

# Table des matières

1	Index thématique	5
2	(CCINP) Arbres Binaires de Recherche * (CCINP 0, ex A, corrigé (R) — oral - 74 lignes)	27
3	(CCINP) Dédution naturelle * (CCINP 0, ex A, corrigé (R) — oral - 160 lignes)	29
4	(CCINP) Langages réguliers * (CCINP 0, ex A, corrigé (R) — oral - 120 lignes)	31
5	(CCINP) Mutex * (CCINP 0, ex A, corrigé (R) — oral - 109 lignes)	33
6	(CCINP) SQL * (CCINP 0, ex A, corrigé (R) — oral - 95 lignes)	35
7	(CCINP) ID3 * (CCINP 0, ex A, corrigé — oral - 175 lignes)	37
8	(CCINP) Tableaux * (CCINP 0, ex B, corrigé (R) — oral - 151 lignes)	39
9	(CCINP) Tableaux autoréférents * (CCINP 0, ex B, corrigé (R) — oral - 195 lignes)	41
10	(CCINP) Arbres * (CCINP 0, ex B, corrigé — oral - 247 lignes)	43
11	(CCINP) Graphes * (CCINP 0, ex B, corrigé — oral - 144 lignes)	45
12	(CCINP) SAT * (CCINP 0, ex B, corrigé — oral - 148 lignes)	47
13	(CCINP) Sac à dos * (CCINP 0, ex B, corrigé — oral - 178 lignes)	49
14	(CCINP) Activation de processus * (CCINP 23, ex A, corrigé (R) — oral - 153 lignes)	51
15	(CCINP) Grammaires algébriques * (CCINP 23, ex A, corrigé (R) — oral - 123 lignes)	53
16	(CCINP) Minima locaux dans des arbres (Couverture dans des arbres) * (CCINP 23, ex A, corrigé (R) — oral - 148 lignes)	55
17	(CCINP) Formules propositionnelles croissantes * (CCINP 23, ex A, corrigé — oral - 142 lignes)	57
18	(CCINP) Langages locaux * (CCINP 23, ex B, corrigé (R) — oral - 215 lignes)	59
19	(CCINP) Programmation dynamique pour la récolte de fleurs * (CCINP 23, ex B, corrigé (R) — oral - 204 lignes)	61
20	(CCINP) chemins simples sans issue * (CCINP 23, ex B, corrigé (R) — oral - 199 lignes)	63
21	(CCINP) calculs avec les flottants * (CCINP 23, ex B, corrigé — oral - 206 lignes)	65
22	(CCINP) chomp * (CCINP 24, ex A, corrigé (R) — oral - 228 lignes)	67
23	(CCINP) Grammaires pour des langages de programmation * (CCINP 24, ex A, corrigé, à déboguer — ds2526 - 90 lignes)	69
24	(CCINP) une relation d'équivalence * (CCINP 24, ex A, corrigé — oral - 118 lignes)	71
25	(CCINP) Fonctions pour la détection de motifs * (CCINP 24, ex B, corrigé (R) — oral - 131 lignes)	73
26	(CCINP) HORNSAT * (CCINP 24, ex B, corrigé (R) — oral - 186 lignes)	75
27	(CCINP) Mots de Dyck * (CCINP 24, ex B, corrigé — oral - 157 lignes)	77
28	(CCINP) Complétude de Nor et d'une variante Nor * (CCINP 25, ex 1, retour élève modifié, corrigé — oral - 150 lignes)	79

29 (CCINP) Graphes, A* * (CCINP 25, ex 1, retour élève très incomplet complété, corrigé — oral - 116 lignes)	81
30 (CCINP) Autour d'un jeu d'alignement * (CCINP 25, ex 2, retour élève complété, corrigé — oral - 201 lignes)	83
31 (CCINP) Graphes bipartis * (CCINP 25, ex 2, retour élève complété, corrigé — oral - 136 lignes)	85
32 (CCINP) Sous-mots * (CCINP 25, ex 2, retour élève très modifié, corrigé — oral - 155 lignes)	87
33 (CCINP) Mots sur un alphabet muni d'un ordre ** (CCINP 25, ex A, retour élève modifié, corrigé — oral - 171 lignes)	89
34 (CCINP) k plus proches voisins * (CCINP 25, ex A, retour élève modifié, figure refaite, corrigé — oral - 108 lignes)	91
35 (CCINP) UF et Kruskal * (CCINP 25, ex A, retour élève très incomplet reconstitué, corrigé — oral - 116 lignes)	93
36 (CCINP) Algorithmes du cours sur les automates, complexités ** (CCINP 25, ex A, retour élève, corrigé — oral - 232 lignes)	95
37 (CCINP) Bin packing * (CCINP 25, ex A, retour élève, corrigé — oral - 143 lignes)	97
38 (CCINP) Jeu sur un graphe * (CCINP 25, ex A, retour élève, corrigé — oral - 120 lignes)	99
39 (CCINP) Mots conjugués * (CCINP 25, ex A, retour élève, corrigé — oral - 110 lignes)	101
40 (CCINP) tas min * (CCINP 25, ex A, retour élève, rallongé, corrigé — oral - 92 lignes)	103
41 (CCINP) Dénombrement du nombre de façons d'obtenir un score avec des dés. * (CCINP 25, ex B, retour élève complété, corrigé — oral - 178 lignes)	105
42 (CCINP) Permutation de Langford * (CCINP 25, ex B, retour élève complété, corrigé — oral - 188 lignes)	107
43 (CCINP) Rendu de monnaie * (CCINP 25, ex B, retour élève complété, corrigé — oral - 126 lignes)	109
44 (CCINP) Dénombrement du nombre de façons d'obtenir un score avec des dés. * (CCINP 25, ex B, retour élève, corrigé — oral - 150 lignes)	111
45 (CCINP) Expressions régulières en OCaml * (CCINP 25, ex B, retour élève, corrigé — oral - 157 lignes)	113
46 (ENS) Jeu des jetons *** (ENS 23 MP, corrigé — oral - 153 lignes)	115
47 (ENS) Monoïdes et Langages *** (ENS 23 MP, corrigé — oral - 238 lignes)	117
48 (ENS) Élimination des coupures dans MLL et MALL **** (ENS 23, corrigé partiellement — oral - 385 lignes)	119
49 (ENS) Graphes parfaits *** (ENS 23, corrigé — oral - 251 lignes)	121
50 (ENS) Inférence de type *** (ENS 23, corrigé — oral - 279 lignes)	123
51 (ENS) Ordonnancement *** (ENS 23, corrigé — oral - 201 lignes)	125
52 (ENS) Permutations triables par pile *** (ENS 23, corrigé — oral - 184 lignes)	127
53 (ENS) Raisonnements ensemblistes *** (ENS 23, corrigé — oral - 234 lignes)	129
54 (ENS) Théorème des amis *** (ENS 23, corrigé — oral - 202 lignes)	131
55 (ENS) Structures pliables et traversables *** (ENS 23, partiellement corrigé — oral - 314 lignes)	133
56 (ENS) Élimination des coupures dans MLL et MALL *** (ENS 23, réécriture plus raisonnable par N.Carré — oral - 377 lignes)	135
57 (ENS) Composition Monadique *** (ENS 24, corrigé très partiellement — oral - 191 lignes)	137
58 (ENS) Dédution de messages *** (ENS 24, corrigé — oral - 232 lignes)	139

59 (ENS) Filtrage par motif en OCaml *** (ENS 24, corrigé — oral - 323 lignes)	141
60 (ENS) Implémentation des circuits réversibles *** (ENS 24, corrigé — oral - 218 lignes)	143
61 (ENS) Résolution d'inéquations linéaires *** (ENS 24, corrigé — oral - 195 lignes)	145
62 (ENS) Mots partiels et Théorème de Dilworth *** (ENS 24, partiellement corrigé, niveau inapproprié, à déboguer — oral - 184 lignes)	147
63 (ENS) Terminaison de lambda ref *** (ENS 24, partiellement corrigé, niveau surréaliste — oral - 238 lignes)	149
64 (ENS) Calcul de l'arbre de dominance *** (ENS 25, non corrigé — oral - 83 lignes)	151
65 (ENS) Forme SSA, arbres de domination *** (ENS 25, non corrigé — oral - 120 lignes)	155
66 (ENS) Tri Externe Polyphasé *** (ENS 25, non corrigé — oral - 114 lignes)	157
67 (MT) Déduction Naturelle * (MT 0, ex 1, corrigé (R) — oral - 200 lignes)	159
68 (MT) Relations entre les nombres de sommets et d'arêtes de graphes * (MT 0, ex 1, corrigé (R) — oral - 112 lignes)	161
69 (MT) couplages * (MT 0, ex 1, corrigé (R) — oral - 64 lignes)	163
70 (MT) logique * (MT 0, ex 1, corrigé (R) — oral - 100 lignes)	165
71 (MT) bdd pour livraison * (MT 0, ex 1, corrigé — oral - 78 lignes)	167
72 (MT) Arbres Binaires de Recherche * (MT 0, ex 2, corrigé (R) — oral - 167 lignes)	169
73 (MT) entrelacements de mots * (MT 0, ex 2, corrigé (R) — oral - 105 lignes)	171
74 (MT) montée et descente d'un bus * (MT 0, ex 2, corrigé (R) — oral - 104 lignes)	173
75 (MT) subset-sum * (MT 0, ex 2, corrigé (R) — oral - 143 lignes)	175
76 (MT) graphes * (MT 0, ex 2, corrigé — oral - 121 lignes)	177
77 (MT) Graphes bipartis * (MT 24, ex 1, corrigé (R) — oral - 92 lignes)	179
78 (MT) Tas binaires * (MT 24, ex 1, corrigé (R) — oral - 129 lignes)	181
79 (MT) déduction naturelle * (MT 24, ex 1, corrigé (R) — oral - 141 lignes)	183
80 (MT) Classification hiérarchique ascendante * (MT 24, ex 1, corrigé, le graphique n'est pas l'original (R) — oral - 44 lignes)	185
81 (MT) Tas binaires * (MT 24, ex 1, corrigé — oral - 137 lignes)	187
82 (MT) Automates de Büchi *** (MT 24, ex 2, (modifié) corrigé (R) — oral - 117 lignes)	189
83 (MT) Décidabilité de programmes engendrés par une grammaire * (MT 24, ex 2, corrigé (R) — oral - 140 lignes)	191
84 (MT) Automates, langages à saut * (MT 24, ex 2, corrigé, complété au jugé pour les dernières questions (R) — oral - 93 lignes)	193
85 (MT) plus courts chemins dans des graphes * (MT 24, ex 2, corrigé, retour d'oral (R) — oral - 129 lignes)	195
86 (MT) Automates et décidabilité ** (MT 25, ex 1, retour élève complété, corrigé — oral - 87 lignes)	197
87 (MT) Récompenses pour un site web * (MT 25, ex 1, retour élève remanié, corrigé (à préciser/compléter) — oral - 68 lignes)	199
88 (MT) Algorithmique des textes * (MT 25, ex 1, retour élève remanié, corrigé — oral - 72 lignes)	201
89 (MT) Langages ** à *** (MT 25, ex 1, retour élève remanié, corrigé — oral - 80 lignes)	203
90 (MT) Piles et files * (MT 25, ex 1, retour élève retravaillé, corrigé — oral - 67 lignes)	205

91 (MT) Points fixes dans des tableaux * (MT 25, ex 1, retour élève retravaillé, corrigé — oral - 79 lignes)	207
92 (MT) Langages des préfixes * (MT 25, ex 1, retour élève à modifier, corrigé — oral - 59 lignes)	209
93 (MT) Substitution régulière ** à **** (MT 25, ex 1, retour élève à reprendre, corrigé — oral - 92 lignes)	211
94 (MT) Langages et facteurs * (MT 25, ex 1, retour élève, corrigé (R) — oral - 150 lignes)	213
95 (MT) Dédution naturelle * (MT 25, ex 1, retour élève, corrigé — oral - 97 lignes)	215
96 (MT) k plus proches voisins * (MT 25, ex 1, retour élève, corrigé — oral - 46 lignes)	217
97 (MT) lemme d'Arden ** (MT 25, ex 1, retour élève, corrigé — oral - 94 lignes)	219
98 (MT) * (MT 25, ex 2, retour élève remanié, corrigé — oral - 117 lignes)	221
99 (MT) Représentation fusionnée de deux mots ** (MT 25, ex 2, retour élève remanié, corrigé (R) — oral - 116 lignes)	223
100(MT) Graphes et couleurs * à *** (MT 25, ex 2, retour élève remanié, corrigé — oral - 128 lignes)	225
101(MT) Langages rationnels et ultime périodicité *** (MT 25, ex 2, retour élève remanié, corrigé — oral - 82 lignes)	227
102(MT) Théorème de compacité ** (MT 25, ex 2, retour élève remanié, corrigé — oral - 101 lignes)	229
103(MT) Programmation dynamique *** (MT 25, ex 2, retour élève retravaillé, corrigé — oral - 131 lignes)	231
104(MT) Automates alternants *** (MT 25, ex 2, retour élève très buggué largement retravaillé, corrigé — oral - 294 lignes)	233
105(MT) Graphes : implémentation en C * (MT 25, ex 2, retour élève à compléter/vérifier++, corrigé — oral - 98 lignes)	235
106(MT) League et Overwatch ** (MT 25, ex 2, retour élève à reprendre, non corrigé — oral - 91 lignes)	237
107(MT) coloriage de graphe * (MT 25, ex 2, retour élève, 3/4 questions manquantes, corrigé — oral - 103 lignes)	239
108(MT) CNS pour cycles eulériens * (MT 25, ex 2, retour élève, corrigé — oral - 71 lignes)	241
109(MT) Cycles et Chemins simples ** (MT 25, ex 2, retour élève, corrigé — oral - 64 lignes)	243
110(X) Algorithme Union-Find *** (X 23, corrigé — oral - 151 lignes)	245
111(X) Bob l'écureuil *** (X 23, corrigé — oral - 141 lignes)	247
112(X) Langages rationnels et lemme de l'étoile *** (X 23, corrigé — oral - 118 lignes)	249
113(X) Rationnalité de langages *** (X 24, corrigé, complété au jugé pour les trois dernières questions — oral - 213 lignes)	251
114(X) Algorithme du tri lent *** (X 24, corrigé — oral - 147 lignes)	253
115(X) Codage par couleur *** (X 24, corrigé — oral - 154 lignes)	255
116(X) Chaînes et anti-chaînes *** (X 25, non corrigé — oral - 44 lignes)	257
117(X) Diamètre dans un arbres *** (X 25, récupéré par élève, non corrigé — oral - 35 lignes)	259
118(X) Vis et écrous *** (X 25, récupéré par élève, non corrigé — oral - 81 lignes)	261
119(Écrit) Logique trivaluée * à ** (Écrit MP 25, utilisable oral, corrigé — oral - 275 lignes)	263

# Chapitre 1

## Index thématique

### Algorithmes d'approximation

- 36-Bin packing (CCINP 25, ex A, retour élève, **corrigé** — oral - 143 lignes) \*  
*algorithmes d'approximation, complexité*  
ccinpbinpack.tex  
**VOIR**
- 74-subset-sum (MT 0, ex 2, **corrigé** (R) — oral - 143 lignes) \*  
*complexité, algorithmes d'approximation, algorithmes gloutons*  
mtsubsum.tex  
**VOIR**
- 97- (MT 25, ex 2, retour élève remanié, **corrigé** — oral - 117 lignes) \*  
*complexité, algorithmes d'approximation, réduction*  
mtapprox2.tex  
**VOIR**

### Algorithmes gloutons

- 12-Sac à dos (CCINP 0, ex B, **corrigé** — oral - 178 lignes) \*  
*algorithmique, c, algorithmes gloutons, backtracking, branch and bound*  
ccinpsacados.tex  
**VOIR**
- 74-subset-sum (MT 0, ex 2, **corrigé** (R) — oral - 143 lignes) \*  
*complexité, algorithmes d'approximation, algorithmes gloutons*  
mtsubsum.tex  
**VOIR**

### Algorithmes probabilistes

- 11-SAT (CCINP 0, ex B, **corrigé** — oral - 148 lignes) \*  
*logique propositionnelle, algorithmes probabilistes*  
ccinpsat.tex  
**VOIR**
- 99-Graphes et couleurs (MT 25, ex 2, retour élève remanié, **corrigé** — oral - 128 lignes) \* à \*\*\*  
*graphes, algorithmes probabilistes*  
mtgraph7.tex  
**VOIR**
- 117-Vis et écrous (X 25, récupéré par élève, **non corrigé** — oral - 81 lignes) \*\*\*  
*algorithmique, tri, complexité, preuves, algorithmes probabilistes*  
xecroux.tex  
**VOIR**

## Algorithmique

- 7-Tableaux (CCINP 0, ex B, **corrigé** (R) — oral - 151 lignes) \*  
*algorithmique, c*  
`ccinptableau.tex`  
**VOIR**
- 8-Tableaux autoréférents (CCINP 0, ex B, **corrigé** (R) — oral - 195 lignes) \*  
*algorithmique, backtracking, ocaml*  
`ccinpautoreferent.tex`  
**VOIR**
- 9-Arbres (CCINP 0, ex B, **corrigé** — oral - 247 lignes) \*  
*arbres, ocaml, algorithmique, pseudo-code*  
`ccinparbre2.tex`  
**VOIR**
- 12-Sac à dos (CCINP 0, ex B, **corrigé** — oral - 178 lignes) \*  
*algorithmique, c, algorithmes gloutons, backtracking, branch and bound*  
`ccinpsacados.tex`  
**VOIR**
- 23-une relation d'équivalence (CCINP 24, ex A, **corrigé** — oral - 118 lignes) \*  
*algorithmique, graphes*  
`ccinpreleq.tex`  
**VOIR**
- 29-Autour d'un jeu d'alignement (CCINP 25, ex 2, retour élève complété, **corrigé** — oral - 201 lignes) \*  
*ocaml, algorithmique, jeux*  
`ccinpalgo.tex`  
**VOIR**
- 31-Sous-mots (CCINP 25, ex 2, retour élève très modifié, **corrigé** — oral - 155 lignes) \*  
*algorithmique, c*  
`ccinpsousmots.tex`  
**VOIR**
- 34-UF et Kruskal (CCINP 25, ex A, retour élève très incomplet reconstitué, **corrigé** — oral - 116 lignes) \*  
*graphes, algorithmique, c*  
`ccinpalgo6.tex`  
**VOIR**
- 39-tas min (CCINP 25, ex A, retour élève, rallongé, **corrigé** — oral - 92 lignes) \*  
*algorithmique, Structures de données, Ocaml*  
`ccinptas.tex`  
**VOIR**
- 40-Dénombrément du nombre de façons d'obtenir un score avec des dés. (CCINP 25, ex B, retour élève complété, **corrigé** — oral - 178 lignes) \*  
*algorithmique, c*  
`ccinpalgo3.tex`  
**VOIR**
- 41-Permutation de Langford (CCINP 25, ex B, retour élève complété, **corrigé** — oral - 188 lignes) \*  
*algorithmique, c*  
`ccinpalgo5.tex`  
**VOIR**
- 42-Rendu de monnaie (CCINP 25, ex B, retour élève complété, **corrigé** — oral - 126 lignes) \*  
*algorithmique, ocaml*  
`ccinpalgo2.tex`  
**VOIR**
- 43-Dénombrément du nombre de façons d'obtenir un score avec des dés. (CCINP 25, ex B, retour élève, **corrigé** — oral - 150 lignes) \*  
*algorithmique, ocaml, programmation dynamique*

ccinpalgo4.tex

**VOIR**

- 49-Inférence de type (ENS 23, **corrigé** — oral - 279 lignes) \*\*\*  
*algorithmique, langages fonctionnels, pseudo-code*  
 ensinferencetypes.tex  
**VOIR**
  
- 50-Ordonnancement (ENS 23, **corrigé** — oral - 201 lignes) \*\*\*  
*algorithmique, complexité*  
 ensordo.tex  
**VOIR**
  
- 51-Permutations triables par pile (ENS 23, **corrigé** — oral - 184 lignes) \*\*\*  
*algorithmique, graphes, complexité*  
 ensperm.tex  
**VOIR**
  
- 60-Résolution d'inéquations linéaires (ENS 24, **corrigé** — oral - 195 lignes) \*\*\*  
*ocaml, algorithmique, complexité*  
 ensresolutioninequationslineaires.tex  
**VOIR**
  
- 63-Calcul de l'arbre de dominance (ENS 25, **non corrigé** — oral - 83 lignes) \*\*\*  
*algorithmique, arbres, complexité*  
 ensarbredom.tex  
**VOIR**
  
- 65-Tri Externe Polyphasé (ENS 25, **non corrigé** — oral - 114 lignes) \*\*\*  
*algorithmique*  
 enstriexterne.tex  
**VOIR**
  
- 90-Points fixes dans des tableaux (MT 25, ex 1, retour élève retravaillé, **corrigé** — oral - 79 lignes) \*  
*algorithmique, complexité, preuves, pseudo-code*  
 mttab1.tex  
**VOIR**
  
- 109-Algorithmes Union-Find (X 23, **corrigé** — oral - 151 lignes) \*\*\*  
*algorithmique, complexité, structures de données*  
 xunionfind.tex  
**VOIR**
  
- 110-Bob l'écureuil (X 23, **corrigé** — oral - 141 lignes) \*\*\*  
*algorithmique, graphes*  
 xbobecureuil.tex  
**VOIR**
  
- 113-Algorithmes du tri lent (X 24, **corrigé** — oral - 147 lignes) \*\*\*  
*algorithmique, tri, complexité, preuves*  
 xtrilent.tex  
**VOIR**
  
- 114-Codage par couleur (X 24, **corrigé** — oral - 154 lignes) \*\*\*  
*algorithmique, complexité, graphes, backtracking*  
 xcodagecouleur.tex  
**VOIR**
  
- 116-Diamètre dans un arbres (X 25, récupéré par élève, **non corrigé** — oral - 35 lignes) \*\*\*  
*algorithmique, arbres, complexité*  
 xdiametre.tex  
**VOIR**
  
- 117-Vis et écrous (X 25, récupéré par élève, **non corrigé** — oral - 81 lignes) \*\*\*  
*algorithmique, tri, complexité, preuves, algorithmes probabilistes*  
 xecroux.tex

## VOIR

## Algorithmique des textes

- 24-Fonctions pour la détection de motifs (CCINP 24, ex B, **corrigé** (R) — oral - 131 lignes) \*  
*algorithmique des textes, langages, ocaml*  
langccinp2.tex  
**VOIR**
- 87-Algorithmique des textes (MT 25, ex 1, retour élève remanié, **corrigé** — oral - 72 lignes) \*  
*algorithmique des textes*  
mttxt2.tex  
**VOIR**
- 98-Représentation fusionnée de deux mots (MT 25, ex 2, retour élève remanié, **corrigé** (R) — oral - 116 lignes) \*\*  
*algorithmique des textes, programmation dynamique*  
mttxt1.tex  
**VOIR**

## Arbres

- 1-Arbres Binaires de Recherche (CCINP 0, ex A, **corrigé** (R) — oral - 74 lignes) \*  
*arbres, complexité*  
arbreCCINPA2.tex  
**VOIR**
- 9-Arbres (CCINP 0, ex B, **corrigé** — oral - 247 lignes) \*  
*arbres, ocaml, algorithmique, pseudo-code*  
ccinparbre2.tex  
**VOIR**
- 15-Minima locaux dans des arbres (Couverture dans des arbres) (CCINP 23, ex A, **corrigé** (R) — oral - 148 lignes) \*  
*arbres, complexité*  
arbreCCINPA.tex  
**VOIR**
- 63-Calcul de l'arbre de dominance (ENS 25, **non corrigé** — oral - 83 lignes) \*\*\*  
*algorithmique, arbres, complexité*  
ensarbredom.tex  
**VOIR**
- 64-Forme SSA, arbres de domination (ENS 25, **non corrigé** — oral - 120 lignes) \*\*\*  
*grammaires, arbres*  
ensformessa.tex  
**VOIR**
- 71-Arbres Binaires de Recherche (MT 0, ex 2, **corrigé** (R) — oral - 167 lignes) \*  
*arbres, complexité, programmation dynamique*  
mtarbres1.tex  
**VOIR**
- 77-Tas binaires (MT 24, ex 1, **corrigé** (R) — oral - 129 lignes) \*  
*arbres, tri, structures de données, complexité*  
mttas.tex  
**VOIR**
- 80-Tas binaires (MT 24, ex 1, **corrigé** — oral - 137 lignes) \*  
*arbres, tri, structures de données, complexité, pseudo-code*  
mttas2.tex  
**VOIR**
- 116-Diamètre dans un arbres (X 25, récupéré par élève, **non corrigé** — oral - 35 lignes) \*\*\*

*algorithmique, arbres, complexité*

`xdiametre.tex`

**VOIR**

## Automates

- 3-Langages réguliers (CCINP 0, ex A, **corrigé** (R) — oral - 120 lignes) \*  
*langages, Automates, lemme de l'étoile*  
`langccinp3.tex`  
**VOIR**
- 17-Langages locaux (CCINP 23, ex B, **corrigé** (R) — oral - 215 lignes) \*  
*langages, ocaml, automates*  
`langloccinp1.tex`  
**VOIR**
- 32-Mots sur un alphabet muni d'un ordre (CCINP 25, ex A, retour élève modifié, **corrigé** — oral - 171 lignes) \*\*  
*langages, automates*  
`ccinplang.tex`  
**VOIR**
- 35-Algorithmes du cours sur les automates, complexités (CCINP 25, ex A, retour élève, **corrigé** — oral - 232 lignes) \*\*  
*langages, automates*  
`ccinplang3.tex`  
**VOIR**
- 38-Mots conjugués (CCINP 25, ex A, retour élève, **corrigé** — oral - 110 lignes) \*  
*langages, automates, lemme de l'étoile*  
`ccinplang2.tex`  
**VOIR**
- 72-entrelacements de mots (MT 0, ex 2, **corrigé** (R) — oral - 105 lignes) \*  
*langages, automates, programmation dynamique*  
`mtlang.tex`  
**VOIR**
- 81-Automates de Büchi (MT 24, ex 2, (modifié) **corrigé** (R) — oral - 117 lignes) \*\*\*  
*langages, automates*  
`mtbuchi.tex`  
**VOIR**
- 83-Automates, langages à saut (MT 24, ex 2, **corrigé**, complété au jugé pour les dernières questions (R) — oral - 93 lignes) \*  
*langages, automates, lemme de l'étoile*  
`mtlang2.tex`  
**VOIR**
- 85-Automates et décidabilité (MT 25, ex 1, retour élève complété, **corrigé** — oral - 87 lignes) \*\*  
*langages, automates, décidabilité*  
`mtautom.tex`  
**VOIR**
- 88-Langages (MT 25, ex 1, retour élève remanié, **corrigé** — oral - 80 lignes) \*\* à \*\*\*  
*langages, automates*  
`mtlang4.tex`  
**VOIR**
- 92-Substitution régulière (MT 25, ex 1, retour élève à reprendre, **corrigé** — oral - 92 lignes) \*\* à \*\*\*\*  
*langages, automates, grammaires*  
`mtlang7.tex`  
**VOIR**
- 93-Langages et facteurs (MT 25, ex 1, retour élève, **corrigé** (R) — oral - 150 lignes) \*  
*langages, automates*

mtlang3.tex

**VOIR**

- 103-Automates alternants (MT 25, ex 2, retour élève très buggué largement retravaillé, **corrigé** — oral - 294 lignes) \*\*\*  
*langages, automates, jeux*  
mtautomatesalternants.tex  
**VOIR**
- 111-Langages rationnels et lemme de l'étoile (X 23, **corrigé** — oral - 118 lignes) \*\*\*  
*langages, automates, lemme de l'étoile*  
xlangrat.tex  
**VOIR**
- 112-Rationnalité de langages (X 24, **corrigé**, complété au jugé pour les trois dernières questions — oral - 213 lignes) \*\*\*  
*langages, automates, lemme de l'étoile*  
xlangrat2.tex  
**VOIR**

### Backtracking

- 8-Tableaux autoréférents (CCINP 0, ex B, **corrigé** (R) — oral - 195 lignes) \*  
*algorithmique, backtracking, ocaml*  
ccinpautoreferent.tex  
**VOIR**
- 12-Sac à dos (CCINP 0, ex B, **corrigé** — oral - 178 lignes) \*  
*algorithmique, c, algorithmes gloutons, backtracking, branch and bound*  
ccinpsacados.tex  
**VOIR**
- 19-chemins simples sans issue (CCINP 23, ex B, **corrigé** (R) — oral - 199 lignes) \*  
*graphes, ocaml, complexité, backtracking*  
ccinpchemins\_simples.tex  
**VOIR**
- 114-Codage par couleur (X 24, **corrigé** — oral - 154 lignes) \*\*\*  
*algorithmique, complexité, graphes, backtracking*  
xcodagecouleur.tex  
**VOIR**

### Branch and bound

- 12-Sac à dos (CCINP 0, ex B, **corrigé** — oral - 178 lignes) \*  
*algorithmique, c, algorithmes gloutons, backtracking, branch and bound*  
ccinpsacados.tex  
**VOIR**

### C

- 7-Tableaux (CCINP 0, ex B, **corrigé** (R) — oral - 151 lignes) \*  
*algorithmique, c*  
ccinp tableau.tex  
**VOIR**
- 10-Graphes (CCINP 0, ex B, **corrigé** — oral - 144 lignes) \*  
*graphes, c*  
ccinpgraphe.tex  
**VOIR**
- 12-Sac à dos (CCINP 0, ex B, **corrigé** — oral - 178 lignes) \*

*algorithmique, c, algorithmes gloutons, backtracking, branch and bound*

`ccinpsacados.tex`

**VOIR**

- 18-Programmation dynamique pour la récolte de fleurs (CCINP 23, ex B, **corrigé** (R) — oral - 204 lignes) \*  
*programmation dynamique, c*  
`ccinprecoltefleurs.tex`  
**VOIR**
- 20-calculs avec les flottants (CCINP 23, ex B, **corrigé** — oral - 206 lignes) \*  
*représentation des nombres, c*  
`ccinplottants.tex`  
**VOIR**
- 26-Mots de Dyck (CCINP 24, ex B, **corrigé** — oral - 157 lignes) \*  
*langages, c, complexité*  
`ccinpdycck.tex`  
**VOIR**
- 31-Sous-mots (CCINP 25, ex 2, retour élève très modifié, **corrigé** — oral - 155 lignes) \*  
*algorithmique, c*  
`ccinpsousmots.tex`  
**VOIR**
- 34-UF et Kruskal (CCINP 25, ex A, retour élève très incomplet reconstitué, **corrigé** — oral - 116 lignes) \*  
*graphes, algorithmique, c*  
`ccinpalgo6.tex`  
**VOIR**
- 40-Dénombrément du nombre de façons d'obtenir un score avec des dés. (CCINP 25, ex B, retour élève complété, **corrigé** — oral - 178 lignes) \*  
*algorithmique, c*  
`ccinpalgo3.tex`  
**VOIR**
- 41-Permutation de Langford (CCINP 25, ex B, retour élève complété, **corrigé** — oral - 188 lignes) \*  
*algorithmique, c*  
`ccinpalgo5.tex`  
**VOIR**
- 82-Décidabilité de programmes engendrés par une grammaire (MT 24, ex 2, **corrigé** (R) — oral - 140 lignes) \*  
*grammaires, décidabilité, c, logique propositionnelle*  
`mthaltbool.tex`  
**VOIR**
- 104-Graphes : implémentation en C (MT 25, ex 2, retour élève à compléter/vérifier++, **corrigé** — oral - 98 lignes) \*  
*graphes, c*  
`mtgraph11.tex`  
**VOIR**

## Circuits

- 59-Implémentation des circuits réversibles (ENS 24, **corrigé** — oral - 218 lignes) \*\*\*  
*circuits*  
`enscircuitsreversibles.tex`  
**VOIR**

## Classification hiérarchique ascendante

- 79-Classification hiérarchique ascendante (MT 24, ex 1, **corrigé**, le graphique n'est pas l'original (R) — oral - 44 lignes) \*  
*ia, classification hiérarchique ascendante*

mtia.tex

**VOIR**

## Complexité

- 1-Arbres Binaires de Recherche (CCINP 0, ex A, **corrigé** (R) — oral - 74 lignes) \*  
*arbres, complexité*  
 arbreCCINPA2.tex  
**VOIR**
- 13-Activation de processus (CCINP 23, ex A, **corrigé** (R) — oral - 153 lignes) \*  
*complexité, réduction*  
 procesCCINP23.tex  
**VOIR**
- 15-Minima locaux dans des arbres (Couverture dans des arbres) (CCINP 23, ex A, **corrigé** (R) — oral - 148 lignes) \*  
*arbres, complexité*  
 arbreCCINPA.tex  
**VOIR**
- 19-chemins simples sans issue (CCINP 23, ex B, **corrigé** (R) — oral - 199 lignes) \*  
*graphes, ocaml, complexité, backtracking*  
 ccinchemins\_simples.tex  
**VOIR**
- 25-HORNSAT (CCINP 24, ex B, **corrigé** (R) — oral - 186 lignes) \*  
*logique propositionnelle, complexité, réduction*  
 ccinphornsat.tex  
**VOIR**
- 26-Mots de Dyck (CCINP 24, ex B, **corrigé** — oral - 157 lignes) \*  
*langages, c, complexité*  
 ccinpdycck.tex  
**VOIR**
- 36-Bin packing (CCINP 25, ex A, retour élève, **corrigé** — oral - 143 lignes) \*  
*algorithmes d'approximation, complexité*  
 ccinpbnpack.tex  
**VOIR**
- 50-Ordonnancement (ENS 23, **corrigé** — oral - 201 lignes) \*\*\*  
*algorithmique, complexité*  
 ensordo.tex  
**VOIR**
- 51-Permutations triables par pile (ENS 23, **corrigé** — oral - 184 lignes) \*\*\*  
*algorithmique, graphes, complexité*  
 ensperm.tex  
**VOIR**
- 60-Résolution d'inéquations linéaires (ENS 24, **corrigé** — oral - 195 lignes) \*\*\*  
*ocaml, algorithmique, complexité*  
 ensresolutioninequationslineaires.tex  
**VOIR**
- 63-Calcul de l'arbre de dominance (ENS 25, **non corrigé** — oral - 83 lignes) \*\*\*  
*algorithmique, arbres, complexité*  
 ensarbredom.tex  
**VOIR**
- 71-Arbres Binaires de Recherche (MT 0, ex 2, **corrigé** (R) — oral - 167 lignes) \*  
*arbres, complexité, programmation dynamique*  
 mtarbres1.tex  
**VOIR**

- 74-subset-sum (MT 0, ex 2, **corrigé** (R) — oral - 143 lignes) \*  
*complexité, algorithmes d'approximation, algorithmes gloutons*  
`mtsubsum.tex`  
**VOIR**
  
- 76-Graphes bipartis (MT 24, ex 1, **corrigé** (R) — oral - 92 lignes) \*  
*graphes, logique propositionnelle, parcours de graphes, complexité*  
`mtgraph5.tex`  
**VOIR**
  
- 77-Tas binaires (MT 24, ex 1, **corrigé** (R) — oral - 129 lignes) \*  
*arbres, tri, structures de données, complexité*  
`mttas.tex`  
**VOIR**
  
- 80-Tas binaires (MT 24, ex 1, **corrigé** — oral - 137 lignes) \*  
*arbres, tri, structures de données, complexité, pseudo-code*  
`mttas2.tex`  
**VOIR**
  
- 84-plus courts chemins dans des graphes (MT 24, ex 2, **corrigé**, retour d'oral (R) — oral - 129 lignes) \*  
*graphes, complexité, programmation dynamique, pseudo-code*  
`mtgraph4.tex`  
**VOIR**
  
- 89-Piles et files (MT 25, ex 1, retour élève retravaillé, **corrigé** — oral - 67 lignes) \*  
*structures de données, complexité*  
`mtstruct1.tex`  
**VOIR**
  
- 90-Points fixes dans des tableaux (MT 25, ex 1, retour élève retravaillé, **corrigé** — oral - 79 lignes) \*  
*algorithmique, complexité, preuves, pseudo-code*  
`mttab1.tex`  
**VOIR**
  
- 97- (MT 25, ex 2, retour élève remanié, **corrigé** — oral - 117 lignes) \*  
*complexité, algorithmes d'approximation, réduction*  
`mtapprox2.tex`  
**VOIR**
  
- 109-Algorithmes Union-Find (X 23, **corrigé** — oral - 151 lignes) \*\*\*  
*algorithmique, complexité, structures de données*  
`xunionfind.tex`  
**VOIR**
  
- 113-Algorithmes du tri lent (X 24, **corrigé** — oral - 147 lignes) \*\*\*  
*algorithmique, tri, complexité, preuves*  
`xtrilent.tex`  
**VOIR**
  
- 114-Codage par couleur (X 24, **corrigé** — oral - 154 lignes) \*\*\*  
*algorithmique, complexité, graphes, backtracking*  
`xcodagecouleur.tex`  
**VOIR**
  
- 116-Diamètre dans un arbre (X 25, récupéré par élève, **non corrigé** — oral - 35 lignes) \*\*\*  
*algorithmique, arbres, complexité*  
`xdiametre.tex`  
**VOIR**
  
- 117-Vis et écrous (X 25, récupéré par élève, **non corrigé** — oral - 81 lignes) \*\*\*  
*algorithmique, tri, complexité, preuves, algorithmes probabilistes*  
`xecroux.tex`  
**VOIR**

## Concurrence

- 4-Mutex (CCINP 0, ex A, **corrigé** (R) — oral - 109 lignes) \*  
*concurrency, ocaml*  
`concccinp1.tex`  
**VOIR**
- 73-montée et descente d'un bus (MT 0, ex 2, **corrigé** (R) — oral - 104 lignes) \*  
*concurrency*  
`mtprocess.tex`  
**VOIR**
- 86-Récompenses pour un site web (MT 25, ex 1, retour élève remanié, **corrigé** (à préciser/compléter) — oral - 68 lignes) \*  
*concurrency*  
`mtconcl.tex`  
**VOIR**
- 105-League et Overwatch (MT 25, ex 2, retour élève à reprendre, **non corrigé** — oral - 91 lignes) \*\*  
*concurrency*  
`mtsync.tex`  
**VOIR**

## Décidabilité

- 82-Décidabilité de programmes engendrés par une grammaire (MT 24, ex 2, **corrigé** (R) — oral - 140 lignes) \*  
*grammars, decidability, c, logic propositional*  
`mthaltbool.tex`  
**VOIR**
- 85-Automates et décidabilité (MT 25, ex 1, retour élève complété, **corrigé** — oral - 87 lignes) \*\*  
*languages, automates, decidability*  
`mtautom.tex`  
**VOIR**

## Déduction naturelle

- 2-Déduction naturelle (CCINP 0, ex A, **corrigé** (R) — oral - 160 lignes) \*  
*deduction naturelle*  
`dednatCCINPA.tex`  
**VOIR**
- 57-Déduction de messages (ENS 24, **corrigé** — oral - 232 lignes) \*\*\*  
*deduction naturelle, reduction*  
`ensdeductiondemessages.tex`  
**VOIR**
- 66-Déduction Naturelle (MT 0, ex 1, **corrigé** (R) — oral - 200 lignes) \*  
*deduction naturelle*  
`mtlog1.tex`  
**VOIR**
- 78-déduction naturelle (MT 24, ex 1, **corrigé** (R) — oral - 141 lignes) \*  
*deduction naturelle*  
`mtlog3.tex`  
**VOIR**
- 94-Déduction naturelle (MT 25, ex 1, retour élève, **corrigé** — oral - 97 lignes) \*  
*deduction naturelle*  
`mtlog5.tex`  
**VOIR**

## Grammaires

- 14-Grammaires algébriques (CCINP 23, ex A, **corrigé** (R) — oral - 123 lignes) \*  
*grammaires*  
`grammccinp1.tex`  
**VOIR**
- 22-Grammaires pour des langages de programmation (CCINP 24, ex A, **corrigé, à déboguer** — ds2526 - 90 lignes) \*  
*grammaires, langages*  
`langccinp1.tex`  
**VOIR**
- 64-Forme SSA, arbres de domination (ENS 25, **non corrigé** — oral - 120 lignes) \*\*\*  
*grammaires, arbres*  
`ensformessa.tex`  
**VOIR**
- 82-Décidabilité de programmes engendrés par une grammaire (MT 24, ex 2, **corrigé** (R) — oral - 140 lignes) \*  
*grammaires, décidabilité, c, logique propositionnelle*  
`mthaltbool.tex`  
**VOIR**
- 91-Langages des préfixes (MT 25, ex 1, retour élève à modifier, **corrigé** — oral - 59 lignes) \*  
*langages, grammaires, lemme de l'étoile*  
`mtlang6.tex`  
**VOIR**
- 92-Substitution régulière (MT 25, ex 1, retour élève à reprendre, **corrigé** — oral - 92 lignes) \*\* à \*\*\*\*  
*langages, automates, grammaires*  
`mtlang7.tex`  
**VOIR**

## Graphes

- 10-Graphes (CCINP 0, ex B, **corrigé** — oral - 144 lignes) \*  
*graphes, c*  
`ccinpgraphe.tex`  
**VOIR**
- 19-chemins simples sans issue (CCINP 23, ex B, **corrigé** (R) — oral - 199 lignes) \*  
*graphes, ocaml, complexité, backtracking*  
`ccinpchemins_simples.tex`  
**VOIR**
- 23-une relation d'équivalence (CCINP 24, ex A, **corrigé** — oral - 118 lignes) \*  
*algorithmique, graphes*  
`ccinppreleq.tex`  
**VOIR**
- 28-Graphes, A\* (CCINP 25, ex 1, retour élève très incomplet complété, **corrigé** — oral - 116 lignes) \*  
*graphes*  
`ccinpgraphe2.tex`  
**VOIR**
- 30-Graphes bipartis (CCINP 25, ex 2, retour élève complété, **corrigé** — oral - 136 lignes) \*  
*graphes, ocaml*  
`ccinpgraphe3.tex`  
**VOIR**
- 34-UF et Kruskal (CCINP 25, ex A, retour élève très incomplet reconstitué, **corrigé** — oral - 116 lignes) \*  
*graphes, algorithmique, c*  
`ccinpalgo6.tex`  
**VOIR**
- 37-Jeu sur un graphe (CCINP 25, ex A, retour élève, **corrigé** — oral - 120 lignes) \*

*graphes, jeux*  
ccinpjeu1.tex  
**VOIR**

- 45-Jeu des jetons (ENS 23 MP, **corrigé** — oral - 153 lignes) \*\*\*  
*jeux, graphes*  
ensjeujeton.tex  
**VOIR**
- 48-Graphes parfaits (ENS 23, **corrigé** — oral - 251 lignes) \*\*\*  
*graphes*  
ensgraphesparfaits.tex  
**VOIR**
- 51-Permutations triables par pile (ENS 23, **corrigé** — oral - 184 lignes) \*\*\*  
*algorithmique, graphes, complexité*  
ensperm.tex  
**VOIR**
- 52-Raisonnements ensemblistes (ENS 23, **corrigé** — oral - 234 lignes) \*\*\*  
*graphes, logique propositionnelle*  
ensensembles.tex  
**VOIR**
- 53-Théorème des amis (ENS 23, **corrigé** — oral - 202 lignes) \*\*\*  
*graphes*  
enstheoremedesamis.tex  
**VOIR**
- 61-Mots partiels et Théorème de Dilworth (ENS 24, partiellement **corrigé**, niveau inapproprié, à **debugguer** — oral - 184 lignes) \*\*\*  
*graphes, réduction, langages*  
ensmotspartiels.tex  
**VOIR**
- 67-Relations entre les nombres de sommets et d'arêtes de graphes (MT 0, ex 1, **corrigé** (R) — oral - 112 lignes) \*  
*graphes*  
mtgraph1.tex  
**VOIR**
- 68-couplages (MT 0, ex 1, **corrigé** (R) — oral - 64 lignes) \*  
*graphes*  
mtgraph3.tex  
**VOIR**
- 75-graphes (MT 0, ex 2, **corrigé** — oral - 121 lignes) \*  
*graphes*  
mtgraph2.tex  
**VOIR**
- 76-Graphes bipartis (MT 24, ex 1, **corrigé** (R) — oral - 92 lignes) \*  
*graphes, logique propositionnelle, parcours de graphes, complexité*  
mtgraph5.tex  
**VOIR**
- 84-plus courts chemins dans des graphes (MT 24, ex 2, **corrigé**, retour d'oral (R) — oral - 129 lignes) \*  
*graphes, complexité, programmation dynamique, pseudo-code*  
mtgraph4.tex  
**VOIR**
- 99-Graphes et couleurs (MT 25, ex 2, retour élève remanié, **corrigé** — oral - 128 lignes) \* à \*\*\*  
*graphes, algorithmes probabilistes*  
mtgraph7.tex  
**VOIR**
- 104-Graphes : implémentation en C (MT 25, ex 2, retour élève à compléter/vérifier++, **corrigé** — oral - 98 lignes) \*

*graphes, c*  
 mtgraph11.tex  
**VOIR**

- 106-coloriage de graphe (MT 25, ex 2, retour élève, 3/4 questions manquantes, **corrigé** — oral - 103 lignes) \*  
*graphes, pseudo-code*  
 mtgraph6.tex  
**VOIR**
- 107-CNS pour cycles eulériens (MT 25, ex 2, retour élève, **corrigé** — oral - 71 lignes) \*  
*graphes*  
 mtgraph9.tex  
**VOIR**
- 108-Cycles et Chemins simples (MT 25, ex 2, retour élève, **corrigé** — oral - 64 lignes) \*\*  
*graphes, programmation dynamique*  
 mtgraph10.tex  
**VOIR**
- 110-Bob l'écureuil (X 23, **corrigé** — oral - 141 lignes) \*\*\*  
*algorithmique, graphes*  
 xbobecureuil.tex  
**VOIR**
- 114-Codage par couleur (X 24, **corrigé** — oral - 154 lignes) \*\*\*  
*algorithmique, complexité, graphes, backtracking*  
 xcodagecouleur.tex  
**VOIR**

## Ia

- 6-ID3 (CCINP 0, ex A, **corrigé** — oral - 175 lignes) \*  
*ia*  
 ccinpid3.tex  
**VOIR**
- 33-k plus proches voisins (CCINP 25, ex A, retour élève modifié, figure refaite, **corrigé** — oral - 108 lignes) \*  
*ia*  
 ccinpia2.tex  
**VOIR**
- 79-Classification hiérarchique ascendante (MT 24, ex 1, **corrigé**, le graphique n'est pas l'original (R) — oral - 44 lignes) \*  
*ia, classification hiérarchique ascendante*  
 mtia.tex  
**VOIR**
- 95-k plus proches voisins (MT 25, ex 1, retour élève, **corrigé** — oral - 46 lignes) \*  
*ia*  
 mtia2.tex  
**VOIR**

## Jeux

- 21-chomp (CCINP 24, ex A, **corrigé** (R) — oral - 228 lignes) \*  
*jeux*  
 ccinpchomp.tex  
**VOIR**
- 29-Autour d'un jeu d'alignement (CCINP 25, ex 2, retour élève complété, **corrigé** — oral - 201 lignes) \*  
*ocaml, algorithmique, jeux*  
 ccinpalgo.tex

**VOIR**

- 37-Jeu sur un graphe (CCINP 25, ex A, retour élève, **corrigé** — oral - 120 lignes) \*  
*graphes, jeux*  
ccinpjeu1.tex  
**VOIR**
- 45-Jeu des jetons (ENS 23 MP, **corrigé** — oral - 153 lignes) \*\*\*  
*jeux, graphes*  
ensjeujeton.tex  
**VOIR**
- 103-Automates alternants (MT 25, ex 2, retour élève très buggué largement retravaillé, **corrigé** — oral - 294 lignes) \*\*\*  
*langages, automates, jeux*  
mtautomatesalternants.tex  
**VOIR**

**Langages**

- 3-Langages réguliers (CCINP 0, ex A, **corrigé** (R) — oral - 120 lignes) \*  
*langages, Automates, lemme de l'étoile*  
langccinp3.tex  
**VOIR**
- 17-Langages locaux (CCINP 23, ex B, **corrigé** (R) — oral - 215 lignes) \*  
*langages, ocaml, automates*  
langloccinp1.tex  
**VOIR**
- 22-Grammaires pour des langages de programmation (CCINP 24, ex A, **corrigé, à déboguer** — ds2526 - 90 lignes) \*  
*grammaires, langages*  
langccinp1.tex  
**VOIR**
- 24-Fonctions pour la détection de motifs (CCINP 24, ex B, **corrigé** (R) — oral - 131 lignes) \*  
*algorithmique des textes, langages, ocaml*  
langccinp2.tex  
**VOIR**
- 26-Mots de Dyck (CCINP 24, ex B, **corrigé** — oral - 157 lignes) \*  
*langages, c, complexité*  
ccinpdyck.tex  
**VOIR**
- 32-Mots sur un alphabet muni d'un ordre (CCINP 25, ex A, retour élève modifié, **corrigé** — oral - 171 lignes) \*\*  
*langages, automates*  
ccinplang.tex  
**VOIR**
- 35-Algorithmes du cours sur les automates, complexités (CCINP 25, ex A, retour élève, **corrigé** — oral - 232 lignes) \*\*  
*langages, automates*  
ccinplang3.tex  
**VOIR**
- 38-Mots conjugués (CCINP 25, ex A, retour élève, **corrigé** — oral - 110 lignes) \*  
*langages, automates, lemme de l'étoile*  
ccinplang2.tex  
**VOIR**
- 44-Expressions régulières en OCaml (CCINP 25, ex B, retour élève, **corrigé** — oral - 157 lignes) \*  
*langages, ocaml*  
ccinpregexp.tex  
**VOIR**

- 46-Monoïdes et Langages (ENS 23 MP, **corrigé** — oral - 238 lignes) \*\*\*  
*langages*  
ensmonoideslang.tex  
**VOIR**
  
- 61-Mots partiels et Théorème de Dilworth (ENS 24, partiellement **corrigé**, niveau inapproprié, à déboguer — oral - 184 lignes) \*\*\*  
*graphes, réduction, langages*  
ensmotspartiels.tex  
**VOIR**
  
- 72-entrelacements de mots (MT 0, ex 2, **corrigé** (R) — oral - 105 lignes) \*  
*langages, automates, programmation dynamique*  
mtlang.tex  
**VOIR**
  
- 81-Automates de Büchi (MT 24, ex 2, (modifié) **corrigé** (R) — oral - 117 lignes) \*\*\*  
*langages, automates*  
mtbuchi.tex  
**VOIR**
  
- 83-Automates, langages à saut (MT 24, ex 2, **corrigé**, complété au jugé pour les dernières questions (R) — oral - 93 lignes) \*  
*langages, automates, lemme de l'étoile*  
mtlang2.tex  
**VOIR**
  
- 85-Automates et décidabilité (MT 25, ex 1, retour élève complété, **corrigé** — oral - 87 lignes) \*\*  
*langages, automates, décidabilité*  
mtautom.tex  
**VOIR**
  
- 88-Langages (MT 25, ex 1, retour élève remanié, **corrigé** — oral - 80 lignes) \*\* à \*\*\*  
*langages, automates*  
mtlang4.tex  
**VOIR**
  
- 91-Langages des préfixes (MT 25, ex 1, retour élève à modifier, **corrigé** — oral - 59 lignes) \*  
*langages, grammaires, lemme de l'étoile*  
mtlang6.tex  
**VOIR**
  
- 92-Substitution régulière (MT 25, ex 1, retour élève à reprendre, **corrigé** — oral - 92 lignes) \*\* à \*\*\*\*  
*langages, automates, grammaires*  
mtlang7.tex  
**VOIR**
  
- 93-Langages et facteurs (MT 25, ex 1, retour élève, **corrigé** (R) — oral - 150 lignes) \*  
*langages, automates*  
mtlang3.tex  
**VOIR**
  
- 96-lemme d'Arden (MT 25, ex 1, retour élève, **corrigé** — oral - 94 lignes) \*\*  
*langages*  
mtlang5.tex  
**VOIR**
  
- 100-Langages rationnels et ultime périodicité (MT 25, ex 2, retour élève remanié, **corrigé** — oral - 82 lignes) \*\*\*  
*langages, lemme de l'étoile*  
mtlangper.tex  
**VOIR**
  
- 103-Automates alternants (MT 25, ex 2, retour élève très buggué largement retravaillé, **corrigé** — oral - 294 lignes) \*\*\*  
*langages, automates, jeux*  
mtautomatesalternants.tex  
**VOIR**

- 111-Langages rationnels et lemme de l'étoile (X 23, **corrigé** — oral - 118 lignes) \*\*\*  
*langages, automates, lemme de l'étoile*  
xlangrat.tex  
**VOIR**
- 112-Rationnalité de langages (X 24, **corrigé**, complété au jugé pour les trois dernières questions — oral - 213 lignes) \*\*\*  
*langages, automates, lemme de l'étoile*  
xlangrat2.tex  
**VOIR**
- 115-Chânes et anti-chânes (X 25, **non corrigé** — oral - 44 lignes) \*\*\*  
*langages*  
xchaines.tex  
**VOIR**

### Langages fonctionnels

- 49-Inférence de type (ENS 23, **corrigé** — oral - 279 lignes) \*\*\*  
*algorithmique, langages fonctionnels, pseudo-code*  
ensinferencetypes.tex  
**VOIR**

### Lemme de l'étoile

- 3-Langages réguliers (CCINP 0, ex A, **corrigé** (R) — oral - 120 lignes) \*  
*langages, Automates, lemme de l'étoile*  
langccinp3.tex  
**VOIR**
- 38-Mots conjugués (CCINP 25, ex A, retour élève, **corrigé** — oral - 110 lignes) \*  
*langages, automates, lemme de l'étoile*  
ccinplang2.tex  
**VOIR**
- 83-Automates, langages à saut (MT 24, ex 2, **corrigé**, complété au jugé pour les dernières questions (R) — oral - 93 lignes) \*  
*langages, automates, lemme de l'étoile*  
mtlang2.tex  
**VOIR**
- 91-Langages des préfixes (MT 25, ex 1, retour élève à modifier, **corrigé** — oral - 59 lignes) \*  
*langages, grammaires, lemme de l'étoile*  
mtlang6.tex  
**VOIR**
- 100-Langages rationnels et ultime périodicité (MT 25, ex 2, retour élève remanié, **corrigé** — oral - 82 lignes) \*\*\*  
*langages, lemme de l'étoile*  
mtlangper.tex  
**VOIR**
- 111-Langages rationnels et lemme de l'étoile (X 23, **corrigé** — oral - 118 lignes) \*\*\*  
*langages, automates, lemme de l'étoile*  
xlangrat.tex  
**VOIR**
- 112-Rationnalité de langages (X 24, **corrigé**, complété au jugé pour les trois dernières questions — oral - 213 lignes) \*\*\*  
*langages, automates, lemme de l'étoile*  
xlangrat2.tex  
**VOIR**

## Logique propositionnelle

- 11-SAT (CCINP 0, ex B, **corrigé** — oral - 148 lignes) \*  
*logique propositionnelle, algorithmes probabilistes*  
ccinpsat.tex  
**VOIR**
  
- 16-Formules propositionnelles croissantes (CCINP 23, ex A, **corrigé** — oral - 142 lignes) \*  
*logique propositionnelle*  
logccinp1.tex  
**VOIR**
  
- 25-HORNSAT (CCINP 24, ex B, **corrigé** (R) — oral - 186 lignes) \*  
*logique propositionnelle, complexité, réduction*  
ccinphornsat.tex  
**VOIR**
  
- 27-Complétude de Nor et d'une variante Nor (CCINP 25, ex 1, retour élève modifié, **corrigé** — oral - 150 lignes) \*  
*logique propositionnelle*  
ccinplog.tex  
**VOIR**
  
- 47-Élimination des coupures dans MLL et MALL (ENS 23, **corrigé** partiellement — oral - 385 lignes) \*\*\*\*  
*logique propositionnelle*  
enseliminationcutmll.tex  
**VOIR**
  
- 52-Raisonnements ensemblistes (ENS 23, **corrigé** — oral - 234 lignes) \*\*\*  
*graphes, logique propositionnelle*  
ensensembles.tex  
**VOIR**
  
- 55-Élimination des coupures dans MLL et MALL (ENS 23, réécriture plus raisonnable par N.Carré — oral - 377 lignes) \*\*\*  
*logique propositionnelle*  
mll.tex  
**VOIR**
  
- 69-logique (MT 0, ex 1, **corrigé** (R) — oral - 100 lignes) \*  
*logique propositionnelle*  
mtlog2.tex  
**VOIR**
  
- 76-Graphes bipartis (MT 24, ex 1, **corrigé** (R) — oral - 92 lignes) \*  
*graphes, logique propositionnelle, parcours de graphes, complexité*  
mtgraph5.tex  
**VOIR**
  
- 82-Décidabilité de programmes engendrés par une grammaire (MT 24, ex 2, **corrigé** (R) — oral - 140 lignes) \*  
*grammaires, décidabilité, c, logique propositionnelle*  
mthaltbool.tex  
**VOIR**
  
- 101-Théorème de compacité (MT 25, ex 2, retour élève remanié, **corrigé** — oral - 101 lignes) \*\*  
*logique propositionnelle*  
mtlog4.tex  
**VOIR**
  
- 118-Logique trivaluée (Écrit MP 25, utilisable oral, **corrigé** — oral - 275 lignes) \* à \*\*  
*logique propositionnelle*  
ecrmplog.tex  
**VOIR**

## Ocaml

- 4-Mutex (CCINP 0, ex A, **corrigé** (R) — oral - 109 lignes) \*  
*concurrency, ocaml*  
`concccinp1.tex`  
**VOIR**
- 8-Tableaux autoréférents (CCINP 0, ex B, **corrigé** (R) — oral - 195 lignes) \*  
*algorithmique, backtracking, ocaml*  
`ccinpautoreferent.tex`  
**VOIR**
- 9-Arbres (CCINP 0, ex B, **corrigé** — oral - 247 lignes) \*  
*arbres, ocaml, algorithmique, pseudo-code*  
`ccinparbre2.tex`  
**VOIR**
- 17-Langages locaux (CCINP 23, ex B, **corrigé** (R) — oral - 215 lignes) \*  
*langages, ocaml, automates*  
`langlocccinp1.tex`  
**VOIR**
- 19-chemins simples sans issue (CCINP 23, ex B, **corrigé** (R) — oral - 199 lignes) \*  
*graphes, ocaml, complexité, backtracking*  
`ccinpchemins_simples.tex`  
**VOIR**
- 24-Fonctions pour la détection de motifs (CCINP 24, ex B, **corrigé** (R) — oral - 131 lignes) \*  
*algorithmique des textes, langages, ocaml*  
`langccinp2.tex`  
**VOIR**
- 29-Autour d'un jeu d'alignement (CCINP 25, ex 2, retour élève complété, **corrigé** — oral - 201 lignes) \*  
*ocaml, algorithmique, jeux*  
`ccinpalgo.tex`  
**VOIR**
- 30-Graphes bipartis (CCINP 25, ex 2, retour élève complété, **corrigé** — oral - 136 lignes) \*  
*graphes, ocaml*  
`ccinpgraphe3.tex`  
**VOIR**
- 39-tas min (CCINP 25, ex A, retour élève, rallongé, **corrigé** — oral - 92 lignes) \*  
*algorithmique, Structures de données, Ocaml*  
`ccinptas.tex`  
**VOIR**
- 42-Rendu de monnaie (CCINP 25, ex B, retour élève complété, **corrigé** — oral - 126 lignes) \*  
*algorithmique, ocaml*  
`ccinpalgo2.tex`  
**VOIR**
- 43-Dénombrément du nombre de façons d'obtenir un score avec des dés. (CCINP 25, ex B, retour élève, **corrigé** — oral - 150 lignes) \*  
*algorithmique, ocaml, programmation dynamique*  
`ccinpalgo4.tex`  
**VOIR**
- 44-Expressions régulières en OCaml (CCINP 25, ex B, retour élève, **corrigé** — oral - 157 lignes) \*  
*langages, ocaml*  
`ccinpregexp.tex`  
**VOIR**
- 54-Structures pliables et traversables (ENS 23, partiellement **corrigé** — oral - 314 lignes) \*\*\*  
*ocaml, programmation fonctionnelle*  
`ensstructurespliables.tex`

**VOIR**

- 56-Composition Monadique (ENS 24, **corrigé** très partiellement — oral - 191 lignes) \*\*\*  
*ocaml, programmation fonctionnelle*  
`enscompositionmonadique.tex`  
**VOIR**
- 58-Filtrage par motif en OCaml (ENS 24, **corrigé** — oral - 323 lignes) \*\*\*  
*ocaml*  
`ensfiltrage0caml.tex`  
**VOIR**
- 60-Résolution d'inéquations linéaires (ENS 24, **corrigé** — oral - 195 lignes) \*\*\*  
*ocaml, algorithmique, complexité*  
`ensresolutioninequationslineaires.tex`  
**VOIR**
- 62-Terminaison de lambda ref (ENS 24, partiellement **corrigé**, niveau surréaliste — oral - 238 lignes) \*\*\*  
*ocaml*  
`ensterminaison_lambda_ref.tex`  
**VOIR**

**Parcours de graphes**

- 76-Graphes bipartis (MT 24, ex 1, **corrigé** (R) — oral - 92 lignes) \*  
*graphes, logique propositionnelle, parcours de graphes, complexité*  
`mtgraph5.tex`  
**VOIR**

**Preuves**

- 90-Points fixes dans des tableaux (MT 25, ex 1, retour élève retravaillé, **corrigé** — oral - 79 lignes) \*  
*algorithmique, complexité, preuves, pseudo-code*  
`mttab1.tex`  
**VOIR**
- 113-Algorithmes du tri lent (X 24, **corrigé** — oral - 147 lignes) \*\*\*  
*algorithmique, tri, complexité, preuves*  
`xtrilent.tex`  
**VOIR**
- 117-Vis et écrous (X 25, récupéré par élève, **non corrigé** — oral - 81 lignes) \*\*\*  
*algorithmique, tri, complexité, preuves, algorithmes probabilistes*  
`xecroux.tex`  
**VOIR**

**Programmation dynamique**

- 18-Programmation dynamique pour la récolte de fleurs (CCINP 23, ex B, **corrigé** (R) — oral - 204 lignes) \*  
*programmation dynamique, c*  
`ccinprecoltefleurs.tex`  
**VOIR**
- 43-Dénombrer le nombre de façons d'obtenir un score avec des dés. (CCINP 25, ex B, retour élève, **corrigé** — oral - 150 lignes) \*  
*algorithmique, ocaml, programmation dynamique*  
`ccinpalgo4.tex`  
**VOIR**
- 71-Arbres Binaires de Recherche (MT 0, ex 2, **corrigé** (R) — oral - 167 lignes) \*

*arbres, complexité, programmation dynamique*

mtarbres1.tex

**VOIR**

- 72-entrelacements de mots (MT 0, ex 2, **corrigé** (R) — oral - 105 lignes) \*  
*langages, automates, programmation dynamique*  
mtlang.tex  
**VOIR**
- 84-plus courts chemins dans des graphes (MT 24, ex 2, **corrigé**, retour d'oral (R) — oral - 129 lignes) \*  
*graphes, complexité, programmation dynamique, pseudo-code*  
mtgraph4.tex  
**VOIR**
- 98-Représentation fusionnée de deux mots (MT 25, ex 2, retour élève remanié, **corrigé** (R) — oral - 116 lignes) \*\*  
*algorithmique des textes, programmation dynamique*  
mttxt1.tex  
**VOIR**
- 102-Programmation dynamique (MT 25, ex 2, retour élève retravaillé, **corrigé** — oral - 131 lignes) \*\*\*  
*programmation dynamique, réduction*  
mtdyn.tex  
**VOIR**
- 108-Cycles et Chemins simples (MT 25, ex 2, retour élève, **corrigé** — oral - 64 lignes) \*\*  
*graphes, programmation dynamique*  
mtgraph10.tex  
**VOIR**

### Programmation fonctionnelle

- 54-Structures pliables et traversables (ENS 23, partiellement **corrigé** — oral - 314 lignes) \*\*\*  
*ocaml, programmation fonctionnelle*  
ensstructurespliables.tex  
**VOIR**
- 56-Composition Monadique (ENS 24, **corrigé** très partiellement — oral - 191 lignes) \*\*\*  
*ocaml, programmation fonctionnelle*  
enscompositionmonadique.tex  
**VOIR**

### Pseudo-code

- 9-Arbres (CCINP 0, ex B, **corrigé** — oral - 247 lignes) \*  
*arbres, ocaml, algorithmique, pseudo-code*  
ccinparbre2.tex  
**VOIR**
- 49-Inférence de type (ENS 23, **corrigé** — oral - 279 lignes) \*\*\*  
*algorithmique, langages fonctionnels, pseudo-code*  
ensinferencetypes.tex  
**VOIR**
- 80-Tas binaires (MT 24, ex 1, **corrigé** — oral - 137 lignes) \*  
*arbres, tri, structures de données, complexité, pseudo-code*  
mttas2.tex  
**VOIR**
- 84-plus courts chemins dans des graphes (MT 24, ex 2, **corrigé**, retour d'oral (R) — oral - 129 lignes) \*  
*graphes, complexité, programmation dynamique, pseudo-code*  
mtgraph4.tex  
**VOIR**

- 90-Points fixes dans des tableaux (MT 25, ex 1, retour élève retravaillé, **corrigé** — oral - 79 lignes) \*  
*algorithmique, complexité, preuves, pseudo-code*  
mttab1.tex  
**VOIR**
- 106-coloriage de graphe (MT 25, ex 2, retour élève, 3/4 questions manquantes, **corrigé** — oral - 103 lignes) \*  
*graphes, pseudo-code*  
mtgraph6.tex  
**VOIR**

## Représentation des nombres

- 20-calculs avec les flottants (CCINP 23, ex B, **corrigé** — oral - 206 lignes) \*  
*représentation des nombres, c*  
ccinpflottants.tex  
**VOIR**

## Réduction

- 13-Activation de processus (CCINP 23, ex A, **corrigé** (R) — oral - 153 lignes) \*  
*complexité, réduction*  
procesCCINP23.tex  
**VOIR**
- 25-HORNSAT (CCINP 24, ex B, **corrigé** (R) — oral - 186 lignes) \*  
*logique propositionnelle, complexité, réduction*  
ccinphornsat.tex  
**VOIR**
- 57-Déduction de messages (ENS 24, **corrigé** — oral - 232 lignes) \*\*\*  
*déduction naturelle, réduction*  
ensdeductiondemessages.tex  
**VOIR**
- 61-Mots partiels et Théorème de Dilworth (ENS 24, partiellement **corrigé**, niveau inapproprié, à **debugger** — oral - 184 lignes) \*\*\*  
*graphes, réduction, langages*  
ensmotspartiels.tex  
**VOIR**
- 97- (MT 25, ex 2, retour élève remanié, **corrigé** — oral - 117 lignes) \*  
*complexité, algorithmes d'approximation, réduction*  
mtapprox2.tex  
**VOIR**
- 102-Programmation dynamique (MT 25, ex 2, retour élève retravaillé, **corrigé** — oral - 131 lignes) \*\*\*  
*programmation dynamique, réduction*  
mtdyn.tex  
**VOIR**

## Sql

- 5-SQL (CCINP 0, ex A, **corrigé** (R) — oral - 95 lignes) \*  
*sql*  
sqlccinp1.tex  
**VOIR**
- 70-bdd pour livraison (MT 0, ex 1, **corrigé** — oral - 78 lignes) \*  
*sql*

mtsql1.tex

**VOIR**

### Structures de données

- 39-tas min (CCINP 25, ex A, retour élève, rallongé, **corrigé** — oral - 92 lignes) \*  
*algorithmique, Structures de données, Ocaml*  
 ccinptas.tex  
**VOIR**
- 77-Tas binaires (MT 24, ex 1, **corrigé** (R) — oral - 129 lignes) \*  
*arbres, tri, structures de données, complexité*  
 mttas.tex  
**VOIR**
- 80-Tas binaires (MT 24, ex 1, **corrigé** — oral - 137 lignes) \*  
*arbres, tri, structures de données, complexité, pseudo-code*  
 mttas2.tex  
**VOIR**
- 89-Piles et files (MT 25, ex 1, retour élève retravaillé, **corrigé** — oral - 67 lignes) \*  
*structures de données, complexité*  
 mtstruct1.tex  
**VOIR**
- 109-Algorithmes Union-Find (X 23, **corrigé** — oral - 151 lignes) \*\*\*  
*algorithmique, complexité, structures de données*  
 xunionfind.tex  
**VOIR**

### Tri

- 77-Tas binaires (MT 24, ex 1, **corrigé** (R) — oral - 129 lignes) \*  
*arbres, tri, structures de données, complexité*  
 mttas.tex  
**VOIR**
- 80-Tas binaires (MT 24, ex 1, **corrigé** — oral - 137 lignes) \*  
*arbres, tri, structures de données, complexité, pseudo-code*  
 mttas2.tex  
**VOIR**
- 113-Algorithmes du tri lent (X 24, **corrigé** — oral - 147 lignes) \*\*\*  
*algorithmique, tri, complexité, preuves*  
 xtrilent.tex  
**VOIR**
- 117-Vis et écrous (X 25, récupéré par élève, **non corrigé** — oral - 81 lignes) \*\*\*  
*algorithmique, tri, complexité, preuves, algorithmes probabilistes*  
 xecroux.tex  
**VOIR**

## Chapitre 2

# (CCINP) Arbres Binaires de Recherche \*

## (CCINP 0, ex A, corrigé (R) — oral - 74 lignes)

\*

Arbres, Complexité,  
sources : `arbreCCINPA2.tex`

Dans cet exercice, on autorise les doublons dans un arbre binaire de recherche et pour le cas d'égalité on choisira le sous-arbre gauche. On ne cherchera pas à équilibrer les arbres.

1. Rappeler la définition d'un arbre binaire de recherche.
2. Insérer successivement *e* et *a* dans un arbre binaire de recherche initialement vide toutes les lettres du mot *bacddabdbae* en utilisant l'ordre alphabétique sur les lettres. Quelle est la hauteur de l'arbre ainsi obtenu ?
3. Montrer que le parcours en profondeur infixe d'un arbre binaire de recherche de lettres est un mot dont les lettres sont rangées dans l'ordre croissant. On pourra procéder par induction structurelle.
4. Proposer un algorithme qui permet de compter le nombre d'occurrences d'une lettre dans un arbre binaire de recherche de lettres. Quelle est sa complexité ?
5. On souhaite supprimer *une* occurrence d'une lettre donnée dans un arbre binaire de recherche de lettres. Expliquer le principe de l'algorithme permettant de résoudre ce problème et le mettre en oeuvre sur l'arbre obtenu à la question 2 en supprimant successivement une occurrence des lettres *e*, *b*, *c* et *d*. Quelle est sa complexité ?



# Chapitre 3

## (CCINP) Dédution naturelle \* (CCINP 0, ex A, corrigé (R) — oral - 160 lignes)

\*

Dédution naturelle,  
sources : `dednatCCINPA.tex`

Rappelons les règles de la déduction naturelle suivantes, où  $A$  et  $B$  sont des formules logiques et  $\Gamma$  un ensemble de formules logiques quelconque :

$$\frac{}{\Gamma, A \vdash A} \text{AX} \quad \frac{\Gamma \vdash \perp}{\Gamma \vdash A} \perp_e \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \rightarrow_i \quad \frac{\Gamma \vdash A \quad \Gamma \vdash A \rightarrow B}{\Gamma \vdash B} \rightarrow_e \quad \frac{\Gamma, A \vdash \perp}{\Gamma \vdash \neg A} \neg_i \quad \frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \perp} \neg_e$$

1. Montrer que le séquent  $\vdash \neg A \rightarrow (A \rightarrow \perp)$  est dérivable, en explicitant un arbre de preuve.
2. Montrer que le séquent  $\vdash (A \rightarrow \perp) \rightarrow \neg A$  est dérivable, en explicitant un arbre de preuve.
3. Donner une règle correspondant à l'introduction du symbole  $\wedge$  ainsi que deux règles correspondant à l'élimination du symbole  $\wedge$ . Montrer que le séquent  $\vdash (\neg A \rightarrow (A \rightarrow \perp)) \wedge ((A \rightarrow \perp) \rightarrow \neg A)$  est dérivable.
4. On considère la formule  $P = ((A \rightarrow B) \rightarrow A) \rightarrow A$  appelée *loi de Peirce*. Montrer que  $\models P$ , c'est-à-dire que  $P$  est une tautologie.
5. Pour montrer que le séquent  $\vdash P$  est dérivable, il est nécessaire d'utiliser la règle d'absurdité classique  $\perp_c$  (ou une règle équivalente), ce que l'on fait ci-dessous (il n'y aura pas besoin de réutiliser cette règle). Terminer la preuve du séquent  $\vdash P$ , dans laquelle on pose  $\Gamma = \{(A \rightarrow B) \rightarrow A, \neg A\}$  :

$$\frac{\frac{\frac{?}{\Gamma \vdash A} \quad ? \quad \frac{}{\Gamma \vdash \neg A} \text{AX}}{\Gamma \vdash \perp} \neg_i}{(A \rightarrow B) \rightarrow A \vdash A} \perp_c}{\vdash ((A \rightarrow B) \rightarrow A) \rightarrow A} \rightarrow_i$$



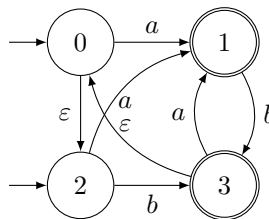
# Chapitre 4

## (CCINP) Langages réguliers \* (CCINP 0, ex A, corrigé (R) — oral - 120 lignes)

\*

Langages, Automates, Lemme de l'étoile,  
sources : langccinp3.tex

1. Rappeler la définition d'un langage régulier.
2. Les langages suivants sont-ils réguliers ? Justifier.
  - (a)  $L_1 = \{a^n b a^m \mid n, m \in \mathbb{N}\}$
  - (b)  $L_2 = \{a^n b a^m \mid n, m \in \mathbb{N}, n \leq m\}$
  - (c)  $L_3 = \{a^n b a^m \mid n, m \in \mathbb{N}, n > m\}$
  - (d)  $L_4 = \{a^n b a^m \mid n, m \in \mathbb{N}, n + m \equiv 0 \pmod{2}\}$
3. On considère l'automate non déterministe suivant :



- (a) Déterminiser cet automate.
- (b) Construire une expression régulière dénotant le langage reconnu par cet automate.
- (c) Décrire simplement avec des mots le langage reconnu par cet automate.



## Chapitre 5

# (CCINP) Mutex \* (CCINP 0, ex A, corrigé (R) — oral - 109 lignes)

\*

Concurrence, Ocaml,  
sources : concccinp1.tex

On considère le programme suivant, ici en OCaml, dans lequel  $n$  fils d'exécution incrémentent tous un même compteur partagé.

```
(* Nombre de fils d'exécution *)
let n = 100
(* Un même compteur partagé *)
let compteur = ref 0
(* Chaque fil d'exécution de numéro i va incrémenter le compteur *)
let fi = compteur := ! compteur + 1
(* Création de n fils exécutant f associant à chaque fil son numéro *)
let threads = Array.init n (fun i → Thread.create fi)
(* Attente de la fin des n fils d'exécution *)
let () = Array.iter (fun t → Thread.join t) threads
```

On rappelle que l'on dispose en OCaml des trois fonctions `Mutex.create : unit -> Mutex.t` pour la création d'un verrou, `Mutex.lock : Mutex.t -> unit` pour le verrouillage et `Mutex.unlock : Mutex.t -> unit` pour le déverrouillage, du module `Mutex` pour manipuler des verrous.

1. Quelles sont les valeurs possibles que peut prendre le compteur à la fin du programme.
2. Identifier la section critique et indiquer comment et à quel endroit ajouter des verrous pour garantir que la valeur du compteur à la fin du programme soit  $n$  de manière certaine.

Dans la suite de l'exercice, on suppose que l'on ne dispose pas d'une implémentation des verrous. On se limite au cas de deux fils d'exécution, numérotés 0 et 1. Nous cherchons à garantir deux propriétés :

- *Exclusion mutuelle* : un seul fil d'exécution à la fois peut se trouver dans la section critique ;
- *Absence de famine* : tout fil d'exécution qui cherche à entrer dans la section critique pourra le faire à un moment.

On utilise pour cela un tableau `veut_entrer` qui indique pour chaque fil d'exécution s'il souhaite entrer en section critique ainsi qu'une variable `tour` qui indique quel fil d'exécution peut effectivement entrer dans la section critique. On propose ci-dessous deux versions modifiées `f_a` et `f_b` de la fonction `f`, l'objectif étant de pouvoir exécuter `f_a 0` et `f_a 1` de manière concurrente, et de même pour `f_b`.

```
let veut_entrer = [| false; false |]
let tour = ref 0
let f_a i =
  let autre = 1 - i in
  veut_entrer.(i) <- true;
  while veut_entrer.(autre) do () done;
  (* section critique *)
  veut_entrer.(i) <- false
let f_b i =
  let autre = 1 - i in
  veut_entrer.(i) <- true;
  tour := i;
  while veut_entrer.(autre) && ! tour = autre do () done;
  (* section critique *)
  veut_entrer.(i) <- false
```

3. Expliquer pourquoi aucune de ces deux versions ne convient, en indiquant la propriété qui est violée.
4. Proposer une version f\_c qui permet de garantir les deux propriétés.
5. Connaissez-vous un algorithme permettant de généraliser à  $n$  fils d'exécution ? Rappeler très succinctement son principe.

## Chapitre 6

# (CCINP) SQL \* (CCINP 0, ex A, corrigé (R) — oral - 95 lignes)

\*

Sql,  
sources : sqlccinp1.tex

On considère le schéma de base de données suivant, qui décrit un ensemble de fabricants de matériel informatique, les matériels qu'ils vendent, leurs clients et ce qu'achètent leurs clients. Les attributs des clés primaires des six premières relations sont soulignés.

```
Production(NomFabricant, Modele)
Ordinateur(Modele, Frequence, Ram, Dd, Prix)
Portable(Modele, Frequence, Ram, Dd, Ecran, Prix)
Imprimante(Modele, Couleur, Type, Prix)
Fabricant(Nom, Adresse, NomPatron)
Client(Num, Nom, Prenom)
Achat(NumClient, NomFabricant, Modele, Quantite)
```

Chaque client possède un numéro unique connu de tous les fabricants. La relation **Production** donne pour chaque fabricant l'ensemble des modèles fabriqués par ce fabricant. Deux fabricants différents peuvent proposer le même matériel. La relation **Ordinateur** donne pour chaque modèle d'ordinateur la vitesse du processeur (en Hz), les tailles de la Ram et du disque dur (en Go) et le prix de l'ordinateur (en €). La relation **Portable**, en plus des attributs précédents, comporte la taille de l'écran (en pouces). La relation **Imprimante** indique pour chaque modèle d'imprimante si elle imprime en couleur (vrai/faux), le type d'impression (laser ou jet d'encre) et le prix (en €). La relation **Fabricant** stocke les nom et adresse de chaque fabricant, ainsi que le nom de son patron. La relation **Client** stocke les noms et prénoms de tous les clients de tous les fabricants. Enfin, la relation **Achat** regroupe les quadruplets (client  $c$ , fabriquant  $f$ , modèle  $m$ , quantité  $q$ ) tels que le client de numéro  $c$  a acheté  $q$  fois le modèle  $m$  au fabricant  $f$ . On suppose que l'attribut **Quantite** est toujours strictement positif.

1. Proposer une clé primaire pour la relation **Achat** et indiquer ses conséquences en terme de modélisation.
2. Identifier l'ensemble des clés étrangères éventuelles de chaque table.
3. Donner en SQL des requêtes répondant aux questions suivantes :
  - (a) Quels sont les numéros des modèles des matériels (ordinateur, portable ou imprimante) fabriqués par l'entreprise du nom de Durand ?
  - (b) Quels sont les noms et adresses des fabricants produisant des portables dont le disque dur a une capacité d'au moins 500 Go ?
  - (c) Quels sont les noms des fabricants qui produisent au moins 10 modèles différents d'imprimantes ?
  - (d) Quels sont les numéros des clients n'ayant acheté aucune imprimante ?



# Chapitre 7

## (CCINP) ID3 \* (CCINP 0, ex A, corrigé — oral - 175 lignes)

\*

Ia,  
sources : ccinpid3.tex

### ID3 :

On considère un problème d'apprentissage supervisé à deux classes dont les données d'apprentissage sont de la forme  $Z = (x_i, y_i)_{i \in \llbracket 1, n \rrbracket}$  avec  $\forall i \in \llbracket 1, n \rrbracket$ ,  $x_i \in \mathbb{B}^d$ ,  $y_i \in \{+, -\}$ , où  $d$  est le nombre d'attributs binaires d'un exemple et  $\mathbb{B} = \{\text{YES}, \text{NO}\}$ , les valeurs possibles pour les attributs.

Par exemple, le tableau ci-dessous est un échantillon  $Z$  de données relatif aux infections à la COVID 19, extrait de IJCRD 2019. La première colonne indique l'identifiant d'un exemple  $x$  (qui comporte trois attributs  $F$ ,  $T$  et  $R$ ) et la dernière colonne son étiquette  $y$  (ici  $I$ ).

ID	Fièvre (F)	Toux (T)	Problèmes respiratoires (R)	Infecté (I)
1	NO	NO	NO	-
2	YES	YES	YES	+
3	YES	YES	NO	-
4	YES	NO	YES	+
5	YES	YES	YES	+
6	NO	YES	NO	-
7	YES	NO	YES	+
8	YES	NO	YES	+
9	NO	YES	YES	+
10	YES	YES	NO	+
11	NO	YES	NO	-
12	NO	YES	YES	-
13	NO	YES	YES	-
14	YES	YES	NO	-

1. Rappeler le principe de l'apprentissage supervisé.
2. Dessiner l'arbre de décision obtenu en considérant successivement et dans l'ordre les attributs  $F$ ,  $T$  et  $R$ . Commenter.  
*On rappelle qu'un arbre de décisions est un arbre binaire dont les noeuds internes sont étiquetés par les attributs et les feuilles par  $\{+, -\}$ . Les fils gauches correspondent à une réponse NO et les fils droits à la réponse YES.*

L'entropie d'un ensemble  $S$  d'exemple est définie par :

$$H(S) = -\frac{n_+}{n} \log_2 \left( \frac{n_+}{n} \right) - \frac{n_-}{n} \log_2 \left( \frac{n_-}{n} \right)$$

où  $n_+$ ,  $n_-$  et  $n$  désignent respectivement le nombre d'éléments de  $S$  dont l'étiquette est  $+$ , le nombre d'éléments de  $S$  dont l'étiquette est  $-$  et enfin le nombre total d'éléments de  $S$ . Dans le cas où  $k = 0$ , on prend la convention que  $k \log_2 k = 0$ . Par exemple l'entropie de l'ensemble de toutes les données  $Z$  ci-dessus est 1.00.

Étant donné un attribut  $A$ , on définit le gain de  $A$  par rapport à  $S$  par :

$$G(S, A) = H(S) - \frac{n_{A=YES}}{n} H(S_{A=YES}) - \frac{n_{A=NO}}{n} H(S_{A=NO})$$

où  $S_{A=YES}$  désigne le sous-ensemble des éléments de  $S$  dont l'attribut  $A$  est YES et  $n_{A=YES}$  désigne son cardinal, de même pour NO et  $n$  désigne toujours le cardinal de  $S$ . Par exemple  $G(Z, F) = 0.26$ ,  $G(Z, T) = 0.07$  et  $G(Z, R) = 0.26$  (les valeurs données sont approchées au centième).

Si l'on considère le sous-ensemble  $Z_{F=YES}$  des individus qui ont eu de la fièvre, et en supprimant l'attribut  $F$ , on obtient le sous-tableau ci-dessous. Le gain d'information de l'attribut  $T$  est alors  $G(Z_{F=YES}, T) = 0.20$ .

ID	Toux (T)	Problèmes respiratoires (R)	Infecté (I)
2	YES	YES	+
3	YES	NO	-
4	NO	YES	+
5	YES	YES	+
7	NO	YES	+
8	NO	YES	+
10	YES	NO	+
14	YES	NO	-

3. Calculer le gain d'entropie  $G(Z_{F=YES}, R)$  de l'attribut problèmes respiratoires pour le sous-ensemble des individus qui ont eu de la fièvre.

L'algorithme *Iterative Dichotomiser 3 (ID3)* (Algorithme 1) peut être utilisé pour construire un arbre de décision. Pour l'appel initial, il suffit de prendre l'ensemble de tous les exemples pour  $S_p$  et pour  $S$ , et l'ensemble de tous les attributs pour  $D$ .

```

1 Entrées:  $S_p$  sous-ensemble des exemples du noeud parent,  $S$  sous-ensemble des ←
   exemples à considérer,  $D$  sous-ensemble des attributs à considérer}
2 Sortie: Un arbre de décision.
3
4 Fonction ID3( $(S_p, S, D)$ );
5   Si l'ensemble des exemples  $S$  est vide
6     Renvoyer ...
7
8   Si l'ensemble  $A$  des attributs est vide
9     Renvoyer ...
10
11  Si tous les exemples de  $S$  ont une même étiquette  $y$ 
12    Renvoyer ...
13
14  Sinon :
15    Soit  $A \in D$  l'attribut de plus grand gain  $G(S, A) \setminus$ ;
16    Construire l'arbre de racine  $A$  et de sous-arbre gauche \texttt{ID3}←
      ( $S, S_{A=NO}, D \setminus \{A\}$ ) et de sous-arbre droit \texttt{ID3}( $S, S_{A=YES}, D \setminus \{A\}$ )

```

4. Indiquer comment compléter l'algorithme 1.

## Chapitre 8

# (CCINP) Tableaux \* (CCINP 0, ex B, corrigé (R) — oral - 151 lignes)

\*

Algorithmique, C,  
sources : `ccinptableau.tex`

L'exercice suivant est à traiter dans le langage *C*.

Dans tout l'exercice, on ne considère que des tableaux d'entiers de longueur  $n \geq 0$ .

Un squelette de programme *C* vous est donné, avec un jeu de tests qu'il ne faut pas modifier. Vous pouvez bien sûr ajouter vos propres tests à part.

1. Écrire une fonction de prototype `int nb_occurrences (int n, int* tab, int x)` qui renvoie le nombre d'occurrences de l'élément *x* dans le tableau *tab* de longueur *n*. Quelle est la complexité de cette fonction ?

Dans toute la suite, on suppose que les tableaux sont triés dans l'ordre croissant. On va chercher à écrire une version plus efficace de la fonction ci-dessus qui exploite cette propriété. On cherche tout d'abord à écrire une fonction

`int une_occurrence (int n, int* tab, int x)`

qui permet de renvoyer un indice d'une occurrence quelconque de l'élément *x* s'il est présent dans le tableau et -1 sinon. On procède par dichotomie.

2. Compléter le code de la fonction `int une_occurrence (int n, int* tab, int x)` qui vous est donnée dans le squelette. Cette fonction doit avoir une complexité en  $O(\log n)$ .
3. Écrire une fonction `int premiere_occurrence (int n, int* tab, int x)` qui renvoie l'indice de la première occurrence d'un élément *x* dans un tableau *tab* de longueur *n* si cet élément est présent et -1 sinon. Cette fonction doit avoir une complexité en  $O(\log n)$ .
4. Écrire une fonction `int nombre_occurrences (int n, int* tab, int x)` qui renvoie le nombre d'occurrences de l'élément *x* dans le tableau *tab* de longueur *n*. Cette fonction devra avoir une complexité en  $O(\log n)$ .
5. Justifier que la fonction `une_occurrence` termine et est correcte. On donnera un variant et un invariant de boucle que l'on justifiera.
6. Montrer que la complexité de la fonction `une_occurrence` est bien en  $O(\log n)$ .



## Chapitre 9

# (CCINP) Tableaux autoréférents \* (CCINP 0, ex B, corrigé (R) — oral - 195 lignes)

\*

Algorithmique, Backtracking, Ocaml,  
sources : `ccinpautoreferent.tex`

L'exercice suivant est à traiter dans le langage *OCaml*.

1. Écrire une fonction `somme : int array → int → int` telle que l'appel `somme t i` calcule la somme partielle

$$\sum_{k=0}^i t.(k) \text{ des valeurs du tableau } t \text{ entre les indices } 0 \text{ et } i \text{ inclus.}$$

Un tableau  $t$  de  $n > 0$  éléments de  $\llbracket 0, n - 1 \rrbracket$  est dit autoréférent si pour tout indice  $0 \leq i < n$ ,  $t.(i)$  est exactement le nombre d'occurrences de  $i$  dans  $t$ , c'est-à-dire que

$$\forall i \in \llbracket 0, n - 1 \rrbracket, \quad t.(i) = \text{card}(\{k \in \llbracket 0, n - 1 \rrbracket \mid t.(k) = i\})$$

Ainsi, par exemple, pour  $n = 4$ , le tableau suivant est autoréférent :

$i$	0	1	2	3
$t.(i)$	1	2	1	0

En effet, la valeur 0 existe en une occurrence, la valeur 1 en deux occurrences, la valeur 2 en une occurrence et la valeur 3 n'apparaît pas dans  $t$ .

2. Justifier rapidement qu'il n'existe aucun tableau autoréférent pour  $n \in \llbracket 1; 3 \rrbracket$  et trouver un autre tableau autoréférent pour  $n = 4$ .
3. Écrire une fonction `est_auto : int array → bool` qui vérifie si un tableau de taille  $n > 0$  est autoréférent. On attend une complexité en  $O(n)$ .

On propose d'utiliser une méthode de retour sur trace (backtracking) pour trouver tous les tableaux autoréférents pour un  $n > 0$  donné. Une fonction `gen_auto` qui affiche tous les tableaux autoréférents pour une taille donnée vous est proposée. Cette version ne fonctionne cependant que pour de toutes petites valeurs de  $n$  (instantané pour  $n = 5$ , un peu long pour  $n = 8$ , sans espoir pour  $n = 15$ ). On pourra vérifier qu'il existe exactement deux tableaux autoréférents pour  $n = 4$ , un seul pour  $n \in \{5, 7, 8\}$  et aucun pour  $n = 6$ .

Pour accélérer la recherche, il faut élaguer l'arbre (repérer le plus rapidement possible qu'on se trouve dans une branche ne pouvant pas donner de solution).

4. Que peut-on dire de la somme des éléments d'un tableau autoréférent ? En déduire une stratégie d'élagage pour accélérer la recherche.

*Indication* : utiliser la fonction `somme` de la première question pour interrompre par un échec l'exploration lorsque `somme t i` dépasse déjà la valeur maximale possible.

5. Que peut-on dire si juste après avoir affecté la case  $t.(i)$ , il y a déjà strictement plus d'occurrences d'une valeur  $0 \leq k \leq i$  que la valeur de  $t.(k)$  ? En déduire une stratégie d'élagage supplémentaire et la mettre en œuvre. Combien de temps faut-il pour résoudre le problème pour  $n = 15$  ?
6. Après avoir affecté la case  $t.(i)$ , combien de cases reste-t-il à remplir ? Combien de ces cases seront complétées par une valeur non nulle ? À quelle condition est-on alors certain que la somme dépassera la valeur maximale possible à la fin ? En déduire une stratégie d'élagage supplémentaire et la mettre en œuvre. Combien de temps faut-il pour résoudre le problème pour  $n = 30$  ?
7. Montrer qu'il existe un tableau autoréférent pour tout  $n \geq 7$ . On pourra conjecturer la forme de ce tableau en testant empiriquement pour différentes valeurs de  $n \geq 7$ . On ne demande pas de montrer que cette solution est unique.



# Chapitre 10

## (CCINP) Arbres \* (CCINP 0, ex B, corrigé — oral - 247 lignes)

\*

Arbres, Ocaml, Algorithmique, Pseudo-code,  
sources : `ccinparbre2.tex`

L'exercice suivant est à traiter dans le langage *OCaml*.

Dans cet exercice on s'interdit d'utiliser les traits impératifs du langage *OCaml* (références, tableaux, champs mutables, etc.).

On représente en *OCaml* une permutation  $\sigma$  de  $\llbracket 0, n-1 \rrbracket$  par la liste d'entier  $[\sigma_0; \sigma_1; \dots; \sigma_{n-1}]$ . Un arbre binaire étiqueté est soit un arbre vide, soit un nœud formé d'un sous-arbre gauche, d'une étiquette et d'un sous-arbre droit :

```
type arbre =  
  | V  
  | N of arbre * int * arbre
```

On représente un arbre binaire non étiqueté par un arbre binaire étiqueté en ignorant simplement les étiquettes. On étiquette un arbre binaire non étiqueté à  $n$  nœuds par  $\llbracket 0, n-1 \rrbracket$  en suivant l'ordre infixé de son parcours en profondeur. La permutation associée à cet arbre est donnée par le parcours en profondeur par ordre préfixe. La figure 1 propose un exemple (on ne dessine pas les arbres vides).

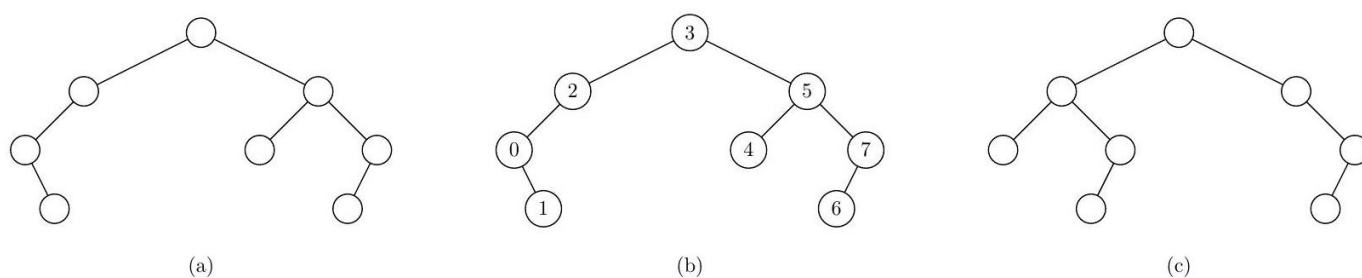


Figure 1 - (a) un arbre binaire non étiqueté; (b) son étiquetage en suivant un ordre infixé, la permutation associée est  $[3; 2; 0; 1; 5; 4; 7; 6]$ ; (c) un autre arbre binaire non étiqueté.

Un fichier source `ccinparbre.ml` qui implémente ces exemples vous est fourni.

1. Étiqueter l'arbre (c) de la figure 1 et donner la permutation associée.
2. Écrire une fonction `parcours_prefixe : arbre -> int list` qui renvoie la liste des étiquettes d'un arbre dans l'ordre préfixe de son parcours en profondeur. On pourra utiliser l'opérateur `@` et on ne cherchera pas nécessairement à proposer une solution linéaire en la taille de l'arbre.
3. Écrire une fonction `etiquette : arbre -> arbre` qui prend en paramètre un arbre dont on ignore les étiquettes et qui renvoie un arbre identique mais étiqueté par les entiers de  $\llbracket 0, n-1 \rrbracket$  en suivant l'ordre infixé d'un parcours en profondeur.

*Indication* : on pourra utiliser une fonction auxiliaire de type `arbre -> int -> arbre * int` qui prend en paramètres un arbre et la prochaine étiquette à mettre et qui renvoie le couple formé par l'arbre étiqueté et la nouvelle prochaine étiquette à mettre.

Une permutation  $\sigma$  de  $\llbracket 0, n-1 \rrbracket$  est triable avec une pile s'il est possible de trier la liste  $[\sigma_0; \sigma_1; \dots; \sigma_{n-1}]$  en utilisant uniquement une structure de pile comme espace de stockage interne. On considère l'algorithme suivant, énoncé ici dans un style impératif :

Initialiser une pile vide ;

Pour chaque élément en entrée :

- \* Tant que l'élément est plus grand que le sommet de la pile, dépiler le  $\leftrightarrow$  sommet de la pile vers la sortie ;
- \* Empiler l'élément en entrée dans la pile ;

Dépiler tous les éléments restant dans la pile vers la sortie.

Par exemple, pour la permutation  $[3; 2; 0; 1; 5; 4; 7; 6]$ , on empile 3, 2, 0, on dépile 0, on empile 1, on dépile 1, 2, 3, on empile 5, 4, on dépile 4, 5, on empile 7, 6, on dépile 6, 7. On obtient la liste triée  $[7; 6; 5; 4; 3; 2; 1; 0]$  en supposant avoir ajouté en sortie les éléments dans une liste. On admet qu'une permutation est triable par pile si et seulement cet algorithme permet de la trier correctement.

4. Dérouler l'exécution de cet algorithme sur la permutation associée à l'arbre (c) de la figure 1 et vérifier qu'elle est bien triable par pile.

5. Écrire une fonction trier : `int list → int list` qui implémente cet algorithme dans un style fonctionnel. Par exemple, trier  $[3; 2; 0; 1; 5; 4; 7; 6]$  doit s'évaluer en la liste  $[7; 6; 5; 4; 3; 2; 1; 0]$ . On utilisera directement une liste pour implémenter une pile.

*Indication* : écrire une fonction auxiliaire de type `int list -> int list -> int list -> int list` qui prend en paramètre une liste d'entrée, une pile et une liste de sortie, et qui, en fonction de la forme de la liste d'entrée et de la pile, applique une étape élémentaire avant de procéder récursivement.

6. Montrer que s'il existe  $0 \leq i < j < k \leq n - 1$  tels que  $\sigma_k < \sigma_i < \sigma_j$ , alors  $\sigma$  n'est pas triable par une pile.

7. On se propose de montrer que les permutations de  $\llbracket 0, n - 1 \rrbracket$  triables par une pile sont en bijection avec les arbres binaires non étiquetés à  $n$  nœuds.

(a) Montrer que la permutation associée à un arbre binaire est triable par pile. On pourra remarquer le lien entre le parcours préfixe et l'opération empiler d'une part et le parcours infixe et l'opération dépiler d'autre part.

(b) Montrer qu'une permutation triable par pile est une permutation associée à un arbre binaire.

*Indication* : on peut prendre  $\sigma_0$  comme racine, puis procéder récursivement avec les  $\sigma_0 - 1$  éléments pour construire le fils gauche et avec le reste pour le fils droit.

# Chapitre 11

## (CCINP) Graphes \* (CCINP 0, ex B, corrigé — oral - 144 lignes)

\*

Graphes, C,  
sources : `ccinpngraphe.tex`

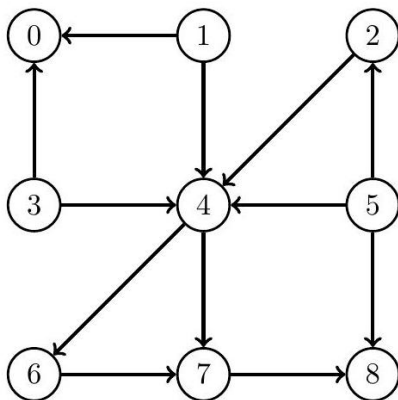
L'exercice suivant est à traiter dans le langage C. Vous pourrez travailler dans le fichier `ccinpngraphe.c` fourni.

Dans cet exercice, tous les graphes seront orientés. On représente un graphe orienté  $G = (S, A)$ , avec  $S = \{0, \dots, n-1\}$ , en C par la structure suivante :

```
struct graph_s {
    int n;
    int degre[100];
    int voisins[100][10];
};
```

L'entier  $n$  correspond au nombre de sommets  $|S|$  du graphe. On suppose que  $n \leq 100$ . Pour  $0 \leq s < n$ , la case `degre[s]` contient le degré sortant  $d^+(s)$ , c'est-à-dire le nombre de successeurs, appelés ici voisins, de  $s$ . On suppose que ce degré est toujours inférieur à 10. Pour  $0 \leq s < n$ , la case `voisins[s]` est un tableau contenant, aux indices  $0 \leq i < d^+(s)$ , les voisins du sommet  $s$ . Il s'agit donc d'une représentation par listes d'adjacences où les listes sont représentées par des tableaux en C.

Un programme en C vous est fourni dans lequel le graphe suivant est représenté par la variable `g_exemple`.



Pour  $s \in S$  on note  $\mathcal{A}(s)$  l'ensemble des sommets accessibles à partir de  $s$ . Pour  $s \in S$ , le maximum des degrés d'un sommet accessible à partir de  $s$  est noté  $d^*(s) = \max \{d^+(s') \mid s' \in \mathcal{A}(s)\}$ . Par exemple, pour le graphe ci-dessus,  $\mathcal{A}(2) = \{2, 4, 6, 7, 8\}$  et  $d^*(2) = 2$  car  $d^+(4) = 2$ . Dans cet exercice, on cherche à calculer  $d^*(s)$  pour chaque sommet  $s \in S$ . On représente un sous-ensemble de sommets  $S' \subseteq S$  par un tableau de booléens de taille  $n$ , contenant `true` à la case d'indice  $s'$  si  $s' \in \mathcal{A}(s)$  et `false` sinon.

1. Écrire une fonction `int degre_max (graph* g, bool* partie)` qui calcule le degré maximal d'un sommet  $s' \in S'$  dans un graphe  $G = (S, A)$  pour une partie  $S' \subseteq S$  représentée par  $S$ , c'est-à-dire qui calcule  $\max \{d^+(s') \mid s' \in S'\}$ .
2. Écrire une fonction `bool* accessibles (graph* g, int s)` qui prend en paramètre un graphe et un sommet  $s$  et qui renvoie un (pointeur sur un) tableau de booléens de taille  $n$  représentant  $\mathcal{A}(s)$ . Une fonction `nb_accessibles` qui utilise votre fonction et un test pour l'exemple ci-dessus vous sont donnés dans le fichier à compléter.
3. Écrire une fonction `int degre_etoile (graph* g, int s)` qui calcule  $d^*(s)$  pour un graphe et un sommet passé en paramètre. Quelle est la complexité de votre approche ?
4. Linéariser le graphe donné en exemple ci-dessus, c'est-à-dire représenter ses sommets sur une même ligne dans l'ordre donné par un tri topologique, tous les arcs allant de gauche à droite.

5. Dans cette question, on suppose que le graphe  $G = (S, A)$  est acyclique. Décrire un algorithme permettant de calculer tous les  $d^*(s)$  pour  $s \in S$  en  $O(|S| + |A|)$ .
6. On ne suppose plus le graphe acyclique. Décrire un algorithme permettant de calculer tous les  $d^*(s)$  pour  $s \in S$  en  $O(|S| + |A|)$ .

## Chapitre 12

# (CCINP) SAT \* (CCINP 0, ex B, corrigé — oral - 148 lignes)

\*

Logique propositionnelle, Algorithmes probabilistes,  
sources : `ccinpsat.tex`

L'exercice suivant est à traiter dans le langage *OCaml*.

On s'intéresse au problème SAT pour une formule en forme normale conjonctive. On se fixe un ensemble fini  $\mathcal{V} = \{v_0, v_1, \dots, v_{n-1}\}$  de variables propositionnelles.

Un littéral  $\ell_i$  est une variable propositionnelle  $v_i$  ou la négation d'une variable propositionnelle  $\neg v_i$ . On représente un littéral en *OCaml* par un type énuméré : le littéral  $v_i$  est représenté par `V i` et le littéral  $\neg v_i$  par `NV i`. Une clause  $c = \ell_0 \vee \ell_1 \vee \dots \vee \ell_{|c|-1}$  est une disjonction de littéraux, que l'on représente en *OCaml* par un tableau de littéraux. On ne considérera dans cet exercice que des formules sous forme normale conjonctive, c'est-à-dire sous forme de conjonctions de clauses  $c_0 \wedge c_1 \wedge \dots \wedge c_{m-1}$ . On représente une telle formule en *OCaml* par une liste de clauses, soit une liste de tableaux de littéraux. On n'impose rien sur les clauses : une clause peut être vide et un même littéral peut s'y trouver plusieurs fois. De même une formule peut n'être formée d'aucune clause, elle est alors notée  $\top$  et est considérée comme une tautologie. Une valuation  $v : \mathcal{V} \rightarrow \{V, F\}$  est représentée en *OCaml* par un tableau de booléens.

Un programme *OCaml* à compléter vous est fourni. La fonction `initialise : int → valuation` permet d'initialiser une valuation aléatoire.

1. Implémenter la fonction `evaluate : clause → valuation → bool` qui vérifie si une clause est satisfaite par une valuation.

Étant donné une formule  $f$  constituée de  $m$  clauses  $c_0 \wedge c_1 \wedge \dots \wedge c_{m-1}$  définies sur un ensemble de  $n$  variables, la fonction `random_sat` a pour objectif de trouver une valuation qui satisfait la formule, s'il en existe une et qu'elle y arrive. Cette fonction doit effectuer au plus  $k$  tentatives et renvoyer un résultat de type `valuation option`, avec une valuation qui satisfait la formule passée en paramètre si elle en trouve une et la valeur `None` sinon.

L'idée de ce programme est d'effectuer une assignation aléatoire des variables puis de vérifier que chaque clause est satisfaite. Si une clause n'est pas satisfaite, on modifie aléatoirement la valeur associée à un littéral de cette clause, qui devient ainsi satisfaite, puis on recommence.

2. Si ce programme renvoie `None`, peut-on conclure que la formule  $f$  en entrée n'est pas satisfiable? De quel type d'algorithme probabiliste s'agit-il?
3. Proposer un jeu de 5 tests élémentaires permettant de tester la correction de ce programme.
4. Compléter la fonction `random_sat`.

On s'intéresse maintenant au problème MAX-SAT qui consiste, toujours pour une formule en forme normale conjonctive comme ci-dessus, à trouver le plus grand nombre de clauses de cette formule simultanément satisfiables. Un algorithme d'approximation probabiliste naïf pour obtenir une solution approchée consiste à générer aléatoirement  $k$  valuations et retenir celle qui maximise le nombre de clauses satisfaites.

1. Implémenter cette approche en *OCaml* et vérifier sur quelques exemples. Quelle est sa complexité dans le meilleur et dans le pire cas?
2. Sous l'hypothèse  $P \neq NP$ , peut-il exister un algorithme de complexité polynomiale pour résoudre MAX-SAT? Justifier.



# Chapitre 13

## (CCINP) Sac à dos \* (CCINP 0, ex B, corrigé — oral - 178 lignes)

\*

Algorithmique, C, Algorithmes gloutons, Backtracking, Branch and bound,  
sources : `ccinpsacados.tex`

L'exercice suivant est à traiter dans le langage C.

On dispose de  $n \geq 1$  objets  $\{o_0, \dots, o_{n-1}\}$  de valeurs respectives  $(v_0, \dots, v_{n-1}) \in \mathbb{N}^n$  et de poids respectifs  $(p_0, \dots, p_{n-1}) \in \mathbb{N}^n$ . On souhaite transporter dans un sac de poids maximum  $p_{\max}$  un sous-ensemble d'objets ayant la plus grande valeur possible. Formellement, on souhaite donc maximiser

$$\sum_{i=0}^{n-1} x_i v_i$$

sous les contraintes

$$(x_0, \dots, x_{n-1}) \in \{0, 1\}^n \quad \text{et} \quad \sum_{i=0}^{n-1} x_i p_i \leq p_{\max}$$

Intuitivement, la variable  $x_i$  vaut 1 si l'objet  $o_i$  est mis dans le sac et 0 sinon.

On propose d'utiliser un algorithme glouton dont le principe est de considérer les objets  $o_0, o_1, \dots, o_{n-1}$  dans l'ordre et de choisir à l'étape  $i$  l'objet  $i$  (donc poser  $x_i = 1$ ) si celui-ci rentre dans le sac avec la contrainte de poids maximal respectée et ne pas le choisir (donc poser  $x_i = 0$ ) sinon. On remarque que les valeurs  $v_0, v_1, \dots, v_{n-1}$  ne sont pas directement utilisées par cet algorithme, elle le seront lors du tri éventuel des objets.

1. Proposer un type de données pour implémenter, pour  $n$  objets, leurs valeurs, leurs poids et les indicateurs  $x_0, x_1, \dots, x_{n-1}$ .
2. Écrire une fonction qui implémente la méthode gloutonne décrite ci-dessus à partir de  $n, p_{\max}$  et  $p_0, p_1, \dots, p_{n-1}$  et qui permet de renvoyer les indicateurs  $x_0, x_1, \dots, x_{n-1}$  pour le choix glouton.
3. Écrire un programme complet qui permet de lire sur l'entrée standard (au clavier par défaut) un entier  $n \geq 1$ , puis un entier naturel  $p_{\max}$ , puis  $n$  entiers naturels correspondant aux valeurs  $v_0, v_1, \dots, v_{n-1}$ , puis  $n$  entiers naturels correspondant aux poids  $p_0, p_1, \dots, p_{n-1}$  et qui affiche sur la sortie standard (l'écran par défaut) sous une forme de votre choix les indicateurs  $x_0, x_1, \dots, x_{n-1}$ , la valeur de la solution  $\sum_{i=0}^{n-1} x_i v_i$  et le poids utilisé  $\sum_{i=0}^{n-1} x_i p_i$ . On rappelle que le spécificateur de format pour lire ou écrire un entier est `%d`.
4. L'algorithme glouton ci-dessus donne-t-il toujours une solution optimale?
  - (a) si on ne suppose rien sur l'ordre des objets a priori ;
  - (b) si les objets sont triés par ordre de valeur décroissante ;
  - (c) si les objets sont triés par ordre de poids croissant ;
  - (d) si les objets sont triés par ordre décroissant des quotients  $\frac{v_i}{p_i}$ .

Justifier à chaque fois votre réponse à l'aide d'un contre-exemple ou d'une démonstration.

5. Le problème du sac à dos étudié dans cet exercice est un problème d'optimisation. Donner le problème de décision associé en utilisant une valeur seuil  $v_{\text{seuil}}$ . Montrer que ce problème de décision est dans la classe NP.
6. Quelle serait la complexité d'une méthode qui examinerait tous les choix possibles pour retenir le meilleur ? Quelles stratégies d'élagage pourrait-on mettre en œuvre pour réduire l'espace de recherche ?

On peut montrer que ce problème de décision est NP-complet.



## Chapitre 14

# (CCINP) Activation de processus \*

## (CCINP 23, ex A, corrigé (R) — oral - 153 lignes)

\*

Complexité, Réduction,  
sources : `procesCCINP23.tex`

Soit un système temps réel à  $n$  processus asynchrones  $i \in \llbracket 1, n \rrbracket$  et  $m$  ressources  $r_j$ . Quand un processus  $i$  est actif, il bloque un certain nombre de ressources listées dans un ensemble  $P_i$  et une ressource ne peut être utilisée que par un seul processus. On cherche à activer simultanément le plus de processus possible.

Le problème de décision **ACTIVATION** correspondant ajoute un entier  $k$  aux entrées et cherche à répondre à la question : "Est-il possible d'activer au moins  $k$  processus en même temps?"

1. Soit  $n = 4$  et  $m = 5$ . On suppose que  $P_1 = \{r_1, r_2\}$ ,  $P_2 = \{r_1, r_3\}$ ,  $P_3 = \{r_2, r_4, r_5\}$  et  $P_4 = \{r_1, r_2, r_4\}$ . Est-il possible d'activer 2 processus en même temps? Même question avec 3 processus.
2. Dans le cas où chaque processus n'utilise qu'une seule ressource, proposer un algorithme résolvant le problème **ACTIVATION**. Évaluer la complexité de votre algorithme.

On souhaite montrer que **ACTIVATION** est NP-complet.

3. Donner un certificat pour ce problème.
4. Écrire en pseudo code un algorithme de vérification polynomial. On supposera disposer de trois primitives, toutes trois de complexité polynomiale :
  - (a) `appartient(c, i)` qui renvoie `Vrai` si le processus  $i$  est dans l'ensemble d'entiers  $c$ .
  - (b) `intersecte(Pi, R)` qui renvoie `Vrai` si le processus  $i$  utilise une ressource incluse dans un ensemble de ressources  $R$ .
  - (c) `ajoute(Pi, R)` qui ajoute les ressources  $P_i$  dans l'ensemble  $R$  et renvoie ce nouvel ensemble.

En théorie des graphes, le problème **STABLE** se pose la question de l'existence dans un graphe non orienté  $G = (S, A)$  d'un ensemble d'au moins  $k$  sommets ne contenant aucune paire de sommets voisins. En d'autres termes, existe-t-il  $S' \subset S$ ,  $|S'| \geq k$  tel que  $s, p \in S' \Rightarrow (s, p) \notin A$ ?

5. En utilisant le fait que **STABLE** est NP-complet, montrer par réduction que le problème **ACTIVATION** est également NP-complet.



## Chapitre 15

# (CCINP) Grammaires algébriques \*

## (CCINP 23, ex A, corrigé (R) — oral - 123 lignes)

\*

Grammaires,  
sources : `grammccinp1.tex`

On considère la grammaire algébrique  $G$  sur l'alphabet  $\Sigma = \{a, b\}$  et d'axiome  $S$  dont les règles sont :

$$S \rightarrow SaS \mid b$$

1. Cette grammaire est-elle ambiguë ? Justifier.
2. Déterminer (sans preuve pour cette question) le langage  $L$  engendré par  $G$ . Quelle est la plus petite classe de langages à laquelle  $L$  appartient ?
3. Prouver que  $L = L(G)$ .
4. Décrire une grammaire qui engendre  $L$  de manière non ambiguë en justifiant de cette non ambiguë.
5. Montrer que tout langage dans la même classe de langages que  $L$  peut être engendré par une grammaire algébrique non ambiguë.



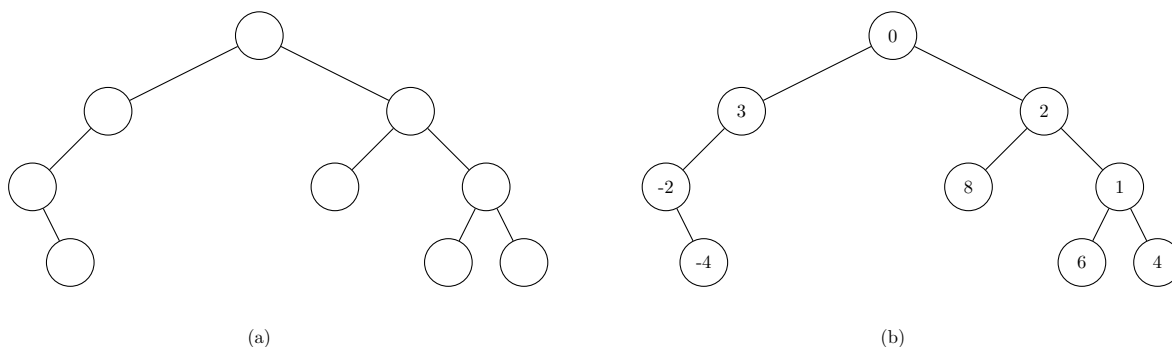
## Chapitre 16

# (CCINP) Minima locaux dans des arbres (Couverture dans des arbres) \* (CCINP 23, ex A, corrigé (R) — oral - 148 lignes)

\*

Arbres, Complexité,  
sources : arbreCCINPA.tex

Dans cet exercice, on considère des arbres binaires étiquetés par des entiers relatifs deux à deux distincts. Un nœud est un minimum local d'un arbre si son étiquette est plus petite que celle de son éventuel père et celles de ses éventuels fils. Considérons par exemple l'étiquetage (b) de l'arbre binaire non étiqueté (a) :



1. Déterminer le ou les minima locaux de l'arbre (b).
  2. Donner une définition inductive permettant de définir les arbres binaires ainsi que la définition de la hauteur d'un arbre. Quelle est la hauteur de l'arbre (b) ?
  3. Montrer que tout arbre non vide possède un minimum local.
  4. Proposer un algorithme permettant de trouver un minimum local d'un arbre non vide et déterminer sa complexité.
- On considère un arbre binaire non étiqueté que l'on souhaite étiqueter par des entiers relatifs distincts deux à deux de manière à maximiser le nombre de minima locaux de cet arbre.
5. Proposer sans justifier un étiquetage de l'arbre (a) qui maximise le nombre de minima locaux.
  6. Proposer un algorithme qui, étant donné un arbre binaire non étiqueté en entrée, permet de calculer le nombre maximal de minima locaux qu'il est possible d'obtenir pour cet arbre. Déterminer la complexité de votre algorithme.
  7. Montrer que, pour un arbre de taille  $n \in \mathbb{N}$ , le nombre maximal de minima locaux est majoré par  $\left\lfloor \frac{2n+1}{3} \right\rfloor$ . On pourra remarquer que les nœuds non minimaux couvrent l'ensemble des arêtes de l'arbre.



# Chapitre 17

## (CCINP) Formules propositionnelles croissantes \* (CCINP 23, ex A, corrigé — oral - 142 lignes)

\*

Logique propositionnelle,  
sources : logccinp1.tex

### Formules propositionnelles croissantes (exn 2023) :

On fixe un entier  $n \geq 1$  et  $E = \{x_1, \dots, x_n\}$  un ensemble de variables propositionnelles. Étant données deux applications  $a : E \rightarrow \{V, F\}$  et  $b : E \rightarrow \{V, F\}$  on dit que  $a$  est plus petite que  $b$  (que l'on note  $a \leq b$ ) si :

$$\forall x \in E, a(x) = V \implies b(x) = V.$$

Dans un but de simplification des calculs, on pourra faire les abus de notation suivants : assimiler  $V$  à 1 et  $F$  à 0 et vice versa. Avec cet abus, la propriété  $a \leq b$  se traduit par :

$$\forall x \in E, a(x) \leq b(x).$$

1. Étant donnée une valuation sur  $E$ , rappeler comment on l'étend naturellement en une valuation sur les formules propositionnelles.

On dit qu'une formule propositionnelle  $P$  est *croissante* si pour tout  $a, b$  des valuations vérifiant  $a \leq b$ , on a :

$$a(P) = V \implies b(P) = V.$$

2. Montrer que si  $P, Q$  sont des formules croissantes, alors  $P \wedge Q$  et  $P \vee Q$  sont des formules croissantes.
3. Soit  $C$  une clause conjonctive satisfiable contenant au moins un littéral. Montrer qu'elle est croissante si et seulement si elle ne contient aucun littéral de la forme  $\neg x$  avec  $x \in E$ .
4. On considère une formule propositionnelle  $P$  qui n'est ni une tautologie, ni une antilogie.
  - (a) Montrer que si  $P$  est logiquement équivalente à une disjonction de clauses conjonctives dont aucune ne contient un littéral de la forme  $\neg x$  avec  $x \in E$ , alors  $P$  est une formule propositionnelle croissante.
  - (b) Réciproquement, montrer que si  $P$  est une formule propositionnelle croissante, alors elle est logiquement équivalente à une disjonction de clauses conjonctives dont aucune ne contient un littéral de la forme  $\neg x$  avec  $x \in E$ .



# Chapitre 18

## (CCINP) Langages locaux \* (CCINP 23, ex B, corrigé (R) — oral - 215 lignes)

\*

Langages, Ocaml, Automates,  
sources : langlocccinp1.tex

*Consignes : Cet énoncé est accompagné d'un code compagnon en OCaml localite.ml fournissant le type décrit par l'énoncé et quelques fonctions auxiliaires : il est à compléter en y implémentant les fonctions demandées. On privilégiera un style de programmation fonctionnel.*

On considère un alphabet  $\Sigma$ . Si  $L$  est un langage sur  $\Sigma$ , on note :

- $P(L) = \{a \in \Sigma \mid a\Sigma^* \cap L \neq \emptyset\}$  l'ensemble des premières lettres des mots de  $L$ .
- $D(L) = \{a \in \Sigma \mid \Sigma^*a \cap L \neq \emptyset\}$  l'ensemble des dernières lettres des mots de  $L$ .
- $F(L) = \{m \in \Sigma^2 \mid \Sigma^*m\Sigma^* \cap L \neq \emptyset\}$  l'ensemble des facteurs de longueur 2 des mots de  $L$ .
- $N(L) = \Sigma^2 \setminus F(L)$  l'ensemble des mots de taille 2 qui ne sont pas facteurs de mots de  $L$ .

On rappelle qu'un langage  $L$  est dit *local* si et seulement si l'égalité de langages suivantes est vérifiée :

$$L \setminus \{\varepsilon\} = (P(L)\Sigma^* \cap \Sigma^*D(L)) \setminus (\Sigma^*N(L)\Sigma^*)$$

1. Calculer les ensembles  $P(L)$ ,  $D(L)$ ,  $F(L)$  et  $N(L)$  dans le cas où  $L$  est le langage dénoté par l'expression régulière  $a^*(ab)^* + aa$  sur l'alphabet  $\{a, b\}$ . Ce langage est-il local ? On vérifiera la cohérence entre les réponses à cette question et celles obtenues via les fonctions demandées dans la suite de l'énoncé.

On cherche dans la suite de l'exercice à concevoir un algorithme répondant à la spécification suivante :

**Entrée** : Une expression régulière  $e$  sur un alphabet  $\Sigma$  ne faisant pas intervenir le symbole  $\emptyset$ .  
**Sortie** : Vrai si le langage dénoté par  $e$  est local, faux sinon.

Par défaut, dans la suite de l'énoncé, "expression régulière" signifie "expression régulière ne faisant pas intervenir le symbole  $\emptyset$ ". Les expressions régulières seront manipulées en OCaml via le type somme suivant :

```
type regexp =  
  | Epsilon  
  | Letter of string (*La chaîne en question sera toujours de longueur 1*)  
  | Union of regexp * regexp  
  | Concat of regexp * regexp  
  | Star of regexp
```

On propose tout d'abord de calculer les ensembles  $P(L)$ ,  $D(L)$  et  $F(L)$  à partir d'une expression régulière dénotant  $L$ . Ces ensembles seront représentés en OCaml par des listes de chaînes de caractères qui vérifieront les propriétés suivantes :

- Elles sont triées dans l'ordre croissant selon l'ordre lexicographique.
- Elles sont sans doublons.

L'énoncé fournit une fonction `union` permettant de calculer l'union sans doublons de deux listes triées. La définition inductive d'une expression régulière invite à calculer inductivement les ensembles  $P(L)$ ,  $D(L)$  et  $F(L)$ . C'est ce que propose la fonction `compute_P` fournie par l'énoncé.

2. En supposant que la fonction `contains_epsilon : regexp -> bool` renvoie `true` si et seulement si le langage dénoté par l'expression régulière en entrée contient le mot  $\varepsilon$ , justifier brièvement la correction de `compute_P`.
3. Implémenter la fonction `contains_epsilon`.
4. Sur le modèle de `compute_P`, implémenter une fonction `compute_D : regexp -> string list` permettant le calcul de l'ensemble  $D(L)$  étant donnée une expression régulière dénotant le langage  $L$ .

5. Expliquer en langage naturel comment calculer récursivement l'ensemble  $F(L)$  étant donnée une expression régulière  $e$  dénotant le langage  $L$ . Si  $e = e_1e_2$  on pourra exprimer  $F(L)$  en fonction notamment de  $P(L_2)$  et  $D(L_1)$  où  $L_1$  (resp.  $L_2$ ) est le langage dénoté par  $e_1$  (resp.  $e_2$ ).
6. Écrire une fonction `prod : string list -> string list -> string list` calculant le produit cartésien des deux listes en entrée, qu'on pourra supposer triées dans l'ordre lexicographique croissant et sans doublons, puis qui pour chaque couple de chaînes dans la liste obtenue les concatène. Par exemple :

$$\text{prod } ["a" ; "c" ; "e"] ["b" ; "c"] = ["ab" ; "ac" ; "cb" ; "cc" ; "eb" ; "ec"]$$

7. En déduire une fonction `compute_F : regexp -> string list` déterminant l'ensemble  $F(L)$  étant donnée une expression régulière dénotant le langage  $L$ .

Dans les questions qui suivent, on ne demande PAS d'implémenter les algorithmes décrits.

8. Décrire en pseudo-code ou en langage naturel un algorithme permettant de calculer un automate reconnaissant  $(P(L)\Sigma^* \cap \Sigma^*D(L)) \setminus (\Sigma^*N(L)\Sigma^*)$  étant donnée une expression régulière dénotant  $L$ .
9. Décrire un algorithme permettant de détecter si le langage dénoté par une expression régulière est local ou non.
10. Pourquoi est-il légitime de ne considérer que les expressions régulières ne faisant pas intervenir  $\emptyset$ ? Comment modifier l'algorithme obtenu dans le cas où cette contrainte n'est plus vérifiée?

# Chapitre 19

## (CCINP) Programmation dynamique pour la récolte de fleurs \* (CCINP 23, ex B, corrigé (R) — oral - 204 lignes)

\*

Programmation dynamique, C,  
sources : `ccinprecoltefleurs.tex`

*Consignes : Cet énoncé est accompagné d'un code compagnon en C `bouquet_enonce.c` fournissant certaines des fonctions mentionnées dans l'énoncé : il est à compléter en y implémentant les fonctions demandées. La ligne de compilation `gcc -o main.exe *.c -lm` vous permet de créer un exécutable `main.exe` à partir du ou des fichiers C fournis. Vous pouvez également utiliser l'utilitaire `make`. En ligne de commande, il suffit de taper `make`. Dans les deux cas, si la compilation réussit, le programme peut être exécuté avec la commande `./main.exe`. Si vous désirez forcer la compilation de tous les fichiers, vous pouvez au préalable nettoyer le répertoire en faisant `make clean` et relancer `make`.*

Une petite fille se trouve en haut à gauche (case A) d'un champ modélisé par un tableau rectangulaire de taille  $m \times n$  et doit se rendre dans la case B en bas à droite du champ où réside sa grand-mère (figure ci-dessous).

A	✿	✿	✿	✿	✿	✿
✿	✿	✿	✿	✿	✿	✿
✿	✿	✿	✿	✿	✿	✿
✿	✿	✿	✿	✿	✿	B

Chaque case du tableau, *y compris les cases A et B*, contient un certain nombre de fleurs. La petite fille, qui connaît depuis sa position initiale le nombre de fleurs de chaque case, doit se déplacer vers B de case en case, les seuls mouvements autorisés étant vers le bas ou vers la droite. À chaque déplacement, elle récolte les fleurs de la case atteinte. L'objectif pour elle est alors de faire le bouquet avec le plus de fleurs possible lors de son déplacement pour l'offrir à sa grand-mère.

1. On considère le champ suivant :

0 (A)	1	2	3
1	2	3	4
2	3	4	0
3	4	0	1 (B)

Donner le nombre maximal de fleurs cueillies par la petite fille.

2. On note  $n(i, j)$  le nombre maximum de fleurs que la petite fille peut récolter en se déplaçant de A à la case  $(i, j)$ . Exprimer  $n(i, j)$  en fonction de  $n(i - 1, j)$  et  $n(i, j - 1)$ . En déduire une fonction récursive de prototype `int recolte(int champ[m][n], int i, int j)` qui, étant données les coordonnées  $i, j$  d'une case, calcule le nombre maximum de fleurs cueillies par la petite fille de A à la case  $(i, j)$ .

3. On suppose  $m = n = 4$  et on effectue donc un appel à `recolte(champ, 3, 3)` pour résoudre le problème posé. Donner le nombre de fois où votre fonction calcule le nombre de fleurs maximum cueillies dans la case  $(1, 1)$  (deuxième case de la diagonale).

D'une manière générale, le nombre d'appels à la fonction récursive est important. On a donc intérêt à transformer l'algorithme récursif en algorithme dynamique. On propose de déclarer dans le programme principal un tableau `fleurs` dont la case  $(i, j)$  est destinée à contenir la récolte maximale que la petite fille peut obtenir en cheminant de A vers la case  $(i, j)$ .

4. Dans quel ordre remplir le tableau `fleurs` de sorte à éviter de recalculer une valeur ?  
5. Écrire une fonction de prototype `int recolte_iterative(int champ[m][n], int i, int j, int fleurs[m][n])` qui calcule, stocke dans `fleurs[i][j]` et retourne la cueillette maximale obtenue en parcourant le champ de A à la case  $(i, j)$ .

La fonction `recolte_iterative` permet de déterminer la cueillette maximale en  $(i, j)$  mais ne précise pas le chemin parcouru pour l'obtenir.

6. Écrire la fonction de prototype `void déplacements(int fleurs[m][n], int i, int j)` qui affiche la suite des déplacements effectués par la petite fille sur un chemin permettant de récolter le nombre maximum de fleurs entre  $(0,0)$  et  $(i, j)$ .
7. Insérer un appel de `déplacements` dans la fonction `recolte_iterative` pour afficher le chemin parcouru.

# Chapitre 20

## (CCINP) chemins simples sans issue \* (CCINP 23, ex B, corrigé (R) — oral - 199 lignes)

\*

Graphes, Ocaml, Complexité, Backtracking,  
sources : ccinchemins\_simples.tex

*Consignes : Cet exercice est à traiter en OCaml. Le fichier chemins\_simples.ml est fourni avec ce sujet. Il est à compléter en y implémentant les fonctions demandées.*

L'objectif de cet exercice est de programmer une fonction générant la liste des chemins simples sans issue d'un graphe. On rappelle les définitions d'un graphe, d'un chemin, et on donne leur représentation en OCaml.

Un *graphe orienté* est un couple  $(V, E)$  où  $V$  est un ensemble fini (ensemble des sommets),  $E$  est un sous-ensemble de  $V \times V$  où tout élément  $(v_1, v_2) \in E$  vérifie  $v_1 \neq v_2$  (ensemble des arcs).

Étant donné un graphe  $G = (V, E)$  un *chemin non vide* de  $G$  est une suite finie  $s_0, \dots, s_n$  de sommets de  $V$  avec  $n \geq 0$  et vérifiant  $\forall i \in \{0, \dots, n-1\}, (s_i, s_{i+1}) \in E$ . On dit que ce chemin est *simple* si  $s_0, \dots, s_n$  sont distincts deux à deux. On dit qu'il est *sans issue* si pour tout  $s_{n+1}$  sommet tel que  $(s_n, s_{n+1}) \in E$ ,  $s_{n+1}$  appartient à  $\{s_0, \dots, s_n\}$ .

Dans la suite, les graphes considérés sont définis sur un ensemble de sommets de la forme  $\{0, 1, \dots, n-1\}$ . Pour représenter un graphe en OCaml, on utilise le type suivant :

```
type graphe = int list array
```

qui correspond à un encodage par un tableau de listes d'adjacence. Par exemple, le graphe

$$G_1 = (\{0, 1, 2, 3\}, \{(0, 1), (0, 3), (2, 0), (2, 1), (2, 3), (3, 1)\})$$

est représenté par le tableau `[ [1 ; 3] ; [] ; [0 ; 1 ; 3] ; [1] ]`. L'ordre dans lequel sont écrits les éléments dans les listes importe peu. Par contre, l'emplacement des listes dans le tableau est important. Par exemple, `[ [] ; [0] ; [0 ; 3 ; 1] ; [1] ]` représente le graphe

$$G_2 = (\{0, 1, 2, 3\}, \{(1, 0), (2, 0), (2, 1), (2, 3), (3, 1)\})$$

On rappelle différentes fonctions pouvant être utiles :

- `List.filter` : `('a -> bool) -> 'a list -> 'a list` où l'expression `List.filter f l` est la liste obtenue en gardant uniquement les éléments `x` de `l` vérifiant `f`.
- `List.iter` : `('a -> unit) -> 'a list -> unit` où `List.iter f l` correspond à `(f a0) ; (f a1) ; ... ; (f an)` dans le cas où on a `l = a0 :: a1 :: ... :: an :: []`.
- `List.rev` : `'a list -> 'a list` est une fonction qui renvoie le retourné d'une liste. Par exemple, `List.rev [3 ; 1 ; 2 ; 2 ; 4]` est égal à `[4 ; 2 ; 2 ; 1 ; 3]`.
- `Array.length` : `'a array -> int` est une fonction qui renvoie la longueur d'un tableau.

Les questions de programmation sont à traiter dans le fichier `chemins_simples.ml`. L'utilisation d'autres fonctions de la bibliothèque que celles mentionnées sont à reprogrammer.

1. Écrire une fonction `est_sommet` : `graphe -> int -> bool` où `est_sommet g a` est égal à `true` si `a` est un sommet du graphe `g` et `false` sinon.
2. Écrire une fonction `appartient` : `'a list -> 'a -> bool` où `appartient l x` est égal à `true` si `x` est un élément de `l` et `false` sinon.
3. Écrire une fonction `est_chemin` : `graphe -> int list -> bool` où `est_chemin g l` est égal à `true` si `l` est un chemin de `g` et `false` sinon. On suppose que la liste vide représente le chemin vide, qui est bien un chemin et que les éléments de `l` sont bien des sommets du graphe `g`.

4. Compléter la fonction `est_chemin_simple_sans_issue` : `graphe -> int list -> bool`, où `est_chemin g l` est égal à `true` si `l` est un chemin simple sans issue de `g` et `false` sinon. On supposera que les éléments de `l` sont des sommets du graphe `g` et que le chemin vide n'est pas simple sans issue.
5. On cherche à écrire une fonction qui construit la liste des chemins simples sans issue d'un graphe. Pour cela, on procède à l'aide de parcours en profondeur et d'un algorithme de retour sur trace. Compléter le code de la fonction `genere_chemins_simples_sans_issue` présent dans le fichier `chemins_simples.ml` et qui permet de générer la liste des chemins simples sans issue d'un graphe.
6. Écrire des expressions donnant les listes des chemins simples pour les deux graphes  $G_1$  et  $G_2$ .
7. Expliciter la complexité des fonctions `appartient` et `est_chemin_simple_sans_issue`.

# Chapitre 21

## (CCINP) calculs avec les flottants \* (CCINP 23, ex B, corrigé — oral - 206 lignes)

\*

Représentation des nombres, C,  
sources : `ccinpfloottants.tex`

*Consignes : Cet exercice est à traiter en C. Le fichier `flottants.c` est fourni avec ce sujet. Il est à compléter en y implémentant les fonctions demandées.*

Un nombre réel  $x$  est représenté en machine en base 2 par un flottant qui a un signe  $s$ , une mantisse  $m$  et un exposant  $e$  tel que  $x = s \times m \times 2^e$ . Dans la norme IEEE 754, en convention normalisée la partie entière de la mantisse est 1 qui est un bit caché. En simple précision, le signe est codé sur 1 bit, la partie décimale de la mantisse sur 23 bits et l'exposant sur 8 bits. En double précision, le signe est codé sur 1 bit, la partie décimale de la mantisse sur 52 bits et l'exposant sur 11 bits.

Dans cet exercice, on observe le résultat de calculs obtenus par un programme. On pourra utiliser la fonction de signature : `double pow(double v, double p)` qui calcule  $v^p$ . On ajoutera alors l'option `-lm` à la fin de la ligne permettant de compiler le fichier.

1. Dans la fonction principale `main`, on a défini 3 variables  $a, b, c$  de type `double`. Compléter le code pour calculer et afficher le résultat des opérations  $(a + b) + c$  et  $a + (b + c)$ . Que constatez-vous ?
2. Compte tenu des approximations faites lors du codage, on peut trouver plusieurs nombres  $x$  tels que  $1 + x = 1$  après un calcul fait par la machine. Le plus petit nombre représentable exactement en machine et supérieur à 1 s'écrit  $1 + \epsilon$ , avec  $\epsilon$  un réel appelé  $\epsilon$  machine. On admet que  $\epsilon$  s'écrit sous la forme  $2^{-n}$  avec  $n$  un entier naturel strictement positif. Écrire une fonction de signature `double epsilon()` qui renvoie la valeur de  $n$ . Justifier cette valeur.
3. On considère une suite  $(u_n)_{n \in \mathbb{N}}$  définie par

$$\begin{cases} u_0 = 2 \\ u_1 = -4 \\ u_n = 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1} \times u_{n-2}} \text{ si } n \geq 2 \end{cases}$$

Écrire une fonction de signature `double u(int n)` qui renvoie la valeur du terme  $u_n$ .

*Le jury attend une explication faisant le lien entre la valeur trouvée et le format de représentation des flottants rappelé dans l'énoncé.*

4. La limite théorique de la suite  $(u_n)_{n \in \mathbb{N}}$  est 6. Compléter la fonction `main` afin d'afficher les 22 premiers termes de la suite. Vers quelle valeur semble tendre la suite ?
5. On définit une liste chaînée de nombres à l'aide d'une structure `nb` comportant un `double` et un pointeur vers une structure `nb` définie ci-dessous

```
struct nb {double x; struct nb* suivant;};
```

Écrire une fonction de signature `double somme(struct nb* tab)` qui calcule la somme des éléments de la liste `tab`.

6. L'algorithme suivant permet d'augmenter la précision du calcul lors du calcul d'une somme.

```

1  Entrées : Une liste  $l$  de réels triée dans l'ordre croissant  $\leftrightarrow$ 
      de taille au moins 2.
2  Sorties : La somme des réels contenus dans la liste  $l$ .
3
4  TantQue la liste  $l$  contient strictement plus d'un élément
5      Calculer la somme  $s = x + y$  des deux premiers éléments  $x \leftrightarrow$ 
      et  $y$  de  $l$ 
6      Supprimer  $x$  et  $y$  de  $l$ 
7      Insérer  $s$  dans  $l$  de sorte à ce que  $l$  reste triée
8  Renvoyer l'unique élément de  $l$ 

```

- Compléter la fonction `somme2` qui implémente cet algorithme.
- La fonction proposée ne prend pas en compte un cas d'insertion. Illustrer ce propos.

# Chapitre 22

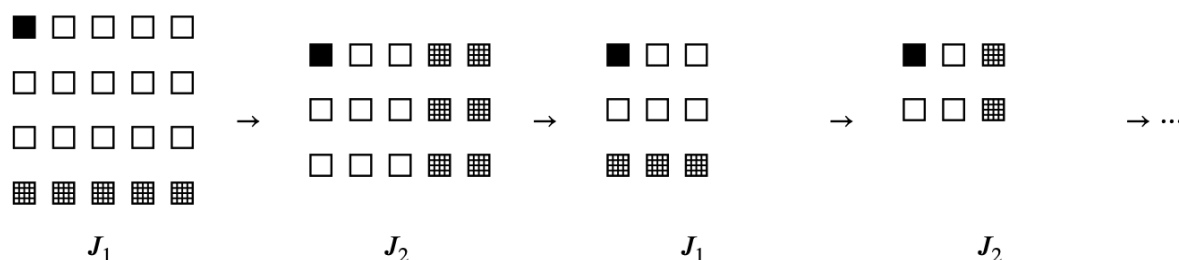
## (CCINP) chomp \* (CCINP 24, ex A, corrigé (R) — oral - 228 lignes)

\*

Jeux,  
sources : ccinpchomp.tex

On considère une variante du jeu de Chomp : deux joueurs  $J_1$  et  $J_2$  s'affrontent autour d'une tablette de chocolat de taille  $l \times c$ , dont le carré en haut à gauche est empoisonné. Les joueurs choisissent chacun leur tour une ou plusieurs lignes (ou une ou plusieurs colonnes) partant du bas (respectivement de la droite) et mangent les carrés correspondants. Il est interdit de manger le carré empoisonné et le perdant est le joueur qui ne peut plus jouer.

Dans la figure ci-dessous, matérialisant un début de partie sur une tablette de taille  $4 \times 5$ , le carré noir est le carré empoisonné, le choix du joueur  $J_i$  est d'abord matérialisé par des carrés hachurés, qui sont ensuite supprimés.



On associe à ce jeu un graphe orienté  $G = (S, A)$ . Les sommets  $S$  sont les états possibles de la tablette de chocolat, définis par un couple  $s = (m, n)$ ,  $m \in \llbracket 1, l \rrbracket$ ,  $n \in \llbracket 1, c \rrbracket$ . De plus,  $(s_i, s_j) \in A$  si un des joueurs peut, par son choix de jeu, faire passer la tablette de l'état  $s_i$  à l'état  $s_j$ . On dit que  $s_j$  est un successeur de  $s_i$  et que  $s_i$  est un prédécesseur de  $s_j$ .

1. Dessiner le graphe  $G$  pour  $l = 2$  et  $c = 3$ . Les états de  $G$  pourront être représentés par des dessins de tablettes plutôt que par des couples  $(m, n)$ .

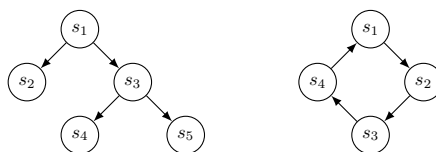
On va chercher à obtenir une stratégie gagnante pour le joueur  $J_1$  par deux manières.

### Utilisation des noyaux de graphe

Soit  $G = (S, A)$  un graphe orienté. On dit que  $N \subset S$  est un noyau de  $G$  si :

- pour tout sommet  $s \in N$ , les successeurs de  $s$  ne sont pas dans  $N$ ,
- tout sommet  $s \in S \setminus N$  possède au moins un successeur dans  $N$

2. Donner tous les noyaux possibles pour les graphes suivants :



Dans la suite, on ne considère que des graphes acycliques.

3. Montrer que tout graphe acyclique admet un puits, c'est-à-dire un sommet sans successeur.

Dans le cas général, le noyau d'un graphe  $G = (S, A)$  est souvent difficile à calculer. Si la dimension du jeu n'est pas trop importante, on peut toutefois le faire en utilisant l'algorithme suivant :

```

1  N = ∅
2  TantQu'il reste des sommets à traiter
3      Chercher un sommet  $s \in S$  sans  $\leftarrow$ 
         successeur
4       $N = N \cup \{s\}$ 
5      Supprimer  $s$  de  $G$  ainsi que ses préd $\leftarrow$ 
         écesseurs

```

4. Justifier que cet algorithme termine et renvoie un noyau.
5. Démontrer que ce noyau est unique. Conclure que le graphe du jeu de Chomp possède un unique noyau  $N$ .
6. Appliquer cet algorithme pour calculer le noyau du jeu de Chomp à 2 lignes et 3 colonnes. Que peut-on dire du sommet (1,1) pour le joueur qui doit jouer ? En déduire à quoi correspondent les éléments du noyau.
7. Montrer que, dans le cas d'un graphe acyclique, tout joueur dont la position initiale n'est pas dans le noyau a une stratégie gagnante. Le joueur  $J_1$  a-t-il une stratégie gagnante pour ce jeu dans le cas  $l = 2$  et  $c = 3$  ?

#### Utilisation des attracteurs

On modélise le jeu par un graphe biparti : pour ce faire, on dédouble les sommets du graphe précédent : un sommet  $s_i$  génère donc deux sommets  $s_i^1$  et  $s_i^2$ ,  $s_i^j$  étant le sommet  $i$  contrôlé par le joueur  $J_j$ . On forme alors deux ensembles de sommets  $S_1 = \{s_i^1\}_i$  et  $S_2 = \{s_i^2\}_i$ , et on construit le graphe de jeu orienté  $G = (S, A)$  avec  $S = S_1 \cup S_2$  et  $S_1 \cap S_2 = \emptyset$ . De plus,  $(s_i^1, s_j^2) \in A$  si le joueur 1 peut, par son choix de jeu, faire passer la tablette de l'état  $s_i^1$  à l'état  $s_j^2$ . On raisonne de même pour  $(s_i^2, s_j^1) \in A$ .

On rappelle la définition d'un attracteur : soit  $F_1$  l'ensemble des positions finales gagnantes pour  $J_1$ . On définit alors la suite d'ensembles  $(\mathcal{A}_i)_{i \in \mathbb{N}}$  par récurrence :  $\mathcal{A}_0 = F_1$  et

$$(\forall i \in \mathbb{N}) \mathcal{A}_{i+1} = \mathcal{A}_i \cup \{s \in S_1 / \exists t \in \mathcal{A}_i, (s, t) \in A\} \cup \{s \in S_2 \text{ non terminal}, \forall t \in S, (s, t) \in A \Rightarrow t \in \mathcal{A}_i\}$$

et  $\mathcal{A} = \bigcup_{i=0}^{\infty} \mathcal{A}_i$  est l'attracteur pour  $J_1$ .

8. Que représente l'ensemble  $\mathcal{A}_i$  ?
9. Dans le cas du jeu de Chomp à deux lignes et trois colonnes (question 1), calculer les ensembles  $\mathcal{A}_i$ . Le joueur  $J_1$  a-t-il une stratégie gagnante ? Comment le savoir à partir de  $\mathcal{A}$  ?

## Chapitre 23

# (CCINP) Grammaires pour des langages de programmation \* (CCINP 24, ex A, corrigé, à débbugger — ds2526 - 90 lignes)

\*

Grammaires, Langages,  
sources : langccinp1.tex

On considère l'alphabet  $\Sigma_0 = \{a, b, c, \dots, z\}$  des lettres minuscules, et une grammaire ayant comme symbole initial  $S$  et les règles  $S \rightarrow AS$  et  $A \rightarrow \sigma A$  avec  $a \in \Sigma_0$ .

1. Quel est le langage engendré par cette grammaire ?

On étudie le langage de programmation  $\hat{c}$  (prononcé « c chapeau »), et on cherche à représenter les identificateurs (i.e. les noms de variables) valides dans ce langage. Un identificateur est valide lorsqu'il contient des caractères de  $\Sigma_1 = \{a, b, c, \dots, z, A, B, \dots, Z, 0, 1, \dots, 9, _\}$  et qu'il ne commence pas par un chiffre.

2. Expliciter une grammaire qui engendre les identificateurs de  $\hat{c}$ .

On note  $E$  le symbole initial d'une grammaire reconnaissant les expressions de  $\hat{c}$  et on définit un non-terminal  $I$  engendrant les programmes de  $\hat{c}$  avec les règles :

- (1).  $I \rightarrow E;$
- (2).  $I \rightarrow \{;$
- (3).  $I \rightarrow \text{while}(E)I$
- (4).  $I \rightarrow B$

$B$  est un non-terminal permettant de représenter des blocs, c'est-à-dire une liste (potentiellement vide) de programmes délimitée par une accolade ouvrante et une accolade fermante. L'exemple suivant est un bloc :

```
{a !=4 ; {while(b) 3+5=t ; ; 5=2 ; {}} d=5-a ;}
```

3. Montrer que les mots engendrés par  $I$  finissent soit par un point-virgule, soit par une accolade fermante.

4. Ajouter aux règles (1) à (4) des règles permettant de décrire les blocs engendrés par  $B$ .

5. Le langage engendré par  $E$  est-il régulier ?



# Chapitre 24

## (CCINP) une relation d'équivalence \* (CCINP 24, ex A, corrigé — oral - 118 lignes)

\*

Algorithmique, Graphes,  
sources : ccinpreleq.tex

On fixe  $n \in \mathbb{N}^*$ . Soient  $\varphi : \llbracket 1, n \rrbracket \rightarrow \llbracket 1, n \rrbracket$  et  $\psi : \llbracket 1, n \rrbracket \rightarrow \llbracket 1, n \rrbracket$ . Soit  $u$  et  $v$  deux éléments de  $\llbracket 1, n \rrbracket$ . On dit que  $u$  et  $v$  sont  $(\varphi, \psi)$ -équivalents s'il existe  $k \in \mathbb{N}$ , un tuple  $(w_0, w_1, \dots, w_{k+1}) \in \llbracket 1, n \rrbracket^{k+2}$  avec  $w_0 = u, w_{k+1} = v$  et vérifiant :

$$\forall i \in \llbracket 0, k \rrbracket, \varphi(w_i) = \varphi(w_{i+1}) \text{ ou } \psi(w_i) = \psi(w_{i+1}).$$

L'objectif est d'écrire un algorithme en pseudo-code permettant de calculer les différentes classes d'équivalence engendrées par cette relation.

1. Justifier rapidement que "être  $(\varphi, \psi)$ -équivalent" est une relation d'équivalence sur l'ensemble  $\llbracket 1, n \rrbracket$ .
2. Pour cette question, on considère les applications  $\varphi$  et  $\psi$  définies par :

$i =$	1	2	3	4	5	6	7	8	9
$\varphi(i) =$	3	2	2	9	6	4	9	5	7
$\psi(i) =$	5	1	3	4	5	1	7	7	4

Calculer les différentes classes d'équivalence.

3. On revient au cas au général. On définit le graphe  $G = (S, A)$  par :

$$S = \llbracket 1, n \rrbracket, A = \{(x, y) \in S^2 \mid x \neq y \text{ et } (\varphi(x) = \varphi(y) \text{ ou } \psi(x) = \psi(y))\}.$$

On fixe  $x$  et  $y$  deux sommets différents de  $S$ . Traduire sur le graphe  $G$  le fait que les sommets  $x$  et  $y$  sont  $(\varphi, \psi)$ -équivalents et en déduire que le calcul des classes d'équivalence de  $G$  se traduit en un problème classique sur les graphes que l'on précisera.

4. Donner en pseudo-code un algorithme permettant de résoudre le problème correspondant sur les graphes.

On fixe  $n$  un nombre pair. On considère deux applications  $\varphi$  et  $\psi$  de  $\llbracket 1, n \rrbracket$  où tout élément de l'image de  $\varphi$  admet exactement deux antécédents par  $\varphi$  et où tout élément de l'image de  $\psi$  admet exactement deux antécédents par  $\psi$ .

Pour  $f \in \{\varphi, \psi\}$ , on note  $G_f$  le graphe  $(S, \{(x, y) \in S^2 \mid x \neq y \text{ et } f(x) = f(y)\})$ .

5. Préciser la forme du graphe  $G_f$  pour  $f \in \{\varphi, \psi\}$ .
6. Expliciter la forme des différentes classes d'équivalence dans le graphe  $G$  correspondant.



## Chapitre 25

# (CCINP) Fonctions pour la détection de motifs \* (CCINP 24, ex B, corrigé (R) — oral - 131 lignes)

\*

Algorithmique des textes, Langages, Ocaml,  
sources : langccinp2.tex

On cherche à déterminer lorsqu'une certaine chaîne de caractères respecte (on dira qu'elle est capturée) un certain motif. On représente les motifs sous forme de liste de caractères pouvant contenir les caractères  $a$ ,  $b$ ,  $*$  et  $?$ , et on cherche à déterminer si une chaîne de caractères de  $\Sigma^*$ , avec  $\Sigma = \{a, b\}$  est capturée par un motif, avec les conditions suivantes :

- $?$  peut correspondre à un caractère quelconque
  - $*$  peut représenter n'importe quel mot de  $\sigma^*$ , y compris le mot vide.
1. Déterminer lesquels de ces mots sont capturés par le motif  $m_0 = *ab * a?$ .
    - (a) *abbaba*
    - (b) *abbaab*
    - (c) *bbbaab*
    - (d) *bbabaab*

Décrire en français les mots capturés par ce motif.

On représente les chaînes par le type `type chaine = char list`.

2. Écrire une fonction `string_to_chaine : string -> chaine` prenant en entrée une chaîne de caractères et renvoyant la liste de caractères associée.
3. Expliquer pourquoi, dans notre étude, il est inutile d'avoir plusieurs caractères  $*$  consécutifs.
4. Écrire une fonction `purge : chaine -> chaine` permettant de supprimer les  $*$  en trop dans une chaîne.

On propose une approche récursive naïve afin de déterminer si un motif capture une chaîne, dont le code est partiellement comme tel :

```
1 let rec capture motif texte = match (motif, texte) with
2   | [], [] -> true
3   | ['*'], [] -> true
4   | _, [] -> false
5   | [], _ -> false
6   | ...
```

5. Montrer que le code partiel de cette approche naïve est correct.
6. Compléter la fonction `capture`.



## Chapitre 26

# (CCINP) HORNSAT \* (CCINP 24, ex B, corrigé (R) — oral - 186 lignes)

\*

Logique propositionnelle, Complexité, Réduction,  
sources : ccinphornsat.tex

On attend un style de programmation fonctionnel. L'utilisation des fonctions du module `List` est autorisée; celle des fonctions du module `Option` est interdite.

Une formule du calcul propositionnel est une *formule de Horn* s'il s'agit d'une formule sous forme normale conjonctive (FNC) dans laquelle chaque clause (éventuellement vide, auquel cas la clause en question est la disjonction d'un ensemble vide de littéraux et est donc sémantiquement équivalente à  $\perp$ ) contient au plus un littéral positif. Dans la suite, on considère qu'une clause d'une telle formule contient au plus une occurrence de chaque variable (en particulier, les clauses sont sans doublons).

1. Les formules suivantes sont-elles des formules de Horn ?

- $F_1 = (\neg x_0 \vee \neg x_1 \vee \neg x_3) \wedge (x_0 \vee \neg x_1) \wedge (\neg x_2 \vee x_0 \vee \neg x_3) \wedge (\neg x_0 \vee \neg x_3 \vee x_2) \wedge x_2 \wedge (\neg x_3 \vee \neg x_2)$ .
- $F_2 = (x_0 \vee \neg x_1) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_0) \wedge \neg x_1 \wedge (x_1 \vee \neg x_1 \vee x_0) \wedge (\neg x_0 \vee x_2)$ .
- $F_3 = (\neg x_1 \vee \neg x_4) \wedge x_1 \wedge (\neg x_0 \vee \neg x_3 \vee \neg x_4) \wedge (x_0 \vee \neg x_1) \wedge (x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_4 \vee \neg x_0 \vee \neg x_1)$ .

On utilise le type suivant pour manipuler les formules de Horn : une formule de Horn est une liste de clauses de Horn ; une clause étant la donnée d'un `int option` valant `None` si la clause ne contient pas de littéral positif et `Some i` si  $x_i$  en est l'unique littéral positif et d'une liste d'entiers correspondants aux numéros des variables intervenant dans les littéraux négatifs.

```
type clause_horn = int option * int list
type formule_horn = clause_horn list
```

2. Écrire une fonction `avoir_clause_vide : formule_horn -> bool` qui renvoie `true` si et seulement si la formule en entrée contient une clause vide (donc ne contenant ni littéral positif, ni aucun littéral négatif).

On appelle *clause unitaire* une clause réduite à un littéral positif. Par ailleurs, *propager* une variable  $x_i$  dans une formule  $F$  sous FNC consiste à modifier  $F$  comme suit :

- Toute clause de  $F$  qui ne fait pas intervenir la variable  $x_i$  est conservée telle quelle.
- Toute clause de  $F$  qui fait intervenir le littéral  $x_i$  est supprimée entièrement.
- On supprime le littéral  $\neg x_i$  de toutes les clauses de  $F$  qui font intervenir ce littéral.

On souligne que supprimer  $\neg x$  d'une clause  $C$  qui ne fait intervenir que ce littéral ne revient pas à supprimer la clause  $C$ . On s'intéresse à l'algorithme  $\mathcal{A}$  suivant dont on admet (pour le moment) qu'il permet de déterminer si une formule de Horn  $F$  est satisfiable :

```
1 TantQu'il y a une clause unitaire  $x_i$  dans  $F$ 
2    $F \leftarrow$  propager  $x_i$  dans  $F$ 
3 Si  $F$  contient une clause vide
4   Renvoyer faux
5 Sinon
6   Renvoyer vrai
```

3. À l'aide de cet algorithme déterminer si les formules de Horn de la question 1 sont satisfiables. On utilisera ces formules pour tester les fonctions implémentées aux questions suivantes.

4. Écrire une fonction `trouver_clause_unitaire : formule_horn -> int option` renvoyant `None` si la formule en entrée n'a pas de clause unitaire et `Some i` où  $x_i$  est l'une des clauses unitaires sinon.

5. Justifier que propager une variable dans une formule de Horn donne une formule de Horn. Écrire une fonction `propager : formule_horn -> int -> formule_horn` qui prend en entrée une formule de Horn  $F$  et un entier  $i$  et calcule la formule résultat de la propagation de  $x_i$  dans  $F$ .
6. Dédire des questions précédentes une fonction `etre_satisfiable : formule_horn -> bool` renvoyant `true` si et seulement si la formule de Horn en entrée est satisfiable.
7. Quelle est la complexité de votre algorithme en fonction de la taille de la formule en entrée? Que peut-on dire des problèmes de décision **SAT** et **HORN-SAT** (dont la définition est la même que celle de **SAT** à ceci près que les formules considérées sont supposées être des formules de Horn)?
8. On s'intéresse à présent à la correction de l'algorithme  $\mathcal{A}$ .
  - a) Si  $F$  est une clause de Horn sans clause unitaire ni clause vide, donner une valuation simple qui satisfait  $F$ .
  - b) On admet que si  $F$  est une formule de Horn faisant intervenir une clause unitaire  $x_i$  et  $F'$  est le résultat de la propagation de  $x_i$  dans  $F$ , alors que  $F$  est satisfiable si et seulement si  $F'$  est satisfiable. En déduire la correction de l'algorithme  $\mathcal{A}$ .
9. Expliquer comment on pourrait modifier les fonctions précédentes afin de déterminer une valuation satisfaisant une formule de Horn dans le cas où elle existe plutôt que de juste dire si elle est satisfiable ou non. On ne demande pas d'implémentation.

## Chapitre 27

# (CCINP) Mots de Dyck \* (CCINP 24, ex B, corrigé — oral - 157 lignes)

\*

Langages, C, Complexité,  
sources : `ccinpydyck.tex`

### Mots de Dyck (type B)

*La compilation du code compagnon initial avec `make safe` provoque des warnings attendus qui seront résolus lors de l'implémentation des fonctions demandées par le sujet.*

On s'intéresse dans cet exercice aux mots de Dyck, c'est-à-dire aux mots bien parenthésés. Dans ce type de mots, toute parenthèse ouverte ("`"` est fermée `"`)" et une parenthèse ne peut être fermée si elle ne correspond pas à une parenthèse préalablement ouverte.

Par exemple pour deux couples de parenthèses, `"(())"` et `"()()"` sont des chaînes de parenthèses bien formées. `"())"` et `"()("` ne le sont pas.

On admet que le nombre de mots bien parenthésés à  $n$  couples de parenthèses est donné par les nombres de Catalan définis par la formule suivante :

$$C_n = \frac{(2n)!}{(n+1)!n!} \text{ pour } n \geq 0$$

On rappelle que le type `uint64_t` est un type entier non signé codé sur 64 bits.

1. Complétez dans le code compagnon la fonction dont le prototype est `uint64_t catalan(int n)`. Vous pouvez utiliser une fonction auxiliaire si cela vous semble pertinent.
2. Que va-t-il se passer si on tente d'afficher `catalan(n)` pour  $n$  un peu grand ? Le constatez-vous ici ?

On cherche maintenant à afficher le nombre de mots (chaînes) bien parenthésés avec  $n$  fixé couples de parenthèses, ainsi que les mots eux-mêmes.

Un algorithme de force brute pour déterminer toutes les chaînes à  $n$  couples de parenthèses bien formées consiste à générer toutes les possibilités puis à ne garder que les chaînes bien formées.

3. Complétez dans le code compagnon la fonction dont le prototype est `bool verification(char * mot)`. Cette fonction renvoie `true` si le mot fourni en paramètre `mot` est bien parenthésé, `false` sinon.
4. Quelle est la complexité de cette vérification ?
5. Quelle est la complexité finale de l'algorithme de force brute ?

On appelle  $n$  le nombre de couples de parenthèses voulu. Dans le fichier compagnon fourni, le nombre de couples a été limité à 18.

On vous propose de coder l'énumération des chaînes de parenthèses bien formées en appliquant l'algorithme de backtracking suivant, dont on admet qu'il est correct : on compte le nombre de parenthèses ouvertes  $o$  et le nombre de parenthèses fermées  $f$  dans une chaîne de caractères courante (vide au départ).

- Si  $o = f = n$ , on a trouvé une chaîne bien formée.
- Si  $o < n$ , on ajoute une parenthèse ouvrante et on relance.
- Si  $f < o$ , on ajoute une parenthèse fermante et on relance.

Cet algorithme est à implémenter dans la fonction dont le prototype est `void dyck(char s[N], int o, int f, int n)` qui affiche sur la sortie standard les chaînes de parenthèses bien formées avec  $n$  couples de parenthèses lorsque  $s$  est la chaîne de caractère courante,  $o$  est son nombre de parenthèses ouvrantes et  $f$  est son nombre de parenthèses fermantes.

6. Compléter la fonction `dyck` pour afficher les chaînes bien parenthésées avec 5 couples de parenthèses.
7. Adapter la fonction `dyck` pour calculer le nombre de mots obtenus. Combien de mots trouvez-vous pour 16 couples de parenthèses ?
8. Adapter la fonction `dyck` pour stocker les mots bien parenthésés dans une liste chaînée et les afficher après l'appel à la fonction. Vous trouverez dans le code compagnon une structure qui peut vous aider.



## Chapitre 28

# (CCINP) Complétude de Nor et d'une variante Nor \* (CCINP 25, ex 1, retour élève modifié, corrigé — oral - 150 lignes)

\*

Logique propositionnelle,  
sources : ccinplog.tex

On se donne un ensemble de variables  $\mathcal{V}$  (avec  $|\mathcal{V}| \geq 2$ ).

On considère l'ensemble  $\mathcal{F}$  des formules propositionnelles défini par induction :

- $\perp \in \mathcal{F}$
- $\top \in \mathcal{F}$
- $\forall v \in \mathcal{V}, v \in \mathcal{F}$
- $\forall f \in \mathcal{F}, \neg f \in \mathcal{F}$
- $\forall (f_1, f_2) \in \mathcal{F}^2, f_1 \wedge f_2 \in \mathcal{F}$
- $\forall (f_1, f_2) \in \mathcal{F}^2, f_1 \vee f_2 \in \mathcal{F}$

On définit l'opérateur  $\downarrow$  (le *Nor*) par la table de vérité suivante :

$a$	$b$	$a \downarrow b$
0	0	1
1	0	0
0	1	0
1	1	0

1. Donner la table de vérité de  $(a \downarrow a)$ . À quel connecteur est-ce équivalent ?
2. Montrer que  $(a \wedge b) \equiv (a \downarrow a) \downarrow (b \downarrow b)$ .
3. Montrer que  $\{\downarrow\}$  est complet – i.e. que toute formule de  $\mathcal{F}$  est équivalente à une formule n'utilisant que le connecteur  $\downarrow$ .  
On raisonnera par induction.

On définit un nouvel opérateur, % *NOR*, noté  $\odot$ , par la table de vérité suivante.

$a$	$b$	$a \odot b$
0	0	1
1	0	0
0	1	0
1	1	1

4. Montrer que les formules  $f = \perp$ ,  $f = \top$  et  $f = v \in \mathcal{V}$  sont satisfaites pour un nombre pair de valuations.
5. On considère  $F = G \odot H$ , où  $G, H \in \mathcal{F}$ . On note  $n_{0,0}$  le nombre de valuations qui ne satisfont ni  $G$  ni  $H$ ,  $n_{0,1}$  le nombre de valuations qui ne satisfont pas  $G$  et satisfont  $H$  et l'on définit de même  $n_{1,0}$  et  $n_{1,1}$ .  
On suppose que  $G$  et  $H$  sont satisfaites par un nombre pair de valuations.  
Montrer que  $F$  est satisfait par un nombre pair de valuations.
6.  $\{\odot\}$  est-il complet ?



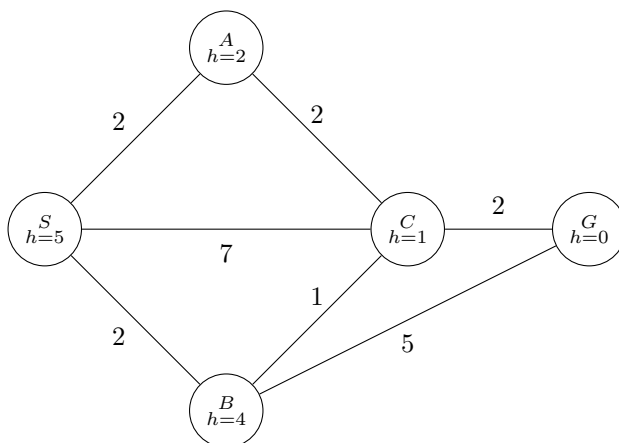
# Chapitre 29

## (CCINP) Graphes, A\* \* (CCINP 25, ex 1, retour élève très incomplet complété, corrigé — oral - 116 lignes)

\*

Graphes,  
sources : ccinpgraphe2.tex

On considère le graphe  $G_0 = (S, A)$  non orienté pondéré par des poids  $l$  ci-dessous :



Les poids figurent sur les arêtes, et les entiers écrits sur les sommets correspondent à une heuristique  $h$ . On définit par ailleurs l'heuristique  $h_2$  comme étant l'heuristique à vol d'oiseau – i.e. la distance euclidienne entre tout sommet et le sommet  $G$  (les sommets étant vus comme des points de  $\mathbb{R}^2$ ).

1. Appliquer  $A^*$  avec l'heuristique  $h$ , avec comme sommet de départ  $S$ , et sommet d'arrivée  $G$ , le résultat est-il optimal ?
2. Que ce serait-il passé si on avait utilisé Dijkstra ? Il n'est pas demandé d'appliquer l'algorithme.

On dit qu'une heuristique est

- *admissible* si

$$\forall s \in S, h(s) \leq d(s, G)$$

- *monotone* si

$$\forall (s, s') \in A, h(s) \leq l(ss') + h(s')$$

3. Prouver que  $h_2$  est monotone.
4. L'heuristique  $h$  est-elle admissible ? monotone ?
5. L'algorithme  $A^*$  donne-t-il toujours le bon résultat avec l'heuristique  $h_2$  ?
6. On utilise désormais  $h_1 : (s, t) = \|s - t\|_1$  (où pour tout  $(x, y \in \mathbb{R}^2)$ ,  $\|(x, y)\|_1 = |x| + |y|$ ). L'algorithme  $A^*$  utilisant l'heuristique  $h_1$  est-il correct ?



## Chapitre 30

# (CCINP) Autour d'un jeu d'alignement \*

## (CCINP 25, ex 2, retour élève complété, corrigé — oral - 201 lignes)

\*

Ocaml, Algorithmique, Jeux,  
sources : `ccinpalgo.tex`

On se donne  $n \geq 2$  – que l'on supposera défini comme variable globale.

On considère un jeu à deux joueurs qui se joue sur le plateau carré de taille  $n \times n$ . Chaque joueur dispose de jetons en nombre illimité, et à son tour, pose l'un de ses jetons sur une case libre.

Un joueur gagne immédiatement s'il parvient à créer une ligne ou une colonne entièrement constituée par ses jetons (on parle d'*alignement*).

S'il n'y a pas d'alignement et que le plateau est rempli, le joueur qui n'a pas commencé a gagné.

Le plateau sera représenté par le type suivant :

```
type plateau = int array array
```

avec la convention suivante pour les cases :

- 0 désigne une case vide ;
- 1 désigne une case contenant un jeton du joueur 1 ;
- 2 désigne une case contenant un jeton du joueur 2.

La case en haut à gauche est d'indice (0,0).

Chaque case sera représentée par un unique identifiant entier, avec une numérotation ligne après ligne.

1. Écrire une fonction `identifiant : int -> int -> int` qui prend en entrée un indice de ligne et un indice de colonne et renvoie l'identifiant de la case sur le plateau.
2. La réciproque `coordonnees : int -> (int*int)`.
3. Coder une fonction `legal` qui prend en entrée un plateau, `l` et `c`, et renvoie un `bool` pour dire si le coup est légal.
4. Coder une fonction `joue`, prenant en entrée un plateau, une ligne, une colonne et un joueur, et renvoie en sortie un nouveau plateau (distinct du plateau en entrée en mémoire). La fonction devra lever une exception si le coup est illégal.
5. Coder une fonction `victoire : plateau -> int option` qui prend en argument un plateau et détecte s'il y a une victoire d'un des deux joueurs  $j$  (auquel cas `Some j` est renvoyé).

On donne une fonction `coups_possibles : plateau -> int list` qui renvoie la liste des coups possibles (sous forme d'une liste d'identifiants de cases).

6. Coder une fonction `gagnante : plateau -> int -> int option` décidant si la position est gagnante pour l'un des joueurs (le second argument indiquant le joueur au trait). On pourra utiliser les fonctions dont les signatures et spécification sont indiquées en annexe.

### Annexe

```
List.map : ('a -> 'b) -> 'a list -> 'b list
```

`map f [a1 ; ... ; an]` applique la fonction `f` à `a1`, ..., `an`, et renvoie la liste `[f a1 ; ... ; f an]`.

```
List.exists : ('a -> bool) -> 'a list -> bool
```

`exists f [a1 ; ... ; an]` vérifie si au moins l'un des éléments de la liste satisfait le prédicat `f`.

```
List.for_all : ('a -> bool) -> 'a list -> bool
```

`for_all f [a1 ; ... ; an]` vérifie si tous les éléments de la liste satisfont le prédicat `f`.

# Chapitre 31

## (CCINP) Graphes bipartis \* (CCINP 25, ex 2, retour élève complété, corrigé — oral - 136 lignes)

\*

Graphes, Ocaml,  
sources : ccinpgraphe3.tex

On définit en *OCaml* les types suivants :

```
type sommet = int
type graphe = sommet list array (*représentation par liste d'adjacences d'un ↔
    graphe dont les sommets sont numérotés de 0 à n-1*)
type graphe_biparti = {
  graphe : graphe;
  e1 : bool array; (*e1.(i) ssi i appartient à e1*)
  e2 : bool array; (*e2.(i) ssi i appartient à e2*)
}
```

1. Écrire une fonction `xor` : `bool -> bool -> bool` qui calcule le ou exclusif de deux booléens passés en argument.
2. Écrire une fonction `bipartition` : `bool array -> bool array -> bool` telle que, si `e1` et `e2` sont de même longueur `n`, `bipartition e1 e2` renvoie vrai si et seulement si  $e1 \cup e2 = \llbracket 0, n - 1 \rrbracket$  et  $e1 \cap e2 = \emptyset$ .
3. Écrire une fonction `bipartition2` : `graphe_biparti -> sommet -> sommet list -> bool` telle que `bipartition2 g i v` renvoie vrai ssi `i` est dans l'une des deux parties `e1` ou `e2` de `g`, et que tous les éléments de `v` sont dans l'autre partie.
4. Écrire une fonction `biparti` : `graphe_biparti -> bool` qui teste si le graphe passé en argument est biparti pour la partition de sommets `(e1, e2)`.

Dans la suite, nous allons tester si un graphe est biparti en partant d'ensembles `e1` et `e2` initialement vides, et en les remplissant en parcourant le graphe.

5. Écrire une fonction `affecte_voisins` : `graphe_biparti -> sommet -> sommt list` qui étant donné un graphe et un sommet `i` qui est dans l'une des parties `e1` ou `e2`, et met les voisins de `i` dans l'autre partie s'ils n'y sont pas déjà. Si un voisin de `i` était déjà dans dans la même partie que `i`, elle lèvera une exception `NonBiparti`. Elle renverra la liste des voisins qui ont été mis dans l'autre partie.
6. Écrire une fonction `parcours` : `graphe_biparti -> sommet -> unit` parcourt le graphe à partir du sommet en affectant les sommets rencontrés à l'une ou l'autre classe. Elle lèvera une exception `NonBiparti` si c'est impossible.
7. Écrire une fonction `calcule_bipartition` : `graphe -> graphe_biparti` qui étant donné un graphe, calcule une bipartition de ce graphe si elle existe, et lève un exception sinon.



## Chapitre 32

# (CCINP) Sous-mots \* (CCINP 25, ex 2, retour élève très modifié, corrigé — oral - 155 lignes)

\*

Algorithmique, C,  
sources : `ccinpsousmots.tex`

Certains systèmes de saisie permettent d'écrire un mot en faisant glisser son doigt sur un clavier. Le doigt passant au dessus de touches qui n'appartiennent pas au mot souhaité, cela nécessite d'être capable de trouver les sous-mots valides d'un mot donné.

On fournit des structures de données et des fonctions dans les fichiers `sousmots_auxiliaire.h` et `sousmots_auxiliaire.c`.

1. Écrire une fonction `void affiche_mot(char* mot, int n)` qui affiche un mot de longueur  $n$ .
2. Écrire une fonction `void affiche_liste(liste lst)` qui affiche le contenu d'une liste de mots.
3. Écrire une fonction `void ajoute(char a, liste lst)` qui ajoute le caractère `a` à tous les éléments de la liste en début de mot.  
On admet qu'un espace mémoire suffisant est alloué au champ `mot` de chaque maillon.
4. Justifier que la fonction précédente a une complexité linéaire en le nombre de mots dans la liste.
5. Justifier brièvement que le nombre de sous-mots d'un mot est exponentiel en la taille du mot dans le cas général. Est-ce toujours le cas ?
6. Écrire une fonction `liste copie_liste(liste l)` qui renvoie une copie de la liste passée en argument. La liste de départ et la liste renvoyée ne devront pas partager les mêmes mots en mémoire.
7. Écrire une fonction `liste concat(liste l1, liste l2)` qui prend deux listes en argument et renvoie leur conténation. On attend une complexité linéaire en la longueur de  $l_1$ , et  $l_1$  pourra être modifiée par la fonction.
8. En utilisant une stratégie de retour sur trace, écrire une fonction `liste sous_mots(char* mot, int n)` qui renvoie la liste des sous-mots du mot passé en argument.
9. Prouver la correction de la fonction précédente.



## Chapitre 33

# (CCINP) Mots sur un alphabet muni d'un ordre \*\* (CCINP 25, ex A, retour élève modifié, corrigé — oral - 171 lignes)

\*\*

Langages, Automates,  
sources : `ccinplang.tex`

Soit  $\Sigma$  un alphabet et  $\leq$  un ordre sur  $\Sigma$  – pas nécessairement total.

On définit l'ordre  $\sqsubseteq$  de la façon suivante : pour deux mots  $w_1$  et  $w_2 \in \Sigma^*$ ,  $w_1 \sqsubseteq w_2$  si et seulement si  $w_1$  est préfixe de  $w_2$ .

On définit alors  $\preceq$  l'ordre lexicographique construit à l'aide de  $\leq$  et  $\sqsubseteq$ .

1. Montrer que si  $|\Sigma| > 1$ , alors  $\sqsubseteq$  est un ordre non total.
2. Montrer que si  $\leq$  est un ordre non total, alors  $\preceq$  aussi.
3. Montrer que  $\sqsubseteq$  est bien fondé.
4. Montrer que  $\preceq$  n'est pas bien fondé.

Étant donné un langage  $L$ , on définit  $\text{Pref}(L)$  comme l'ensemble des préfixes des mots de  $L$ .

5. Montrer que si  $L$  est rationnel, alors  $\text{Pref}(L)$  l'est aussi.

Étant donné un langage  $L$ , on définit  $\text{min}(L)$  comme l'ensemble

$$\text{min}(L) = \{u \in L \mid \forall v \in L, v \preceq u \Rightarrow u = v\}.$$

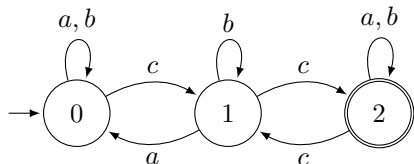
On suppose  $L$  rationnel, reconnu par un AFD émondé  $\mathcal{A} = (\Sigma, Q, q_0, F, \Delta)$ .

On construit  $\mathcal{A}'$  à partir de  $\mathcal{A}$  en

- (i) retirant tous les arcs sortants des états terminaux
- (ii) pour tout état  $q \in Q$ , ne gardant que les arcs sortants ayant une étiquette  $a \in \Sigma$  telle que pour tout arc sortant de  $q$  d'étiquette  $b \in \Sigma$ , on n'ait pas  $b < a$ .

Soit  $\Sigma = \{a, b, c\}$ , où  $a \leq b$ , et  $c$  n'est comparable ni à  $a$  ni à  $b$ .

6. Trouver  $\text{min}(\{ab, abac, aca, ca, cb, baa\})$ .
7. Appliquer la construction précédente à l'automate suivant :



8. Expliquer pourquoi, dans le cas général,  $\text{min}(L) = \text{rec}(\mathcal{A}')$ .



## Chapitre 34

# (CCINP) $k$ plus proches voisins \* (CCINP 25, ex A, retour élève modifié, figure refaite, corrigé — oral - 108 lignes)

\*

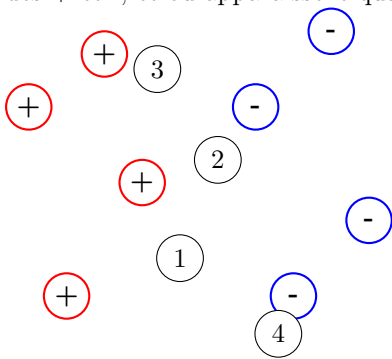
Ia,  
sources : ccinpia2.tex

On considère des escargots de course, que l'on représentera par leur taille et leur masse.

On cherche à savoir s'ils sont dopés ou non.

1. De quel type de problème d'apprentissage s'agit-il ?
2. Est-ce-que l'apprentissage par classification hiérarchique conviendrait pour répondre à ce problème ? Justifier brièvement.
3. On désire implémenter l'algorithme des  $k$ -plus proches voisins. Rappeler le principe de cet algorithme, et décrire comment on l'utiliserait.

On se donne le jeu de données suivant, où les escargots dopés et non-dopés de la base d'apprentissage sont représentés par des + et -, et où apparaissent quatre escargots auxquels appliquer l'algorithme :



4. Pour chaque escargot de 1 à 4, peut-on estimer qu'il est dopé en utilisant l'algorithme pour  $k = 3$  ?
5. Rappeler la complexité dans le cas général de l'algorithme.
6. Peut-on améliorer cette complexité avec une nouvelle structure de données ?
7. Dresser la matrice de confusion des données ci-dessus dans le cas  $k = 3$ .
8. Peut-on affirmer à 70% que notre algorithme est bon pour  $k = 3$  ?



## Chapitre 35

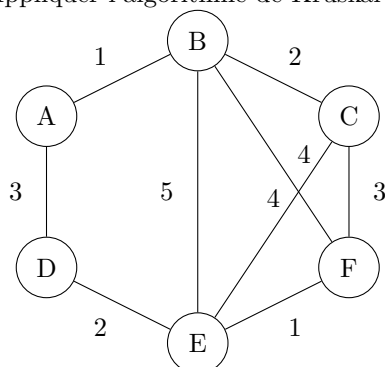
# (CCINP) UF et Kruskal \* (CCINP 25, ex A, retour élève très incomplet reconstitué, corrigé — oral - 116 lignes)

\*

Graphes, Algorithmique, C,  
sources : ccinpalgo6.tex

On s'intéresse à la structure abstraite *Unir et Trouver*, permettant de représenter une classe d'équivalence d'un ensemble fini  $E$ , et devant être munies des deux opérations suivantes :

- *Unir*, qui prend en argument deux éléments de  $E$  et fusionne les classes de ces deux éléments ;
  - *Trouver*, qui prend en argument un élément de  $E$  et renvoie la classe à laquelle il appartient.
1. Rappeler brièvement le principe de l'algorithme de Kruskal, et indiquer en particulier à quel endroit il utilise une structure *Unir et Trouver*.
  2. Appliquer l'algorithme de Kruskal au graphe suivant :



On implémente une structure *Unir et Trouver* à l'aide d'une forêt – i.e. un ensemble d'arbres enracinés – chaque arbre correspondant à une classe d'équivalence.

3. Quelle est la forêt correspondant à la relation d'équivalence  $\ll = \gg$  (égalité) sur l'ensemble  $E = \llbracket 0, 5 \rrbracket$  ?
4. En partant de la forêt précédente, appliquer successivement les opérations  $\text{Unir}(0,1)$ ,  $\text{Unir}(2,3)$ ,  $\text{Unir}(5,4)$ ,  $\text{Unir}(0,3)$ , et indiquer les classes d'équivalences obtenues.
5. Comment implémenter efficacement une telle forêt en *C* ou en *OCaml* ?
6. Donner le pseudo-code de l'opération *Unir*.
7. Donner le pseudo-code de l'opération *Trouver*.



## Chapitre 36

# (CCINP) Algorithmes du cours sur les automates, complexités \*\* (CCINP 25, ex A, retour élève, corrigé — oral - 232 lignes)

\*\*

Langages, Automates,  
sources : `ccinplang3.tex`

On considère le langage  $L$  dénoté par l'expression régulière  $e_0 = (ab)^*(\epsilon|a)$ .

1. Appliquer l'algorithme de Berry-Sethi pour obtenir un automate  $\mathcal{A}_{e_0}$  reconnaissant ce langage.
2. Quel est la complexité de l'algorithme de Berry-Sethi en fonction de  $n = |e|$  ?
3. Déterminer  $\mathcal{A}_{e_0}$ .
4. Appliquer l'algorithme de Thompson à  $e_0$  pour obtenir un nouvel automate reconnaissant ce langage.
5. Discuter sa complexité.



## Chapitre 37

# (CCINP) Bin packing \* (CCINP 25, ex A, retour élève, corrigé — oral - 143 lignes)

\*

Algorithmes d'approximation, Complexité,  
sources : ccinpbinpack.tex

Le problème *BIN PACKING* consiste, étant donné un volume  $V$  et un ensemble de  $n$  objet ayant chacun un certain volume à déterminer le nombre minimiser le nombre de boîtes permettant de contenir tous les objets (en supposant ceux-ci déformables!).

On supposera dans la suite que chaque objet a un volume inférieur ou égal à  $V$ .

On se donne l'instance suivante de ce problème :

$$V = 10, n = 7, (v_i)_{i \in \llbracket 0,6 \rrbracket} = (2, 5, 4, 7, 1, 5, 8)$$

**Algorithme**<sub>1</sub>( $V, n, (v_i)$ ) :

```
k ← 0
Vk ← 0
i ← 0
TantQue i < n :
  Si Vk + vi ≤ V :
    ni ← k
    Vk ← Vk + vi
    i ++
  Sinon :
    k ++
    Vk ← 0
Renvoyer k + 1
```

1. Que représentent les  $n_i$  dans cet algorithme ?
2. Quel est le  $k$  renvoyé par l'algorithme sur l'instance donnée en exemple, et quelle est la répartition correspondante des objets ?
3. On note  $k^*$  la solution optimale pour une instance quelconque. Montrer que

$$k^*V \geq \sum_{i=0}^{n-1} v_i.$$

4. On note  $k$  la solution renvoyée par l'algorithme. Montrer que

$$\sum_{i=0}^{n-1} v_i \geq V(k-1)/2.$$

*Indication : étudier le volume des objets de deux boîtes consécutives.*

5. Rappeler ce qu'est une  $\alpha$ -approximation, et montrer que l'algorithme ci-dessus en est une, en précisant la valeur de  $\alpha$ .

On se donne maintenant l'algorithme suivant :

**Algorithme**<sub>2</sub>( $V, n, (v_i)$ ) :

```
k ← 1
Vk ← 0
Pour i variant de 0 à n - 1 : (*)
```

Pour  $j$  variant de 1 à  $k$  :

Si  $V_j + v_i \leq V$  :

$n_i \leftarrow j$

$V_j \leftarrow V_j + v_i$

Passer à l'itération suivante de la boucle (\*)

$k++$

$n_i \leftarrow k$

$V_k \leftarrow v_i$

Renvoyer  $k+1$

6. Quel est le  $k$  renvoyé par ce nouvel algorithme sur l'instance donnée en exemple, et quelle est la répartition des objets correspondante ?
7. Indiquer les complexité temporelles dans le pire des cas des deux algorithmes.
8. Donner une idée naturelle permettant d'améliorer l'algorithme 2. Indiquer quels changements cela produirait sur la complexité.

## Chapitre 38

# (CCINP) Jeu sur un graphe \* (CCINP 25, ex A, retour élève, corrigé — oral - 120 lignes)

\*

Graphes, Jeux,  
sources : ccinpjeu1.tex

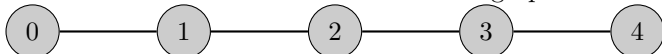
Soit  $G = (S, A)$  un graphe non orienté, connexe.

On s'intéresse à un jeu à deux joueurs : R (rouge) et B (bleu).

R commence en choisissant un sommet et en le coloriant en rouge. Puis chacun à son tour, le joueur choisit un sommet non colorié voisin d'un sommet de la couleur de l'adversaire mais non voisin d'un sommet de sa propre couleur, puis colorie ce sommet de sa couleur. Si aucun sommet n'est jouable, le joueur passe son tour.

Quand aucun joueur ne peut jouer, la partie est terminée.

- Si le graphe est entièrement colorié, R gagne.
- Sinon, B gagne.
  1. Donner la définition d'une stratégie.
  2. Donner la définition d'une stratégie gagnante.
  3. Donner l'ensemble des états finaux sur le graphe suivant – en supposant que R commence par colorier 1.



4. Montrer que s'il existe une partie gagnée par R, alors  $G$  est biparti.

On suppose maintenant que  $G$  est biparti. On note  $(S_1, S_2)$  une bipartition des sommets, et l'on suppose que le premier sommet joué par R est dans  $S_1$ .

5. Montrer l'invariant suivant : «Tout sommet colorié en rouge est dans  $S_1$ , et tout sommet colorié en bleu est dans  $S_2$ .»
6. Montrer que s'il reste des sommets non coloriés, alors au moins l'un d'entre eux est coloriable.
7. En déduire ( $E$ ) :  $G$  est biparti si et seulement si toutes les stratégies de R sont gagnantes.
8. En déduire un algorithme déterminant si un graphe connexe non orienté est biparti.



## Chapitre 39

# (CCINP) Mots conjugués \* (CCINP 25, ex A, retour élève, corrigé — oral - 110 lignes)

\*

Langages, Automates, Lemme de l'étoile,

sources : `ccinplang2.tex`

Soit  $V$  un alphabet, et  $x$  et  $z \in V^*$ .

On dit que  $x$  est *conjugué* avec  $z$ , et l'on note  $xCz$  si et seulement si

$$\exists y \in V^*, xy = yz.$$

1. Soient  $x = 001$ ,  $z_1 = 010$  et  $z_2 = 01$ . Montrer que  $x$  est conjugué avec  $z_1$  mais pas avec  $z_2$ .
2. Donner l'ensemble des mots  $z$  avec lesquels  $x = \epsilon$  est conjugué.
3. Montrer que s'il existe  $u$  et  $v \in V^*$  tels que  $x = uv$  et  $z = vu$ , alors  $x$  et  $z$  sont conjugués.
4. Supposons  $x \neq \epsilon$ . Montrer que si  $xCz$ , alors le plus petit  $y$  (en nombre de lettres) tel que  $xy = yz$  vérifie  $|x| \geq |y|$ .
5. En déduire que

$$xCz \Leftrightarrow \exists u, v \in V^*, \begin{cases} x = uv \\ z = vu \end{cases}.$$

6. Montrer que  $C$  est une relation d'équivalence.

On notera dans la suite  $C(x)$  la classe d'équivalence de  $x$ .

7.  $C(x)$  est-il régulier ?

Soit  $L$  un langage. On définit  $C(L) = \bigcup_{x \in L} C(x)$ .

8. Montrer que  $C(L)$  peut ne pas être régulier. *Indication* : on pourra considérer un langage  $L$  sur un alphabet à une seule lettre.
9. Montrer que si  $L$  est régulier,  $C(L)$  l'est également.



## Chapitre 40

# (CCINP) tas min \* (CCINP 25, ex A, retour élève, rallongé, corrigé — oral - 92 lignes)

\*

Algorithmique, Structures de données, Ocaml,  
sources : `ccinptas.tex`

1. Rappeler ce qu'est un *tas-min*, et comment le représenter sous forme d'arbre et sous forme de tableau.

Dans la suite, on se propose d'implémenter un tas-min à l'aide d'un tableau `t`. On se donnera également en argument des fonctions un entier `fin` indiquant que la partie du tableau `t` représentant effectivement le tas est compris entre les indices 0 et `fin-1`.

2. Écrire une fonction `est_tas_min t fin` qui teste si `t` est bien un tas-min. La tester sur les deux exemples donnés.
3. Écrire une fonction `plus_petit_fils t fin i`, qui renvoie l'indice du plus petit fils de l'élément d'indice `i` dans le tas. On supposera que cet élément a au moins un fils, et en cas d'égalité, on choisira le fils gauche.
4. Écrire une fonction `descente t fin i` qui descend l'élément d'indice `i` à sa place dans le tas.
5. Écrire une fonction `supprime t fin e` qui supprime et renvoie l'élément minimal d'un tas min (le nouvel élément inutilisé du tableau à l'issue de l'exécution pourra avoir une valeur arbitraire).  
Quelle est la complexité de cette fonction ?



# Chapitre 41

## (CCINP) Dénombrement du nombre de façons d'obtenir un score avec des dés. \*

### (CCINP 25, ex B, retour élève complété, corrigé — oral - 178 lignes)

\*

Algorithmique, C,  
sources : ccinpalgo3.tex

Cet exercice est à traiter en langage C.

On cherche dans cet exercice à déterminer de combien de façons on peut obtenir un score  $s \geq 0$  en lançant un nombre  $d \geq 0$  de dés distinguables à  $f \geq 1$  faces.

1. De combien de façons peut-on faire 3 avec deux dés à 2 faces ?
2. Écrire une fonction `int** init_matrice(int n, int p, int v)` qui renvoie une matrice `int** m` telle que `m[i-1][j-1]` soit égal à  $[M]_{i,j}$ , où  $M \in \mathcal{M}_{n,p}(\mathbb{N})$  est la matrice dont tous les éléments sont égaux à  $v$ .
3. Écrire une fonction `void libere_mat(int** m, int n, int p)` qui prend en argument une matrice comme celle renvoyée à la question précédente, et libère l'espace qui lui est alloué.

On note  $t_{i,j}$  le nombre de manières dont on peut obtenir le score  $j$  en lançant  $i$  dés à  $f$  faces. Par convention,  $t_{0,0} = 1$ .

4. Que vaut  $t_{0,j}$  lorsque  $j > 0$  ?
5. Montrer que  $t_{i,j} = \sum_{k=1}^{\min(j,f)} t_{i-1,j-k}$  pour tous  $i$  et  $j$  à préciser.
6. Implémenter récursivement la fonction `int nb_possibilites(int i, int j, int f)` qui renvoie  $t_{i,j}$ . Tester avec  $(d, s, f) = (6, 12, 6)$  puis  $(d, s, f) = (12, 42, 20)$ . Commenter les résultats obtenus.
7. Implémenter une `int nb_possibilites_memo_asc(int i, int j, int f)` résolvant le même problème en utilisant une mémoïsation et une implémentation ascendante.
8. On dispose dans les fichiers `structuresMPI.h` et `structuresMPI.c` d'une structure de table de hachage, dont la documentation est donnée en annexe.  
Écrire une fonction `int nb_possibilites_memo_desc(int i, int j, int f)` utilisant une implémentation descendante avec mémoïsation par table de hachage pour résoudre le problème.

### Documentation pour les tables de hachage

On donne l'extrait du fichier `structuresMPI.h` contenant l'interface d'une structure de table de hachage – où les clés et les valeurs sont des entiers non signés sur 64 bits (de type `uint64_t`).

```
typedef struct hashtable hashtable;

//créé une nouvelle table de hachage de capacité indiquée
hashtable *hashtable_new();

//libère la mémoire occupée par une table de hachage
void hashtable_free(hashtable *t);

//ajoute une clé k avec une valeur v à la table - **sans vérifier si elle est déjà
à présente**.
void hashtable_add(hashtable *t, uint64_t k, uint64_t v);
```

```
//teste si une clé k est présente dans la table
bool hashtab_contains(hashtab *t, uint64_t k);

//modifie à v la valeur correspondant à la clé k dans la table - **sans vérifier↔
//si elle est déjà présente**
void hashtab_mod(hashtab *t, uint64_t k, uint64_t v);

//récupère la valeur correspondant à la clé k dans la table - **sans vérifier si↔
//elle est déjà présente**
uint64_t hashtab_get(hashtab *t, uint64_t k);

//supprime la clé k et la valeur correspondante, si k est présente dans la table
void hashtab_remove(hashtab *t, uint64_t k);
```

## Chapitre 42

# (CCINP) Permutation de Langford \*

## (CCINP 25, ex B, retour élève complété, corrigé — oral - 188 lignes)

\*

Algorithmique, C,  
sources : ccinpalgo5.tex

Une *permutation de Langford* d'ordre  $n$  est une permutation du multi-ensemble  $\{0, 0, 1, 1, 2, 2, \dots, n-1, n-1\}$  telle que pour tout  $k \in \llbracket 0, n-1 \rrbracket$ , il y a exactement  $k$  nombres entre les deux occurrences de  $k$ .

On notera  $E = \{0, 0, 1, 1, 2, 2, \dots, n-1, n-1\}$ .

Par exemple,  $(3, 1, 2, 1, 3, 2, 0, 0)$  est une permutation de Langford d'ordre 4.

1. Donner un autre exemple de permutation de Langford d'ordre 4.
2. Parmi les deux permutations suivantes, laquelle est une permutation de Langford ?
  - $(3, 4, 0, 0, 3, 2, 4, 1, 2, 1)$
  - $(3, 1, 4, 1, 3, 2, 0, 0, 2, 4)$
3. Écrire une fonction `int* initialise_tableau(int n, int v)` allouant un tableau de taille  $n \geq 1$  avec des cases toutes de valeur  $v$ . Comment libérer un tel tableau ?

Une permutation de  $E$  est représentée par un tableau de taille  $2n$ .

On vérifie que c'est une permutation de Langford en trois étapes :

1. Toutes les valeurs sont dans  $\llbracket 0, n-1 \rrbracket$ .
2. C'est une permutation de  $E$ .
3. C'est une permutation de Langford.
4. Écrire une fonction `bool valeurs_valables(int n, int* t)` vérifiant l'étape 1 (en supposant  $t$  de taille  $2n$ ). Quelle est sa complexité temporelle dans le meilleur et dans le pire des cas ?
5. Vérifier que votre code passe le premier test donné.
6. Écrire une fonction `bool est_permutation(int n, int* t)` vérifiant l'étape 2. On attend une complexité linéaire en  $n$ .
7. Vérifier que votre code passe le deuxième test donné.
8. Écrire une fonction `bool est_langford(int n, int* t)` vérifiant l'étape 3. On attend une complexité linéaire en  $n$ .
9. Vérifier que votre code passe le troisième test donné.

On souhaite maintenant compter le nombre de permutations de Langford en utilisant une stratégie de retour sur trace.

Pour cela, on démarre avec

- un tableau `t` de  $2 * n$  initialisé à -1 correspondant aux permutations de Langford en cours de construction ;
  - un tableau `b` de  $n$  booléens, `b[i]` étant vrai si et seulement si  $i$  a déjà été utilisé dans le tableau `t`.
10. Écrire une fonction `int backtrack(int n, bool* b, int* t, int i)` qui compte le nombre de permutations de Langford commençant par les  $t$  premiers éléments du tableau `t`.  
*Indication* : cette fonction essaiera de mettre dans `t[i]` toutes les valeurs  $v$  possibles (ce qui permet de mettre également  $v$  dans une autre case du tableau), et s'appellera récursivement avec pour nouveau paramètre  $i$  la prochaine case encore à  $-1$  dans `t`.
  11. Écrire une fonction `int nb_langford(int n)` résolvant le problème posé.
  12. En utilisant cette fonction, établir une conjecture lorsque  $n \equiv 2[4]$  ou  $n \equiv 3[4]$ . La démontrer.



## Chapitre 43

# (CCINP) Rendu de monnaie \* (CCINP 25, ex B, retour élève complété, corrigé — oral - 126 lignes)

\*

Algorithmique, Ocaml,  
sources : ccinpalgo2.tex

On considère un ensemble  $P$  de valeurs de pièces (appelé *système de pièces*). On suppose disposer d'une infinité de chaque type de pièce.

Un tel ensemble sera représenté par un tableau d'entiers.

On souhaite implémenter un algorithme permettant de déterminer le nombre minimal de pièces permettant de rendre la monnaie sur une certaine valeur  $V$ .

1. Spécifier précisément et implémenter une fonction `verification`, qui prend en entrée un système de pièces  $p$ , une somme  $v$  à atteindre et un tableau indiquant en la case d'indice  $i$  le nombre de pièce d'indice  $i$  prises.
2. On suppose  $P$  donnée par une liste de pièces triées par ordre décroissant de valeur (dont une pièce de valeur 1). Donner un algorithme glouton qui «semble fonctionner».
3. On prend  $P = \{200, 100, 50, 20, 10, 5, 2, 1\}$  et  $v = 874$ , puis  $P = \{8, 7, 1\}$  et  $v = 14$ . L'algorithme fonctionne-t-il dans chacun de ces cas ?
4. L'algorithme glouton est-il un algorithme d' $\alpha$ -approximation pour un certain  $\alpha > 1$  ?
5. On suppose  $P$  représenté par un tableau trié par ordre croissant de valeur. Compléter la fonction `rendu_memo` résolvant ce problème en utilisant une stratégie par programmation dynamique descendante avec mémoïsation.
6. Quelle est la complexité de cette fonction ? Quelle stratégie algorithmique permettrait de réduire le temps d'exécution ?
7. Implémenter une fonction `rendu_asc` : `int array -> int -> int` résolvant le problème par une approche par programmation dynamique ascendante.



## Chapitre 44

# (CCINP) Dénombrement du nombre de façons d'obtenir un score avec des dés. \*

## (CCINP 25, ex B, retour élève, corrigé — oral - 150 lignes)

\*

Algorithmique, Ocaml, Programmation dynamique,  
sources : ccinpalgo4.tex

Dans tout cet exercice, il est interdit d'utiliser les fonctions `List.iter`, `List.map` et `List.filter`. L'opérateur `@` est autorisé.

On cherche dans cet exercice à déterminer de combien de façon on peut obtenir un score  $s \geq 0$  en lançant un nombre  $d \geq 0$  de dés distinguables à  $f \geq 1$  faces numérotées de 1 à  $f$ .

1. De combien de façons peut-on faire 3 avec deux dés à 2 faces ?

On définit le type suivant pour représenter le résultat d'un lancer :

```
type lancer = int list
```

2. Écrire une fonction `somme` : `lancer`  $\rightarrow$  `int` calculant le résultat d'un lancer.
3. Écrire une fonction `ajouter` : `lancer`  $\rightarrow$  `int`  $\rightarrow$  `lancer` qui ajoute l'entier passé en argument au lancer passé en argument.

On considère la fonction suivante :

```
let myst l i =  
  if i = 0 then []  
  else (ajouter l i) :: (myst l (i-1))
```

4. Que renvoie cette fonction ?
5. Écrire une fonction `toutes_possibilites` : `int`  $\rightarrow$  `int`  $\rightarrow$  `lancer list` qui prend en argument deux entiers  $d$  et  $f$ , et renvoie la liste des lancers possibles pour ces paramètres.
6. Combien y a-t-il de possibilités pour  $d = 6$  et  $f = 6$  ?
7. Écrire une fonction `toutes_possibilites_somme` : `int`  $\rightarrow$  `int`  $\rightarrow$  `int`  $\rightarrow$  `lancer list` qui prend en argument trois entiers  $d$ ,  $f$  et  $s$ , et renvoie la liste des lancers pour  $d$  et  $f$  dont la somme vaut  $s$ .
8. Tester pour les paramètres  $(d, f, s) = (6, 6, 24)$ , puis pour  $(12, 8, 30)$ , et commenter.

On note  $t_{i,j}$  le nombre de manières dont on peut obtenir le score  $j$  en lançant  $i$  dés à  $f$  faces. Par convention,  $t_{0,0} = 1$ .

On remarque que  $t_{0,j} = 0$  lorsque  $j > 0$ , et que pour  $i \geq 1$ ,  $t_{i,j} = \sum_{k=1}^{\min(j,f)} t_{i-1,j-k}$ .

On donne une fonction `nb_possibilites_somme` implémentant ce calcul dans le fichier `.ml` fourni.

9. Tester cet algorithme avec les deux triplets précédents, et commenter.
10. Modifier la fonction en utilisant une stratégie de mémoïsation à l'aide de dictionnaires. On rappelle les signature des fonctions du module `Hashtbl` ci-dessous.

### Tables de hachage en OCaml (Hashtbl)

```
Hashtbl.create : int -> ('a, 'b) Hashtbl.t (*l'entier passé en argument est une←  
  taille de départ indicative*)
```

```
Hashtbl.add : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit
```

```
Hashtbl.find_opt : ('a, 'b) Hashtbl.t -> 'a -> 'b option (*None si la clé est ↵  
absente*)
```

## Chapitre 45

# (CCINP) Expressions régulières en OCaml \* (CCINP 25, ex B, retour élève, corrigé — oral - 157 lignes)

\*

Langages, Ocaml,  
sources : ccinpregexp.tex

On représente les *expressions régulières* avec le type suivant en *OCaml* :

```
type expr =
  | Vide
  | Lettre of char
  | Union of expr * expr
  | Concat of expr * expr
  | Etoile of expr
```

1. Écrire une fonction `est_vide : expr -> bool` qui teste si une expression régulière dénote le langage vide.
2. Justifier que votre fonction termine.
3. Écrire une fonction `est_fini : expr -> bool` qui teste si une expression régulière dénote un langage fini.
4. Écrire une fonction `fusion l1 l2` où `l1` et `l2` sont des listes de chaînes de caractères triées dans l'ordre lexicographique, et renvoie une liste triée par ordre lexicographique contenant les mots de `l1` et `l2` – sans doublons. On indique que l'opérateur `<` permet de comparer des chaînes de caractères selon l'ordre lexicographique. Une complexité linéaire est attendue.
5. Donner la spécification de la fonction suivante.

```
let rec myst l1 l2 =
  let rec aux e l = match l with
    [] -> []
  | t :: q -> (e ^ t) :: (aux e q)
  in
  match l1 with
  [] -> []
  | t :: q -> aux t l2 @ myst q l2
```

6. Écrire une fonction `langage l : expr -> string list` qui prend en argument une expression régulière dénotant un langage fini  $L$ , et renvoie la liste des mots  $L$  triée par ordre lexicographique.  
*Indication* : on pourra utiliser `String.make`, qui a un fonctionnement similaire à `Array.make`.
7. Écrire une fonction `denote l : string list -> expr` qui prend en entrée une liste de mots non vides, et renvoie une expression régulière qui dénote l'ensemble de ces mots.
8. Soit  $L$  un langage reconnu par un automate à  $n$  états. Donner un algorithme pour savoir si  $L$  est vide (et préciser sa complexité).
9. Même question si  $L$  est un langage fini.



# Chapitre 46

## (ENS) Jeu des jetons \*\*\* (ENS 23 MP, corrigé — oral - 153 lignes)

\*\*\*

Jeux, Graphes,

sources : ensjeujeton.tex

Jeu des jetons (info MP 2023) :

### Définition 1

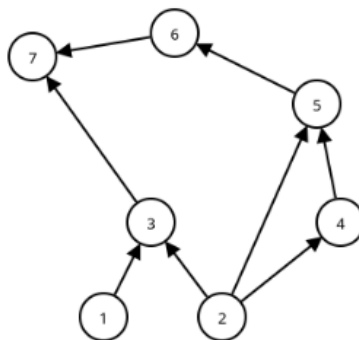
Un graphe orienté  $G$  est défini par un ensemble fini de sommets  $V$  et d'arêtes  $E \subseteq V \times V$ . Pour un sommet  $x \in V$ , ses *prédécesseurs* forment l'ensemble des sommets  $y \in V$  tels que  $(y, x) \in E$  et ses *successeurs* forment l'ensemble des sommets  $z \in V$  tels que  $(x, z) \in E$ . On suppose de plus que le graphe ne contient pas de cycle, y compris des arêtes d'un sommet vers lui-même.

On s'intéresse au problème suivant : on fixe un sommet  $v$  du graphe, appelé *sommet distingué*. On dispose d'un ensemble de *jetons* qui peuvent être placés sur les sommets du graphe, de sorte que chaque sommet ne comporte que zéro ou un jeton. S'il a un jeton, le sommet est dit *marqué*. On peut placer les jetons selon les règles suivantes :

1. Si  $x$  n'a aucun prédécesseur, on peut placer un jeton sur  $x$
2. Si tous les prédécesseurs de  $x$  sont marqués, on peut placer un jeton sur  $x$
3. On peut toujours retirer un jeton

Une *étape* du jeu consiste à placer ou retirer un unique jeton en suivant ces règles. À la fin du jeu, on veut qu'il ne reste qu'un seul jeton placé sur le sommet distingué.

1. Soit le graphe représenté ci-dessous, où "7" est le sommet distingué. Montrer qu'il existe une stratégie utilisant 4 jetons.



2. Démontrer qu'un graphe orienté acyclique comporte au moins un sommet sans prédécesseur.
3. En déduire que pour tout graphe orienté acyclique à  $n$  sommets, il existe toujours une stratégie pour résoudre le jeu avec  $n$  jetons.
4. Soit un arbre binaire (chaque sommet non-feuille a deux enfants) à  $\ell$  feuilles, dans lequel les prédécesseurs d'un sommet sont ses enfants, et le sommet distingué est la racine  $r$ . Montrer qu'il existe une stratégie utilisant  $\lceil \log_2 \ell \rceil + 2$  jetons, où  $\lceil x \rceil$  est la partie entière supérieure de  $x$ .

**Définition 2**

On remplace maintenant la règle 3. par les deux règles suivantes :

1. On peut toujours retirer un jeton d'un sommet sans prédécesseurs
2. On ne peut retirer un jeton d'un sommet que si ses prédécesseurs sont tous marqués

À partir de maintenant, on considère uniquement le graphe-ligne  $L_n := x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_n$ , où  $x_n$  sera toujours le sommet distingué.

5. Montrer qu'avec ces nouvelles règles :
  - (a) Il existe toujours une stratégie pour résoudre le jeu en utilisant  $n$  jetons et  $O(n)$  étapes.
  - (b) S'il existe une stratégie pour marquer  $x_n$  en  $k$  jetons et  $t$  étapes, alors il existe aussi une stratégie pour, en partant de la configuration finale ( $x_n$  est le seul sommet marqué), retrouver la configuration initiale (aucun sommet marqué) en  $k$  jetons et  $t$  étapes.
6. Montrer qu'on peut marquer  $x_n$  en  $O(n)$  étapes et  $O(\sqrt{n})$  jetons.
7. Montrer qu'on peut marquer  $x_n$  en utilisant au total  $\lceil \log_2 n \rceil + 1$  jetons. Combien d'étapes comporte cette stratégie ?
8. On cherche maintenant à obtenir un meilleur compromis entre le nombre de jetons utilisés et le nombre d'étapes. Montrer que pour tout  $\varepsilon$ , il existe une stratégie utilisant  $O(\log n)$  jetons et  $O(n^{1+\varepsilon})$  étapes.

# Chapitre 47

## (ENS) Monoïdes et Langages \*\*\* (ENS 23 MP, corrigé — oral - 238 lignes)

\*\*\*

Langages,

sources : `ensmonoïdeslang.tex`

Monoïdes et Langages :

### Définition 3

Un monoïde est un triplet  $(M, \cdot_M, e_M)$  où  $e_M \in M$  et  $\cdot_M : (M \times M) \rightarrow M$  vérifie  $\forall x \in M, x \cdot_M e_M = e_M \cdot_M x = x$  et  $\forall x, y, z \in M, (x \cdot_M y) \cdot_M z = x \cdot_M (y \cdot_M z)$ . (Informellement, un monoïde est un groupe sans garantie d'existence d'inverses.) On abrège souvent  $x \cdot_M y$  en  $xy$  et  $(xy)z$  ou  $x(yz)$  en  $xyz$ . Fixons un monoïde  $(M, \cdot_M, e_M)$  pour l'ensemble du sujet, avec  $M$  fini.

- Soit  $x \in M$ .
  - Montrer que  $j_x := \min E$ , où  $E := \{j \in \mathbb{N} \mid \exists i \in [0, j-1], x^i = x^j\}$ , est bien défini.
  - Montrer que les éléments  $1, x, \dots, x^{j_x-1}$  sont distincts deux à deux.
  - Montrer qu'il existe un unique  $i_x \in [0, j_x-1]$  tel que  $x^{i_x} = x^{j_x}$ .
  - Soit  $p_x := j_x - i_x$ . Montrer que pour tout  $n \in \mathbb{N}, x^{i_x+n} = x^{i_x+r}$ , où  $r := (n \bmod p_x)$  est le reste de la division euclidienne de  $n$  par  $p_x$ .
- On ne demande pas de justification pour cette question. Soit  $x \in M$ . Trouver  $e_x \in M$  tel que  $(G_x, \cdot_M, e_x)$  soit un groupe, où  $G_x := \{x^{i_x}, \dots, x^{i_x+p_x-1}\}$ . On ne demande pas de vérifier l'existence d'inverses.

### Définition 4

Un groupe  $(G, \cdot_G, e_G)$  est dit contenu dans  $(M, \cdot_M, e_M)$  si  $G \subseteq M$  et  $\forall x, y \in G, x \cdot_G y = x \cdot_M y$ . On n'exige pas  $e_G = e_M$ , donc pour tout  $x \in M, (\{x\}, \cdot_x, x)$  est un groupe, où  $\cdot_x$  est une restriction de  $\cdot_M$ . Un monoïde est dit apériodique s'il ne contient que des groupes réduits à un élément.

- On rappelle que  $M$  est fini. Montrer que les trois assertions suivantes sont équivalentes, par exemple en montrant  $1 \Rightarrow 2$  puis  $2 \Rightarrow 3$  puis  $3 \Rightarrow 1$ .
  - $M$  est un monoïde apériodique.
  - Pour tout  $x \in M$ , on a  $p_x = 1$  (et donc  $x^{i_x+1} = x^{i_x}$ )
  - $\exists k \in \mathbb{N}, \forall x \in M, x^{k+1} = x^k$

### Définition 5

On suppose dans la suite que  $(M, \cdot_M, e_M)$  est apériodique et fini.

- (Lemme de simplification).** Soient  $p, m, q \in M$ .
  - Montrer que si  $m = pmq$ , alors  $m = pm = mq$ .
  - En déduire que si  $pq = e_M$ , alors  $p = q = e_M$ .
- (Caractérisation de l'égalité).** Soit  $m, x \in M$ . Montrer que  $m = x$  ssi  $x \in (mM \cap Mm)$  et  $m \in MxM$ . (Où  $mM := \{my \mid y \in M\}$  et  $MxM := \{yxz \mid y, z \in M\}$ .)

**Définition 6**

Fixons un ensemble fini non vide  $\Sigma$ . Soit  $\Sigma^*$  l'ensemble des mots finis sur  $\Sigma$ , et  $\varepsilon$  le mot vide. Soit  $\mu : \Sigma^* \rightarrow M$  un morphisme, i.e.  $\mu(\varepsilon) = e_M$  et  $\mu(uv) = \mu(u) \cdot_M \mu(v)$  pour tout  $u, v \in \Sigma^*$ , où  $uv$  est la concaténation des mots  $u$  et  $v$ .

6. Soient  $m \in M$  et  $w \in \Sigma^*$ . Montrer que les deux assertions suivantes sont équivalentes :
- (a)  $m \notin M\mu(w)M$
  - (b)  $w$  se décompose soit en  $w = uav$  où  $m \notin M\mu(a)M$ , soit en  $w = uavbt$  où  $m \in M\mu(av)M \cap M\mu(vb)M$  et  $m \notin M\mu(avb)M$ . (Où  $u, v, t \in \Sigma^*$  et  $a, b \in \Sigma$ .)
7. Soient  $m \in M \setminus \{e_M\}$  et  $w \in \Sigma^*$ . Montrer que les deux assertions suivantes sont équivalentes :
- (a)  $\mu(w) \in mM$
  - (b) Il existe  $u, v \in \Sigma^*$  et  $a \in \Sigma$  tels que  $w = uav$  et  $\mu(u) \notin mM$  et  $\mu(ua)M \subseteq mM$ .
8. Soit  $m \in M \setminus \{e_M\}$ . Montrer que  $\mu^{-1}(m) = (U\Sigma^* \cap \Sigma^*V) \setminus (\Sigma^*Y\Sigma^*)$ , où  $\mu^{-1}(m) := \{w \in \Sigma^* \mid \mu(w) = m\}$  et

$$U := \bigcup_{\substack{a \in \Sigma \\ x \in M \setminus mM \\ x\mu(a)M \subseteq mM}} \mu^{-1}(x)a \quad V := \bigcup_{\substack{a \in \Sigma \\ x \in M \setminus mM \\ M\mu(a)x}} a\mu^{-1}(x) \subseteq Mm$$

$$Y := \{a \in \Sigma \mid m \notin M\mu(a)M\} \cup Z \quad Z := \bigcup_{\substack{a, b \in \Sigma \\ m \in M\mu(a)xM \cap Mx\mu(b)M \\ m \notin M\mu(a)x\mu(b)M}} a\mu^{-1}(x)b$$

# Chapitre 48

## (ENS) Élimination des coupures dans MLL et MALL \*\*\*\* (ENS 23, corrigé partiellement — oral - 385 lignes)

\*\*\*\*

Logique propositionnelle,  
sources : `enseliminationcutmll.tex`

### Elimination des coupures dans MLL et MALL

La logique linéaire additive-multiplicative *MALL* a pour syntaxe :

$A, B ::= p \mid p^\perp \mid A \otimes B \mid A \wp B \mid A \oplus B \mid A \& B$

avec  $p$  appartenant à un ensemble infini de *formules atomiques*.

L'opération de *dualité*  $\cdot^\perp$  est définie inductivement sur les formules :

$(A \otimes B)^\perp = A^\perp \wp B^\perp \quad (A \oplus B)^\perp = A^\perp \& B^\perp$

$(A \wp B)^\perp = A^\perp \otimes B^\perp \quad (A \& B)^\perp = A^\perp \oplus B^\perp \quad (p^\perp)^\perp = p$

On note  $\Gamma, \Delta$  des multi-ensembles  $A_1, \dots, A_n$  de formules (c'est-à-dire que les  $A_i$  ne sont pas ordonnées, mais on tient compte de leur multiplicité). Un *séquent linéaire* (ou seulement *séquent*, dans ce sujet)  $\vdash \Gamma$  est prouvable quand on peut le déduire des règles suivantes (c'est-à-dire construire avec ces règles une *preuve* dont  $\vdash \Gamma$  est la conclusion) :

$$\frac{}{\vdash A, A^\perp} (\text{Ax}) \quad \frac{\vdash \Gamma, A^\perp \vdash A, \Delta}{\vdash \Gamma, \Delta} (\text{Cut}) \quad \frac{\vdash \Gamma, A \quad \vdash B, \Delta}{\vdash \Gamma, A \otimes B, \Delta} (\otimes) \quad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \wp B} (\wp)$$

$$\frac{\vdash \Gamma, A \quad \vdash \Gamma, B}{\vdash \Gamma, A \& B} (\&) \quad \frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B} (\oplus_1) \quad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B} (\oplus_2)$$

Une occurrence de (Cut) dans une preuve qui ajoute les formules  $A$  et  $A^\perp$  dans ses séquents prémisses est appelée une *coupure sur A*.

Usuellement on appelle "tenseur" le connecteur  $\otimes$ , "par" le  $\wp$ , "plus" le  $\oplus$  et "avec" le  $\&$ .

On définit la formule  $A \multimap B$  comme  $A^\perp \wp B$ .

**Question 1.** Montrer que  $\vdash A \otimes (B \oplus C) \multimap (A \otimes B) \oplus (A \otimes C)$  est prouvable.

**Question 2.** Montrer qu'il n'existe pas de preuve de  $\vdash A \multimap (A \otimes A)$  sans coupure.

Informellement, on interprète la formule  $A$  comme "un exemplaire de la ressource  $A$ ".

**Question 3.** Question supprimée.

On dit qu'une règle *élimine la formule A*, si  $A$  est une formule qui apparaît dans le séquent conclusion de la règle mais dans aucun des séquents prémisses. Par exemple, la règle  $(\otimes)$  élimine la formule  $A \otimes B$ . Dans une preuve, on dit qu'une coupure sur la formule  $C$  est *principale* si  $C^\perp$  et  $C$  sont toutes les deux éliminées par (respectivement) la première règle appliqué à la prémisse gauche et la première règle appliquée à la prémisse droite de la coupure.

**Question 4.** Soit  $\Pi$  une preuve de  $\vdash \Gamma_0$  contenant une coupure principale sur la formule  $C$ .

Montrer qu'on peut supprimer la coupure sur  $C$  de  $\Pi$ , c'est-à-dire obtenir une preuve de  $\vdash \Gamma_0$  sans cette coupure sur  $C$ .

**Question 5.** Soit  $\Pi$  une preuve de  $\vdash \Gamma_0$  contenant une coupure non-principale sur la formule  $C$ . Montrer qu'on peut supprimer cette coupure sur  $C$  de  $\Pi$ .

Le théorème d'élimination des coupures dans un système de preuve (un ensemble de règles)  $\mathcal{L}$  contenant (Cut) est donné par :

*Tout séquent  $\vdash \Gamma$  prouvable dans  $\mathcal{L}$  est prouvable dans  $\mathcal{L}$  privé de la règle (Cut)*

**Question 6.** Les transformations induites par les questions 4. et 5. forment-elle un bon algorithme pour prouver l'élimination des coupures dans MALL ?

Dans la suite, nous allons donner une preuve plus graphique, permettant de s'abstraire du mécanisme de la question 5.

MLL est la logique obtenue à partir de MALL en n'utilisant que les formules atomiques, la dualité  $\perp$  et les connecteurs  $\otimes$  et  $\wp$ , et seulement les règles (Ax), (Cut), ( $\otimes$ ) et ( $\wp$ ).

Un *réseau* de MLL (ou simplement *réseau*) est un graphe orienté avec arêtes pendantes (des arêtes sans destination dont la source est un sommet du graphe), dont les sommets (appelés *nœuds*) sont étiquetés par (Ax), (Cut), ( $\wp$ ), ( $\otimes$ ), et les arêtes par des formules. Chaque nœud étiqueté par ( $\otimes$ ) ou ( $\wp$ ) a deux arêtes entrantes (étiquetées par des formules  $A$  et  $B$ ) et une arête sortante (étiquetée par la respectivement  $A \otimes B$  et  $A \wp B$ ). Les nœuds étiquetés par (Ax) n'ont pas d'arêtes entrante et deux arêtes sortantes, étiquetées par une variable et sa négation. Les nœuds étiquetés par (Cut) n'ont pas d'arête sortante et deux arêtes entrantes (étiquetées par  $C$  et  $C^\perp$ , où  $C$  est une variable).

**Question 7.** Donner un algorithme qui prend une preuve  $\Pi$  de MLL et la transforme en un réseau correspondant, de sorte que les arêtes pendantes correspondent aux conclusions de la preuve.

Un *réseau de preuve* est un réseau dans l'image de l'algorithme précédent (obtenu à partir d'une preuve d'un séquent de MLL).

Un *interrupteur* d'un réseau  $\mathcal{R}$  est un graphe non-orienté obtenu en supprimant les arêtes pendantes de  $\mathcal{R}$ , en supprimant, pour chaque nœud étiqueté par  $\wp$  dans  $\mathcal{R}$ , une des deux arêtes entrantes dans ce nœud, et en désorientant les arêtes restantes.

On admet le *théorème de Danos-Régnier* :

*Un réseau  $\mathcal{R}$  est un réseau de preuve si et seulement si tout interrupteur de  $\mathcal{R}$  est connexe et acyclique.*

L'idée sous-jacente aux réseaux est que cette représentation permet de s'abstraire de l'endroit où les coupures s'effectuent dans une preuve.

**Question 8.**

1. Illustrer la remarque précédente en donnant deux preuves différentes d'un même séquent linéaire envoyées par l'algorithme de la question 7. sur le même réseau.
2. Donner un réseau qui n'est pas un réseau de preuve.
3. Donner un algorithme prend un réseau de preuve obtenu à partir d'une preuve prouvant le séquent  $\vdash \Gamma$  et renvoie une preuve de  $\vdash \Gamma$ .

**Question 9.** Montrer le théorème d'élimination des coupures dans MLL.

**Question 10.** Peut-on appliquer cette technique à MALL ?

# Chapitre 49

## (ENS) Graphes parfaits \*\*\* (ENS 23, corrigé — oral - 251 lignes)

\*\*\*

Graphes,

sources : `ensgraphesparfaits.tex`

Pour  $S$  un ensemble, on note  $\mathcal{P}_2(S) = \{\{x, y\} : x, y \in S, x \neq y\}$  les sous-ensembles de  $S$  de cardinalité 2. Dans tout le sujet, on considère des graphes non-orientés  $G = (V, E)$ , avec un ensemble fini de sommets  $V$ , et un ensemble d'arêtes  $E \subseteq \mathcal{P}_2(V)$ .

**Coloriage.** Un coloriage d'un graphe  $G = (V, E)$  est une application  $V \rightarrow \mathbb{N}$ . Dans ce contexte, on appelle les entiers des *couleurs*. Un coloriage est *valide* si toute paire de sommets reliés par une même arête a des couleurs différentes. Le *nombre chromatique* de  $G$ , noté  $\chi(G)$ , est le nombre minimal de couleurs nécessaires pour créer un coloriage valide de  $G$ .

**Sous-graphe induit.** Étant donné un graphe  $G = (V, E)$  et un sous-ensemble de sommets  $W \subseteq V$ , le *sous-graphe induit* par  $W$  est le graphe  $G[W] = (W, E \cap \mathcal{P}_2(W))$ . On dit que c'est un sous-graphe induit *propre* si  $W$  est inclus strictement dans  $V$ .

**Cliques.** Une *clique* d'un graphe  $G = (V, E)$  est un sous-ensemble de sommets  $W \subseteq V$  tel que le sous-graphe  $G[W]$  induit par  $W$  est un graphe complet, c'est-à-dire :  $G[W] = (W, \mathcal{P}_2(W))$ . On note  $\omega(G)$  la cardinalité de la plus grande clique de  $G$ .

**Anticliques.** Une *anticlique* d'un graphe  $G = (V, E)$  est un sous-ensemble de sommets  $W \subseteq V$  tel que le sous-graphe  $G[W]$  induit par  $W$  ne contient pas d'arête, c'est-à-dire :  $G[W] = (W, \emptyset)$ . On note  $\alpha(G)$  la cardinalité de la plus grande anticlique de  $G$ .

**Question 1.** Soit  $G$  un graphe quelconque. Montrer  $\chi(G) \geq \omega(G)$ .

**Question 2.** Soit  $G = (V, E)$  un graphe quelconque.

- Montrer qu'un coloriage valide de  $G$  avec  $c$  couleurs existe si et seulement si il existe une partition  $\{A_1, \dots, A_c\}$  de  $V$  en  $c$  anticliques.
- Montrer  $\chi(G)\alpha(G) \geq |V|$ .

**Graphe parfait.** Un graphe  $G$  est dit *parfait* si tous ses sous-graphes induits  $G[W]$  satisfont :  $\chi(G[W]) = \omega(G[W])$ .

**Graphe imparfait minimal.** Un graphe  $G$  est dit *imparfait minimal* s'il n'est pas parfait, et que tous ses sous-graphes induits propres sont parfaits.

**Question 3.** Donner un exemple de graphe parfait, et de graphe imparfait minimal.

Pour simplifier les notations, dans les questions 4 à 8, on fixe  $G$  un graphe imparfait minimal quelconque. Sans perte de généralité, on pose  $V = \{1, \dots, n\}$ . On note  $\alpha = \alpha(G), \omega = \omega(G), \chi = \chi(G)$ .

**Question 4.** Soit  $A$  une anticlique de  $G$ . Montrer  $\omega(G[V \setminus A]) = \omega$ .

**Question 5.** Soit  $A_0$  une anticlique de  $G$  de cardinalité  $\alpha$ . Montrer qu'il existe  $\alpha\omega$  anticliques  $A_1, \dots, A_{\alpha\omega}$ , telles que pour chaque sommet  $v \in V$ ,  $v$  fait partie d'exactly  $\alpha$  anticliques parmi  $A_0, \dots, A_{\alpha\omega}$ . (Formellement :  $\forall v \in V, |\{i \in \{0, \dots, \alpha\omega\} : v \in A_i\}| = \alpha$ .)

**Question 6.** On considère une suite d'anticliques  $A_0, \dots, A_{\alpha\omega}$  définie comme dans la question précédente. Montrer que pour tout  $i$  dans  $\{0, \dots, \alpha\omega\}$ , il existe une clique  $C_i$  telle que  $C_i \cap A_i = \emptyset$ , et  $\forall j \neq i, |C_i \cap A_j| = 1$ .

*Indication :* utiliser la question 4

**Matrice d'incidence.** Étant donné une suite  $V_0, \dots, V_{\alpha\omega}$  de sous-ensembles de  $V = \{1, \dots, n\}$ , on définit la *matrice d'incidence* de la suite  $(V_i)$  comme la matrice  $M = (M_{i,j})_{1 \leq i \leq n, 0 \leq j \leq \alpha\omega} \in \{0, 1\}^{n \times (\alpha\omega + 1)}$  définie par  $M_{i,j} = 1$  si  $i \in V_j$ , 0 sinon.

**Question 7.** Soit  $M_A$  la matrice d'incidence de la suite  $A_0, \dots, A_{\alpha\omega}$  de la question 5, et  $M_C$  la matrice d'incidence de la suite  $C_0, \dots, C_{\alpha\omega}$  de la question 6. On note  $M_A^\top$  la transposée de  $M_A$ . Montrer que  $M_A^\top M_C$  est de rang  $\alpha\omega + 1$ , où les matrices sont vues comme à coefficient dans  $\mathbb{Q}$ .

**Question 8.** Montrer  $n \geq \alpha\omega + 1$ .

Dans les questions suivantes,  $G = (V, E)$  est un graphe quelconque.

**Question 9.** Montrer que  $G$  est parfait si et seulement si  $\omega(G[W])\alpha(G[W]) \geq |W|$  pour tout sous-graphe induit  $G[W]$  de  $G$ .

**Question 10.** Soit  $\bar{G} = (V, \mathcal{P}_2(V) \setminus E)$  le graphe complémentaire de  $G$ . Montrer que  $G$  est parfait si et seulement si  $\bar{G}$  est parfait.

# Chapitre 50

## (ENS) Inférence de type \*\*\* (ENS 23, corrigé — oral - 279 lignes)

\*\*\*

Algorithmique, Langages fonctionnels, Pseudo-code,  
sources : `ensinferencetypes.tex`

Soit  $\mathcal{A} = \{A, B, C, \dots\}$  un ensemble infini de *variables de types*.  
On considère  $\mathcal{T}$  l'ensemble des *types* définis inductivement par :

- $\mathbb{N} \in \mathcal{T}$ . ( $\mathbb{N}$  est un type)
- $\mathcal{A} \subseteq \mathcal{T}$ . (une variable de types est un type)
- si  $T_1 \in \mathcal{T}$  et  $T_2 \in \mathcal{T}$  alors  $T_1 \rightarrow T_2 \in \mathcal{T}$ . (la "flèche" de deux types est un type)

Une *substitution de types*  $\sigma : \mathcal{A} \rightarrow \mathcal{T}$  est une fonction qui vaut l'identité presque partout, c'est-à-dire telle que  $\{A \in \mathcal{A} \mid \sigma(A) \neq A\}$  est fini.

Si  $\sigma$  est une substitution de types et  $T \in \mathcal{T}$  un type, on note  $\sigma_{\uparrow}(T)$  (par abus de notation, on s'autorisera à écrire  $\sigma(T)$ ) le type obtenu inductivement par :

- $\sigma_{\uparrow}(\mathbb{N}) = \mathbb{N}$ .
- pour  $\mathcal{A}$ ,  $\sigma_{\uparrow}(A) = \sigma(A)$ .
- pour tout  $T_1, T_2 \in \mathcal{T}^2$ ,  $\sigma_{\uparrow}(T_1 \rightarrow T_2) = \sigma_{\uparrow}(T_1) \rightarrow \sigma_{\uparrow}(T_2)$

Un *système d'équations de types* (ou juste *système*) est un ensemble fini  $\{(T_k, T'_k)\}_{1 \leq k \leq n}$  de couples de types.

Un *unificateur* d'un système  $\{(T_k, T'_k)\}_{1 \leq k \leq n}$  est une substitution de types  $\sigma$  telle que  $\forall k, \sigma_{\uparrow}(T_k) = \sigma_{\uparrow}(T'_k)$ .

Par exemple, on pose  $S_0 = \{(B \rightarrow A, C), (\mathbb{N}, B)\}$

et  $\sigma_0$  définie par 
$$\begin{cases} \sigma_0(B) = \mathbb{N} \\ \sigma_0(C) = \mathbb{N} \rightarrow A \\ \sigma_0(X) = X \text{ pour tout } X \notin \{B, C\} \end{cases}$$

Alors  $\sigma_0$  est un unificateur de  $S_0$  car :

1.  $\sigma_0(B \rightarrow A) = \mathbb{N} \rightarrow A$  et  $\sigma_0(C) = \mathbb{N} \rightarrow A$
2.  $\sigma_0(\mathbb{N}) = \mathbb{N}$  et  $\sigma_0(B) = \mathbb{N}$

**Question 1.** Trouver si possible un unificateur des systèmes suivants :

1.  $\{(\mathbb{N} \rightarrow A, B \rightarrow \mathbb{N})\}$
2.  $\{(\mathbb{N} \rightarrow A, B \rightarrow (C \rightarrow \mathbb{N})), (\mathbb{N} \rightarrow \mathbb{N}, C)\}$
3.  $\{(A \rightarrow B), (B \rightarrow A)\}$
4.  $\{(\mathbb{N} \rightarrow (A \rightarrow \mathbb{N}), \mathbb{N} \rightarrow B), (B, A)\}$

**Question 2.** Un système  $S$  admet-il toujours un unificateur ?

Quand un unificateur pour  $S$  est-il unique ?

**Question 3.** En considérant les différentes possibilités pour les formes des types  $T_1$  et  $T'_1$ , exprimer l'existence d'un unificateur pour  $\{(T_k, T'_k)\}_{1 \leq k \leq n}$  en fonction de l'existence d'un unificateur pour un autre système, bien choisi.

**Question 4.** Rédiger l'algorithme induit par la question précédente, étudier sa terminaison.

Soit  $\mathcal{X} = \{x, y, z, \dots\}$  un ensemble de variables de termes.

On considère un langage d'expressions fonctionnelles  $\mathcal{E}$  défini inductivement par :

- $\mathbb{N} \subseteq \mathcal{E}$ , (les entiers sont des expressions)
- $\mathcal{X} \subseteq \mathcal{E}$  (les variables de termes sont des expressions)
- Si  $(E_1, E_2) \in \mathcal{E}^2$ ,  $E_1 + E_2 \in \mathcal{E}$  (la somme de deux expressions est une expression)
- Si  $(E_1, E_2) \in \mathcal{E}^2$ ,  $E_1(E_2) \in \mathcal{E}$  (l'application d'une expression à une expression est une expression)
- Si  $x \in \mathcal{X}$  et  $E \in \mathcal{E}$  alors  $(x \mapsto E) \in \mathcal{E}$  (si  $E$  est une expression et  $x$  une variable, l'expression fonctionnelle  $x \mapsto E$  est une expression)

On supposera que les variables  $x$  apparaissant dans des sous-expressions  $x \mapsto E'$  d'une expression  $E$  sont toutes distinctes et distinctes deux à deux des variables de  $E$  qui n'apparaissent jamais directement à gauche d'un  $\mapsto$ . On note  $FV(E)$  ce dernier ensemble de variables.

Par exemple,  $f_0 = (x \mapsto (y \mapsto x(y) + 1))$  est un élément de  $\mathcal{E}$ .

Les contextes de types sont des fonctions de  $D \subseteq \mathcal{X}$  dans  $T$ .

On note  $\emptyset$  le contexte de domaine vide et  $(\Gamma, x : T)$ , le contexte  $\Gamma'$  défini par  $\Gamma'(x) = T$  et pour tout  $y$  dans le domaine de  $\Gamma$ ,  $y \neq x \Rightarrow \Gamma'(y) = \Gamma(y)$ .

Dans la suite on s'intéresse au problème de donner un type de  $\mathcal{T}$  à une expression de  $E \in \mathcal{E}$  avec un contexte  $\Gamma$  qui sert d'oracle donnant le type des variables de  $FV(E)$ , quand c'est possible. On dit qu'on infère un type de  $E$  dans le contexte  $\Gamma$ .

**Question 5.** Donner deux types différents qu'il semble légitime de donner à  $f_0$  dans le contexte  $\emptyset$ .

**Question 6.** Donner des règles qui formalisent quand il est légitime de donner un type  $T$  à  $E$  dans le contexte  $\Gamma$ .

**Question 7.** En déduire un algorithme qui infère un type d'une expression  $E$ .

**Question 8.** Utiliser cet algorithme pour trouver un type de  $\mathbf{S} = x \mapsto (y \mapsto (z \mapsto x(z)(y(z))))$  dans le contexte  $\emptyset$ .

# Chapitre 51

## (ENS) Ordonnancement \*\*\* (ENS 23, corrigé — oral - 201 lignes)

\*\*\*

Algorithmique, Complexité,  
sources : ensordo.tex

Ordonnancement :

### Définition 7

On cherche à ordonnancer  $n$  tâches indépendantes  $T_1, \dots, T_n \in \mathcal{T}$  sur un seul processeur, qui ne peut exécuter qu'une seule tâche à la fois. Chaque tâche  $T_i$  est décrite par deux entiers  $d_i$  et  $w_i$  représentant respectivement la date limite avant laquelle une tâche doit s'exécuter, et la pénalité à payer si la tâche est exécutée en retard. Le temps est représenté par des entiers naturels. Chaque tâche s'exécute en une unité de temps. Un ordonnancement est une fonction  $\sigma : \mathcal{T} \rightarrow \mathbb{N}$  associant à chaque tâche sa date d'activation. La première tâche peut être activée à la date 0. On dit qu'une tâche est ordonnancée à l'heure si elle finit son exécution avant ou à sa date limite, et qu'elle est en retard sinon.

Le but est de trouver un ordonnancement qui minimise la somme des pénalités de retard. Les ordonnancements qui minimisent la somme des pénalités de retard sont dits optimaux.

1. Quelle condition l'ordonnancement  $\sigma$  doit-il satisfaire pour être bien formé ?

### Définition 8

Un ordonnancement est dit canonique si :

- Les tâches à l'heure sont exécutées avant les tâches en retard.
- Les tâches à l'heure sont ordonnées par date limite croissantes.

2. Montrer qu'il existe un ordonnancement optimal qui est canonique.
3. On suppose que les tâches sont ordonnées par pénalité décroissante. Écrire un algorithme glouton qui résout le problème. Quelle est sa complexité ? On ne cherchera pas à prouver l'optimalité de cet algorithme dans cette question.
4. Illustrer l'algorithme sur l'exemple suivant :

$$w_1 = 7, w_2 = 6, w_3 = 5, w_4 = 4, w_5 = 3, w_6 = 2, w_7 = 1 \\ d_1 = 4, d_2 = 2, d_3 = 4, d_4 = 3, d_5 = 1, d_6 = 4, d_7 = 6$$

### Définition 9

Soit  $S$  un ensemble à  $n$  éléments,  $\mathcal{I} \subseteq \mathcal{P}(S)$ .  $(S, \mathcal{I})$  est un matroïde s'il satisfait les propriétés suivantes :

1. Hérité :  $X \in \mathcal{I} \implies (\forall Y \subset X, Y \in \mathcal{I})$
2. Échange :  $(A \in \mathcal{I}, B \in \mathcal{I}, |A| < |B|) \implies \exists x \in B \setminus A$  tel que  $A \cup \{x\} \in \mathcal{I}$ .

Les  $X \in \mathcal{I}$  sont appelés des indépendants.

On dit qu'un indépendant  $F \in \mathcal{I}$  est maximal s'il n'existe pas de  $x \in S \setminus F$  tel que  $F \cup \{x\} \in \mathcal{I}$ .

Un matroïde pondéré est un matroïde enrichi d'une fonction de poids  $w : S \rightarrow \mathbb{N}$ . Le poids d'un sous-ensemble  $X \subseteq S$  est la somme des poids des éléments :  $w(X) = \sum_{x \in X} w(x)$ .

On considère l'algorithme suivant, cherchant à trouver un indépendant de poids maximal :

```
fonction IndependantMax( $M = (S, \mathcal{I}), w : S \rightarrow \mathbb{N}$ ) ;  
   $S \leftarrow \text{tri\_décroissant}(S, w) \triangleright \text{Ainsi} : w(S[0]) \geq \dots \geq w(S[n-1])$  ;  
   $A \leftarrow \emptyset$  ;  
  Pour  $i = 0$  à  $n-1$  {  
    Si  $A \cup \{S[i]\} \in \mathcal{I}$  {  
       $A \leftarrow A \cup \{S[i]\}$  ;  
    }  
  }  
  Renvoyer  $A$ 
```

5. Soit  $x$  le premier élément de  $S$  (trié par ordre décroissant de poids) tel que  $\{x\}$  est indépendant. Montrer que si  $x$  existe, alors il existe une solution optimale  $A$  qui contient  $x$ .
6. Montrer que l'algorithme retourne une réponse optimale, puis donner la complexité de cet algorithme.
7. Prouver l'optimalité de l'algorithme d'ordonnement de la question 3.

# Chapitre 52

## (ENS) Permutations triables par pile \*\*\* (ENS 23, corrigé — oral - 184 lignes)

\*\*\*

Algorithmique, Graphes, Complexité,  
sources : `ensperm.tex`

Permutations triables par pile :

### Définition 10

On s'intéresse aux permutations  $\mathfrak{P}(n)$  de  $\{1, \dots, n\}$ . Une permutation  $\pi \in \mathfrak{P}(n)$  est assimilée à la suite  $(\pi(1), \pi(2), \dots, \pi(n))$ .

**Machine à pile.** Une *machine à pile* maintient en interne une structure de pile, initialement vide. Elle prend en entrée une permutation  $(\pi(1), \dots, \pi(n))$ , et effectue une suite fixée d'opérations **empile** et **dépile**.

- **empile** : lit le premier élément non encore lu de la permutation, et l'ajoute à la pile. La  $i$ -ième opération empile ajoute donc  $\pi(i)$  à la pile.
- **dépile** : enlève le dernier élément ajouté à la pile, et le renvoie en sortie.

La sortie de la machine est une suite d'éléments de  $\{1, \dots, n\}$  renvoyés par les opérations **dépile**, pris dans l'ordre où ils sont renvoyés. *Exemple* : la machine EEDEDD qui effectue (**empile**, **empile**, **dépile**, **empile**, **dépile**, **dépile**) avec en entrée la permutation  $(3, 1, 2)$  renvoie  $(1, 2, 3)$  :

$$\text{EEDEDD}(3, 1, 2) = (1, 2, 3)$$

On remarque que le comportement d'une machine à pile est entièrement défini par la suite (fixe) d'opérations **empile** et **dépile** qu'elle effectue.

**Suite valide.** Une telle suite d'opérations est dite *valide* si elle contient exactement  $n$  opérations **empile** et  $n$  opérations **dépile**, et si à tout instant, le nombre d'opérations **dépile** effectuées est inférieur ou égal au nombre d'opérations **empile** effectuées. Dans tout le sujet, on ne considère que des machines effectuant des suites valides d'opération.

**Permutation triable par pile.** Une permutation  $\pi \in \mathfrak{P}(n)$  est dite *triable par pile* s'il existe une machine à pile qui prend en entrée  $(\pi(1), \dots, \pi(n))$ , et qui renvoie  $(1, \dots, n)$ .

*Exemple* : l'égalité  $\text{EEDEDD}(3, 1, 2) = (1, 2, 3)$  plus haut montre que  $(3, 1, 2)$  est triable par pile.

1. Montrer que les permutations  $(1, 2, 3, 4)$ ,  $(4, 3, 2, 1)$ ,  $(1, 3, 2, 4)$  sont triables par pile.

### Définition 11

**Motif.** On dit que  $\pi \in \mathfrak{P}(n)$  contient le motif  $\rho \in \mathfrak{P}(k)$  pour  $k \leq n$  s'il existe  $x_1 < \dots < x_k$  dans  $\{1, \dots, n\}$  tel que  $(\pi(x_1), \dots, \pi(x_k)) = (x_{\rho(1)}, \dots, x_{\rho(k)})$  (ou plus informellement : tel que  $(\pi(x_1), \dots, \pi(x_k))$  et  $(\rho(1), \dots, \rho(k))$  sont «dans le même ordre»).

2. Montrer que si  $\pi$  est triable par pile, alors elle ne contient pas le motif  $(2, 3, 1)$ .
3. Montrer que si  $\pi$  ne contient pas le motif  $(2, 3, 1)$ , alors elle est triable par pile.
4. L'ensemble des permutations triables par pile est-il clos par composition ? Par inverse ?
5. Proposer un algorithme qui détermine en temps linéaire en  $n$  si une permutation  $\pi \in \mathfrak{P}(n)$  est triable par pile.  
*Indication* : on peut utiliser les questions 2 et 3, mais ce n'est pas obligatoire.

**Définition 12**

**Graphe associé à une permutation.** À une permutation  $\pi \in \mathfrak{P}(n)$ , on associe le graphe non-orienté  $G(\pi) = (V, E)$  de sommets  $V = \{1, \dots, n\}$ , et d'arêtes  $E = \{\{a, b\} \in V : a < b \text{ et } \pi(a) > \pi(b)\}$ .

**Graphe triable par pile.** On dit qu'un graphe  $G = (V, E)$  avec  $V = \{1, \dots, n\}$  est *triable par pile* s'il existe  $\pi \in \mathfrak{P}(n)$  tel que  $G = G(\pi)$  et  $\pi$  est triable par pile.

**Graphe réalisable.** On dit qu'un graphe  $G = (V, E)$  est réalisable s'il est possible de renommer ses sommets  $V$  avec les entiers  $\{1, \dots, n\}$ , de telle sorte que le graphe obtenu est triable par pile.

6. Montrer que  $\pi \mapsto G(\pi)$  est une injection de  $\mathfrak{P}(n)$  vers l'ensemble des graphes de sommets  $\{1, \dots, n\}$ . Est-ce une bijection ?
7. Donner un exemple de graphe qui n'est pas réalisable.
8. On considère l'ensemble des graphes formés en partant du graphe vide  $(\emptyset, \emptyset)$ , et en utilisant uniquement les deux opérations suivantes : ajout d'un sommet universel à un graphe de l'ensemble (un sommet est dit universel s'il est relié à tous les autres sommets), union disjointe de deux graphes de l'ensemble. Montrer que l'ensemble obtenu est exactement l'ensemble des graphes réalisables.
9. En s'inspirant de la question précédente, proposer un algorithme qui détermine si un graphe  $G = (V, E)$  est réalisable, en temps  $O(|V|^3)$ .

# Chapitre 53

## (ENS) Raisonnements ensemblistes \*\*\* (ENS 23, corrigé — oral - 234 lignes)

\*\*\*

Graphes, Logique propositionnelle,  
sources : `ensembles.tex`

L'ensemble des entiers naturels est noté  $\mathbb{N}$ , et l'ensemble des parties de  $\mathbb{N}$  est noté  $\mathcal{P}(\mathbb{N})$ .  
Soit  $\mathcal{T}$  un ensemble fini dont les éléments sont appelés des *variables ensemblistes*. Une *inclusion* est une formule s'écrivant  $(X \subseteq Y)$  où  $X$  (resp.  $Y$ ) est une variable ensembliste, ou  $\emptyset$ , ou  $\mathbb{N}$ . Par abus de notation,  $\emptyset$  et  $\mathbb{N}$  représentent ici respectivement les ensembles vide et plein.  
Attention : dans la suite, par convention, les lettres  $A, B, C, D \dots$  désigneront des éléments de  $\mathcal{T}$ , tandis que les lettres  $X, Y, Z$  désigneront des éléments de  $\mathcal{T} \cup \{\emptyset, \mathbb{N}\}$ .  
Une conjonction d'inclusions sur les variables de  $\mathcal{T}$  est dite *satisfiable* s'il existe une *valuation*  $f : \mathcal{T} \rightarrow \mathcal{P}(\mathbb{N})$  des variables qui vérifie toutes les inclusions.  
Pour deux formules  $\Phi$  et  $\Psi$ ,  $\Phi \models \Psi$  signifie que toute valuation satisfaisant  $\Phi$  satisfait également  $\Psi$ .

1. (a) Montrer que la conjonction d'inclusions  $(\mathbb{N} \subseteq A_1) \wedge (A_2 \subseteq A_3) \wedge (A_3 \subseteq \emptyset)$  est satisfiable.  
(b) Montrer qu'en ajoutant l'inclusion  $\mathbb{N} \subseteq A_2$ , ce n'est plus satisfiable.
2. Soit une conjonction d'inclusions ne contenant aucune inclusion de la forme  $\mathbb{N} \subseteq X$ . Montrer que cette conjonction est satisfiable.

Soit une conjonction d'inclusions  $\Phi = (X_1 \subseteq Y_1) \wedge \dots \wedge (X_k \subseteq Y_k)$ . On définit un graphe orienté  $G_\Phi$  tel que :

- Les sommets de  $G_\Phi$  sont les éléments de  $\mathcal{T} \cup \{\emptyset, \mathbb{N}\}$
- Pour toute inclusion  $X_i \subseteq Y_i$  de  $\Phi$ , on crée une arête  $X_i \rightarrow Y_i$  dans  $G_\Phi$
- Pour tout sommet  $A \neq \mathbb{N}$ , on crée une arête  $A \rightarrow \mathbb{N}$ , et pour tout sommet  $A \neq \emptyset$ , une arête  $\emptyset \rightarrow A$

Un *chemin* dans  $G_\Phi$  est une séquence de sommets  $Z_1, \dots, Z_\ell$  reliés par des arêtes  $Z_1 \rightarrow Z_2, Z_2 \rightarrow Z_3, \dots, Z_{\ell-1} \rightarrow Z_\ell$ .  
Par convention, on autorise des chemins ne comportant qu'un seul sommet (et aucune arête).

3. Montrer l'équivalence entre les propriétés suivantes :
  - (a)  $\Phi$  est satisfiable
  - (b) Il n'existe pas de chemin dans  $G_\Phi$  menant de  $\mathbb{N}$  à  $\emptyset$
  - (c)  $\Phi$  est satisfiable à valeurs dans  $\{\emptyset, \mathbb{N}\}$
4. Montrer que si  $\Phi$  est satisfiable et  $X$  et  $Y$  sont des sommets de  $G_\Phi$ , alors  $\Phi \models (X \subseteq Y)$  si et seulement s'il existe un chemin dans  $G_\Phi$  menant de  $X$  à  $Y$ .
5. En déduire un algorithme **Résolution** qui prend en entrée une conjonction d'inclusions  $\Phi$  et une inclusion  $(X \subseteq Y)$ , et détermine si  $\Phi \models (X \subseteq Y)$ . Montrer qu'avec la bonne structure de données, cet algorithme est de complexité linéaire en le nombre d'inclusions de  $\Phi$  et de variables de  $\mathcal{T}$ .

On considère maintenant des formules logiques dont les littéraux sont des inclusions, par exemple :

$$(X_1 \subseteq X_2) \vee \neg(X_2 \subseteq X_3) \vee \neg(X_4 \subseteq X_3)$$

Ces formules sont écrites en forme normale conjonctive. On suppose que toutes les clauses sont *Horn*, c'est-à-dire qu'elles contiennent au plus un seul littéral positif (sans négation). On les interprète alors comme des implications. La formule ci-dessus devient ainsi :

$$((X_2 \subseteq X_3) \wedge (X_4 \subseteq X_3)) \implies (X_1 \subseteq X_2)$$

On appelle une telle formule *inclusion-Horn* et on lui étend la notion de satisfiabilité ci-dessus.

6. Montrer que la formule inclusion-Horn avec les clauses suivantes :

$$\begin{aligned} &\neg(A_3 \subseteq \emptyset) \\ &\neg(A_1 \subseteq A_3) \\ &\neg(A_2 \subseteq A_3) \vee \neg(A_1 \subseteq A_2) \\ &(\mathbb{N} \subseteq A_1) \\ &(A_2 \subseteq \emptyset) \end{aligned}$$

est satisfiable, mais qu'aucune de ses valuations satisfaisantes n'est à valeurs dans  $\{\emptyset, \mathbb{N}\}$ .

On admet la propriété (P) :

« Soit  $\Psi$  une conjonction d'inclusions. Si  $\Psi$  est satisfiable, il existe une valuation  $f$  telle que pour tous sommets  $X, Y$  dans le graphe  $G_\Psi$ , s'il n'existe pas de chemin entre  $X$  et  $Y$ , alors  $f(X)$  et  $f(Y)$  sont incomparables (i.e.,  $f(X) \not\subseteq f(Y)$  et  $f(Y) \not\subseteq f(X)$ ). »

7. Soit  $\Phi$  une formule inclusion-Horn, et soit  $\Psi$  la conjonction de toutes les clauses de  $\Phi$  ne contenant aucun littéral négatif. C'est donc une conjonction d'inclusions de la forme :  $\Psi = (X_1 \subseteq Y_1) \wedge \dots \wedge (X_k \subseteq Y_k)$ .

Montrer que si  $\Psi$  est satisfiable, et si pour tout littéral négatif  $\neg(X \subseteq Y)$  dans les clauses de  $\Phi$ ,  $\Psi \not\models (X \subseteq Y)$ , alors  $\Phi$  est satisfiable.

8. En déduire un algorithme déterminant, en temps polynomial, si une formule inclusion-Horn est satisfiable.

9. Prouver la propriété (P).

# Chapitre 54

## (ENS) Théorème des amis \*\*\* (ENS 23, corrigé — oral - 202 lignes)

\*\*\*

Graphes,

sources : `enstheoremedesamis.tex`

### Théorème des amis

L'objectif du sujet est de montrer que dans tout groupe de personnes, si chaque paire de personnes a exactement un ami en commun, alors il existe quelqu'un (le diplomate) qui est ami de tout le monde. Le problème est modélisé par des graphes.

**Grphe.** Pour  $S$  un ensemble, on note  $\mathcal{P}_2(S) = \{\{x, y\} \subseteq S : x \neq y\}$  l'ensemble des paires d'éléments distincts de  $S$ . Dans tout le sujet, on considère des graphes non-orientés  $G = (V, E)$ , composés d'un ensemble fini de sommets  $V$ , et d'un ensemble d'arêtes  $E \subseteq \mathcal{P}_2(S)$ .

**Voisins.** Étant donné un graphe  $G = (V, E)$  et un sommet  $x \in V$ , l'ensemble des voisins de  $x$  est  $N(x) = \{y \in V : \{x, y\} \in E\}$ . Le *degré* de  $x$  est le nombre de voisins de  $x$ .

**Grphe d'amis.** Un **graphe d'amis** est un graphe  $G = (V, E)$  tel que  $|V| \geq 2$  et pour tout  $x, y \in V$  avec  $x \neq y$ , il existe un unique sommet, noté  $x \star y$ , tel que :  $\{x, x \star y\} \in E$  et  $\{y, x \star y\} \in E$ .

**Diplomate.** Étant donné un graphe  $G = (V, E)$ , un diplomate est un sommet de degré  $|V| - 1$ .

*Exemple.* Le graphe complet à trois sommets  $G = (\{1, 2, 3\}, \{\{1, 2\}, \{2, 3\}, \{1, 3\}\})$  est un graphe d'amis contenant 3 diplomates.

#### Question 1.

a. Montrer qu'un graphe d'amis  $G = (V, E)$  ne contient pas de 4-cycle, c'est-à-dire :

$$\neg \exists \{a, b, c, d\} \subseteq V, \{\{a, b\}, \{b, c\}, \{c, d\}, \{d, a\}\} \in E.$$

b. Donner un exemple de graphe d'amis à 5 sommets.

**Question 2.** Dans cette question, on suppose que  $G$  est un graphe d'amis contenant un sommet  $x$  de degré 2. On note  $y, z$  les deux voisins de  $x$ .

a. Montrer  $\{y, z\} \in E$ .

b. Montrer  $V = N(y) \cup N(z)$ .

c. Montrer que  $y$  ou  $z$  est un diplomate.

Dans les questions 3 à 5, on fixe  $G = (V, E)$  un graphe d'amis ne contenant pas de sommet de degré 2. (On suppose donc pour l'instant qu'un tel graphe existe; on verra plus tard si cela amène une contradiction.) Soit  $n = |V|$ . Sans perte de généralité, on pose  $V = \{1, \dots, n\}$ .

**Question 3.** Soit  $\{x, y\}$  une arête de  $G$  et  $z = x \star y$ . Soit  $X = N(x) \setminus \{y, z\}$ ,  $Y = N(y) \setminus \{x, z\}$ ,  $Z = N(z) \setminus \{x, y\}$ , et  $W = V \setminus (X \cup Y \cup Z \cup \{x, y, z\})$ .

a. Montrer que l'application suivante est bien définie et bijective :

$$f_\star : X \times Y \rightarrow W$$

$$(a, b) \mapsto a \star b.$$

b. Montrer que tous les sommets de  $G$  ont le même degré (on dit que  $G$  est régulier).

c. On note  $\delta$  le degré d'un sommet de  $G$ . Montrer  $n = \delta^2 - \delta + 1$ .

**Matrice d'adjacence.** La matrice d'adjacence de  $G$  est la matrice  $M_{i,j \in \{1, \dots, n\}} \in \mathbb{R}^{n \times n}$  définie par  $M_{i,j} = 1$  si  $\{i, j\} \in E$ , 0 sinon.

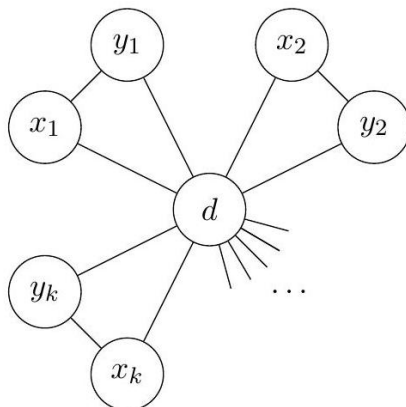
**Question 4.** Soit  $M$  la matrice d'adjacence du graphe  $G$ . On a vu dans la question 3 que  $G$  est régulier ; et on note  $\delta$  le degré d'un sommet de  $G$ . Montrer  $M^2 = \mathbf{1}_n + (\delta - 1) \text{ld}_n$ , où  $\mathbf{1}_n \in \mathbb{R}^{n \times n}$  est la matrice ne contenant que des 1, et  $\text{ld}_n \in \mathbb{R}^{n \times n}$  est la matrice identité.

On admet (ou rappelle) les faits suivants.

- Toute matrice symétrique réelle est diagonalisable.
- La trace d'une matrice est invariante par changement de base.
- Si  $k \in \mathbb{N}$  et  $\sqrt{k} \in \mathbb{Q}$ , alors  $\sqrt{k} \in \mathbb{N}$  (**lemme de Dirichlet**).
- Les valeurs propres de  $M^2 = \mathbf{1}_n + (\delta - 1) \text{ld}_n$  sont  $\delta^2$  (multiplicité 1) et  $\delta - 1$  (multiplicité  $n - 1$ ).

**Question 5.** Montrer que la trace de  $M$  doit être nulle, et aboutir à une contradiction.

**Question 6.** Montrer que pour tout graphe d'amis  $G = (V, E)$ , il existe un diplomate  $d \in V$ , et une partition de  $V \setminus \{d\}$  en paires  $\{\{x_1, y_1\}, \dots, \{x_k, y_k\}\}$ , telle que  $E = \bigcup_{i=1}^k \{\{x_i, y_i\}, \{d, x_i\}, \{d, y_i\}\}$ .



Graphe moulin

# Chapitre 55

## (ENS) Structures pliages et traversables \*\*\* (ENS 23, partiellement corrigé — oral - 314 lignes)

\*\*\*

Ocaml, Programmation fonctionnelle,  
sources : `ensstructurespliages.tex`

### Structures pliages et traversables

On se place dans un langage récursif, fonctionnel, avec des définitions de types inductifs et un système de types polymorphes similaire à OCaml. On pourra utiliser indifféremment du pseudocode fonctionnel ou la syntaxe OCaml. On suppose l'existence :

- d'une fonction identité `id` de type  $A \rightarrow A$ ;
- d'un opérateur  $(\circ) : (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C)$  de composition de fonctions, noté `o` dans sa version infixé; et
- de types de bases usuels, comme `int` le type des entiers et `unit` le type de l'unique élément  $()$ .

On donne la définition d'un type polymorphe `liste A` (où  $A$  est une variable de type) et, à titre d'exemple, de la fonction `longueur` de type  $A \rightarrow \text{int}$  en pseudocode (par exemple) :

```
liste A = Nil | Cons A (liste A)

longueur : liste A → int
  longueur Nil = 0
  longueur (Cons _ qu) = 1+ (longueur qu)
```

et en OCaml :

```
type 'a liste = Nil | Cons of ('a * 'a liste)

let rec longueur : 'a liste → int = function
  Nil → 0
  | Cons (_, qu) → 1+ (longueur qu)
```

Un type  $T$  est un *monoïde*, s'il existe (c'est-à-dire, si on peut définir)  $\varepsilon$  de type  $T$  et  $(\diamond)$  de type  $T \rightarrow T \rightarrow T$ , noté  $\diamond$  dans sa version infixé, tels que pour tous  $a, b, c$  de type  $T$ ,  $(a \diamond b) \diamond c = a \diamond (b \diamond c)$  et  $\varepsilon \diamond a = a \diamond \varepsilon = a$ .

**Question 1.** Montrer que `liste A` est un monoïde pour tout  $A$ .

Un *constructeur de type* est une fonction mathématique  $C$  qui associe à un type paramètre  $T$  un type résultat  $C T$ . Par exemple `liste` est un constructeur de type.

On dit qu'un constructeur de type  $C$  est *pliable* quand il existe un `fold` de type

$$(A \rightarrow B \rightarrow B) \rightarrow B \rightarrow CA \rightarrow B$$

pour tous types  $A$  et  $B$ , tel que `fold f e t` applique de manière répétée la fonction  $f$  à chacun des éléments de type  $A$  de la structure  $t$  (elle-même de type  $CA$ ) pour produire une valeur de type  $B$ , en partant initialement de l'élément  $e$

de type  $B$ .

**Question 2.** Montrer que `liste` est pliable.  
Un `fold` pour `liste` est-il unique ?

**Question 3.**

1. Donner la définition d'un constructeur de type `barbre` tel que les éléments de `barbre A` sont des arbres binaires dont les nœuds sont des éléments de  $A$ .
2. Montrer que `barbre` est pliable.
3. Expliquer comment différentes définitions de `fold` permettent de générer les parcours infixe et préfixe d'un arbre.

**Question 4.** Utiliser `fold` (sur les arbres) pour écrire une fonction `taille` : `barbre A`  $\rightarrow$  `int` qui renvoie la taille (i.e. le nombre de nœuds) d'un arbre.

Une définition alternative d'un constructeur de type pliable est l'existence d'un `foldmap` de type

$$(A \rightarrow M) \rightarrow CA \rightarrow M$$

pour tout type  $A$  et pour tout monoïde  $M$ , tel que `foldmap f t` parcourt la structure  $t$  en accumulant une valeur de type  $M$ .

**Question 5.** Montrer que `liste` et `barbre` sont pliables pour cette définition.  
Donner deux définitions de `foldmap` pour les `barbre` permettant de générer les parcours infixes et préfixes.

**Question 6.** Montrer que les deux définitions de pliabilité sont équivalentes.

Un foncteur applicatif est un constructeur de type  $F$  pour lequel il existe pour tous types  $A, B$  :

1. `fmap` de type  $(A \rightarrow B) \rightarrow FA \rightarrow FB$
2. `pure` de type  $A \rightarrow FA$
3.  $\odot$  de type  $F(A \rightarrow B) \rightarrow FA \rightarrow FB$ , noté  $\odot$  dans sa version infixe.

qui vérifie les lois suivantes :

$$\text{fmap id} = \text{id} \quad \text{fmap}(f \circ g) = (\text{fmap } f) \circ (\text{fmap } g)$$

$$(\text{pure id}) \odot v = v$$

**Question 7.** Sans vérifier formellement les lois, donner deux manières différentes de montrer que `liste` est un foncteur applicatif.

Un constructeur de types  $C$  est *traversable* s'il existe `traverse` de type

$$(A \rightarrow FB) \rightarrow CA \rightarrow F(CB)$$

pour tout foncteur applicatif  $F$  et types  $A$  et  $B$ .

**Question 8.** Montrer que `barbre` est traversable.

**Question 9.** On dispose d'un arbre  $t$  de type `barbre A` et une fonction `futurs` qui à un élément  $A$  associe un `liste` de type `liste A` qui correspond aux «futurs possibles» (dans un sens non-déterministe) d'un élément. À quoi correspond `traverse futurs t` ?

# Chapitre 56

## (ENS) Élimination des coupures dans MLL et MALL \*\*\* (ENS 23, réécriture plus raisonnable par N.Carré — oral - 377 lignes)

\*\*\*

Logique propositionnelle,

sources : mll.tex

### Définition

On considère un ensemble dénombrable  $\mathcal{V}$  dit de **variables**. L'ensemble  $\mathcal{F}$  des **formules de la logique linéaire multiplicative** est défini par induction par :

- si  $x \in \mathcal{V}$ , alors  $x \in \mathcal{F}$  et  $x^\perp \in \mathcal{F}$ . L'opérateur  $^\perp$  est appelé **négation linéaire** ;
- si  $A, B \in \mathcal{F}$ , alors  $A \otimes B \in \mathcal{F}$  et  $A \wp B \in \mathcal{F}$ . L'opérateur  $\otimes$  est appelé **tenseur** et l'opérateur  $\wp$  est appelé **par**.

Par sucre syntaxique, on étend l'opérateur  $^\perp$  aux formules par  $(x^\perp)^\perp = x$ ,  $(A \otimes B)^\perp = A^\perp \wp B^\perp$  et  $(A \wp B)^\perp = A^\perp \otimes B^\perp$ .

Un **séquent linéaire** (ou simplement séquent) est de la forme  $\vdash \Gamma$  où  $\Gamma$  est un multi-ensemble d'éléments de  $\mathcal{F}$  (c'est-à-dire un ensemble où les éléments peuvent apparaître plusieurs fois et où leur multiplicité est comptée).

Les quatre règles de déduction de la logique linéaire sont les suivantes (où  $x \in \mathcal{V}$ ,  $A, B \in \mathcal{F}$  et  $\Gamma, \Delta \subseteq \mathcal{F}$ ) :

$$\frac{}{\vdash x, x^\perp} \text{ ax} \quad \frac{\vdash \Gamma, A \quad \vdash A^\perp, \Delta}{\vdash \Gamma, \Delta} \text{ cut} \quad \frac{\vdash \Gamma, A \quad \vdash B, \Delta}{\vdash \Gamma, A \otimes B, \Delta} \otimes \quad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \wp B} \wp$$

Une occurrence de la règle (cut) dans une preuve est appelée une **coupure**. Enfin, on note  $A \multimap B$  la formule  $A^\perp \wp B$ . L'opérateur  $\multimap$  est appelé **implication linéaire**.

1. Montrer que  $\vdash ((x \wp y) \wp z) \multimap (x \wp (y \wp z))$  est prouvable en logique linéaire.
2. Montrer que pour  $A \in \mathcal{F}$ ,  $\vdash A, A^\perp$  est prouvable en logique linéaire.

On considère la règle d'inférence  $\frac{}{\vdash A, A^\perp} \text{ axF}$ , où  $A$  est une formule.

3. Montrer que la règle  $\frac{\vdash \Gamma, A \multimap B \quad \vdash A, \Delta}{\vdash \Gamma, B, \Delta} \multimap_e$  est dérivable en logique linéaire à laquelle on ajoute la règle (axF).
4. Montrer que  $\vdash A \multimap (A \otimes A)$  n'est pas prouvable sans coupure.

**Définition**

On appelle **réseau** un graphe orienté dont certaines arêtes sont pendantes (avec un sommet source, mais pas de destination), dont les sommets sont étiquetés par  $(ax)$ ,  $(cut)$ ,  $(\otimes)$  ou  $(\wp)$  et dont les arêtes sont étiquetées par des formules :

- un sommet  $(ax)$  n'a pas d'arête entrante et deux arêtes sortantes, étiquetées par  $x$  et  $x^\perp$  respectivement, où  $x \in \mathcal{V}$  ;
- un sommet  $(cut)$  n'a pas d'arêtes sortantes et deux arêtes entrantes, étiquetées par  $A$  et  $A^\perp$ , où  $A \in \mathcal{F}$  respectivement ;
- un sommet  $(\otimes)$  (resp.  $(\wp)$ ) a deux arêtes entrantes étiquetées par  $A$  et  $B$  respectivement, où  $A, B \in \mathcal{F}$ , et une arête sortante étiquetée par  $A \otimes B$  (resp.  $A \wp B$ ).

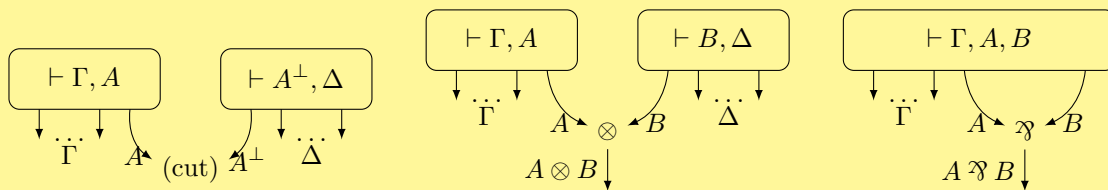
Les étiquettes des arêtes pendantes d'un réseau sont appelées les **conclusions**.

5. Montrer qu'un réseau est acyclique.

**Définition**

Un **réseau de preuve** est un réseau dont les conclusions forment le multi-ensemble  $\Gamma$ , associé inductivement à une preuve d'un séquent de la manière suivante :

- une règle  $(ax)$  forme un sommet  $(ax)$  avec les mêmes conclusions ;
- pour les autres règles, on associe les réseaux associés aux preuves des prémisses selon les schémas suivants :



6. Montrer qu'il existe un réseau qui n'est pas un réseau de preuve.

**Définition**

Un **interrupteur** d'un réseau  $\mathcal{R}$  (pas nécessairement de preuve) est un graphe non orienté obtenu à partir de  $\mathcal{R}$  en :

- supprimant toutes les arêtes pendantes ;
- supprimant une des deux arêtes entrantes de chaque nœud  $(\wp)$  ;
- désorientant toutes les arêtes restantes.

On considère le théorème suivant.

**Théorème : Danos-Régnier**

Un réseau  $\mathcal{R}$  est un réseau de preuve si et seulement si tout interrupteur de  $\mathcal{R}$  est un arbre.

7. Montrer le sens direct du théorème de Danos-Régnier.

On admet la réciproque du théorème de Danos-Régnier, et on cherche à montrer le théorème suivant.

**Théorème : Élimination des coupures**

Tout séquent  $\vdash \Gamma$  prouvable en logique linéaire est prouvable en logique linéaire privée de la règle  $(cut)$ .

On considère pour cela un réseau de preuve  $\mathcal{R}$ , contenant éventuellement des sommets  $(cut)$ . On cherche à transformer ce réseau en supprimant les nœuds  $(cut)$ , tout en conservant le critère de Danos-Régnier (c'est-à-dire que les interrupteurs restent des arbres), et sans changer les conclusions.

8. Montrer qu'un sommet  $(cut)$  a pour prédécesseur un sommet  $(\otimes)$  si et seulement si il a pour prédécesseur un sommet  $(\wp)$ . En déduire une transformation qui supprime un tel sommet  $(cut)$  pour obtenir un réseau ayant strictement moins de sommets.
9. Montrer le théorème d'élimination des coupures.

## Chapitre 57

# (ENS) Composition Monadique \*\*\* (ENS 24, corrigé très partiellement — oral - 191 lignes)

\*\*\*

Ocaml, Programmation fonctionnelle,  
sources : `enscompositionmonadique.tex`

## Composition Monadique

On se place dans un langage récursif, purement fonctionnel, avec des définitions de types inductifs et un système de types polymorphes similaires à ceux d'*OCaml*. On pourra utiliser indifféremment du pseudocode fonctionnel ou de la syntaxe *OCaml*.

On suppose l'existence :

- d'une fonction identité `id` polymorphe de type  $A \rightarrow A$  ;
- d'un opérateur `(o)` :  $(A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C)$  de composition de fonctions, noté `o` dans sa version infixe ;
- d'un opérateur `↦` de définition de fonction anonyme (`fun` en *OCaml*) ;
- un opérateur de reconnaissance de motifs (`match... with` en *OCaml*) ; et
- de types de bases usuels, comme `int` le type des entiers, `string` le type des chaînes de caractères, et `unit` le type de l'unique élément `()`.

Un constructeur de type est une entité informatique qui prend un type et renvoie un type.

On définit par exemple le constructeur de type `liste` en définissant le type polymorphe `liste A` (où `A` est une variable de type). À titre d'exemple, on présente une définition de la fonction `longueur` de type `liste A → int` en pseudocode :

```
liste A = Nil | Cons A (liste A)
longueur : liste A -> int
longueur Nil = 0
longueur (Cons _ qu) = 1 + (longueur qu)
```

et en *OCaml* :

```
type 'a liste = Nil | Cons of ('a * 'a liste)
let rec longueur (l : 'a liste) : int = match l with
  Nil -> 0
  | Cons (_, qu) -> 1 + (longueur qu)
```

**Question 1.** Définir un constructeur de type `option` tel qu'un élément de `option A` est soit vide, soit un élément de type `A`.

Un constructeur de types `F` est un *foncteur* quand il existe une fonction polymorphe `fmap` :  $(A \rightarrow B) \rightarrow (FA) \rightarrow (FB)$

qui vérifie les lois suivantes (pour tout  $f$  et  $g$ ) :

```
fmap id = id
fmap (f o g) = (fmap f) o (fmap g)
```

Attention : Dans ce sujet, on ne demande pas de preuves que les lois sont respectées.

**Question 2.** Montrer que `option` et `liste` sont des foncteurs.

Un foncteur  $M$  est une *monade* quand il existe deux fonctions polymorphes

```
return : A -> (M A)
bind : (M A) -> (A -> (M B)) -> (M B)
```

qui vérifient les lois suivantes, pour tout  $f, g$  et  $x$  :

```
bind (return x) f = f x
bind x return = x
bind (bind x f) g = bind x (y -> (bind (f y) g))
```

**Question 3.** Montrer que `option` et `liste` sont des monades.

Pour tout type  $S$ , on définit (`etat S`)  $A$  comme étant le type polymorphe  $S \rightarrow (S, A)$   
 On suppose l'existence d'un type abstrait `memoire` représentant une table d'association entre `string` et `int` avec les deux fonctions suivantes :

```
trouve : string -> memoire -> (option int)
majour : string -> int -> memoire -> memoire
```

La première recherche la valeur associée à une clef, la seconde met-à-jour un couple clef-valeur.

**Question 4.** Montrer que `etat S` est une monade pour tout  $S$ .

En déduire l'écriture avec `bind` d'une *procédure* qui prend en entrée trois clefs, récupère successivement deux valeurs d'une mémoire à partir des deux premières clefs puis associe dans la mémoire la somme des deux valeurs récupérée à la troisième clef.

On donne une définition alternative des monades :

Un foncteur  $M$  est une *monade* quand il existe deux fonctions polymorphes

```
return : A -> (M A)
join : M (M A) -> M A
```

qui vérifient les lois suivantes, pour tout  $f, g$  et  $x$  :

```
join o return = id
join o (fmap return) = id
join o join = join o (fmap join)
```

**Question 5.** Montrer que les deux définitions sont équivalentes.

Si  $M_1$  et  $M_2$  sont deux constructeurs de types, on définit (`compose  $M_1 M_2$` )  $A = M_1 (M_2 A)$ .

**Question 6.** Donner une condition suffisante pour que (`compose  $M_1 M_2$` ) soit un foncteur.

**Question 7.** Donner une condition suffisante pour que (`compose  $M_1 M_2$` ) soit une monade.  
 En déduire une nouvelle écriture, plus simple, de la procédure de la question 4.

# Chapitre 58

## (ENS) Dédution de messages \*\*\* (ENS 24, corrigé — oral - 232 lignes)

\*\*\*

Dédution naturelle, Réduction,  
sources : `ensdeductiondemessages.tex`

### Dédution de messages

Nous souhaitons nous intéresser au problème suivant appelé problème de déduction :

**entrée** un ensemble fini de termes clos  $T$  et un terme clos  $u$

**sortie** est-ce que  $u$  est déductible depuis  $T$ , noté  $T \vdash u$  ?

**Terme et sous-terme** Nous nous intéressons aux termes construits inductivement à partir du symbole binaire  $f(\cdot, \cdot)$ , d'un ensemble infini dénombrable de constantes  $\mathcal{C}$ , et d'un ensemble infini dénombrable de variables  $\mathcal{V}$ .

Un terme est donc généré par la grammaire :  $t, t_1, t_2 := v \in \mathcal{C} \mid x \in \mathcal{V} \mid f(t_1, t_2)$ .

Si un terme ne contient pas de variable, alors ce terme est dit clos.

Étant donné un terme  $t$  nous notons  $st(t)$  l'ensemble des sous-termes de  $t$ , i.e., le plus petit ensemble  $S$  tel que  $t \in S$ , et si  $f(t_1, t_2) \in S$  alors  $t_1, \dots, t_n \in S$ .

**Règle d'inférence** Une règle d'inférence est une règle de déduction de la forme :

$$\frac{T \vdash t_1 \dots T \vdash t_n}{T \vdash t} \text{ REGLE}$$

où  $T$  est un ensemble fini de termes et  $t_1, \dots, t_n, t$  sont des termes.

Un système d'inférence  $\mathcal{I}$  est un ensemble fini de règles d'inférence.

**Preuve** Une *preuve* (ou *arbre de preuve*)  $\Pi$  de  $T \vdash u$  dans  $\mathcal{I}$  est un arbre tel que :

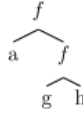
- chaque feuille est étiquetée avec un terme  $v \in T$  ;
- pour chaque noeud ayant pour étiquette  $v_0$  et enfants  $v_1, \dots, v_n$  il existe une règle d'inférence dans  $\mathcal{I}$  ayant pour conclusion  $v_0$  et hypothèses  $v_1, \dots, v_n$  (à instanciation près des variables) ;
- la racine de l'arbre est étiquetée par  $u$ .

La taille d'une preuve  $\Pi$ , notée  $size(\Pi)$ , est son nombre de noeuds.  $Termes(\Pi)$  dénote l'ensemble des étiquettes, i.e., termes, apparaissant dans  $\Pi$ .

Lorsque  $T \vdash u$  nous disons que  $u$  est déductible à partir de l'ensemble de termes  $T$ .

$$\frac{\text{si } u \in T}{T \vdash u} \text{Ax} \quad \frac{T \vdash x \quad T \vdash y}{T \vdash f(x, y)} \text{APP-F} \quad \frac{T \vdash f(x, y) \quad T \vdash y}{T \vdash x} \text{RED-F}$$

Figure 1 - Système d'inférence  $\mathcal{I}_0$

Figure 2 - Représentation sous forme d'arbre du terme  $f(a, f(g, h))$ 

**Question 1.** Soit  $T = \{f(f(a, k_1), k_2), k_2, f(k_1, k_2)\}$ . Donner l'arbre de preuve de  $T \vdash a$  dans  $\mathcal{I}_0$ .

**Question 2.** Soit  $T$  un ensemble de termes clos. Montrer que pour tout  $T \vdash u$  dans  $\mathcal{I}_0$ , un arbre de preuve de taille minimale  $\Pi$  de  $T \vdash u$  contient seulement des termes issus de  $\text{st}(T \cup \{u\})$ , i.e.,  $\text{Termes}(\Pi) \subseteq \text{st}(T \cup \{u\})$ .  
Montrer de plus que si  $\Pi$  est réduit à une feuille ou termine par une règle Ax ou RED-F alors il contient uniquement des termes issus de  $\text{st}(T)$ , i.e.  $\text{Termes}(\Pi) \subseteq \text{st}(T)$ .

**Question 3.** En déduire que le problème de déduction dans  $\mathcal{I}_0$  est décidable en temps polynomial. Nous considérerons que la taille du problème est :  $\text{size}(T, u) = |\text{st}(u)| + \sum_{t \in T} |\text{st}(t)|$ .

On définit le problème HORN-SAT :

entrée une formule  $\Phi$  étant une conjonction finie de clauses de Horn

sortie est-ce que  $\Phi$  satisfiable ?

Une clause de Horn est une formule du calcul propositionnel qui contient au plus un littéral positif. Une clause de Horn peut donc avoir trois formes :

- un littéral positif et aucun négatif :  $C = (\text{true} \Rightarrow x)$
- un littéral positif et au moins un littéral négatif :  $C = (x_1 \wedge \dots \wedge x_n \Rightarrow x)$
- aucun littéral positif :  $C = (x_1 \wedge \dots \wedge x_n \Rightarrow \text{false})$ .

On admettra que HORN-SAT est P-complet, c'est-à-dire (intuitivement) que tout problème de décision dans P admet une réduction linéaire à HORN-SAT.

**Question 4.** Montrer que le problème de déduction dans  $\mathcal{I}_0$  est P-complet.

Nous souhaiterions maintenant nous intéresser au même problème mais en ajoutant le ou-exclusif. Un terme est donc maintenant généré par la grammaire :

$$t, t_1, t_2 := v \in \mathcal{C} | x \in \mathcal{V} | f(t_1, t_2) | t_1 \oplus t_2$$

Nous ne prouverons pas ici que le problème de déduction est encore décidable en temps polynomial. Nous nous intéresserons à prouver une étape de la preuve : étant donné un ensemble de termes  $T$  et un terme  $t$ , est-ce que  $T \vdash t$  en utilisant uniquement les règles GX et Ax' ?

$$\frac{T \vdash u_1 \dots T \vdash u_n}{T \vdash u_1 \oplus \dots \oplus u_n} \text{GX} \quad \frac{\text{si } u = ACv \text{ et } v \in T}{T \vdash u} \text{Ax}$$

On note  $=_{AC}$  la plus petite relation telle que :

$$\begin{array}{lll} \text{(refl.) } x = x & \text{(sym.) } (x = y) \Rightarrow (y = x) & \text{(trans.) } (x = y) \wedge (y = z) \Rightarrow (x = z) \\ \text{(comm.) } x \oplus y = x \oplus y & \text{(assoc.) } x \oplus (y \oplus z) = (x \oplus y) \oplus z & \\ \text{(congr.) } (x_1 = y_1) \wedge (x_2 = y_2) \Rightarrow f(x_1, x_2) = f(y_1, y_2) & & \end{array}$$

**Question 5.** Soit  $u$  et  $v$  deux termes clos. Donner un algorithme en temps polynomial qui décide si  $u =_{AC} v$ .

**Question 6.** Soit  $T$  un ensemble de termes clos. Soit  $t$  un terme clos. Montrer que  $T \vdash t$  dans  $\{\text{GX}, \text{Ax}'\}$  est décidable en temps polynomial.

# Chapitre 59

## (ENS) Filtrage par motif en OCaml \*\*\* (ENS 24, corrigé — oral - 323 lignes)

\*\*\*

Ocaml,  
sources : `ensfiltrage0caml.tex`

### Filtrage par motif en OCaml

Ce sujet s'intéresse à modéliser le comportement du filtrage par motif utilisé dans la construction `match ... with ...` d'OCaml.

On s'intéresse à un sous-ensemble d'OCaml. Les constructeurs des types algébriques sont notés  $C(e_1, \dots, e_k)$ , où  $k$  est l'arité du constructeur  $C$ . Dans l'exemple des listes ci-dessous, le premier constructeur de liste (`Empty`) est d'arité 0, le second (`Cons`) est d'arité 2.

```
type 'a list = Empty | Cons of 'a * 'a list
```

Dans la suite, on considère que toutes les valeurs OCaml sont des constructeurs  $v ::= C(v_1, \dots, v_k)$ . En particulier, les constantes `true` et `false` sont des valeurs du type `bool = true | false`.

Les motifs sont  $m ::= v \mid |C(m_1, \dots, m_k)| (m_1 \mid m_2)$ , i.e :

- Des identifiants de variables (cas  $v$ ).
- Le motif joker `_`.
- Un constructeur appliqué à  $k$  motifs.
- La disjonction de deux motifs.

On introduit une notion de matrice de filtrage, selon la transformation illustrée ci-dessous. Les expressions à droite des motifs ( $a_i$ ) sont appelées des actions.

```
match (e1, ..., em) with  
| m1,1, ..., m1,m → a1  
| ...  
| mn,1, ..., mn,m → an
```

$$\rightarrow \begin{pmatrix} e_1 & \dots & e_m & & \\ m_{1,1} & \dots & m_{1,m} & \rightarrow & a_1 \\ \dots & \dots & \dots & \rightarrow & \dots \\ m_{n,1} & \dots & m_{n,m} & \rightarrow & a_n \end{pmatrix}$$

Étant donné une expression  $e = C(e_1, \dots, e_k)$ , on définit des opérateurs d'accès au constructeur : `constr(e) = C` et d'accès au  $i$ -ème champ : `#i(e) = ei` (si  $1 \leq i \leq k$ ).

Un *environnement* est une fonction partielle des variables aux valeurs.

**Question 1.** Définir une relation  $m \leq_\sigma v$  établissant la compatibilité entre un motif  $m$  et une valeur  $v$ , étant donné un environnement  $\sigma$ .

À titre d'exemple, `Cons(1, x) ≤σ Cons(1, Cons(2, Empty))` lorsque  $\sigma(x) = \text{Cons}(2, \text{Empty})$ .

Dans la suite, on note  $m \leq v$  si et seulement si  $\exists \sigma, m \leq_{\sigma} v$ .

**Question 2.** On note  $Match((v_1, \dots, v_m), M)$  le résultat du filtrage de la matrice  $M$  sur le vecteur de valeurs  $(v_1, \dots, v_m)$ .

- (a) Soit  $a_i$  une action et  $\sigma$  un environnement. Sous quelle(s) condition(s)  $Match((v_1, \dots, v_m), M) = (a_i, \sigma)$  ?
- (b) Y a-t-il d'autres cas de définition de  $Match$  ?

**Question 3.**

- (a) Décrire informellement comment éliminer le motif `joker` sans ajouter de cas, sur l'exemple de `len` ci-dessous.

```
let rec len l =
  match l with
  | Empty -> 0
  | Cons(_, tl) -> 1 + (len tl)
```

- (b) Définir cette transformation sur les matrices de filtrage.
- (c) Donner un critère justifiant de la correction de la transformation.

**Question 4.** Soit  $F$  une matrice de filtrage. On note  $S(c, F)$  l'opérateur qui transforme la matrice de filtrage en supposant que  $e_1$  (la première expression filtrée par  $F$ ) commence par un constructeur  $c$  d'arité  $a$ .

- (a) Illustrer le résultat de  $S(\text{Cons}, F)$  sur la matrice de filtrage induite par le code ci-dessous.

```
let rec merge l1 l2 = match l1, l2 with
| Empty, l | l, Empty -> l
| Cons(h1, t1), Cons(h2, t2) ->
  if h1 < h2 then Cons(h1, merge t1 l2)
  else Cons(h2, merge l1 t2)
```

- (b) Décrire la transformation opérée par  $S(c, F)$ .
- (c) Donner un critère justifiant de la correction de la transformation.

**Question 5.** Donner une fonction  $C$  permettant de transformer ces matrices de filtrage vers un langage OCaml modifié, où les filtres `match ... with ...` sont supprimés au profit de `switch` sur le constructeur d'une expression :

```
switch constr(e) with
  case ... -> ...
  ...
  default -> ...
```

**Question 6.**

- (a) Cette transformation a-t-elle un intérêt ?
- (b) L'algorithme de transformation peut-il être amélioré ?

**Question 7.** Prouver la terminaison et la correction de l'algorithme en question 5.

# Chapitre 60

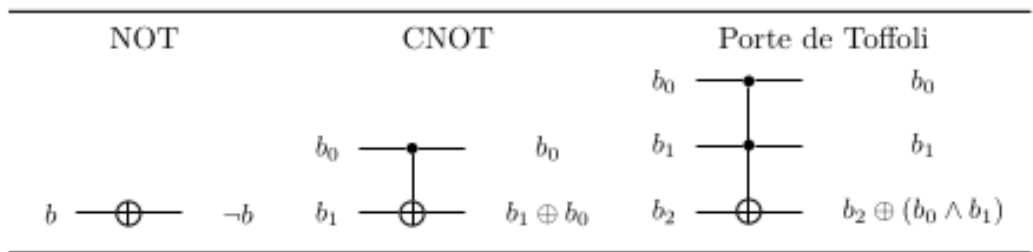
## (ENS) Implémentation des circuits réversibles \*\*\* (ENS 24, corrigé — oral - 218 lignes)

\*\*\*

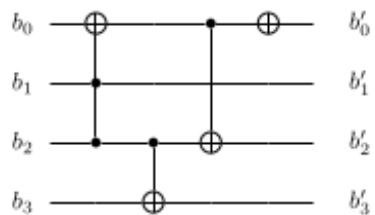
Circuits,  
sources : `enscircuitsreversibles.tex`

### Implémentation des circuits réversibles

On rappelle que l'opération de OU exclusif (XOR) est définie par :  $a \oplus b = (a \vee b) \wedge (\neg a \vee \neg b)$ , et qu'on a la propriété de distributivité suivante :  $(a \oplus b) \wedge c = (a \wedge c) \oplus (b \wedge c)$ . On définit les *portes logiques réversibles* suivantes.



Soit  $n \in \mathbb{N}$ . Un *circuit logique réversible* sur  $n$  bits est une séquence de portes logiques réversibles appliquées sur  $n$  entrées booléennes, qui seront numérotées de 0 à  $n - 1$ . À toute porte logique du circuit correspond une fonction de  $\{0, 1\}^n$  vers  $\{0, 1\}^n$  qui modifie la valeur des entrées booléennes auxquelles elle est appliquée, selon sa définition donnée ci-dessus. Ici  $b_0$  (porte CNOT) et respectivement  $b_0, b_1$  (porte de Toffoli) sont nommés les *bits de contrôle*. À tout circuit réversible  $\mathcal{C}$  correspond une fonction  $f_{\mathcal{C}} : \{0, 1\}^n \rightarrow \{0, 1\}^n$ , obtenue en composant les fonctions de chaque porte logique, dans l'ordre de lecture de gauche à droite. On dit que  $\mathcal{C}$  implémente la fonction  $f_{\mathcal{C}}$ . Ainsi, le circuit suivant :



implémente la fonction :

$$(b_0, b_1, b_2, b_3) \mapsto (\neg(b_0 \oplus (b_1 \wedge b_2)), b_1, b_2 \oplus b_0 \oplus (b_1 \wedge b_2), b_3 \oplus b_2)$$

Deux circuits réversibles  $\mathcal{C}_1, \mathcal{C}_2$  sur  $n$  bits peuvent être composés en écrivant les portes de  $\mathcal{C}_2$  suivies des portes de  $\mathcal{C}_1$  ; on implémente alors la fonction  $f_{\mathcal{C}_2} \circ f_{\mathcal{C}_1}$ .

**Question 1.**

1. Montrer que tout circuit réversible  $\mathcal{C}$  constitué d'une seule porte (NOT, CNOT ou Toffoli) implémente une permutation. Quel est son inverse ?
2. Comment implémenter l'inverse d'un circuit réversible quelconque ?

Une porte de Toffoli à  $k$  contrôles implémente la fonction :

$$(b_0, \dots, b_{k-1}, b_k) \mapsto \left( b_0, \dots, b_{k-1}, b_k \oplus \bigwedge_{i=0}^{k-1} b_i \right)$$

**Question 2.** Montrer que dans un circuit sur  $n$  bits, une porte de Toffoli à  $n - 2$  contrôles peut être implémentée à l'aide de 2 portes de Toffoli à  $n - 3$  contrôles, et deux portes de Toffoli.

Dans la suite de cet exercice, on va démontrer le théorème suivant :

Pour tout  $n \geq 7$ , une permutation  $\Pi$  de  $\{0, 1\}^n$  est paire si et seulement si il existe un circuit réversible  $\mathcal{C}$  sur  $n$  bits tel que  $\Pi = f_{\mathcal{C}}$ .

On rappelle qu'une permutation est paire (de signature 1) si et seulement si toutes ses décompositions en transpositions ont un nombre de transpositions pair.

**Question 3.** Trouver des contre-exemples simples pour  $n = 2$  et  $n = 3$ .

**Question 4.** Soit  $\mathcal{C}$  un circuit réversible sur  $n \geq 4$  bits ne comportant qu'une seule porte logique. Soit  $S$  l'ensemble des cycles de  $f_{\mathcal{C}}$ .

1. Montrer qu'il existe une partition  $S = S_0 \cup S_1$  et une bijection  $S_0 \rightarrow S_1$  préservant la longueur des cycles.
2. En déduire une implication du théorème.

**Question 5.**

1. Soit  $n \geq 4$ . Soit  $a, b \in \{0, 1\}^n, a \neq b$ . Montrer qu'il existe un circuit  $\mathcal{C}$  n'utilisant que des portes NOT et CNOT tel que  $f_{\mathcal{C}}(a) = (1, 1, \dots, 1)$  et  $f_{\mathcal{C}}(b) = (0, 1, \dots, 1)$ .
2. Soit  $c, d \in \{0, 1\}^{n-1}, c \neq d$ . Montrer qu'il existe un circuit  $\mathcal{C}$  tel que  $f_{\mathcal{C}}$  est la paire de transpositions  $((0, c)(0, d))((1, c)(1, d))$ .
3. En déduire que si  $\Pi$  est une permutation de  $\{0, 1\}^n$  laissant au moins un bit invariant, il existe un circuit réversible  $\mathcal{C}$  tel que  $\Pi = f_{\mathcal{C}}$ .

**Question 6.** Montrer que pour  $n \geq 6$ , toute permutation paire de  $\{0, 1\}^n$  peut s'écrire comme une composition de paires de transpositions disjointes, c'est-à-dire :

$$\Pi = (x_0 y_0) (z_0 t_0) (x_1 y_1) (z_1 t_1) \dots (x_k y_k) (z_k t_k)$$

où pour tout  $i, x_i, y_i, z_i, t_i$  sont distincts deux à deux (mais on peut avoir par exemple  $x_0 = y_1$ ).

**Question 7.** Soit  $n \geq 7$  et soit  $x, y, z, t$  des éléments de  $\{0, 1\}^n$  distincts deux à deux. Montrer qu'il existe un circuit réversible  $\mathcal{C}$  implémentant une permutation  $P$  telle que :

$$P(x) = (0, 0, 1, \dots, 1), \quad P(y) = (0, 1, 1, \dots, 1), \quad P(z) = (1, 0, 1, \dots, 1), \quad P(t) = (1, 1, 1, \dots, 1)$$

En déduire un circuit réversible qui implémente la paire de transpositions  $(xy)(zt)$ .

# Chapitre 61

## (ENS) Résolution d'inéquations linéaires

### \*\*\* (ENS 24, corrigé — oral - 195 lignes)

\*\*\*

Ocaml, Algorithmique, Complexité,

sources : `ensresolutioninequationslineaires.tex`

### Résolution d'inéquations linéaires

Soit  $X = \{x_1, \dots, x_n\}$  un ensemble de variables et  $S = \{L_1, \dots, L_m\}$  un ensemble de fonctions linéaires en les variables de  $X$  à coefficients rationnels, assimilé à un système d'inéquations :

$$\forall i, L_i(x_1, \dots, x_n) \leq 0$$

Si deux équations peuvent se réécrire sous la forme :  $L \leq x$  et  $x \leq U$  pour  $x \in X$ , où les expressions  $L$  et  $U$  ne contiennent pas  $x$ , on définit le « $x$ -résultant» de  $L$  et  $U$  comme l'inéquation :  $L \leq U$  (ou de manière équivalente :  $L - U \leq 0$ ).

Les nombres rationnels manipulés dans cet exercice n'étant a priori pas de taille constante, la complexité en temps des algorithmes sera comptée uniquement en opérations rationnelles (additions, multiplications, divisions, etc.)

Notre objectif est de résoudre le problème suivant de *satisfiabilité des inéquations rationnelles* :

Soit  $S$  un ensemble d'inéquations à au plus deux variables, déterminer si  $S$  est satisfiable, i.e., s'il existe une valuation  $X \rightarrow \mathbb{Q}$  satisfaisant toutes les inéquations de  $S$ .

**Question 1.** Le système suivant :

$$\begin{cases} 2x_1 + x_2 + 1 \leq 0 \\ 2x_2 \leq 0 \\ -x_1 \leq 0 \\ -x_2 \leq 0 \end{cases}$$

est-il satisfiable ?

Soit  $S$  un système d'inéquations linéaires,  $x$  une variable de  $S$ , et  $T \subseteq S$  l'ensemble des inéquations ne contenant pas  $x$ . Soit  $S'$  l'ensemble des  $x$ -résultats de paires d'inéquations dans  $S \setminus T$ . On définit un nouveau système  $FM(S, x) := T \cup S'$ . L'opération  $FM$  est appelée l'*élimination de Fourier-Motzkin*.

**Question 2.** Soit  $S$  un système d'inéquations linéaires et  $x$  une variable de  $S$ . Montrer que  $S$  est satisfiable si et seulement si  $FM(S, x)$  l'est.

Posons maintenant :  $y := \max_i L_i(y_1, \dots, y_{n-1})$ . Alors toutes les inéquations  $L_i \leq x$  sont satisfaites par  $y, y_1, \dots, y_{n-1}$  par définition de  $y$ . De plus pour tout  $j, y \leq U_j(y_1, \dots, y_{n-1})$ . En effet, sinon il existerait  $i$  et  $j$  tels que  $L_i(y_1, \dots, y_{n-1}) > U_j(y_1, \dots, y_{n-1})$  ce qui contredirait l'hypothèse.

S'il n'existe pas d'inéquation de la forme  $L_i \leq x$  (respectivement, de la forme  $x \leq U_j$ ) alors on choisit  $y := \min_j U_j(y_1, \dots, y_{n-1})$  (respectivement, le même  $y$  que précédemment).

**Question 3.** En déduire un algorithme pour la satisfiabilité des inéquations rationnelles (il n'est pas nécessaire d'en faire une description détaillée). Donner une borne simple sur sa complexité en temps.

Dans la suite de l'exercice, on considère des inéquations à au plus deux variables. On s'intéresse à l'algorithme suivant.

```
Entrée:  $S$  un système de  $m$  inéquations, contenant  $n$  variables au total
pour  $i=1$  à  $\lceil \log_2 n \rceil + 2$  faire
    Soit  $S' := \cup_{x \in X} FM(S, x)$  (si des inéquations apparaissent en double, on les ←
        simplifie)
     $S \leftarrow S \cup S'$ 
    Si une inéquation de  $S$  est insatisfiable, renvoyer "insatisfiable"
fin pour
Renvoyer "satisfiable"
```

Nous admettons la propriété (P) suivante :

(P) Soit  $S$  un système d'inéquations en  $k$  variables. Si tout sous-ensemble de  $S$  avec  $k + 1$  inéquations est satisfiable, alors  $S$  est satisfiable.

Si  $V \subseteq X$ , nous noterons également  $S_V$  la restriction de  $S$  aux inéquations ne contenant que des variables de  $V$ .

**Question 4.** Soit  $S$  un système insatisfiable minimal, c'est-à-dire tel que tout sous-système  $S' \subsetneq S$  est satisfiable.

1. Montrer qu'aucune variable n'apparaît que dans une seule inéquation.
2. Montrer qu'au plus deux variables apparaissent dans trois inéquations ou plus.

**Question 5.** Soit  $S$  un système insatisfiable minimal sur un ensemble de  $k > 3$  variables  $X$ , et soit  $S' := \cup_{x \in X} FM(S, x)$ .

1. Montrer qu'il existe un ensemble  $X' \subseteq X$  de  $\lceil k/2 \rceil - 1$  variables telles qu'aucune paire  $\{x_1, x_2\}$  de  $X'$  n'apparaît dans une inéquation de  $S$ .
2. Montrer que  $(S \cup S')_{X \setminus X'}$  est insatisfiable.

**Question 6.** Soit  $f(x) = \lfloor x/2 \rfloor + 1$ . On pose  $f^0(n) = n$  et on admet que pour tout entier naturel  $n$ ,  $f^{\lceil \log_2 n \rceil}(n) = 2$ .

1. Soit  $S$  un système insatisfiable en entrée de l'algorithme. Montrer que pour tout  $k \geq 1$ , à la sortie de la  $k$ -ième itération de la boucle, il existe un sous-ensemble  $X_k \subseteq X$  de taille  $\leq f^k(n)$  tel que  $S_{X_k}$  est insatisfiable.
2. En déduire que l'algorithme est correct.
3. Donner une borne simple sur sa complexité (à facteur polynomial près).

On s'autorise à modifier l'algorithme en ajoutant entre l'étape 4 et 5 :  $S \leftarrow \text{Simplifie}(S)$ , où Simplifie est une opération transformant  $S$  en un système équivalent avec moins d'inéquations.

**Question 7.** On considère maintenant le cas particulier où les coefficients des équations sont  $+1, -1$  ou  $0$ . En utilisant un choix opportun pour Simplifie, proposer un algorithme en temps polynomial en  $n$ .

# Chapitre 62

## (ENS) Mots partiels et Théorème de Dilworth \*\*\* (ENS 24, partiellement corrigé, niveau inapproprié, à déboguer — oral - 184 lignes)

\*\*\*

Graphes, Réduction, Langages,  
sources : ensmotspartiels.tex

### Mots partiels et Théorème de Dilworth

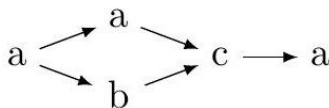
Soit  $\Sigma$  un ensemble fini de lettres, et  $\Sigma^*$  l'ensemble des mots de  $\Sigma$ , qui sont des séquences finies de lettres. Nous généralisons la notion de mots à des structures partiellement ordonnées.

Soit  $G = (E, R)$  un graphe dirigé fini :  $E$  est un ensemble fini et  $R$  un sous ensemble de  $E \times E$ . Un *chemin* de  $G$  est une séquence  $x_0, x_1, \dots, x_n$  de sommets tels que pour tout  $0 \leq i < n$ ,  $(x_i, x_{i+1}) \in R$ . Le graphe  $G$  est dit *acyclique* s'il n'existe pas de chemin non réduit à un seul sommet tel que  $x_n = x_0$ .

Un *mot partiel* sur  $\Sigma$  est un triplet  $(E, R, \mu)$  avec  $(E, R)$  un graphe fini acyclique et  $\mu : E \rightarrow \Sigma$ . La *taille* d'un mot partiel est le nombre d'éléments de  $E$ .

Soit  $p = (E, R, \mu)$  un mot partiel de taille  $n$ . Un mot  $u_1 \dots u_n \in \Sigma^*$  *généralise*  $p$  s'il existe  $\psi : E \rightarrow \{0, \dots, n-1\}$ , bijective, tel que pour tout  $(x, y) \in R$ ,  $\psi(x) < \psi(y)$  et si pour tout  $e \in E$ , on a  $u_{\psi(e)} = \mu(e)$ . Dans la suite on note  $\text{Total}(p)$  l'ensemble des mots qui généralisent  $p$ .

**Question 1.** Que vaut  $\text{Total}(p)$  pour  $p$  dessiné ci-dessous, où chaque sommet est résumé par son image par  $\mu$ .



Comment caractériser sur le graphe l'ordre dans lequel les lettres d'un mot  $u$  généralisant  $p$  est lu ?

**Question 2.** Donnez une borne supérieure sur la taille de  $\text{Total}(p)$  en fonction de la taille de  $\Sigma$ .

Soit  $L$  un langage de  $\Sigma^*$ . Un mot partiel  $p$  sur  $\Sigma$  *appartient* à  $\text{Partiel}(L)$  s'il existe  $v \in \text{Total}(p)$  qui appartient à  $L$ .

**Question 3.** Montrez que si  $L$  est dans  $\mathbf{P}^1$ , alors  $\text{Partiel}(L)$  appartient à  $\mathbf{NP}^2$ .

1. ND M.Péchaud i.e. s'il existe un algorithme en temps polynomial déterminant si un mot est dans  $L$

2. ND M.Péchaud i.e. s'il existe un certificat de taille polynomiale en un mot partiel  $p$  et un vérifieur en temps polynomial prenant en argument ce certificat et  $p$  et vérifiant si  $p$  est dans  $\text{Partiel}(L)$

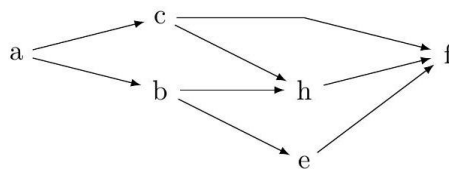
(Unary) 3-partition est un problème qui prend en entrée une suite finie de  $3n$  entiers écrits **en unaire**<sup>3</sup>  $x_1, \dots, x_{3n}$  et vérifie s'il existe une partition en  $n$  triplets dont les sommes deux à deux sont égales. On admet dans la suite que ce problème est **NP-complet**.

**Question 4.** Montrez par une réduction depuis 3-partition qu'il existe un langage  $L$  dans **P** tel que  $\text{Partiel}(L)$  est **NP-complet**<sup>4</sup>.

**Question 5.** Montrez que  $\text{Partiel}(\Sigma^*ab^*a\Sigma^*)$  est vérifiable en temps polynomial.

Soit  $p = (E, R, \mu)$  un mot partiel sur  $\Sigma$ . Une *décomposition en chemins* de  $p$  est un ensemble  $X$  de chemins tel que tout sommet de  $E$  appartient à exactement un chemin.

**Question 6.** Donnez une décomposition en chemins avec le moins de chemins possibles pour le mot partiel suivant :



**Question 7.** Soit  $L$  un langage régulier (reconnu par un automate déterministe). Montrez que  $\text{Partiel}(L)$  est vérifiable en temps  $O(n^w)$  avec  $n$  la taille du mot partiel et  $w$  la taille de sa plus petite décomposition en chemins.

Soit  $p = (E, R, \mu)$  un mot partiel sur  $\Sigma$ . Une anti-chaîne de  $p$  est un ensemble  $X$  tel qu'il n'existe pas de chemin entre deux éléments de  $X$ . On appelle largeur de  $p$  la taille de la plus grosse anti-chaîne de  $p$ .

On admet le théorème suivant :

**Théorème 1 (Dilworth).** Soit  $p$  un mot partiel. La largeur de  $p$  est égale à la taille de sa plus petite décomposition en chemins. Cette dernière est calculable en temps polynomial.

**Question 8.** En déduire que pour les mots partiels de largeur fixée, le problème  $\text{Partiel}(L)$  pour  $L$  un langage régulier est calculable en temps polynomial. Peut-on en déduire que le problème est calculable en temps polynomial, sans la contrainte que la largeur est fixée ?

4. i.e.  $\text{Partiel}(L)$  est dans la classe **NP**, et tout problème **NP** peut se réduire en temps polynomial au fait de tester si un mot partiel appartient à  $\text{Partiel}(L)$ .

# Chapitre 63

## (ENS) Terminaison de lambda ref \*\*\* (ENS 24, partiellement corrigé, niveau surréaliste — oral - 238 lignes)

\*\*\*

Ocaml,  
sources : ensterminaison\_lambda\_ref.tex

### Terminaison de $\lambda_{\text{ref}}$

Le  $\lambda$ -calcul, langage formel modélisant la programmation fonctionnelle, est défini par la syntaxe :  $M, N ::= x \mid MN \mid \lambda x.M \mid ()$

où  $x$  est issu d'un ensemble infini de variables de termes.

Ainsi, un terme  $M$  est :

- soit une *variable*  $x$ ,
- soit une *abstraction*  $\lambda x.M$  (qu'il faut comprendre comme la fonction qui au paramètre  $x$  associe le terme  $M$ ), on dit dans ce cas que la variable  $x$  est *liée* dans  $M$ ,
- soit une *application*  $MN$  (qu'il faut comprendre comme le terme - fonction -  $M$  appliqué au terme - argument -  $N$ )
- soit l'*unité*  $()$ , un terme de base.

Ces termes correspondent respectivement aux expressions OCaml suivantes :  $x$  un nom,  $\text{fun } x \rightarrow e$  une fonction anonyme,  $e1 \ e2$  une application et  $()$ , l'unité.

On note  $M\{N/x\}$  le terme obtenu en remplaçant toutes les occurrences (non-liées) de la variable  $x$  dans  $M$  par le terme  $N$ .

Une relation d' $\alpha$ -conversion  $\equiv_{\alpha}$  autorise le renommage des variables liées : si  $y \notin M$ , alors  $\lambda x.M \equiv_{\alpha} \lambda y.(M\{x/y\})$ . Dans la suite, on considèrera les termes *modulo*  $\alpha$ -conversion (on s'autorisera à renommer les variables liées dans les sous-termes), et on utilisera cette opération pour présenter des termes dans lesquels les variables liées sont distinctes deux à deux, et distinctes des variables non-liées.

Les *contextes d'évaluation* sont définis par :  $\mathbf{E} ::= [] \mid M\mathbf{E} \mid \mathbf{E}M$

Si  $M$  est un terme et  $\mathbf{E}$  un contexte, on note  $\mathbf{E}[M]$  le terme obtenu en remplaçant  $[]$  dans  $\mathbf{E}$  par  $M$ .

La sémantique (le comportement d'un terme) est donnée par une relation de réduction  $\longrightarrow$  donnée par :

1.  $(\lambda x.M)N \longrightarrow M\{N/x\}$  (on applique la fonction qui à  $x$  associe  $M$  à  $N$ , et on récupère le corps de la fonction  $M$  dans lequel l'argument  $N$  remplace le paramètre  $x$ .)
2. si  $M \longrightarrow M'$  alors  $\mathbf{E}[M] \longrightarrow \mathbf{E}[M']$  (on peut réduire dans un contexte).

Comme en OCaml, l'application est parenthésée à gauche, ainsi  $M_1M_2M_3$  désigne  $(M_1M_2)M_3$  et on écrit  $\lambda xy.M$  pour  $\lambda x.(\lambda y.M)$

$\longrightarrow^*$  désigne la clôture réflexive et transitive de  $\longrightarrow$ . Les *réduits* d'un terme  $M$  sont les éléments de l'ensemble  $\{M' \mid M \longrightarrow^* M'\}$ .

**Question 1.** Soit  $\delta = \lambda x.(xx)$  et  $I = \lambda x.x$ . Expliciter les réduits du terme  $A = (\lambda x.I)(\delta\delta)$ .

On donne un système de *types simples* au  $\lambda$ -calcul. Les types sont donnés par :

$$S, T ::= S \rightarrow T \mid \text{unit}$$

Ainsi un type est soit le type fonctionnel  $S \rightarrow T$  des fonctions de  $S$  dans  $T$  soit **unit** le type de base.

Une *hypothèse*  $x : T$  associe une variable de terme à un type. Un *contexte de typage*  $\Gamma$  est un ensemble d'hypothèses et on note  $\Gamma, x : T$  le contexte  $\Gamma \cup \{x : T\}$  quand  $x$  n'est pas dans  $\Gamma$ . Un *jugement* de typage  $\Gamma \vdash M : T$  indique que le terme  $M$  a le type  $T$  dans le contexte  $\Gamma$  (on dit qu'un terme est *typable* s'il existe  $\Gamma, T$  tel que  $\Gamma \vdash M : T$ ).

Les règles de typage permettant de déduire un jugement sont données par :

1.  $\Gamma \vdash () : \mathbf{unit}$
2.  $\Gamma, x : T \vdash x : T$
3. si  $\Gamma, x : T_1 \vdash M : T_2$  alors  $\Gamma \vdash \lambda x.M : T_1 \rightarrow T_2$
4. si  $\Gamma \vdash M : T_1 \rightarrow T_2$  et  $\Gamma \vdash N : T_1$ , alors  $\Gamma \vdash MN : T_2$

On note  $\lambda_{ST}$  l'ensemble des termes typables.

**Question 2.** Montrer que si  $M$  est typable et  $M = \mathbf{E}[N]$ , alors  $N$  est typable, puis expliquer pourquoi  $A$  n'est pas typable.

On dispose d'un ensemble infini d'adresses  $\mathcal{A}$ . Une mémoire  $\sigma$  est une fonction qui à chaque élément d'un sous-ensemble fini de  $\mathcal{A}$  (appelé son support) associe un  $\lambda$ -terme.

On définit un  $\lambda$ -calcul appelé  $\lambda_{\text{ref}}$  contenant une valeur  $()$  de type **unit** et trois opérateurs qui permettent de manipuler une mémoire, inspirés de leurs homologues en OCaml :

1. une opération **ref** de référencement qui prend un  $\lambda$ -terme, le stocke en mémoire à une nouvelle adresse  $\alpha$  et renvoie  $\alpha$ .
2. une opération **deref** (! en OCaml) de déréférencement qui prend une adresse et renvoie le terme contenu à cette adresse en mémoire.
3. une opération **assig** ( := en OCaml) d'assignation qui prend une adresse  $\alpha$  et un terme  $M$ , modifie la mémoire pour remplacer la valeur en  $\alpha$  par  $M$ , et renvoie **unit**.

**Question 3.** Donner des règles de typage pour les termes de  $\lambda_{\text{ref}}$  et les mémoires.

**Question 4.** Donner une sémantique pour  $\lambda_{\text{ref}}$  explicitant comment réduire un couple composé d'un terme typable et d'une mémoire.

Les réductions qui consomment un opérateur **ref**, **deref** ou **assig** sont appelées *impures*, les autres *pures*.

**Question 5.** Définir une fonction d'élagage  $\mathbf{p}$  qui associe à chaque terme typable de  $\lambda_{\text{ref}}$  un terme typable de  $\lambda_{ST}$  telle que si  $(M, \sigma) \rightarrow (M', \sigma)$  avec une réduction pure, alors  $\mathbf{p}(M) \rightarrow \mathbf{p}(M')$ .

Un terme  $M$  est *terminant* quand il n'existe pas de suite infinie  $(M_n)_{n \in \mathbb{N}}$  telle que

$M_0 = M$  et  $\forall n \in \mathbb{N}, M_n \rightarrow M_{n+1}$ . On admettra le résultat suivant :

Si  $M$  est un terme de  $\lambda_{ST}$ , alors  $M$  est terminant.

**Question 6.** Montrer que toute chaîne de réduction infinie de termes typables dans  $\lambda_{\text{ref}}$  contient une infinité de réductions impures. Proposer un terme de  $\lambda_{\text{ref}}$  typable et non-terminant.

On divise la mémoire en *régions* identifiée par des entiers naturels. Les opérateurs **ref** <sub>$n$</sub> , **deref** <sub>$n$</sub>  et **assig** <sub>$n$</sub>  sont maintenant indexés par la région qu'ils manipulent.

**Question 7.** Modifier le système de types pour qu'il associe à un terme typable son *effet* c'est à dire la région la plus haute qu'une réduction de ce terme peut manipuler (en effectuant une réduction d'un des trois opérateurs impurs).

**Question 8.** En contraignant les types par les régions, délimiter un sous-ensemble de  $\lambda_{\text{ref}}$  terminant.

# Chapitre 64

## (ENS) Calcul de l'arbre de dominance \*\*\* (ENS 25, non corrigé — oral - 83 lignes)

\*\*\*

Algorithmique, Arbres, Complexité,

sources : ensarbredom.tex

### Calcul de l'arbre de dominance

**Préambule :** *Le but de cette épreuve est d'évaluer la progression des candidates et candidats dans les questions, mais aussi la qualité de leur exposé des solutions, ainsi que l'autonomie dont elles ou ils font preuve pendant l'oral.*

Dans ce sujet, nous nous intéressons à des graphes orientés enracinés, i.e.,  $G = (V, E, r)$  où  $V$  est l'ensemble fini des sommets,  $E$  l'ensemble des arêtes orientées,  $r$  le sommet racine. Nous supposons que pour tout sommet  $v \in V$  il existe un chemin de  $r$  à  $v$ , noté  $r \rightarrow^* v$ .

Définition. Étant donné un graphe  $G = (V, E, r)$ , on dit que  $x \in V$  domine  $y \in V$ , noté  $x > y$ , si,  $x \neq y$  et pour tout chemin  $r = s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_n = y$  dans  $G$ , il existe  $i$  tel que  $s_i = x$ .

On dit que  $x$  est un dominant immédiat de  $y$ , noté  $\text{idom}(y)$ , si  $x > y$  et pour tout  $z \in V$  tel que  $z > y$  et  $z \neq x$ , on a  $z > x$ .

Définition. Étant donné un graphe  $G = (V, E, r)$ . On note  $A(G) = (V, E_d, r)$  où  $E_d = \{x \rightarrow y \mid x, y \in V \text{ et } x = \text{idom}(y)\}$ .

$A(G)$  est appelé l'arbre de dominance de  $G$ .

L'objectif de ce sujet est de calculer efficacement  $A(G)$ .

**Question 1.** Donner l'arbre de dominance du graphe donné en Figure 1.

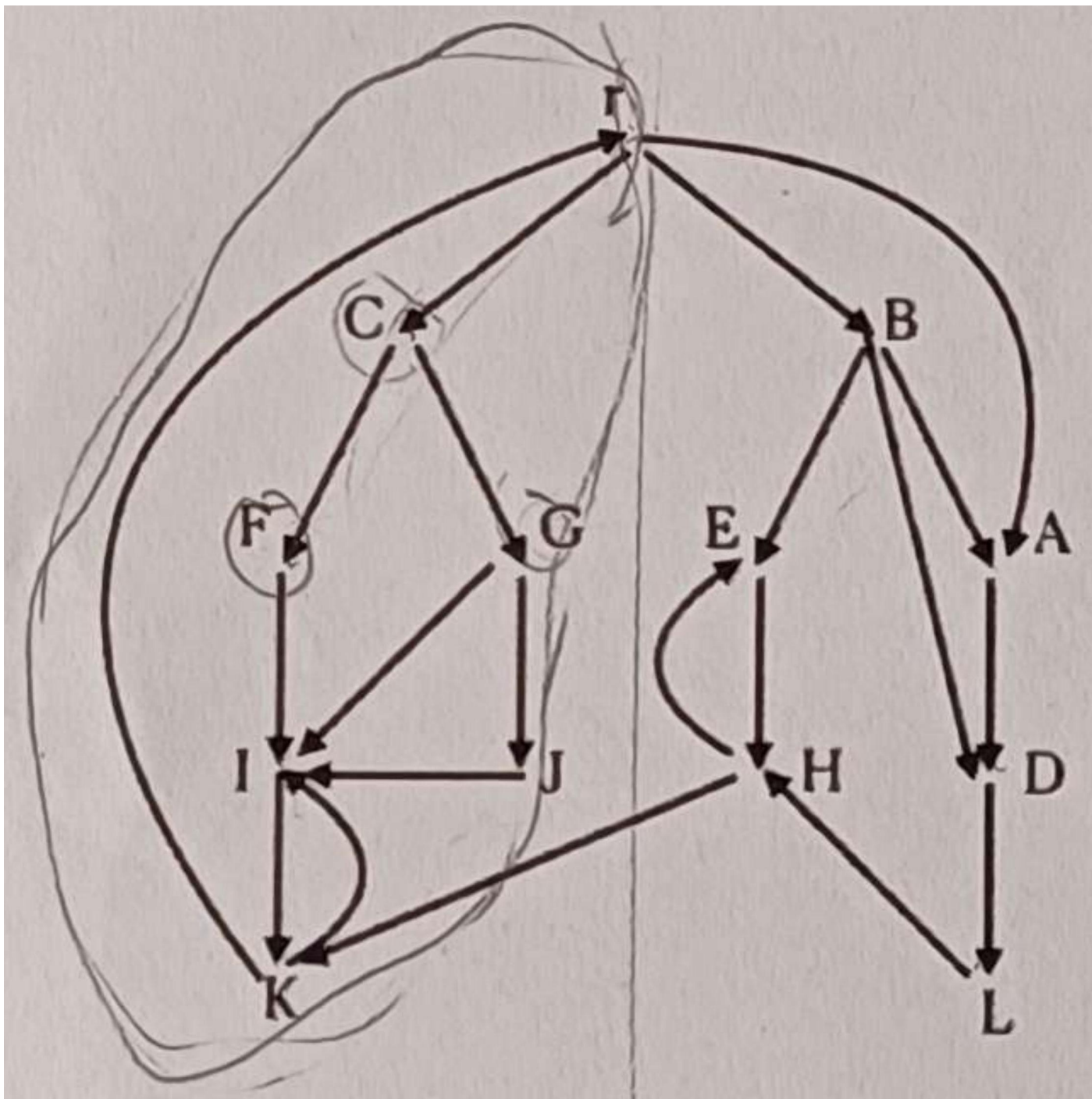


Figure 1 - Graphe de la question 1

**Question 2.** Montrer que  $A(G)$  est bien défini, i.e.,  $\text{idom}(x)$  est bien défini pour tout  $x \neq r$  et  $A(G)$  forme un arbre enraciné en  $r$ .

**Question 3. a.** Donner un algorithme basé uniquement sur des parcours, permettant de calculer  $A(G)$ .  
**b.** Quelle est sa complexité ?

Soit  $G = (V, E, r)$  un graphe. Pour chaque sommet  $x \in V$ , nous notons  $i(x)$  l'indice de visite lors d'un parcours en profondeur issu de  $r$ . Nous notons  $T(G)$  l'arbre couvrant obtenu à la fin du parcours, i.e., le graphe  $(V, E_T, r)$  où  $E_T$  est composé des arcs de liaisons durant le parcours.

Définition. Soit  $w \neq r$ . On note  $\text{sdom}(w) = \text{argmin}_v \{i(v) \mid \text{il existe un chemin } v = v_0 \rightarrow \dots \rightarrow v_k = w \text{ dans } G \text{ tel que } i(v_j) > i(w) \text{ pour } 1 \leq j \leq k-1\}$ .

**Question 4.**

**a.** Donner l'arbre couvrant ainsi que les indices de visite obtenus via un parcours en profondeur depuis  $r$  dans le graphe donné en Figure 1.  
**b.** Préciser pour chaque sommet  $w$ , le  $\text{sdom}(w)$  correspondant.

**Question 5.** Soient  $x, y \in V$  tels que  $i(x) \leq i(y)$ . Montrer que tout chemin  $x \rightarrow^* y$  dans  $G$  contient un sommet  $a$  qui est un ancêtre commun à  $x$  et  $y$  dans  $T(G)$ .

**Question 6.** Montrer que pour tout  $w \neq r$ ,  $\text{sdom}(w)$  est un ancêtre de  $w$  dans  $T(G)$ .

**Question 7.** Montrer que :

$$\begin{aligned} \text{sdom}(w) = \text{argmin}(\{i(v) \mid (v, w) \in E \text{ and } i(v) < i(w)\} \\ \cup \{i(\text{sdom}(u)) \mid i(u) > i(w) \text{ and } \exists(v \rightarrow w) \in E \text{ tel que } u \rightarrow^* v \text{ dans } T(G)\}) \end{aligned}$$

**Axiome.** soit  $w \neq r$  et  $u = \text{argmin}_u \{i(\text{sdom}(u)) \mid \text{sdom}(w) \rightarrow^+ u \rightarrow^* w \text{ dans } T(G)\}$ . On a

$$\text{idom}(w) = \begin{cases} \text{sdom}(w) & \text{si } \text{sdom}(w) = \text{sdom}(u) \\ \text{idom}(u) & \text{sinon} \end{cases}$$

**Question 8. a.** Donner un algorithme qui permet de calculer  $\text{idom}(w)$  pour tout  $w \neq r$ .

*Conseil :* nous pourrions envisager de re-construire  $T(G)$  en partant d'une forêt de sommets et en les fusionnant lors d'un parcours des sommets dans un ordre décroissant de leur index.

**b.** Étudier la complexité de l'algorithme. Nous tiendrons compte de la structure de données utilisée pour manipuler la forêt.



# Chapitre 65

## (ENS) Forme SSA, arbres de domination \*\*\* (ENS 25, non corrigé — oral - 120 lignes)

\*\*\*

Grammaires, Arbres,  
sources : `ensformessa.tex`

### Forme SSA - placement optimal des fonctions de jointure

*Préambule* : Le but de cette épreuve est d'évaluer la progression des candidates et candidats dans les questions, mais aussi la qualité de leur exposé des solutions, ainsi que l'autonomie dont elles ou ils font preuve pendant l'oral.

Nous considérons le petit langage de programmation composé des commandes suivantes :

**Commandes :**

```
P, Q := return expr int
      | x := expr int où x ∈ X
      | P; Q
      | if expr bool then P else Q endif
      | repeat P until expr bool
```

**Expressions :**

$\text{expr}_{\text{bool}} := \text{expr}_{\text{int}} \circ \text{expr}_{\text{int}}$  avec  $\circ \in \{=, <, \leq\}$

**Expressions d'entiers :**

```
expr int | := n ∈ Z
          | x ∈ X
          | expr int + expr int
          | expr int * expr int
          | expr int / expr int
          | expr int % expr int
```

**Définition.** Étant donné un programme, son graphe de flot de contrôle (CFG - control flow graph) est le graphe orienté  $G = (V, E)$  tel que  $V$  est l'ensemble des points de contrôle du programme (c.-à-d., les commandes) et  $s_1 \rightarrow s_2 \in E$  si  $s_2$  peut suivre  $s_1$  lors de l'exécution du programme.

**Définition.** Un graphe de flot de contrôle  $G = (V, E)$  est sous forme SSA (single-assignment form) si, pour toute variable  $x \in X$ , il existe au plus une affectation de  $x$ , c.-à-d., au plus un sommet  $s \in V$  de la forme  $x := \text{expr}$ .

Nous supposons l'existence d'un sommet Entry relié à l'ensemble des points d'entrées du programme.

Une façon classique de mettre un CFG quelconque sous forme SSA est de dupliquer les variables utilisées plusieurs fois.

Une difficulté peut alors apparaître après les boucles **repeat ... until** ou les conditionnelles **if-then-else**; comment savoir quelle variable utiliser ?

Pour résoudre ce problème, il est habituel d'ajouter au CFG une fonction de jointure, qui prend la forme d'une affectation  $x_i := \varphi(x_{i_1}, \dots, x_{i_n})$  où  $\varphi(\cdot)$  représente une fonction abstraite (c.-à-d., sans sémantique fixée) lorsque

nécessaire. L'arité de la fonction  $\varphi(\cdot)$  dans un sommet  $s$  du CFG dépend du nombre d'affectations de  $x$  dont dépend  $s$ , c.-à-d., du nombre de sommets  $s'$  contenant une affectation de  $x$  et tel que  $s$  est accessible depuis  $s'$  dans le CFG. L'objectif de ce sujet est de trouver un placement optimal des fonctions de jointure afin d'en réduire le nombre.

**Question 1.** *a.* Donner le CFG correspondant au programme décrit en Figure 1.  
*b.* Mettre ce CFG sous forme SSA.

Étant donné un CFG  $G = (V, E)$ , on dit que  $X \in V$  domine  $Y \in V$ , noté  $X \geq Y$ , si, pour tout chemin  $\text{Entry} = s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_n = Y$  dans  $G$ , il existe  $i$  tel que  $s_i = X$ .  
On dit que  $X$  domine strictement  $Y$ , noté  $X > Y$ , si  $X$  domine  $Y$  et  $X \neq Y$ .  
On note  $X \not> Y$  si  $X$  ne domine pas strictement  $Y$ .  
On note  $i \text{ dom}(Y)$  le plus proche dominant strict de  $Y$ .

**Question 2.** *a.* Montrer que la relation de *domination* est une relation réflexive transitive.

```
x := 42;
i := 1;
repeat
  if x%2 == 0 then
    y := x/2
  else y := 3 * x + 1
  endif;
  x := y;
  i := i + 1;
until i < 10;]Q
return x
```

Figure 1 - Programme calculant le 10ième terme de la suite de Syracuse commençant à 42.

*b.* Montrer que si  $X \rightarrow^* Y$  est un chemin simple (c.-à-d., sans cycle) et  $X > Y$  alors  $X > Z$  pour tout  $Z$  appartenant au chemin (et différent de  $X$  ).  
*c.* En déduire que  $\text{idom}(X)$  est bien définie pour tout  $X \neq \text{Entry}$ .

Soit  $G = (V, E)$  un CFG et  $G_d = (V, E_d)$  son arbre de dominance associé, c.-à-d.,  $E_d = \{X \rightarrow Y \mid X, Y \in V \text{ et } X = i \text{ dom}(Y)\}$ .  
On note :

- $DF(X) = \{Y \mid \exists P \in V. (P \rightarrow Y) \in E \wedge X \geq P \wedge X \not> Y\}$
- $DF_{\text{local}}(X) = \{Y \mid (X \rightarrow Y) \in E \wedge X \not> Y\}$
- $DF_{\text{up}}(Z) = \{Y \in DF(Z) \mid \text{idom}(Z) \not> Y\}$

**Question 3.** Montrer que  $DF(X) = DF_{\text{local}}(X) \cup \bigcup_{(X \rightarrow Z) \in E_d} DF_{\text{up}}(Z)$ .

**Question 4.** En déduire un algorithme permettant de calculer  $DF(X)$  ainsi que sa complexité. Nous supposons que nous avons déjà pré-calculé l'arbre de dominance.

Soient  $p : X_0 \rightarrow^+ X_{n_p}$  et  $q : Y_0 \rightarrow^+ Y_{n_q}$  deux chemins non-nuls dans un CFG. On dit que  $p$  et  $q$  convergent vers  $Z$  si :  
(1)  $X_0 \neq Y_0$ , (2)  $X_{n_p} = Y_{n_q} = Z$ , et (3)  $(X_i = Y_j) \Rightarrow (i = n_p \text{ ou } j = n_q)$ .  
Nous notons  $J(S) = \{Z \mid \exists p : X_0 \rightarrow^+ X_{n_p}, q : Y_0 \rightarrow^+ Y_{n_q} \text{ tel que } p \text{ et } q \text{ convergent vers } Z \text{ et } X_0, Y_0 \in S\}$ .

**Question 5.** Définir, à partir de la fonction  $J(\cdot)$ , l'ensemble des sommets où il est nécessaire de placer une fonction  $\varphi(\cdot)$ . Nous supposons que toutes les variables sont initialisées (avec donc une affectation à ces variables) dans  $\text{Entry}$ .

**Question 6.** Nous notons  $DF(S) = \bigcup_{X \in S} DF(X)$  et définissons  $DF^+(S) = \lim_{i \rightarrow +\infty} DF_i(S)$  où  $DF_1(S) = DF(S)$  et  $DF_{i+1}(S) = DF(S \cup DF_i(S))$ .

Montrer que pour toute variable  $x \in \mathcal{X}$ ,  $DF^+(A(x))$  est égal à l'ensemble des positions où nous souhaitons positionner des fonctions  $\varphi(\cdot)$ , avec  $A(x) = \{Z \mid \text{il y a une affectation de } x \text{ dans } Z\}$ .

**Question 7.** En déduire un algorithme qui permet de définir les positions où ajouter les fonctions  $\varphi$ . Donner sa complexité.

# Chapitre 66

## (ENS) Tri Externe Polyphasé \*\*\* (ENS 25, non corrigé — oral - 114 lignes)

\*\*\*

Algorithmique,  
sources : `enstriexterne.tex`

### Tri externe polyphasé

*Préambule* : Le but de cette épreuve est d'évaluer la progression des candidates et candidats dans les questions, mais aussi la qualité de leur exposé des solutions, ainsi que l'autonomie dont elles ou ils font preuve pendant l'oral.

Nous nous intéressons au problème suivant. Imaginons que nous voulions trier le contenu d'un fichier dont la taille dépasse très largement la taille de la mémoire vive (où peuvent s'effectuer rapidement les opérations de tri), mais que nous ayons accès à de la mémoire externe (ici des bandes magnétiques) pour stocker les résultats des opérations de tri (partiels et finaux). Dans ce contexte, les coûts des opérations sur la mémoire vive sont négligeables par rapport à celles de lecture, d'écriture et de déplacements de la tête de lecture sur la mémoire externe.

Nous formalisons le problème de la manière suivante. Supposons que nous voulons trier  $N$  éléments. Supposons que nous avons accès à  $T$  mémoires externes supposées infinies. Chaque mémoire externe a une tête de lecture qui pointe sur une case donnée de la mémoire. Trois opérations peuvent être effectuées sur cette mémoire :

1. écrire un élément de la mémoire vive sur la case pointée par la tête de lecture,
2. lire l'élément pointé par la tête de lecture pour le mettre sur la mémoire vive et l'enlever de la mémoire externe,
3. déplacer la tête de lecture d'une case vers la gauche ou vers la droite.

Lorsque nous disons qu'une mémoire est vide, nous entendons que chacune de ses cases sont vides. Nous représenterons une case vide par le symbole  $\emptyset$ .

Sur la mémoire vive, nous ne pouvons avoir qu'au plus  $K$  éléments simultanément. Sur la liste des éléments sur la mémoire vive, il est possible d'appliquer tout algorithme par exemple pour la trier, y trouver l'élément maximal, comparer des éléments, etc... En plus des éléments de la liste à trier, la mémoire vive peut contenir des variables qui ne comptent pas dans le compte des  $K$  éléments, et qui peuvent être utilisés dans les algorithmes à écrire. Toutes les opérations sur la mémoire vive n'entreront pas dans le calcul de la complexité des algorithmes à écrire.

Le but est d'écrire la liste triée des  $N$  éléments sur l'une des mémoires externes.

**Question 1.** Supposons  $T = 3$ ,  $N = 6$  et  $K = 3$ . Montrez comment passer de la configuration :

1 : vide  
2 : vide  
3 : 6; 2; 4; 5; 3; 1

à la configuration :

1 : vide  
2 : vide  
3 : 1; 2; 3; 4; 5; 6

où dans les deux cas, les têtes de lecture sont toutes sur les premières cases des mémoires externes. Nous ne chercherons pas une solution optimale, mais nous attendrons un calcul précis du nombre d'opérations sur les mémoires externes.

**Question 2.** Supposons que  $T \geq 3$  et que  $N$  est un multiple de  $K$ , disons  $N = Kp$ . Fixons  $q \leq p$ . Supposons qu'initialement nous sommes dans la configuration suivante :

- 1 : vide  
 2 : vide  
 3 :  $L_0$

où  $L_0$  est la liste non triée des  $N$  éléments et où les têtes de lecture sont toutes sur la première case de chaque mémoire. Décrire un algorithme en  $6N$  opérations (écriture, lecture et déplacement) pour obtenir la configuration suivante :

- 1 :  $B_1; B_2; \dots; B_q$   
 2 :  $B_{q+1}; B_{q+2}; \dots; B_{N/K}$   
 3 : vide

où  $B_1, \dots, B_{N/K}$  sont des blocs triés de  $K$  éléments et où les têtes de lecture sont sur les premières cases de chaque mémoire.

**Question 3.** Supposons que  $T = 4$  et que  $N/K$  soit une puissance de 2, disons  $N = K2^n$  avec  $n > 0$ . En partant de la configuration finale de la question précédente avec  $q = N/2K = 2^{n-1}$ , écrivez un algorithme en  $(6nN)$  opérations afin d'obtenir la configuration suivante (ou toute autre configuration à permutation des mémoires externes près) :

- $C_5 6(n+1)N$  1 :  $L$  2 : vide 3 : vide

où  $L$  est la liste triée de tous les éléments et toutes les têtes de lecture pointent vers la première case.

Nous avons donc trouvé un premier algorithme de tri externe en  $O(N \log_2(N/K))$  opérations sur les mémoires externes dans le cas où  $T = 4$ . Cet algorithme peut être adapté pour  $T = 2p$  en  $O(N \log_p(N/K))$ . Nous nous intéressons maintenant au cas  $T = 3$ .

**Question 4.** En partant de la même configuration initiale que la question 3, adaptez le cas  $T = 4$  de l'algorithme au cas  $T = 3$  en effectuant  $9nN$  opérations.

Le problème du cas  $T = 3$  avec une configuration initiale équilibrée est qu'il est nécessaire de copier une partie du contenu d'une mémoire sur une autre mémoire sans réellement progresser dans la construction de la liste triée.

**Question 5.** En partant d'une autre configuration initiale obtenue en question 2 bien choisie, adaptez l'algorithme de la question 3 de façon à ne faire *aucune* de ces copies de mémoire. Vous pourrez supposer que  $N$  est de la forme  $N = KF_n$  pour une suite  $F_n$  bien choisie, dans quel cas vous pourriez montrer un algorithme en

$$3N + 6K \sum_{i=1}^{n-1} F_{n-i} F_i$$

opérations.

# Chapitre 67

## (MT) Dédution Naturelle \* (MT 0, ex 1, corrigé (R) — oral - 200 lignes)

\*

Dédution naturelle,  
sources : mtlog1.tex

cf annexe pour un rappel des règles de la déduction naturelle

1. Prouver le séquent  $A \wedge B \vdash B \wedge A$
2. Prouver le séquent  $A \wedge (B \vee C) \vdash (A \wedge B) \vee (A \wedge C)$

Soit  $\Gamma \vdash C$  un séquent prouvable à l'aide d'un arbre de preuve  $\Pi$ .

3. Montrer qu'il existe un arbre de preuve de  $\Gamma \vdash C$  tel que cet arbre ne possède pas la succession de la règle élimination de la conjonction puis introduction de la conjonction.
4. Que peut-on dire pour les successions de règles de disjonction ?
5. Prouver le séquent  $\vdash A \vee \neg A$ .

### Annexe 1

#### Rappel des règles de la déduction naturelle

Les arbres de preuves doivent être effectués à partir de l'ensemble de règles fourni ci-dessous.

$$\begin{array}{c}
 \frac{}{\Gamma, A \vdash A} \text{ax} \quad \frac{\Gamma \vdash B}{\Gamma, A \vdash B} \text{aff} \quad \frac{\Gamma, \neg A \vdash \perp}{\Gamma \vdash A} \text{RAA} \\
 \frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \rightarrow_i \quad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \rightarrow_e \\
 \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge_i \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge_e^g \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge_e^d \\
 \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee_i^g \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee_i^d \quad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \vee_e \\
 \frac{\Gamma, A \vdash \perp}{\Gamma \vdash \neg A} \neg_i \quad \frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash \perp} \neg_e
 \end{array}$$



## Chapitre 68

# (MT) Relations entre les nombres de sommets et d'arêtes de graphes \* (MT 0, ex 1, corrigé (R) — oral - 112 lignes)

\*

Graphes,  
sources : `mtgraph1.tex`

Soit  $G$  un graphe non-orienté à  $n \geq 1$  sommets et  $p$  arêtes.

1. Montrer que si  $G$  est connexe alors  $p \geq n - 1$ .
2. Montrer que si  $G$  est acyclique alors il possède un sommet de degré au plus 1.
3. Montrer que si  $G$  est acyclique alors  $p \leq n - 1$ .
4. Montrer que les trois propriétés suivantes sont équivalentes :
  - (i)  $G$  est connexe et acyclique.
  - (ii)  $G$  est connexe et  $p = n - 1$ .
  - (iii)  $G$  est acyclique et  $p = n - 1$ .



## Chapitre 69

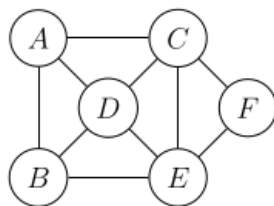
# (MT) couplages \* (MT 0, ex 1, corrigé (R)) — oral - 64 lignes)

\*

Graphes,

sources : mtgraph3.tex

1. Rappeler la définition de couplage dans un graphe non orienté. Quand peut-on qualifier un couplage de maximal ? maximum ? parfait ?
2. Donner un couplage maximum dans le graphe suivant. Donner un couplage maximal qui n'est pas maximum.



3. Qu'est ce qu'un chemin augmentant alternant pour un graphe  $G = (S, A)$  et un couplage  $C \subset A$  sur ce graphe ?
4. Soit  $G = (S, A)$  un graphe. Montrer qu'un couplage est maximum si et seulement s'il n'existe pas de chemin augmentant alternant pour ce couplage dans le graphe  $G$ .



## Chapitre 70

(MT) logique \* (MT 0, ex 1, corrigé (R) —  
oral - 100 lignes)

\*

Logique propositionnelle,  
sources : mtlog2.tex

Soit  $F = ((a \wedge b) \vee \neg c) \wedge (a \vee \neg b)$

1.  $F$  est-elle satisfiable?  $\neg F$  est-elle satisfiable?
2. Mettre  $F$  en forme normale conjonctive.
3. Mettre  $F$  en forme normale disjonctive.
4. Rappeler le théorème de Cook-Levin.



# Chapitre 71

## (MT) bdd pour livraison \* (MT 0, ex 1, corrigé — oral - 78 lignes)

\*

Sql,  
sources : mtsql1.tex

Une entreprise de livraison de nourriture dispose d'une base de donnée pour représenter ses clients et les commandes passées. On présente le schéma relationnel correspondant.

```
Clients(id : int, id_adresse : int, nom : text, prenom : text)
Adresses(id : int, ville : text, nom_rue : text, numero : int)
Commandes(id_client : int, plat : text, prix : int, date : date)
```

L'entreprise souhaite ajouter une fonctionnalité pour vérifier qu'un client n'est pas allergique à un plat qu'il commande. Pour cela, il est nécessaire de pouvoir enregistrer les allergènes présents dans chaque plat ainsi que les allergies de chaque client.

1. Proposer des modifications à notre schéma de relation pour pouvoir ajouter ces informations. Justifier pourquoi votre choix convient.

Dans la suite, on se base dans le modèle tel que vous l'avez modifié.

2. Écrire une requête pour trouver les noms et prénoms des clients étant allergiques à au moins un ingrédient d'une pizza.
3. Écrire une requête pour trouver les trois villes où le plus d'argent est dépensé, ainsi que cette somme totale.
4. Écrire une requête pour effectuer la moyenne d'argent dépensé en fonction des prénoms des clients.



## Chapitre 72

# (MT) Arbres Binaires de Recherche \* (MT 0, ex 2, corrigé (R) — oral - 167 lignes)

\*

Arbres, Complexité, Programmation dynamique,  
sources : `mtarbres1.tex`

On considère un ensemble de clés  $\mathcal{K} \subseteq \mathbb{N}$  fini de taille  $m$ . On note  $\mathcal{A}_{\mathcal{K}}$  l'ensemble des arbres binaires dont les noeuds sont étiquetés par des éléments *distincts* de  $\mathcal{K}$ . Pour  $A \in \mathcal{A}_{\mathcal{K}}$ , on notera  $\kappa(A)$  l'ensemble des étiquettes des noeuds de  $A$ . Par convention, on dira que la racine de  $A$  est de profondeur 0.

1. Rappeler la définition inductive d'un *arbre binaire de recherche* (ABR).
2. Rappeler la complexité dans le pire des cas de la recherche dans un ABR ayant  $n$  noeuds.

Soit  $A \in \mathcal{A}_{\mathcal{K}}$  tel que  $\kappa(A) = \mathcal{K}$ . Soit  $\mathbb{P}$  une loi de probabilité quelconque, sur les clés de  $\mathcal{K}$ . On définit  $H_A$  la variable aléatoire qui, à une clé de  $\mathcal{K}$ , associe sa profondeur dans  $A$ . Ainsi  $H_A(k)$  est la profondeur de  $k$  pour une clé  $k$ .

3. Soit un ABR fixé  $A \in \mathcal{A}_{\mathcal{K}}$ . Donner l'expression de la complexité en moyenne de la recherche d'une clé dans  $A$  en fonction de  $H_A$ .
4. On suppose ici que  $\mathbb{P}$  est la loi uniforme sur les clés de  $\mathcal{K}$ . Quelles sont les particularités des ABR  $A \in \mathcal{A}_{\mathcal{K}}$  pour lesquels  $\mathbb{E}(H_A)$  est minimale. Montrer que  $\mathbb{E}(H_A) = O(\log(m))$ .

On dit que  $A \in \mathcal{A}_{\mathcal{K}}$  est optimal pour  $\mathcal{K}$  et  $\mathbb{P}$  si et seulement si  $\kappa(A) = \mathcal{K}$  et il minimise  $\mathbb{E}(H_A)$ .

On cherche un algorithme calculant un ABR optimal étant donné un ensemble de clés fini et une loi de probabilité sur ces clés.

5. De quel type de problème algorithmique s'agit-il?
6. Pour cette question, on suppose  $\mathcal{K} = \{1, 2, 3\}$ ,  $\mathbb{P}(1) = \frac{2}{3}$ ,  $\mathbb{P}(2) = \mathbb{P}(3) = \frac{1}{6}$ . Dessiner un ABR optimal pour  $\mathcal{K}$  et  $\mathbb{P}$
7. Soit  $A \in \mathcal{A}_{\mathcal{K}}$  un ABR optimal non vide.  $A$  possède donc un sous-arbre gauche  $A_g$ . On se place dans le cas où  $A_g$  est non vide. On note  $S_g = \sum_{k \in \kappa(A_g)} \mathbb{P}(k)$  et  $\mathbb{P}_g = \frac{1}{S_g} \mathbb{P}$ . Montrer que  $A_g$  est optimal pour  $\kappa(A_g)$  et  $\mathbb{P}_g$ . Montrer le résultat analogue pour le sous-arbre droit.
8. Proposer un algorithme de programmation dynamique pour trouver un ABR optimal.
9. Quel est sa complexité?



## Chapitre 73

# (MT) entrelacements de mots \* (MT 0, ex 2, corrigé (R) — oral - 105 lignes)

\*

Langages, Automates, Programmation dynamique,  
sources : `mtlang.tex`

Soit  $\Sigma$  un alphabet.

Pour deux mots  $u, v$  dans  $\Sigma^*$ , on appelle entrelacement de  $u$  et  $v$ , un mot  $w$  qui utilise exactement les lettres de  $u$  et de  $v$  dans leur ordre dans chaque mot.

Par exemple  $babaa$  est un entrelacement de  $bb$  et  $aaa$  ou encore  $abcabc$  est un entrelacement de  $aba$  et  $cbc$ . Mais  $bacb$  n'est pas un entrelacement de  $ab$  et  $cb$  car l'ordre n'est pas respecté.

On peut voir le lien avec les entrelacements des tâches de plusieurs fils d'exécution.

On note  $\mathcal{E}(u, v)$  l'ensemble des entrelacements de  $u$  et  $v$  pour deux mots  $u, v$ .

1. Donner les entrelacements de  $ab$  et  $cb$ .
2. Soit  $u, v$  deux mots.
  - (a) Majorer grossièrement le cardinal de  $\mathcal{E}(u, v)$ .
  - (b) Montrer que  $\mathcal{E}(u, v)$  est un langage régulier.
3. Soit  $L_1, L_2$  deux langages réguliers. Montrer que  $\mathcal{E}(L_1, L_2) = \bigcup_{u \in L_1, v \in L_2} \mathcal{E}(u, v)$  est un langage régulier.
4. Soit  $u, v, w$  trois mots dans  $\Sigma^*$ , proposer un algorithme de programmation dynamique permettant de savoir si  $w \in \mathcal{E}(u, v)$ . Préciser la complexité.



## Chapitre 74

# (MT) montée et descente d'un bus \* (MT 0, ex 2, corrigé (R) — oral - 104 lignes)

\*

Concurrence,

sources : `mtprocess.tex`

On s'intéresse à un bus touristique pouvant contenir  $C$  passagers.

On suppose que l'on a  $n > C$  passagers qui attendent leur tour, puis se remettent en attente à l'arrêt du bus dès qu'ils ont fini pour le revoir.

Le bus ne démarre que lorsqu'il est plein.

Pour formaliser ce problème, on utilise des fonctions fictives :

- `board` et `unboard` permettent au passager de monter et de descendre ;
- le bus doit appeler les fonctions `load` lorsqu'il se remplit, `run` lorsqu'il démarre son tour et `unload` lorsqu'il se vide.

Attention, les passagers ne peuvent pas descendre avant que le bus ait ouvert ses portes pour une fin de tour avec `unload` et ne peuvent pas monter avant que le bus ait ouvert ses portes pour un nouveau tour avec `load`.

1. Écrire les fonctions en pseudo-code correspondant au bus et aux passagers **sans** prendre en compte les problèmes de synchronisation dans un premier temps.
2. Proposer une solution en pseudo-code utilisant deux compteurs protégés par des mutex et quatre sémaphores.
3. Votre solution peut-elle être utilisée dans le cas où l'on a plusieurs bus ? Justifier.
4. Proposer une nouvelle solution pour le pseudo-code du bus dans le cas où il y a  $m$  bus numérotés en respectant les règles suivantes :
  - un seul bus peut ouvrir ses portes aux passagers à la fois ;
  - plusieurs bus peuvent faire un tour en même temps ;
  - les bus ne peuvent pas se doubler donc ils se chargent et se déchargent toujours dans le même ordre
  - un bus doit avoir fini de décharger avant qu'un autre bus vienne décharger.

La solution doit utiliser deux tableaux de sémaphores en plus des sémaphores utilisés dans la solution précédente, permettant de gérer la coordination entre les bus.



## Chapitre 75

# (MT) subset-sum \* (MT 0, ex 2, corrigé (R) — oral - 143 lignes)

\*

Complexité, Algorithmes d'approximation, Algorithmes gloutons,  
sources : mtsubsum.tex

On considère le problème suivant :

### SUBSET-SUM :

**Entrée :** Un tableau  $T = [t_1, \dots, t_n]$  de  $n$  entiers positifs et un entier  $c$ .

**Sortie :** La valeur maximum de  $\sum_{i \in I} t_i$  où  $I \subseteq \{1, \dots, n\}$  et  $\sum_{i \in I} t_i \leq c$ .

1. Décrire un algorithme naïf qui résout ce problème et préciser sa complexité.
2. On note  $s_{i,j}$  la plus grande somme inférieure à  $j$  que l'on peut obtenir avec des éléments  $t_1, \dots, t_i$ .  
Donner une équation de récurrence pour  $s_{i,j}$  et en déduire un algorithme pour SUBSET-SUM. Comparer sa complexité avec l'algorithme précédent.

On considère l'algorithme glouton suivant :

```
Trier les éléments de  $T$  par ordre décroissant.  
 $S \leftarrow 0$   
Pour  $i$  de 1 à  $n$  :  
  | Si  $S + t_i \leq c$  :  
  | |  $S \leftarrow S + t_i$   
Renvoyer  $S$ 
```

3. Donner la complexité de cet algorithme.
4. Montrer que l'algorithme glouton donne une  $\frac{1}{2}$ -approximation de la solution optimale.
5. Soit  $\alpha \in \left] \frac{1}{2}, 1 \right]$ . Donner une instance de SUBSET-SUM telle que la somme  $S$  renvoyée par l'algorithme glouton vérifie  $S \leq \alpha S^*$  où  $S^*$  est la solution optimale.



# Chapitre 76

## (MT) graphes \* (MT 0, ex 2, corrigé — oral - 121 lignes)

\*

Graphes,

sources : `mtgraph2.tex`

Dans cet exercice, on ne considère que des graphes orientés. Soit  $G = (S, A)$  un graphe, on appelle clôture transitive d'un graphe, qu'on note  $G^* = (S, A^*)$ , le graphe ayant les mêmes sommets que  $S$  et tel que  $(x, y)$  soit une arête de  $G^*$  si et seulement si il existe un chemin de  $x$  à  $y$  dans  $G$ . On note  $|S| = n$ .

1. Justifier que la relation  $\mathcal{R}$  définie par  $x\mathcal{R}y$  si et seulement si  $(x, y)$  appartient à  $A^*$  est transitive.
2. On suppose que  $A$  est donné sous la forme d'une matrice d'adjacence. Donner un algorithme pour calculer  $A^*$  en  $O(n^3)$  opérations.

Une réduction transitive d'un graphe  $G = (S, A)$  est un sous-graphe  $G_R = (S, A_R)$  avec un nombre minimum d'arêtes tel que  $G^* = G_R^*$ .

3. Montrer que dans le cas général, une réduction transitive n'est pas unique.

Dans toute la suite, on ne considèrera plus que des graphes acyclique.

Considérons pour tout  $x, y$  dans  $S$   $A_{x,y}$  définie par :

$$A_{x,y} = \{e \in A \mid e \text{ appartient à un chemin de longueur maximal entre } x \text{ et } y\}$$

De plus on considère

$$A' = \bigcup_{x,y \in S} A_{x,y}$$

4. Montrer l'égalité suivante :

$$A' = \{(x, y) \in A \mid \text{la longueur du plus long chemin entre } x \text{ et } y \text{ est } 1\}$$

5. En déduire que  $G' = (S, A')$  est l'unique réduction transitive de  $G$ .
6. Donner un algorithme pour calculer l'unique réduction transitive d'un graphe acyclique.



## Chapitre 77

# (MT) Graphes bipartis \* (MT 24, ex 1, corrigé (R) — oral - 92 lignes)

\*

Graphes, Logique propositionnelle, Parcours de graphes, Complexité,  
sources : `mtgraph5.tex`

1. Donner le nombre minimal et maximal d'arêtes dans un graphe biparti.
2. Montrer qu'un graphe est biparti si et seulement si il ne contient aucun cycle de longueur impaire.
3. Donner un algorithme qui permet de déterminer si un graphe est biparti. Donner sa complexité.
4. Donner une formule de logique propositionnelle telle qu'une valuation de cette formule permette de déterminer si un graphe est biparti. En déduire un autre algorithme ainsi que sa complexité permettant de déterminer si un graphe est biparti.



## Chapitre 78

# (MT) Tas binaires \* (MT 24, ex 1, corrigé (R) — oral - 129 lignes)

\*

Arbres, Tri, Structures de données, Complexité,  
sources : `mttas.tex`

1. Donner la définition d'un tas binaire
2. Donner 2 représentation d'un tas binaire
3. Effectuer un tri par tas sur l'exemple suivant :

5	4	0	2	1	3
---	---	---	---	---	---

4. Donner la complexité du tri par tas et le prouver.



# Chapitre 79

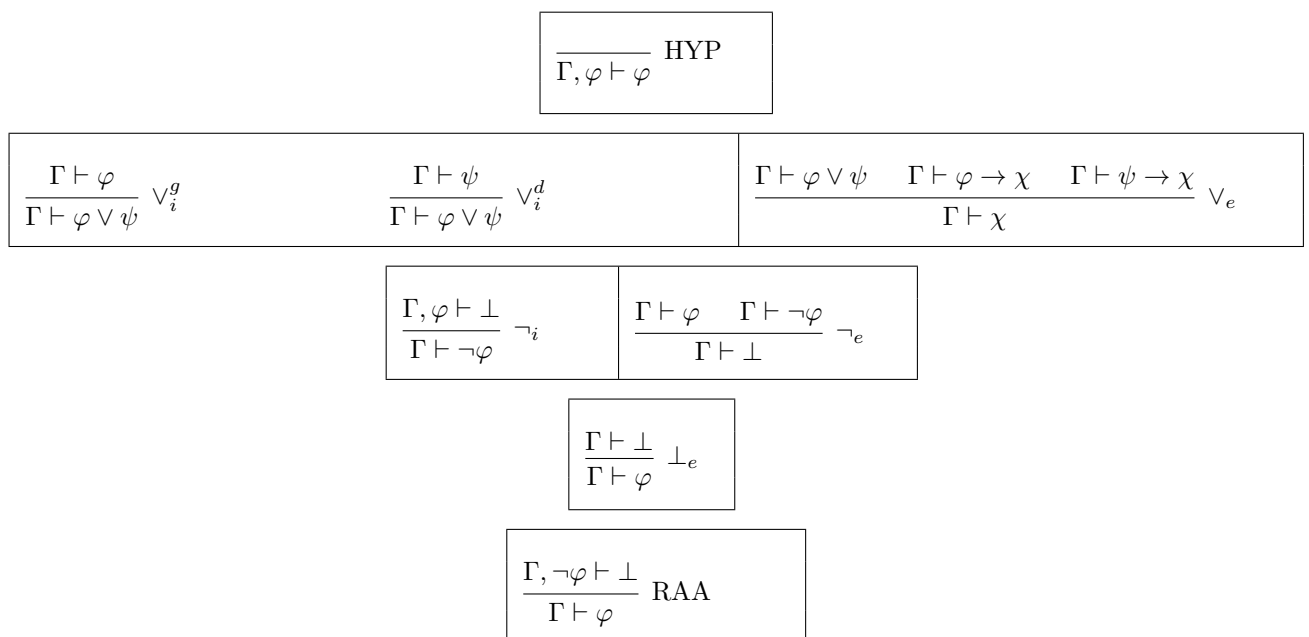
## (MT) déduction naturelle \* (MT 24, ex 1, corrigé (R) — oral - 141 lignes)

\*

Déduction naturelle,  
sources : mtlog3.tex

1. Prouver le séquent  $\neg(A \vee \neg A) \vdash \neg A$ .
2. Prouver le séquent  $\vdash A \vee \neg A$ .

*Annexe* : règles de déduction utilisables.





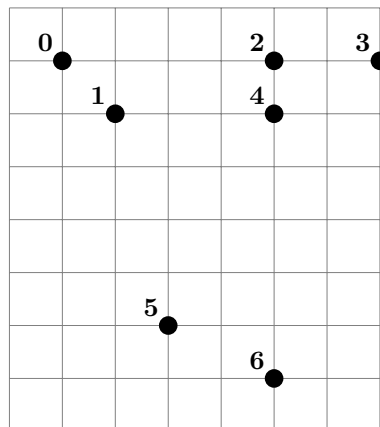
## Chapitre 80

# (MT) Classification hiérarchique ascendante \* (MT 24, ex 1, corrigé, le graphique n'est pas l'original (R) — oral - 44 lignes)

\*

Ia, Classification hiérarchique ascendante,  
sources : `mtia.tex`

1. Rappeler l'intérêt et le fonctionnement de l'algorithme de classification hiérarchique ascendante.
2. L'effectuer sur un ensemble de 7 points répartis sur une grille  $7 \times 8$  (donné en sujet) en utilisant la distance euclidienne pour deux points et  $d(C_1, C_2) = \min_{x \in C_1, y \in C_2} d(x, y)$  pour la distance entre deux classes.





# Chapitre 81

## (MT) Tas binaires \* (MT 24, ex 1, corrigé — oral - 137 lignes)

\*

Arbres, Tri, Structures de données, Complexité, Pseudo-code,  
sources : `mttas2.tex`

1. Donner la définition d'un tas binaire max.  
Citer deux implémentations possibles d'un tel tas.
2. Construire un tas binaire max avec ces valeurs :  
20, 14, 15, 16, 8, 7, 9, 30, 18
3. Comment extraire la racine de ce tas ?
4. Donner la complexité temporelle des algorithmes pour ajouter un élément et extraire la racine d'un tas.
5. Un arbre binaire décroissant (ABD) est un arbre binaire dont chaque chemin allant de la racine à une feuille est une suite finie et décroissante.
  - (a) Donner une définition inductive d'un ABD.
  - (b) Écrire en pseudo-code un algorithme permettant de déterminer si un arbre binaire est décroissant ou non.
  - (c) Quel lien y a-t-il entre ABD et tas binaire max ?
6. On note  $a_n$  le nombre d'ABD à  $n$  sommets étiquetés de 1 à  $n$ .  
Calculer  $a_3$ , puis montrer que pour tout  $n$ ,  $a_n = n!$ .



## Chapitre 82

# (MT) Automates de Büchi \*\*\* (MT 24, ex 2, (modifié) corrigé (R) — oral - 117 lignes)

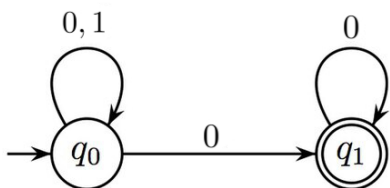
\*\*\*

Langages, Automates,  
sources : `mtbuchi.tex`

Un automate de Büchi est un automate  $(Q, \Sigma, T, I, F)$  permettant de reconnaître des mots infinis, avec  $I \subset Q$ ,  $F \subset Q$  et  $T \subset Q \times \Sigma \times Q$ .

Soit  $X$  un ensemble de mots. On note  $X^\omega$  l'ensemble des mots infinis de  $X$  :  $x$  est dans  $X^\omega$  s'il existe une suite  $(x_i)_{i \in \mathbb{N}}$  telle que  $x = x_0 x_1 x_2 \dots$ . Un mot infini  $x$  de  $\Sigma^\omega$  est reconnu par un automate de Büchi lorsqu'il existe un chemin infini dans cet automate étiqueté par  $x$ , commençant par un état initial et passant une infinité de fois par un état final.

1. Déterminer le langage reconnu par l'automate de Büchi ci-dessous.



2. Proposer un automate de Büchi reconnaissant  $(01)^\omega$ .

Soient  $A$  et  $B$  deux automates de Büchi.

3. Montrer que  $L(A) \cup L(B)$  est reconnu par un automate de Büchi.
4. Soit  $K$  un langage régulier. Montrer que  $K.L(A) = \{u.v, u \in K, v \in L(A)\}$  est reconnu par un automate de Büchi.
5. Si  $K$  est un langage régulier autre que  $\emptyset$  ou  $\{\epsilon\}$ , montrer que  $K^\omega$  est reconnu par un automate de Büchi.
6. Montrer que  $L(A) \cap L(B)$  est reconnu par un automate de Büchi.



## Chapitre 83

# (MT) Décidabilité de programmes engendrés par une grammaire \* (MT 24, ex 2, corrigé (R) — oral - 140 lignes)

\*

Grammaires, Décidabilité, C, Logique propositionnelle,  
sources : mthaltbool.tex

Les langages et problèmes dont il est question dans l'énoncé sont définis en annexe.

1. `HALT_BOOL_BOOL_C` est-il décidable ?
2. Soit  $\eta \in ASCII^*$  un code sur `bool_bool_c` à partir d'une dérivation de symbole initial **S**. Montrer qu'il existe une formule  $\varphi_\eta$  tel que pour  $\nu_i$ , une valuation,  $\varphi_\eta(\nu_i)$  correspond à la valeur de sortie de  $\eta$  en identifiant  $x_i$  et  $\nu_i$ .
3. Donner la formule associée au code suivant :

```
if(x[0]){
    if(x[0] == x[1]) {return false;}
    else {return true;}
}
else {return x[1] == x[2];}
```

4. `HALT_BOOL_C` est-il décidable ?
5. `HALT_ALL_BOOL_C` est-il décidable ?

### Annexe :

On note *ASCII* l'alphabet constitué des caractères ACSII – contenant notamment les chiffres, lettres et divers symboles (ponctuation, parenthèses, opérateurs...).

On définit la grammaire hors-contexte  $\mathcal{C}$  sur l'alphabet *ASCII* avec les règles de dérivation suivantes :

$$\begin{aligned} \mathbf{S} &\rightarrow \text{bool f (bool x[])} \{\mathbf{I}\} \\ \mathbf{E} &\rightarrow \text{true} \mid \text{false} \mid \mathbf{V} \mid (\mathbf{E} == \mathbf{E}) \\ \mathbf{V} &\rightarrow \text{x[i]} \text{ (pour chaque } i \in \mathbb{N}) \\ \mathbf{I} &\rightarrow \text{return } \mathbf{E}; \mid \mathbf{B} \mid \mathbf{C} \\ \mathbf{B} &\rightarrow \text{while}(\mathbf{E}) \{\mathbf{I}\} \\ \mathbf{C} &\rightarrow \text{if}(\mathbf{E})\{\mathbf{I}\} \text{ else } \{\mathbf{I}\} \end{aligned}$$

$S$  est le symbole initial.

Cette grammaire engendre des chaînes de caractères correspondant à des programmes en langage  $\mathcal{C}$ .

Le langage *bool\_c* est le langage à partir de la grammaire  $\mathcal{C}$ .

Le langage *bool\_bool\_c* est le langage des fonctions obtenu à partir de la grammaire  $\mathcal{C}$  privée de la règle de membre de gauche **B**.

On définit de plus les problèmes suivants :

**HALT\_BOOL\_C :**

**Entrée**  $\kappa \in \text{bool}_c$

**Sortie** le programme  $\kappa$  termine-t-il sur une entrée  $x$  ?

**HALT\_ALL\_BOOL\_C :**

**Entrée**  $\kappa \in \text{bool}_c$

**Sortie** le programme  $\kappa$  termine-t-il sur toute entrée ?

**HALT\_BOOL\_BOOL\_C :**

**Entrée**  $\kappa \in \text{bool\_bool\_c}$

**Sortie** le programme  $\kappa$  termine-t-il sur toute entrée ?

## Chapitre 84

# (MT) Automates, langages à saut \* (MT 24, ex 2, corrigé, complété au jugé pour les dernières questions (R) — oral - 93 lignes)

\*

Langages, Automates, Lemme de l'étoile,  
sources : `mtlang2.tex`

On se place sur  $\Sigma = \{a, b\}$ .

Soit  $A$  un automate à un seul état initial  $q_{\text{init}}$ . On note  $Q$  l'ensemble des sommets,  $R = Q \times \Sigma \times Q$  les transitions,  $F$  les états finaux.

Soient  $u, v, u', v'$  dans  $\Sigma^*$ ,  $a \in \Sigma \cup \{\varepsilon\}$  et  $q_1, q_2 \in Q^2$ . On pose la relation  $(u, q_1, av) \curvearrowright (u', q_2, v')$  valable uniquement si  $(q_1, a, q_2) \in R \wedge uv = u'v'$ .

On a  $(u, q_1, av) \curvearrowright^* (u', q_2, v')$  si un nombre fini de  $\curvearrowright$  permettent de passer de  $(u, q_1, av)$  à  $(u', q_2, v')$

On définit le langage à saut de l'automate comme suit :

$$L_{\ddagger}(A) = \{uv \in \Sigma^* \mid \exists f \in F, (u, q_{\text{init}}, v) \curvearrowright^* (\varepsilon, f, \varepsilon)\}$$

On note  $L_{ab} = \{u \in \Sigma^* \mid u \text{ contienne autant de } a \text{ que de } b\}$

1. Montrer que  $L_{ab}$  n'est pas régulier.
2. Montrer que  $L_{ab}$  est le langage à saut d'un certain automate.
3. On note  $\text{Perm}(L) = \{u \in \Sigma^* \mid \exists v \in L, u \text{ soit une permutation de } v\}$ .  
Si  $L$  est régulier,  $\text{Perm}(L)$  l'est-il ?
4. Montrer que si  $L$  est fini, alors  $\text{Perm}(L)$  est régulier.
5. Donner un exemple de langage  $L$  infini tel que  $\text{Perm}(L)$  soit régulier.
6. Étant donné un automate  $A$ , montrer que  $\text{Perm}(L(A)) = L_{\ddagger}(A)$ .
7. On se place sur l'alphabet  $\Sigma = \{a, b\}$ . Soit  $L$  un langage régulier tel que  $L \subset a^*b^*$ . Montrer que  $\text{Perm}(L)$  est régulier.



## Chapitre 85

# (MT) plus courts chemins dans des graphes \* (MT 24, ex 2, corrigé, retour d'oral (R) — oral - 129 lignes)

\*

Graphes, Complexité, Programmation dynamique, Pseudo-code,  
sources : `mtgraph4.tex`

On considère  $G = (S, A)$  un graphe orienté pondéré par la fonction  $w : A \rightarrow \mathbb{R}$  pouvant être à poids négatifs. On note  $n = |S|$  et  $p = |A|$ . On étudiera ici un algorithme de calcul des plus courts chemins nommé *algorithme de Johnson*.

1. Donner la complexité de l'algorithme de Dijkstra.
2. Soit  $h : S \rightarrow \mathbb{R}$  et  $w_h(u, v) = w(u, v) + h(u) - h(v)$ . Montrer que les plus courts chemins pour  $w$  sont les mêmes que ceux pour  $w_h$ , et qu'il existe un cycle de poids strictement négatif pour  $w$  si et seulement si il en existe aussi un pour  $w_h$ .

On considère maintenant un graphe  $G$  ne contenant aucun cycle de poids strictement négatif, et tel qu'il existe un sommet  $r$  depuis lequel tous les autres sommets sont accessibles à partir de  $r$ .

3. Trouver  $h$  tel que  $w_h$  soit à valeurs positives ou nulles.
4. On suppose que l'on peut calculer toutes les distances entre un sommet fixé et les autres sommets de  $G$  en  $O(np)$ . Proposer un algorithme permettant de calculer tous les plus courts chemins entre tous les sommets en  $O(np \log(n) + n^2 \log(n))$ . Comparer avec l'algorithme de Floyd-Warshall.
5. Afin de calculer les distances entre un sommet  $s$  fixé et tous les autres sommets en temps  $O(np)$ , on propose une approche par programmation dynamique. On note  $d(t, k)$  la longueur d'un plus court chemin de  $s$  à  $t$  empruntant au plus  $k$  arcs.
  - (a) Établir une relation de récurrence vérifiée par les  $d(t, k)$ .
  - (b) En déduire le pseudocode d'un algorithme de complexité spatiale  $O(np)$  répondant au problème posé.
6. Généraliser la question 3 et les suivantes au cas où on n'a plus nécessairement l'existence d'un sommet  $r$  à partir duquel tous les sommets sont accessibles.



## Chapitre 86

# (MT) Automates et décidabilité \*\* (MT 25, ex 1, retour élève complété, corrigé — oral - 87 lignes)

\*\*

Langages, Automates, Décidabilité,  
sources : mtautom.tex

1. Soit  $\mathcal{A}$  un AFD à  $n$  états sur un alphabet  $\Sigma$ . Montrer que  $\mathcal{L}(\mathcal{A})$  est infini si et seulement si

$$\exists w \in \Sigma^*, |w| \in \llbracket n, 2n - 1 \rrbracket \text{ et } w \in \mathcal{L}(\mathcal{A}).$$

2. Montrer que les problèmes suivants sont décidables, et indiquer s'ils sont dans la classe **P** ou dans la classe **NP** (ou dans aucune des deux).
  - (a) **Entrée** un AFD  $\mathcal{A}$   
**Sortie** a-t-on  $\mathcal{L}(\mathcal{A}) = \emptyset$  ?
  - (b) **Entrée** un AFD  $\mathcal{A}$  et un mot  $u$   
**Sortie** a-t-on  $u \in \mathcal{L}(\mathcal{A})$  ?
  - (c) **Entrée** un AFD  $\mathcal{A}$   
**Sortie** le langage  $\mathcal{L}(\mathcal{A})$  est-il infini ?
  - (d) **Entrée** une grammaire  $G$  non contextuelle avec uniquement des règles de la forme  $X \rightarrow \alpha$ , où  $|\alpha| > 1$ , et un mot  $u$ .  
**Sortie**  $u$  est-il dans le langage engendré par  $G$ .
3. Citer un problème indécidable.



## Chapitre 87

# (MT) Récompenses pour un site web \*

## (MT 25, ex 1, retour élève remanié, corrigé (à préciser/compléter) — oral - 68 lignes)

\*

Concurrence,  
sources : mtconcl.tex

On veut donner une récompense aux 144 premières personnes se connectant à un site web.  
On propose le code suivant :

```
Nb_ressources = 144
```

Processus() :

```
Si Nb_ressources > 0 :  
    verification ← Verification des donnees de connexion  
    Si verification ok :  
        Nb_ressources --
```

On suppose que l'opération `Nb_ressources--` est atomique, ainsi que les tests.

1. Indiquer un problème posé par cet algorithme.
2. Rappeler le principe de fonctionnement des mutex et des sémaphores.
3. On considère qu'on a des ID entières strictement positives deux à deux distinctes pour les personnes pouvant se connecter.  
Proposer un algorithme répondant au problème dans ce contexte, sans mutex ni sémaphore.
4. Proposer un algorithme avec un mutex ou un sémaphore.



## Chapitre 88

# (MT) Algorithmique des textes \* (MT 25, ex 1, retour élève remanié, corrigé — oral - 72 lignes)

\*

Algorithmique des textes,  
sources : `mttxt2.tex`

On considère la chaîne suivante sur l'alphabet  $\{A, C, D, L, E, S\}$  :

*LESCASCADESDELESCALADE*

1. Lui appliquer l'algorithme de Huffman.
2. Comparer le nombre de bits du texte initial et celui du texte compressé.
3. Lui appliquer l'algorithme LZW en utilisant le dictionnaire suivant :

A	C	D	L	E	S
1	2	3	4	5	6



## Chapitre 89

# (MT) Langages \*\* à \*\*\* (MT 25, ex 1, retour élève remanié, corrigé — oral - 80 lignes)

\*\* à \*\*\*

Langages, Automates,  
sources : `mtlang4.tex`

On note  $\Sigma_n = \{0, \dots, n-1\}$ .

Étant donné  $m = m_0 \dots m_k \in \Sigma_n^{k+1}$ , on note  $V(m) = \sum_{l=0}^k m_l n^{k-l}$  ( $m$  est donc la représentation en base  $n$  de  $V(m)$ ).

1. Montrer que le langage des mots de  $\Sigma_2$  tels que  $V(m)$  soit pair est rationnel.
2. Montrer que le langage des mots de  $\Sigma_4$  multiples de 3 est rationnel.
3. Le langage des mots de  $\Sigma_5$  multiples de 3 mais pas de 2 est-il rationnel ?



## Chapitre 90

# (MT) Piles et files \* (MT 25, ex 1, retour élève retravaillé, corrigé — oral - 67 lignes)

\*

Structures de données, Complexité,

sources : `mtstruct1.tex`

1. Définir les structures de piles et de files, et préciser quelles sont les opérations élémentaires sur ces structures.
2. Comment implémenter une file avec deux piles ?
3. Discuter la complexité de votre solution.
4. Prouver la correction de votre implémentation.



## Chapitre 91

# (MT) Points fixes dans des tableaux \* (MT 25, ex 1, retour élève retravaillé, corrigé — oral - 79 lignes)

\*

Algorithmique, Complexité, Preuves, Pseudo-code,  
sources : `mttab1.tex`

On s'intéresse à la recherche de points fixes (i.e. d'indices  $i$  tels que  $T[i] = i$ ) dans un tableau  $T$  d'entiers relatifs triés par ordre strictement croissant.

On notera  $n$  la longueur du tableau.

1. Donner un algorithme pour trouver un tel point fixe s'il existe en  $o(n)$ .
2. Prouver la terminaison et la correction de votre algorithme.
3. Prouver rigoureusement sa complexité.
4. Proposer une solution permettant de calculer tous les points fixes du tableau. Quelle est sa complexité ?



## Chapitre 92

# (MT) Langages des préfixes \* (MT 25, ex 1, retour élève à modifier, corrigé — oral - 59 lignes)

\*

Langages, Grammaires, Lemme de l'étoile,  
sources : `mtlang6.tex`

Soit  $\Sigma$  un alphabet fini.

Pour tout langage  $L \subset \Sigma^*$ , on note  $\text{Pref}(L)$  l'ensemble des préfixes des mots de  $L$  :  $\text{Pref}(L) = \{u \in \Sigma^* \mid \exists v \in \Sigma^*, uv \in L\}$ .

1. Montrer que si  $L$  est régulier,  $\text{Pref}(L)$  l'est également.
2. La réciproque est-elle vraie ?
3. Montrer que si  $L$  est hors-contexte,  $\text{Pref}(L)$  l'est également.

*Indication* : pour chaque non terminal  $A$  d'une grammaire  $\mathcal{G}$  engendrant  $L$ , on introduira un nouveau non terminal  $A'$  engendrant tous les préfixes des mots dérivant de  $A$  dans  $\mathcal{G}$ .



## Chapitre 93

# (MT) Substitution régulière \*\* à \*\*\*\* (MT 25, ex 1, retour élève à reprendre, corrigé — oral - 92 lignes)

\*\* à \*\*\*\*

Langages, Automates, Grammaires,  
sources : mtlang7.tex

On se donne un alphabet fini  $\Sigma$ , et l'on note  $\mathcal{L}$  l'ensemble des langages sur  $\Sigma$ .

On définit une *substitution régulière* comme une application  $f : \Sigma^* \rightarrow \mathcal{L}$  vérifiant les propriétés suivantes (où les langages dans l'ensemble image sont dénotés par des expressions régulières) :

- $f(\epsilon) = \epsilon$ .
- $\forall \omega, \omega' \in \Sigma^*, f(\omega.\omega') = f(\omega).f(\omega')$ .
- $\forall a \in \Sigma, f(a)$  est un dénoté par une expression régulière.

Si  $L \in \mathcal{L}$ , on définit alors  $f(L) = \bigcup_{u \in L} f(u)$ .

1. Montrer que si  $L$  est un langage régulier,  $L' = f(L)$  est également un langage régulier.
2. Montrer que si  $L$  est un langage non contextuel,  $L' = f(L)$  est également un langage non contextuel.
3. On considère le langage  $G_0 = \{b^p a^q c, q \neq p + 1\}$ . Montrer que  $G_0$  est un langage non contextuel.
4. Montrer que  $G_0$  n'est pas régulier.
5. ???? Appliquer une certaine substitution régulière à  $G_0$  pour obtenir un certain langage  $G$  donné.
6. Montrer que  $G$  est non contextuel.



## Chapitre 94

# (MT) Langages et facteurs \* (MT 25, ex 1, retour élève, corrigé (R) — oral - 150 lignes)

\*

Langages, Automates,  
sources : `mtlang3.tex`

1. On note  $L_1$  l'ensemble des mots sur  $\{a, b\}$  ne contenant pas le facteur  $ab$ .  
Trouver une expression régulière dénotant ce langage, et prouver qu'elle est correcte.
2. On note  $L_2$  l'ensemble des mots sur  $\{a, b, c\}$  ne contenant pas le facteur  $ab$ .  
Trouver un automate qui reconnaît  $L_2$ .
3. Appliquer l'algorithme d'élimination des états sur cet automate pour trouver une expression régulière  $R$  qui dénote  $L_2$ .
4. Appliquer l'algorithme de Berry-Sethi pour trouver un automate reconnaissant  $L_1$ .



# Chapitre 95

## (MT) Dédution naturelle \* (MT 25, ex 1, retour élève, corrigé — oral - 97 lignes)

\*

Dédution naturelle,  
sources : mtlog5.tex

1. Prouver le séquent

$$\vdash p \longrightarrow (q \longrightarrow (r \longrightarrow (p \wedge (q \wedge r))))).$$

2. Montrer que l'on ne peut pas prouver le séquent suivant en déduction naturelle :

$$p, \neg q \vdash p \longrightarrow \neg(q \longrightarrow \perp)$$

3. Montrer que  $p_n \longrightarrow (\dots \longrightarrow (p_0 \longrightarrow (p_n \wedge (p_1 \wedge (\dots \wedge p_0)) \dots)))$  est une tautologie en utilisant la déduction naturelle.

### Annexe : règles de la déduction naturelle

#### Introduction et élimination de $\rightarrow$

$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \rightarrow \psi} \rightarrow_i$	$\frac{\Gamma \vdash \varphi \rightarrow \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi} \rightarrow_e$
--	---

#### Introduction et élimination de $\wedge$

$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi} \wedge_i$	$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} \wedge_e^g$	$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi} \wedge_e^d$
---	--	---



## Chapitre 96

**(MT)  $k$  plus proches voisins \* (MT 25, ex 1, retour élève, corrigé — oral - 46 lignes)**

\*

la,

sources : mtia2.tex

1. Expliquer le principe de l'algorithme des  $k$ -plus proches voisins. Donner sa complexité en fonction de  $k$ , du nombre de points de données  $n$  et de la dimension  $d$  de l'espace dans lequel sont ces points.
2. Donner une façon d'améliorer cette complexité en dimension 1.
3. Donner un autre algorithme qui répond au même problème que les  $k$ -plus proches voisins.



## Chapitre 97

# (MT) lemme d'Arden \*\* (MT 25, ex 1, retour élève, corrigé — oral - 94 lignes)

\*\*

Langages,  
sources : mtlang5.tex

Sur l'alphabet  $\Sigma = \{a, b\}$ , on définit l'équation suivante

$$L = (\{a\}L) \cup \{b\} \quad (E)$$

d'inconnue  $L$  un langage sur  $\Sigma$ .

1. Montrer que  $(E)$  admet une unique solution, que l'on explicitera.

On généralise en donnant deux langages  $A$  et  $B$  sur un même alphabet tels que  $\epsilon \notin A$ , et l'équation suivante :

$$L = AL \cup B \quad (E')$$

2. Montrer que  $(E')$  admet une unique solution  $L = A^*B$ .

3. Que peut-on dire de  $(E')$  si  $\epsilon \in A$ ?



# Chapitre 98

## (MT) \* (MT 25, ex 2, retour élève remanié, corrigé — oral - 117 lignes)

\*

Complexité, Algorithmes d'approximation, Réduction,  
sources : `mtapprox2.tex`

On définit les deux problèmes suivants.

### Problème 1

entrées :  $G = (S, A)$  orienté,  $k \in \mathbb{N}$ .

sortie : Existe-t-il  $E \subset A$  tel que  $|E| = k$  et tous les cycles de  $G$  ont un arc dans  $E$  ?

### Problème 2

entrées :  $G = (S, A)$  orienté,  $k \in \mathbb{N}$ .

sortie : Existe-t-il  $A' \subset A$  tel que  $|A'| = k$  et  $G' = (S, A')$  est acyclique ?

On admet que le problème 1 est **NP-complet**.

1. Montrer que le problème 2 est **NP-complet**.
2. Définir un problème d'optimisation  $\mathcal{P}_{opt}$  associé à 2.

On considère l'algorithme suivant pour ce problème d'optimisation :

Algorithme1 :

Fixer un ordre arbitraire sur  $V$ .

$A_1 \leftarrow$  l'ensemble des arcs  $(e, f)$  tels que  $f > e$ .

$A_2 \leftarrow A \setminus A_1$ .

**Si**  $|A_1| > |A_2|$  :

renvoyer  $A_1$

**Sinon** :

renvoyer  $A_2$

3. Montrer que cet algorithme renvoie bien un ensemble d'arcs valide pour  $\mathcal{P}_{opt}$ .
4. Montrer que cet algorithme est une  $1/2$ -approximation pour  $\mathcal{P}_{opt}$ .
5. Discuter sa complexité.



## Chapitre 99

# (MT) Représentation fusionnée de deux mots \*\* (MT 25, ex 2, retour élève remanié, corrigé (R) — oral - 116 lignes)

\*\*

Algorithmique des textes, Programmation dynamique,  
sources : `mttxt1.tex`

On s'intéresse dans cet exercice à une représentation compacte de deux mots sur un même alphabet  $\Sigma$ . Plus précisément, étant donnés deux mots  $A$  et  $B$  sur  $\Sigma$ , on appelle *résumé* de  $A$  et  $B$  tout mot sur l'alphabet  $\Sigma \cup \{\uparrow, \downarrow, \updownarrow\}$  de la forme

$R = f_1 u_1 \cdots f_n u_n$  où

- $\forall i \in \llbracket 1, n \rrbracket$ ,  $f_i \in \{\uparrow, \downarrow, \updownarrow\}$ , et  $u_i \in \Sigma^+$
- $A$  est la concaténation des  $u_i$  pour les  $i$  tels que  $i \in \{\uparrow, \updownarrow\}$
- $B$  est la concaténation des  $u_i$  pour les  $i$  tels que  $i \in \{\downarrow, \updownarrow\}$

On se donne un entier  $\Delta \in \mathbb{N}^*$ , et l'on définit la *taille* de  $R$  par  $\ell_\Delta(R) = n\Delta + \sum_{i=1}^n |u_i|$ .

Par exemple, pour  $A = \ll CHATON \gg$  et  $B = \ll CAMION \gg$ , on peut obtenir les résumés suivants

- $\updownarrow C \uparrow H \updownarrow A \downarrow MI \uparrow T \updownarrow ON$ , de longueur  $8 + 6\Delta$ ,
- $\uparrow CHATON \downarrow CAMION$ , de longueur  $12 + 2\Delta$ ,
- $\uparrow CHAT \downarrow CAMI \updownarrow ON$ .

1. Quelle est la taille de ce dernier résumé ?
2. En fonction de  $\Delta$ , lequel des trois résumés ci-dessus a la taille minimale ?

On note  $m^{(i)}$  le mot composé des  $i$  premières lettres de  $m$ , et  $d(i, j)$  la taille minimale d'un résumé de  $A^{(i)}$  et  $B^{(j)}$ .

3. Que valent  $d(i, 0)$  et  $d(0, j)$  pour tout  $i, j \in \mathbb{N}$  ?  
Que représente  $d(|A|, |B|)$  ?
4. Trouver une relation de récurrence pour  $d(i, j)$ .
5. Donner en pseudocode un algorithme prenant en entrée  $\Delta$  et deux mots  $A$  et  $B$ , et renvoyant la taille minimale d'un résumé des deux mots.
6. Quelle est sa complexité ?



## Chapitre 100

# (MT) Graphes et couleurs \* à \*\*\* (MT 25, ex 2, retour élève remanié, corrigé — oral - 128 lignes)

\* à \*\*\*

Graphes, Algorithmes probabilistes,  
sources : `mtgraph7.tex`

On considère un graphe  $G = (S, A)$  non orienté, connexe et ayant au moins deux sommets.

On se donne une fonction  $\gamma(s)$  indiquant si le sommet  $s$  est ROUGE ou NOIR.

On considère le processus suivant – que l'on applique simultanément à chaque sommet :

- chaque sommet  $s$  tire l'un de ses voisins  $t$  uniformément au hasard
- $s$  prend la couleur de  $t$ .

On répète indéfiniment ce processus.

On dit qu'il y a *consensus* à l'issue d'une itération donnée si tous les sommets sont de la même couleur à l'issue de cette itération.

1. Donner un exemple où la situation de consensus ne sera jamais atteinte.
2. Montrer que pour tout graphe biparti, il existe une configuration initiale de couleurs empêchant qu'il y ait jamais un consensus.
3. Soient  $s_1, s_2$  et  $s_3 \in S$  tels que  $s_1$  et  $s_2$  soient voisins, et que  $\gamma(s_1) = \gamma(s_2) \neq \gamma(s_3)$ .  
On note  $p$  la probabilité d'obtenir un consensus à l'étape suivante. Montrer que  $p < 1$ .
4. On se place dans la même situation qu'à la question précédente, et l'on note  $C$  la composante connexe sommets de couleur  $\gamma(s_1)$  contenant  $s_1$ . On note  $C'$  cette composante connexe après une itération. On note  $q$  la probabilité que  $|C'| > |C|$ . Montrer que  $q > 0$ .
5. Montrer que pour tout graphe non-biparti, la probabilité de finir par obtenir un consensus est égale à 1.



## Chapitre 101

# (MT) Langages rationnels et ultime périodicité \*\*\* (MT 25, ex 2, retour élève remanié, corrigé — oral - 82 lignes)

\*\*\*

Langages, Lemme de l'étoile,  
sources : `mtlangper.tex`

On se place sur l'alphabet  $\Sigma = \{a\}$ .

1.  $\{a^{3k+1}, k \in \mathbb{N}\}$  est-il rationnel ?
2.  $\{a^{3^k+1}, k \in \mathbb{N}\}$  est-il rationnel ?

On note  $\text{Taille}(L)$  l'ensemble des tailles des mots  $m$  tels que  $m$  appartienne à  $L$ .

On dit qu'un ensemble  $E$  est ultimement périodique s'il existe  $P \in \mathbb{N}$  et  $N \geq P$  tels que pour tout  $n \geq N$ ,  $n \in E \Rightarrow n + P \in E$ .

3. Montrer qu'un langage  $L$  sur l'alphabet  $\Sigma$  est régulier si et seulement si  $\text{Taille}(L)$  est ultimement périodique.
4. Pour un alphabet quelconque, que dire du lien entre ultime périodicité et régularité du langage ?



## Chapitre 102

# (MT) Théorème de compacité \*\* (MT 25, ex 2, retour élève remanié, corrigé — oral - 101 lignes)

\*\*

Logique propositionnelle,  
sources : mtlog4.tex

On considère pour l'instant un ensemble fini de  $n \geq 1$  variables propositionnelles  $\mathcal{V} = \{p_0 \cdots p_{n-1}\}$ .

On dit qu'un ensemble  $E$  de formules propositionnelles sur ces variables est *satisfiable* s'il existe une valuation  $\mu$  sur  $\mathcal{V}$  telle que  $\mu$  soit un modèle de toute formule de  $E$ .

1. Quel est le nombre de tables de vérités possibles pour les formules utilisant ces variables ?
2. On considère un ensemble  $E$  dénombrable infini de formules propositionnelles. Montrer qu'il existe un ensemble fini  $F \subset E$  tel que  $F$  est satisfiable ssi  $E$  est satisfiable.
3. Prouver le théorème de compacité :  $E$  est satisfiable ssi tout  $F$  fini inclus dans  $E$  l'est.

On considère à partir de maintenant un ensemble dénombrable de variables  $\mathcal{V} = \{p_0, p_1, p_2 \cdots\}$ .

4. On considère  $E = \bigcup_{i \in \mathbb{N}} \{p_i, \neg p_i\}$ .

Montrer que chaque formule contenue dans  $E$  est satisfiable, mais que  $E$  ne l'est pas.

5. On considère  $E = \bigcup_{i \in \mathbb{N}} \{p_{2i} \rightarrow p_{2i+1}, p_{2i+1} \rightarrow \neg p_{2i+2}, \neg p_{2i} \rightarrow p_{2i+2}\}$ .

Chaque formule contenue dans  $E$  est-elle satisfiable ?

$E$  est-elle satisfiable.



## Chapitre 103

# (MT) Programmation dynamique \*\*\* (MT 25, ex 2, retour élève retravaillé, corrigé — oral - 131 lignes)

\*\*\*

Programmation dynamique, Réduction,  
sources : `mtdyn.tex`

On considère une matrice de taille  $n \times n$  remplie d'entiers. Chaque entier est associé à une couleur. On dit que le 0 correspond à la couleur blanche.

On note  $D_{i,j,k} = 1$  si et seulement si il y a un carré de côté  $k$  dont le coin en haut à gauche est  $(i, j)$  entièrement constitué de cases blanches,  $D_{i,j,k} = 0$  sinon.

1. Proposer un algorithme (de complexité au plus  $O(n^3)$ ) permettant de trouver le plus grand carré blanc possible.
2. Adapter l'algorithme précédent pour trouver le plus grand carré de chaque couleur.

On donne en annexe trois problèmes classiques dont on admet le caractère **NP**-complet.

On considère maintenant le problème suivant :

**CARRE1**

**Entrée** : une matrice  $n \times n$  d'entiers et un entier  $k$ .

**Sortie** : existe-t-il une permutation des lignes permettant d'obtenir un carré blanc de taille  $k$  ?

3. Montrer que ce problème est **NP**-complet, ou bien donner un algorithme polynomial le résolvant.

On considère maintenant le problème suivant :

**CARRE2**

**Entrée** : une matrice  $n \times n$  d'entiers et un entier  $k$ .

**Sortie** : existe-t-il une permutation des lignes et une permutations des colonnes permettant d'obtenir un carré blanc de taille  $k$  ?

4. Montrer que ce problème est **NP**-complet, ou bien donner un algorithme polynomial le résolvant.

### Annexe :

On indique que les problèmes suivants sont **NP**-complet :

$k$ -SAT (où  $k \geq 3$ )

**Entrée** : une formule sous forme normale conjonctive, avec au plus 3 littéraux par clauses.

**Sortie** : la formule est-elle satisfiable ?

**CLIQUE**

**Entrée** : Un graphe  $G = (S, A)$  non orienté, et  $k \in \mathbb{N}$ .

**Sortie** : existe-t-il  $S' \subset S$  tel que  $|S'| = k$  et que tous sommets distincts  $s, t \in S'$  sont reliés par un arête ?

**BICLIQUE**

**Entrée** : Un graphe  $G = (S, A)$  non-orienté biparti (avec une bipartition  $S = U \cup V$  telle que  $|U| = |V|$ ), et  $k \in \mathbb{N}$ .

**Sortie** : existe-t-il  $U' \subset U$  et  $V' \subset V$  tels que  $|U'| = |V'| = k$  et que pour tout  $(u, v) \in U' \times V'$ , il y a une arête entre  $u$  et  $v$  ?



# Chapitre 104

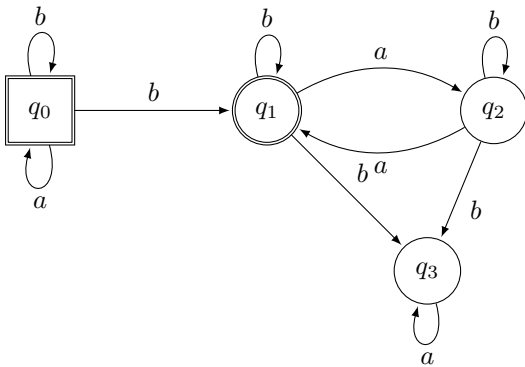
## (MT) Automates alternants \*\*\* (MT 25, ex 2, retour élève très buggué largement retravaillé, corrigé — oral - 294 lignes)

\*\*\*

Langages, Automates, Jeux,

sources : mtautomatesalternants.tex

On appelle  $\mathcal{A}_1$  l'automate suivant.



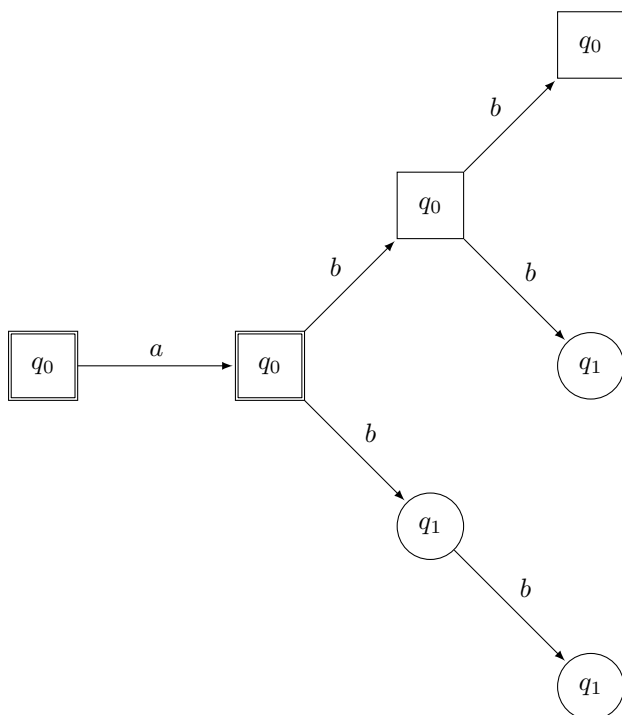
On appelle *automate alternant* la donnée d'un sextuplet  $\mathcal{A} = (\Sigma, Q_{\forall}, Q_{\exists}, q_0, F, \delta)$ .

Dans le cas ci-dessus,  $\mathcal{A}_1 = (\{a, b\}, \{q_0\}, \{q_1, q_2, q_3\}, q_0, \{q_0, q_1\}, \delta)$ .

Un *calcul* d'un mot  $u = u_0 \cdots u_{n-1}$  dans cet automate est un arbre de hauteur  $|u|$  (avec la convention qu'un arbre réduit à sa racine est de hauteur 0), dont les sommets sont étiquetés par des sommets de l'automate, et tel que

- la racine est l'état initial
- si un sommet  $s \in Q_{\forall}$  est à profondeur  $i$ , alors  $s$  a pour fils tous les  $\delta(s, u_i)$ .
- si un sommet  $s \in Q_{\exists}$  est à profondeur  $i$ , alors  $s$  a pour fils l'un  $\delta(s, u_i)$ .

Par exemple, un calcul de «abb» dans  $\mathcal{A}_1$  est



Un calcul est dit *réussi* dans un automate lorsque toutes les feuilles du calcul sont des états acceptants.

Un mot est *accepté* par l'automate s'il existe un calcul réussi pour ce mot.

Le *graphe des configurations* d'un automate  $G(u, \mathcal{A})$  est un graphe pour lequel  $S = Q \times \{0 \dots |u|\}$  et il existe un arc allant de  $(q, i)$  à  $(q', i + 1)$  si et seulement si  $q' \in \delta(q, u_{i+1})$ .

On définit enfin le jeu suivant, se jouant sur un graphe des configurations à partir de l'état  $(q_0, 0)$ .

**Jeu :** Pathfinder & Automaton.

- Pathfinder contrôle les états existentiels, et Automaton les états universels.
- Une partie s'arrête lorsque le sommet en jeu n'a pas d'arête sortante.
- Pathfinder gagne si le jeu se termine dans  $F \times \{|u|\}$ .
- Automaton gagne dans tous les autres cas.

On attire l'attention sur le fait que *les joueurs ne jouent pas nécessairement à tour de rôle* : à tout moment, si l'on est dans un état  $(q, i)$ , c'est le joueur contrôlant  $q$  qui joue.

1. Exhiber un calcul pour le mot  $bbaa$ . Ce mot est-il accepté par l'automate ?
2. Donner en justifiant rapidement un mot non accepté par l'automate.
3. Dessiner  $G(abbaa, \mathcal{A}_1)$  en se restreignant aux états accessibles à partir de  $(q_0, 0)$ .
4. Montrer un état non final gagnant pour Automaton, et un pour Pathfinder dans l'instance de la question précédente.
5. Quel personnage a une stratégie gagnante dans cette instance ?

On s'intéresse maintenant à un automate alternant  $\mathcal{A}$  quelconque.

6. Montrer que si  $u$  est accepté par l'automate  $\mathcal{A}$ , alors Pathfinder possède une stratégie gagnante dans le jeu correspondant.
7. Prouver la réciproque.

## Chapitre 105

# (MT) Graphes : implémentation en C \*

## (MT 25, ex 2, retour élève à compléter/vérifier++, corrigé — oral - 98 lignes)

\*

Graphes, C,  
sources : mtgraph11.tex

On considère le code C suivant :

```
struct sommet {
    uint8_t degre_sortant;
    struct sommet** successeurs;
}

typedef struct sommet sommet;

struct graphe = sommet*;

int main() {
    sommet s0, s1, s2, s3;
    s0.successeurs = malloc(4*sizeof(sommet *));
    s1.successeurs = malloc(4*sizeof(sommet *));
    s2.successeurs = malloc(4*sizeof(sommet *));
    s3.successeurs = malloc(4*sizeof(sommet *));

    s0.degre_sortant = 2;
    s1.degre_sortant = 2;
    s2.degre_sortant = 1;
    s3.degre_sortant = 1;

    s0.successeurs[0] = &s1;
    s0.successeurs[1] = &s2;

    s1.successeurs[0] = &s0;
    s1.successeurs[1] = &s2;

    s2.successeurs[0] = &s3;
    s3.successeurs[0] = &s1;

    graphe g = &s0;
}
```

1. Dans quelle partie de la mémoire sont `s0`, `s0.successeurs`, et `g`?
2. Tracer le graphe correspondant.
3. Y a-t-il des graphes que l'on ne peut pas créer avec cette structure?
4. Définir une représentation de graphe par listes d'adjacence.

5. Comme modifier la structure donnée pour implémenter un graphe par listes d'adjacence ?
6. Avec cette nouvelle représentation, écrire une fonction `void supprime(graphe g, sommet* s)` qui supprime le sommet `s` du graphe.
7. 3 questions non faites.

## Chapitre 106

# (MT) League et Overwatch \*\* (MT 25, ex 2, retour élève à reprendre, non corrigé — oral - 91 lignes)

\*\*

Concurrence,  
sources : `mtsync.tex`

On a deux groupes de personnes : celles qui jouent à League of Legends, et celles qui jouent à Overwatch.

Ces deux groupes partagent le même espace de détente, et pour éviter les conflits, on impose les contraintes suivantes :

- on ne peut pas avoir deux personnes venant de groupes différents dans l'espace ;
- on ne peut avoir plus de cinq personnes dans l'espace.

On souhaite écrire des fonction `league_entrer`, `league_sortir`, et les fonctions équivalentes pour Overwatch permettant de gérer cette situation.

1. Pourquoi est-il suffisant d'écrire les deux fonctions pour League ?
2. Écrire en pseudocode des fonctions `league_` grâce à un mutex, des variables globales de votre choix et de l'attente active.
3. On essaye maintenant d'éviter l'attente active, et on souhaite proposer une implémentation à l'aide de
  - deux compteurs : `nb_league` et `nb_overwatch`
  - deux mutex protégeant ces compteurs : `m_league` et `m_overwatch`
  - un sémaphore `vide` qui indique si la salle est vide
  - deux sémaphores `s_league` et `s_overwatch`Proposez une implémentation correspondante.
4. Y a-t-il un risque de famine avec cette approche ?



## Chapitre 107

# (MT) coloriage de graphe \* (MT 25, ex 2, retour élève, 3/4 questions manquantes, corrigé — oral - 103 lignes)

\*

Graphes, Pseudo-code,  
sources : `mtgraph6.tex`

Soit  $G = (S, A)$  un graphe non orienté.

Soit  $c : S \rightarrow \llbracket 0, k - 1 \rrbracket$  une fonction donnant un *coloriage* du graphe.

On dit que  $G$  est *k-coloriable* ssi il existe une telle fonction  $c$  telle que  $\forall \{u, v\} \in A, c(u) \neq c(v)$ .

On note  $\chi(G)$  la plus petite valeur de  $k$  telle que  $G$  est *k-coloriable*.

1. Donner un algorithme permettant de déterminer si  $G$  est *k-coloriable*.
2. On note  $v_s^i$  une variable propositionnelle que l'on interprète comme «vraie» si le sommet  $s$  est colorié de la couleur  $i$ . Donner une formule propositionnelle qui est vraie ssi  $G$  est *k-coloriable*.
3. Montrer que  $\chi \leq \Delta(G) + 1$  où  $\Delta(G)$  est le degré maximum de  $G$ .
4. Donner un exemple de graphe  $G$  tel que  $\chi < \Delta(G) + 1$ .
5. Donner un exemple de graphe  $G$  tel que  $\chi = \Delta(G) + 1$  (avec  $\chi = n \in \mathbb{N}^*$  quelconque).
6. Montrer que déterminer si un graphe est *k-coloriable* est dans la classe **NP**.
7. Montrer que déterminer si un graphe est 2-coloriable est dans la classe **P**.
8. On admet que déterminer si un graphe est 3-coloriable est un problème **NP-complet**. Montrer que pour tout  $k \geq 4$ , déterminer si un graphe est *k-coloriable* est également un problème **NP-complet**.



## Chapitre 108

# (MT) CNS pour cycles eulériens \* (MT 25, ex 2, retour élève, corrigé — oral - 71 lignes)

\*

Graphes,  
sources : `mtgraph9.tex`

On se donne un graphe non orienté  $G = (S, A)$  *connexe*.

On appelle *cycle eulérien* de ce graphe un cycle passant exactement une fois par chaque arête du graphe.

1. Si  $G$  a un cycle eulérien, montrer que tous ses sommets sont de degré pair.

On considère à partir de maintenant un graphe  $G$  dont tous les sommets sont de degré pair.

On appelle *chemin eulérien partiel* tout chemin dans le graphe qui ne passe pas deux fois par la même arête.

2. On fixe un sommet  $s$ . Montrer que tout chemin eulérien partiel maximal partant de  $s$  est un cycle (i.e. que son dernier sommet est également  $s$ ).
3. Soient  $C$  et  $C'$  deux cycles distincts ayant un sommet en commun, mais aucune arête en commun. Expliquer comment construire un cycle  $C''$  contenant toutes les arêtes de  $C$  et de  $C'$ .
4. Prouver la réciproque du résultat de la question 1.



## Chapitre 109

# (MT) Cycles et Chemins simples \*\* (MT 25, ex 2, retour élève, corrigé — oral - 64 lignes)

\*\*

Graphes, Programmation dynamique,  
sources : `mtgraph10.tex`

On se donne un graphe non orienté  $G = (S, A)$  contenant au moins un cycle.

1. Donner un algorithme qui trouve un cycle dans ce graphe.  
Quelle est sa complexité ?
2. Donner un algorithme qui trouve un plus court cycle dans ce graphe.  
Quelle est sa complexité ?

On suppose désormais que le graphe est orienté et les arcs pondérés par des poids  $p$  positifs.

3. Donner un algorithme qui trouve un cycle de poids minimal dans le graphe.
4. On note  $d_k(v)$  la longueur d'un plus court chemin d'un sommet initial  $s_0$  fixé au sommet  $v$ , passant exactement par  $k$  sommets. Donner un algorithme permettant de calculer  $d_k(v)$ .  
Quelle est sa complexité ?



# Chapitre 110

## (X) Algorithme Union-Find \*\*\* (X 23, corrigé — oral - 151 lignes)

\*\*\*

Algorithmique, Complexité, Structures de données,  
sources : xunionfind.tex

### Algorithme Union-Find

L'algorithme Union-Find a pour but de gérer dynamiquement une partition d'un ensemble  $\{1, 2, \dots, n\}$  fixé. Initialement, on part de la partition maximale  $\{\{1\}, \{2\}, \dots, \{n\}\}$ . En pratique, on représente les partitions de  $\{1, \dots, n\}$  comme suit :

- ▷ chaque entier  $k$  possède un (seul) père  $p_k \leq n$ , ce que l'on notera  $k \rightarrow p_k$  ; on peut avoir  $k = p_k$  ;
- ▷ chaque entier  $k$  possède un poids  $w_k \in \mathbb{N}$  ;
- ▷ deux entiers distincts  $k$  et  $\ell$  ne peuvent être ancêtres l'un de l'autre, et appartiennent au même sous-ensemble si et seulement s'ils ont un ancêtre commun ;
- ▷ si l'entier  $k$  appartient à un singleton,  $w_k = 0$ .

**Question 1.** On dit que  $m$  est le chef de  $k$  si  $m$  est un ancêtre de  $k$  tel que  $m = p_m$ . Démontrer que tout entier a un unique chef, et deux entiers  $k$  et  $\ell$  appartiennent au même sous-ensemble si et seulement s'ils ont le même chef.

L'algorithme est censé gérer trois types de requêtes, en procédant comme suit :

1. La requête auxiliaire **Chef**( $k$ ) : on recherche le chef de  $k$ . Si  $k = p_k$ , il s'agit de  $k$  lui-même. Sinon, il s'agit du chef de  $p_k$ , et ce chef devient le nouveau parent de  $k$ .
2. La requête **Test**( $k, \ell$ ) : on se demande si  $k$  et  $\ell$  appartiennent au même sous-ensemble. Cela revient à identifier les entiers  $k' = \mathbf{Chef}(k)$  et  $\ell' = \mathbf{Chef}(\ell)$  puis à vérifier si  $k' = \ell'$ .
3. La requête **Union**( $k, \ell$ ) : on souhaite réunir les sous-ensembles auxquels appartiennent  $k$  et  $\ell$ . Cela revient à identifier les entiers  $k' = \mathbf{Chef}(k)$  et  $\ell' = \mathbf{Chef}(\ell)$ , puis :
  - ▷ si  $w_{k'} > w_{\ell'}$ , l'entier  $k'$  devient le nouveau parent de  $\ell'$  ;
  - ▷ si  $w_{\ell'} > w_{k'}$ , l'entier  $\ell'$  devient le nouveau parent de  $k'$  ;
  - ▷ si  $k' \neq \ell'$  et  $w_{k'} = w_{\ell'}$ , l'entier  $k'$  devient le nouveau parent de  $\ell'$  et son poids augmente de 1.

On souhaite démontrer que répondre à  $m$  de ces requêtes peut se faire en temps  $\mathcal{O}(m \log^*(m))$ , où  $\log^*$  est la fonction définie par  $\log^*(m) = 1$  lorsque  $m \leq 1$  et  $\log^*(m) = 1 + \log^*(\log_2(m))$  lorsque  $m > 1$ .

**Question 2.** En partant de l'ensemble  $\{\{1\}, \{2\}, \{3\}, \{4\}\}$ , on effectue successivement les requêtes **Union**(1, 2), **Union**(3, 4), **Union**(2, 4) et **Test**(2, 4). Indiquer le père et le poids de chaque entier  $k \leq 4$ .

**Question 3.** On note  $w_k^\infty$  le poids d'un entier  $k$  à la fin de l'algorithme. Démontrer que  $w_u^\infty < w_v^\infty$  pour tous les entiers  $u \neq v$  tels que  $v$  a été le parent de  $u$ .

**Question 4.** Démontrer, pour tout entier  $\ell \geq 1$ , qu'il y a au plus  $m/2^{\ell-1}$  entiers  $k \leq n$  pour lesquels  $w_k^\infty = \ell$ .

**Question 5.** Démontrer que toute requête **Test** ou **Union** effectuée au cours de l'algorithme a une complexité majorée par  $\mathcal{O}(\log_2(\min\{m, n\}))$ .

**Question 6.** Soit  $\mathcal{G} = (V, E)$  le graphe dont les sommets sont les entiers de 1 à  $n$  et les arêtes sont les paires  $(u, v)$  pour lesquelles  $u \neq v$  et  $v$  a été le parent de  $u$  au cours de l'algorithme, mais pas quand celui-ci se termine. Démontrer que la complexité totale de nos  $m$  requêtes est majorée par  $\mathcal{O}(m + |E|)$ .

**Question 7.** Soit  $(a_\ell)_{\ell \geq 0}$  la suite définie par  $a_0 = 0$  et  $a_{\ell+1} = 2^{a_\ell}$ . Pour tout entier  $\ell \geq 0$ , on note  $V_\ell$  l'ensemble des entiers  $k$  tels que  $a_\ell \leq w_k^\infty < a_{\ell+1}$ . Démontrer que  $\mathcal{G}$  contient au plus  $4m$  arêtes reliant deux sommets de  $V_\ell$ , et au plus  $2m$  arêtes allant d'un sommet de  $V_\ell$  à un sommet en dehors de  $V_\ell$ .

**Question 8.** Que conclure sur la complexité de l'algorithme **Union-Find** ?

# Chapitre 111

## (X) Bob l'écureuil \*\*\* (X 23, corrigé — oral - 141 lignes)

\*\*\*

Algorithmique, Graphes,  
sources : `xbobecureuil.tex`

Bob est un écureuil qui vit le long d'une ligne de chemin de fer. Il a caché ses réserves de noisettes dans les différentes gares, et cherche maintenant à établir son nid dans l'une de ces gares. Pour des raisons pratiques, il souhaite construire son nid dans une gare où il a déjà caché des noisettes, tout en minimisant la durée moyenne entre son nid et ses réserves de noisettes. Comment choisir dans quelle gare établir son nid ?

**Question 1.** La ligne est composée de 9 gares, numérotées de 0 à 8 ; Bob a mis une cachette dans chacune de ces gares. On met  $k$  minutes pour aller de la gare  $\ell$  à la gare  $k + \ell$ . Quelle gare Bob choisira-t-il d'établir son nid ?

**Question 2.** On suppose maintenant que les gares ne sont plus régulièrement espacées. Désormais, il y a  $n$  gares, toujours numérotées de 0 à  $n - 1$ , toujours en ligne droite. Bob connaît des entiers  $t_0, t_1, \dots, t_{n-2}$  pour lesquels aller de la gare  $\ell$  à la gare  $\ell + 1$  requiert  $t_\ell$  minutes ; ainsi, aller de la gare  $\ell$  à la gare  $\ell + 2$  requiert  $t_\ell + t_{\ell+1}$  minutes. Donner un algorithme qui permettra à Bob de choisir, en temps  $\mathcal{O}(n)$ , dans quelle gare installer son nid.

**Question 3.** Désormais, le réseau prend la forme d'un graphe connexe acyclique. Bob dispose, pour chaque gare  $k$ , d'une liste des gares  $\ell$  auxquelles la gare  $k$  est reliée directement, ainsi que de la durée  $d_{k,\ell}$ , en nombre de minutes, du trajet entre les gares  $k$  et  $\ell$ .

Donner un algorithme qui permettra à Bob de choisir, en temps  $\mathcal{O}(n)$ , dans quelle gare installer son nid.

Bob déménage à Manhattan, là où toutes les rues sont orientées selon un axe Nord-Sud ou Est-Ouest, et espacées de 100 mètres l'une de l'autre. Il a caché ses  $n$  réserves de noisettes à  $n$  croisements de rue, chaque croisement étant identifié par des coordonnées entières. Par exemple, le croisement  $(2, 3)$  se trouve 400 mètres à l'Ouest et 500 mètres au Sud du croisement  $(6, 8)$ , donc Bob doit marcher au minimum 900 mètres pour aller d'un croisement à l'autre. Bob souhaite maintenant établir son nid à l'un des  $n$  croisements de rue où se trouve déjà une réserve de noisettes, mais tout en minimisant la distance moyenne entre son nid et des réserves de noisettes. À quel croisement établir son nid ?

**Question 4.** Bob a disposé ses 6 réserves aux points de coordonnées  $(0, 0), (1, 1), (2, 3), (3, 1), (3, 2)$  et  $(4, 3)$ . Auquel de ces six endroits choisira-t-il d'établir son nid ?

**Question 5.** Bob a simplement noté les coordonnées entières  $(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})$  des  $n$  réserves qu'il a choisies pour entreposer ses noisettes. Donner un algorithme qui permettra à Bob de choisir, en temps  $\mathcal{O}(n \log(n))$ , à quelle intersection installer son nid.

**Question 6.** On suppose désormais que chacune des coordonnées  $x_i$  et  $y_i$  est comprise entre 0 et  $n^2 - 1$ . Comment adapter l'algorithme précédent pour s'assurer qu'il fonctionne désormais en temps  $\mathcal{O}(n)$  ?

**Question 7.** Pour fluidifier le trafic, des urbanistes ont ajouté des routes orientées du Nord-Est vers le Sud-Ouest, et du Nord-Ouest vers le Sud-Est. Ainsi, le croisement  $(2, 3)$  se désormais à  $400\sqrt{2} + 100 \approx 666$  mètres du croisement  $(6, 8)$  : il suffit d'aller quatre fois vers le Nord-Est puis une fois vers le Nord. On suppose toujours que chacune des coordonnées  $x_i$  et  $y_i$  est comprise entre 0 et  $n^2 - 1$ , et que Bob veut installer son nid en l'une des  $n$  intersections  $(x_i, y_i)$ . Donner un algorithme qui permettra à Bob de choisir, en temps  $\mathcal{O}(n)$ , à quelle intersection installer son nid.

# Chapitre 112

## (X) Langages rationnels et lemme de l'étoile \*\*\* (X 23, corrigé — oral - 118 lignes)

\*\*\*

Langages, Automates, Lemme de l'étoile,  
sources : xlangrat.tex

### Langages rationnels et lemme de l'étoile

On recherche un langage  $\mathcal{L}$  non rationnel mais pour lequel  $\mathcal{L}$  et son complémentaire satisfont le lemme de l'étoile.

**Question 1.** Soit  $\Sigma$  un alphabet fini. Le langage  $\mathcal{L}_1$  formé des mots  $w \in \Sigma^*$  dont la longueur est paire est-il rationnel ?

**Question 2.** Le langage  $\mathcal{L}_2$  formé des mots  $w \in \Sigma^*$  dont la longueur est le carré d'un entier est-il rationnel ?

Soit  $\mathcal{L}$  et  $\mathcal{L}'$  deux langages sur un alphabet  $\Sigma$ . On dit que  $\mathcal{L}$  est  $\mathcal{L}'$ -étoilé s'il existe un entier  $n \geq 0$  tel que tout mot  $w \in \mathcal{L}'$  de longueur  $|w| \geq n$  admet une factorisation  $w = s \cdot t \cdot u$  pour laquelle  $|s| \leq n, 1 \leq |t| \leq n$  et  $s \cdot t^* \cdot u \subseteq \mathcal{L} \cup \{w\}$ . Si  $\mathcal{L}$  est  $\mathcal{L}$ -étoilé, on dit même que  $\mathcal{L}$  satisfait le lemme de l'étoile.

**Question 3.** Démontrer que tout langage rationnel satisfait le lemme de l'étoile.

**Question 4.** Démontrer que, si  $|\Sigma| = 1$ , les langages qui satisfont le lemme de l'étoile sont les langages rationnels.

Soit  $\mathcal{S}$  une partie finie de  $\Sigma^*$ . On note  $\mathcal{M}(\mathcal{S})$  l'ensemble des mots  $s_1 \cdot s_2 \cdots s_n$  que l'on peut factoriser en  $n$  éléments de  $\mathcal{S}$  et pour lesquels il existe deux entiers  $i$  et  $j = i + 1$  ou  $j = i + 2$  tels que  $s_i = s_j$ .

**Question 5.** Démontrer que tout ensemble  $\mathcal{M}(\mathcal{S})$  est rationnel.

**Question 6.** Démontrer que, si  $|\mathcal{S}| \leq 4$ , le langage  $\mathcal{M}(\mathcal{S})$  est  $\mathcal{S}^*$ -étoilé.

**Question 7.** On considère l'alphabet  $\Sigma = \{a, b, c, d\}$ . Pour tout entier  $k \leq 3$ , on note  $\sigma_k$  le mot  $(abc)^k \cdot (abd)^{3-k}$ . Les langages  $\mathcal{L}_3 = \left\{ (\sigma_0 \cdot \sigma_1 \cdot \sigma_2)^\ell \cdot (\sigma_0 \cdot \sigma_1 \cdot \sigma_3)^\ell : \ell \geq 0 \right\}$  et  $\mathcal{L} = \mathcal{M}(\{\sigma_0, \sigma_1, \sigma_2, \sigma_3\}) \cup \mathcal{L}_3$  sont-ils rationnels ?

**Question 8.** Démontrer que les ensembles  $\mathcal{L}$  et  $\Sigma^* \setminus \mathcal{L}$  satisfont le lemme de l'étoile.



# Chapitre 113

## (X) Rationnalité de langages \*\*\* (X 24, corrigé, complété au jugé pour les trois dernières questions — oral - 213 lignes)

\*\*\*

Langages, Automates, Lemme de l'étoile,  
sources : xlangrat2.tex

1. Pour les langages suivants, établir s'ils sont rationnels ou non :
  - (a) Les mots de  $\{0, 1\}^*$  dont la somme des chiffres est paire.
  - (b) Les mots de la forme  $0^k 1^\ell$ , où  $k \wedge \ell = 1$ .
2. Soit  $\Sigma$  un alphabet. Pour un mot  $w \in \Sigma^*$ , on définit  $\text{Pompe}(w)$  comme le plus petit langage sur  $\Sigma$  contenant  $w$  et tel que  $\forall (x, y, z) \in \Sigma^+ \times \Sigma^* \times \Sigma^+ \quad xyz \in L \Rightarrow xyyz \in L$ . Que dire de  $\text{Pompe}(ab)$ , où  $\Sigma = \{a, b\}$  ?
3. On définit  $\text{Cycle}(L) = \{vu \mid uv \in L\}$ . Si  $L$  est rationnel,  $\text{Cycle}(L)$  l'est-il ? Que pensez vous de la réciproque ?

On définit le *mélange*  $\wr$  de deux mots  $u$  et  $v$  par

$$\epsilon \wr u = \{u\} \quad v \wr \epsilon = \{v\}$$

$$u_1 u_{\geq 2} \wr v_1 v_{\geq 2} = \{u_1 x, x \in u_{\geq 2} \wr v_1 v_{\geq 2}\} \cup \{v_1 x, x \in u_1 u_{\geq 2} \wr v_{\geq 2}\}$$

4. Si  $L$  et  $L'$  sont rationnels, est-ce que  $L \wr L'$  l'est ?
5. Soit  $L$  rationnel. On définit

$$\begin{aligned} L_0 &= L \\ L_{n+1} &= L \wr L_n \\ L_\infty &= \bigcup_{n=0}^{\infty} L_n \end{aligned}$$

$L_\infty$  est-il rationnel ?

Soit  $\sigma = \{a_1, a_2, \dots, a_k\}$  un alphabet. Le *vecteur de Parikh* d'un mot  $w$  est le vecteur  $p(w) \in \mathbb{N}^k$  donné par

$$p(w) = (|w|_{a_1}, |w|_{a_2}, \dots, |w|_{a_k}),$$

où  $|w|_{a_i}$  dénote le nombre d'occurrences de la lettre  $a_i$  dans le mot  $w$ .

Un sous-ensemble de  $\mathbb{N}^k$  est dit *linéaire* s'il est de la forme  $u_0 + u_1 \mathbb{N} + \dots + u_k \mathbb{N} = \{u_0 + t_1 u_1 + \dots + t_m u_m \mid t_1, \dots, t_m \in \mathbb{N}\}$  pour des vecteurs  $u_0, \dots, u_m \in \mathbb{N}^k$ .

Un sous-ensemble de  $\mathbb{N}^k$  est dit *semi-linéaire* s'il est une union finie de parties linéaires.

6. Montrer que pour tout ensemble semi-linéaire  $E$ , il existe un langage régulier  $L$  tel que l'ensemble  $P(L)$  des vecteurs de Parikh des mots de  $L$  est égal à  $E$ .
7. Démontrer la réciproque.
8. Prouver que le résultat de la question précédente reste vrai pour tout langage hors-contexte.



# Chapitre 114

## (X) Algorithme du tri lent \*\*\* (X 24, corrigé — oral - 147 lignes)

\*\*\*

Algorithmique, Tri, Complexité, Preuves,  
sources : xtrilent.tex

### Algorithme du tri lent

On souhaite trier un tableau  $A$  de longueur  $n$  en utilisant l'algorithme de TriLent présenté ci-dessous. L'objectif de ce problème est de démontrer que l'algorithme est correct et d'évaluer certaines statistiques liées à sa complexité.

```
1  Fonction TriLent( $A_{1\dots n}$ ) :  
2      pour  $i=1,2,\dots,n$   
3          pour  $j=1,2,\dots,n$  :  
4              si  $A_i < A_j$  : échanger  $A_i$  et  $A_j$ 
```

**Question 1.** Quelle est, en fonction de  $n$ , la complexité de l'algorithme de TriLent, en nombre de comparaisons, dans le pire cas ? dans le meilleur cas ?

**Question 2.** Un algorithme de tri est stable si deux objets  $A_i$  et  $A_j$  égaux pour notre fonction de comparaison et pour lesquels  $i < j$  sont envoyés en deux positions  $u$  et  $v$  telles que  $u < v$ . Le TriLent est-il stable ?

**Question 3.** Soit  $i$  un entier tel que  $1 \leq i \leq n$ . Démontrer que, juste après avoir effectué  $i$  fois la boucle pour des lignes 3 et 4, l'élément  $A_i$  est l'élément maximal du tableau, et le sous-tableau formé des entrées  $A_1, A_2, \dots, A_i$  est trié dans l'ordre croissant.

**Question 4.** Démontrer que le TriLent permet effectivement de trier le tableau fourni en entrée.

**Question 5.** On suppose que les  $n$  éléments du tableau fourni en entrée sont deux à deux distincts. Donner le plus petit nombre possible d'échanges que le TriLent peut être amené à effectuer.

**Question 6.** Le tableau fourni en entrée est une permutation des entiers  $1, 2, \dots, n$  choisie uniformément au hasard. Démontrer que le nombre moyen d'échanges qu'effectuera le TriLent est égal à  $n(n-1)/4 + 2H_n - 2$ , où

$$H_n = \frac{1}{1} + \frac{1}{2} + \dots + \frac{1}{n}$$

est le  $n^{\text{ème}}$  nombre harmonique.

**Question 7.** Donner le nombre maximal d'échanges que l'algorithme de TriLent peut être amené à effectuer.



# Chapitre 115

## (X) Codage par couleur \*\*\* (X 24, corrigé — oral - 154 lignes)

\*\*\*

Algorithmique, Complexité, Graphes, Backtracking,  
sources : `xcodagecouleur.tex`

On considère des graphes orientés, supposés sans boucle. On cherchera dans ces graphes un chemin d'une longueur spécifique (sans imposer de point de départ et d'arrivée), où la longueur d'un chemin est le nombre de sommets qui apparaissent dans le chemin. Sauf mention explicite du contraire, on considère toujours des chemins simples, c'est-à-dire qui ne passent pas deux fois par le même sommet.

**Question 1.** Y a-t-il des graphes fortement connexes arbitrairement grands sans chemin de longueur 4 ?

**Question 2.** Proposer un algorithme naïf pour déterminer, étant donné un graphe  $G$  et un entier  $k \in \mathbb{N}$ , si  $G$  contient un chemin de longueur  $k$ . Quelle est la complexité de cet algorithme ?

**Question 3.** Dans cette question seulement, on s'intéresse à des chemins non nécessairement simples. Proposer un algorithme efficace pour déterminer, étant donné un graphe  $G$  et un entier  $k \in \mathbb{N}$ , si  $G$  contient un tel chemin de longueur  $k$ .

**Question 4.** Dans cette question seulement, on suppose que les sommets du graphe d'entrée  $G$  portent chacun une couleur parmi l'ensemble  $\{1, \dots, k\}$ , et on souhaite savoir si  $G$  admet un chemin multicolore de longueur  $k$ , c'est-à-dire un chemin  $v_1, \dots, v_k$  où les couleurs des sommets sont deux à deux distinctes. Proposer un algorithme pour résoudre ce problème, et en expliciter la complexité.

Pour améliorer le temps d'exécution de l'algorithme de la question 2, on va concevoir un algorithme probabiliste. Un tel algorithme a la possibilité de tirer au hasard certaines valeurs au cours de son exécution ; et on regarde, sur chaque entrée, quelle est la probabilité que l'algorithme réponde correctement, en fonction de ces tirages aléatoires. On veut résoudre le problème suivant : étant donné un graphe  $G$  et un entier  $k$ , on veut savoir si  $G$  a un chemin de longueur  $k$ . On considère d'abord l'algorithme (1) que voici. On répète  $M$  fois l'opération suivante (où  $M$  sera déterminé ensuite) : tirer  $k$  sommets au hasard et vérifier si ces sommets forment un chemin. On répond OUI si l'un de ces tirages est réussi et NON dans le cas contraire.

**Question 5.** On suppose que le graphe  $G$  a  $n$  sommets et contient précisément  $1 \leq c \leq n^k$  chemins de longueur  $k$ . Exprimer la probabilité que l'algorithme (1) réponde OUI, en fonction de  $M$ , de  $k$ , de  $n$ , et de  $c$ .

**Question 6.** Si on suppose que  $G$  n'a pas de chemin de longueur  $k$ , que répond l'algorithme (1) ?

**Question 7.** Pour  $M = n^k$ , montrer que l'algorithme répond correctement dans les deux cas avec probabilité au moins  $1/2$ .

**Question 8.** Quel est le temps d'exécution de l'algorithme (1) pour ce choix de  $M$  ? Commenter.

On considère à présent l'algorithme (2) dont le principe est le suivant. On répète  $M$  fois l'opération suivante (où  $M$  sera déterminé ensuite) : tirer un ordre total aléatoire  $v_1 < \dots < v_n$  sur les sommets de  $G$ , retirer les arêtes  $(u, v)$  où on a  $u > v$ , et vérifier si le graphe résultant  $G_{<}$  a un chemin de longueur  $k$  (d'une manière qui reste à déterminer).

**Question 9.** Quelle propriété ont les graphes  $G_{<}$  construits par cet algorithme ? Comment exploiter cette propriété pour trouver efficacement les chemins de longueur  $k$  ?

**Question 10.** Proposer un  $M$  qui garantisse que cet algorithme réponde correctement avec une probabilité  $\geq 1/2$ . Quelle complexité obtient-on ? Commenter.

**Question 11.** En utilisant les chemins multicolores, proposer un algorithme probabiliste qui répond correctement avec probabilité au moins  $1/2$  et s'exécute en  $O((2e)^k \times (n + m))$  sur un graphe à  $n$  sommets et  $m$  arêtes.

# Chapitre 116

## (X) Chaînes et anti-chaînes \*\*\* (X 25, non corrigé — oral - 44 lignes)

\*\*\*

Langages,

sources : `xchaines.tex`

On considère un ensemble  $E$  partiellement ordonné par  $\leq$ .

La relation d'ordre nous est donné sous la forme d'un ensemble de couples  $(a_i, b_i)$  vérifiant  $a_i \leq b_i$ .

### Définitions :

- **chaîne** : sous-ensemble totalement ordonné
- **anti-chaîne** : sous-ensemble dont deux élts ne sont pas comparables
  1. Donner un exemple de chaînes et d'anti-chaînes dans un ensemble de cardinal 5.
  2. Donner un algorithme déterminant si deux éléments sont comparables entre eux.
  3. Donner un algorithme déterminant la longueur d'une plus longue chaîne.
  4. Montrer que le cardinal minimal d'une partition en anti-chaînes est égal au cardinal d'une plus longue chaîne.
  5. On nous donne une liste d'intervalles  $[d_i, f_i[$ . Donner un algorithme construisant un sous-ensemble de cardinal maximal d'intervalles disjoints.
  6. Montrer que le cardinal minimal d'une partition en chaînes est égal au cardinal d'une plus longue anti-chaîne.
  7. Donner un algorithme qui détermine la longueur de la plus longue antichaîne.



## Chapitre 117

# (X) Diamètre dans un arbres \*\*\* (X 25, récupéré par élève, non corrigé — oral - 35 lignes)

\*\*\*

Algorithmique, Arbres, Complexité,  
sources : `xdiametre.tex`

**Définitions :** On définit les notions suivantes dans un graphe non orienté.

- On appelle **excentricité** d'un sommet la distance maximale de ce sommet à un autre.
- On appelle **diamètre** d'un graphe la longueur maximale d'un chemin.
- On appelle **rayon** d'un graphe l'excentricité minimale de ses sommets.

Dans la suite, on considère un graphe à  $n$  sommets.

1. Donner un algorithme permettant de calculer l'excentricité d'un sommet, et préciser sa complexité.
2. Montrer qu'il existe au plus 2 sommets d'excentricité minimale.
3. Donner un algorithme «efficace» permettant de trouver un sommet d'excentricité minimale.
4. Montrer que  $r = \left\lfloor \frac{D}{2} \right\rfloor$ , où  $D$  est le diamètre et  $r$  le rayon du graphe.
5. Donner un algorithme efficace pour calculer  $D$ .
6. On considère un ensemble d'ordinateurs connectés entre eux.

Comment trouver l'ordinateur à connecter à internet pour minimiser l'impact des disfonctionnements au niveau des connexions entre les ordinateurs ?



# Chapitre 118

## (X) Vis et écrous \*\*\* (X 25, récupéré par élève, non corrigé — oral - 81 lignes)

\*\*\*

Algorithmique, Tri, Complexité, Preuves, Algorithmes probabilistes,  
sources : xecroux.tex

### Vis et écrous

On a  $n$  vis et  $n$  écrous correspondants mais on fait tomber les boîtes qui sont désormais mélangées...

Avec  $v_1, \dots, v_m$  les diamètres de vis, at  $e_1, \dots, e_m$  ceux des écrous, les  $v_i$  sont deux à deux distincts, ainsi que les  $e_i$ , et on a  $\{v_1, \dots, v_n\} = \{e_1, \dots, e_n\}$ .

Pour chaque couple vis-écrou, on peut essayer de les combiner : soit la vis ne rentre pas, soit elle rentre parfaitement, soit elle est trop petite (i.e. on sait si  $v_i > e_j$ ,  $v_i = e_j$ , ou  $v_i < e_j$ ).

L'objectif est d'associer chaque vis à son écrou, efficacement.

1. Quel est le nombre minimum d'opérations pour résoudre le problème dans les cas  $n = 1$  et  $n = 2$ ?
2. Même question avec  $n = 3$ .

Dans la suite, on considèrera un tableau  $T$  contenant une permutation uniformément aléatoires des diamètres des écrous.

On donne les algorithmes suivants :

```
1 partitionner(tableau T, entier premier, entier dernier, entier pivot) :
2   échanger T[pivot] et T[dernier] // échange le pivot avec le dernier du ←
   tableau , le pivot devient le dernier du tableau
3   j := premier
4   pour i de premier à dernier - 1 // la boucle se termine quand i = (dernier él←
   ément du tableau).
5     si T[i] <= T[dernier] alors
6       échanger T[i] et T[j]
7       j := j + 1
8   échanger T[dernier] et T[j]
9   renvoyer j
10
11 tri_rapide(tableau T, entier premier, entier dernier)
12   si premier < dernier alors
13     pivot := un indice choisi aléatoirement uniformément dans ←
       {premier...dernier}
14     pivot := partitionner(T, premier, dernier, pivot)
15     tri_rapide(T, premier, pivot-1)
16     tri_rapide(T, pivot+1, dernier)
```

3. Quelle est la complexité du `tri_rapide` dans le pire cas?
4. Dans un appel à `tri_rapide`, montrer que la permutation des éléments d'un sous-tableau sur lequel on effectue un appel récursif est uniformément aléatoire.
5. Donner un algorithme qui mélange uniformément un tableau en temps linéaire (et le justifier).

6. On introduit les  $e_k$  : le  $k^{\text{e}}$  élément du tableau dans l'ordre croissant de ses éléments. Montrer que la probabilité de comparer  $e_k$  et  $e_l$  (où  $k < l$ ) dans le tri rapide est  $\frac{2}{l+1-k}$ .
7. Donner l'expression exacte (en fonction de  $H_n = \sum_{i \leq n} \frac{1}{i}$ ) du nombre moyen de comparaisons effectuées par le tri rapide.
8. Proposer une solution au problème des vis et des écrous de complexité moyenne  $O(n \log n)$ .

# Chapitre 119

## (Écrit) Logique trivaluée \* à \*\* (Écrit MP 25, utilisable oral, corrigé — oral - 275 lignes)

\* à \*\*

Logique propositionnelle,  
sources : ecrmplog.tex

On désire étendre la logique propositionnelle bivaluée classique (notée par la suite  $L_2$ ) de sorte à prendre en compte, en plus de  $V$  (vrai) et  $F$  (faux), une troisième valeur de vérité, notée  $I$  (pour indéterminé).

La **logique de Lukasiewicz**, notée  $L_3$ , est une telle logique tri-valuée dans laquelle le connecteur de négation ( $\neg$ ), le connecteur de conjonction ( $\wedge$ ), le connecteur de disjonction ( $\vee$ ) et le connecteur d'implication ( $\Rightarrow$ ) sont définis par les tables de vérités suivantes :

$x$	$\neg x$
$V$	$F$
$F$	$V$
$I$	$I$

$\neg x$

$x \backslash y$	$V$	$F$	$I$
$V$	$V$	$F$	$I$
$F$	$F$	$F$	$F$
$I$	$I$	$F$	$I$

$x \wedge y$

$x \backslash y$	$V$	$F$	$I$
$V$	$V$	$V$	$V$
$F$	$V$	$F$	$I$
$I$	$V$	$I$	$I$

$x \vee y$

$x \backslash y$	$V$	$F$	$I$
$V$	$V$	$F$	$I$
$F$	$V$	$V$	$V$
$I$	$V$	$I$	$V$

$x \Rightarrow y$

On dit que deux propositions sont **équivalentes** dans une logique donnée  $L$  si elles ont même table de vérité dans cette logique.

**Q1.** Indiquer une proposition  $P_1$  équivalente dans  $L_2$  à la proposition «  $x \vee y$  » et n'utilisant pas d'autres connecteurs que  $\neg$  et  $\wedge$ . Les propositions  $P_1$  et «  $x \vee y$  » sont-elles équivalentes dans  $L_3$  ?

**Q2.** Indiquer une proposition  $P_2$  équivalente dans  $L_2$  à la proposition «  $x \Rightarrow y$  » et n'utilisant pas d'autres connecteurs que  $\neg$  et  $\vee$ . Les propositions  $P_2$  et «  $x \Rightarrow y$  » sont-elles équivalentes dans  $L_3$  ?

**Q3.** Établir (en détaillant son obtention) la table de vérité dans  $L_3$  de la proposition  $P_3$  suivante :

$$((x \Rightarrow y) \wedge ((\neg x) \Rightarrow y)) \Rightarrow y.$$

Cette proposition est-elle une tautologie dans  $L_3$  (la définition d'une tautologie dans la logique  $L_3$  reste la même que dans la logique  $L_2$ ) ?

**Q4.** On appelle **littéral** une variable propositionnelle ou sa négation. Dans  $L_2$ , toute proposition logique admet une forme normale disjonctive (c'est-à-dire une disjonction de conjonctions de littéraux) qui lui est équivalente. Établir une proposition  $P_4$  sous forme normale disjonctive équivalente dans  $L_2$  à la proposition «  $(x \vee y) \Rightarrow z$  ». Les propositions  $P_4$  et «  $(x \vee y) \Rightarrow z$  » sont-elles équivalentes dans  $L_3$  ?

**Q5.** Peut-on faire un raisonnement par contraposition dans  $L_3$  (on justifiera la réponse) ?

**Q6.** On définit un ordre sur les valeurs  $F, I$  et  $V$  par :  $F < I < V$ . On considère une logique  $L$  tri-valuée pour laquelle simultanément :

- $L$  est une extension de  $L_2$  pour les quatre connecteurs usuels ( $\neg, \wedge, \vee, \Rightarrow$ ); autrement dit, les restrictions aux valeurs  $V$  et  $F$  des tables de vérité de  $L$  donnent les tables de vérité de  $L_2$ ;
- le connecteur  $\wedge$  est commutatif;
- la proposition «  $x \vee y$  » est équivalente à  $P_1$ ;
- la proposition «  $x \Rightarrow y$  » est équivalente à  $P_2$ ;
- pour toute valeur de  $y$ , la fonction  $x \mapsto x \wedge y$  croît au sens large avec  $x$ ;
- les propositions «  $\neg(\neg x)$  » et «  $x$  » sont équivalentes.

Combien y a-t-il de telles logiques (on justifiera la réponse) ?

THE END

# Index

Algorithmes d'approximation, 97, 175, 221  
Algorithmes gloutons, 49, 175  
Algorithmes probabilistes, 47, 225, 261  
Algorithmique, 39, 41, 43, 49, 71, 83, 87, 93, 103, 105, 107,  
109, 111, 123, 125, 127, 145, 151, 157, 207, 245, 247,  
253, 255, 259, 261  
Algorithmique des textes, 73, 201, 223  
Arbres, 27, 43, 55, 151, 155, 169, 181, 187, 259  
Automates, 31, 59, 89, 95, 101, 171, 189, 193, 197, 203, 211,  
213, 233, 249, 251

Backtracking, 41, 49, 63, 255  
Branch and bound, 49

C, 39, 45, 49, 61, 65, 77, 87, 93, 105, 107, 191, 235  
Circuits, 143  
Classification hiérarchique ascendante, 185  
Complexité, 27, 51, 55, 63, 75, 77, 97, 125, 127, 145, 151,  
169, 175, 179, 181, 187, 195, 205, 207, 221, 245, 253,  
255, 259, 261  
Concurrence, 33, 173, 199, 237

Décidabilité, 191, 197  
Dédution naturelle, 29, 139, 159, 183, 215

Grammaires, 53, 69, 155, 191, 209, 211  
Graphes, 45, 63, 71, 81, 85, 93, 99, 115, 121, 127, 129, 131,  
147, 161, 163, 177, 179, 195, 225, 235, 239, 241, 243,  
247, 255

Ia, 37, 91, 185, 217

Jeux, 67, 83, 99, 115, 233

Langages, 31, 59, 69, 73, 77, 89, 95, 101, 113, 117, 147, 171,  
189, 193, 197, 203, 209, 211, 213, 219, 227, 233, 249,  
251, 257  
Langages fonctionnels, 123  
Lemme de l'étoile, 31, 101, 193, 209, 227, 249, 251  
Logique propositionnelle, 47, 57, 75, 79, 119, 129, 135, 165,  
179, 191, 229, 263

Ocaml, 33, 41, 43, 59, 63, 73, 83, 85, 103, 109, 111, 113, 133,  
137, 141, 145, 149

Parcours de graphes, 179  
Preuves, 207, 253, 261  
Programmation dynamique, 61, 111, 169, 171, 195, 223, 231,  
243  
Programmation fonctionnelle, 133, 137  
Pseudo-code, 43, 123, 187, 195, 207, 239

Représentation des nombres, 65  
Réduction, 51, 75, 139, 147, 221, 231

Sql, 35, 167  
Structures de données, 103, 181, 187, 205, 245

Tri, 181, 187, 253, 261