, \mathbb{Q}, \{i\}, F, \delta)\big)$, où x_i est l'unique point de départ. Ainsi, $\mathscr{L}(\mathscr{A}) = \bigcup_{i\in I} X_i$. Déterminons les valeurs de X_1, X_2 et X_3. On applique un algorithme similaire au « pivot de Gauß. »

$$\left.\begin{array}{lll} X_1 & = \{a\}\cdot X_2 \cup \{a\}X_1 \\ X_2 & = \{b\}\cdot X_1 \cup \{a\}\cdot X_3 \cup \{\varepsilon\} \\ X_3 & = \{b\}X_3 \cup \{\varepsilon\} \end{array}\right\} \iff \begin{cases} X_1 & = \{a\}^\star\cdot\{a\}\cdot X_2 \\ X_2 & = \mathscr{L}(ba^\star\cdot a)X_2 \cup \{a\}X_3 \cup \{\varepsilon\} \\ X_3 & = \{b\}\cdot X_3 \cup \{\varepsilon\} \end{cases}$$

$$\iff \begin{cases} X_1 & = \mathscr{L}(a^\star\cdot a)X_2 \\ X_2 & = \mathscr{L}\big((b\cdot a^\star\cdot a)^\star\big)\cdot\big(\{a\}X_3 \cup \{\varepsilon\}\big) \\ X_3 & = \{b\}X_3 \cup \{\varepsilon\} \end{cases}$$

$$\iff \begin{cases} X_1 & = \mathscr{L}(a^\star\cdot a)X_2 \\ X_2 & = \mathscr{L}\big((b\cdot a^\star\cdot a)^\star\big)\cdot\big(\{a\}X_3 \cup \{\varepsilon\}\big) \\ X_3 & = \mathscr{L}(b^\star) \end{cases}$$

$$\iff \begin{cases} X_1 & = \mathscr{L}(a^\star\cdot a)X_2 \\ X_2 & = \mathscr{L}\big((ba^\star a)^\star\cdot(ab^\star \mid \varepsilon)\big) \\ X_3 & = \mathscr{L}(b^\star) \end{cases}$$

$$\iff \begin{cases} X_1 & = \mathscr{L}\big(a^\star\cdot a\cdot(ba^\star a)^\star\cdot(ab^\star \mid \varepsilon)\big) \\ X_2 & = \mathscr{L}\big((ba^\star a)^\star\cdot(ab^\star \mid \varepsilon)\big) \\ X_3 & = \mathscr{L}(b^\star) \end{cases}$$

On peut généraliser la méthode employée dans l'exemple précédent pour montrer que tout langage reconnaissable est régulier.

ANNEXE

E

TAS ET FILES DE PRIORITÉS

L'objectif d'une file de priorité est de récupérer l'élément de priorité minimale. On organise cette structure de données sous forme d'un arbre tournois. [1] Un arbre tournois est un arbre dont la priorité d'un nœud est supérieur à celle de ses fils. On impose une structure supplémentaire, l'arbre doit être parfait : l'arbre est complet jusqu'à l'avant dernier niveau, où il est replis à gauche. On définit plusieurs opérations sur cette file de priorité (de type `fp`, où les éléments sont de type `elem`) :

— `insérer` : `fp` $\to$ `elem` $\to$ `fp` qui insère un élément,

— `lire_min` : `fp` $\to$ `elem` qui récupère l'élément de priorité minimale,

— `supprimer_min` : `fp` $\to$ `fp` qui supprime l'élément de priorité minimale,

— (`diminuer_priorité` : `fp` $\to$ `elem` $\to$ `fp`), [2]

— `créer` : ( ) $\to$ `fp`.

On définit un type `btree`, représentant un arbre binaire, et on implémente les opérations ci-dessous en OCaml.

```
type 'a btree =
| Node of 'a * 'a btree * 'a btree
| Empty
```

CodeAnnexe E.1 – Définition du type `btree`

Pour l'opération `créer`, on retourne `Empty` (cela donne une complexité en $\Theta(1)$). Pour l'opération `insérer`, on insère l'élément comme feuille (de manière à conserver la propriété de l'arbre parfait), et on inverse le nœud avec son parent jusqu'à ce que la propriété soit vérifiée ($\Theta(\log_2 n)$). Pour l'opération `lire_min`, on lit la racine ($\Theta(1)$). Pour l'opération `supprimer_min`, on permute la racine et le dernier nœud (*i.e.* le nœud le plus à droite de hauteur maximale), et on restore la structure d'arbre tournois en permutant un nœud et son fils de valeur minimale, et en répétant ($\Theta(\log_2 n)$). Pour trouver le dernier nœud, on garde en mémoire cet emplacement. On peut aussi implémenter cet algorithme avec un tableau (▷ TP), ou avec une liste triée (mais la complexité est moins bien).

1. Un arbre tournois n'est pas un arbre binaire de recherche.
2. Cette opération est parfois omise car trop compliquée à implémenter.

ANNEXE

F

ARITHMÉTIQUE

Un des premiers algorithmes codé est l'algorithme d'Euclide pour calculer le PGCD. Pour $a \neq 0$, on a $a \wedge 0 = a$ et $a \wedge b = b \wedge (a \bmod b)$. On peut le coder en OCaml avec la fonction `euclid` suivante.

```
let rec euclid (a: int) (b: int): int =
  (* Hyp: a >= b et a != 0 *)
  if b = 0 then a
  else euclide b (a mod b)
```

CodeAnnexe F.1 – Algorithme d'Euclide calculant le PGCD

Quelle est la complexité de cet algorithme ? On représente le nombre d'appels récursifs à `euclid`, et on devine une courbe logarithmique. En notant (u_n) les divisions euclidiennes réalisées et (q_n) les quotients, ainsi, on $u_n = q_{n-1} \cdot u_{n-1} + u_{n-2}$. Alors, $\texttt{euclid}(u_n, u_{n-1}) = \cdots = \texttt{euclid}(u_3, u_2) = \texttt{euclid}(u_2, u_1) = \texttt{euclid}(u_1, u_0)$.

En fixant la complexité, on cherche les valeurs de (u_n) maximisant les appels récursifs. On peut montrer par récurrence que si $\texttt{euclid}(a, b)$ conduit à n appels récursifs de `euclid`, alors $a \geqslant F_n$ et $b \geqslant F_{n-1}$, où $(F_n)_{n\in\mathbb{N}}$ est la suite de Fibonacci.

En effet, soit un tel couple (a, b). Alors, $(b, a \bmod b)$ conduit à $n-1$ appels récursifs donc $b \geqslant F_{n-1}$ et $a \bmod b \geqslant F_{n-2}$ par hypothèse de recurrence. Et, $a = bq + (a \bmod b)$ et donc $a \geqslant F_{n-1} + F_{n-2} = F_n$.

De plus, pour tout $n \in \mathbb{N} \setminus \{0, 1\}$, $F_n \geqslant \varphi^{n-2}$ où φ est le nombre d'or.[1] En effet, $F_2 = 1 \geqslant \varphi^0 = 1$ et $F_3 = 2 \geqslant \varphi^1 = \varphi = (1 + \sqrt{5})/2$. Et, $F_n = F_{n-1} + F_{n-2} \geqslant \varphi^{n-3} + \varphi^{n-4} \geqslant \varphi^{n-4}(1 + \varphi) \geqslant \varphi^{n-2}$.

Soient (p, q), où $p \geqslant q$, une entrée de l'algorithme d'Euclide. Si l'appel $\texttt{euclid}(p, q)$ conduit à plus de $\lceil \log_\varphi p \rceil + 4$ appels, alors $p \geqslant F_{\lceil \log_\varphi p \rceil + 4} \geqslant \varphi^{\lceil \log_\varphi p \rceil + 4 - 2} > \varphi^{\log_\varphi p} = p$, ce qui est absurde.

Ceci conduit à une complexité en $\mathcal{O}(\log p)$.

Soit n un entier premier. Pour l'algorithme RSA, on cherche un inverse de $a \in \mathbb{Z}/n\mathbb{Z}$: on cherche $b \in \mathbb{Z}/n\mathbb{Z}$ tel que $ab \equiv 1\ [1]$. D'après le théorème de Bézout, on a $au + nv = 1$ car $a \wedge n = 1$. L'inverse est v. D'où l'importance des coefficients de Bézout.

1. C'est la solution positive de $X^2 - X - 1 = 0$.

Comment calculer les coefficients de Bézout ? On peut utiliser l'algorithme d'Euclide. On pose r_n la valeur de a après n appels récursifs.

r_i	u_i		v_i	
$r_0 = a$	1	a	0	b
$r_1 = b$	0	b	1	a
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
r_{i-2}	u_{i-2}	a	v_{i-2}	b
r_{i-1}	u_{i-1}	a	v_{i-1}	b

TableAnnexe F.1 – Valeurs de r_i avec invariant $r_i = au_i + bv_i$

Alors,

$$\begin{aligned} r_i &= u_{i-2}a + v_{i-2}b - (r_{i-2}/r_{i-1})(u_{i-1}a + v_{i-1}b) \\ &= \big(u_{i-2} - (r_{i-2}/r_{i-1})u_{i-1}\big)a + \big(v_{i-2} - (r_{i-2}/r_{i-1})v_{i-1}\big)b \end{aligned}$$

Ainsi, on a bien $\mathrm{pgcd}(a, b) = u_{n-1}a + v_{n-1}b$.

ANNEXE

G

ARBRES ROUGES-NOIRS

Un arbre rouge-noir est un cas particulier des arbres binaires de recherches. On l'utilise notamment pour représenter des ensembles, on veut donc réaliser deux opérations simples : l'insertion et le test d'appartenance. Initialement, on pense représenter un ensemble par une liste triée. Mais, on utilise plutôt un arbre binaire pour représenter des données avec une hauteur logarithmique, contrairement à une hauteur linéaire. Les arbres binaires de recherches sont des arbres dans lesquels ont peut réaliser une dichotomie.

```
1  type 'a btree = E | N of 'a * 'a btree * 'a btree
2
3  let rec mem (x: 'a) (t: 'a btree): bool =
4    match t with
5    | E -> false
6    | N(y, g, d) ->
7        if x < y       then mem x g
8        else if x = y then true
9        else                mem x d
10
11 let rec insere (x: 'a) (t: 'a btree): 'a btree =
12   match t with
13   | E -> false
14   | N(y, g, d) ->
15       if x < y       then N(y, insere x g, d)
16       else if x = y then t
17       else                N(y, g, insere x d)
```

CodeAnnexe G.1 – Arbre binaire de recherche

La fonction `mem` permet de réaliser ce test d'appartenance et la fonction `insere` insère l'insertion dans l'arbre. Ainsi, on peut représenter un ensemble avec le type `'a btree`.

Mais, cet arbre peut être déséquilibré, et l'utilisation de la dichotomie ne donne pas de résultats très avantageux. On utilise donc un arbre *auto-équilibrant*, comme les AVL du 1er DM. Pour les AVL, la différence de hauteur est -1, 0 ou 1.

On introduit donc le concept d'arbre rouge-noir. Un arbre rouge-noir est un arbre parfait, qui a une certaine « élasticité. » On colorie chaque nœuds pour imposer des contraintes sur cette élasticité. Les branches de l'arbre a une longueur de rupture. Un arbre contenant uniquement des nœuds noirs est un arbre parfait. Et, entre deux nœuds noirs, on peut insérer un nœud rouge. Un arbre rouge-noir vérifie donc les trois propriétés suivantes :

(a) la racine est noire,

(*b*) le père d'un nœud rouge est noir,

(*c*) la hauteur noir de chaque feuille externe est constante. [important]

Une *feuille externe* est, dans le code OCaml, l'expression `E` ; et, la *hauteur noir* d'une feuille externe est le nombre de nœuds noirs depuis la racine. On définit la *hauteur noir* d'un arbre comme la hauteur noir de chaque feuille externe (qui est constante).

Le problème est l'insertion d'un nœud. Insérer un nœud noir est, en général, plus dangereux car il modifie la hauteur noir de tout l'arbre. On préfère donc insérer un nœud rouge, sauf dans le cas de la racine.

On considère donc la propriété (*c*) comme invariante. En effet, corriger un arbre pour valider la propriété (*a*) ou la propriété (*b*) est bien plus simple.

On traite tous les cas dans le diaporama sur *cahier-de-prépa*, et on réalise l'exemple sur les lettres A, L, G, O, R, I, T, H, M, E.

Pour supprimer un nœud dans un arbre binaire classique, on peut le remplacer par le maximal de son sous-arbre droit, ou le minimum de son sous-arbre gauche. Si on supprime un nœud ayant un seul fils, on n'a qu'à re-brancher le sous-arbre. Pour les arbres rouges-noirs, c'est supprimer un nœud noir qui pose problème. Pour cela, on introduit les nœuds doublement noirs, qui comptent pour deux dans la hauteur noir. Ainsi, on supprime le nœud noir et on remplace les autres nœuds par des nœuds doublement noirs. L'algorithme n'a donc qu'à faire remonter le nœud doublement noir, jusqu'à la racine, où il sera transformé en nœud simplement noir.

Un arbre de hauteur h a une hauteur noir $\mathrm{bh} \geqslant h/2$. Et, la taille, *i.e.* le nombre de nœuds, est supérieure à $2^{\mathrm{bh}} - 1$. On conclut que $h \leqslant 2\log_2(\mathrm{taille}+1)$. De même par la propriété (*c*) permet de conclure que $h = \Theta(\log_2 \mathrm{taille})$.

ANNEXE

H

COMPLEXITÉ MOYENNE

Dans les annexes et cours précédents, on a vu la complexité « pire cas » et la complexité amortie. On considère les nombres d'opérations possibles pour toute entrée de taille n. La complexité « pire cas » est la complexité obtenue en prenant le $\max$ d'opération possibilité. Pour la complexité moyenne, on suppose que chaque ensemble d'entrée de taille n est munit d'une probabilité P_n. Par exemple, on considère qu'une entrée est une permutation de n éléments, *i.e.* un élément de $\mathfrak{S}_n$. On suppose que chaque entrée arrive avec équiprobabilité. Ainsi, $\forall \sigma \in \mathfrak{S}_n$, $P_n(\sigma) = 1/n!$. Ainsi, on a

$$C_{\max}(n) = \max_{\sigma \in \mathfrak{S}_n} C(s) \text{ et } C_{\text{moy}} = \sum_{\sigma \in \mathfrak{S}_n} P(\sigma) \cdot C(\sigma).$$

La complexité moyenne est la moyenne des complexité pondérées par les probabilités.

Exemple :
On considère l'algorithme ci-dessous.

AlgorithmeAnnexe H.1 Calcul d'inverse d'une permutation

Entrée $\sigma \in \mathfrak{S}_n$ et $i \in [\![1, n]\!]$
Sortie $j \in [\![1, n]\!]$ tel que $\sigma(j) = i$.
1 : **pour** $j \in [\![1, n]\!]$ **faire**
2 : ⌊ **si** $\sigma(j) = i$ **alors retourner** j

On munit $\mathfrak{S}_n$ de la probabilité uniforme. Soit $i \in [\![1, n]\!]$. Notons, pour tout $j \in [\![1, n]\!]$,

$\mathfrak{S}_n^j = \{\sigma \in \mathfrak{S}_n \mid \sigma(j) = i\}$. Remarquons que $\mathfrak{S}_n = \bigcup_{j=1}^n \mathfrak{S}_n^j$. Ainsi,

$$\begin{aligned} C_{\text{moy}} &= \sum_{j=1}^{n} \sum_{\sigma \in \mathfrak{S}_n^j} P_n(\sigma) C(\sigma) \\ &= \sum_{j=1}^{n} j \times \frac{|\mathfrak{S}_n^j|}{n!} \\ &= \sum_{j=1}^{n} j \times \frac{(n+1)!}{n!} \\ &= \frac{1}{n} \cdot \frac{n(n+1)}{2} \\ &= \frac{n+1}{2} \end{aligned}$$

ANNEXE

I

PREUVES DE CORRECTION POUR LES FONCTIONS RÉCURSIVES

On considère l'insertion dans un arbre binaire de recherche. Démontrons qu'elle est correcte. On adopte les notations de l'annexe G sur les arbres rouges-noirs. Montrons que, pour tout ABR t, et pour tout étiquette $e \in \mathbb{E}$,

$$\text{étiquettes}(\texttt{insertion}(t, x)) = \text{étiquettes}(t) \cup \{x\}.$$

Montrons le par induction.

1. Si $t = \texttt{E}$. Soit $x \in \mathbb{E}$:

$$\text{étiquettes}(\texttt{insertion}(\texttt{E}, x)) = \text{étiquettes}(N(x, \texttt{E}, \texttt{E})) = \{x\} = \underbrace{\text{étiquettes}(\texttt{E})}_{\varnothing} \cup \{x\}.$$

2. Si $t = \texttt{N}(y, g, d)$. Soit $x \in \mathbb{E}$.
   — si $x < y$, alors

$$\begin{aligned}
\text{étiquettes}(\texttt{insertion}(t, x)) &= \text{étiquettes}(\texttt{N}(y, \texttt{insertion}(g, x), d)) \\
&= \{x\} \cup \text{étiquettes}(\texttt{insertion}(g, x)) \cup \text{étiquettes}(d) \\
&= \{y\} \cup \text{étiquettes}(g) \cup \{x\} \cup \text{étiquettes}(d) \\
&= \text{étiquettes}(\texttt{N}(y, g, d)) \cup \{x\} \\
&= \text{étiquettes}(t) \cup \{x\}
\end{aligned}$$

   — on procède de même pour les autres cas.

On procède de même pour les autres propriétés.

INDEX

A

algorithme
  de type Las Vegas . . . 81
  de type Monte-Carlo . . . 81
  déterministe . . . 79
  probabiliste . . . 79
alphabet . . . 41
arbre . . . 133
  couvrant . . . 133
    de poids minimum . . . 133
    poids . . . 133
arbre de preuve . . . 144
automate
  avec ε-transitions . . . 52
  complet . . . 48
  déterministe . . . 48
  généralisé . . . 69
    « bien détouré » . . . 69
    langage reconnu . . . 69
  langage reconnu . . . 47
  langage sans ε . . . 53
  local . . . 64
    standard . . . 64
  sans ε-transitions . . . 46
  émondé . . . 52
  équivalent . . . 50
  état accessible . . . 51
  état co-accessible . . . 51
axiome . . . 143

B

booléens
  (ensemble) . . . 23
  opération $+$ . . . 24
  opération $\bar{\square}$ . . . 24
  opération $\cdot$ . . . 24

C

clause
  conjonctive . . . 30
  disjonctive . . . 30
conclusion . . . 143

D

donnée . . . 92

E

ensemble de mots
  de longueur n, Σ^n . . . 41
  de longueur positive ou nulle, $\Sigma^\star$ 41
  de longueur positive, Σ^+ . . . 41
ensemble des formules . . . 20
ensemble défini par induction nommée 12
entropie
  jeu de données classifié . . . 98
  partition . . . 98
  variable aléatoire . . . 97
environment propositionnel . . . 24
expression régulière . . . 44
  langage . . . 45
  linéaire . . . 62
  variables . . . 45
exécution . . . 47

F

faux négatif . . . 94
faux positif . . . 94
fonction
  calculable . . . 109
    en temps polynômial . . . 116
  partielle . . . 106
  totale . . . 106
fonction `interprète` . . . 112

fonction booléenne 24
fonction de classification 92
forme
  n-CNF 119
  n-FNC 119
  normale conjonctive 30
  normale disjonctive 30
formule
  conséquence sémantique 26, 27
  fonction booléenne associée 25
  insatisfiable 26
  interprétation 25
  satisfiable 26
  tautologique 26
  valide 26
  équivalence 26
formule logique 20

G

graphe
  CFC 124
  accessibilité 123
  bordure 125
  co-accessibilité 123
  composante fortement connexe . . 124
  couplage 137
    maximal 137
    maximum 137
  ensemble fortement connexe 124
  fortement connexe 124
  induit 124
  notation $u \sim_G v$ 124
  parcours 126
    en largeur 127
    en profondeur 127
    partitionnement associé 126
    point de régénération 126
  rang 128
  relation $\preccurlyeq_T$ 128
  réduit 125
  sommet ouvert 127
  transposé 129
  tri
    préfixe 128
    topologique 128

H

hauteur (ensemble défini par induction nommée) 13

J

jeu de données 92
  classifié 92

L

langage 42
  concaténation 42
  concaténation répétée L^n 43
  décidable 110
  facteur de taille 2 58
  lettre préfixe 58
  lettre suffixe 58
  local 58
    engendré 58
  non facteur 58
  reconnaissable 47
  régulier 43
  substitution 63
lettre 41
littéral 30
  notation $\mathrm{lit}_\rho(p)$ 28
logique
  constante 149
  fonction 149
  premier ordre
    formules 149
  signature du premier ordre 149

M

machine 108
  de complexité polynômiale 116
  ensemble $\textsc{Time}(f)$ 116
  langage 110
  nombre d'opérations élémentaires $(C^{\mathcal{M}}(w))$ 115
    maximal pour un mot de taille n $(C^{\mathcal{M}}_n)$ 115
machine universelle 112
matrice de confusion 93
mesure de dissimilarité 101
modèle (formule logique) 26
modèle de calcul 108
mot 41
  concaténation 41
  facteur 42
  longueur 41
  préfixe 42
  sans ε $(\tilde{w})$ 53
  sous-mot 42
  suffixe 42
  vide ε 41

O

ordre bien fondé (ensemble ordonné) 5

P

problème 106
  3SAT 119
  n-CNF-SAT 119
  NP 117
  NP-complet 121
  NP-difficile 118
  P 116
  de décision 107
  décidable 110
prémisses 143

R

raffinement
  classe d'équivalence 126
relation
  déterministe 105
  totale à gauche 105

relation d'ordre
(ensemble défini par induction nommée) . . . 13
lexicographique ($\preccurlyeq_\ell$) . . . 8, 10
produit ($\preccurlyeq_\times$) . . . 6
sous-formule . . . 23
règle
d'induction . . . 12
de base . . . 12
de construction nommée . . . 12
règle de base . . . 143
règle de construction de preuves . . . 143
réduction . . . 114
polynômiale . . . 118

S

signature de données . . . 91
substitution . . . 22
application à une formule . . . 22
clés . . . 22
composée . . . 22
suite de transitions . . . 47
acceptante . . . 47
exécution . . . 47
étiquette . . . 47
système de preuves . . . 144
complétude . . . 145
correction . . . 144
séquent . . . 143
sérialisation . . . 111

T

taille (d'une formule) . . . 21
taille d'entrée . . . 115
type UnionFind . . . 135

V

variable propositionnelle . . . 20
variables (d'une formule) . . . 21
vrai négatif . . . 94
vrai positif . . . 94

élément minimal (ensemble ordonné) . . . 5
être prouvable . . . 144