

Table des matières

1	Index thématique	5
2	(CCINP) Arbres Binaires de Recherche * (CCINP 0, ex A, corrigé — oral - 74 lignes)	27
3	(CCINP) Dédution naturelle * (CCINP 0, ex A, corrigé — oral - 160 lignes)	29
4	(CCINP) ID3 * (CCINP 0, ex A, corrigé — oral - 175 lignes)	31
5	(CCINP) Langages réguliers * (CCINP 0, ex A, corrigé — oral - 120 lignes)	33
6	(CCINP) Mutex * (CCINP 0, ex A, corrigé — oral - 109 lignes)	35
7	(CCINP) SQL * (CCINP 0, ex A, corrigé — oral - 95 lignes)	37
8	(CCINP) Arbres * (CCINP 0, ex B, corrigé — oral - 247 lignes)	39
9	(CCINP) Graphes * (CCINP 0, ex B, corrigé — oral - 144 lignes)	43
10	(CCINP) SAT * (CCINP 0, ex B, corrigé — oral - 148 lignes)	47
11	(CCINP) Sac à dos * (CCINP 0, ex B, corrigé — oral - 178 lignes)	51
12	(CCINP) Tableaux * (CCINP 0, ex B, corrigé — oral - 151 lignes)	55
13	(CCINP) Tableaux autoréférents * (CCINP 0, ex B, corrigé — oral - 195 lignes)	59
14	(CCINP) Activation de processus * (CCINP 23, ex A, corrigé — oral - 153 lignes)	63
15	(CCINP) Formules propositionnelles croissantes * (CCINP 23, ex A, corrigé — oral - 142 lignes)	67
16	(CCINP) Grammaires algébriques * (CCINP 23, ex A, corrigé — oral - 123 lignes)	71
17	(CCINP) Minima locaux dans des arbres (Couverture dans des arbres) * (CCINP 23, ex A, corrigé — oral - 148 lignes)	73
18	(CCINP) Langages locaux * (CCINP 23, ex B, corrigé — oral - 215 lignes)	75
19	(CCINP) Programmation dynamique pour la récolte de fleurs * (CCINP 23, ex B, corrigé — oral - 204 lignes)	79
20	(CCINP) calculs avec les flottants * (CCINP 23, ex B, corrigé — oral - 206 lignes)	83
21	(CCINP) chemins simples sans issue * (CCINP 23, ex B, corrigé — oral - 199 lignes)	87
22	(CCINP) Grammaires pour des langages de programmation * (CCINP 24, ex A, corrigé, à débbugger — ds2526 - 90 lignes)	91
23	(CCINP) chomp * (CCINP 24, ex A, corrigé — oral - 228 lignes)	93
24	(CCINP) une relation d'équivalence * (CCINP 24, ex A, corrigé — oral - 138 lignes)	97
25	(CCINP) Fonctions pour la détection de motifs * (CCINP 24, ex B, corrigé — oral - 130 lignes)	99
26	(CCINP) HORNSAT * (CCINP 24, ex B, corrigé — oral - 186 lignes)	101
27	(CCINP) Mots de Dyck * (CCINP 24, ex B, corrigé — oral - 157 lignes)	105
28	(CCINP) Complétude de Nor et d'une variante Nor * (CCINP 25, ex 1, retour élève modifié, corrigé — oral - 150 lignes)	109

29 (CCINP) Graphes, A* * (CCINP 25, ex 1, retour élève très incomplet complété, corrigé — oral - 116 lignes)	111
30 (CCINP) Autour d'un jeu d'alignement * (CCINP 25, ex 2, retour élève complété, corrigé — oral - 201 lignes)	113
31 (CCINP) Graphes bipartis * (CCINP 25, ex 2, retour élève complété, corrigé — oral - 136 lignes)	117
32 (CCINP) Sous-mots * (CCINP 25, ex 2, retour élève très modifié, corrigé — oral - 155 lignes)	121
33 (CCINP) Mots sur un alphabet muni d'un ordre ** (CCINP 25, ex A, retour élève modifié, corrigé — oral - 171 lignes)	125
34 (CCINP) k plus proches voisins * (CCINP 25, ex A, retour élève modifié, figure refaite, corrigé — oral - 108 lignes)	127
35 (CCINP) UF et Kruskal * (CCINP 25, ex A, retour élève très incomplet reconstitué, corrigé — oral - 116 lignes)	129
36 (CCINP) Algorithmes du cours sur les automates, complexités ** (CCINP 25, ex A, retour élève, corrigé — oral - 232 lignes)	131
37 (CCINP) Bin packing * (CCINP 25, ex A, retour élève, corrigé — oral - 143 lignes)	133
38 (CCINP) Jeu sur un graphe * (CCINP 25, ex A, retour élève, corrigé — oral - 115 lignes)	137
39 (CCINP) Mots conjugués * (CCINP 25, ex A, retour élève, corrigé — oral - 110 lignes)	139
40 (CCINP) tas min * (CCINP 25, ex A, retour élève, rallongé, corrigé — oral - 92 lignes)	141
41 (CCINP) Dénombrement du nombre de façons d'obtenir un score avec des dés. * (CCINP 25, ex B, retour élève complété, corrigé — oral - 172 lignes)	143
42 (CCINP) Permutation de Langford * (CCINP 25, ex B, retour élève complété, corrigé — oral - 188 lignes)	147
43 (CCINP) Rendu de monnaie * (CCINP 25, ex B, retour élève complété, corrigé — oral - 126 lignes)	151
44 (CCINP) Dénombrement du nombre de façons d'obtenir un score avec des dés. * (CCINP 25, ex B, retour élève, corrigé — oral - 150 lignes)	155
45 (CCINP) Expressions régulières en OCaml * (CCINP 25, ex B, retour élève, corrigé — oral - 157 lignes)	159
46 (ENS) Jeu des jetons *** (ENS 23 MP, corrigé — oral - 153 lignes)	163
47 (ENS) Monoïdes et Langages *** (ENS 23 MP, corrigé — oral - 238 lignes)	167
48 (ENS) Élimination des coupures dans MLL et MALL **** (ENS 23, corrigé partiellement — oral - 385 lignes)	171
49 (ENS) Graphes parfaits *** (ENS 23, corrigé — oral - 251 lignes)	177
50 (ENS) Inférence de type *** (ENS 23, corrigé — oral - 280 lignes)	181
51 (ENS) Ordonnancement *** (ENS 23, corrigé — oral - 201 lignes)	185
52 (ENS) Permutations triables par pile *** (ENS 23, corrigé — oral - 184 lignes)	189
53 (ENS) Raisonnements ensemblistes *** (ENS 23, corrigé — oral - 234 lignes)	193
54 (ENS) Théorème des amis *** (ENS 23, corrigé — oral - 202 lignes)	197
55 (ENS) Structures pliables et traversables *** (ENS 23, partiellement corrigé — oral - 306 lignes)	203
56 (ENS) Élimination des coupures dans MLL et MALL *** (ENS 23, réécriture plus raisonnable par N.Carré — oral - 377 lignes)	207
57 (ENS) Composition Monadique *** (ENS 24, corrigé très partiellement — oral - 191 lignes)	211
58 (ENS) Déduduction de messages *** (ENS 24, corrigé — oral - 232 lignes)	215

59 (ENS) Filtrage par motif en OCaml *** (ENS 24, corrigé — oral - 323 lignes)	219
60 (ENS) Implémentation des circuits réversibles *** (ENS 24, corrigé — oral - 218 lignes)	223
61 (ENS) Résolution d'inéquations linéaires *** (ENS 24, corrigé — oral - 195 lignes)	227
62 (ENS) Mots partiels et Théorème de Dilworth *** (ENS 24, partiellement corrigé, niveau inapproprié, à déboguer — oral - 184 lignes)	231
63 (ENS) Terminaison de lambda ref *** (ENS 24, partiellement corrigé, niveau surréaliste — oral - 238 lignes)	235
64 (ENS) Calcul de l'arbre de dominance *** (ENS 25, non corrigé — oral - 83 lignes)	239
65 (ENS) Forme SSA, arbres de domination *** (ENS 25, non corrigé — oral - 120 lignes)	243
66 (ENS) Tri Externe Polyphasé *** (ENS 25, non corrigé — oral - 114 lignes)	245
67 (MT) Déduction Naturelle * (MT 0, ex 1, corrigé — oral - 200 lignes)	247
68 (MT) Relations entre les nombres de sommets et d'arêtes de graphes * (MT 0, ex 1, corrigé — oral - 112 lignes)	251
69 (MT) bdd pour livraison * (MT 0, ex 1, corrigé — oral - 78 lignes)	253
70 (MT) couplages * (MT 0, ex 1, corrigé — oral - 64 lignes)	255
71 (MT) logique * (MT 0, ex 1, corrigé — oral - 100 lignes)	257
72 (MT) Arbres Binaires de Recherche * (MT 0, ex 2, corrigé — oral - 161 lignes)	259
73 (MT) entrelacements de mots * (MT 0, ex 2, corrigé — oral - 109 lignes)	263
74 (MT) graphes * (MT 0, ex 2, corrigé — oral - 121 lignes)	265
75 (MT) montée et descente d'un bus * (MT 0, ex 2, corrigé — oral - 104 lignes)	267
76 (MT) subset-sum * (MT 0, ex 2, corrigé — oral - 143 lignes)	271
77 (MT) Classification hiérarchique ascendante * (MT 24, ex 1, corrigé, le graphique n'est pas l'original — oral - 44 lignes)	275
78 (MT) Graphes bipartis * (MT 24, ex 1, corrigé — oral - 92 lignes)	277
79 (MT) Tas binaires * (MT 24, ex 1, corrigé — oral - 129 lignes)	279
80 (MT) Tas binaires * (MT 24, ex 1, corrigé — oral - 137 lignes)	281
81 (MT) déduction naturelle * (MT 24, ex 1, corrigé — oral - 141 lignes)	285
82 (MT) Automates de Büchi *** (MT 24, ex 2, (modifié) corrigé — oral - 117 lignes)	287
83 (MT) Automates, langages à saut * (MT 24, ex 2, corrigé, complété au jugé pour les dernières questions — oral - 91 lignes)	289
84 (MT) plus courts chemins dans des graphes * (MT 24, ex 2, corrigé, retour d'oral — oral - 120 lignes)	291
85 (MT) Décidabilité de programmes engendrés par une grammaire * (MT 24, ex 2, corrigé — oral - 127 lignes)	293
86 (MT) Automates et décidabilité ** (MT 25, ex 1, retour élève complété, corrigé — oral - 87 lignes)	295
87 (MT) Récompenses pour un site web * (MT 25, ex 1, retour élève remanié, corrigé (à préciser/compléter) — oral - 68 lignes)	297
88 (MT) Algorithmique des textes * (MT 25, ex 1, retour élève remanié, corrigé — oral - 72 lignes)	299
89 (MT) Langages ** à *** (MT 25, ex 1, retour élève remanié, corrigé — oral - 80 lignes)	301
90 (MT) Piles et files * (MT 25, ex 1, retour élève retravaillé, corrigé — oral - 67 lignes)	303

91 (MT) Points fixes dans des tableaux * (MT 25, ex 1, retour élève retravaillé, corrigé — oral - 79 lignes)	305
92 (MT) Langages des préfixes * (MT 25, ex 1, retour élève à modifier, corrigé — oral - 56 lignes)	307
93 (MT) Substitution régulière ** à **** (MT 25, ex 1, retour élève à reprendre, corrigé — oral - 92 lignes)	309
94 (MT) Dédution naturelle * (MT 25, ex 1, retour élève, corrigé — oral - 97 lignes)	311
95 (MT) Langages et facteurs * (MT 25, ex 1, retour élève, corrigé — oral - 150 lignes)	313
96 (MT) k plus proches voisins * (MT 25, ex 1, retour élève, corrigé — oral - 46 lignes)	315
97 (MT) lemme d'Arden ** (MT 25, ex 1, retour élève, corrigé — oral - 94 lignes)	317
98 (MT) * (MT 25, ex 2, retour élève remanié, corrigé — oral - 117 lignes)	319
99 (MT) Graphes et couleurs * à *** (MT 25, ex 2, retour élève remanié, corrigé — oral - 128 lignes)	321
100 (MT) Langages rationnels et ultime périodicité *** (MT 25, ex 2, retour élève remanié, corrigé — oral - 82 lignes)	323
101 (MT) Représentation fusionnée de deux mots ** (MT 25, ex 2, retour élève remanié, corrigé — oral - 116 lignes)	325
102 (MT) Théorème de compacité ** (MT 25, ex 2, retour élève remanié, corrigé — oral - 101 lignes)	327
103 (MT) Programmation dynamique *** (MT 25, ex 2, retour élève retravaillé, corrigé — oral - 131 lignes)	329
104 (MT) Automates alternants *** (MT 25, ex 2, retour élève très buggué largement retravaillé, corrigé — oral - 294 lignes)	331
105 (MT) Graphes : implémentation en C * (MT 25, ex 2, retour élève à compléter/vérifier++, corrigé — oral - 98 lignes)	335
106 (MT) League et Overwatch ** (MT 25, ex 2, retour élève à reprendre, non corrigé — oral - 91 lignes)	337
107 (MT) coloriage de graphe * (MT 25, ex 2, retour élève, 3/4 questions manquantes, corrigé — oral - 94 lignes)	339
108 (MT) CNS pour cycles eulériens * (MT 25, ex 2, retour élève, corrigé — oral - 71 lignes)	341
109 (MT) Cycles et Chemins simples ** (MT 25, ex 2, retour élève, corrigé — oral - 64 lignes)	343
110 (X) Algorithme Union-Find *** (X 23, corrigé — oral - 151 lignes)	345
111 (X) Bob l'écureuil *** (X 23, corrigé — oral - 141 lignes)	349
112 (X) Langages rationnels et lemme de l'étoile *** (X 23, corrigé — oral - 118 lignes)	353
113 (X) Rationnalité de langages *** (X 24, corrigé, complété au jugé pour les trois dernières questions — oral - 213 lignes)	357
114 (X) Algorithme du tri lent *** (X 24, corrigé — oral - 147 lignes)	361
115 (X) Codage par couleur *** (X 24, corrigé — oral - 154 lignes)	367
116 (X) Chaînes et anti-chaînes *** (X 25, non corrigé — oral - 44 lignes)	371
117 (X) Diamètre dans un arbres *** (X 25, récupéré par élève, non corrigé — oral - 35 lignes)	373
118 (X) Vis et écrous *** (X 25, récupéré par élève, non corrigé — oral - 81 lignes)	375
119 (Écrit) Logique trivaluée * à ** (Écrit MP 25, utilisable oral, corrigé — oral - 275 lignes)	377

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
- 75-subset-sum (MT 0, ex 2, **corrigé** — 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

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

Algorithmes probabilistes

- 9-SAT (CCINP 0, ex B, **corrigé** — oral - 148 lignes) *
logique propositionnelle, algorithmes probabilistes
ccinpsat.tex
VOIR
- 98-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-Arbres (CCINP 0, ex B, **corrigé** — oral - 247 lignes) *
arbres, ocaml, algorithmique, pseudo-code
`ccinparbre2.tex`
VOIR
- 10-Sac à dos (CCINP 0, ex B, **corrigé** — oral - 178 lignes) *
algorithmique, c, algorithmes gloutons, backtracking, branch and bound
`ccinpsacados.tex`
VOIR
- 11-Tableaux (CCINP 0, ex B, **corrigé** — oral - 151 lignes) *
algorithmique, c
`ccinptableau.tex`
VOIR
- 12-Tableaux autoréférents (CCINP 0, ex B, **corrigé** — oral - 195 lignes) *
algorithmique, backtracking, ocaml
`ccinpautoreferent.tex`
VOIR
- 23-une relation d'équivalence (CCINP 24, ex A, **corrigé** — oral - 138 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 - 172 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 - 280 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
mmtab1.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 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

Algorithmique des textes

- 24-Fonctions pour la détection de motifs (CCINP 24, ex B, **corrigé** — oral - 130 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
- 100-Représentation fusionnée de deux mots (MT 25, ex 2, retour élève remanié, **corrigé** — oral - 116 lignes) **
algorithmique des textes, programmation dynamique
mttxt1.tex
VOIR

Arbres

- 1-Arbres Binaires de Recherche (CCINP 0, ex A, **corrigé** — oral - 74 lignes) *
arbres, complexité
arbreCCINPA2.tex
VOIR
- 7-Arbres (CCINP 0, ex B, **corrigé** — oral - 247 lignes) *
arbres, ocaml, algorithmique, pseudo-code
ccinparbre2.tex
VOIR
- 16-Minima locaux dans des arbres (Couverture dans des arbres) (CCINP 23, ex A, **corrigé** — 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** — oral - 161 lignes) *
arbres, complexité, programmation dynamique
mtarbres1.tex
VOIR
- 78-Tas binaires (MT 24, ex 1, **corrigé** — oral - 129 lignes) *
arbres, tri, structures de données, complexité
mttas.tex
VOIR
- 79-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

- 4-Langages réguliers (CCINP 0, ex A, **corrigé** — oral - 120 lignes) *
langages, Automates, lemme de l'étoile
`langccinp3.tex`
VOIR
- 17-Langages locaux (CCINP 23, ex B, **corrigé** — 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** — oral - 109 lignes) *
langages, automates, programmation dynamique
`mtlang.tex`
VOIR
- 81-Automates de Büchi (MT 24, ex 2, (modifié) **corrigé** — oral - 117 lignes) ***
langages, automates
`mtbuchi.tex`
VOIR
- 82-Automates, langages à saut (MT 24, ex 2, **corrigé**, complété au jugé pour les dernières questions — oral - 91 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
- 94-Langages et facteurs (MT 25, ex 1, retour élève, **corrigé** — 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

- 10-Sac à dos (CCINP 0, ex B, **corrigé** — oral - 178 lignes) *
algorithmique, c, algorithmes gloutons, backtracking, branch and bound
ccinpsacados.tex
VOIR
- 12-Tableaux autoréférents (CCINP 0, ex B, **corrigé** — oral - 195 lignes) *
algorithmique, backtracking, ocaml
ccinpautoreferent.tex
VOIR
- 20-chemins simples sans issue (CCINP 23, ex B, **corrigé** — 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

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

C

- 8-Graphes (CCINP 0, ex B, **corrigé** — oral - 144 lignes) *
graphes, c
ccinpgraphe.tex
VOIR
- 10-Sac à dos (CCINP 0, ex B, **corrigé** — oral - 178 lignes) *
algorithmique, c, algorithmes gloutons, backtracking, branch and bound
ccinpsacados.tex
VOIR
- 11-Tableaux (CCINP 0, ex B, **corrigé** — oral - 151 lignes) *
algorithmique, c

ccinptableau.tex

VOIR

- 18-Programmation dynamique pour la récolte de fleurs (CCINP 23, ex B, **corrigé** — oral - 204 lignes) *
programmation dynamique, c
ccinprecoltefleurs.tex
VOIR

- 19-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
ccinpyck.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 - 172 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

- 84-Décidabilité de programmes engendrés par une grammaire (MT 24, ex 2, **corrigé** — oral - 127 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

- 76-Classification hiérarchique ascendante (MT 24, ex 1, **corrigé**, le graphique n'est pas l'original — oral - 44 lignes) *
ia, classification hiérarchique ascendante
mtia.tex
VOIR

Complexité

- 1-Arbres Binaires de Recherche (CCINP 0, ex A, **corrigé** — oral - 74 lignes) *
arbres, complexité
`arbreCCINPA2.tex`
VOIR
- 13-Activation de processus (CCINP 23, ex A, **corrigé** — oral - 153 lignes) *
complexité, réduction
`procesCCINP23.tex`
VOIR
- 16-Minima locaux dans des arbres (Couverture dans des arbres) (CCINP 23, ex A, **corrigé** — oral - 148 lignes) *
arbres, complexité
`arbreCCINPA.tex`
VOIR
- 20-chemins simples sans issue (CCINP 23, ex B, **corrigé** — oral - 199 lignes) *
graphes, ocaml, complexité, backtracking
`ccinpchemins_simples.tex`
VOIR
- 25-HORNSAT (CCINP 24, ex B, **corrigé** — 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
`ccinpdyck.tex`
VOIR
- 36-Bin packing (CCINP 25, ex A, retour élève, **corrigé** — oral - 143 lignes) *
algorithmes d'approximation, complexité
`ccinpbinpack.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** — oral - 161 lignes) *
arbres, complexité, programmation dynamique
`mtarbres1.tex`
VOIR
- 75-subset-sum (MT 0, ex 2, **corrigé** — oral - 143 lignes) *

complexité, algorithmes d'approximation, algorithmes gloutons

mtsubsum.tex

VOIR

- 77-Graphes bipartis (MT 24, ex 1, **corrigé** — oral - 92 lignes) *
graphes, logique propositionnelle, parcours de graphes, complexité
mtgraph5.tex
VOIR
- 78-Tas binaires (MT 24, ex 1, **corrigé** — oral - 129 lignes) *
arbres, tri, structures de données, complexité
mttas.tex
VOIR
- 79-Tas binaires (MT 24, ex 1, **corrigé** — oral - 137 lignes) *
arbres, tri, structures de données, complexité, pseudo-code
mttas2.tex
VOIR
- 83-plus courts chemins dans des graphes (MT 24, ex 2, **corrigé**, retour d'oral — oral - 120 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-Algorithme Union-Find (X 23, **corrigé** — oral - 151 lignes) ***
algorithmique, complexité, structures de données
xunionfind.tex
VOIR
- 113-Algorithme 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

Concurrence

- 5-Mutex (CCINP 0, ex A, **corrigé** — oral - 109 lignes) *
concurrency, ocaml
`concccinp1.tex`
VOIR
- 74-montée et descente d'un bus (MT 0, ex 2, **corrigé** — 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

- 84-Décidabilité de programmes engendrés par une grammaire (MT 24, ex 2, **corrigé** — oral - 127 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** — 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** — oral - 200 lignes) *
deduction naturelle
`mtlog1.tex`
VOIR
- 80-déduction naturelle (MT 24, ex 1, **corrigé** — oral - 141 lignes) *
deduction naturelle
`mtlog3.tex`
VOIR
- 93-Déduction naturelle (MT 25, ex 1, retour élève, **corrigé** — oral - 97 lignes) *
deduction naturelle
`mtlog5.tex`
VOIR

Grammaires

- 15-Grammaires algébriques (CCINP 23, ex A, **corrigé** — oral - 123 lignes) *
grammaires
`grammccinp1.tex`
VOIR
- 21-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
- 84-Décidabilité de programmes engendrés par une grammaire (MT 24, ex 2, **corrigé** — oral - 127 lignes) *
grammaires, décidabilité, c, logique propositionnelle
`mthaltbool.tex`
VOIR
- 91-Langages des préfixes (MT 25, ex 1, retour élève à modifier, **corrigé** — oral - 56 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

- 8-Graphes (CCINP 0, ex B, **corrigé** — oral - 144 lignes) *
graphes, c
`ccinpgraphe.tex`
VOIR
- 20-chemins simples sans issue (CCINP 23, ex B, **corrigé** — oral - 199 lignes) *
graphes, ocaml, complexité, backtracking
`ccinpchemins_simples.tex`
VOIR
- 23-une relation d'équivalence (CCINP 24, ex A, **corrigé** — oral - 138 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 - 115 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 débbuguer — oral - 184 lignes) ***
graphes, réduction, langages
ensmotspartiels.tex
VOIR
- 67-Relations entre les nombres de sommets et d'arêtes de grphes (MT 0, ex 1, **corrigé** — oral - 112 lignes) *
graphes
mtgraph1.tex
VOIR
- 69-couplages (MT 0, ex 1, **corrigé** — oral - 64 lignes) *
graphes
mtgraph3.tex
VOIR
- 73-graphes (MT 0, ex 2, **corrigé** — oral - 121 lignes) *
graphes
mtgraph2.tex
VOIR
- 77-Graphes bipartis (MT 24, ex 1, **corrigé** — oral - 92 lignes) *
graphes, logique propositionnelle, parcours de graphes, complexité
mtgraph5.tex
VOIR
- 83-plus courts chemins dans des graphes (MT 24, ex 2, **corrigé**, retour d'oral — oral - 120 lignes) *
graphes, complexité, programmation dynamique, pseudo-code
mtgraph4.tex
VOIR
- 98-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 - 94 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

- 3-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
- 76-Classification hiérarchique ascendante (MT 24, ex 1, **corrigé**, le graphique n'est pas l'original — 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

- 22-chomp (CCINP 24, ex A, **corrigé** — 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 - 115 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

- 4-Langages réguliers (CCINP 0, ex A, **corrigé** — oral - 120 lignes) *
langages, Automates, lemme de l'étoile
langccinp3.tex
VOIR
- 17-Langages locaux (CCINP 23, ex B, **corrigé** — oral - 215 lignes) *
langages, ocaml, automates
langloccinp1.tex
VOIR
- 21-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** — oral - 130 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** — oral - 109 lignes) *
langages, automates, programmation dynamique
 mtlang.tex
VOIR

- 81-Automates de Büchi (MT 24, ex 2, (modifié) **corrigé** — oral - 117 lignes) ***
langages, automates
 mtbuchi.tex
VOIR

- 82-Automates, langages à saut (MT 24, ex 2, **corrigé**, complété au jugé pour les dernières questions — oral - 91 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 - 56 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

- 94-Langages et facteurs (MT 25, ex 1, retour élève, **corrigé** — 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

- 99-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 - 280 lignes) ***

algorithmique, langages fonctionnels, pseudo-code

ensinferencetypes.tex

VOIR

Lemme de l'étoile

- 4-Langages réguliers (CCINP 0, ex A, **corrigé** — 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

- 82-Automates, langages à saut (MT 24, ex 2, **corrigé**, complété au jugé pour les dernières questions — oral - 91 lignes) *

langages, automates, lemme de l'étoile

mtlang2.tex

VOIR

- 91-Langages des préfixes (MT 25, ex 1, retour élève à modifier, **corrigé** — oral - 56 lignes) *

langages, grammaires, lemme de l'étoile

mtlang6.tex

VOIR

- 99-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

- 9-SAT (CCINP 0, ex B, **corrigé** — oral - 148 lignes) *
logique propositionnelle, algorithmes probabilistes
ccinpsat.tex
VOIR

- 14-Formules propositionnelles croissantes (CCINP 23, ex A, **corrigé** — oral - 142 lignes) *
logique propositionnelle
logccinp1.tex
VOIR

- 25-HORNSAT (CCINP 24, ex B, **corrigé** — 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

- 70-logique (MT 0, ex 1, **corrigé** — oral - 100 lignes) *
logique propositionnelle
mtlog2.tex
VOIR

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

- 84-Décidabilité de programmes engendrés par une grammaire (MT 24, ex 2, **corrigé** — oral - 127 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

- 5-Mutex (CCINP 0, ex A, **corrigé** — oral - 109 lignes) *
concurrency, ocaml
`concccinp1.tex`
VOIR
- 7-Arbres (CCINP 0, ex B, **corrigé** — oral - 247 lignes) *
arbres, ocaml, algorithmique, pseudo-code
`ccinparbre2.tex`
VOIR
- 12-Tableaux autoréférents (CCINP 0, ex B, **corrigé** — oral - 195 lignes) *
algorithmique, backtracking, ocaml
`ccinpautoreferent.tex`
VOIR
- 17-Langages locaux (CCINP 23, ex B, **corrigé** — oral - 215 lignes) *
langages, ocaml, automates
`langlocccinp1.tex`
VOIR
- 20-chemins simples sans issue (CCINP 23, ex B, **corrigé** — oral - 199 lignes) *
graphes, ocaml, complexité, backtracking
`ccinpchemins_simples.tex`
VOIR
- 24-Fonctions pour la détection de motifs (CCINP 24, ex B, **corrigé** — oral - 130 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 - 306 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

- 77-Graphes bipartis (MT 24, ex 1, **corrigé** — 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-Algorithmme 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** — oral - 204 lignes) *
programmation dynamique, c
`ccinprecoltefleurs.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
- 71-Arbres Binaires de Recherche (MT 0, ex 2, **corrigé** — oral - 161 lignes) *

arbres, complexité, programmation dynamique

mtarbres1.tex

VOIR

- 72-entrelacements de mots (MT 0, ex 2, **corrigé** — oral - 109 lignes) *
langages, automates, programmation dynamique
mtlang.tex
VOIR
- 83-plus courts chemins dans des graphes (MT 24, ex 2, **corrigé**, retour d'oral — oral - 120 lignes) *
graphes, complexité, programmation dynamique, pseudo-code
mtgraph4.tex
VOIR
- 100-Représentation fusionnée de deux mots (MT 25, ex 2, retour élève remanié, **corrigé** — 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 - 306 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

- 7-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 - 280 lignes) ***
algorithmique, langages fonctionnels, pseudo-code
ensinferencetypes.tex
VOIR
- 79-Tas binaires (MT 24, ex 1, **corrigé** — oral - 137 lignes) *
arbres, tri, structures de données, complexité, pseudo-code
mttas2.tex
VOIR
- 83-plus courts chemins dans des graphes (MT 24, ex 2, **corrigé**, retour d'oral — oral - 120 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 - 94 lignes) *
graphes, pseudo-code
mtgraph6.tex
VOIR

Représentation des nombres

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

Réduction

- 13-Activation de processus (CCINP 23, ex A, **corrigé** — oral - 153 lignes) *
complexité, réduction
procesCCINP23.tex
VOIR
- 25-HORNSAT (CCINP 24, ex B, **corrigé** — 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

- 6-SQL (CCINP 0, ex A, **corrigé** — oral - 95 lignes) *
sql
sqlccinp1.tex
VOIR
- 68-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
- 78-Tas binaires (MT 24, ex 1, **corrigé** — oral - 129 lignes) *
arbres, tri, structures de données, complexité
 mttas.tex
VOIR
- 79-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

- 78-Tas binaires (MT 24, ex 1, **corrigé** — oral - 129 lignes) *
arbres, tri, structures de données, complexité
 mttas.tex
VOIR
- 79-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 — oral - 74 lignes)

*

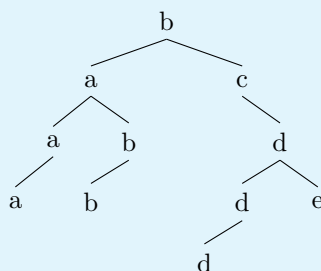
Arbres, Complexité,
sources : arbreCCINPA2.tex

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 et une à une 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*, *b*, *c* et *d*. Quelle est sa complexité ?

correction

1. Un arbre binaire de recherche est un arbre binaire dont les clés sont à valeurs dans un ensemble totalement ordonné. De plus, pour chaque noeud de cet arbre, son étiquette x est strictement supérieure aux étiquettes dans son fils gauche et strictement inférieure aux étiquettes dans son fils droit (des variantes autorisent une inégalité large d'un des deux côtés).
2. On décide de mettre les cas d'égalité à gauche (par exemple) :



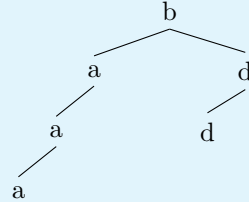
3. On procède par induction. On note $P(a)$ le parcours infixe de l'ABR a . Le parcours infixe d'un ABR vide est bien sûr trié. De plus, si a est un ABR de racine x et de fils gauche (respectivement droit) g (respectivement d) alors $P(a) = P(g), x, P(d)$. Par hypothèse inductive, $P(g)$ et $P(d)$ sont triés dans l'ordre croissant. Par définition d'un ABR, x est plus grand que les éléments apparaissant dans $P(g)$ et plus petit que ceux dans $P(d)$: $P(a)$ est donc trié dans l'ordre croissant.
4. On procède récursivement :
 - Le nombre d'occurrences d'une lettre u dans un arbre vide vaut 0.
 - Si l'ABR n'est pas vide, on compare u à sa racine x . Si $u < x$, on compte le nombre d'occurrences de u à gauche, si $u > x$, on compte le nombre d'occurrences de u à droite et si $u = x$, on compte le nombre d'occurrences de u à gauche et on lui ajoute 1.

En fait, dès qu'on a trouvé une occurrence de u , les autres sont forcément à gauche et on peut ainsi compter les occurrences de u en ne faisant le parcours que d'une branche de l'arbre. D'où une complexité en $O(h)$ où

h est la hauteur de l'arbre.

5. On commence par chercher une occurrence de l'élément s à supprimer en descendant soit à gauche soit à droite selon comment s se compare à la racine courante. Une fois s trouvé :
- S'il n'a pas de fils, on peut le supprimer directement.
 - S'il en a un, on remplace l'arbre enraciné en s par ce fils.
 - S'il en a deux, on récupère le maximum m du fils gauche, on écrase s avec m puis on supprime m . Cet appel récursif aboutit à l'un des deux cas précédents car par définition le maximum d'un ABR se trouve en descendant systématiquement à droite dont m n'aura pas de fils droit.

Sur l'arbre de la question 2 on obtient après suppressions :



La complexité d'une suppression est linéaire en la taille de l'arbre. (On pourrait obtenir une complexité logarithmique si l'arbre était équilibré.)

Chapitre 3

(CCINP) Dédution naturelle * (CCINP 0, ex A, corrigé — 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 Γ 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.

correction

Corrigé de M.Péchaud

L'arbre de preuve A_1 suivant convient :

$$\frac{\frac{\frac{\frac{}{A, \neg A \vdash A} \text{AX}}{A, \neg A \vdash \perp} \neg_e}{\neg A \vdash A \rightarrow \perp} \rightarrow_i}{\vdash \neg A \rightarrow (A \rightarrow \perp)} \rightarrow_i$$

2. Montrer que le séquent $\vdash (A \rightarrow \perp) \rightarrow \neg A$ est dérivable, en explicitant un arbre de preuve.

correction

L'arbre de preuve A_2 suivant convient :

$$\frac{\frac{\frac{\frac{}{A, A \rightarrow \perp \vdash A} \text{AX}}{A, A \rightarrow \perp \vdash \perp} \neg_e}{A, A \rightarrow \perp \vdash \neg A} \neg_i}{\vdash (A \rightarrow \perp) \rightarrow \neg A} \rightarrow_i$$

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.

correction

Dans cet ordre : l'introduction, l'élimination à gauche et celle à droite :

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge_i \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge_{eg} \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge_{ed}$$

Le séquent demandé se dérive immédiatement des arbres A_1 et A_2 et de l'application de \wedge_i .

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.

correction

On peut par exemple dresser une table de vérité :

A	B	$A \rightarrow B$	$(A \rightarrow B) \rightarrow A$	P
0	0	1	0	1
0	1	1	0	1
1	0	0	1	1
1	1	1	1	1

On constate que P est effectivement 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 $\Gamma = \{(A \rightarrow B) \rightarrow A, \neg A\}$:

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

correction

On complète la branche de gauche avec l'arbre suivant (l'idée étant de prouver $A \rightarrow B$ à partir de Γ . En effet, il ne suffit plus ensuite que d'utiliser $(A \rightarrow B) \rightarrow A$ pour obtenir A) :

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

Chapitre 4

(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 .

```

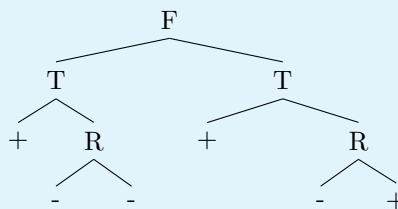
1 Entrées :  $S_p$  sous-ensemble des exemples du noeud parent,  $S$  sous-ensemble des ←
   exemples à considérer,  $D$  sous-ensemble des attributs à considérer}
2 Sortie : Un arbre de décision.
3
4 Fonction ID3( $(S_p, S, D)$ ) ;
5   Si l'ensemble des exemples  $S$  est vide
6     Renvoyer ...
7
8   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  $\text{ID3} \leftarrow$ 
      ( $S, S_{A=NO}, D \setminus \{A\}$ ) et de sous-arbre droit  $\text{ID3}(S, S_{A=YES}, D \setminus \{A\})$ 

```

4. Indiquer comment compléter l'algorithme 1.

correction

1. Dans un problème d'apprentissage supervisé, on dispose d'objets ayant des attributs et une étiquette. L'objectif est de construire une fonction (un modèle) à partir d'exemples connus qui à partir des attributs d'un objet potentiellement inconnu indique son étiquette.
2. On obtient l'arbre suivant en prenant comme conventions qu'une feuille est étiquetée par la classe majoritaire des exemples qu'elle représente et que dans le cas où une branche ne recoupe plus aucun exemple, on crée une feuille dont la classe est la classe majoritaire de son noeud père :



Par exemple, sur la branche gauche, droite, droite, qui correspond à $F = \text{NO}$, $T = \text{YES}$, $R = \text{YES}$, les exemples correspondants sont 9 (+), 12 (-) et 13 (-) donc on étiquette la feuille par -. Sur la branche droite, gauche, gauche, qui correspond à $F = \text{YES}$, $T = \text{NO}$, $R = \text{NO}$, il n'y a aucun exemple associé : on crée une feuille dont l'étiquette est celle majoritaire parmi 4, 7 et 8 qui sont les exemples pour lesquels $F = \text{YES}$ et $T = \text{NO}$.

Remarquons que cet arbre est inutilement complexe : certains branchements semblent inutiles.

3. On devrait trouver 0.47 (arrondi au centième), cf le fichier ID3.ml pour les calculs.
4. Indirectement on l'a déjà fait en question 2 :
 - Si l'ensemble des exemples est vide, on crée une feuille avec une étiquette par défaut ; par exemple, on peut choisir l'étiquette majoritaire des exemples dans le noeud parent.
 - Si l'ensemble des attributs est vide, on ne peut plus séparer les exemples : il faut donc créer une feuille qu'on étiquette par la classe majoritaire des exemples encore présents.
 - Si tous les exemples sont dans la même classe, on crée une feuille étiquetée par cette classe.

Chapitre 5

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

*

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

1. Rappeler la définition d'un langage régulier.

correction

Un langage est régulier s'il est dénoté par une expression régulière sur un certain alphabet Σ . Les expressions régulières sur Σ sont définies par induction par :

- \emptyset , ε et a pour $a \in \Sigma$ sont des expressions régulières.
- Si f et g sont des expressions régulières, $(f + g)$, fg et f^* aussi.

2. Les langages suivants sont-ils réguliers ? Justifier.

(a) $L_1 = \{a^n ba^m \mid n, m \in \mathbb{N}\}$

correction

L_1 est régulier car il est dénoté par a^*ba^* .

(b) $L_2 = \{a^n ba^m \mid n, m \in \mathbb{N}, n \leq m\}$

correction

L_2 n'est pas régulier. Par l'absurde, s'il l'était, le lemme de l'étoile fournirait $N \in \mathbb{N}$ tel que le mot $a^N ba^N \in L_2$ se factoriserait en $a^{n_1} a^{n_2} a^{N-n_1-n_2} ba^N$ avec $n_1 \in \mathbb{N}$, $n_2 \in \mathbb{N}^*$ et de sorte à ce que pour tout $k \in \mathbb{N}$ on ait $a^{N+(k-1)n_2} ba^N \in L_2$. Pour $k = 2$, on aurait $a^{N+n_2} ba^N \in L_2$ ce qui n'est pas le cas car $N + n_2 > N$ en vertu du fait que n_2 est non nul.

(c) $L_3 = \{a^n ba^m \mid n, m \in \mathbb{N}, n > m\}$

correction

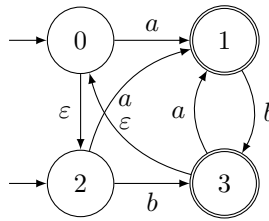
L_3 n'est pas régulier. S'il l'était, le langage $L_3 \cap a^*ba^*$ le serait aussi par stabilité des langages réguliers par le complémentaire et l'intersection ; or ce langage est L_2 qui n'est pas régulier.

(d) $L_4 = \{a^n ba^m \mid n, m \in \mathbb{N}, n + m \equiv 0 \pmod{2}\}$

correction

L_4 est le langage des mots de la forme $a^n ba^m$ avec n, m tous les deux pairs ou n, m tous les deux impairs. Il est donc dénoté par $(a^2)^*b(a^2)^* + a(a^2)^*ba(a^2)^*$ et est donc régulier.

3. On considère l'automate non déterministe suivant :



(a) Déterminiser cet automate.

correction

On applique l'algorithme de déterminisation habituel, en prenant en compte la présence des ϵ -transitions. On peut l'écrire directement ou passer par la table :

	a	b
$\rightarrow \{0, 2\}$	$\{1\}$	$\{0, 2, 3\}$
$\{1\}$	\emptyset	$\{0, 2, 3\}$
$\{0, 2, 3\}$	$\{1\}$	$\{0, 2, 3\}$

On trouve l'automate des parties suivant (si on ne complète pas) :

(b) Construire une expression régulière dénotant le langage reconnu par cet automate.

correction

En appliquant l'algorithme d'élimination des états en commençant par éliminer (0,2,3) puis 1 (on garde l'état initial), on obtient :

$(a + ba)(b^+a)^*b^*$.

(c) Décrire simplement avec des mots le langage reconnu par cet automate.

correction

Les mots de ce langage sont exactement les mots ne contenant pas 2 a d'affilée.

Chapitre 6

(CCINP) Mutex * (CCINP 0, ex A, corrigé — 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.

correction

1. A priori, toute valeur entre 1 et 100 (inclus de deux côtés) est possible.
2. Pour obtenir 100 de manière certaine, on protège la section critique comme suit :

```
let fi = Mutex.lock(m) ; compteur := ! compteur + 1
```

3. Avec `f_a` on peut avoir famine avec la trace d'exécution suivante :
 - Le fil 0 calcul `autre` puis indique que `veut_entrer.(0)` vaut `true`.
 - Le fil 1 calcul `autre` puis indique que `veut_entrer.(1)` vaut `true`.
 - Les deux fils sont alors bloqués dans la boucle `while`.
 Avec `f_b` il n'y a pas exclusion mutuelle ce qui s'observe via la trace suivante :
 - Le fil 0 calcule `autre`, indique qu'il veut entrer, fixe `tour` à 0 puis entre en section critique (ce qui est possible puisque ce n'est pas le tour du fil 1).
 - Le fil 1 fait exactement la même chose.
 - Les deux fils sont alors en même temps en section critique.
4. Il s'agit de retrouver l'algorithme de Peterson. En l'occurrence, il suffit de changer dans `f_b` le `tour := i` par `tour := autre`.
5. L'algorithme de la boulangerie de Lamport permet de généraliser à n fils. On y utilise un tableau `veut_entrer` à n cases et un tableau d'entiers `tickets` à n cases. Lorsqu'un fil i veut entrer en section critique, il modifie sa case `veut_entrer.(i)` puis calcule et stocke son ticket comme étant (le maximum du tableau `tickets`) +1. Il attend ensuite d'être le fil souhaitant aller en section critique disposant du couple `(tickets.(i), i)` le plus petit pour l'ordre lexicographique.

Chapitre 7

(CCINP) SQL * (CCINP 0, ex A, corrigé — 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 ?

correction

Corrigé de J.B.Bianquis et M.Péchaud

1. A mon avis, la seule clé primaire pertinente pour **Achat** serait un nouvel attribut permettant d'identifier de manière unique chaque achat. En effet :
 - Aucun attribut seul ne peut être une clé primaire.
 - La couple (**NumClient**, **NomFabricant**) ne convient pas non plus puisqu'un client pourrait acheter plusieurs objets différents chez le même fabricant.
 - Le triplet (**NumClient**, **NomFabricant**, **Modele**) ne convient pas vraiment non plus car un même client pourrait acheter un même matériel au même fabricant plusieurs fois. Ou alors il faudrait qu'à chaque fois que cela arrive le champ **Quantite** soit modifié.
 - Prendre les quatre attributs comme clé primaire n'est pas très indiqué non plus : un client pourrait acheter

le même matériel au même fabricant dans les mêmes quantités plusieurs fois.

2.
 - Modele est une clé étrangère dans Production référant à Modele dans les tables de matériels.
 - NomFabricant est une clé étrangère dans Production référant à Nom dans la table Fabricant.
 - NumClient est une clé étrangère dans Achat référant à Num dans la table Client.
 - NomFabricant est une clé étrangère dans Achat référant à Nom dans la table Fabricant.
 - Modele est une clé étrangère dans Achat référant à Modele dans les tables de matériels.
 - (NomFabricant, Modele) est une clé étrangère dans Achat référant à la clé primaire du même nom dans la table Production.

3. (a) `SELECT Modele FROM Production WHERE NomFabricant = 'Durand'`

(b) `SELECT Nom, Adresse FROM Fabricant
JOIN Production ON Nom = NomFabricant
JOIN Portable ON Portable.Modele = Production.Modele
WHERE Dd >= 500`

(c) `SELECT NomFabricant FROM Production JOIN Imprimante ON Production.↔
Modele = Imprimante.Modele GROUP BY Fabricant HAVING count(Modele)↔
>= 10`

(d) On calcule le numéro des clients qui ont acheté une imprimante puis on les supprime des numéros de l'ensemble des clients avec EXCEPT :

```
SELECT NumClient FROM Client
EXCEPT
SELECT NumClient FROM Achat JOIN Imprimante ON Achat.Modele = ↔
Imprimante.Modele
```

Chapitre 8

(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 σ 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 infixe 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).

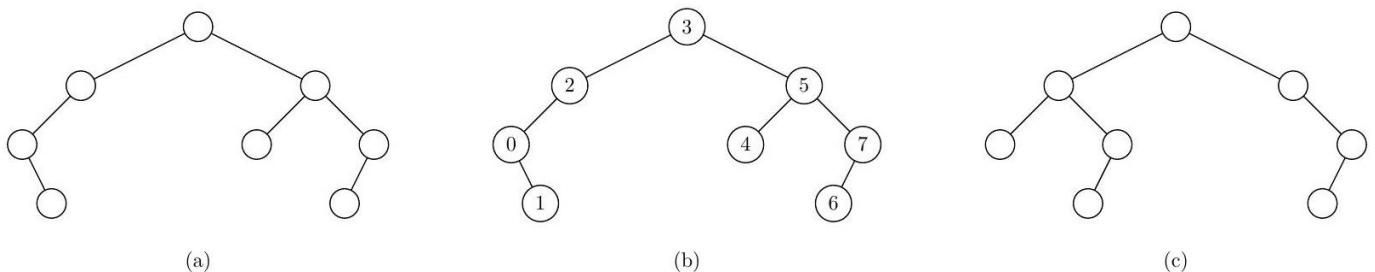


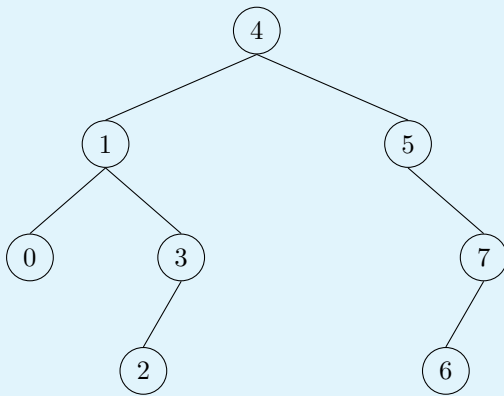
Figure 1 - (a) un arbre binaire non étiqueté; (b) son étiquetage en suivant un ordre infixe, la permutation associée est $[3; 2; 0; 1; 5; 4; 7; 6]$; (c) un autre arbre binaire non étiqueté.

Un fichier source *OCaml* qui implémente ces exemples vous est fourni.

1. Étiqueter l'arbre (c) de la figure 1 et donner la permutation associée.

correction

Correction de M.Péchaud



La permutation associée est donc [4; 1; 0; 3; 2; 5; 7; 6].

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.

correction

```

1 let rec parcours_prefixe a = match a with
2   | V -> []
3   | N(ag, e, ad) -> e : parcours_prefixe ag@parcours_prefixe ad

```

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 infixe 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.

correction

```

1 let etiquette a =
2   let rec aux a e =
3     match a with
4     | V -> V, e
5     | N(ag, _, ad) -> let new_ag, eg = aux ag e in
6                       let new_ad, ed = aux ad (eg+1) in
7                       N(new_ag, eg, new_ad), ed
8   in
9   aux a 0

```

Une permutation σ 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 \leftarrow 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.

correction

La permutation est [4; 1; 0; 3; 2; 5; 7; 6].

- On empile 4, 1 et 0.
- On dépile 0 et 1.
- On empile 3 et 2.
- On dépile 2, 3 et 4.
- On empile 5.
- On dépile 5.
- On empile 7 et 6.
- On dépile 6 et 7.

Les sommets ont bien été dépilés dans l'ordre croissant, donc la permutation est triable par liste.

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.

correction

```

1 let trier l =
2   let rec aux entree pile sortie =
3     match entree, pile with
4       [], [] -> sortie
5       | [], tp :: qp -> aux [] qp (tp :: sortie)
6       | te :: qe, [] -> aux qe (te :: pile) sortie
7       | te :: qe, tp :: qp when te > tp -> aux entree qp (tp :: sortie)
8       | te :: qe, _ -> aux qe (te :: pile) sortie
9   in
10  aux l [] []

```

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

correction

Dans la situation donnée, σ_i finit par être empilé.

Comme $\sigma_j > \sigma_i$, au plus tard lorsque σ_j est étudié en tête d'entrée, σ_i est dépilé.

σ_k sera donc dépilé après σ_i – et comme $\sigma_k < \sigma_i$, la sortie ne sera pas triée.

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 infixé et l'opération dépiler d'autre part.

correction

- Les sommets sont empilés dans l'ordre de parcours infixé de l'arbre (i.e. l'ordre dans lequel ils apparaissent dans la permutation donnée)
- et dépilés par ordre de parcours préfixe – car étant donné un nœud d'étiquette e avec des fils gauches et droits d'étiquettes e_g et e_d , e sera dépilé après le dépillement de e_g (qui a été empilé après), et e_d ne pourra être empilé que lorsque e aura été dépilé.

Or par définition, l'ordre de parcours préfixe est l'ordre croissant, donc le résultat renvoyé est bien correct, et la permutation est triable par pile.

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

Indication : on peut prendre σ_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.

correction

Considérons une permutation σ triable par liste.

Procédons comme dans l'énoncé et construisons un arbre comme indiqué.

Montrons que l'arbre obtenu représente bien la permutation σ – i.e. que l'on obtient bien cette permutation en le parcourant dans l'ordre infixe (plus précisément en considérant des permutations de *parties* de $\llbracket 0, n - 1 \rrbracket$).

On peut procéder par induction sur l'arbre – le résultat étant évident pour un arbre réduit à la racine. Considérons alors un arbre de la forme $\mathbf{a} = \mathbf{N}(\mathbf{ag}, \mathbf{e}, \mathbf{ad})$ – et supposons que la propriété est vraie pour ag et ad .

Si l'on effectue un parcours préfixe de l'arbre, on obtient successivement

- e est la racine – qui est bien le premier élément de la permutation ;
- ag est ensuite parcouru. Ces éléments sont par construction les e éléments suivants de la permutation, et par hypothèse d'induction, le parcours préfixe les voit dans l'ordre de la permutation.
- ad est ensuite parcouru. Ces éléments sont par construction les éléments restants de la permutation, et par hypothèse d'induction, le parcours préfixe les voit dans l'ordre de la permutation.

Le résultat est donc démontré.

Chapitre 9

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

*

Graphes, C,
sources : `ccinpgraphe.tex`

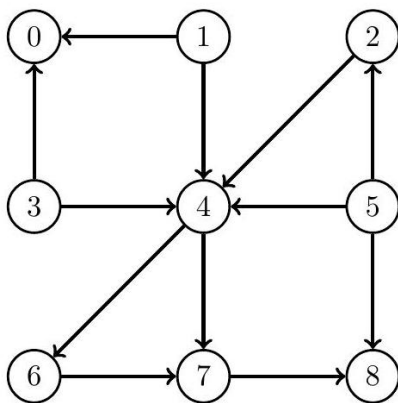
L'exercice suivant est à traiter dans le langage C. Vous pourrez travailler dans le fichier `ccinpgraphe.c` fourni.

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 $\max \{d^+(s') \mid s' \in S'\}$.

correction

Correction de M.Péchaud

```

1 int degre_max (graph* g, bool* partie) {
2     int r = 0;
3     for (int i = 0; i < g->n; i++) {
4         if (partie[i] && g->degre[i] > r)
5             r = g->degre[i];
6     }
7     return r;
8 }

```

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_accessible` qui utilise votre fonction et un test pour l'exemple ci-dessus vous sont donnés dans le fichier à compléter.

correction

```

1 bool* accessibles (graph* g, int s) {
2     int n = g->n;
3     bool* r = malloc(n * sizeof(bool));
4     for (int i = 0; i < n; i++)
5         r[i] = false;
6     parcours(g, s, r);
7     return r;
8 }

```

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 ?

correction

```

1 int degre_etoile(graph* g, int s) {
2     return degre_max(g, accessibles(g, s));
3 }

```

`accessibles` effectue un parcours du graphe (de complexité $O(|S| + |A|)$).
`degre_max` s'exécute en temps $O(|S|)$.
 La complexité est donc $O(|S| + |A|) = O(n^2)$.

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.

correction

Un tri topologique possible est 5, 2, 3, 1, 0, 4, 6, 7, 8.
 Le tracé du graphe est laissé au lecteur ou à la lectrice.

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 $O(|S| + |A|)$.

correction

On effectue un tri topologique du graphe à l'aide d'un parcours en profondeur, qui a une complexité $O(|S| + |A|)$.
 On crée un tableau pour mémoriser les résultats, et on parcourt alors les sommets par ordre inverse de ce tri topologique, en remarquant que le dernier élément n'a pas de successeur, et donc que son d^* vaut 0, et que pour chaque élément s voisins dans V_s , $d^*(s) = \max_{v \in V_s} d^*(v_s)$ (les $d^*(v_s)$ ont déjà été calculés et mémorisés – donc on y accède en $O(1)$).
 L'algorithme obtenu est donc bien de complexité $O(|S| + |A|)$ – chaque arc étant considéré exactement une fois dans le calcul de V_s .

6. On ne suppose plus le graphe acyclique. Décrire un algorithme permettant de calculer tous les $d^*(s)$ pour $s \in S$ en $O(|S| + |A|)$.

correction

On effectue le calcul des composantes fortement connexes du graphe à l'aide de l'algorithme de Kosaraju, qui a une complexité $O(|S| + |A|)$.

On utilise une stratégie de mémorisation comme à la section précédente.

On parcourt les composantes connexes par ordre inverse d'un tri topologique.

- Notons C la dernière composante connexe (donc la première considérée), et $s \in C$. On parcourt le graphe à partir de s pour calculer $d^*(s)$ – et l'on mémorise le résultat.

On remarque alors que tous les autres sommets de C ont la même image par d^* – on mémorise ces résultats.

- Notons C' la seconde composante connexe considérée, et $s \in C'$. On utilise la même méthode qu'au point précédent – en interrompant le parcours dès que l'on rencontre un sommet de C pour lequel d^* a déjà été calculé.

- ...

L'algorithme obtenu est donc bien de complexité $O(|S| + |A|)$.

Chapitre 10

(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 ℓ_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?

correction

Non.

Il s'agit d'un algorithme de Monte-Carlo (complexité fixée, mais pas de garantie que le résultat soit correct).

3. Proposer un jeu de 5 tests élémentaires permettant de tester la correction de ce programme.

correction

On propose le jeu de tests suivant, qui couvre les cas où

- la formule est vide,
- il y a une clause vide,
- la formule est satisfiable avec plusieurs solutions,
- la formule est satisfiable avec une unique solution,
- la formule n'est pas satisfiable.

```

1
2 random_sat [[|V(0) ; NV(1)|] ; [|V(0) ; V(1)|]] 2 3;;
3
4 random_sat [[|V(0) ; NV(1)|] ; [|V(0) ; V(1)|]] 2 1;;
5
6 random_sat [[|V(0) ; NV(1)|] ; [|V(0) ; V(1)|] ; [|NV(0) ; NV(1)|] ; [|NV(0) ; V↔
  (1)|]] 2 5;;
7
8 random_sat [[|V(0) ; NV(1)|] ; [|V(0) ; V(1)|] ; [|]] 2 5;;
9
10 random_sat [[|V(0) ; NV(1) ; V(2)|] ; [|V(0) ; NV(1) ; NV(2)|] ; [|NV(0)|] ; [|V↔
  (0) ; V(1) ; V(2)|]] 3 50;;

```

4. Compléter la fonction `random_sat`.

correction

```

1 (*Étant donné une formule, renvoie Some(une clause) non satisfaite par v, ↔
  None si la formule est satisfaite*)
2 let rec clause_non_sat f v = match f with
3   [] -> None
4   | t :: q -> if evaluate t v
5                 then clause_non_sat q v
6                 else Some t
7 ;;
8
9 let random_sat f n k =
10  let v = initialise n in
11  let rec aux num =
12    if num = k then None
13    else match clause_non_sat f v with
14         | None -> Some v
15         | Some c ->
16           let na = Array.length c in
17           if na = 0
18           then None
19           else let rand_index = Random.int na in
20                match c.(rand_index) with
21                 | V(i) | NV(i) -> v.(i) <- not v.(i); aux (num + 1)
22
23  in
24  aux 0

```

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 ?

correction

```

(*nombre de clauses de f satisfaites par v*)
let rec nb_sat f v =
  match f with
  [] -> 0
  | t :: q -> nb_sat q v + if evaluate t v then 1 else 0
;;

```

```

let max_sat f n k =
  let r = ref 0 in
  for i = 0 to k-1 do
    let res = nb_sat f (initialise n) in
    if res > !r then r := res
  done;
  !r
;;

```

La complexité temporelle est dans le pire des cas $O(k(n + |f|))$ – `nb_sat` s'exécutant en temps linéaire en la taille de la formule.

Dans le meilleur des cas, chaque clause va être immédiatement satisfaite, et l'on obtiendra une complexité $O(kc)$, où c est le nombre de clauses de la formule.

2. Sous l'hypothèse $P \neq NP$, peut-il exister un algorithme de complexité polynomiale pour résoudre MAX-SAT? Justifier.

correction

Non, car sinon, on pourrait résoudre SAT en temps polynomial en testant simplement si le résultat renvoyé par l'algorithme résolvant MAX-SAT renvoie le nombre de clauses.

Chapitre 11

(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} .

correction

Correction de M.Péchaud

Je trouve curieux de n'avoir qu'un type de données pour l'entrée et la sortie.
Je propose la structure suivante pour une instance du problème,

```
1 struct sacados_t {
2     int n;
3     int* vals;
4     int* poids;
5 };
6
7 typedef struct sacados_t sacados;
```

et de représenter les indicateurs par un `int*` (on pourrait penser à un `bool*`, mais cela ne respecterait pas la convention de l'énoncé).

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.

correction

```
1 int* naif(sacados* sac, int pmax) {
2     int* ind = malloc(sac->n * sizeof(int));
3     int poids_restant = pmax;
4     for (int i = 0; i < sac->n; i++) {
5         if (sac->poids[i] <= poids_restant) {
6             ind[i] = 1;
7             poids_restant -= sac->poids[i];
8         }
9         else {
10            ind[i] = 0;
11        }
12    }
13    return ind;
14 }
```

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.

correction

```

1 int main() {
2     sacados s;
3     printf("Entrez n svp\n");
4     scanf("%d", &s.n);
5     s.poids = malloc(s.n * sizeof(int));
6     s.vals = malloc(s.n * sizeof(int));
7
8     int pmax;
9
10    printf("Entrez pmax\n");
11    scanf("%d", &pmax);
12
13    printf("Entrez les n poids\n");
14    for (int i = 0; i < s.n; i++) {
15        scanf("%d", &s.poids[i]);
16    }
17    printf("Entrez les n valeurs\n");
18    for (int i = 0; i < s.n; i++) {
19        scanf("%d", &s.vals[i]);
20    }
21
22    int* ind = naif(&s, pmax);
23
24    printf("Indicateurs : ");
25    for (int i = 0; i < s.n; i++)
26        printf("%d", ind[i]);
27
28    printf("\n");
29
30    int val_tot = 0;
31    int poids_tot = 0;
32
33    for (int i = 0; i < s.n; i++)
34        if (ind[i] == 1) {
35            val_tot += s.vals[i];
36            poids_tot += s.poids[i];
37        }
38
39    printf("valeur : %d\npoids : %d", val_tot, poids_tot);
40
41 }

```

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.

correction

- (a) Non. Exemple : $p_{max} = 5$, $v = p = (4, 3, 2)$ (l'optimum est 5, l'algorithme glouton renvoie 4).
- (b) Non, même exemple que précédemment.
- (c) Non. Exemple : $p_{max} = 5$, $p = (1, 1, 1, 1, 1, 5)$, $v = (1, 1, 1, 1, 1, 10)$ (l'optimum est 10, l'algorithme glouton renvoie 5).
- (d) Non. Exemple : $p_{max} = 3$, $p = (2, 3)$, $v = (3, 4)$ (l'optimum est 4, l'algorithme glouton renvoie 3).

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_{seuil} . Montrer que ce problème de décision est dans la classe NP.

correction

Il s'agit du problème consistant à déterminer – étant donnée une instance du problème – s'il existe des indicateurs respectant la contrainte de poids et donnant une valeur supérieure ou égale à v_{seuil} . Ce problème est immédiatement dans NP en prenant comme certificat les indicateurs (on vérifie alors en temps linéaire que la contrainte de poids est satisfaite, et que la valeur dépasse le seuil donné).

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 ?

correction

Il y a 2^n choix pour les indicateurs, donc on obtiendrait naïvement une complexité $O(n2^n)$ pour trouver l'optimum.

Si l'on donne des valeurs 0 ou 1 séquentiellement à chaque objet, on peut couper une branche dès que l'on ne satisfait plus la contrainte de poids.

On peut également utiliser en plus une approche de type branch and bound en coupant une branche dès que le fait de prendre tous les objets restants ne permettrait plus de dépasser la valeur maximale trouvée jusqu'à présent.

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

Chapitre 12

(CCINP) Tableaux * (CCINP 0, ex B, corrigé — 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 ?

correction

```
1 int nb_occurrences(int n, int* tab, int x) {
2     int r = 0;
3     for (int i = 0; i < n; i++) {
4         if (tab[i] == x) r+=1;
5     }
6     return r;
7 }
```

La complexité temporelle de cette fonction est immédiatement $O(n)$.

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 $O(\log n)$.

correction

```

1 int une_occurrence (int n , int* tab, int x) {
2     int d = 0;
3     int f = n-1;
4     while (d <= f) {
5         int m = (d + f)/2;
6         if (tab[m] == x) return m;
7         if (tab[m] < x) d = m + 1;
8         else f = m - 1;
9     }
10    return -1
11 }

```

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 $O(\log n)$.

correction

On prend pour invariant «si l'élément apparaît dans le tableau, sa première occurrence est entre les indices d et f inclus».

```

1 int premiere_occurrence(int n, int *tab, int x) {
2     int d = 0;
3     int f = n-1;
4     while (d <= f) {
5         if (d == f && tab[d] == x) return d;
6         int m = (d + f)/2;
7         if (tab[m] == x) f = m; // diminue strictement dès lors que d != f
8         if (tab[m] > x) f = m - 1;
9         if (tab[m] < x) d = m + 1;
10    }
11    return -1;
12 }

```

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 $O(\log n)$.

correction

On commence par écrire une fonction `derniere_occurrence` en adaptant la fonction précédente, mais il faut être attentif à la terminaison : si l'on écrit $d = m$, d peut ne pas être modifié dans le cas où $f = d + 1$.

```

1 int derniere_occurrence(int n, int *tab, int x) {
2     int d = 0;
3     int f = n-1;
4     while (d <= f) {
5         if (d + 1 >= f) {
6             if (tab[f] == x) return f;
7             else if (tab[d] == x) return d;
8         }
9         int m = (d + f)/2;
10        if (tab[m] == x) d = m;
11        if (tab[m] > x) f = m - 1;
12        if (tab[m] < x) d = m + 1;
13    }
14    return -1;
15 }

```

5. Justifier que la fonction `une_occurrence` termine et est correcte. On donnera un variant et un invariant de boucle que l'on justifiera.

correction

Un invariant est ici : «si l'élément apparaît dans le tableau, il apparaît entre les indices d et f inclus, et si $f < d$, il n'apparaît pas».

- si un résultat autre que -1 est renvoyé, c'est un m tel que $tab[m] = x$, ce qui est correct.
- si -1 est renvoyé, c'est que l'on est sorti de la boucle, donc que $d > f$ – ce qui d'après l'invariant ci-dessus signifie que l'élément n'apparaît pas.

Un variant possible est $f - d$, qui par construction décroît strictement à chaque passage dans la boucle qui ne termine pas la fonction.

6. Montrer que la complexité de la fonction `une_occurrence` est bien en $O(\log n)$.

correction

$f - d + 1$ (la taille du sous-tableau sur laquelle on travaille) est au moins divisé par 2 à chaque passage dans la boucle qui ne termine pas la fonction.

Cette longueur est donc inférieure à $n/2^i$ après i itérations.

La fonction termine au plus tard lorsque $n/2^i \leq 1$, ce qui équivaut à $i \geq \log_2(n)$ – qui est vrai dès que $i = \lceil \log_2(n) \rceil$.

La complexité est donc $O(\log_2(n))$.

Chapitre 13

(CCINP) Tableaux autoréférents * (CCINP 0, ex B, corrigé — 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

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

correction

```
let rec somme t i =  
  if i = -1 then 0  
  else t.(i) + somme t (i-1)
```

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 $n = 4$.

correction

- Pour $n = 3$,
 - $T[0]$ ne peut contenir 0 pour la même raison que précédemment.
 - Si $T[0] = 1$, T est de la forme $[1, 0, ?]$ ou $[1, ?, 0]$. Le premier cas est impossible car 1 apparaît au moins une fois. Pour le second cas, $T[1] = 1$ est impossible car 1 apparaît au moins deux fois. $T[1] = 2$ ne convient pas non plus.
 - Si $T[0] = 2$, la seule possibilité est $[2, 0, 0]$, qui ne convient pas car 2 apparaît une fois.
- Pour $n = 4$, $[2; 0; 2; 0]$ convient.

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 $O(n)$.

correction

```

let est_auto t =
  let n = Array.length t in
  let cnt = Array.make n 0 in
  for i = 0 to n-1 do
    cnt.(t.(i)) <- cnt.(t.(i)) + 1
  done;
  t = cnt

```

La complexité est bien linéaire en la taille du tableau.

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.

correction

La somme des éléments d'un tableau autoréférent est la longueur du tableau – chaque élément indiquant combien de fois l'un des indices y apparaît.

Donc on peut arrêter l'exploration d'une branche à partir du moment où la somme partielle des éléments alloués jusqu'à présent dépasse la longueur totale du tableau.

```

let gen_auto_opt n =
  let t = Array.make n 0 in
  let rec explore i = (*explore tous les tableaux où les i premières ←
    cases sont celles de t*)
    if i = n then (
      if est_auto t
      then print_array t
    )
    else if somme t (i-1) <= n then
      for j = 0 to n - 1 do
        t.(i) <- j;
        explore (i+1)
      done;
  in
  explore 0

```

Il serait préférable de ne pas recalculer la somme partielle à chaque fois.

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 $n = 15$?

correction

Dans cette condition, le tableau ne pourra pas être autoréférent, et on peut donc élaguer.

```

let gen_auto_opt2 n =
  let t = Array.make n 0 in
  let rec explore i = (*explore tous les tableaux où les i premières ←
    cases sont celles de t*)
    if i = n then (
      if est_auto t
      then print_array t
    )
    else if somme t (i-1) <= n then
      for j = 0 to n - 1 do
        t.(i) <- j;

```

```

        if count t i i <= j then
            explore (i+1)
        done;
    in
    explore 0

```

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 $n = 30$?

correction

Il reste $n - 1 - i$ cases à remplir, dont $t.(0)$ seront mises à 0, donc $n - 1 - i - t.(0)$ à une valeur au moins 1.

Donc la somme totale sera au moins égale à $n - 1 - i - t.(0) + s_i$ - où s_i est la somme partielle de 0 à i . Cette somme dépasse n ssi $n - 1 - i - t.(0) + s_i > n$ ssi $s_i > 1 + i + t.(0)$.

```

let gen_auto_opt3 n =
  let t = Array.make n 0 in
  let rec explore i = (* explore tous les tableaux où les i premières ↵
                       cases sont celles de t*)
    if i = n then (
      if est_auto t
      then print_array t
    )
    else let s = somme t (i-1) in
         if s <= n then
           for j = 0 to n - 1 do
             t.(i) <- j;
             if count t i i <= j && s + t.(i) <= 1 + i + t.(0) then
               explore (i+1)
             done;
         in
  explore 0

```

Pour $n = 30$, je n'obtiens pas un temps d'exécution raisonnable, mais le code doit pouvoir être largement optimisé.

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.

correction

Pour $n \geq 7$, le tableau suivant est autoréférent :

```
[|n-4, 2, 1, 0, 0, ..., 0, 1, 0, 0, 0|]
```


Chapitre 14

(CCINP) Activation de processus * (CCINP 23, ex A, corrigé — 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.

correction

Correction fournie par le concours.

Oui pour $k = 2$ car on peut activer P_2 et P_3 en même temps. Non pour $k = 3$ car si on active trois processus, au moins deux utilisent la ressource r_1 .

Jury : On attend du candidat une réponse en oui / non avec une précision de quels processus activer dans le cas où la réponse est oui et une très rapide justification dans le cas où la réponse est non. Une réponse orale suffit.

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.

correction

On suppose que l'entrée de l'algorithme est un ensemble de n couples (numéro de la machine, numéro de la ressource qu'elle utilise). On peut alors proposer l'algorithme suivant :

```
1 Initialiser un tableau  $T$  à  $m$  cases  $\leftarrow$   
   contenant faux  
2 nb_machines = 0  
3 Pour Chaque couple  $(n_{\text{machine}}, n_{\text{ressource}})$   
4   Si  $T[n_{\text{ressource}}] = \text{faux}$   
5      $T[n_{\text{ressource}}] = \text{vrai}$   
6     Incrémenter nb_machines  
7 Renvoyer nb_machines
```

Le principe est que le tableau T indique en case i si la i -ème ressource est utilisée ou non. On pourrait même calculer sans surcoût les machines activées en plus de leur nombre avec cette méthode. La complexité de cette méthode est en $O(n + m)$ (à cause de l'initialisation de T) mais on pourrait la faire passer à un $O(n)$ en stockant les informations de T dans un dictionnaire (les clés seraient les numéros des ressources, et les valeurs n'ont pas d'importance).

Jury : De multiples réponses sont possibles sur cette question et sont acceptées du moment qu'elles sont clairement décrites et correctement analysées. On n'attend pas une complexité optimale.

On souhaite montrer que ACTIVATION est NP-complet.

3. Donner un certificat pour ce problème.

correction

Un sous-ensemble de numéros de machines convient (et il est de taille au plus n donc polynomial en la taille de l'entrée de ACTIVATION).

Jury : Une courte phrase décrivant la nature d'un tel certificat et indiquant sa polynomialité en la taille de l'entrée suffit à obtenir tous les points.

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(P_i ,R)` qui renvoie `Vrai` si le processus i utilise une ressource incluse dans un ensemble de ressources R .
 (c) `ajoute(P_i ,R)` qui ajoute les ressources P_i dans l'ensemble R et renvoie ce nouvel ensemble.

correction

On propose le code de vérification suivant étant donné un certificat c :

```

1 Ressources_utilisées = ∅
2 Pour toute machine i
3 {
4     Si appartient(c,i)
5     {
6         Si intersecte(Pi, Ressources_utilisées)
7         {
8             Renvoyer faux
9         }
10        Ressources_utilisées = ajoute(Pi, ←
11        Ressources_utilisées)
12    }
13 Renvoyer vrai
  
```

Pour chaque machine, si elle appartient à l'ensemble des machines qu'on souhaite activer simultanément, on vérifie qu'elle n'entre pas en conflit sur les ressources avec une des machines qui la précède (ce qui suffit par symétrie de la relation "entrer en conflit"). La variable `Ressources_utilisées` représente l'ensemble des ressources utilisées par les machines considérées jusque là.

Jury : On attend un algorithme utilisant à bon escient les primitives proposées par le sujet.

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.

correction

Les questions 3 et 4 montrent déjà que ACTIVATION est dans NP puisque l'algorithme en Q4 est de complexité polynomiale en la taille de l'entrée du problème d'après l'énoncé

Soit $I_S = \{G = (S, A), k\}$ une instance de STABLE. Considérons alors l'instance suivante I_A de ACTIVATION : on crée une machine i par sommet de S , une ressource r par arête et la liste de ressources nécessaires à la machine i est $\{r \text{ ressource} \mid r \text{ correspond à une arête incidente à } i\}$; on conserve par ailleurs k . Elle est constructible à partir de I_S en temps $O(|S| + |A|)$. De plus S' est un stable de taille k dans G si et seulement si les k machines de S' sont activables en même temps.

Ceci montre que $\text{STABLE} \leq \text{ACTIVATION}$ et comme le premier est NP-difficile, il en va de même pour le second, ce qui achève la preuve de NP-complétude de ACTIVATION.

Jury : Le jury est attentif au fait que tous les arguments soient bien présents : description de la réduction, preuve

que c'en est une et justification de son caractère polynomial pour le caractère NP-difficile ; caractère NP pour pouvoir conclure quant à la NP-complétude.

Chapitre 15

(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.

correction

Corrigé fourni par le concours.

On fixe $a : E \rightarrow \{V, F\}$ une valuation. On étend a par induction comme suit

$$\begin{aligned} \text{Si } P \text{ et } Q \text{ sont des formules propositionnelles, alors} \quad & a(\neg P) = \text{Négation de } a(P). \\ & a(P \wedge Q) = a(P) \text{ et } a(Q), \\ & a(P \vee Q) = a(P) \text{ ou } a(Q). \end{aligned}$$

Jury : Le jury profite de cette question pour vérifier que le candidat sait faire la différence entre syntaxe et sémantique.

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.

correction

Soient P, Q deux formules croissantes. Montrons que $P \wedge Q$ et $P \vee Q$ sont croissantes.

Soient a, b deux valuations vérifiant $a \leq b$. En utilisant l'abus de notation, on a les égalités suivantes :

$$\begin{aligned} a(P \wedge Q) &= \min(a(P), a(Q)), & a(P \vee Q) &= \max(a(P), a(Q)), \\ b(P \wedge Q) &= \min(b(P), b(Q)), & b(P \vee Q) &= \max(b(P), b(Q)). \end{aligned}$$

Sachant que $a \leq b$, on a donc $a(P) \leq b(P)$ et $a(Q) \leq b(Q)$. On en déduit :

$$\min(a(P), a(Q)) \leq \min(b(P), b(Q)) \text{ et } \max(a(P), a(Q)) \leq \max(b(P), b(Q))$$

Autrement dit $a(P \wedge Q) \leq b(P \wedge Q)$ et $a(P \vee Q) \leq b(P \vee Q)$.
 $P \wedge Q$ et $P \vee Q$ sont bien croissantes.

Jury : Le candidat est autorisé à utiliser l'abus introduit par l'énoncé (assimiler V à 1 et F à 0) mais le jury se réserve le droit de vérifier que le candidat a bien compris en quoi c'est un abus. Un candidat qui prouve rigoureusement la propriété souhaitée pour un des connecteurs et indique ce qu'il suffirait d'y changer pour l'autre obtient tous les points.

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 $x \in E$.

correction

On montre par double implication.

- (Sens réciproque). Soit C une clause conjonctive satisfiable contenant au moins un littéral. Une clause contenant un unique littéral positif est clairement croissante. Par récurrence et par stabilité des formules croissantes par l'opération \wedge , on en déduit que toute clause conjonctive contenant uniquement des littéraux positifs est croissante.
- (Sens direct, par la contraposée). Considérons une clause conjonctive C satisfiable contenant au moins un littéral. Montrons que si celle-ci est croissante, alors tout littéral apparaissant dans C est positif. Supposons que C contient au moins un littéral négatif et quitte à réarranger les termes, on considère que $C = \neg x \wedge C'$ avec C' une clause conjonctive ne contenant pas x (possible car C est satisfiable). Considérons une valuation a vérifiant $a(C) = V$. On a alors $a(\neg x) = V$ et $a(C') = V$. Donc $a(x) = F$. On choisit b qui coïncide avec a sur toutes les variables propositionnelles exceptée sur x où on a $b(x) = V$. Par construction, $a \leq b$. Mais $b(C) = F$. Donc C n'est pas une formule croissante.

On a bien l'équivalence demandée.

Jury : Il ne faut pas oublier de traiter le caractère nécessaire et suffisant.

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.

correction

Si P est logiquement équivalente à une disjonction de clauses conjonctives sans littéral négatif, par stabilité de la croissance par disjonction et les clauses conjonctives sans littéral négatif étant croissantes, on en déduit que P est 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 $x \in E$.

correction

Réciproquement, soit P une formule croissante. On pose :

$$P' = \bigvee_{C \models P} C$$

où pour deux formules P, Q , on a $P \models Q$ si pour toute valuation a , $a(P) = V \implies a(Q) = V$.

Dans notre cas, C désigne une clause conjonctive croissante inférieure à P (Autrement dit, pour toute valuation a : $a(C) \leq a(P)$) qui ne contient aucun littéral négatif. Vérifions que P' est logiquement équivalente à P .

- soit a une valuation vérifiant $a(P') = V$. Il existe $C \leq P$ une clause telle que $a(C) = V$. On a bien $a(P) = V$.
- Réciproquement, si a est une valuation vérifiant $a(P) = V$, on pose :

$$C = \bigwedge_{x \text{ positif } a(x)=V} x$$

Remarquons que si C était indexé par l'ensemble vide, a enverrait tous les littéraux sur F et dans ce cas par croissance, P serait une tautologie ce qui contredit l'hypothèse de l'énoncé. Les propriétés suivantes sont alors vérifiées :

- C n'est pas indexé par le vide (sinon, a enverrait tous les littéraux positifs sur F et par croissance, P serait une tautologie)
- C contient uniquement des littéraux positifs,
- on a $C \models P$ par croissance de P .

Donc C apparaît dans P' . D'où $a(P') = V$.
Ainsi, P et P' sont bien logiquement équivalentes.

Jury : Le jury peut donner une indication pour aider le candidat à trouver une formule permettant de répondre à cette question.

Chapitre 16

(CCINP) Grammaires algébriques * (CCINP 23, ex A, corrigé — 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 :

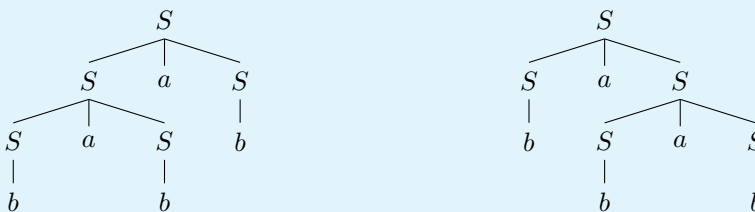
$$S \rightarrow SaS \mid b$$

1. Cette grammaire est-elle ambiguë? Justifier.

correction

Correction de J.B.Bianquis

Cette grammaire est ambiguë car le mot $babab \in L(G)$ admet les deux arbres syntaxiques suivants :



Or ces derniers sont différents.

Jury : Le jury attend du candidat qu'il exhibe un mot montrant l'ambiguïté, en justifiant cette dernière via deux arbres syntaxiques différents ou bien deux dérivations gauches (ou droites) différentes.

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?

correction

On constate que $L(G)$ est rationnel car dénoté par l'expression rationnelle $(ba)^*b$.

Jury : Il est attendu que le candidat exhibe une expression régulière pour ce langage (qui est donc régulier). Plusieurs sont possibles et toutes sont acceptées. Aucune justification n'est nécessaire à ce stade, sauf si l'expression proposée est démesurément complexe, auquel cas le jury invite le candidat à lui expliquer. Certains candidats n'ont pas compris ce qui était attendu par «la plus petite classe de langages», le jury a alors précisé ses attentes sans pénaliser pour autant ces candidats.

Au moins un candidat a répondu à cette question en affirmant que $L(G)$ est en fait local. Il a été invité à poursuivre l'exercice avec cette réponse.

3. Prouver que $L = L(G)$.

correction

Montrons par récurrence forte sur $n \in \mathbb{N}^*$ la propriété $H(n)$ suivante : si $u \in L$ est un mot de taille n alors u est engendré par la grammaire G . C'est bien sûr le cas pour $n = 1$ puisque le seul mot de L de cette taille est b , engendré par la deuxième règle de G .

Soit donc $n \in \mathbb{N}^*$ et u un mot de L de taille $n + 1$. Comme $|u| \geq 2$, ba est nécessairement préfixe de u et il existe donc $v \in (ba)^*b = L$ tel que $u = bav$. Par hypothèse, ce mot v est engendré par G : il existe une dérivation telle que $S \Rightarrow^* v$. On en déduit que

$$S \Rightarrow SaS \Rightarrow baS \Rightarrow^* bav = u$$

est une dérivation licite et donc que $u \in L(G)$. Cette récurrence montre que $L \subset L(G)$.

Montrons réciproquement que $L(G) \subset L$ en montrons par récurrence forte sur $n \in \mathbb{N}^*$ la propriété $H(n)$ suivante : si $u \in \Sigma^*$ se dérive de S en n dérivations alors $u \in L$. C'est acquis pour $n = 1$: le seul mot de Σ^* qu'on peut obtenir en une dérivation est $b \in L$.

Soit donc $n \in \mathbb{N}^*$ et u un mot dans $L(G)$ tel que $S \Rightarrow^{n+1} u$. Comme ce mot est obtenu en au moins 2 dérivations, les règles de G nous informent que la première est nécessairement $S \rightarrow SaS$ (sans quoi ce serait $S \rightarrow b$ et dans ce cas u serait obtenu en une seule dérivation). Donc la dérivation permettant d'obtenir u se décompose en :

$$S \Rightarrow SaS \Rightarrow^n u$$

On en déduit qu'il existe $v, w \in \Sigma^*$ et $k_1, k_2 \in \llbracket 1, n \rrbracket$ tels que $u = vaw$, $S \Rightarrow^{k_1} v$, $S \Rightarrow^{k_2} w$ et $k_1 + k_2 = n$. L'hypothèse de récurrence (forte) s'applique à v et w et on en déduit que ces deux mots appartiennent au langage dénoté par $(ba)^*b$ donc qu'il existe $r_1, r_2 \in \mathbb{N}$ tels que $v = (ba)^{r_1}b$ et $w = (ba)^{r_2}b$. Par conséquent, $u = (ba)^{r_1}ba(ba)^{r_2}b = (ba)^{r_1+r_2+1}b \in L$.

Jury : On attend une preuve précise et rigoureuse, par exemple par double inclusion. Les explications vagues et les arguments d'évidence ne satisfont pas le jury sur cette question.

4. Décrire une grammaire qui engendre L de manière non ambiguë en justifiant de cette non ambiguë.

correction

On sait à présent que $L(G) = (ba)^*b$; il s'agit donc de trouver une grammaire non ambiguë engendrant ce langage. On peut proposer par exemple la grammaire dont les règles sont :

$$S \rightarrow Tb \quad T \rightarrow baT \mid \varepsilon$$

les règles sur T permettant de générer le facteur dans $(ba)^*$ et la première de rajouter le b final.

Cette grammaire G' est non ambiguë car pour tout mot dans $L(G')$, il existe une unique dérivation permettant de le construire (donc évidemment un seul arbre syntaxique) ; cette unicité découlant du fait que dans cette grammaire un non terminal se dérive toujours en un mot qui contient au plus un seul non terminal.

Jury : Comme pour la deuxième question, plusieurs grammaires sont ici possibles. On n'attend pas une preuve rigoureuse de non ambiguë.

5. Montrer que tout langage dans la même classe de langages que L peut être engendré par une grammaire algébrique non ambiguë.

correction

La question demande de montrer que tout langage rationnel peut être engendré par une grammaire non ambiguë. Soit donc L un langage rationnel. Par le théorème de Kleene, il existe un automate fini $A = (\Sigma, Q, \{q_0\}, F, \delta)$ qui reconnaît L qu'on peut loiblement supposer déterministe.

Considérons alors la grammaire dont les non terminaux sont $\{V_q \mid q \in Q\}$, l'axiome est V_{q_0} , les terminaux sont les lettres de Σ et dont les règles sont données par :

- Pour toute transition $q \xrightarrow{a} q'$ dans A , on ajoute la règle $V_q \rightarrow aV_{q'}$.
- Pour tout $q \in F$, on ajoute la règle $V_q \rightarrow \varepsilon$.

Cette grammaire engendre L de manière non ambiguë grâce au déterminisme de A .

Jury : Proposer une construction correcte d'une grammaire non ambiguë pour un langage régulier (a priori à partir de l'automate associé) suffit à obtenir tous les points. Le jury n'hésitait pas à aiguiller le candidat si nécessaire pour cette question.

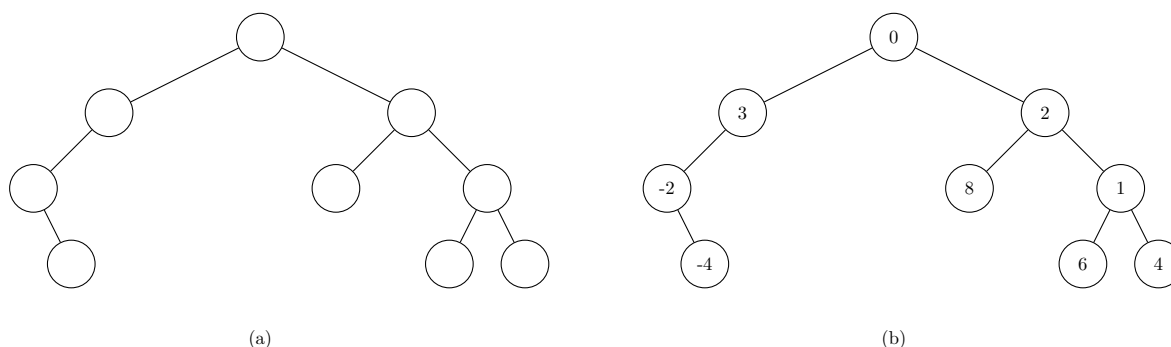
Chapitre 17

(CCINP) Minima locaux dans des arbres (Couverture dans des arbres) * (CCINP 23, ex A, corrigé — 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.

correction

Corrigé de M.Péchaud

1. Les nœuds d'étiquettes -4 , 0 et 1 sont les trois minima locaux.

Jury : On attend simplement du candidat qu'il propose les minima trouvés sans justification. Une réponse orale suffit. En cas d'erreur, le candidat est invité à expliquer son raisonnement.

2. Un arbre est soit un arbre vide soit un nœud formé d'une étiquette et de deux sous-arbres. La hauteur est la profondeur maximale d'une feuille, c'est-à-dire la longueur maximale d'un chemin de la racine à une feuille. La hauteur de l'arbre (b) est 3.

Jury : Dans le cas où le candidat propose une définition inductive des arbres avec une feuille pour cas de

base, il est guidé vers une définition dont le cas de base est l'arbre vide de sorte à rendre compte du fait que les arbres considérés ne sont pas binaires stricts.

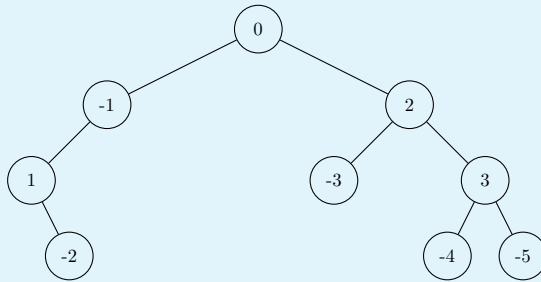
3. L'arbre possède un nombre fini non vide d'étiquettes et donc une étiquette de valeur minimale, qui est un minimum global et donc local.

Jury : Plusieurs stratégies sont ici acceptées (preuve par induction ou existence d'un minimum global qui est local par exemple).

4. Si la racine de l'arbre est un minimum local on a trouvé notre minimum local. Sinon, un des deux fils est non vide, avec une étiquette à sa racine plus petite que celle de la racine de l'arbre. Un appel récursif permet d'obtenir un minimum local de ce sous-arbre, qui est également un minimum local de l'arbre (que ce soit la racine du sous-arbre ou un descendant strict). La complexité est linéaire en la hauteur de l'arbre.

Jury : Si le candidat propose une solution linéaire en la taille de l'arbre, il est guidé vers une solution linéaire en la hauteur.

5. On propose l'étiquetage à la figure (c) dans lesquels les 5 minima locaux sont étiquetés par des entiers strictement négatifs.



(c)

Proposer un étiquetage correct suffit à obtenir tous les points.

6. On propose une approche récursive qui pour un arbre a en entrée calcule $m(a)$ le nombre maximal de nœuds qui peuvent être des minima locaux dans un étiquetage de a , ainsi que, en même temps, la quantité $m_-(a)$ correspondant à cette même quantité mais en supposant de plus que la racine n'est pas un minimum local. Pour un arbre vide, ces deux valeurs valent 0. Pour un arbre a de fils gauche f_g et de fils droit f_d , on peut obtenir par appels récursifs les quantités $m(f_g)$, $m_-(f_g)$, $m(f_d)$ et $m_-(f_d)$. On a alors $m_-(a) = m(f_g) + m(f_d)$ et $m(a) = \max\{m_-(a), 1 + m_-(f_g) + m_-(f_d)\}$. La complexité est linéaire en la taille de l'arbre, chaque nœud est visité exactement une fois avec un nombre d'opérations constant.

Jury : Le jury attend une complexité linéaire. Des indications peuvent être apportées pour aiguiller le candidat vers une telle solution.

7. Le résultat est vrai pour $n = 0$ et on peut donc supposer que $n \geq 1$. Considérons un étiquetage et notons X l'ensemble des nœuds qui sont des minima locaux et Y ceux qui ne le sont pas. On remarque que deux nœuds adjacents ne peuvent pas être tous les deux des minima locaux, puisque toutes les étiquettes sont deux à deux distinctes. Ainsi toute arête de l'arbre est adjacente à au moins un nœud de Y et l'ensemble Y couvre donc toutes les arêtes. Comme chaque nœud de Y est incident à au plus 3 arêtes et qu'il y a exactement $n - 1$ arêtes dans l'arbre, il faut au moins $\frac{n-1}{3}$ nœuds pour couvrir toutes les arêtes, c'est-à-dire $|Y| \geq \frac{n-1}{3}$. On

a donc $|X| = n - |Y| \leq n - \frac{n-1}{3} = \frac{2n+1}{3}$ et donc $|X| \leq \left\lfloor \frac{2n+1}{3} \right\rfloor$.

Jury : Très peu de candidats ont pu aborder cette question.

Chapitre 18

(CCINP) Langages locaux * (CCINP 23, ex B, corrigé — 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 Σ . Si L est un langage sur Σ , 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.

correction

Correction fournie par le concours

On obtient $P(L) = \{a\}$, $D(L) = \{a, b\}$, $F(L) = \{aa, ab, ba\}$ et donc $N(L) = \{bb\}$. Le langage L n'est pas local : s'il l'était, il devrait contenir le mot aba ce qui n'est pas le cas.

On attend du candidat qu'il donne les ensembles demandés sans justification, possiblement à l'oral uniquement puis qu'il exhibe un mot montrant la non localité avec une brève justification.

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 Σ 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 ε , justifier brièvement la correction de `compute_P`.

correction

La seule difficulté est de justifier la disjonction de cas pour les concaténations. Si $e = e_1e_2$ et $\varepsilon \notin L(e_1)$, une première lettre d'un mot de e est nécessairement une première lettre d'un mot de e_1 (la réciproque étant évidente). Sinon, un mot de $L(e)$ peut aussi commencer de la même façon qu'un mot de $L(e_2)$.

Le jury attend principalement deux arguments : le fait que l'on raisonne par induction et la justification de la disjonction de cas dans le cas d'une concaténation.

3. Implémenter la fonction `contains_epsilon`.

correction

On filtre sans grande difficulté.

```
let rec contains_epsilon (e : regexp) : bool = match e with
| Epsilon -> true
| Letter _ -> false
| Union (e1, e2) -> (contains_epsilon e1) || (contains_epsilon e2)
| Concat (e1, e2) -> (contains_epsilon e1) && (contains_epsilon e2)
| Star _ -> true
```

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 .

correction

Comme pour `compute_P`, la seule difficulté est le cas d'une concaténation.

```
let rec compute_D (e : regexp) : string list = match e with
| Epsilon -> []
| Letter a -> [a]
| Union (e1, e2) -> union (compute_D e1) (compute_D e2)
| Concat (e1, e2) when contains_epsilon e2 -> union (compute_D e1) (←
  compute_D e2)
| Star e1 | Concat (_, e1) -> (compute_D e1)
```

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).

correction

Soit e une expression régulière et L le langage qu'elle dénote. On calcule $F(L)$ inductivement :

- Si $e = \varepsilon$ ou $e = a$ avec $a \in \Sigma$, $F(L) = \emptyset$.
- Si $e = e_1 + e_2$, $F(L) = F(L_1) \cup F(L_2)$ où L_i est le langage dénoté par e_i .
- Si $e = e_1e_2$, un facteur de taille 2 d'un mot de L est : soit un facteur de taille 2 de L_1 , soit un facteur de taille 2 de L_2 , soit un facteur "à cheval entre L_1 et L_2 " c'est-à-dire dont la première lettre est la fin d'un mot de L_1 et la deuxième est le début d'un mot de L_2 . Autrement dit :

$$F(L) = F(L_1) \cup F(L_2) \cup D(L_1) \times P(L_2)$$

- Si $e = e_1^*$, on obtient similairement au cas précédent $F(L) = F(L_1) \cup D(L_1) \times P(L_1)$.

Le jury attend un argument inductif et que le candidat précise les cas délicats (étoile et concaténation). Cette question peut être abordée en même temps que la question 7.

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 :

```
prod ["a" ; "c" ; "e"] ["b" ; "c"] = ["ab" ; "ac" ; "cb" ; "cc" ; "eb" ; "ec"]
```

correction

On peut remplacer le `List.map` par une petite fonction auxiliaire au besoin.

```
let rec prod (l1 :string list) (l2 :string list) = match l1, l2 with
| [], _ | _, [] -> []
| t :: q, l -> union (List.map (fun x -> t^x) l) (prod q l)
```

Le jury souhaite une fonction récursive et la demande si cette approche n'est pas proposée. Le candidat est libre de proposer une solution faisant intervenir les fonctions du module `List` ou de s'aider d'une fonction auxiliaire.

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

correction

C'est une traduction de la question 5.

```
let rec compute_F (e :regexp) :string list = match e with
| Epsilon | Letter _ -> []
| Union (e1, e2) -> union (compute_F e1) (compute_F e2)
| Concat (e1, e2) ->
  let d_e1 = compute_D e1 and p_e2 = compute_P e2 in
  union (union (compute_F e1) (compute_F e2)) (prod d_e1 p_e2)
| Star e1 ->
  let d_e1 = compute_D e1 and p_e1 = compute_P e1 in
  union (compute_F e1) (prod d_e1 p_e1)
```

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 .

correction

On remarque qu'une fois calculé $F(L)$ on en déduit immédiatement $N(L)$ en calculant tous les facteurs de taille 2 de Σ et en éliminant ceux qui sont dans $F(L)$.

Il est ensuite très facile de concevoir un automate pour chacun des langages $P(L)\Sigma^*$, $\Sigma^*D(L)$ et $\Sigma^*N(L)\Sigma^*$; les ensembles $P(L)$, $D(L)$ et $N(L)$ étant finis. De plus, on sait algorithmiquement construire l'intersection (automate produit) et le complémentaire (déterminisation puis échange des états finaux et non finaux) d'automates donc on peut construire un automate reconnaissant

$$P(L)\Sigma^* \cap \Sigma^*D(L) \cap (\Sigma^*N(L)\Sigma^*)^c = (P(L)\Sigma^* \cap \Sigma^*D(L)) \setminus (\Sigma^*N(L)\Sigma^*)$$

Une description haut niveau de la construction de cet automate (exploitant par exemple les stabilités des langages rationnels) suffit.

9. Décrire un algorithme permettant de détecter si le langage dénoté par une expression régulière est local ou non.

correction

On propose la méthode suivante :

- Calculer un automate A reconnaissant $(P(L)\Sigma^* \cap \Sigma^*D(L)) \setminus (\Sigma^*N(L)\Sigma^*)$ via la question 8.
- Calculer un automate B reconnaissant $L \setminus \{\varepsilon\}$ à partir de l'expression régulière dénotant L , par exemple à l'aide de l'algorithme de Berry-Sethi.
- Calculer l'automate $A\Delta B$ reconnaissant la différence symétrique de ces deux langages (la différence symétrique se construisant à partir d'unions, intersections et complémentaires, opérations qu'on sait faire sur des automates finis).
- À l'aide d'un parcours, déterminer si un état final de $A\Delta B$ est accessible depuis un de ses états initiaux. Si non, le langage reconnu par $A\Delta B$ est vide donc les langages reconnus par A et B sont égaux donc L est local. Si oui, le langage reconnu par $A\Delta B$ n'est pas vide donc les langages reconnus par A et B sont différents et L n'est pas local.

Le jury s'attend à ce que l'égalité de ces deux langages soit vérifiée par le biais d'automates les reconnaissant, à nouveau par une description de haut niveau.

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?

correction

Si e est une expression régulière telle que $L(e) \neq \emptyset$, il existe une expression régulière e' ne contenant pas le symbole \emptyset telle que $L(e') = L(e)$. Cela se montre aisément par induction sur e .

Si on autorise les expressions régulières contenant \emptyset , et qu'on souhaite à partir de e déterminer si $L(e)$ est local, on pourrait donc :

- Déterminer si $L(e)$ est vide à l'aide d'un automate reconnaissant ce langage.
- S'il est vide, il est local. Sinon, on peut inductivement transformer e en e' telle que $L(e) = L(e')$ et e' ne contient pas le symbole \emptyset et on applique l'algorithme de la question 9 à e' .

Un code source `localite_corrige.ml` est aussi disponible.

Là encore une justification haut niveau du fait qu'on peut se ramener à une expression régulière ne contenant pas le symbole vide suffit (on ne demande par exemple pas une preuve par induction de ce fait, simplement le fait qu'on pourrait le faire).

Chapitre 19

(CCINP) Programmation dynamique pour la récolte de fleurs * (CCINP 23, ex B, corrigé — 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.

correction

Correction proposée par le jury

Un code source `bouquet_corrige.c` est aussi disponible.

On trouve 11 fleurs.

On attend une réponse orale ; si elle est correcte, le jury ne demande même pas de justification.

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) .

correction

La récolte en une case dépend récursivement de celle de la case du haut et de celle de la case de gauche. La formule générique est donc à adapter lorsqu'on se trouve sur le bord gauche ou le bord haut du champ.

```

1 int recolte(int champ[m][n], int i, int j){
2     /* Cas de base */
3     if ( (i == 0) && (j == 0) )
4         return champ[0][0];
5     if (i == 0)
6         return champ[0][j] + recolte(champ, 0, j-1);
7     if (j == 0)
8         return champ[i][0] + recolte(champ, i-1, 0);
9     /* Cas général */
10    return champ[i][j] + max(recolte(champ, i-1, j), recolte(champ, i, j-1));
11 }

```

Il est attendu des candidats qu'ils soient attentifs aux cas de base. Ils peuvent factoriser la présentation de leur code et l'établissement de la relation de récurrence.

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).

correction

On trouve 6 appels à `recolte`.

Plusieurs méthodes sont acceptées pour répondre à cette question ; le candidat peut par exemple compter le nombre d'appels en modifiant la fonction précédente ou à la main en déployant l'arbre d'appels.

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 ?

correction

On calcule d'abord les valeurs des cases sur les bords haut et gauche puis on propage soit en remplissant les lignes de gauche à droite, soit en remplissant les colonnes de haut en bas.

Une réponse orale claire peut suffire. Le candidat peut aussi s'aider d'un schéma indiquant les dépendances entre les différents termes à calculer.

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) .

correction

Il s'agit d'une traduction des questions 2 et 4. Le code ci-dessous corrige aussi la question 7.

```

1
2 int recolte_iterative(int champ[m][n], int i, int j, int fleurs[m][n]){
3     int x, y;
4     fleurs[0][0] = champ[0][0];
5     /* Bord haut */
6     for (x = 1; x <= i; x++) {
7         fleurs[x][0] = champ[x][0] + fleurs[x-1][0];
8     }
9     /* Bord gauche */
10    for (y = 1; y <= j; y++) {
11        fleurs[0][y] = champ[0][y] + fleurs[0][y-1];
12    }
13    /* Autres cases */
14    for (y = 1; y <= j; y++) {
15        for (x = 1; x <= i; x++) {
16            fleurs[x][y] = champ[x][y] + max(fleurs[x-1][y], fleurs[x][y-1]);
17        }
18    }
19
20    deplacements(fleurs, i, j);
21    return fleurs[i][j];
22 }

```

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 deplacements(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) .

correction

On commence par distinguer les cas limites. Puis on appelle récursivement `deplacements` en observant quelle est la case voisine qui avait permis d'obtenir le plus de fleurs.

```

1 void deplacements(int fleurs[m][n], int i, int j){
2     if (i == 0 && j == 0) {
3         printf("Case A, ");
4         return;
5     }
6     if (i == 0) {
7         deplacements(fleurs, 0, j-1);
8         printf("Aller à droite, ");
9         return;
10    }
11    if (j == 0) {
12        deplacements(fleurs, i-1, 0);
13        printf("Descendre, ");
14        return;
15    }
16    if (fleurs[i-1][j] > fleurs[i][j-1]) {
17        deplacements(fleurs, i-1, j);
18        printf("Descendre, ");
19    }
20    else {
21        deplacements(fleurs, i, j-1);
22        printf("Aller à droite, ");
23    }
24 }

```

On attend bien l'affichage des déplacements, et pas uniquement leur calcul. La mise en forme de l'affichage (avec sauts de lignes, tabulations, ...) est en revanche libre.

7. Insérer un appel de `deplacements` dans la fonction `recolte_iterative` pour afficher le chemin parcouru.

correction

Voir le code proposé à la question 5.

Le jury est attentif à l'endroit où l'appel à cette fonction est effectué.

Chapitre 20

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

*

Représentation des nombres, C,
sources : `ccinpfloissants.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 ?

correction

Correction de M.Péchaud

```
1 printf("(a+b)+c=%f\n a+(b+c)=%f", (a+b)+c, a+(b+c));
```

Les résultats obtenus sont «très» différents : on obtient respectivement 1. et 0..

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 ϵ un réel appelé ϵ machine. On admet que ϵ 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.

correction

```
1 int epsilon(){
2     int n = 0;
3     while (1 + pow(2, -n) != 1)
4         n++;
5     return n - 1;
6 }
```

Ce code part du principe que l'arrondi sera fait par troncature de la mantisse.

On obtient 52 – ce qui est cohérent avec la taille de la mantisse – le plus petit réel > 1 représentable à pour

représentation binaire $1.00\dots 1$ avec 51 «0», c'est donc bien $1 + 2^{-52}$.

Le jury attend une brève explication du comportement constaté.

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.

correction

```
1 double u(int n){
2     if (n == 0) return 2;
3     double uad = 2; //avant-dernier terme calculé
4     double ud = -4; //dernier terme calculé
5     for (int i = 2; i <= n; i++) {
6         double tmp = 111 - 1130/ud + 3000/(ud * uad);
7         uad = ud;
8         ud = tmp;
9     }
10    return ud;
11 }
```

Une fonction récursive naïve suffit à obtenir tous les points à cette question mais le candidat est bien entendu libre de stocker les valeurs de la suite demandée pour éviter d'avoir à les recalculer.

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 ?

correction

```
1     for (int i = 0; i <= 21; i++)
2         printf("u%d = %f\n", i, u(i));
```

La suite semble tendre vers 100.

La question ne demande pas de justifier la limite théorique. Le jury doit voir apparaître les premiers termes correctement et clairement affichés.

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`.

correction

```
1 double somme(struct nb* tab){
2     if (tab->suivant == NULL) return tab->x;
3     return tab->x + somme(tab->suivant);
4 }
```

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.

correction

```

1      struct nb* s;
2      //Création du maillon s contenant la somme des 2 premiers é↔
      éléments
3      s = malloc(sizeof(struct nb));
4  s -> x = tab -> x + tab -> suivant -> x;
5  s -> suivant = NULL;
6      //temp pointe vers le 3e élément de la liste
7  temp=(temp->suivant)->suivant;
8      //recherche de l'emplacement de s
9
10     struct nb* t=temp;
11     struct nb* t2=temp;
12
13     while(t!=NULL){
14         /*si la valeur pointée par t
15         est supérieure strictement à celle de s,
16         t pointe sur l'élément suivant
17         sinon on quitte la boucle
18         */
19         if (t -> x < s -> x)
20     {
21         t = t -> suivant;
22         t2 = t;
23     }
24         else break;
25     }

```

À cette question, un candidat qui aurait fait le choix d'implémenter `somme2` directement et sans utiliser le code déjà fourni et en expliquant sa démarche aurait eu tous les points.

- La fonction proposée ne prend pas en compte un cas d'insertion. Illustrer ce propos.

correction

Il y a un problème lorsque $t -> x \geq s -> x$ à la première itération – i.e. lorsque l'insertion doit se faire au début – car alors $t = t2$.

Chapitre 21

(CCINP) chemins simples sans issue * (CCINP 23, ex B, corrigé — 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.

correction

Corrigé proposé par le jury

```
let est_sommet g a = (0 <= a) && (a < Array.length g)
```

Le candidat ne doit pas oublier de tester la positivité de l'entier en entrée.

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.

correction

```
let rec appartient liste a = match liste with
| [] -> false
| b :: suite -> (b = a) || (appartient suite a)
```

Le jury s'attend à une solution récursive. Dans cet exercice une approche impérative n'est néanmoins pas pénalisée.

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`.

correction

```
let rec est_chemin g liste = match liste with
| [] -> true
| [a] -> est_sommet g a
| a :: b :: suite -> (appartient (g.(a)) b) && (est_chemin g (b ←
:: suite))
```

Même remarque que pour la question précédente.

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.

correction

```
let est_chemin_simple_sans_issue g liste =
let n = Array.length g in
let visites = Array.make n false in
let rec test_aux liste = match liste with
| [] -> false
| [a] -> [] = (List.filter (fun x -> not visites.(x)) (g.(a)))
| a :: b :: suite ->
begin
visites.(a) <- true ;
(appartient g.(a) b) && (not visites.(b))
&& (test_aux (b :: suite))
end
in
test_aux liste
```

Plusieurs variantes de réponses à cette question sont possibles.

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.

correction

```
let genere_chemins_simples_sans_issue (g : graphe) =
let taille = Array.length g in
(*garde en mémoire les chemins déjà trouvés*)
let liste_chemins = ref [] in
(*garde en mémoire les sommets en cours de visite *)
let visites = Array.make taille false in
```

```

(*garde en mémoire le début d'un chemin*)
let chemin_courant_envers = ref [] in

(*trouve tous les chemins simples sans issue commençant par s*)
let rec profondeur s =
  if not visites.(s) then begin
    visites.(s) <- true ;
    chemin_courant_envers := s :: (!chemin_courant_envers) ;
    let voisins_libres =
      List.filter (fun x -> not visites.(x)) g.(s)
    in
    if voisins_libres = [] then begin
      liste_chemins := (List.rev !chemin_courant_envers) :: (!←
        liste_chemins)
    end else begin
      List.iter profondeur voisins_libres
    end ;
    (*pour revenir en arrière *)
    visites.(s) <- false ;
    chemin_courant_envers := List.tl !chemin_courant_envers ;
  end
in
for i = 0 to (taille-1) do
  profondeur i
done ;
!liste_chemins

```

Le jury s'attend à ce que les ajouts dans une liste soient faits en tête, quitte à renverser la liste à la fin.

6. Écrire des expressions donnant les listes des chemins simples pour les deux graphes G_1 et G_2 .

correction

```

let liste1 = genere_chemins_simples_sans_issue g1
let liste2 = genere_chemins_simples_sans_issue g2

```

Le jury souhaite voir le résultat des tests demandés dans cette question : les candidats qui souhaitent compiler leur code doivent donc coder des fonctions d'affichage pertinentes pour obtenir tous les points ici.

7. Expliciter la complexité des fonctions `appartient` et `est_chemin_simple_sans_issue`.

correction

La complexité de `appartient` est en $O(|l|)$: en effet, on effectue un parcours de liste.

On note A l'ensemble des arêtes du graphe en argument. La complexité de `est_chemin_sans_issue` est en $O(|l| + |A|)$. En effet, pour chaque sommet s de la liste l , on effectue en terme de calcul de l'ordre de $1 + |g.(s)|$. En sommant sur tous les éléments de l (dans le cas où il serait tous distincts), on trouve une complexité en $O(|l| + |A|)$. Si un même sommet apparaît deux fois, le calcul est interrompu lors de la visite d'une première répétition et on retrouve la même complexité.

Le jury attend une borne précise sur ces complexités. Si le candidat propose une borne en $\mathcal{O}(|l||A|)$ pour `est_chemin_simple_sans_issue`, il est invité à l'affiner.

Chapitre 22

(CCINP) Grammaires pour des langages de programmation * (CCINP 24, ex A, corrigé, à déboguer — 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 ?

correction

Trivialement, $\emptyset \rightarrow$ question bugguée.

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} .

correction

$$\begin{aligned} S &\rightarrow PN \\ P &\rightarrow a|\dots|Z|_ \\ N &\rightarrow aN|\dots|zN|AN|\dots|ZN|0N|\dots|9N|_N|\epsilon \end{aligned}$$

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.

correction

Immédiat par induction sur les arbres de dérivation.

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

correction

$$S \rightarrow I|IS$$
$$B \rightarrow \{S\}$$

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

correction

E n'est pas défini précisément par l'énoncé ? !

$\mathcal{L}(E) \cap (*)^* = \{(n)^n, n \in \mathbb{N}\}$, qui est l'exemple le plus classique de langage non régulier.

Par contraposée, $\mathcal{L}(E)$ n'est pas régulier.

Chapitre 23

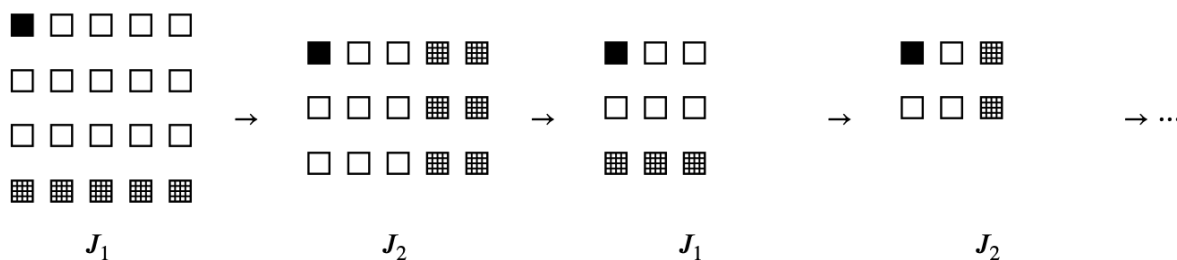
(CCINP) chomp * (CCINP 24, ex A, corrigé — oral - 228 lignes)

*

Jeux,
sources : ccinpchomp.tex

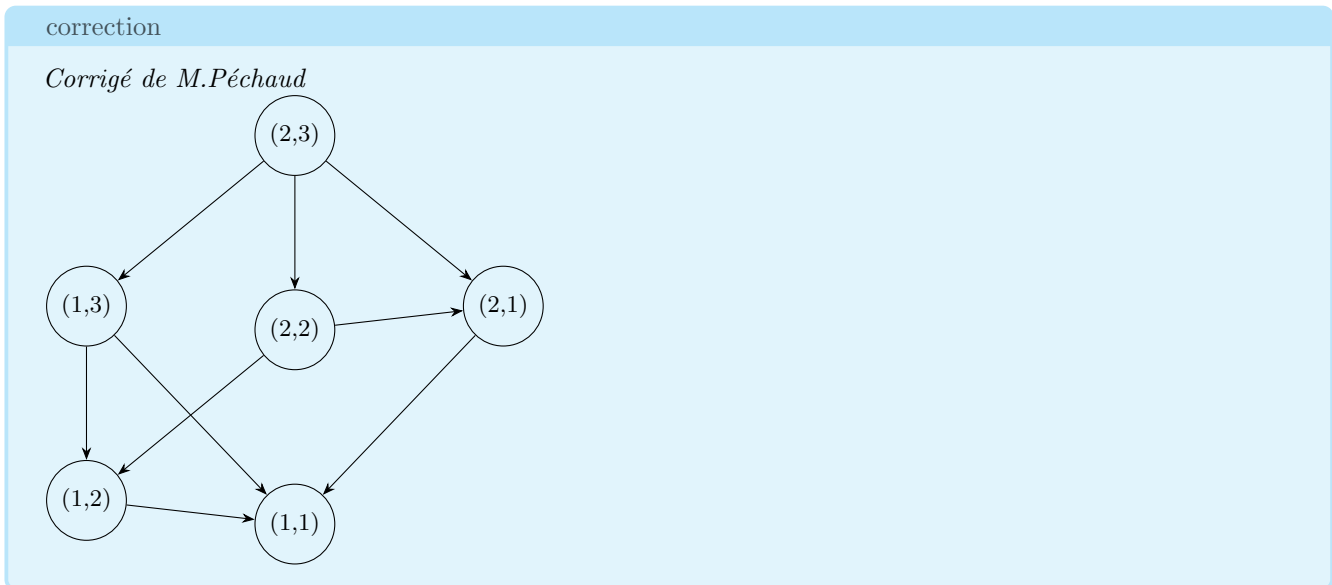
On considère une variante du jeu de Chomp : deux joueurs J_1 et J_2 s'affrontent autour d'une tablette de chocolat de taille $l \times c$, dont le carré en haut à gauche est empoisonné. Les joueurs choisissent chacun leur tour une ou plusieurs lignes (ou une ou plusieurs colonnes) partant du bas (respectivement 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 4×5 , 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) .



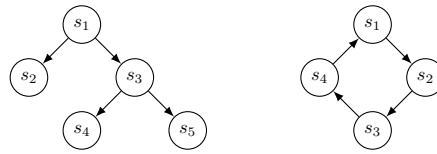
On va chercher à obtenir une stratégie gagnante pour le joueur J_1 par deux manières.

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 :



correction

Pour le premier, un seul noyau possible : $\{s_2, s_4, s_5\}$.
 Pour le second, deux noyaux possibles : $\{s_1, s_3\}$ et $\{s_2, s_4\}$.

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.

correction

On considère un chemin maximal. Son dernier sommet est nécessairement un puits, sans quoi on contredit la maximalité ou l'acyclicité.

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 ∈ S sans ←
      successeur
4    N = N ∪ {s}
5    Supprimer s de G ainsi que ses préd←
      écesseurs
  
```

4. Justifier que cet algorithme termine et renvoie un noyau.

correction

Supprimer des sommets d'un graphe acyclique le laisse acyclique donc la question 3 assure qu'on trouvera toujours un sommet s convenable en ligne 3. Par conséquent, le nombre de sommets dans G décroît strictement à chaque itération ce qui assure la terminaison de l'algorithme.

Par ailleurs, un sommet sans successeur doit être dans un noyau à cause de la deuxième propriété et donc tous ses prédécesseurs doivent être en dehors à cause de la première. Ces deux contraintes nécessaires sont garanties par l'algorithme. Elles sont suffisantes dans le cadre d'un graphe acyclique car tout sommet qui devrait nécessairement être dans un noyau et nécessairement ne pas y être ferait partie d'un cycle.

5. Démontrer que ce noyau est unique. Conclure que le graphe du jeu de Chomp possède un unique noyau N .

correction

Travaillons par récurrence forte sur le nombre n de sommets de G . Si $n = 1$ ce seul sommet (puits) constitue le noyau.

Sinon, soit s un puits de G (qui existe d'après la question 3). Ce sommet fait nécessairement partie de tous les noyaux de G . On note $P(s)$ l'ensemble des prédécesseurs de s . Le graphe G privé de s et des sommets de $P(s)$ reste acyclique et donc, par hypothèse de récurrence, a un noyau unique N . L'ensemble $N \cup \{s\}$ est un noyau de G et par unicité de N et nécessité du fait que s soit dans tout noyau, c'est le seul.

Le jeu de Chomp a un graphe acyclique car le nombre de carrés décroît strictement à chaque coup, il a donc un unique noyau.

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.

correction

Les sommets (2,2) et (1,1) constituent le noyau. La position (1,1) est perdante pour le joueur qui doit jouer (il mange le chocolat empoisonné), les éléments du noyau sont les sommets perdants et les autres sommets les positions gagnantes.

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 $c = 3$?

correction

On montre qu'un joueur qui peut choisir s_1 dans le noyau ne peut pas perdre. Si s_1 n'a pas de successeur, l'adversaire ne peut plus jouer, il a perdu. Sinon, l'adversaire va choisir s_2 dans les successeurs de s_1 . On a donc $s_2 \in S \setminus N$ donc s_2 admet au moins un successeur dans N .

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 ?

correction

\mathcal{A}_i est l'ensemble des sommets de S à partir desquels J_1 peut forcer la partie à arriver en F_1 en moins de i coups.

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} ?

correction

$\mathcal{A}_0 = \{S_6^2\}$, $\mathcal{A}_1 = \{S_6^2, S_2^1, S_4^1\} = \mathcal{A}$. L'attracteur contient exactement toutes les positions gagnantes pour J_1 .

Chapitre 24

(CCINP) une relation d'équivalence * (CCINP 24, ex A, corrigé — oral - 138 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 (φ, ψ) -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 (φ, ψ) -équivalent" est une relation d'équivalence sur l'ensemble $\llbracket 1, n \rrbracket$.
2. Pour cette question, on considère les applications φ et ψ 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 (φ, ψ) -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 φ et ψ de $\llbracket 1, n \rrbracket$ où tout élément de l'image de φ admet exactement deux antécédents par φ et où tout élément de l'image de ψ admet exactement deux antécédents par ψ .

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.

correction

Corrigé proposé par le jury

Proposition de corrigé

1. La réflexivité correspond au cas $k = 0$, la symétrie consiste à réindexer à l'envers, la transitivité consiste à mettre bout à bout les deux séquences.
2. On trouve comme classes : $C_1 = \{1, 5\}, C_2 = \{2, 3, 6\}, C_3 = \{4, 7, 8, 9\}$.
3. Il existe alors un chemin commençant par x et terminant par y . Sachant que le graphe est en réalité symétrique, calculer les classes d'équivalence revient alors à calculer les composantes connexes du graphe correspondant (on peut aussi calculer les composantes fortement connexes si on n'a pas remarqué la symétrie).
4. Dans le cas où on remarque la symétrie : un parcours du graphe suffit (n'importe lequel).

```

1      \SetKwFunction{comp}{Composantes\_connexes}
2      \SetKwFunction{parcours}{parcours\_largeur}
3      Entrée : graphe G
4      Sortie : tableau numComp avec numComp[i] égal au numéro d'↔
           une composante connexe
5      numComp = [0, 0, ..., 0] (indexation de 1 à n inclus)
6      parcours(i,num)
7      file = [i]
8      TantQue file non vide
9          a = defiler(file)
10         Si numComp[a] vaut 0
11             numComp[a] := num
12             Pour tout x voisin de a
13                 enfiler(x)
14     num = 0
15     Pour i = 1 à n
16         Si numComp[i] = 0
17             num := num + 1
18             parcours(i,num)
19
20     Renvoyer numComp

```

Dans le cas où la symétrie n'est pas remarquée, on peut utiliser un algorithme de calcul des composantes fortement connexes, comme l'algorithme de Kosaraju.

5. On reconnaît un graphe biparti où chaque sommet est de degré 1 (c'est le graphe induit par un couplage parfait).
6. On obtient deux types de composantes connexes :
 - composantes à deux sommets (même image par φ et ψ)
 - des cycles ayant un nombre pair de sommets (on peut remarquer qu'il y a alternance entre arêtes créées par φ et créées par ψ).

On pourra remarquer qu'un sommet est soit d'arité 1, soit d'arité 2. Lorsqu'il est d'arité 1, cela signifie que son voisin et lui ont même image par φ et ψ . Donc ces deux sommets constituent une composante connexe. Dans le cas où un sommet est d'arité 2, par la contraposition de la remarque précédente, tous les sommets dans sa composante connexe sont d'arité 2. Ainsi, on a uniquement des cycles ayant un nombre pair de sommets.

Chapitre 25

(CCINP) Fonctions pour la détection de motifs * (CCINP 24, ex B, corrigé — oral - 130 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 Σ^* , 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 σ^* , y compris le mot vide.
1. Déterminer lesquels de ces mots sont capturés par le motif $m_0 = *ab*a?$.
 - (a) *aabbaba*
 - (b) *aabbaab*
 - (c) *bbbaab*
 - (d) *bbabaab*Décrire en français les mots capturés par ce motif.

correction

Corrigé de M.Péchaud

Les mots *b* et *d*.

Ce sont les mots contenant un facteur *ab* ailleurs qu'en dernière position, et dont l'avant-dernière lettre est un *a*.

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.

correction

```
1 let string_to_chaine s =
2   let rec aux i =
3     if i = String.length s then []
4     else s.[i] :: aux (i+1)
5   in aux 0
```

3. Expliquer pourquoi, dans notre étude, il est inutile d'avoir plusieurs caractères $*$ consécutifs.

correction

Le motif $**$ est équivalent au motif $*$.

4. Écrire une fonction `purge : chaine -> chaine` permettant de supprimer les $*$ en trop dans une chaîne.

correction

```

1 let rec purge l = match l with
2   [] | [x] -> c
3   | '*' :: '*' :: q -> purge ('*' :: q)
4   | t :: q -> t :: (purge q)

```

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.

correction

Les lignes traduisent les faits que

- la chaîne vide est capturée par le motif vide
- la chaîne vide est capturée par le motif *
- aucun motif non vide autre que * ne reconnaît la chaîne vide (car on a supprimé les * multiples avec le `purge`)
- une chaîne non vide n'est pas capturée par le motif vide

6. Compléter la fonction `capture`.

correction

```

1 let rec capture motif texte = match (motif, texte) with
2   | [], [] -> true
3   | ['*'], [] -> true
4   | _, [] -> false
5   | [], _ -> false
6   | '?' :: q, t' :: q' -> capture q q'
7   | t :: q, t' :: q' -> t = t' && capture q q'
8   | _ -> false

```

Chapitre 26

(CCINP) HORNSAT * (CCINP 24, ex B, corrigé — 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 ?

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

correction

Corrigé proposé par le jury

F_1 et F_3 sont des formules de Horn mais pas F_2 , ce qui est implicitement suggéré par le code.

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 correspondant 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).

correction

Avec notre modélisation, la clause vide est `(None, [])` d'où :

```
let avoir_clause_vide (f : formule_horn) : bool =
  List.mem (None, []) f
```

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.

correction

La formule F_1 est satisfiable d'après cet algorithme mais pas F_3 .
Techniquement on peut arrêter les propagations dès qu'on produit une clause vide.

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.

correction

```

let rec trouver_clause_unitaire (f : formule_horn) : int option = ←
  match f with
  | [] -> None
  | (Some v, l) : q -> if l = [] then Some v else ←
    trouver_clause_unitaire q
  | _ : q -> trouver_clause_unitaire q

```

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 .

correction

La fonction auxiliaire `enlever_neg` supprime si elle existe la seule occurrence d'un entier dans une liste d'entiers (on utilise ici l'hypothèse selon laquelle il n'y a pas de littéral en doublon dans nos clauses de Horn) puis on applique le principe décrit par l'énoncé.

```

let rec propager (f : formule_horn) (v : int) : formule_horn =
  let rec enlever_neg (l : int list) : int list = match l with
  | [] -> []
  | t : q when t = v -> q
  | t : q -> t : (enlever_neg q)
  in
  match f with
  | [] -> []
  | (i, l) : q -> match i with
  | Some t when t = v -> propager q v
  | _ -> (i, enlever_neg l) : (propager q v)

```

La formule résultant d'une propagation dans une formule de Horn reste une formule de Horn : c'est toujours une FNC et on ne fait que supprimer des clauses / littéraux (donc s'il y avait au plus un littéral positif par clause, en supprimant des choses cette propriété est conservée).

6. Dédurre des questions précédentes une fonction `etre_satisfiable : formule_horn -> bool` renvoyant `true` si et seulement si la formule de Horn en entrée est satisfiable.

correction

On applique l'algorithme donné par l'énoncé en imbriquant les fonctions précédentes :

```

let rec etre_satisfiable (f : formule_horn) : bool =
  match (trouver_clause_unitaire f) with
  | None -> not (avoir_clause_vide f)
  | Some v -> etre_satisfiable (propager f v)

```

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)?

correction

La fonction `trouver_clause_unitaire` est linéaire en n la taille de la formule en entrée. Une propagation se fait aussi linéairement en la taille de la formule dans laquelle on propage. Comme une propagation supprime au moins une clause de la formule (la clause unitaire responsable de la propagation), on fera au plus m étapes de "recherche de clause unitaire + propagation" où m est le nombre de clauses.

De plus, la recherche d'une clause vide dans une formule se fait linéairement en le nombre de clauses. On aboutit donc grossièrement (c'est-à-dire sans compter qu'au fil des propagations la taille de la formule réduit - de toutes façons ça ne changerait probablement pas grand chose) à une complexité pour `etre_satisfiable` en $O(mn + m) = O(n^2)$.

On en déduit que HORN-SAT $\in P$ alors qu'on sait que SAT est NP-complet. Sous réserve que $P \neq NP$, étudier la satisfiabilité de formules de Horn est donc un problème bien plus facile que lorsqu'il n'y a pas d'hypothèse sur les formules.

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 .

correction

La valuation qui assigne faux à toutes les variables convient. En effet, s'il n'y a ni clause unitaire ni clause vide, toutes les clauses contiennent au moins une variable niée.

- 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} .

correction

Si F est une formule de Horn, notons $\Pi(F)$ la formule de Horn obtenue après toutes les propagations successives de la boucle tant que dans l'algorithme \mathcal{A} . La propriété de l'énoncé permet de montrer par récurrence que F et $\Pi(F)$ sont équivalents.

Or $\Pi(F)$ est satisfiable si et seulement si cette formule de Horn ne contient pas la clause vide : le sens direct est immédiat puisque une FNC contenant une clause vide n'est jamais satisfiable (cette clause ne l'étant pas); le sens réciproque est fourni par la question 7a.

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.

correction

Le principe est de garder trace des variables que l'on propage : on les assigne toutes à vrai. Toutes les variables non propagées sont quant à elles assignées à faux.

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.

correction

Correction proposée par le jury

Proposition de corrigé

1. Une solution est d'implémenter une factorielle. On peut aussi simplifier la formule donnée (ce qui permet de repousser un peu plus loin le dépassement).

```

1  uint64_t fact(int n) {
2      uint64_t r = 1;
3      for(int i = 2; i <= n; i = i + 1){
4          r = r * i;
5      }
6      return r;
7  }
8
9  uint64_t catalan1(int n) {
10     return fact(2*n)/fact(n+1)/fact(n);
11 }

```

2. On obtient un dépassement d'entier. Expérimentalement, si on utilise la fonction factorielle, le résultat devient faux pour $n = 11$. En simplifiant la formule en $2n \times \dots \times (n+2)/n!$, c'est pour $n = 16$ que ça coince. Dans tous les cas, le dépassement finira par arriver provoquant un comportement indéfini.
3. On utilise un compteur qui indique le nombre de parenthèses ouvertes : on l'incrémente lorsqu'on rencontre une parenthèse ouvrante et on le décrémente lorsqu'on rencontre une parenthèse fermante. Si ce compteur devient négatif, le mot n'est pas bien parenthésé puisqu'il y a trop de parenthèses fermantes et on peut donc interrompre l'exécution (même si on peut se passer de cette subtilité). En fin de décompte, le compteur devrait être nul si le mot est bien parenthésé.

```

1  bool verification(char * mot) {
2      int i = 0;
3      bool fin = false;
4      int compteur = 0;
5      bool resultat = false;
6      while (!fin && (mot[i] != '\0')) {
7          if (mot[i]=='(') {
8              compteur++;
9          }
10         else if (mot[i] == ')') {
11             compteur = compteur - 1;
12         }
13         if (compteur<0) {
14             fin = true;
15         }
16         i = i + 1;
17     }
18     if (compteur == 0) {
19         resultat = true;
20     }
21     return resultat;
22 }

```

4. On obtient un algorithme linéaire en la taille du mot en entrée.
5. Il y a 4^n mots à générer (puisque un mot à n couples de parenthèses possède $2n$ lettres). Pour chacun, la vérification de s'il est bien formé se fait en $O(2n) = O(n)$. À supposer que la construction des mots ne soit pas coûteuse, on obtient une complexité pour l'algorithme naïf en $O(4^n \times n)$.
6. Voici une proposition qui répond aux questions 6, 7 et 8.

```

1 void dyck(char s[N], int o, int f, int n, uint64_t* r,
2 struct liste_s ** liste) {
3     if (f == n) {
4         if (liste != NULL) {
5             struct liste_s * c = malloc(sizeof(struct liste_s));
6             strcpy(c->chaine, s);
7             c->suivant = *liste;
8             *liste = c;
9         }
10        else {
11            printf("%s\n", s);
12        }
13        (*r) = (*r) + 1;
14        return;
15    }
16    if (o < n) {
17        // printf("on ajoute (\n");
18        char res[N];
19        strcpy(res, s);
20        strcat(res, "(");
21        dyck(res, o+1, f, n, r, liste);
22    }
23    if (f < o) {
24        // printf("on ajoute )\n");
25        char res[N];
26        strcpy(res, s);
27        strcat(res, ")");
28        dyck(res, o, f+1, n, r, liste);
29    }
30 }

```

7. Une façon de faire est d'ajouter un paramètre passé par pointeur pour compter ce nombre. On pourrait aussi utiliser une variable globale. On trouve 35357670 mots corrects pour $n = 16$.
8. Voir la proposition en question 6.

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 ?

correction

Corrigé de M. Péchaud

On obtient

a	$a \downarrow a$
0	1
1	0

et c'est donc équivalent à \neg .

2. Montrer que $(a \wedge b) \equiv (a \downarrow a) \downarrow (b \downarrow b)$.

correction

D'après la question précédente, le membre de droite est équivalent à $(\neg a) \downarrow (\neg b)$.

On obtient donc la table de vérité suivante :

a	b	$\neg a$	$\neg b$	$(\neg a) \downarrow (\neg b)$
0	0	1	1	0
1	0	0	1	0
0	1	1	0	0
1	1	0	0	1

dont la dernière colonne est bien celle de $a \wedge b$.

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 raisonne par induction.

correction

D'après les deux questions précédentes, $a \vee b \equiv \neg((\neg a) \wedge (\neg b))$ peut s'écrire uniquement à l'aide de \downarrow .
Montrons le résultat voulu par induction sur la structure de \mathcal{F} .

- Si $f = \perp$, $f = \top$ où $f \in \mathcal{V}$, f ne contient aucun opérateur, donc le résultat est trivialement vrai.
- Si $f = \neg f'$, où f' vérifie la propriété, on note \tilde{f}' une formule n'utilisant que \downarrow équivalente à f' . On a alors $f \equiv \tilde{f}' \downarrow \tilde{f}' \equiv \tilde{f}' \downarrow \tilde{f}'$ qui n'utilise que \downarrow .
- Si $f = f_1 \wedge f_2$, où f_1 et f_2 vérifient la propriété, avec les mêmes notations que précédemment on obtient $f \equiv (f_1 \downarrow f_1) \downarrow (f_2 \downarrow f_2) \equiv (\tilde{f}_1 \downarrow \tilde{f}_1) \downarrow (\tilde{f}_2 \downarrow \tilde{f}_2)$ qui n'utilise que \downarrow .
- De même pour \vee d'après la remarque en début de réponse.

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.

correction

Notons $n = |\mathcal{V}|$.

- \perp est satisfaite par 0 valuation, et 0 est pair.
- \top est satisfaite par 2^n valuations, et 2^n est pair (car $n \neq 0$).
- v est satisfaite par 2^{n-1} valuations (celles qui évaluent v à 1), et 2^{n-1} est pair (car $n - 1 \neq 0$).

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.

correction

Par hypothèse, $n_{0,1} + n_{1,1}$ est pair (c'est le nombre de valuations satisfaisant H), et de même $n_{1,0} + n_{1,1}$ est pair.

Donc $n_{0,1}$, $n_{1,1}$, et $n_{1,0}$ ont la même parité.

Comme $n_{0,1} + n_{1,0} + n_{0,0} + n_{1,1} = 2^n$, $n_{0,0}$ a également la même parité.

Or F satisfait $n_{0,0} + n_{1,1}$ valuations, donc un nombre pair de valuations.

6. $\{\odot\}$ est-il complet ?

correction

Non, car par exemple $v_1 \wedge \dots \wedge v_n$ (où $\mathcal{V} = \{v_1 \dots v_n\}$) n'est satisfaite par une unique valuation, et par une induction immédiate utilisant la question précédente, toute formule n'utilisant que \odot doit être satisfaite par un nombre pair de valuations.

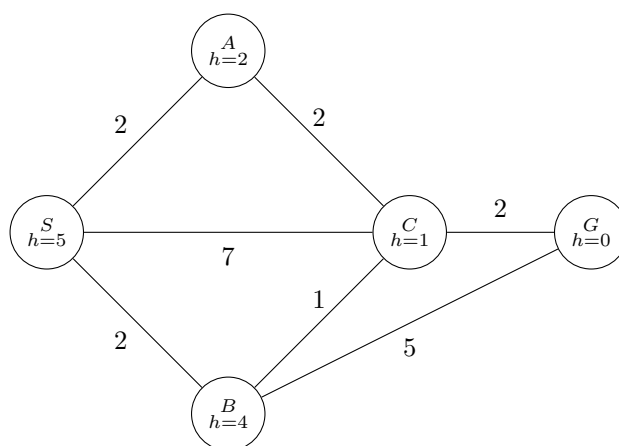
Chapitre 29

(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 ?

correction

- On défile S , on enfile les sommets suivants avec les priorités indiquées : $(A, 2 + 2 = 4)$, $(B, 2 + 4 = 6)$, $(C, 7 + 1 = 8)$.
- On défile A . On met à jour C avec une nouvelle priorité $2 + 2 + 1 = 5$.
- On défile C . On enfile G avec priorité $4 + 1 = 5$.
- On défile G , et l'algorithme termine.

Le chemin trouvé est $S - A - C - G$ – de longueur 6, ce qui n'est pas optimal car $S - B - C - G$ est de longueur 5.

2. Que ce serait-il passé si on avait utilisé Dijkstra ? Il n'est pas demandé d'appliquer l'algorithme.

correction

L'algorithme de Dijkstra aurait trouvé le plus court chemin.

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.

correction

C'est une conséquence directe

4. L'heuristique h est-elle admissible ? monotone ?

correction

L'heuristique n'est pas admissible, car $h(B) = 4 > 3 = d(B, G)$, ni monotone car $h(B) = 4 > 1 + 1 = l(BC) + h(C)$.

5. L'algorithme A^* donne-t-il toujours le bon résultat avec l'heuristique h_2 ?

correction

Oui, l'algorithme est correct si l'heuristique est monotone, ce qui est le cas de 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 ?

correction

h_1 étant une norme, la distance associée vérifie l'inégalité triangulaire, donc est monotone. Donc A^* est correct en utilisant cette heuristique.

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.

correction

```
1 let identifiant i j =  
2   n*i+j
```

2. La réciproque `coordonnees : int -> (int*int)`.

correction

```
1 let coordonnees c =  
2   (c/n, c mod n)
```

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.

correction

```

1 let legal p l c =
2   p.(l).(c) == 0

```

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.

correction

```

1 let copy_matrix m =
2   let r = Array.make_matrix n n 0 in
3   for i = 0 to n-1 do
4     for j = 0 to m-1 do
5       r.(i).(j) ← m.(i).(j)
6     done;
7   done;
8   r

```

```

1 let joue p l c j =
2   if not (legal p l c) then raise CoupIllégal;
3   let r = copy_matrix p in
4   r.(l).(c) ← j;
5   r

```

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).

correction

```

let victoire_ligne p i =
  if p.(i).(0) = 0 then None (*1ère case vide*)
  else let j = ref 1 in
        while !j < n && p.(i).(j) = p.(i).(0) do
          incr j
        done;
        if !j = n then Some p.(i).(0) (*victoire*)
        else None
;;

let victoire_colonne p j =
  if p.(0).(j) = 0 then None (*1ère case vide*)
  else let i = ref 1 in
        while !i < n && p.(i).(j) = p.(0).(j) do
          incr i
        done;
        if !i = n then Some p.(0).(j) (*victoire*)
        else None
;;

let victoire p =
  let i = ref 0 in
  let r = ref (victoire_colonne p 0) in
  while !i < n && !r = None do
    r := victoire_colonne p !i;
    incr i
  done;

```

```

match !r with
  Some(j) -> Some(j)
| None ->
  i := 0;
  let r = ref (victoire_ligne p 0) in
  while !i < n && !r = None do
    r := victoire_ligne p !i;
    incr i
  done;
  !r
;;

```

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.

correction

```

let rec gagnante p j =
  match victoire p with
  | Some(joueur) -> Some(joueur) (*position finale gagante*)
  | None -> let possuiv = List.map (fun e -> let l, c = coordonnees e in
    gagnante (joue p l c j) (3-j)) (coups_possibles p) in (*calcul ré↔
    cursif du statut gagnant ou pas des positions accessibles en un ↔
    coup*)
    if possuiv = [] then None (*p est un état final nul*)
    else
      if j = 1 then
        if List.exists (fun e -> e = Some(1)) possuiv
        then Some(1)
        else if List.for_all (fun e -> e = Some(2)) possuiv
        then Some(2)
        else None
      else
        if List.exists (fun e -> e = Some(2)) possuiv
        then Some(2)
        else if List.for_all (fun e -> e = Some(1)) possuiv
        then Some(1)
        else None

```

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.

correction

```
1 let xor a b =
2   (a && not b) || (not a && b)
```

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 $e1 \cap e2 = \emptyset$.

correction

```
1 let bipartition e1 e2 =
2   let n = Array.length e1 in
3   let rec aux i =
4     i = n || (xor e1.(i) e2.(i) && aux (i+1))
5   in
6   aux 0
```

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.

correction

```

1 let bipartition2 g i v =
2   let rec aux l = match l with
3     [] -> true
4     | t :: q -> ((g.e1.(i) && g.e2.(t)) || (g.e2.(i) && g.e1.(t))) && ←
        aux q
5   in
6   aux v

```

4. Écrire une fonction `biparti : graphe_biparti -> bool` qui teste si le graphe passé en argument est biparti pour la partition de sommets $(e1, e2)$.

correction

```

1 let biparti g =
2   let n = Array.length g.graphe in
3   let i = ref 0 in
4   while !i < n && bipartition2 g !i g.graphe.(!i) do
5     incr i
6   done;
7   !i = n

```

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.

correction

```

1 let affecte_voisins g i =
2   let rec aux e autree l = match l with
3     [] -> []
4     | t :: q -> if e.(t) then raise NonBiparti
5                 else if autree.(t) then aux e autree q
6                 else (autree.(t) <- true; i : aux e autree q)
7   in
8   in
9   if g.e1.(i) then aux g.e1 g.e2 g.graphe.(i) else aux g.e2 g.e1 g.←
        graphe.(i)

```

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.

correction

```

1 let rec parcours g i =
2   let lv = affecte_voisins g i in (*lv contient les voisins non encore ←
3     vus*)
4   let rec aux l = match l with (*parcours récursif*)
5     [] -> ()
6     | t :: q -> parcours g t; aux q
7   in aux lv

```

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.

correction

```
1 let calcule_bipartition g =
2   let n = Array.length g in
3   let r = {graphe = g;
4             e1 = Array.make n false;
5             e2 = Array.make n false;
6           }
7   in
8   for i = 0 to n-1 do
9     if not r.e1.(i) && not r.e2.(i) then (*sommet pas encore affecte (↔
10      nouvelle composant connexe)*)
11      r.e1.(i) <- true; (*on affecte arbitrairement l'élément à e1*)
12      parcours r i;
13 done;
14 r
```


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 .

correction

```
1 void affiche_mot(char* mot, int n) {
2     for (int i = 0; i < n; i++) {
3         printf("%c", mot[i]);
4     }
5     printf("\n");
6 }
```

2. Écrire une fonction `void affiche_liste(liste lst)` qui affiche le contenu d'une liste de mots.

correction

```
1 void affiche_liste(liste lst) {
2     if (lst != NULL) {
3         affiche_mot(lst->mot, lst->n);
4         affiche_liste(lst->suite);
5     }
6 }
```

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.

correction

```

1 void ajoute(char a, liste lst) {
2     if (lst != NULL) {
3         lst->mot[lst->n] = a;
4         lst->n++;
5         ajoute(a, lst->suite);
6     }
7 }

```

4. Justifier que la fonction précédente a une complexité linéaire en le nombre de mots dans la liste.

correction

La complexité est immédiate, les opérations pour chaque maillon étant effectuées en temps constant (en l'absence de copies des chaînes).

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 ?

correction

Si on considère un mot de longueur n dont toutes les lettres sont distinctes, il y a 2^n sous-mots. Pour un mot de longueur n constitué de la même lettre répétée, il n'y a que $n + 1$ sous-mots.

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.

correction

```

1 liste copie_liste(liste l) {
2     if (l == NULL) return NULL;
3     liste tete = NULL;
4     liste fin = NULL;
5     while (l != NULL) {
6         liste nouveau = malloc(sizeof(struct maillon));
7         nouveau->n = l->n;
8         nouveau->mot = copie_mot(l->mot);
9         nouveau->suite = NULL;
10        if (tete == NULL) {
11            tete = nouveau;
12            fin = nouveau;
13        } else {
14            fin->suite = nouveau;
15            fin = nouveau;
16        }
17        l = l->suite;
18    }
19    return tete;
20 }

```

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.

correction

```

1 liste concat(liste l1, liste l2) {
2     if (l1 == NULL) return l2;
3     liste tmp = l1;
4     while (tmp->suite != NULL) {
5         tmp = tmp->suite;
6     }
7     tmp->suite = l2;
8     return l1;
9 }

```

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.

correction

```

1 liste sous_mots_aux(liste l, char* mot, int n, int i) {
2     if (i == n) return l;
3     liste lp = copie_liste(l);
4     ajoute(mot[i], lp);
5     l = concat(l, lp);
6     return sous_mots_aux(l, mot, n, i+1);
7 }
8
9 liste sous_mots(char* mot, int n) {
10    liste l = malloc(sizeof(struct maillon));
11    l->mot = malloc(NMAX * sizeof(char));
12    l->n = 0;
13    l->suite = NULL;
14    return sous_mots_aux(l, mot, n, 0);
15 }

```

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 Σ un alphabet et \leq un ordre sur Σ – 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.

correction

Notons arbitrairement a et b deux lettres distinctes de Σ .
Alors a et b ne sont pas comparables pour \sqsubseteq , qui n'est donc pas un ordre total.

2. Montrer que si \leq est un ordre non total, alors \preceq aussi.

correction

Notons arbitrairement a et b deux lettres distinctes de Σ , non comparables pour \leq .
Alors a et b ne sont pas comparables pour \preceq , qui n'est donc pas un ordre total.

3. Montrer que \sqsubseteq est bien fondé.

correction

Si $u \sqsubseteq v$ et $u \neq v$, alors $|u| < |v|$. Comme \mathbb{N} est bien fondé, il ne peut exister de suite infinie strictement décroissante pour \sqsubseteq .

4. Montrer que \preceq n'est pas bien fondé.

correction

Il faut pour cela supposer que $|\Sigma| > 1$. On considère deux lettres $a < b$ de Σ .
On a alors une suite infinie $\dots aab \preceq aab \preceq ab \preceq b$, donc \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.

correction

Supposons L rationnel, et considérons un automate émondé \mathcal{A} reconnaissant L .
On construit \mathcal{A}' comme \mathcal{A} en rendant acceptant tous les états de \mathcal{A} .
Les états de \mathcal{A} étant tous co-accessibles, u est reconnu par \mathcal{A}' si et seulement si il existe v tel que uv est reconnu par \mathcal{A} , si et seulement si $u \in \text{Pref}(L)$.
 $\text{Pref}(L)$ est reconnaissable, donc rationnel.

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

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

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 .

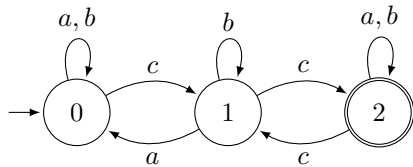
6. Trouver $\min(\{ab, abac, aca, ca, cb, baa\})$.

correction

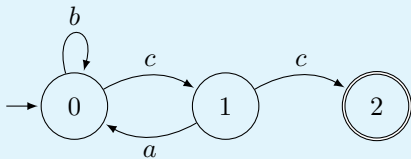
$\min(\{ab, abac, aca, ca, cb, baa\}) = \{ab, aca, ca\}$, car

- ces trois éléments ne sont pas comparables entre eux ;
- tous les autres éléments ont un élément strictement inférieur :
 - $ab \prec abac$
 - $ca \prec cb$
 - $ab \prec baa$

7. Appliquer la construction précédente à l'automate suivant :



correction



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

correction

On le prouve par double inclusion.

▷ Soit $u \in \text{rec}(\mathcal{A}')$. Immédiatement, $u \in L$, car \mathcal{A}' est construit à partir de \mathcal{A} en retirant des transitions.

Raisonnons par l'absurde, et supposons qu'il existe $v \prec u$ dans L .

- si v est préfixe (strict) de u , le calcul étiqueté u dans \mathcal{A}' arrive dans un état acceptant après lecture de $|v|$ lettres, et ne peut continuer d'après (i) : absurde ;
- sinon, notons w le plus grand préfixe commun de u et v , de sorte qu'il existe $u' et v' tels que $u = wu'$ et $v = wv'$, avec la première lettre (notée a) de v' strictement inférieure à celle (notée b) de u' . Notons q l'état atteint après lecture de w . Alors q a au moins deux transitions sortantes : l'une étiquetée par a et l'autre par b , avec $a \leq b$, ce qui contredit (ii)$

◁ Soit $u \in \min(L)$. Montrons que le calcul étiqueté par u dans \mathcal{A} n'emprunte que des transitions conservées dans \mathcal{A}' , ce qui montrera que $u \in \text{rec}(\mathcal{A}')$.

- Ce calcul ne peut emprunter une transition sortante d'un état acceptant : sinon u serait suffixe strict du mot v étiquetant le calcul menant à cet état acceptant, ce qui est exclu.
- Ce calcul ne peut emprunter une transition sortante d'un état q étiqueté par b telle qu'il existe $a < b$ étiquetant une autre transition sortante de q : sinon, notons w le préfixe de u dont la lecture a amené en q . Alors wa est le début d'un mot w' de L (car \mathcal{A} est émondé donc l'état atteint après lecture de wb est co-accessible), et $w' \prec u$, ce qui est exclu.

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 ?

correction

Il s'agit d'un problème de classification.

2. Est-ce que l'apprentissage par classification hiérarchique conviendrait pour répondre à ce problème ? Justifier brièvement.

correction

Non, car on cherche ici à classer les données en 2 classes.

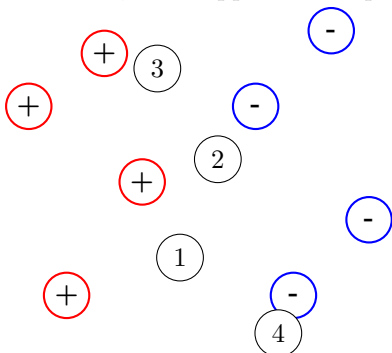
3. On désire implémenter l'algorithme des k -plus proches voisins. Rappeler le principe de cet algorithme, et décrire comment on l'utiliserait.

correction

(Ébauche)

C'est du cours.

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 $k = 3$?

correction

Avec $k = 3$, les escargots 1, 2 et 3 sont dopés, et pas 4.

5. Rappeler la complexité dans le cas général de l'algorithme.

correction

$O(kn)$ (ou $O(n \log(n))$ si $k > \log(n)$) – où n est le nombre de points de données.

6. Peut-on améliorer cette complexité avec une nouvelle structure de données ?

correction

Oui, à l'aide d'un arbre kd .

7. Dresser la matrice de confusion des données ci-dessus dans le cas $k = 3$.

correction

Avec les données, seul l'escargot correspondant au point «-» le plus à gauche est mal classé.
On obtient la matrice de confusion suivante (avec les vraies classes en lignes, et les classes prédites en colonnes).

$$\begin{array}{c|cc} & - & + \\ \hline - & 3 & 1 \\ \hline + & 0 & 4 \end{array}$$

8. Peut-on affirmer à 70% que notre algorithme est bon pour $k = 3$?

correction

Question ambiguë : oui si l'on considère le taux d'erreur global ($1/8$), non si l'on considère le taux d'erreur sur chaque classe ($1/4$ pour -).

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 :

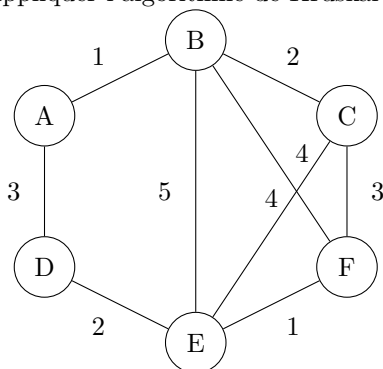
- *Unir*, qui prend en argument deux éléments de E et fusionne les classes de ces deux éléments ;
- *Trouver*, qui prend en argument un élément de E et renvoie la classe à laquelle il appartient.

1. Rappeler brièvement le principe de l'algorithme de Kruskal, et indiquer en particulier à quel endroit il utilise une structure *Unir et Trouver*.

correction

On part d'une structure Unir et Trouver où chaque élément a sa propre classe. On parcourt les arêtes par poids croissants, et si les extrémités ne sont pas dans la même classe, on choisit l'arête, et l'on unit les classes des extrémités.

2. Appliquer l'algorithme de Kruskal au graphe suivant :



correction

On sélectionne successivement $A - B$, $E - F$, $B - C$, $D - E$ et $A - D$.

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 $E = \llbracket 0, 5 \rrbracket$?

correction

C'est la forêt où chaque arbre est réduit à sa racine.

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.

correction

(Ébauche)

À détailler graphiquement.

On obtient les classes $\{0, 1, 2, 3\}$ et $\{4, 5\}$.

5. Comment implémenter efficacement une telle forêt en *C* ou en *OCaml*?

correction

À l'aide d'un tableau d'entier t , où $t[i]$ désigne le parent de i dans la forêt ($t[i]=i$ si c'est une racine).

6. Donner le pseudo-code de l'opération *Unir*.

correction

Sans compression de chemin :

```

1 Trouver(t, i) :
2   TantQue t[i] ≠ i :
3     i ← t[i]
4   Renvoyer i

```

```

1 Unir(t, i, j) :
2   ri ← Trouver(t, i)
3   rj ← Trouver(t, j)
4   t[ri] ← rj.

```

7. Donner le pseudo-code de l'opération *Trouver*.

correction

Cf question précédente.

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 $e_0 = (ab)^*(\epsilon|a)$.

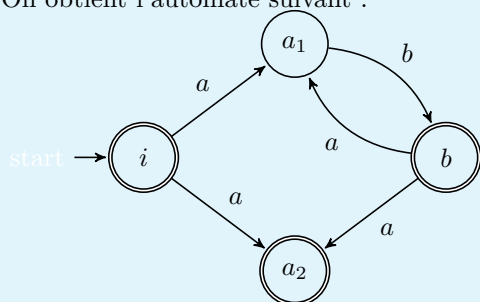
1. Appliquer l'algorithme de Berry-Sethi pour obtenir un automate \mathcal{A}_{e_0} reconnaissant ce langage.

correction

L'expression régulière linéarisée est $e_0 = (a_1b)^*(\epsilon|a_2)$, donc $L = L_{\langle I, F, P, \alpha \rangle}$, où

- $I = \{a_1, a_2\}$
- $F = \{b, a_2\}$
- $P = \{a_1b, ba_1, ba_2\}$
- $\alpha = \top$

On obtient l'automate suivant :



2. Quel est la complexité de l'algorithme de Berry-Sethi en fonction de $n = |e|$?

correction

- L'indiciage des lettres s'effectue en $O(n)$ (en utilisant un dictionnaire pour stocker l'indice maximal de chaque lettre rencontrée jusqu'à présent).
- I , F et α se calculent en $O(n)$, et P en $O(n^2)$.
- L'automate se construit alors en temps linéaire en la somme des tailles de l'expression e , de I , F et P . On obtient donc une complexité quadratique en n dans le pire des cas.

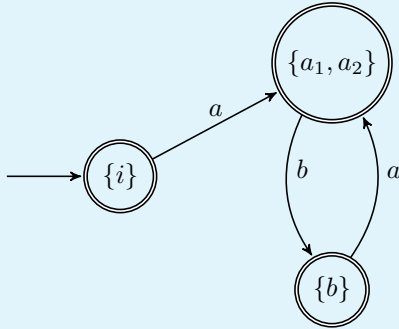
3. Déterminer \mathcal{A}_{e_0} .

correction

On calcule la table de transition du déterminisé.

	a	b
$\{i\}$	$\{a_1, a_2\}$	\emptyset
$\{a_1, a_2\}$	\emptyset	$\{b\}$
$\{b\}$	$\{a_1, a_2\}$	\emptyset

On obtient l'automate suivant (sans faire figurer l'état rebut).



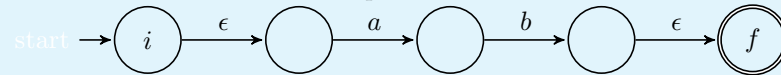
4. Appliquer l'algorithme de Thompson à e_0 pour obtenir un nouvel automate reconnaissant ce langage.

correction

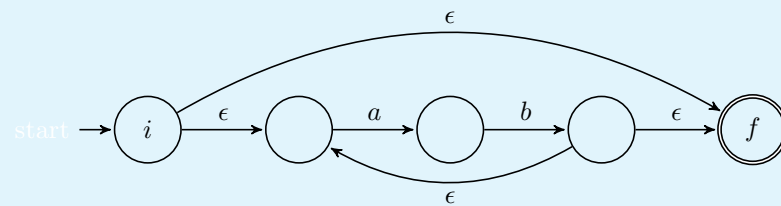
En l'absence de précision, on utilise une version avec automates avec transitions spontanées.

(Rem : on pourrait obtenir d'autres constructions que celles données ci-dessous, les constructions «à la Thompson» n'étant pas pour moi parfaitement standardisées.)

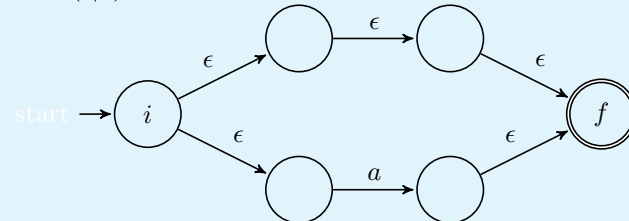
On obtient l'automate suivant pour ab :



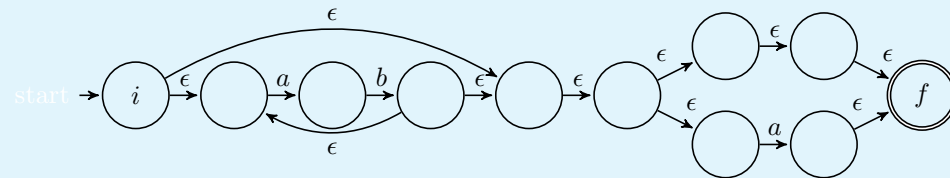
Puis le suivant pour $(ab)^*$:



Pour $(\epsilon|a)$, on obtient



et finalement, pour L



5. Discuter sa complexité.

correction

Si l'on ne souhaite pas supprimer les ϵ -transitions, les constructions pour la concaténation, l'étoile et l'union s'effectue en temps constant.

La complexité de la construction est alors en $O(n)$ (avec une structure de données représentant un automate par un dictionnaire de liste d'adjacence – et en ayant également en accès en temps constant le sommet initial et le sommet acceptant de tout automate).

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 :

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

Algorithme₁($V, n, (v_i)$) :

$k \leftarrow 0$

$V_k \leftarrow 0$

$i \leftarrow 0$

TantQue $i < n$:

Si $V_k + v_i \leq V$:

$n_i \leftarrow k$

$V_k \leftarrow V_k + v_i$

$i++$

Sinon :

$k++$

$V_k \leftarrow 0$

Renvoyer $k + 1$

1. Que représentent les n_i dans cet algorithme?

correction

n_i est la boîte dans laquelle est l'objet d'indice i .

2. Quel est le k renvoyé par l'algorithme sur l'instance donnée en exemple, et quelle est la répartition correspondante des objets?

correction

Sur l'instance donnée, on obtient la répartition suivante des objets :

- boîte 0 : 2, 5
- boîte 1 : 4
- boîte 2 : 7
- boîte 3 : 1, 5
- boîte 4 : 8

L'algorithme renvoie 5.

3. On note k^* la solution optimale pour une instance quelconque. Montrer que

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

correction

Par définition, les n objets doivent pouvoir rentrer dans k^* boîtes, d'où l'inégalité.

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.

correction

Le volume des objets de deux boîtes consécutives est supérieur à V , sans quoi l'ouverture de la seconde boîte n'aurait pas été nécessaire.

Si le nombre de boîtes est pair, on a donc $\sum_{i=0}^{n-1} v_i \geq Vk/2$, et s'il est impair, $\sum_{i=0}^{n-1} v_i \geq V(k-1)/2$, ce qui donne l'inégalité demandée dans les deux cas.

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

correction

Une α -approximation pour un problème de minimisation est un algorithme renvoyant toujours une solution inférieure à α fois la solution optimale.

D'après les questions précédentes, on a ici $V(k-1)/2 \leq k^*V$, d'où $k \leq 2k^* + 1 \leq 3k^*$, donc l'algorithme est une 3-approximation (ou asymptotiquement une α -approximation pour tout $\alpha > 2$).

On se donne maintenant l'algorithme suivant :

Algorithme₂($V, n, (v_i)$) :

$k \leftarrow 1$

$V_k \leftarrow 0$

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 ?

correction

Sur l'instance donnée, on obtient la répartition suivante des objets :

- boîte 0 : 2, 5, 1
- boîte 1 : 4, 5
- boîte 2 : 7
- boîte 3 : 8

L'algorithme renvoie 4.

7. Indiquer les complexités temporelles dans le pire des cas des deux algorithmes.

correction

Le premier algorithme est de complexité $O(n)$ (s'il y a eu création d'une nouvelle boîte, i sera nécessairement incrémenté à l'itération suivante, le nombre d'itération est donc majoré par $2n$).

Le second est de complexité $O(n^2)$ (on a $k \leq n$ à tout moment).

8. Donner une idée naturelle permettant d'améliorer l'algorithme 2. Indiquer quels changements cela produirait sur la complexité.

correction

On pourrait penser à trier les objets par volume décroissant. Ce pré-tri nécessiterais une complexité $O(n \log n)$, et donc la complexité de l'algorithme resterait $O(n^2)$.

Chapitre 38

(CCINP) Jeu sur un graphe * (CCINP 25, ex A, retour élève, corrigé — oral - 115 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 de 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.

correction

Une stratégie pour un joueur est une application de l'ensemble des états non finaux contrôlé par le joueur, qui à tout état associe l'un de ses états successeurs.

2. Donner la définition d'une stratégie gagnante.

correction

Une stratégie est gagnante pour le joueur J si toute partie respectant cette stratégie depuis la position initiale termine dans un état final gagnant pour J .

3. Donner l'ensemble des états finaux sur le graphe suivant – en supposant que R commence par colorier 1.



correction

Le seul état final est une alternance de couleurs bleu/rouge/bleu/rouge/bleu.

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

correction

S'il y a un état final gagnant pour R, tous les sommets y sont coloriés, et d'après les règles du jeu, deux états adjacents ne peuvent pas être de même couleur.
Le graphe est donc biparti.

On suppose maintenant que G est biparti. On note (S_1, S_2) 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 . »

correction

Si l'invariant est vérifié à une étape donnée, et que R joue, il doit jouer sur un sommet adjacent à un sommet colorié en bleu, qui est dans S_2 par hypothèse. Donc le nouveau sommet colorié en rouge est dans S_1 . De même si c'est à B de jouer.

6. Montrer que s'il reste des sommets non coloriés, alors au moins l'un d'entre eux est coloriable.

correction

Si aucun sommet n'est colorié, tout sommet est coloriable par R par définition. Sinon, soit s un sommet non colorié. Par connexité du graphe, il existe un tel sommet voisin d'un sommet colorié (le dernier sommet non colorié d'un chemin reliant s à un sommet colorié). Le graphe étant biparti, d'après la question précédente, ce sommet n'est adjacent qu'à des sommets colorés rouges et des sommets non colorés, ou à des sommets colorés bleus et des sommets non colorés. Il est donc coloriable en bleu ou rouge respectivement.

7. En déduire (E) : G est biparti si et seulement si toutes les stratégies de R sont gagnantes.

correction

Si G est biparti, d'après la question précédente, une partie ne peut pas s'arrêter tant qu'il reste des sommets non coloriés. Toute position finale est donc une position gagnante pour R, et comme il n'y a pas de cycle, toute partie est gagnée par R – donc toute stratégie est gagnante pour R. Réciproquement, si toute stratégie est gagnante pour R, il existe une position finale gagnante pour R, et donc d'après la question 4, le graphe est biparti.

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 .

correction

$x0 = 0z_1$ donc xCz_1 .

Par l'absurde, si xCz_2 , il existerait y tel que $xy = yz_2$, ce qui est impossible car les deux membres sont de longueurs différents.

2. Donner l'ensemble des mots z avec lesquels $x = \epsilon$ est conjugué.

correction

ϵ est conjugué avec lui-même (en prenant $y = \epsilon$), et uniquement avec lui-même (en reprenant l'argument de longueur précédent).

3. Montrer que s'il existe u et $v \in V^*$ tels que $x = uv$ et $z = vu$, alors x et z sont conjugués.

correction

Dans ce cas, on a $xu = uvu = uz$, donc 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 $|x| \geq |y|$.

correction

Soit y tel que $xy = yz$, et $|y| > |x|$. Il existe alors y' et y'' non vide tels que $y = xy'$ et $y = y''z$.

$xy''z = xy'z$, d'où $y' = y''$. Alors $xy' = y'z$, où $|y'| < |y|$.

En répétant ce procédé à partir de y' , on finit par obtenir y de longueur $\leq x$ tel que $xy = yz$.

5. En déduire que

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

correction

La réciproque a été prouvé à la question 3.

Pour le sens direct, on se donne y comme à la question précédente, de sorte que $xy = yz$, et $|y| \leq |x| = |z|$.

Alors x s'écrit $x = yx'$ et z s'écrit $z = z'y$.

On a alors $yx'y = yz'y$, d'où $x' = z'$. On a le résultat voulu en notant ce dernier mot v .

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

correction

Réflexivité Immédiat en prenant $y = \epsilon$.

Symétrie Immédiat d'après la question précédente.

Transitivité Supposons que xCy et yCz . Il existe alors t et u tels que $xt = ty$ et $yu = uz$. Alors $x(tu) = tyu = (tu)z$, d'où xCz .

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

7. $C(x)$ est-il régulier ?

correction

$C(x)$ est fini (car ne contient que des mots de longueur $|x|$), donc est 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.

correction

Considérons $L = \{a^{n^2}, n \in \mathbb{N}\}$.

On montre classiquement en utilisant le lemme de l'étoile que L n'est pas régulier.

Immédiatement, $C(L) = L$, qui n'est donc pas régulier.

9. Montrer que si L est régulier, $C(L)$ l'est également.

correction

(Ébauche)

On considère un AF standardisé émondé \mathcal{A} reconnaissant $L \setminus \epsilon$.

Pour chaque état e , on crée un nouvel automate en dupliquant \mathcal{A} en deux versions \mathcal{A}_1 et \mathcal{A}_2 , en prenant comme unique état initial la première copie de e , comme unique état acceptant la seconde copie de e , et en fusionnant l'état acceptant de \mathcal{A}_1 avec l'état initial de \mathcal{A}_2 .

Cet automate reconnaît l'ensemble des vu où $uv \in L$ et u étiquette un calcul qui arrive en e dans l'automate initial.

En effectuant l'union de ces automates pour tous les états e , on obtient un automate reconnaissant $L \setminus \epsilon$, qui est donc rationnel, donc L 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.

correction

(Ébauche)
C'est du cours.

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.

correction

```
1 let pere i = (i - 1) / 2
2
3 let est_tas_min t fin =
4   let rec aux i = (*teste si i est supérieur à son pere*)
5     i >= fin || (t.(i) >= t.(pere i) && aux (i+1))
6   in aux 1
```

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.

correction

```
1 let plus_petit_fils t fin i =
2   if 2*i+2 < fin && t.(2*i+2) < t.(2*i+1)
3   then 2*i+2 else 2*i+1
```

4. Écrire une fonction `descente t fin i` qui descend l'élément d'indice `i` à sa place dans le tas.

correction

```

1 let permute t i j =
2   let tmp = t.(i) in
3   t.(i) <- t.(j);
4   t.(j) <- tmp
5
6 let rec descente t fin i =
7   if 2*i+1 < fin (*sinon, pas de fils*)
8   then let j = plus_petit_fils t fin i in
9         if t.(j) < t.(i) then (
10          permute t i j;
11          descente t fin j
12         )

```

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 ?

correction

```

1 let supprime t fin =
2   let r = t.(0) in
3   permute t 0 (fin-1);
4   descente t (fin-1) 0;
5   r

```

La complexité est $O(\log(fin))$.

Chapitre 41

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

(CCINP 25, ex B, retour élève complété, corrigé — oral - 172 lignes)

*

Algorithmique, C,

sources : ccinpalgo3.tex

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.

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

correction

On peut faire (2, 1) ou (1, 2) : il y a donc 2 façons de faire.

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 .

correction

```
1 int** init_matrice(int n, int p, int v) {
2     int** r = malloc(n * sizeof(int*));
3     for (int i = 0; i < n; i++) {
4         r[i] = malloc(p * sizeof(int));
5         for (int j = 0; j < p; j++) {
6             r[i][j] = v;
7         }
8     }
9     return r;
10 }
```

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.

correction

```

1 void libere_mat(int** m, int n, int p) {
2     for (int i = 0; i < n; i++)
3         free(m[i]);
4     free(m);
5 }

```

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 $j > 0$?

correction

0, car on ne peut pas faire un score $j > 0$ en lançant 0 dé.

5. Montrer que $t_{i,j} = \sum_{k=1}^{\min(j,f)} t_{i-1,j-k}$ pour tous i et j à préciser.

correction

On considère $j > 0$, et $i > 0$.

On note $k \in \{1, \dots, f\}$ le résultat du dernier dé.

On remarque que si $k > j$, le score j ne peut être atteint pour les i dés.

Si l'on fixe $k \in \{1, \dots, \min(j, f)\}$, pour que j soit atteint, il faut et il suffit que les $i - 1$ autres dés aient donné le résultat $j - k$. Il y a $t_{i-1,j-k}$ façons de faire cela.

En sommant sur toutes les valeurs possibles de k , on obtient donc la formule voulue.

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.

correction

```

1 int nb_possibilites(int i, int j, int f) {
2     if (i == 0 && j == 0) return 1;
3     if (i == 0) return 0;
4     int r = 0;
5     for (int k = 1; k < min(j, f); k++)
6         r += nb_possibilites(i-1, j-k, f);
7     return r;
8 }

```

Le deuxième test ne renvoie pas un résultat dans un temps raisonnable, à cause du fort recouvrement sous-problèmes.

7. Implémenter une `int nb_possibilites_memo_asc(int i, int j, int f)` résolvant le même problème en utilisant une mémorisation et une implémentation ascendante.

correction

```

1 int aux(int i, int j, int f, int** memo) {
2     if (memo[i][j] > 0) return memo[i][j];
3     if (i == 0 && j == 0) return 1;
4     if (i == 0) return 0;
5     int r = 0;
6     for (int k = 1; k <= min(j, f); k++)
7         r += aux(i-1, j-k, f, memo);
8     memo[i][j] = r;
9     return r;
10 }
11
12 int nb_possibilites_memo_asc(int i, int j, int f) {
13     int** memo = init_matrice(i+1, j+1, 0);
14     int r = aux(i, j, f, memo);
15     libere_mat(memo, i, j);
16     return r;
17 }

```

(Un test avec (12,42,20) fait apparaître un overflow – le résultat renvoyé est strictment négatif.)

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 pour résoudre le problème.

correction

(Ébauche)

Il est nécessaire de transformer les couples (i, j) en un unique code pour utiliser la table de hachage.

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 hashtab hashtab;
```

```
//créé une nouvelle table de hachage de capacité indiquée
hashtab *hashtab_new();
```

```
//libère la mémoire occupée par une table de hachage
void hashtab_free(hashtab *t);
```

```
//ajoute une clé k avec une valeur v à la table - **sans vérifier si elle est déjà ←
à présente**.
void hashtab_add(hashtab *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.

correction

$(0, 0, 3, 1, 2, 1, 3, 2)$ convient.

2. Parmi les deux permutations suivantes, laquelle est une permutation de Langford ?

- $(3, 4, 0, 0, 3, 2, 4, 1, 2, 1)$
- $(3, 1, 4, 1, 3, 2, 0, 0, 2, 4)$

correction

La première, mais pas la seconde, car il y a 5 nombres entre les deux 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 ?

correction

```
1 int* initialise_tableau(int n, int v) {
2     int* r = malloc(n* sizeof(int));
3     for (int i = 0; i < n; i++)
4         r[i] = v;
5     return r;
6 }
```

On libère un tableau `t` créé par cette fonction par `free(t)`.

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 ?

correction

```
1 bool valeurs_valables(int n, int* t) {
2     for (int i = 0; i < 2*n; i++)
3         if (t[i] < 0 || t[i] >= n) return false;
4     return true;
5 }
```

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 .

correction

```
1 bool est_permutation(int n, int* t) {
2     int* c = malloc(n * sizeof(int));
3     for (int i = 0; i < n; i++)
4         c[i] = 0;
5     for (int i = 0; i < 2*n; i++)
6         c[t[i]]++;
7     for (int i = 0; i < n; i++)
8         if (c[i] != 2) return false;
9     return true;
10 }
```

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 .

correction

```
1 bool est_langford(int n, int* t) {
2     int* c = malloc(n * sizeof(int));
3     for (int i = 0; i < n; i++)
4         c[i] = -1;
5     for (int i = 0; i < 2*n; i++)
6         if (c[t[i]] == -1) c[t[i]] = i; // première occurrence on stocke l'←
7         // indice
8         else if (i - c[t[i]] != t[i] + 1) return false; // seconde ←
9         // occurrence : on vérifie que l'écart est correct
10    return true;
11 }
```

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 .

correction

```

1 int backtrack(int n, bool* b, int* t, int i) {
2     if (i == 2*n) { // permutation trouvée
3         // print_array(2*n, t);
4         return 1;
5     }
6     int r = 0;
7     for (int j = 0; j < n; j++)
8         if (!b[j] && i+j+1 < 2*n && t[i+j+1] == -1) { // j pas encore utilis←
9             // é, et utilisable en i
10            b[j] = true;
11            t[i] = j;
12            t[i+j+1] = j;
13            int inext = i+1;
14            while (inext < 2*n && t[inext] >= 0) {
15                inext++;
16            }
17            r += backtrack(n, b, t, inext);
18            // retour sur trace
19            t[i] = -1;
20            t[i+j+1] = -1;
21            b[j] = false;
22        }
23    return r;
24 }

```

11. Écrire une fonction `int nb_langford(int n)` résolvant le problème posé.

correction

```

1 int nb_langford(int n) {
2     int* t = initialise_tableau(2*n, -1);
3     bool* b = malloc(n*sizeof(bool));
4     for (int i = 0; i < n; i++) {
5         b[i] = false;
6     }
7     int r = backtrack(n, b, t, 0);
8     free(b);
9     free(t);
10    return r;
11 }

```

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.

correction

La fonction `verification : int array -> int -> int array -> bool` prend en argument un système de pièces non-vide, une somme v à atteindre et un tableau indiquant en la case d'indice i le nombre de pièce d'indice i prises, de même taille que le premier argument, et teste si la sommes des pièces est égale à v .

```
1 let verification p v t =  
2   let r = ref 0 in  
3   for i = 0 to Array.length p - 1 do  
4     r := !r + p.(i) * t.(i)  
5   done;  
6   !r = v
```

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».

correction

On parcourt le système de pièces de la gauche vers la droite (i.e. par valeurs décroissantes), et on prend autant de pièce que possible à chaque itération – jusqu'à ce qu'on atteigne la somme. On renvoie le nombre de pièces utilisées.

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 ?

correction

On obtient pour le premier exemple $b = 4.200 + 0.100 + 1.50 + 1.20 + 0.5 + 2.2 + 0.1$, donc 8 pièces, ce qui semble optimal (on peut montrer que c'est bien le cas).

Pour le second, on obtient $1.8 + 0.7 + 6.1$, donc 7 pièces – ce qui n'est pas optimal car on pourrait n'utiliser

que 2 pièces de 7.

4. L'algorithme glouton est-il un algorithme d' α -approximation pour un certain $\alpha > 1$?

correction

Non. On considère le système de pièces $\{n, n-1, 1\}$, et la somme à atteindre $2n-2$, où n est suffisamment grand. La solution optimale est 2 (car $2n-2 = 2(n-1)$), et l'algorithme glouton renvoie $n-2$ (car $2n-2 = 2.n + (n-2).1$). Comme $(n-2)/2$ peut être rendu arbitrairement grand, l'algorithme n'est pas une α -approximation, quelquesoit la valeur de α .

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.

correction

```

1 let rendu_memo p v =
2   let np = Array.length p in
3   let t = Array.make (v+1) (-1) in
4   t.(0) <- 0;
5   let rec aux i =
6     if t.(i) >= 0 then t.(i) else begin
7       let m = ref max_int in
8       for j = 0 to np - 1 do
9         if i >= p.(j) then begin
10          let c = 1 + aux (i - p.(j)) in
11          if c < !m then
12            m := c
13          end;
14        done;
15        t.(i) <- !m;
16        !m;
17      end;
18    in aux v
19  ;;

```

6. Quelle est la complexité de cette fonction ? Quelle stratégie algorithmique permettrait de réduire le temps d'exécution ?

correction

$O(|p|v)$.

On pourrait penser à une approche de type branch and bound, où l'on utiliserait comme heuristique de branchement l'ordre décroissant des pièces (i.e. on commencerait par explorer les possibilités en rendant la plus grande pièce possible).

7. Implémenter une fonction `rendu_asc` : `int array -> int -> int` résolvant le problème par une approche par programmation dynamique ascendante.

correction

```
1 let rendu_asc p v =
2   let np = Array.length p in
3   let t = Array.make (v+1) max_int in
4   t.(0) <- 0;
5   for i = 1 to v do
6     for j = 0 to np - 1 do
7       if i >= p.(j) then begin
8         let c = 1 + t.(i - p.(j)) in
9         if c < t.(i) then
10          t.(i) <- c
11       end;
12     done;
13   done;
14   t.(v)
```


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 ?

correction

On peut faire (2, 1) ou (1, 2) : il y a donc 2 façons de faire.

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

```
type lancer = int list
```

2. Écrire une fonction `somme` : `lancer` → `int` calculant le résultat d'un lancer.

correction

```
1 let rec somme l = match l with
2   [] -> 0
3   | t :: q -> t + somme q
```

3. Écrire une fonction `ajouter` : `lancer` → `int` → `lancer` qui ajoute l'entier passé en argument au lancer passé en argument.

correction

```
1 let ajouter e l = e :: l
```

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 ?

correction

`myst l i` renvoie la liste de toutes les listes obtenues en ajoutant à `l` les entiers de 0 à `i - 1`.

5. Écrire une fonction `toutes_possibilites : int → int → lancer list` qui prend en argument deux entiers `d` et `f`, et renvoie la liste des lancers possibles pour ces paramètres.

correction

```
1 let rec toutes_possibilites d f =
2   if d = 0 then [[]] else
3     let rec aux l = match l with (*applique myst à tous les éléments de ←
4       [] -> []
5       | t :: q -> myst t f @ aux q
6     in aux (toutes_possibilites (d-1) f)
```

6. Combien y a-t-il de possibilités pour $d = 6$ et $f = 6$?

correction

Il y en a 6^6 .

7. Écrire une fonction `toutes_possibilites_somme : int → int → int → 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`.

correction

```
1 let rec toutes_possibilites_somme d f s =
2   let rec aux l = match l with
3     [] -> []
4     | t :: q -> if somme t = s then t :: (aux q) else aux q
5   in aux (toutes_possibilites d f)
```

8. Tester pour les paramètres $(d, f, s) = (6, 6, 24)$, puis pour $(12, 8, 30)$, et commenter.

correction

Le deuxième test donne un `Stack overflow`. `toutes_possibilites` devrait renvoyer une liste de $8^{12} = 2^{36} > 10^{11}$ éléments, ce qui fait déborder la pile d'appel lors des concaténations.

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.

correction

Cette fois-ci, le second test ne termine pas en un temps raisonnable – on a un fort recouvrement de sous-problèmes.

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.

correction

```
1 let rec est_vide e = match e with
2   Vide -> true
3   | Union(e1, e2) -> est_vide e1 && est_vide e2
4   | Concat(e1, e2) -> est_vide e1 || est_vide e2
5   | _ -> false
```

2. Justifier que votre fonction termine.

correction

Par induction immédiate sur la structure d'expression régulière.

Si l'on veut détailler :

Cas de base : si l'expression est `Vide` ou une lettre, la fonction termine immédiatement.

Induction : • si l'expression est une étoile, la fonction termine immédiatement.

- si c'est l'union ou la concaténation de deux expressions e_1 et e_2 sur lesquelles la fonction termine, la fonction effectue au plus un appel récursif sur chacune de ces expressions, puis calcule une expression booléenne, donc elle termine.

3. Écrire une fonction `est_fini : expr -> bool` qui teste si une expression régulière dénote un langage fini.

correction

Il y a une subtilité pour l'étoile – l'étoile de e' étant finie si et seulement si e' est vide ou réduit à ϵ – ce que l'on teste à l'aide d'une fonction auxiliaire.

```

1 let rec est_vide_ou_epsilon e = match e with
2   Vide -> true
3   | Lettre(a) -> false
4   | Union(e1, e2) -> est_vide_ou_epsilon e1 && est_vide_ou_epsilon e2
5   | Concat(e1, e2) -> est_vide e1 || est_vide e2 || (est_vide_ou_epsilon e1 && est_vide_ou_epsilon e2)
6   | Etoile(e1) -> est_vide_ou_epsilon e1
7 ;;
8
9 let rec est_fini e = match e with
10  Vide -> true
11  | Lettre(a) -> true
12  | Union(e1, e2) -> est_fini e1 && est_fini e2
13  | Concat(e1, e2) -> est_vide e1 || est_vide e2 || (est_fini e1 && est_fini e2)
14  | Etoile(e1) -> est_vide_ou_epsilon e1
15 ;;

```

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.

correction

```

1 let rec fusion l1 l2 = match l1, l2 with
2   [], _ -> l2
3   | _, [] -> l1
4   | t1 :: q1, t2 :: q2 -> if t1 < t2
5                             then t1 :: fusion q1 l2
6                             else t2 :: fusion l1 q2

```

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

```

correction

`myst : string list -> string list -> string list` prend en argument deux listes ℓ_1 et ℓ_2 de chaînes de caractères, et renvoie une liste constituée de toutes les concaténations possibles entre un élément de ℓ_1 et un élément de ℓ_2 .

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`.

correction

Subtilité : il peut y avoir un intermédiaire infini avec l'étoile, mais un tel intermédiaire sera nécessairement concaténé in fine avec le langage vide sans quoi le langage initial ne serait pas fini. On peut donc toujours renvoyer le langage ϵ pour une étoile.

```

1 let rec langage l = match l with
2   Vide -> []
3   | Lettre(a) -> [String.make 1 a]
4   | Union(e1, e2) -> fusion (langage e1) (langage e2)
5   | Concat(e1, e2) -> myst (langage e1) (langage e2)
6   | Etoile(e) -> [" "]

```

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.

correction

On utilise une fonction auxiliaire renvoyant une expression régulière dénotant le langage réduit au mot non vide passé en argument.

```

1 let denotemot s =
2   let n = String.length s in
3   let r = ref (Lettre(s.[0])) in
4   for i = 1 to n-1 do
5     r := Concat(!r, Lettre(s.[i]));
6   done;
7   !r
9 let rec denote l = match l with
10  [] -> Vide
11  | t :: q -> Union(denotemot t, denote q)

```

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).

correction

On cherche s'il existe un chemin entre un sommet initial et un sommet acceptant par un algorithme de parcours. En notant p le nombre de transitions, si l'automate est représenté par listes d'adjacences, on obtient une complexité $O(n + p)$.

9. Même question si L est un langage fini.

correction

Même question, même réponse.

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

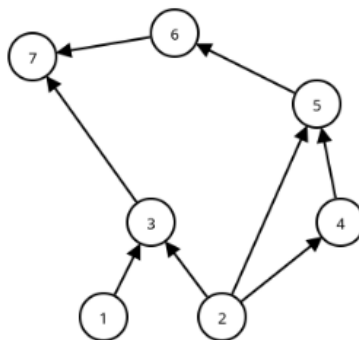
Un graphe orienté G est défini par un ensemble fini de sommets V et d'arêtes $E \subseteq V \times V$. Pour un sommet $x \in V$, ses *prédécesseurs* forment l'ensemble des sommets $y \in V$ tels que $(y, x) \in E$ et ses *successeurs* forment l'ensemble des sommets $z \in V$ tels que $(x, z) \in E$. 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 ℓ 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 ε , il existe une stratégie utilisant $O(\log n)$ jetons et $O(n^{1+\varepsilon})$ étapes.

correction



Ceci est une **ébauche de solution**, proposée par le jury du rapport X-ENS 2023.

1. Comme pour le moment on peut enlever un jeton n'importe quand, il suffit de s'assurer qu'on arrive à marquer 7. Les étapes du marquage sont les suivantes.

Placer jeton	2	1	3	4	5	6	7
Enlever jeton			1		2	4	
Nombre de jetons	1	2	3	2	3	4	3

2. On peut procéder par l'absurde. Si tous les sommets ont au moins un prédécesseur, on peut former un cycle de la manière suivante : on sélectionne un sommet, puis un prédécesseur de ce sommet, et ainsi de suite. On finit forcément par trouver un sommet déjà rencontré : cela crée un cycle.
3. Procéder par induction sur le nombre de sommets du graphe. Un graphe ne contenant aucun cycle contient toujours au moins un sommet qui n'a pas de prédécesseur. Placer un premier jeton sur ce sommet nous ramène au problème des jetons sur le graphe duquel on a enlevé ce sommet. Ce graphe est lui-même acyclique et de taille inférieure.
4. On montre par récurrence que : tout arbre ayant moins de ℓ feuilles a une stratégie avec $\lceil \log_2 \ell \rceil + 2$ jetons. Si l'arbre contient une seule feuille, il suffit d'un jeton (a fortiori 2). S'il en a 2, il faut 3 jetons. Si on prend maintenant un arbre à ℓ feuilles, la racine comporte deux sous-arbres, tous deux ayant $\ell - 1$ feuilles ou moins. L'un d'entre eux a plus de $\ell/2$ feuilles (disons ℓ'). On commence par celui-ci. On le marque en utilisant $\lceil \log_2 \ell' \rceil + 1 \leq \lceil \log_2 \ell \rceil + 1$ jetons (par hypothèse de récurrence). On laisse le jeton sur la racine de ce sous-arbre.

Ensuite, on s'occupe de l'autre sous-arbre, qui a moins de $\ell/2$ feuilles. On peut donc marquer sa racine en utilisant $\lceil \log_2(\ell/2) \rceil + 1 \leq \lceil \log_2 \ell \rceil$ jetons. Enfin, on marque la racine de l'arbre (à cette étape il n'y a plus que 3 jetons sur l'arbre). Le nombre de jetons utilisés est donc bien $\lceil \log_2 \ell \rceil + 1$.

5. Pour la première question : on place des jetons sur tous les sommets x_1 jusqu'à x_n , et on les retire de x_{n-1} jusqu'à x_1 dans l'ordre inverse.

Pour la deuxième, on peut remarquer que les conditions pour placer ou pour enlever un jeton sont les mêmes (tous les prédécesseurs marqués). Par conséquent le jeu est réversible : en partant de la configuration finale on peut ré-appliquer les mêmes étapes dans l'ordre inverse. On n'attendait pas forcément une démonstration très formelle pour cette sous-question, qui est principalement utile pour les questions suivantes.

6. Soit $m = \lfloor \sqrt{n} \rfloor$. La stratégie se découpe en plusieurs sous-étapes de la manière suivante :
 - pour tout j de 1 à $m - 1$, marquer le sommet jm de sorte qu'aucun sommet entre $(j - 1)m$ et jm n'est marqué : on utilise pour cela la stratégie naïve, qui utilise m jetons
 - marquer le sommet n à l'aide de la stratégie naïve (qui utilise donc $\leq m$ jetons)
 - pour tout j de $m - 1$ à 1, enlever le jeton placé sur le sommet jm . On utilise pour cela la stratégie naïve. En repartant du sommet $(j - 1)m$, on marque tous les sommets jusqu'à jm . On retire le jeton du sommet jm , puis de $jm - 1$, et ainsi de suite jusqu'à arriver au seul $(j - 1)m$ marqué.

La correction de cette stratégie est évidente : dans la configuration finale le sommet n est marqué et aucun sommet $1, \dots, n - 1$ ne l'est. Le nombre de jetons utilisés est de $O(\sqrt{n})$ pour les marquages intermédiaires et $O(\sqrt{n})$ pour les sous-chaînes. Le nombre d'étapes est $O(n)$.

7. On va décrire récursivement cette stratégie. Pour commencer, le sommet x_1 peut être marqué avec un seul jeton. Le sommet x_2 a besoin de deux jetons.

Comme dans la question précédente, on a besoin du fait que les stratégies de marquage sont réversibles, i.e., si on est capable d'arriver à la configuration où seul x_{2^k} est marqué, alors on est également capable, en partant de cette configuration, de retirer un jeton placé sur ce sommet (de sorte que tous les sommets de 1 à 2^k sont non marqués).

On montre donc par induction que :

ii) Pour tout k , en partant d'une configuration où aucun sommet n'est marqué, on peut marquer tout sommet en position $\leq 2^k$ avec $k + 1$ jetons et $T(2^k)$ étapes, de sorte qu'aucun autre sommet n'est marqué. \square

Supposons la propriété vraie pour k . Pour tout sommet $2^k < i \leq 2^{k+1}$, on procède de la manière suivante :

- marquer le sommet numéro 2^k en utilisant un appel récursif. On utilise $k + 1$ jetons ; à ce moment seul le sommet numéro 2^k est marqué

- marquer le sommet i . Pour ce faire, on considère la séquence d'étapes dans la stratégie de marquage du sommet $(i - 2^k)$, et on répète ces étapes, décalées de 2^k positions. Au lieu de partir de la racine, on part du sommet $2^k + 1$ (puisque 2^k est marqué). Comme le sommet 2^k est marqué, cela revient au même (on peut partir du sommet $2^k + 1$ comme on partait précédemment de la racine).

On utilise $k + 1$ jetons d'après l'hypothèse de récurrence, et comme 2^k reste marqué durant ces étapes, on a au total $k + 2$ jetons. À ce stade seuls les sommets 2^k et i sont marqués.

- on utilise une troisième fois l'hypothèse de récurrence et la réversibilité des stratégies pour enlever le jeton du sommet 2^k . On a donc bien que seul i est marqué.

Le nombre d'étapes nécessaires est : $T(2^{k+1}) = 3T(2^k)$. Comme on a $T(1) = 1$ cela implique $\implies T(2^k) = 3^k$ étapes. Pour un entier n donné, on utilise donc de l'ordre de $n^{\log_2 3}$ étapes.

8. On peut généraliser la construction précédente, qui consistait à faire une récursion en coupant le problème en deux.

On va couper en ℓ parties récursivement. Soient T_k et J_k le nombre d'étapes et de jetons pour marquer tous les sommets à distance ℓ^k : $T_k \leq 2\ell T_{k-1}$ et $J_k = J_{k-1} + \ell - 1$, donc $T_k = (2\ell - 1)^k \leq \ell^k 2^k$ et $J_k = (\ell - 1)k$. (Remarquer qu'on retrouve bien le cas $\ell = 2$).

En posant $n = \ell^k$ et en faisant varier k on a donc : $k = \log_\ell n$ et donc $T \leq n 2^{\log n / \log \ell} = n^{1 + \log 2 / \log \ell}$ et $J = (\ell - 1) \log_\ell n = O(\log n)$.

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.

1. Soit $x \in M$.
 - (a) 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.
 - (b) Montrer que les éléments $1, x, \dots, x^{j_x-1}$ sont distincts deux à deux.
 - (c) Montrer qu'il existe un unique $i_x \in [0, j_x-1]$ tel que $x^{i_x} = x^{j_x}$.
 - (d) 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 .
2. Qn 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.

3. 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 $3 \Rightarrow 1$.
 - (a) M est un monoïde apériodique.
 - (b) Pour tout $x \in M$, on a $p_x = 1$ (et donc $x^{i_x+1} = x^{i_x}$)
 - (c) $\exists k \in \mathbb{N}, \forall x \in M, x^{k+1} = x^k$

Définition 5

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

4. (Lemme de simplification). Soient $p, m, q \in M$.
 - (a) Montrer que si $m = pmq$, alors $m = pm = mq$.
 - (b) En déduire que si $pq = e_M$, alors $p = q = e_M$.
5. (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 Σ . Soit Σ^* l'ensemble des mots finis sur Σ , et ε 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

$$\begin{aligned}
 U &:= \bigcup_{\substack{a \in \Sigma \\ x \in M \setminus mM \\ x\mu(a)M \subseteq mM}} \mu^{-1}(x)a & V &:= \bigcup_{\substack{a \in \Sigma \\ x \in M \setminus mM \\ M\mu(a)x}} a\mu^{-1}(x) \subseteq Mm \\
 Y &:= \{a \in \Sigma \mid m \notin M\mu(a)M\} \cup Z & Z &:= \bigcup_{\substack{a, b \in \Sigma \\ m \in M\mu(a)xM \cap Mx\mu(b)M \\ m \notin M\mu(a)x\mu(b)M}} a\mu^{-1}(x)b
 \end{aligned}$$

correction

Remarques

- Le sujet note 1 à la place de e_M .
- Il me semble qu'entre les questions 2. et 3., le groupe à un élément devrait être $(\{e_x\}, \cdot_x, e_x)$ où \cdot_x est la restriction de \cdot_M .

1. (a) On peut définir l'application φ_x de \mathbb{N} dans M par $\varphi_x(k) = x^k$. Comme M est infini et M est fini, φ_x n'est pas injective donc l'ensemble $E = \{j \in \mathbb{N} \mid \exists i, 0 \leq i < j, x^i = x^j\}$ est une partie non vide de \mathbb{N} . E admet donc un minimum j_x .
 - (b) Si on avait $x^p = x^q$ avec $0 \leq p < q < j_x$ alors on aurait $q \in E$ avec $q < j_x$, ce qui est impossible : $1, x, x^2, \dots, x^{j_x-1}$ sont distincts deux-à-deux.
 - (c) Comme on a $j_x \in E$, il existe i_x tel que $x^{i_x} = x^{j_x}$ avec $0 \leq i_x < j_x$. D'après la question ci-dessus, i_x est unique.
 - (d) On a $x^{i_x+p_x} = x^{j_x} = x^{i_x}$ donc, par récurrence $x^{i_x+kp_x} = x^{i_x}$ puis $x^{i_x+kp_x+r} = x^{i_x+kp_x} \cdot x^r = x^{i_x} \cdot x^r = x^{i_x+r}$. Ce résultat est valide en particulier si $r = n \bmod p_x$.

2. On considère la division euclidienne de i_x par p_x , $i_x = kp_x + r$.
 Le produit de 2 éléments de G_x , x^{i_x+a} et x^{i_x+b} avec $0 \leq a, b < p_x$, peut s'écrire $x^{i_x+a} \cdot x^{i_x+b} = x^{i_x+a+i_x+b} = x^{i_x+a+kp_x+r+b} = x^{i_x+a+b+r}$.
 Si $r = 0$, $e_x = x^{i_x}$ est élément neutre et le symétrique de x^{i_x+a} est $x^{i_x+p_x-a}$ pour $1 \leq a < p_x$.
 Pour $0 < r < p_x$, $e_x = x^{i_x+p_x-r}$ est élément neutre, le symétrique de x^{i_x+a} est $x^{i_x+p_x-a-r}$ pour $0 \leq a \leq p_x - r$ et le symétrique de x^{i_x+a} est $x^{i_x+2p_x-a-r}$ pour $p_x - r < a < p_x - 1$. (Ceci marche donc aussi pour $r = 0$ avec l'abus $0 = p_x$.)

3. M contient les groupes G_x pour tout x avec G de cardinal p_x .
 - 1 \Rightarrow 2 : Si M est apériodique, G_x est de cardinal 1 donc $p_x = 1$ pour tout $x \in M$.
 - 2 \Rightarrow 3 : Si $p_x = 1$ pour tout $x \in M$ alors $x^{i_x+1} = x^{i_x}$ pour tout x donc, pour $k = \max\{i_x \mid x \in M\}$, on a $k - i_x \geq 0$ pour tout x puis (par récurrence immédiate) $x^k = x^{i_x+(k-i_x)} = x^{i_x} = x^{i_x+(k-i_x+1)} = x^{k+1}$.

3 \Rightarrow 1 : Si G est un groupe contenu dans M alors, pour $x \in G$ et $p \in \mathbb{N}$, x^p a la même valeur dans M et dans G . Ainsi, si $x^{k+1} = x^k$, on peut simplifier par x^k dans G donc $x = e_G$. Si la condition 3 est vérifiée les groupes contenu dans M sont réduits à leur élément neutre donc leur cardinal est 1 : M est apériodique.

4. (a) Si $m = pmq$ alors, par récurrence sur k , $m = p^r m q^r$ pour tout r . Si on choisit $r = k$ avec $x^k = x^{k+1}$ pour tout $x \in M$ alors $pm = p \cdot (p^k m q^k) = p^{k+1} m q^k = p^k m q^k = m$ et, de même, $mq = m$.
 - (b) Si on prend $m = e_M$, $e_M = pq = pe_M q$ donne $e_M = pe_M = p$ et $e_M = e_M q = q$.
5. Si $m = x$ alors $x = me_M \in mM$, $x = e_M m \in Mm$ et $m = e_M x e_M \in MxM$.

Inversement, si $x \in mM \cap Mm$ et $m \in MxM$, on peut écrire $x = mq = pm$ et $m = axb$ avec $a, b, p, q \in M$. $m = axb = apmb$ donc $m = apm = ax$ et $m = mp$; on en déduit $x = mq = axq$ d'où $x = ax = m$.

6. On appelle **facteur** d'un mot w , un mot w' tel qu'il existe $u, v \in \Sigma^*$ avec $m = uw'v$.

Sens direct On suppose que $m \notin M\mu(w)M$.

Si $m \in M\mu(a)M$ pour toute lettre de w , il existe des facteurs non vides w' de w tels que $m \in M\mu(w')M$, par exemple les facteurs de taille 1, et des facteurs w' de w tels que $m \notin M\mu(w')M$, par exemple $w = w'$. On considère un mot de longueur minimale w_0 tel que $m \notin M\mu(w_0)M$. w_0 est de longueur 2 au moins donc on peut l'écrire $w_0 = avb$ avec $a, b \in \Sigma$ et $v \in \Sigma^*$ donc $m = uavbt$. Comme av et vb sont de longueur strictement inférieure à celle de w' , on a $m \in M\mu(av)M$ et $m \in M\mu(vb)M$.

Ainsi l'une des deux propriétés du 2 est vérifiée.

Contraaposée On suppose que $m \in M\mu(w)M$: $m = x\mu(w)y$ avec $x, y \in M$.

Pour $w = uav$ avec $u, v \in \Sigma^*$ et $a \in \Sigma$, on a $m = x\mu(u)\mu(a)\mu(v)y \in M\mu(a)M$.

Pour $w = uavbt$ avec $u, v, t \in \Sigma^*$ et $a, b \in \Sigma$, on a $m = (x\mu(u))\mu(av)(\mu(bt)y) \in M\mu(av)M$, $m = (x\mu(ua))\mu(vb)(\mu(t)y) \in M\mu(vb)M$ et $m = (x\mu(u))\mu(avb)(\mu(t)y)$.

Ainsi la propriété 2 n'est pas vérifiée.

On utilisera la propriété $x \in mM$ équivaut à $xM \subset mM$.

En effet si $x \in mM$ alors $x = mm'$ avec $m' \in M$ donc, pour tout $y \in M$, $xy = m(m'y) \in mM$ d'où $xM \subset mM$. Inversement, si $xM \subset mM$, alors $x = xe_M \in mM$.

7. On appelle **facteur** d'un mot w , un mot w' tel qu'il existe $u, v \in \Sigma^*$ avec $w = uw'v$.

Sens direct On suppose que $\mu(w) \in mM$ avec $m \neq e_M$.

On remarque d'abord $e_M \notin mM$, car si on a $e_M = mx$, on a vu au **4. 2** qu'alors $m = x = e_M$.

Ainsi le préfixe ε de w vérifie $\mu(\varepsilon) = e_M \notin mM$ et le préfixe w vérifie $\mu(w) \in mM$. On considère le préfixe de w de longueur minimale w' tel que $\mu(w') \in mM$. w' est de longueur 1 au moins donc on peut l'écrire $w' = ua$ avec $u \in \Sigma^*$ et $a \in \Sigma$ et $w = w'v = uav$. u est un préfixe de longueur strictement inférieure à celle de $w' = ua$ donc $\mu(u) \notin mM$.

On a alors $\mu(ua) \in mM$ donc, $\mu(ua)M \subset mM$: la propriété 2 est vérifiée.

Réciproque On suppose que $w = uav$ et $\mu(ua)M \subset mM$ avec $u, v \in \Sigma^*$ et $a \in \Sigma$.

On a alors $\mu(w) = \mu(ua) \cdot \mu(v) \in \mu(ua)M$ donc $\mu(w) \in mM$.

8. Inclusion de $\mu^{-1}(m)$ Si $w \in \mu^{-1}(m)$ alors $\mu(w) = m \in mM$. D'après la question **7.**, on peut écrire $w = uav$ avec $\mu(u) \notin mM$ et $\mu(ua) \in mM$. Si on note $x = \mu(u)$ on a $x \in M \setminus mM$, $x\mu(a) = \mu(u)\mu(a) = \mu(ua) \in mM$ donc $x\mu(a)M \in mM$ et $u \in \mu^{-1}(x)$ donc $ua \in \mu^{-1}(x)$; ainsi $ua \in U$ puis $w = uav \in U\Sigma^*$.

On peut prouver une propriété analogue à celle du **7.** pour les préfixes : $\mu(w) \in Mm$ si et seulement si $w = uav$ avec $\mu(v) \notin Mm$ et $\mu(av)M \subset Mm$. On en déduit comme ci-dessus que $w \in \Sigma^*V$.

Si w appartenait à $\Sigma^*Y\Sigma^*$ alors

- soit $w = uaw$ avec $m \notin M\mu(a)M$, ce qui est impossible car $m = \mu(u)\mu(a)\mu(v)$
- soit $w = uw'v$ avec $w' \in Z$ c'est-à-dire $w' = atb$ avec $t \in \mu^{-1}(x)$, $m \in M\mu(a)xM$, $m \in Mx\mu(b)M$ et $m \notin M\mu(a)x\mu(b)M$. On a donc $w = uatbv$, $m \in M\mu(at)M \cap M\mu(tb)M$ et $m \notin M\mu(atb)$ ce qui donne, d'après la question **6.**, $m \notin M\mu(w)M$. On aboutit donc à la contradiction $m \notin MmM$.

Il est donc impossible d'avoir $w \in \Sigma^*Y\Sigma^*$: $w \in U\Sigma^* \cap \Sigma^*V \setminus \Sigma^*Y\Sigma^*$ d'où l'inclusion de $\mu^{-1}(m)$.

Caractérisation de $\mu^{-1}(m)$ Soit $w \in U\Sigma^* \cap \Sigma^*V \setminus \Sigma^*Y\Sigma^*$.

$w \in U\Sigma^*$ signifie qu'on peut écrire $w = uav$ avec $x = \mu(u) \notin mM$ et $x\mu(a) = \mu(ua) \in mM$. On en déduit que $\mu(w) = \mu(ua)\mu(v) \in \mu(ua)M \subset M$. De même $w \in \Sigma^*V$ donne $\mu(w) \in Mm$.

On va maintenant utiliser la question **6.** pour prouver qu'on a $m \in M\mu(w)M$.

Si on pouvait écrire $w = uav$ avec $m \notin M\mu(a)M$ alors a appartiendrait à Y d'où $w \in \Sigma^*Y\Sigma^*$, ce qui est exclu.

Si on pouvait écrire $w = uavbt$ avec $m \in M\mu(av)M$, $m \in M\mu(vb)M$ et $m \notin M\mu(avb)M$ alors, en posant $x = \mu(v)$, avb appartiendrait à $a\mu^{-1}xb$ donc à $Z \subset Y$ d'où $w \in \Sigma^*Y\Sigma^*$, ce qui est exclu.

Ainsi il n'est pas possible d'avoir $m \notin M\mu(w)M$. d'après la question **6.**

On a alors $\mu(w) \in mM \cap Mm$ et $m \in M\mu(w)M$ d'où $m = \mu(w)$ d'après la question **5.**

Chapitre 48

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

Logique propositionnelle,
sources : `enseliminationcutmll.tex`

Élimination 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 Γ, Δ 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.

correction

Corrigé de M.Péchaud

On réécrit le membre de droite du séquent :

$$\begin{aligned} A \otimes (B \oplus C) \multimap (A \otimes B) \oplus (A \otimes C) &= (A \otimes (B \oplus C))^\perp \wp ((A \otimes B) \oplus (A \otimes C)) \\ &= (A^\perp \wp (B^\perp \& C^\perp)) \wp ((A \otimes B) \oplus (A \otimes C)). \end{aligned}$$

On prouve alors le séquent demandé.

$$\begin{array}{c}
\frac{\frac{\frac{}{\vdash A^\perp, A} Ax}{} \quad \frac{\frac{}{\vdash B^\perp, B} Ax}{} \otimes}{\vdash A^\perp, B^\perp, (A \otimes B)} \oplus \quad \frac{\text{de même}}{\vdash A^\perp, C^\perp, ((A \otimes B) \oplus (A \otimes C))} \& \\
\hline
\frac{\frac{\frac{}{\vdash A^\perp, (B^\perp \& C^\perp)}, ((A \otimes B) \oplus (A \otimes C)) \wp}{\vdash (A^\perp \wp (B^\perp \& C^\perp)), ((A \otimes B) \oplus (A \otimes C))} \wp}{\vdash (A^\perp \wp (B^\perp \& C^\perp)) \wp ((A \otimes B) \oplus (A \otimes C))} \wp
\end{array}$$

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

correction

$$A \multimap (A \otimes A) = A^\perp \wp (A \otimes A)$$

L'arbre de preuve commence nécessairement de la façon suivante :

$$\begin{array}{c}
\frac{\frac{\frac{}{\vdash A^\perp, A} Ax}{} \quad \frac{}{\vdash A} \otimes}{\vdash A^\perp, A \otimes A} \wp \\
\vdash A^\perp \wp (A \otimes A)
\end{array}$$

Or $\vdash A$ ne peut pas être déduit sans coupure en général, car par induction immédiate sur l'arbre de preuve, tout séquent $\vdash \Gamma$ prouvable sans coupure contient au moins un connecteur logique, ce qui n'est pas le cas si A est une formule atomique.

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 appliquée à la prémisse gauche et la première règle appliquée à la prémisse droite de la coupure.

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

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

correction

On omet les \vdash pour alléger les notations.

L'arbre de preuve a nécessairement la forme suivante :

$$\frac{\frac{\Gamma'}{\Gamma, C^\perp} g \quad \frac{\Delta'}{C, \Delta} d}{\Gamma, \Delta} \text{Cut}$$

où Γ' et Δ' ne contiennent pas respectivement C^\perp et C .

Distinguons des cas – en remarquant que g et d ne peuvent pas être des règles de coupure.

- Si $g = Ax$, $\Gamma = C$, et l'arbre de preuve peut être réduit à

$$\frac{\Delta'}{C(=\Gamma), \Delta} d$$

où la coupure a été éliminée.

- Si $g = \otimes$, $C^\perp = A \otimes B$, d'où $C = A^\perp \wp B^\perp$.

L'arbre est de la forme

$$\frac{\frac{\frac{\Gamma_1, A}{\Gamma, A \otimes B} \otimes \quad \frac{\Gamma_2, B}{A^\perp, B^\perp, \Delta} \wp}{\Gamma, \Delta} \text{Cut}$$

où $\Gamma = \Gamma_1, \Gamma_2$

On peut alors le remplacer par l'arbre suivant, où la coupure sur C a été éliminée :

$$\frac{\frac{\overline{A^\perp, B^\perp, \Delta} \quad \overline{\Gamma_1, A}}{\Gamma_1, B^\perp, \Delta} \text{Cut} \quad \overline{\Gamma_2, B}}{\Gamma, \Delta} \text{Cut}$$

- Si $g = \oplus$, $C^\perp = A \oplus B$, d'où $C = A^\perp \& B^\perp$.
L'arbre est de la forme (à symétrie près)

$$\frac{\frac{\overline{\Gamma, A}}{\Gamma, A \oplus B} \oplus \quad \frac{\overline{A^\perp, \Delta} \quad \overline{B^\perp, \Delta}}{A^\perp \& B^\perp, \Delta} \&}{\Gamma, \Delta} \text{Cut}$$

que l'on peut remplacer par

$$\frac{\overline{\Gamma, A} \quad \overline{A^\perp, \Delta}}{\Gamma, \Delta} \text{Cut}$$

qui ne contient plus de coupure sur C .

- Les dernier cas sont symétriques des précédents.

Question 5. Soit Π 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 Π .

correction

L'arbre de preuve a nécessairement la forme suivante :

$$\frac{\frac{\overline{L}}{\Gamma, C^\perp} g \quad \frac{\overline{R}}{C, \Delta} d}{\Gamma, \Delta} \text{Cut}$$

où L correspond à un ou plusieurs séquents dont l'un au moins contient C^\perp (on procède de même s'il s'agit de R) :

$$\frac{\frac{\overline{\Gamma', C^\perp} \quad \dots}{\Gamma, C^\perp} g \quad \frac{\overline{R}}{C, \Delta} d}{\Gamma, \Delta} \text{Cut}$$

On remplace cette portion d'arbre par

$$\frac{\frac{\overline{\Gamma', C^\perp} \quad \frac{\overline{R}}{C, \Delta} d}{\Gamma', \Delta} \text{Cut} \quad \dots}{\Gamma, \Delta} g$$

(La règle g ne portant que sur des séquents de Γ' et Γ)

On a ainsi remonté la coupure sur C (dans le sens où le nombre de formules apparaissant dans la preuve au dessus du Cut a strictement décreu). En itérant le procédé jusqu'à ce que la coupure devienne principale (le nombre d'itérations est fini car l'arbre de preuve est fini), on se ramène à la question précédente, et l'on peut donc bien éliminer la coupure sur C .

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 ?

correction

Oui. En appliquant les transformations ci-dessus, on élimine les coupures sur C en faisant éventuellement apparaître des coupures sur des sous-formules de C .

L'ordre lexicographique sur la liste par ordre décroissant des tailles des C apparaissant dans une coupure dans la formule étant bien fondé, on a donc bien un bon algorithme.

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).

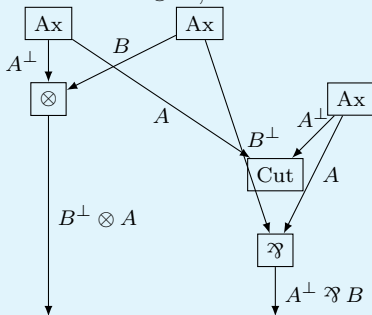
Question 7. Donner un algorithme qui prend une preuve Π de MLL et la transforme en un réseau correspondant, de sorte que les arêtes pendantes correspondent aux conclusions de la preuve.

correction

On commence par se donner un exemple de construction, pour la preuve suivante :

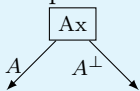
$$\frac{\frac{\frac{}{Ax} \quad \frac{}{Ax}}{A, A^\perp} \quad \frac{}{B, B^\perp}}{A, A^\perp \otimes B, B^\perp} \otimes \quad \frac{}{Ax}}{A, A^\perp} \text{Cut}$$

$$\frac{\frac{A^\perp \otimes B, B^\perp, A}{A^\perp \otimes B, B^\perp \wp A} \wp}{A^\perp \otimes B, B^\perp \wp A}$$



On donne alors une inductive d'un réseau pour une preuve, dont les arêtes pendantes correspondent aux conclusions de la preuve :

- Si la preuve est réduite à un axiome sur A , son réseau correspondant est



- Si la termine par une coupure de prémisses Γ, A^\top et A, Δ , on construit inductivement les réseaux pour chacune de ces prémisses. Les arêtes pendantes correspondant à A^\top et A sont reliées à l'entrée d'un nouveau sommet Cut sans arête sortante.
- Si la preuve termine par un \wp de prémisses Γ, A, B , on construit inductivement des réseaux pour Γ, A et B . Les arêtes pendantes correspondant à A et B sont reliées à l'entrée d'un nouveau sommet \wp , d'arête sortante $A \wp B$.
- De même pour $A \otimes B$.

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.

correction

On reprend le réseau précédent. Il correspond également à la preuve suivante :

$$\frac{\frac{\frac{}{Ax}}{A, A^\perp} \text{Ax} \quad \frac{}{A, A^\perp} \text{Ax}}{A, A^\perp} \text{Cut} \quad \frac{}{B, B^\perp} \text{Ax}}{A, A^\perp \otimes B, B^\perp} \otimes \quad \frac{}{A^\perp \otimes B, B^\perp} \wp \quad A$$

2. Donner un réseau qui n'est pas un réseau de preuve.

correction

C'est par exemple le cas avec deux nœuds (Ax) reliés à deux nœuds ().

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 $\chi(G) \geq \omega(G)$.

correction

Correction de L.Lanteri

Il faut au moins $\omega(G)$ couleurs pour colorier la plus grande clique. Formellement, si φ est un coloriage tel que $|\varphi(V)| < \omega(G)$ et si C est une clique de G de taille $\omega(G)$, alors il existe $v, v' \in C$ tels que $\varphi(v) = \varphi(v')$. Or, comme C est une clique, $\{v, v'\} \in E$, donc φ n'est pas un coloriage valide.

$$\boxed{\chi(G) \geq \omega(G)}$$

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 $\chi(G)\alpha(G) \geq |V|$.

correction

- On remarque que pour un coloriage valide, l'ensemble des noeuds d'une même couleur forme une anticlique.

- Étant donné un coloriage valide φ , on partitionne $V = \bigsqcup_{c \in \varphi(V)} \{v \in V : \varphi(v) = c\}$
 - Étant donnée une partition en anticliques $V = \bigsqcup_i A_i$, on colorie chaque noeud de A_i de la couleur i pour obtenir un coloriage valide.
- b. Par la question précédente, il existe une partition de V en $\chi(G)$ anticliques :

$$V = \bigsqcup_{i=1}^{\chi(G)} A_i$$

Or, par définition de $\alpha(G)$, $|A_i| \leq \alpha(G)$ pour tout i :

$$|V| = \sum_{i=1}^{\chi(G)} |A_i| \leq \sum_{i=1}^{\chi(G)} \alpha(G) = \chi(G)\alpha(G).$$

$$\boxed{\chi(G)\alpha(G) \geq |V|}$$

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

Grphe 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.

correction

Le graphe vide, le graphe singleton, les graphes complets sont des exemples de graphes parfaits. Un cycle de longueur 4 est un exemple 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 $\omega(G[V \setminus A]) = \omega$.

correction

Pour alléger les notations, on note $G' := G[V \setminus A]$.

On remarque que toute clique de G' est une clique de G , d'où $\omega(G') \leq \omega$. Il nous suffit donc de prouver que $\omega(G') \geq \omega$ pour conclure.

Si $A = \emptyset$, l'égalité est triviale : $\omega(G') = \omega(G) = \omega$.

Si $A \neq \emptyset$, alors G' est un sous-graphe induit propre de G , donc G' parfait puisque G est imparfait minimal. On en déduit donc que $\chi(G') = \omega(G')$.

De plus, $\omega \neq \chi$ puisque G n'est pas parfait et que $\omega(G[W]) = \chi(G[W])$ pour tout $W \subsetneq V$. D'après la question 1, on a donc $\omega + 1 \leq \chi$.

Si φ un coloriage valide de G' en $\chi(G') = \omega(G')$ couleurs, on peut prolonger φ en un coloriage valide de G en coloriant A d'une nouvelle couleur $c \notin \varphi(V \setminus A)$: on en déduit que $\chi \leq \omega(G') + 1$. On peut alors établir l'inégalité voulue :

$$\begin{aligned} \omega + 1 &\leq \chi \leq \omega(G') + 1 \\ \omega &\leq \omega(G') \end{aligned}$$

ce qui nous permet de conclure :

$$\boxed{\omega(G[V \setminus A]) = \omega}$$

Question 5. Soit A_0 une anticlique de G de cardinalité α . 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 α anticliques parmi $A_0, \dots, A_{\alpha\omega}$. (Formellement : $\forall v \in V, |\{i \in \{0, \dots, \alpha\omega\} : v \in A_i\}| = \alpha$.)

correction

Notons $A_0 = \{v_0, \dots, v_{\alpha-1}\}$.

Soit $v_i \in A_0$. On note $G_i := G[V \setminus \{v_i\}]$. G_i est ω -coloriable puisque G l'est. D'après la question 2.a, il existe donc une partition de $V \setminus \{v_i\}$ en ω anticliques, qu'on note $A_{i\omega+1}, \dots, A_{(i+1)\omega}$.

$$V \setminus \{v_i\} = \bigsqcup_{k=1}^{\omega} A_{i\omega+k}$$

Ainsi, on a construit α « paquets » de ω anticliques telles que :

- pour tout $v_i \in A_0$, v_i apparaît dans A_0 et une seule fois dans chaque « paquet » d'anticliques sauf dans le i -ème paquet ($A_{i\omega+1}, \dots, A_{(i+1)\omega}$), donc v_i fait partie d'exactly α anticliques ;
- pour tout $v \notin A_0$, v apparaît une fois dans chaque « paquet » d'anticliques, donc exactement α fois.

On a donc bien construit une famille d'anticliques qui convient.

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

correction

Soit $i \in \{0, \dots, \alpha\omega\}$. D'après la question 4, $\omega(G[V \setminus A_i]) = \omega$, on peut donc prendre C_i une clique de $G[V \setminus A_i]$ de taille ω , de sorte que $C_i \cap A_i = \emptyset$.

On remarque que pour tout j , $|C_i \cap A_j| \leq 1$. En effet, si l'on avait $u, v \in C_i \cap A_j$ distincts, alors

- u serait voisin de v puisque C_i est une clique ;
- u ne serait pas voisin de v car A_j est une anticlique : contradiction.

Or, chacun des ω sommets de C_i apparaît α fois dans $A_0, \dots, A_{\alpha\omega}$:

$$\sum_{i \neq j} |C_i \cap A_j| = \sum_{j=0}^{\alpha\omega} |C_i \cap A_j| = \alpha\omega$$

d'où on déduit

$$\forall j \neq i, |C_i \cap A_j| = 1$$

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^T la transposée de M_A .

Montrer que $M_A^T M_C$ est de rang $\alpha\omega + 1$, où les matrices sont vues comme à coefficient dans \mathbb{Q} .

correction

Pour tous $i, j \in \{0, \dots, \alpha\omega\}$, on a

$$(M_A^T M_C)_{i,j} = \sum_{v=1}^n (M_A)_{v,i} (M_C)_{v,j} = |A_i \cap C_j| = \begin{cases} 0 & \text{si } i = j \\ 1 & \text{si } i \neq j \end{cases}$$

$$M_A^T M_C = U_{\alpha\omega+1} - I_{\alpha\omega+1} = \begin{pmatrix} 0 & 1 & 1 & \dots & 1 \\ 1 & 0 & 1 & \dots & 1 \\ 1 & 1 & 0 & \dots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

où U_n est la matrice carrée d'ordre n dont tous les coefficients valent 1.

Un rapide argument d'algèbre linéaire nous permet de montrer que $M_A^T M_C$ est inversible :

- $\text{Sp } U_{\alpha\omega+1} = \{0, \alpha\omega + 1\}$
- $\text{Ker}(M_A^T M_C) = \text{Ker}(U_{\alpha\omega+1} - I_{\alpha\omega+1}) = \{0\}$ car $1 \notin \text{Sp}_{\mathbb{Q}} U_{\alpha\omega+1}$. (Note : $\alpha\omega \geq 1$ par leur définition.)

$$\text{rg}_{\mathbb{Q}}(M_A^T M_C) = \alpha\omega + 1$$

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

correction

Encore un argument purement algébrique :

$$\alpha\omega + 1 = \text{rg}(M_A^T M_C) \leq \text{rg}(M_C) \leq n$$

$$n \geq \alpha\omega + 1$$

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 .

correction

Montrons par récurrence forte la propriété $\mathcal{P}(n)$: \forall Pour tout graphe $G = (V, E)$ avec $|V| = n$, G est parfait si et seulement si $\omega(G[W])\alpha(G[W]) \geq |W|$ pour tout sous-graphe induit $G[W]$ de G .

Initialisation : pour $n = 0$, le cas du graphe vide est trivial.

Hérédité : soit $n \in \mathbb{N}$ et supposons $\mathcal{P}(0), \dots, \mathcal{P}(n)$ vraies. Soit $G = (V, E)$ avec $|V| = n + 1$.

Supposons que pour tout sous-graphe induit $G[W]$ de G , $\omega(G[W])\alpha(G[W]) \geq |W|$. Par hypothèse de récurrence, tous les sous-graphes induits propres (avec $|W| \leq n$) sont parfaits. Si l'on suppose que G n'est pas parfait, il est donc imparfait minimal, ce qui implique d'après la question 8 que $|V| \geq \alpha(G)\omega(G) + 1$; or, $|V| \leq \alpha(G)\omega(G)$, contradiction. G est donc parfait.

Supposons maintenant que G soit parfait. Tous ses sous-graphes induits propres sont donc parfaits, ce qui par hypothèse de récurrence implique que pour tout sous-graphe induit propre $G[W]$, $\omega(G[W])\alpha(G[W]) \geq |W|$. Enfin, la question 2.b. nous permet de montrer que $\omega(G)\alpha(G) = \chi(G)\alpha(G) \geq |V|$.

On a donc bien la condition nécessaire et suffisante voulue pour tout graphe avec $|V| = n + 1$: $\mathcal{P}(n + 1)$ est vraie.

Conclusion : 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.

correction

Soit G un graphe parfait. Une clique de G est une anticlique de \bar{G} et une anticlique de G est une clique de \bar{G} . On a donc :

$$\omega(G) = \alpha(\bar{G})$$

$$\alpha(G) = \omega(\bar{G})$$

Or, par la caractérisation de la question précédente, pour tout $W \subset V$,

$$|W| \leq \omega(G[W])\alpha(G[W]) = \alpha(\bar{G}[W])\omega(\bar{G}[W]) = \alpha(\bar{G}[W])\omega(\bar{G}[W])$$

donc \bar{G} est un graphe parfait.

La réciproque suit trivialement : si \bar{G} est parfait, $\bar{\bar{G}} = G$ est parfait.

Un graphe est donc parfait si et seulement si son complémentaire est parfait.

Chapitre 50

(ENS) Inférence de type *** (ENS 23, corrigé — oral - 280 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 σ 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 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 σ 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 σ_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 σ_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. $\{(\mathbb{N} \rightarrow A, B \rightarrow \mathbb{N})\}$

correction

Corrigé de M.Péchaud

$\sigma_0(A) = \sigma_0(B) = \mathbb{N}$, et $\sigma_0(X) = X$ pour tout autre X .

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

correction

On pose $\sigma_0(C) = \mathbb{N} \rightarrow \mathbb{N}$, $\sigma_0(B) = \mathbb{N}$ et $\sigma_0(A) = ((\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N})$ et $\sigma_0(X) = X$ pour tout autre X .

3. $\{(A \rightarrow B), (B \rightarrow A)\}$

correction

$\sigma_0(A) = \sigma_0(B) = C$, et $\sigma_0(X) = X$ pour tout autre X .

4. $\{(\mathbb{N} \rightarrow (A \rightarrow \mathbb{N}), \mathbb{N} \rightarrow B), (B, A)\}$

correction

Impossible car on devrait avoir

$$\sigma_0(A) = \sigma_0(B)$$

et $\mathbb{N} \rightarrow (\sigma_0(A) \rightarrow \mathbb{N}) = \mathbb{N} \rightarrow \sigma_0(B)$, et le membre de droite à strictement moins de \rightarrow que celui de gauche.

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

correction

Non, cf question précédente.

Quand un unificateur pour S est-il unique ?

correction

Jamais : si on a un unificateur σ , on peut en construire un nouveau en choisissant deux variables V_1 et V_2 n'apparaissant nulle part dans la substitution de types ou dans l'unificateur et en posant $\sigma'(V_1) = V_2$, et $\sigma'(X) = \sigma(X)$ pour tout autre X .

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.

correction

- Si $T_1 = T'_1$, il y a un unificateur pour $\{(T_k, T'_k)\}_{1 \leq k \leq n}$ si et seulement si il y a un unificateur pour $\{(T_k, T'_k)\}_{2 \leq k \leq n}$.
- Si $T_1 = V_i$ est une variable et T'_1 est quelconque, il y a un unificateur pour $\{(T_k, T'_k)\}_{1 \leq k \leq n}$ si et seulement si il y a un unificateur pour $\{(T_k, T'_k)\}_{2 \leq k \leq n}$ dans lequel toutes les occurrences de V_i ont été remplacées par T'_1 (ce choix étant forcé).
De même pour le cas réciproque.
- Si $T_1 = \mathbb{N}$ et $T_2 = U_2 \rightarrow V_2$, l'unificateur n'existe pas.
De même pour le cas symétrique.
- Reste le cas où $T_1 = U \rightarrow V$ et $T'_1 = U' \rightarrow V'$.
Dans ce cas, l'existence d'un unificateur pour le système de départ est équivalente à celle pour le système $\{(T_k, T'_k)\}_{2 \leq k \leq n} \cup \{(U, U'), (V, V')\}$.

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

correction

```

1  Entrées : un système S
2  Sorties : un unificateur de S s'il existe, ⊥ sinon.
3
4  σ ← Id
5
6  Tant que S ≠ la liste vide :
7      Extraire le premier couple (T1, T'1)
8      Si T1 = T'1 :
9          continuer
10     Sinon si T1 est une variable :
11         σ(T1) ← T'1
12         Substituer toutes les occurrences de T1 par T'1 dans S
13     Sinon si T'1 est une variable :
14         // cas symétrique du précédent
15     Sinon, si T1 = U → V et T'1 = U' → V' :
16         ajouter (U, V) et (U', V') à S
17     Sinon :
18         Renvoyer ⊥
19 Renvoyer σ

```

À chaque itération qui ne termine pas, l'ordre lexicographique sur «nombre de variables dans S » \times «nombre total de " \rightarrow " dans S » diminue strictement, ce qui garantit la 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 Γ' défini par $\Gamma'(x) = T$ et pour tout y dans le domaine de Γ , $y \neq x \Rightarrow \Gamma'(y) = \Gamma(y)$.

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 Γ 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 Γ .

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

correction

$x(y)$ est nécessairement de type \mathbb{N} .

Notons T_y et T_x les types de y et x .

$y \mapsto x(y) + y$ est de type $T_y \rightarrow \mathbb{N}$.

Donc f_0 est de type $T_x \rightarrow (T_y \rightarrow \mathbb{N})$.

Deux exemples de types légitimes :

- avec $T_x = A \rightarrow \mathbb{N}$ et $T_y = A$,
 $f_0 : (A \rightarrow \mathbb{N}) \rightarrow (A \rightarrow \mathbb{N})$
- avec $T_x = (A \rightarrow B) \rightarrow \mathbb{N}$ et $T_y = A \rightarrow B$,

$$f_0 : ((A \longrightarrow B) \longrightarrow \mathbb{N}) \longrightarrow ((A \longrightarrow B) \longrightarrow \mathbb{N})$$

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

correction

On crée une variable $V_{E'}$ pour chaque sous-expression de E autre qu'un entier ou qu'une variable.

On donne les règles directement sous forme d'équations de types – ce qui fait déjà un pas vers la question suivante.

On note alors $S_\Gamma(E)$ le système d'équations de type de E dans le contexte Γ , que l'on définit inductivement.

- Si $E \in \mathbb{N} : S_\Gamma(E) = \{(E, \mathbb{N})\}$.
- Si $E \in \mathcal{X} : S_\Gamma(E) = \{(E, \Gamma(E))\}$.
- Si $E = E_1 + E_2, S_\Gamma(E) = \{(V_E, V_{E_1}), (V_E, V_{E_2})\} \cup S_\Gamma(E_1) \cup S_\Gamma(E_2)$.
- Si $E = E_1(E_2), S_\Gamma(E) = \{(V_{E_1}, V_{E_2} \longrightarrow V_E)\} \cup S_\Gamma(E_1) \cup S_\Gamma(E_2)$.
- Si $E = x \mapsto E', S_\Gamma(E) = \{V_E, V_x \longrightarrow V_{E'}\} \cup S_{\Gamma, x:T}(V_{E'})$ où V_x est une nouvelle variable de type.

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

correction

On calcule le système donné à la question précédente, puis on en calcule une solution à l'aide de l'algorithme décrit à la question 4 (en gardant en mémoire les types donnés aux variables substituées). Le type donné à V_E est alors solution au problème.

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 .

correction

On obtient successivement les équations suivantes en descendant dans la formule (en notant $E' : y \mapsto (z \mapsto x(z)(y(z)))$, $E'' : z \mapsto x(z)(y(z))$ et $E''' : x(z)(y(z))$) :

$(V_E, V_x \mapsto V_{E'})$ (le contexte contient maintenant $\Gamma(x) = V_x$).

$(V'_E, V_y \mapsto V_{E''})$ (le contexte contient maintenant $\Gamma(y) = V_y$).

$(V''_E, V_z \mapsto V_{E'''})$ (le contexte contient maintenant $\Gamma(z) = V_z$).

$(V_{x(z)}, V_{y(z)} \longrightarrow V_{E'''})$

$(V_x, V_z \longrightarrow V_{x(z)})$

$(V_y, V_z \longrightarrow V_{y(z)})$

On applique donc l'algorithme de la question 4 au système $S = \{(V_E, V_x \mapsto V_{E'}), (V'_E, V_y \mapsto V_{E''}), (V''_E, V_z \mapsto V_{E'''}), (V_{x(z)}, V_{y(z)} \longrightarrow V_{E'''}), (V_x, V_z \longrightarrow V_{x(z)}), (V_y, V_z \longrightarrow V_{y(z)})\}$.

On obtient successivement

• $\{(V'_E, V_y \mapsto V_{E''}), (V''_E, V_z \mapsto V_{E'''}), (V_{x(z)}, V_{y(z)} \longrightarrow V_{E'''}), (V_x, V_z \longrightarrow V_{x(z)}), (V_y, V_z \longrightarrow V_{y(z)})\} \quad (\sigma(V_E) = \sigma(V_x) \longrightarrow \sigma(V_{E'}))$

• $\{(V''_E, V_z \mapsto V_{E'''}), (V_{x(z)}, V_{y(z)} \longrightarrow V_{E'''}), (V_x, V_z \longrightarrow V_{x(z)}), (V_y, V_z \longrightarrow V_{y(z)})\} \quad (\sigma(V_{E'}) = \sigma(V_y) \longrightarrow \sigma(V_{E''}))$

• $\{(V_{x(z)}, V_{y(z)} \longrightarrow V_{E'''}), (V_x, V_z \longrightarrow V_{x(z)}), (V_y, V_z \longrightarrow V_{y(z)})\} \quad (\sigma(V_{E''}) = \sigma(V_z) \longrightarrow \sigma(V_{E'''}))$

• $\{(V_x, V_z \longrightarrow (V_{y(z)} \longrightarrow V_{E'''})), (V_y, V_z \longrightarrow V_{y(z)})\} \quad (\sigma(V_{x(z)}) = \sigma(V_{y(z)}) \longrightarrow \sigma(V_{E'''}))$

• $\{(V_y, V_z \longrightarrow V_{y(z)})\} \quad (\sigma(V_x) = \sigma(V_z) \longrightarrow (\sigma(V_{y(z)}) \longrightarrow \sigma(V_{E'''})))$.

• $\emptyset. \quad (\sigma(V_y) = \sigma(V_z) \longrightarrow \sigma(V_{y(z)}))$

En remontant, on obtient alors

• $\sigma(V_y) = \sigma(V_z) \longrightarrow \sigma(V_{y(z)})$

• $\sigma(V_x) = \sigma(V_z) \longrightarrow (\sigma(V_{y(z)}) \longrightarrow \sigma(V_{E'''}))$

• $\sigma(V_{x(z)}) = \sigma(V_{y(z)}) \longrightarrow \sigma(V_{E'''})$

• $\sigma(V_{E''}) = \sigma(V_z) \longrightarrow \sigma(V_{E'''})$

• $\sigma(V_{E'}) = (\sigma(V_z) \longrightarrow \sigma(V_{y(z)})) \longrightarrow \sigma(V_{E''}) = \sigma(V_y) \longrightarrow (\sigma(V_z) \longrightarrow \sigma(V_{E'''}))$

• $\sigma(V_E) = \sigma(V_x) \longrightarrow \sigma(V_{E'}) = (\sigma(V_z) \longrightarrow (\sigma(V_{y(z)}) \longrightarrow \sigma(V_{E'''}))) \longrightarrow ((\sigma(V_z) \longrightarrow \sigma(V_{y(z)})) \longrightarrow (\sigma(V_z) \longrightarrow \sigma(V_{E'''})))$

En renommant les variables de cette dernière expression pour y voir plus clair, on obtient comme type possible pour S :

$$\sigma(V_E) = (A \longrightarrow (B \longrightarrow C)) \longrightarrow ((A \longrightarrow B) \longrightarrow (A \longrightarrow C))$$

ce qui a une bonne tête.

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 σ 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. Hérédité : $X \in \mathcal{I} \implies (\forall Y \subset X, Y \in \mathcal{I})$
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.

correction



Ceci est une **ébauche de solution**, proposée par le jury du rapport X-ENS 2023.

1. Il n'y a qu'un seul processeur, deux tâches ne peuvent pas s'exécuter en même temps :

$$\forall i, j, i \neq j \implies \sigma(T_i) \neq \sigma(T_j)$$

Dit autrement, σ doit être injective.

2. Soit σ un ordonnancement optimal.
 - Supposons qu'une tâche à l'heure T_i ait lieu après une tâche en retard T_j . L'échange des deux tâches ne change rien (ou cela contredit l'optimalité).
 - Supposons que l'on ait deux tâches à l'heure $\sigma(T_i) < \sigma(T_j)$ et $d_i > d_j$. On peut échanger ces deux tâches : T_j reste à l'heure car elle est exécutée avant, et T_i car T_j était à l'heure et $d_j < d_i$.
3. (a) Trier les tâches par pénalité décroissante.
 - (b) Itérer sur les tâches :
 - (a) Considérer un nouvel ordonnancement où la tâche courante est insérée dans l'ordonnancement en préservant l'ordre sur les dates limites.
 - (b) Si ce nouvel ordonnancement a des tâches en retard, on continue avec l'ancien ordonnancement.
 - (c) Ajouter toutes les tâches en retard dans un ordre arbitraire (il faudra de toute façon payer les pénalités).

Complexité $\mathcal{O}(n^2)$ (car il faut tester à chaque fois si toutes les tâches sont à l'heure.)

4. Itérations :

- 1
- 2, 1
- 2, 1, 3
- 2, 4, 1, 3
- On ne peut pas ajouter T_5 car dans l'ordre canonique 5, 2, 4, 1, 3, T_3 est en retard
- On ne peut pas ajouter T_6 , le résultat final est donc 2, 4, 1, 3, 7, avec un coût de 5.

Au final, l'ordonnancement des tâches à l'heure donne 2, 4, 1, 3, 7. Il faut ensuite ajouter les tâches 5 et 6 dans n'importe quel ordre, cela ne change rien car elles seront en retard et qu'il faudra payer la pénalité.

5. Soit B une solution optimale. Si $x \in B$, c'est bon. On observe que $w(x) \geq w(y) \forall y \in B$, par tri de S et le fait que $\{y\}$ est indépendant (hérédité de B). On pose $A = \{x\}$ et on ajoute (par la propriété d'échange) des éléments de B vers A jusqu'à ce que $|A| = |B|$. On a alors $A = (B \setminus \{y\}) \cup \{x\}$. Ainsi $w(A) \geq w(B)$. Comme B est une solution optimale, $w(A) = w(B)$.
6. On prouve la propriété par récurrence sur la taille de S .

- Initialisation triviale.
- Hérédité : supposons la propriété vraie au rang n . Soit $M = (S, I)$ un matroïde pondéré avec S de taille $n+1$. Soit x le premier élément choisi par l'algorithme, on sait qu'il existe une solution optimale qui contient x . On applique l'hypothèse de récurrence sur $M' = (S \setminus \{x\}, \{X \subseteq S \setminus \{x\} \mid X \cup \{x\} \in I\})$ (qui est aussi un matroïde) pour obtenir A' . On note $A = A' \cup \{x\}$ est un indépendant. A' était une solution de poids maximal sur M' , donc A est une solution de poids maximal sur M .

Complexité : $\mathcal{O}(n \log n + n f(n))$ où $f(n)$ est le temps pour tester si un ensemble d'au plus n éléments est indépendant.

7. S est l'ensemble des tâches, les indépendants sont les ensembles de tâches qui peuvent être exécutées à l'heure.

- Hérédité : immédiat.
- Échange : soient $(A, B) \in \mathcal{I}^2$ avec $|A| < |B|$. On cherche $T_i \in B$ telle que $A \cup \{T_i\}$ soit encore exécutable à l'heure.

On note $N_t(A)$ le nombre de tâches de A avec une date limite $\leq t$.

On montre le résultat intermédiaire suivant : A est indépendant ssi $\forall 0 \leq t \leq n, N_t(A) \leq t$

— (1) \rightarrow (2) par contraposée on aura trop de tâches.

— (2) \rightarrow (1) la i ème plus grande date limite est au plus i , donc il n'y a pas de soucis d'ordonnement

On pose $k = \max_t N_t(A) \geq N_t(B)$ (ce maximum existe car $N_0(A) = N_0(B) = 0$). En particulier $N_{k+1}(B) >$

$N_{k+1}(A)$: il y a donc plus de tâches avec date limite $k + 1$ dans B que dans A. On choisit donc $T_i \in B \setminus A$ avec $d_i = k + 1$.

$A' := A \cup \{T_i\}$ est un indépendant, on le montre en prouvant que $\forall t, N_t(A') \leq t$ puis en utilisant le lemme ci-dessous. Pour $0 \leq i \leq t'$, $N_t(A') = N_t(A) \leq t$ car A indep. Pour $t \leq i \leq n$, $N_t(A') \leq N_t(B) \leq t$.

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 EEDDED qui effectue (**empile**, **empile**, **dépile**, **empile**, **dépile**, **dépile**) avec en entrée la permutation $(3, 1, 2)$ renvoie $(1, 2, 3)$:

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

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 $\text{EEDDED}(3, 1, 2) = (1, 2, 3)$ 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 π est triable par pile, alors elle ne contient pas le motif $(2, 3, 1)$.
3. Montrer que si π 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 π 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 $O(|V|^3)$.

correction

Correction proposée par le jury

1. Elles sont triables par respectivement EDEDED, EEEEDDD et EDEEDED.
2. Par contraposée, supposons que π contient le motif $(2, 3, 1)$. Soient $x_1 < x_2 < x_3$ tels que $\pi(x_2) < \pi(x_3) < \pi(x_1)$. Pour faciliter le raisonnement, supposons sans perte de généralité que $x_1 = 1$, $x_2 = 2$ et $x_3 = 3$. Lors de l'exécution d'une machine qui trie π , 2 est empilé avant 3 qui est empilé avant 1. Nécessairement, 1 doit être le premier à être dépilé (sinon 2 ou 3 apparaît avant 1). Mais alors, 3 apparaîtra avant 2 en sortie, donc la machine ne trie pas π .
3. Soit π une permutation ne contenant pas le motif $(2, 3, 1)$. Alors la suite d'instructions construite de la manière suivante trie π :

```

1  Entrée : permutation  $\pi \in \mathfrak{S}_n$ 
2   $x \leftarrow 1$ 
3  TantQue tous les éléments n'ont pas été empilés et dépilés
4      Si  $x$  n'a pas encore été empilé
5          Empiler.
6      Sinon
7          Dépiler.
8           $x \leftarrow x + 1$ 

```

Une telle suite d'instructions trie effectivement la pile, car si x (qui correspond à l'élément minimal qui n'a pas encore été dépilé) a déjà été empilé, il se trouve en haut de la pile. En effet, distinguons deux cas :

- si x est le dernier élément à avoir été empilé, il est effectivement en haut de la pile ;
 - sinon, on a continué d'empiler après $x = x_2$ jusqu'à empiler un élément $x_1 < x_2$, dont on a dû attendre l'empilage avant de commencer à dépiler. Si on suppose qu'il existe un élément $x_3 > x_2$ qui se trouve sur la pile entre x_1 et x_2 , alors π contiendrait le motif $(2, 3, 1)$, ce qui est absurde. Donc tous les éléments au dessus de x_2 sont inférieurs à x_2 .
4. L'ensemble des permutations triables par pile n'est pas stable par inverse. En effet, $\pi = (3, 1, 2)$ est triable par EDEDD, mais son inverse est $(2, 3, 1)$ qui n'est pas triable par pile. De même, il n'est pas stable par composition, car $\pi^2 = (2, 3, 1)$.
 5. On peut appliquer l'algorithme décrit à la question 3 : si, au moment de dépiler, l'élément en haut de la pile n'est pas égal à x , alors la permutation n'est pas triable par pile. L'algorithme s'effectue bien en temps linéaire (la vérification de la boucle Tant Que peut se faire avec un compteur, le test du Si peut se faire en utilisant un tableau de booléens).
 6. Supposons que $\pi, \rho \in \mathfrak{S}_n$, avec $\pi \neq \rho$. Soit $I \subset \{1, \dots, n\}$ l'ensemble des indices i tels que $\pi(i) \neq \rho(i)$. Nécessairement, $|I| \geq 2$. Posons $x = \max \pi(I) = \max \rho(I)$, et $i = \pi^{-1}(x)$, $j = \rho^{-1}(x)$. Sans perte de généralité, $i < j$. Dès lors, $\pi(j) < \pi(i) = x$, donc $\{i, j\}$ est une arête dans $G(\pi)$, et $\rho(i) < \rho(j)$, donc $\{i, j\}$ n'est pas une arête dans $G(\rho)$.

La fonction est donc bien injective.

Elle n'est pas bijective par un argument de cardinalité (fonction d'un ensemble de cardinal $n!$ dans un ensemble de cardinal $2^{\frac{n(n-1)}{2}}$), sauf pour $n = 1$ et $n = 2$.

7. Le 4-cycle n'est pas réalisable. En effet, si le 4-cycle est réalisé par une permutation $\pi \in \mathfrak{S}_4$, alors on a :
 - $\pi(1) = 3$, sinon le sommet 1 est de degré différent de 2,
 - $\pi(2) = 4$, sinon on aurait $\pi(4) = 4$ donc le sommet 4 est de degré 0 ou $\pi(3) = 4$ donc le sommet 3 est de degré au plus 1,

- $\pi(3) = 1$, sinon le sommet 3 est de degré 3 car alors $\pi(3) = 2$ et $\pi(4) = 1$
- Et donc $\pi(4) = 2$

On a alors $\pi = (3, 4, 1, 2)$. Or $(3, 4, 1, 2)$ n'est pas triable par pile, car elle possède le motif $(2, 3, 1)$ (les 3 premiers éléments).

8. Montrons par induction que tous les graphes obtenus par cette construction sont réalisables :

- le graphe vide est réalisable par la permutation vide ;
- soit $G_1 = G(\pi)$ et $G_2 = G(\rho)$, avec $\pi \in \mathfrak{S}_k$ et $\rho \in \mathfrak{S}_\ell$ triables par pile par les machines M_π et M_ρ .
 - si on rajoute un sommet adjacent à tous les autres à G_1 pour obtenir le graphe G'_1 , alors ce graphe est réalisé par $(k+1, \pi(1), \pi(2), \dots, \pi(k))$. Cette permutation est triable par pile par la machine $EM_\pi D$;
 - si on fait l'union disjointe de G_1 et G_2 pour obtenir G'_2 , alors ce graphe est réalisé par $(\pi(1), \dots, \pi(k), \rho(1) + k, \dots, \rho(\ell) + k)$. Cette permutation est triable par pile par la machine $M_\pi M_\rho$.

On conclut par induction que ces graphes sont réalisables.

Réciproquement, soit $\pi \in \mathfrak{S}_n$ une permutation triable par pile. Posons $G = G(\pi) = (S, A)$.

- supposons qu'au cours de l'exécution de la machine sur π , la pile ne se vide qu'au dernier dépileage. Alors $\pi(1)$ est dépilé en dernier, donc $\pi(1) = n$. Par définition de $G(\pi)$, le sommet 1 est adjacent à tous les autres. De plus, $G[S \setminus \{1\}] = G(\pi(2) - 1, \dots, \pi(n) - 1)$, qui est triable par pile ;
- sinon, il existe $k \in \llbracket 1, n \rrbracket$ tel que $\{\pi(1), \dots, \pi(k)\} = \{1, \dots, k\}$. Dès lors, aucun sommet de $\{1, \dots, k\}$ n'est adjacent à un sommet de $\{k+1, \dots, n\}$. De plus, G est l'union disjointe (non numérotée) de $G(\pi_1)$ et $G(\pi_2)$, avec $\pi_1 = (\pi(1), \dots, \pi(k))$ et $\pi_2 = (\pi(k+1) - k, \dots, \pi(n) - k)$, qui sont tous les deux triables par pile.

On conclut par récurrence descendante sur le nombre de sommets des graphes.

9. On propose l'algorithme suivant, avec des graphes représentés par tableau de listes d'adjacence :

- si le graphe est réduit à 1 ou 2 (même 3) sommets, il est réalisable ;
- sinon, s'il n'est pas connexe (vérifiable en temps $O(|V|^2)$), on appelle récursivement l'algorithme sur chaque composante connexe (elles peuvent être construites en temps $O(|V|^2)$) ;
- sinon, s'il existe un sommet adjacent à tous les autres (trouvable en temps $O(|V|^2)$), on le supprime et on appelle récursivement l'algorithme sur le graphe obtenu (constructible en temps $O(|V|^2)$) ;
- sinon, le graphe n'est pas réalisable.

La complexité totale est bien celle attendue (on aurait pu affiner la complexité en $O(|V|(|V| + |A|))$).

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 Φ et Ψ , $\Phi \models \Psi$ signifie que toute valuation satisfaisant Φ satisfait également Ψ .

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.

correction

Correction proposée par le jury
Prendre $A_1 = \mathbb{N}, A_2 = A_3 = \emptyset$.

- (b) Montrer qu'en ajoutant l'inclusion $\mathbb{N} \subseteq A_2$, ce n'est plus satisfiable.

correction

On a une contradiction entre $A_3 = \mathbb{N}$ et $A_3 = \emptyset$.

2. Soit une conjonction d'inclusions ne contenant aucune inclusion de la forme $\mathbb{N} \subseteq X$. Montrer que cette conjonction est satisfiable.

correction

Il suffit d'évaluer toutes les variables à \emptyset .

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_Φ tel que :

- Les sommets de G_Φ sont les éléments de $\mathcal{T} \cup \{\emptyset, \mathbb{N}\}$
 - Pour toute inclusion $X_i \subseteq Y_i$ de Φ , on crée une arête $X_i \rightarrow Y_i$ dans G_Φ
 - 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_Φ est une séquence de sommets Z_1, \dots, Z_ℓ 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) Φ est satisfiable

- (b) Il n'existe pas de chemin dans G_Φ menant de \mathbb{N} à \emptyset
 (c) Φ est satisfiable à valeurs dans $\{\emptyset, \mathbb{N}\}$

correction

Pour $3 \implies 1$: si Φ est satisfiable à valeurs dans $\{\emptyset, \mathbb{N}\}$, en particulier elle est satisfiable.

Pour $1 \implies 2$: tout chemin $Z_1 \rightarrow Z_2 \rightarrow \dots \rightarrow Z_k$ dans G se traduit en une chaîne d'inclusions $Z_1 \subseteq \dots \subseteq Z_k$. Par conséquent, s'il existe un chemin de \mathbb{N} à \emptyset , par transitivité de l'inclusion (et récurrence immédiate sur la longueur du chemin) Φ implique $\mathbb{N} \subseteq \emptyset$ ce qui est une contradiction, et elle n'est pas satisfiable.

Pour $2 \implies 3$: Pour un sommet X de G_Φ , soit $s(X)$ l'ensemble des sommets Y tels que $X \rightarrow^* Y$ (successeurs, par clôture transitive), et $p(X)$ l'ensemble des sommets Y tels que $X \rightarrow^* Y$ par des chemins dans le graphe (prédécesseurs).

Supposons qu'il n'existe pas de chemin de \mathbb{N} à \emptyset . On montre qu'il existe une valuation qui fonctionne, qui sera à valeurs dans $\{\emptyset, \mathbb{N}\}$. On propose d'évaluer à \mathbb{N} pour $V_N = s(\mathbb{N})$ et \emptyset pour $V_0 = p(\emptyset)$. Par hypothèse V_0 et V_N sont disjoints. Quant aux sommets restants, on les évalue tous à \emptyset . Pour vérifier que cette valuation fonctionne, il faut vérifier qu'elle satisfait toutes les inclusions de Φ .

Soit $X \subseteq Y$ une inclusion dans Φ . On étudie les cas possibles : si $X \in V_N$, alors $Y \in V_N$ et l'inclusion est satisfaite. Si $Y \in V_0$, alors $X \in V_0$ et de même. Si $X \in V_0$, alors $X \subseteq D$ sera toujours satisfait. Si $Y \in V_N$, de même. Reste le cas où : $X \notin (V_0 \cup V_N)$ et $Y \notin (V_0 \cup V_N)$. Comme on a évalué ces deux variables à \emptyset cette inclusion est satisfaite.

Nd M. Péchaud : on peut également raisonner en terme de composantes fortement connexes du graphe et d'ordre topologique, ce qui facilite également la question suivante – je rédigerai ça plus tard.

4. Montrer que si Φ est satisfiable et X et Y sont des sommets de G_Φ , alors $\Phi \models (X \subseteq Y)$ si et seulement s'il existe un chemin dans G_Φ menant de X à Y .

correction

Une des deux implications est facile : s'il existe un chemin de X à Y , on a $X \subseteq Y$.

Supposons qu'il n'existe pas de chemin de X à Y (en particulier $X \neq Y$, car $X \rightarrow^* X$ est aussi un chemin). Par hypothèse Φ est satisfiable, donc il n'y a pas de chemin de \mathbb{N} à \emptyset . Cette fois, on va trouver une valuation de Φ qui ne satisfait pas $X \subseteq Y$.

L'idée est d'assigner \emptyset à Y et \mathbb{N} à X , et de voir si on peut encore satisfaire toutes les inclusions de Φ . (Notons qu'on ne peut pas avoir $Y = \mathbb{N}$ ni $X = \emptyset$ ici puisque cela contredirait l'hypothèse selon laquelle il n'y a aucun chemin). On propose la valuation suivante : \mathbb{N} pour $V_N = s(X)$ (qui contient aussi les successeurs de \mathbb{N} puisque \mathbb{N} est successeur de X) ; \emptyset pour $V_0 = p(Y)$ (qui contient aussi les prédécesseurs de \emptyset) ; \emptyset pour les sommets restants. Par hypothèse V_0 et V_N sont disjoints, sinon on aurait un chemin de X à Y . On réutilise le raisonnement ci-dessus pour montrer que toutes les inclusions de Φ sont satisfaites.

On peut remarquer que par exemple une formule Φ force $\emptyset \subseteq A$ pour toute variable A , même si A n'apparaît pas dans Φ .

Nd M. Péchaud : cf remarque précédente, il est beaucoup plus agréable de raisonner en terme d'ordre topologique – je rédigerai ça plus tard.

5. En déduire un algorithme **Résolution** qui prend en entrée une conjonction d'inclusions Φ 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 Φ et de variables de \mathcal{T} .

correction

Il faut d'abord déterminer si Φ est satisfiable (dans ce cas on renvoie Vrai tout le temps).

Ensuite, il faut déterminer s'il existe un chemin entre X et Y dans le graphe G_Φ . On fait simplement un parcours en profondeur à partir de X (il n'est pas demandé de le détailler, puisqu'ils sont censés le connaître). Quand le graphe est représenté comme une liste d'adjacence, la complexité est linéaire en le nombre d'arêtes. Ce nombre d'arêtes dépend du nombre d'inclusions dans Φ et de variables dans \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 $\{\emptyset, \mathbb{N}\}$.

correction

On est obligé de prendre $A_1 = \mathbb{N}, A_2 = \emptyset$. Ensuite, comme $A_1 \subseteq A_2$ est faux, il ne reste plus que les contraintes : $\neg(A_1 \subseteq A_3) \implies A_3 \neq \mathbb{N}$ et $\neg(A_3 \subseteq \emptyset) \implies A_3 \neq \emptyset$. Pour A_3 on peut prendre tout ensemble non vide qui n'est ni \emptyset ni \mathbb{N} .

On admet la propriété (P) :

« Soit Ψ une conjonction d'inclusions. Si Ψ est satisfiable, il existe une valuation f telle que pour tous sommets X, Y dans le graphe G_Ψ , 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 Φ une formule inclusion-Horn, et soit Ψ la conjonction de toutes les clauses de Φ 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 Ψ est satisfiable, et si pour tout littéral négatif $\neg(X \subseteq Y)$ dans les clauses de Φ , $\Psi \not\models (X \subseteq Y)$, alors Φ est satisfiable.

correction

Noter que, puisque le graphe G_Ψ contient toutes les variables de \mathcal{T} (y compris les variables qui ne sont pas dans Ψ), la valuation donne bien des valeurs à toutes ces variables.

Supposons que Ψ est satisfiable. Utilisons la valuation ci-dessus. Pour toutes variables X, Y , si $\Psi \not\models (X \subseteq Y)$ alors il n'existe pas de chemin entre X et Y par la question 4. Donc $f(X)$ et $f(Y)$ sont incomparables, donc le terme $\neg(X \subseteq Y)$ est vrai.

Par conséquent cette valuation rend tous les termes négatifs dans les clauses de Φ vrais. Toutes les clauses sans termes négatifs étant vraies (puisque Ψ l'est), toutes les clauses de Φ sont vraies. Elle est donc satisfiable.

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

correction

On définit un algorithme récursif.

Soit Φ une formule inclusion-Horn et soit Ψ l'ensemble de ses clauses positives. On effectue une disjonction de cas. On voit avec la question 7 qu'on va avoir par exemple le cas où Ψ est satisfiable et n'implique aucun littéral négatif, dans ce cas on peut s'arrêter et la formule est satisfiable. Sinon il reste d'autres cas :

(a) Si Φ contient une clause vide, on renvoie « Non satisfiable »

(b) Si Ψ n'est pas satisfiable, on renvoie « Non satisfiable »

(c) Sinon, il existe un littéral négatif $\neg(X \subseteq Y)$ dans une des clauses de Φ tel que $\Psi \models X \subseteq Y$. Il faut observer que la formule obtenue en enlevant $\neg(X \subseteq Y)$ de la clause correspondante est équivalente à Φ .

Cela nous donne un algorithme récursif qui transforme Φ, Ψ en Φ', Ψ' où Φ' est plus petite que Φ (quand on ne s'arrête pas) d'un littéral négatif. Chaque appel récursif réduit la taille de la formule, donc le temps est polynomial. La correction de l'algorithme découle des questions précédentes. On peut raffiner le temps d'exécution de quadratique à linéaire en utilisant des structures de données adaptées.

9. Prouver la propriété (P).

correction

La valuation se construit en résolvant d'abord les cycles de G_Ψ , de façon à enlever toutes les inclusions triviales. On enlève aussi les successeurs de \mathbb{N} et les prédécesseurs de \emptyset (dont les valuations sont évidentes). On fait ensuite un tri topologique sur les sommets restants, de façon à les parcourir dans l'ordre suivant : on doit s'assurer que pour chaque sommet, on a d'abord étudié tous ses prédécesseurs. On démarre de \emptyset . Soit A_0, \dots, A_k, \dots la séquence des sommets rencontrés, alors l'ensemble $S(A_k)$ correspondant à A_k est calculé de la manière suivante :

$$S(A_k) = \{k\} \cup \bigcup_{X, X \rightarrow A_k} S(X)$$

Cette valuation satisfait bien toutes les inclusions, et de plus, elle a la propriété demandée par construction.

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 :

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

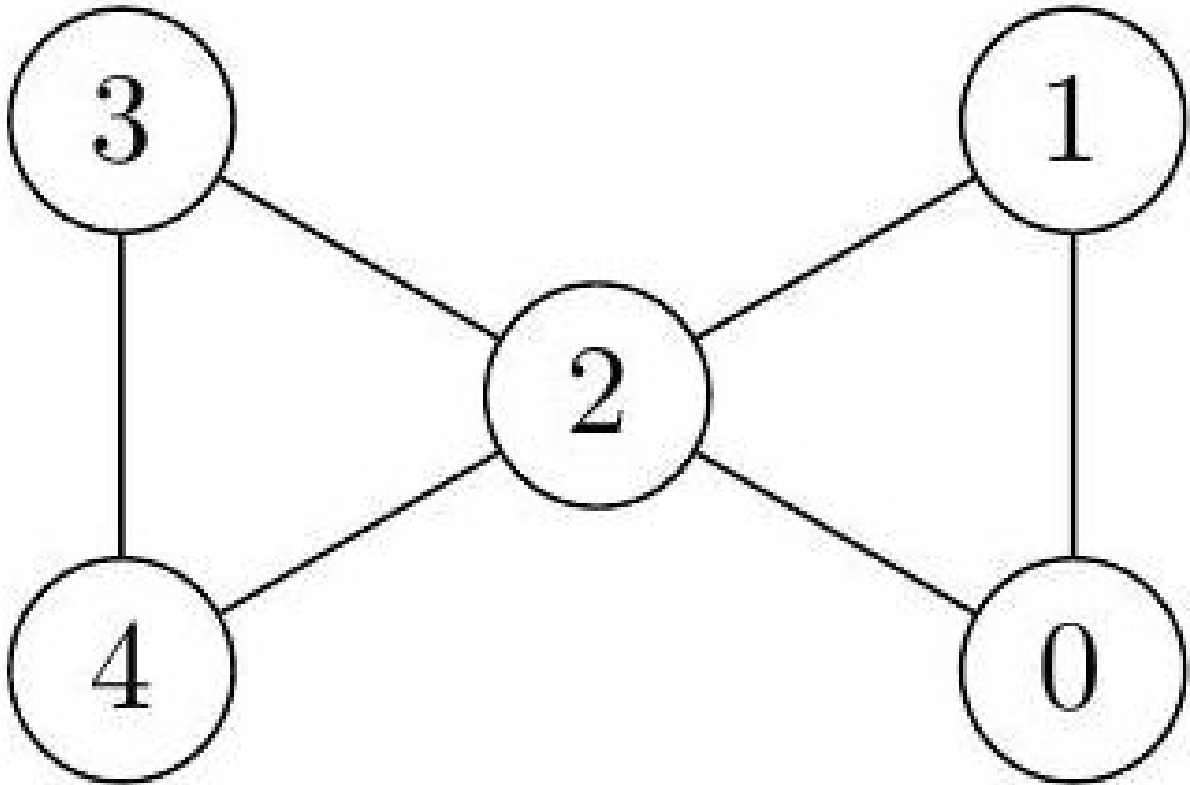
correction

(Auteur de la correction inconnu)

Si un graphe contient un 4-cycle (a, b, c, d) alors b et d sont tous les deux voisins communs de a et c donc le graphe n'est pas un graphe d'amis. On en déduit le résultat par contraposition.

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

correction



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

correction

$x * y$ doit être un voisin de x donc appartenir à $N(x) = \{y, z\}$ et un voisin de y donc $x * y \neq y$.
On a donc $x * y = z$ d'où $z \in N(y), \{y, z\} \in E$.

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

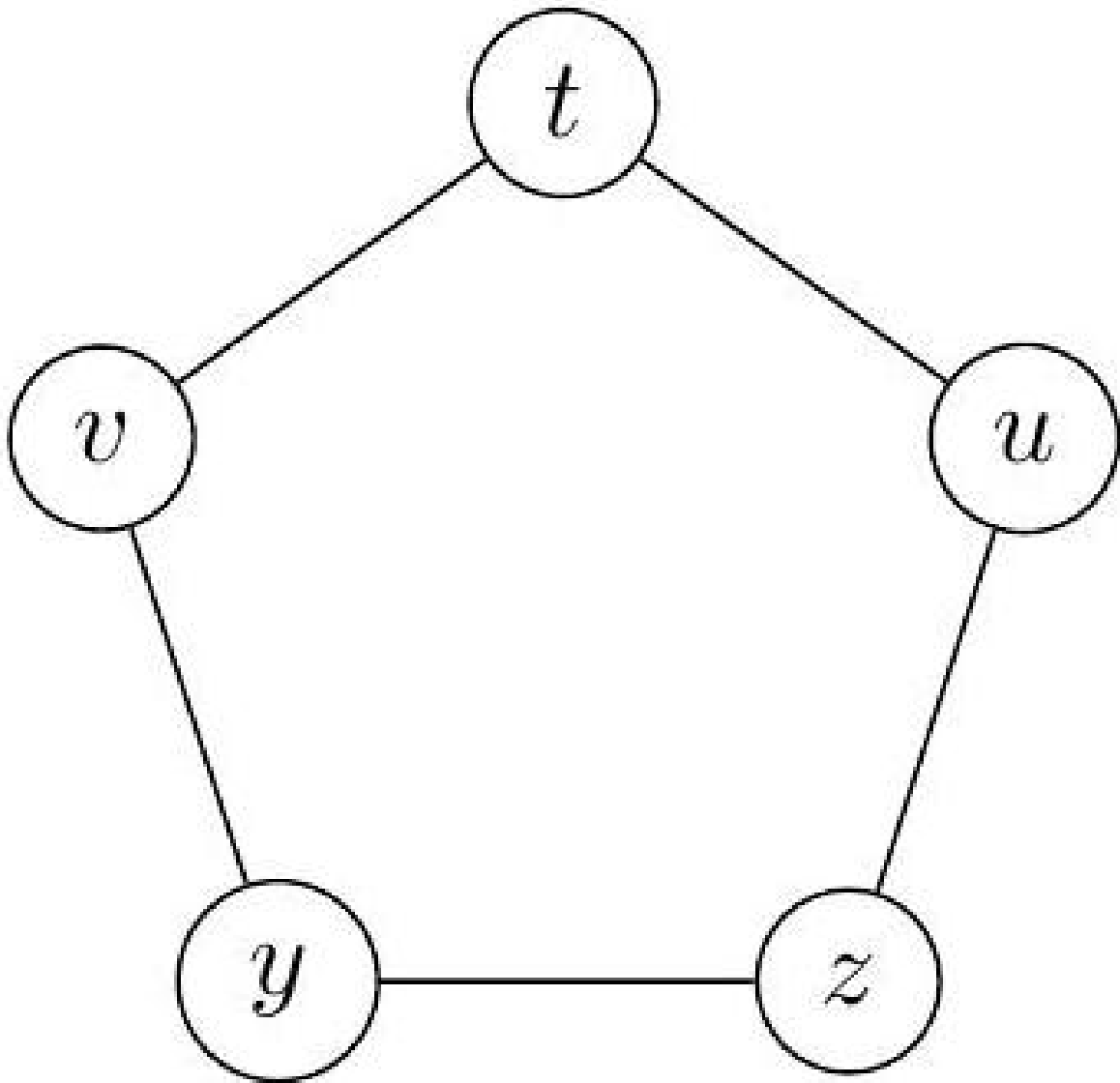
correction

On a $y \in N(y) \subset N(y) \cup N(z)$. Pour $t \neq z, t * z \in N(x) = \{y, z\}$ donc $z * t = y$ d'où $t \in N(y)$ ou $z * t = y$ d'où $t \in N(z)$, on a bien $t \in N(y) \cup N(z)$.

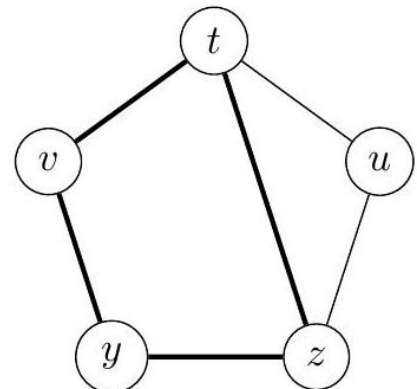
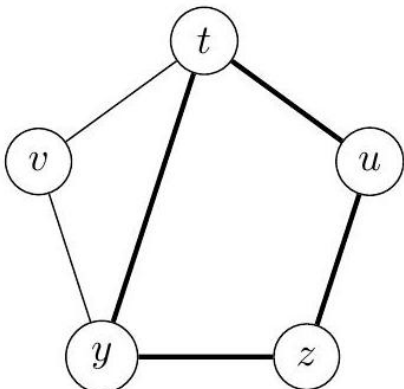
- c. Montrer que y ou z est un diplomate.

correction

On suppose que ni y ni z ne sont des diplomates. Il existe donc un sommet u dans $V \setminus \{y\}$ non voisin de y . D'après la question précédente $u \in N(z)$. De même il existe $v \in N(y) \setminus N(z)$ avec $v \neq z$.
 $u \neq z$ car $z \in N(y)$ et aussi $v \neq y$. Le sommet $t = u * v$ est distinct de u et v . De plus $t \neq y$ car y n'est pas voisin de u et $t \neq z$ pour la même raison. On a 5 sommets distincts y, z, u, v, t .



D'après la question précédente $t \in N(y) \cup N(z)$. Si on a $t \in N(y)$ alors on a un 4-cycle (y, z, u, t) , ce qui est impossible pour un graphe d'ami. De même $t \in N(z)$ est impossible. y ou z doit être 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

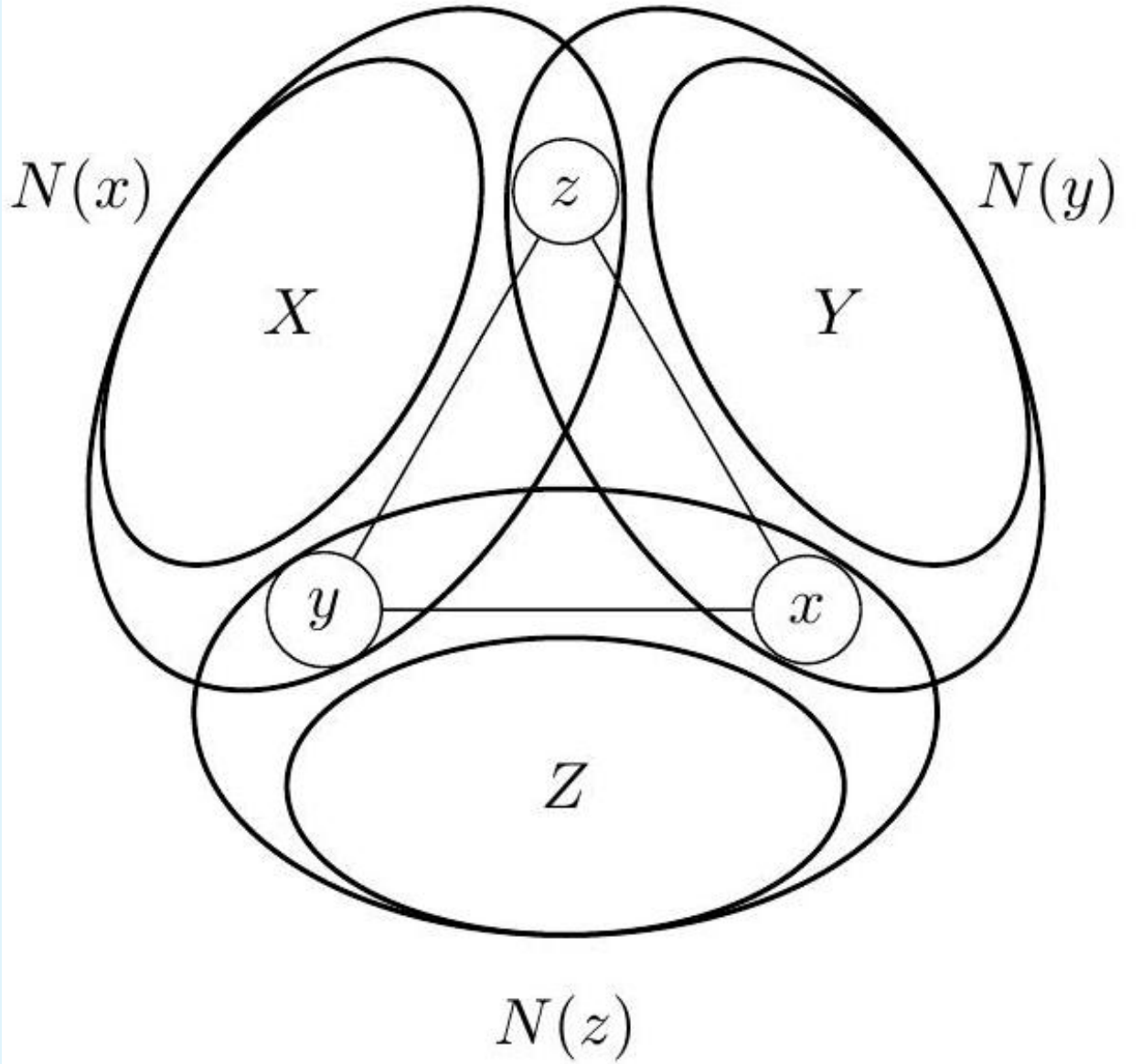
$$W = V \setminus (X \cup Y \cup Z \cup \{x, y, z\}).$$

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

$$f_* : X \times Y \rightarrow W \\ (a, b) \mapsto a * b.$$

correction

La définition d'un graphe d'ami signifie que, pour $x \neq y$, et $z = x * y$, $N(y) \cap N(x) = \{z\}$. On en déduit que $N(x) \setminus \{z\}$ et $N(y) \setminus \{z\}$ sont disjoints. Or $X \subset N(x) \setminus \{z\}$ et $Y \subset N(y) \setminus \{z\}$ donc X et Y sont disjoints. On a aussi $x * z = y$ et $y * z = x$ donc X et Z sont disjoints ainsi que Y et Z .



L'union des éléments est une partition : $V = W \sqcup X \sqcup Y \sqcup Z \sqcup \{x, y, z\}$.

Comme X et Y sont disjoints, $a * b$ est défini pour tous $a \in X$ et $b \in Y$.

$a \notin N(y)$ donc $a * b \neq y$, $a \notin N(z)$ donc $a * b \neq z$, $b \notin N(x)$ donc $a * b \neq x$.

On ne peut pas avoir $a * b \in X$ car sinon on aurait le 4-cycle $(x, a * b, b, t)$.

On ne peut pas avoir $a * b \in Y$ car sinon on aurait le 4-cycle $(y, a * b, a, x)$.

On ne peut pas avoir $a * b \in Z$ car sinon on aurait le 4-cycle $(z, a * b, a, x)$.

Ainsi f_* est bien définie de $X \times Y$ vers W .

Si on avait $a * b = a' * b'$ avec $a, a' \in X$ et $b, b' \in Y$ alors $(x, a, a * b, a')$ serait un 4-cycle, ce qui est impossible : f_* est injective.

Pour $t \in W$, on a $t * x \in N(x)$. Comme t n'appartient ni à $N(y)$, ni à $N(z)$ on ne peut pas avoir $t * x = y$ ni $t * x = z$ donc $t * x \in X$. De même $t * y \in Y$.

Comme t est voisin de $t * x$ et de $t * y$, on a $t = f_*(t * x, t * y)$: f_* est surjective.

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

correction

Si on note $|E|$ le cardinal d'un ensemble E , la bijection précédente implique $|W| = |X \times Y| = |X| \cdot |Y|$. Dans la question précédente x, y et z jouent des rôles symétriques donc on a aussi $|W| = |X| \cdot |Z| = |Y| \cdot |Z|$.

Comme on a supposé qu'aucun élément n'est de degré 2, on a Z non vide d'où $|Z| > 0$. L'égalité $|X| \cdot |Z| = |Y| \cdot |Z|$ donne donc $|X| = |Y|$ dès que x et y sont voisins. Comme le degré de X est égal à $|X| + 2$, on ajoute y et z , deux sommets adjacents ont même degré.

Comme G est connexe, il existe un chemin entre x et tout sommet de G , le long de ce chemin le degré est conservé donc le degré de tout sommet est égal au degré de x , le degré est constant sur G .

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

correction

On a $|X| = |Y| = |Z| = \delta - 2$ donc $|W| = (\delta - 2)^2$ et la partition de V donne $n = |V| = |W| + |X| + |Y| + |Z| + 3 = (\delta - 2)^2 + 3(\delta - 2) + 3 = \delta^2 - \delta + 1$.

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 δ 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.

correction

Si on note $m_{i,j}$ les coefficients de M , on a $m_{i,j} \in \{0, 1\}$ donc $m_{i,j}^2 = m_{i,j}$ et $m_{i,j} = m_{j,i}$.

Les coefficients de M^2 sont $a_{i,j} = \sum_{k=1}^n m_{i,k} m_{k,j} = \sum_{k=1}^n m_{k,i} m_{k,j}$.

Pour $i \neq j$, $m_{k,i} m_{k,j} = 0$ sauf si k est voisin de i et de j donc seulement pour $k = i * j$ donc $a_{i,j} = 1$.

Pour $i = j$, $a_{i,i} = \sum_{k=1}^n m_{k,i}^2 = \sum_{k=1}^n m_{k,i}$, c'est le nombre de k voisins de i , c'est-à-dire le degré de i d'où $a_{i,i} = \delta$.

$$\text{Ainsi } M^2 = \begin{pmatrix} \delta & 1 & \cdots & 1 \\ 1 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 1 & \cdots & 1 & \delta \end{pmatrix} = (\delta - 1)I_n + \mathbf{1}_n.$$

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 δ^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.

correction

M est diagonalisable de valeurs propres $\lambda_1, \lambda_2, \dots, \lambda_n$ donc les valeurs propres de M^2 sont $\lambda_1^2, \lambda_2^2, \dots, \lambda_n^2$ et doivent être égales à δ^2 et $\delta - 1$ $n - 1$ fois.

Le vecteur propre de M^2 pour $\delta^2 = \delta - 1 + n$ est $(1 \ 1 \ \cdots \ 1)^t$ qui est aussi vecteur propre de M pour la valeur propre δ car $\sum_{k=1}^n m_{k,i}^2 = \sum_{k=1}^n m_{i,k} = \delta$; on retrouve $\delta^2 = \delta - 1 + n$.

Les autres valeurs propres de M sont $\sqrt{\delta - 1}$ d'ordre p et $-\sqrt{\delta - 1}$ d'ordre q avec $p + q = n - 1$.

La trace de M est donc $\delta + p \cdot \sqrt{\delta - 1} + q \cdot (-\sqrt{\delta - 1})$. Or les termes diagonaux de M sont nuls, i n'est pas voisin

de i d'où $\delta + (p - q) \cdot \sqrt{\delta - 1} = 0$ et $\sqrt{\delta - 1} = \frac{\delta}{q - p} \in \mathbb{Q}$.

D'après le théorème rappelé, on a $\sqrt{\delta - 1} \in \mathbb{N}$.

Si on pose $\sqrt{\delta - 1} = m$ alors $\delta = 1 + m^2 = (q - p) \cdot m$ donc m divise $1 + m^2$, ce qui impose $m = 1$ puis $\delta = 2$ ce qui contredit l'inexistence de sommets de degré 2.

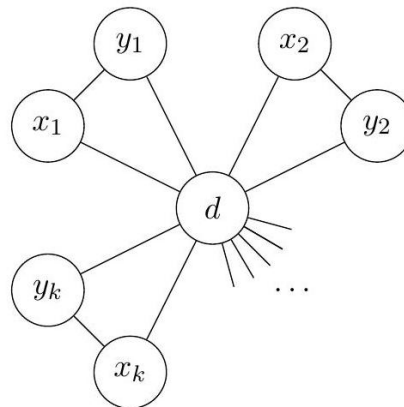
Ainsi un graphe d'amis doit avoir au moins un sommet de degré 2 donc, d'après la question 2, admet un diplomate.

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 $E = \bigcup_{i=1}^k \{\{x_i, y_i\}, \{d, x_i\}, \{d, y_i\}\}$.

correction

Si d est un diplomate du graphe d'amis G et si x est un sommet distinct de d alors d est un voisin de x et tous les autres voisins de x sont des voisins communs à d et à x . Comme G est un graphe d'amis, x a un unique voisin distinct de d , il est de degré 2. Les voisins de x sont d et $x * d$. Dans ce cas les voisins de $y = x * d$ sont d et $y * d = x$.

On peut donc regrouper les sommets distincts de d par paires (x_i, y_i) avec $N(x_i) = \{y_i, d\}$ et $N(y_i) = \{x_i, d\}$ et G est l'union des 3-cycles (x_i, y_i, d) .



Graphe moulin

Chapitre 55

(ENS) Structures pliages et traversables *** (ENS 23, partiellement corrigé — oral - 306 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 $(o) : (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C)$ de composition de fonctions, noté `o` dans sa version infixe ;
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 `liste A` \rightarrow `int` en pseudocode (par exemple) :

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

```
longueur : liste A  $\rightarrow$  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  $\rightarrow$  int = function  
  Nil  $\rightarrow$  0  
  | Cons (_, qu)  $\rightarrow$  1+ (longueur qu)
```

Un type T est un *monoïde*, s'il existe (c'est-à-dire, si on peut définir) ε de type T et (\diamond) de type $T \rightarrow T \rightarrow T$, noté \diamond dans sa version infixe, 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 .

correction

Correction de M.Péchaud

\diamond correspond à l'opérateur de concaténation.

On définit $\varepsilon = \text{Nil}$, et \diamond inductivement par, pour toutes listes L, L' de type `liste A` et pour tout élément e de type A .

- $\text{Nil} \diamond L' = L'$
- $(\text{Cons } e L) \diamond L' = \text{Cons } e (L \diamond L')$.

On vérifie alors que $L \diamond \varepsilon = L$ par induction sur L :

Cas de base $\varepsilon \diamond \varepsilon = \text{Nil} \diamond \varepsilon = \varepsilon$ par définition.

Induction On suppose le résultat vrai pour L . Alors $(\text{Cons } e L) \diamond \varepsilon = \text{Cons } e (L \diamond \varepsilon) = \text{Cons } e L$

On montre alors par une induction similaire que pour toutes listes L, L' et L'' , $(L \diamond L') \diamond L'' = L \diamond (L' \diamond L'')$:

Cas de base $(\varepsilon \diamond L') \diamond L'' = L' \diamond L'' = \varepsilon \diamond (L' \diamond L'')$

Induction On suppose le résultat vrai pour L . Alors $((\text{Cons } e L) \diamond L') \diamond L'' = (\text{Cons } e (L \diamond L')) \diamond L'' = \text{Cons } e ((L \diamond L') \diamond L'') = \text{Cons } e (L \diamond (L' \diamond L''))$ par hypothèse d'induction
 $= (\text{Cons } e L) \diamond (L' \diamond L'')$.

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

$$(A \rightarrow B \rightarrow B) \rightarrow B \rightarrow C A \rightarrow B$$

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 ?

correction

On définit `fold` par induction sur t .

```
fold : (A -> B -> B) -> B -> C A -> B
      fold f e Nil = e
      fold f e (Cons a L) = f a (fold f e L)
```

Il n'y a pas unicité (par analogie avec les `fold_left` et `fold_right` de *OCaml*) : on peut définir un `fold` traitant les éléments par ordre inverse – ce qui peut donner a priori un résultat différent :

```
fold_right : (A -> B -> B) -> B -> C A -> B
            fold_right f e Nil = e
            fold_right f e (Cons a L) = fold_right f (f a e) L
```

Question 3.

- 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 .

correction

```
barbre A = Vide | Noeud A (barbre A) (barbre A)
```

- Montrer que `barbre` est pliable.

correction

On définit par exemple

```
fold : (A -> B -> B) -> B -> C A -> B
      fold f e Vide = e
      fold f e (Noeud a T T') = fold f (f a (fold f e T)) T'
```

- Expliquer comment différentes définitions de `fold` permettent de générer les parcours infixe et préfixe d'un arbre.

correction

Le `fold` donné précédemment correspond à un parcours infixe – car le premier calcul de f est effectué dans le sous-arbre gauche.

Pour un parcours préfixe, on donne la définition suivante, où la racine est traitée en premier :

```
fold : (A -> B -> B) -> B -> C A -> B
```

```

fold f e Vide = e
fold f e (Noeud a T T') = fold f (fold (f a e) T) T'

```

Question 4. Utiliser `fold` pour écrire `arbreTaille` : `int → liste (barbre unit)`, la fonction générant une liste des arbres binaires de taille n (c'est-à-dire, à n nœuds) avec les nœuds prenant leur valeur dans `unit`.

correction

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.

correction

Pour les listes :

```

foldmap : (A -> M) -> liste A -> M
  foldmap f Nil = ε
  foldmap f (Cons a l) -> (f a) ◊ (foldmap f l)

```

Pour les arbres par parcours préfixe

```

foldmap : (A -> M) -> barbre A -> M
  foldmap f Vide = ε
  foldmap f (Noeud a t t') -> (f a) ◊ (foldmap f t) ◊ (foldmap f t')

```

et par parcours infixes

```

foldmap : (A -> M) -> barbre A -> M
  foldmap f Vide = ε
  foldmap f (Noeud a t t') -> (foldmap f t) ◊ (f a) ◊ (foldmap f t')

```

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

correction

⇒ Supposons disposer d'un `fold` : $(A \rightarrow B \rightarrow B) \rightarrow B \rightarrow C A \rightarrow B$, et donnons-nous un monoïde quelconque, muni d'un neutre ε et d'un opérateur \diamond .

On pose alors `foldmap f s = fold (fun a m -> (f a) ◊ m) ε s`

⇐ Réciproquement, si l'on suppose disposer d'un `foldmap` pour un monoïde quelconque, et d'un type B , on remarque que l'ensemble des applications de B dans B est un monoïde (l'élément neutre est l'identité, et \diamond la composition).

On se donne un `foldmap` agissant sur ce monoïde, et l'on définit alors

```

fold f e t = (foldmap f t)e

```

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 infixes.

qui vérifie les lois suivantes :

```

fmap id = id   fmap(f ◊ g) = (fmap f) ◊ (fmap g)

```

```

(pure id) ◊ v = v

```

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

correction

On peut se donner `fmap = map` (le `map` de *OCaml*), `pure = fun a -> [a]`, et `⊙ = fun [f1 ... fn] l -> ↔
fmap f1 l @ ... fmap fn l – où @ est l'opérateur de concaténation (on pourrait bien sûr définir cela plus
formellement sans «...»).`

On peut également définir `⊙ = fun [f1 ... fn] [l1 ... lp] -> [f1l1; ... fnl1; f2l2; ... fnl2; ... f1lp; ... fnlp]`.

L'idée est en quelque sorte que l'opérateur `⊙` applique toutes les fonctions de son membre de gauche à tous les
éléments de son membre de droite, de toutes les façons possibles.

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.

correction

NDMP : le niveau déjà très élevé avant me paraît déraisonnable à partir de cette question.

Question 9. On dispose d'un arbre t de type `barbre A` et une fonction `futurs` qui à un élément A associe une 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 Γ 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.

correction

On propose :

$$\frac{\frac{\frac{\frac{}{\vdash x, x^\perp} \text{ ax}}{\vdash x^\perp \otimes y^\perp, x, y} \otimes \quad \frac{\frac{}{\vdash y, y^\perp} \text{ ax}}{\vdash z, z^\perp} \text{ ax}}{\vdash (x^\perp \otimes y^\perp) \otimes z^\perp, x, y, z} \otimes}{\vdash ((x^\perp \otimes y^\perp) \otimes z^\perp) \wp (x \wp (y \wp z))} \wp \times 3}{\vdash ((x \wp y) \wp z)^\perp \wp (x \wp (y \wp z))} \wp}{\vdash ((x \wp y) \wp z) \multimap (x \wp (y \wp z))} \multimap$$

2. Montrer que pour $A \in \mathcal{F}$, $\vdash A, A^\perp$ est prouvable en logique linéaire.

correction

On raisonne par induction sur A :

- si $A = x \in \mathcal{V}$, alors $\frac{}{\vdash x, x^\perp}$ ax est une preuve de $\vdash A, A^\perp$.
- supposons que $\vdash B, B^\perp$ et $\vdash C, C^\perp$ soient prouvables. Distinguons selon la formule A :

— si $A = B^\perp$, alors $\frac{\vdash B^\perp, B}{\vdash B^\perp, (B^\perp)^\perp}$ est une preuve de $\vdash A, A^\perp$;

— si $A = B \wp C$, alors $\frac{\frac{\frac{\vdash A, A^\perp}{\vdash B, B^\perp} \quad \vdash C, C^\perp}{\vdash B \wp C, B^\perp \otimes C^\perp} \otimes}{\vdash B \wp C, (B \wp C)^\perp}$ est une preuve de $\vdash A, A^\perp$;

— si $A = B \otimes C$, alors $\frac{\frac{\frac{\vdash A, A^\perp}{\vdash B, B^\perp} \quad \vdash C, C^\perp}{\vdash B \otimes C, B^\perp \wp C^\perp} \otimes}{\vdash B \otimes C, (B \otimes C)^\perp}$ est une preuve de $\vdash A, A^\perp$.

On conclut par induction.

On considère la règle d'inférence $\frac{}{\vdash A, A^\perp}$ 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).

correction

On a l'arbre de preuve suivant :

$$\frac{\frac{\frac{\vdash \Delta, A \quad \frac{}{\vdash B, B^\perp} \text{ax}}{\vdash \Delta, B, A \otimes B^\perp} \otimes}{\vdash \Gamma, A \multimap B \quad \vdash \Delta, B, (A \multimap B)^\perp} \text{cut}}{\vdash \Gamma, \Delta, B}$$

4. Montrer que $\vdash A \multimap (A \otimes A)$ n'est pas prouvable sans coupure.

correction

On a $A \multimap (A \otimes A) = A^\perp \wp (A \otimes A)$. Sans coupure, la dernière règle à appliquer doit nécessairement être la règle (\wp). On doit donc prouver $A^\perp, A \otimes A$. Seule la règle (\otimes) est alors applicable, ce qui nécessite de prouver A^\perp, A d'une part et A d'autre part. Mais pour $A = x \in \mathcal{V}$ par exemple, une telle preuve sans coupure n'existe pas.

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.

correction

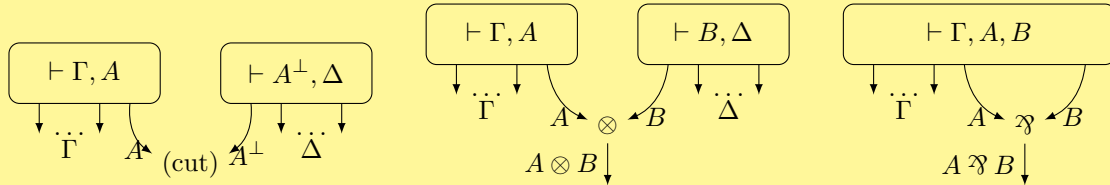
On remarque que les seuls sommets qui ont à la fois des arêtes entrantes et sortantes sont les sommets (\otimes) et (\wp). Pour chacun de ces sommets, on remarque que si A et B sont les étiquettes des arêtes entrantes et C l'étiquette de l'arête sortante, alors $|C| > |A|$ et $|C| > |B|$ (où $|A|$ est la taille de la formule, c'est-à-dire le nombre de symboles).

S'il existe un cycle orienté, ce cycle est de la forme $s_0 \xrightarrow{A_1} s_1 \xrightarrow{A_2} \dots \xrightarrow{A_n} s_n$, avec $s_n = s_0$. Par ce qui a été dit plus haut, on a, pour $i \in \llbracket 1, n-1 \rrbracket$, $|A_i| < |A_{i+1}|$. De même, $|A_n| < |A_1|$. Mais alors, par transitivité, $|A_1| < |A_1|$, ce qui est absurde.

Définition

Un **réseau de preuve** est un réseau dont les conclusions forment le multi-ensemble Γ , 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.

correction

Le réseau avec un sommet (ax) dont les arêtes sont reliées à un sommet (cut) 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-Régner

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-Régner.

correction

Par induction sur les arbres de preuve :

- pour un axiome $\frac{}{\vdash x, x^\perp}$ ax, tous les interrupteurs sont un graphe à un seul sommet, qui est effectivement un graphe connexe sans cycle.
 - pour une règle $(\otimes) \frac{\vdash \Gamma, A \quad \vdash \Delta, B}{\vdash \Gamma, A \otimes B, \Delta}$ \otimes , supposons que les réseaux \mathcal{S} et \mathcal{S}' de conclusions Γ, A et Δ, B ont des interrupteurs connexes sans cycle. Alors un interrupteur associé au réseau de conclusions $\Gamma, A \otimes B, \Delta$ s'obtient en ajoutant un sommet (\otimes) et une arête le reliant à \mathcal{S} et une à \mathcal{S}' . Cela forme bien un graphe connexe sans cycle.
 - pour une règle $(\wp) \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \wp B}$ \wp , supposons que le réseau \mathcal{S} de conclusions Γ, A, B ait des interrupteurs connexes sans cycle. Alors un interrupteur associé au réseau de conclusions $\Gamma, A \wp B$ s'obtient en ajoutant un nœud (\wp) et en le reliant à \mathcal{S} par une arête. Cela donne bien un graphe connexe sans cycle.
 - pour une règle (cut), on raisonne comme pour (\otimes) .
- On conclut par induction.

On admet la réciproque du théorème de Danos-Régner, 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).

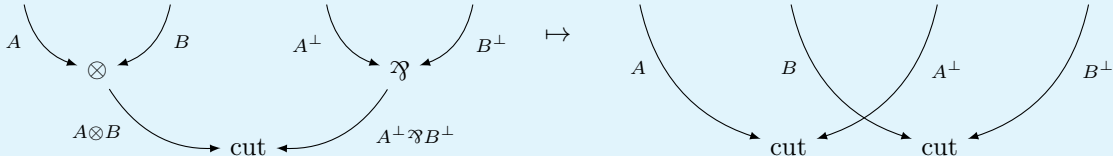
On considère pour cela un réseau de preuve \mathcal{R} , contenant éventuellement des sommets (cut). On cherche à transformer ce réseau en supprimant les nœuds (cut), tout en conservant le critère de Danos-Régner (c'est-à-dire que les interrupteurs restent des arbres), et sans changer les conclusions.

8. Montrer qu'un sommet (cut) a pour prédécesseur un sommet (\otimes) si et seulement s'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.

correction

On sait qu'un nœud (cut), dans le cas général, a une prémisse A et une prémisse A^\perp . Ainsi, si l'un des deux côtés est un nœud (\otimes), de conclusion $A \otimes B$, alors l'autre côté est nécessairement de conclusion $(A \otimes B)^\perp = A^\perp \wp B^\perp$, qui est bien conclusion d'un nœud (\wp).

Dès lors, on applique la transformation :



On suppose que $a(A)$ et $b(B)$ sont les arêtes prémisses du nœud (\otimes), et que $c(A^\perp)$ et $d(B^\perp)$ sont les arêtes prémisses du nœud (\wp).

Soit I un interrupteur du réseau \mathcal{R}' obtenue après élimination de la coupure.

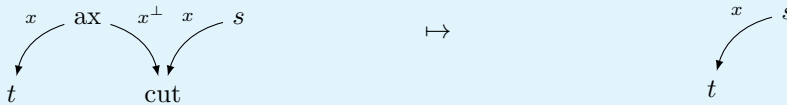
- I est acyclique : s'il existe un cycle dans I , s'il passe uniquement par la coupure $a - c$ (resp. $b - d$), alors dans l'interrupteur dans \mathcal{R} où on déconnecte d (resp. c) du nœud (\wp), ce cycle existe dans \mathcal{R} , ce qui est absurde. Sinon, c'est un cycle de toute façon dans tout interrupteur de \mathcal{R} ;
- I est connexe : soit J l'interrupteur de \mathcal{R} obtenu à partir de I où on déconnecte c du nœud (\wp). Par connexité, toute arête est reliée à a, b, c ou d . Dans I , il y a donc au plus deux composantes connexes : celle reliée à $a - c$ et celle à $b - d$. Cependant, s'il existe un autre chemin de c à a ou b que l'arête $a - c$, il existerait un cycle. c est donc reliée à b via l'arête $c - d$, donc le graphe est connexe.

Par ailleurs, les conclusions n'ont pas changé, et le nombre de sommets a diminué de 1.

9. Montrer le théorème d'élimination des coupures.

correction

Si un sommet (cut) a pour prédécesseur un (ax), on applique la transformation suivante :



Cela garde bien les propriétés voulues.

Cette transformation et la question précédente donnent l'idée d'un algorithme qui élimine les coupures, sans changer les conclusions, et en continuant à vérifier le critère de Danos-Régnier. L'algorithme termine car le nombre de sommets est un variant. Une fois terminé, le réseau obtenu ne contient plus de coupure et c'est un réseau de preuve. La preuve associée n'utilise pas de coupure.

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 $(\circ) : (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C)$ de composition de fonctions, noté \circ dans sa version infixe ;
- d'un opérateur \mapsto 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* \rightarrow *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*.

correction

Début de correction de M.Péchaud
NDMP : ce sujet ne me paraît pas raisonnable

```
1 option A = Vide | Some A
```

Un constructeur de types F est un *foncteur* quand il existe une fonction polymorphe

$$\text{fmap} : (A \rightarrow B) \rightarrow (FA) \rightarrow (FB)$$

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

$$\text{fmap id} = \text{id}$$

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

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.

correction

Pour `option`, on définit

```
let fmap f (a : 'a option) = match a with
  None -> None
  | Some a -> Some (f a)
```

Pour `list`, on utilise le `map` habituel :

```
let rec fmap f (l : 'a list) = match l with
  [] -> []
  | t :: q -> f t :: (fmap f q)
```

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

$$\text{return} : A \rightarrow (M A)$$

$$\text{bind} : (M A) \rightarrow (A \rightarrow (M B)) \rightarrow (M B)$$

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

$$\text{bind } (\text{return } x) f = f x$$

$$\text{bind } x \text{return} = x$$

$$\text{bind } (\text{bind } x f) g = \text{bind } x (y \mapsto (\text{bind } (f y) g))$$

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

correction

Pour `option` :

```
let return a = Some a ;;
let bind m f = match m with
  None -> None
  | Some a -> Some (f a)
```

Pour `liste` :

```
let return a = [a] ;;
let rec bind l f =
  match l with
  | [] -> []
  | t :: q -> f t @ bind q f
```

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` \rightarrow `memoire` \rightarrow (`option int`)

`majour` : `string` \rightarrow `int` \rightarrow `memoire` \rightarrow `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` : $A \rightarrow (M A)$

`join` : $M (M A) \rightarrow M A$

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

`join` \circ `return` = `id`

`join` \circ (`fmap` `return`) = `id`

`join` \circ `join` = `join` \circ (`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*) Π 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 Π , notée $size(\Pi)$, est son nombre de noeuds. $Termes(\Pi)$ dénote l'ensemble des étiquettes, i.e., termes, apparaissant dans Π .

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

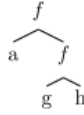


Figure 2 - Représentation sous forme d'arbre du terme $f(a, f(g, h))$

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 .

correction

$$\frac{\frac{\overline{T \vdash f(f(a, k_1), k_2)} \text{Ax}}{T \vdash f(a, k_1)} \text{RED-F} \quad \frac{\overline{T \vdash k_2} \text{Ax}}{\overline{T \vdash f(k_1, k_2)} \text{Ax} \quad \overline{k_2} \text{Ax}} \frac{T \vdash k_1}{T \vdash a} \text{REF-F}}{T \vdash a} \text{RED-F}$$

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 Π 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 Π 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. $\text{Termes}(\Pi) \subseteq \text{st}(T)$.

correction

On fait une preuve par induction sur l'arbre de preuve.
 Si l'arbre de preuve est réduit à la règle Ax alors la preuve est immédiate pour le cas général et le cas particulier.
 Sinon, l'arbre de preuve se termine par une autre règle. Nous pouvons faire une étude de cas :

- App-F : par hypothèse d'induction, la propriété que nous souhaitons prouver est vraie.
- Red-F : par minimalité de l'arbre de preuve, nous savons que les sous-arbres Π_1 et Π_2 ne terminent pas par une règle App-F. En effet, si tel était le cas, alors nous pourrions construire un arbre plus petit en omettant les deux dernières étapes. Par conséquent, Π ne contient que des termes issus de $\text{st}(T)$.

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 : $\text{size}(T, u) = |\text{st}(u)| + \sum_{t \in T} |\text{st}(t)|$.

correction

- On calcule tous les sous-termes de $T \cup \{u\}$
- Tant qu'on n'a pas atteint un point fixe, on sature T avec les termes déductibles en une étape (si le terme déduit n'est pas dans $\text{st}(T) \cup \{u\}$, alors on ne l'ajoute pas). Le nombre maximal d'itérations est $|\text{st}(T) \cup \{u\}|$. On remarquera que $|\text{st}(T)|$ est au plus quadratique en la taille du problème car chaque sous terme d'un terme de T est plus petit que ce même terme.
- si u est dans l'ensemble saturé alors on retourne « oui », sinon on retourne « non ».

On définit le problème HORN-SAT :
 entrée une formule Φ étant une conjonction finie de clauses de Horn
 sortie est-ce que Φ 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.

correction

Le problème de déduction est clairement dans P avec la question précédente (la taille du problème est le nombre de sous-termes).

L'idée est la suivante :

- nous allons associer à chaque variable propositionnelle x , une constante $v_x \in \mathcal{C}$
- nous allons remarquer que le terme t est déductible depuis $f(\dots f(f(t, t_1), t_2), \dots, t_n)$ si et seulement si t_1, \dots, t_n le sont aussi.

Pour montrer que le problème de déduction est P-complet, on peut utiliser HORN-SAT. Soit $\Phi = C_1 \wedge \dots \wedge C_n$ une conjonction finie de clauses de Horn. Nous construisons :

1. Pour tout $x \in \mathcal{V}$, on associe une constante $v_x \in \mathcal{C}$. On définit également une constante v_\perp .
2. pour tout i , on définit

$$t_i = \begin{cases} f(\dots f(f(v_x, v_{x_1}), v_{x_2}), \dots, v_{x_n}) & \text{si } C_i = (x_1 \wedge \dots \wedge x_n \Rightarrow x) \\ f(\dots f(f(v_\perp, v_{x_1}), v_{x_2}), \dots, v_{x_n}) & \text{si } C_i = (x_1 \wedge \dots \wedge x_n \Rightarrow \text{false}) \\ v_x & \text{si } C_i = (\text{true} \Rightarrow x) \end{cases}$$

3. Soit $T = \{t_1, \dots, t_n\}$. Par construction, $T \vdash v_x$ implique $x = \text{true}$ (pour tout v_x apparaissant dans Φ) dans toute valuation satisfaisant Φ .
Par conséquent, $T \vdash v_\perp$ implique Φ est non-satisfiable. Réciproquement, si $T \not\vdash v_\perp$ alors nous définissons la valuation $\{x \rightarrow 1 \mid T \vdash v_x\} \cup \{x \rightarrow 0 \mid T \not\vdash v_x\}$ qui satisfait Φ .

Nous souhaiterions maintenant nous intéresser au même problème mais en ajoutant le ou-exclusif. Un terme est donc maintenant généré par la grammaire :

$$t, t_1, t_2 := v \in \mathcal{C} \mid x \in \mathcal{V} \mid f(t_1, t_2) \mid t_1 \oplus t_2$$

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' ?

$$\frac{T \vdash u_1 \dots T \vdash u_n \text{ GX}}{T \vdash u_1 \oplus \dots \oplus u_n} \quad \frac{\text{si } u = ACv \text{ et } v \in T \text{ Ax}}{T \vdash u}$$

On note $=_{AC}$ la plus petite relation telle que :

$$\begin{aligned} (\text{refl.}) \quad & x = x & (\text{sym.}) \quad & (x = y) \Rightarrow (y = x) & (\text{trans.}) \quad & (x = y) \wedge (y = z) \Rightarrow (x = z) \\ (\text{comm.}) \quad & x \oplus y = x \oplus y & (\text{assoc.}) \quad & x \oplus (y \oplus z) = (x \oplus y) \oplus z \\ (\text{congr.}) \quad & (x_1 = y_1) \wedge (x_2 = y_2) \Rightarrow f(x_1, x_2) = f(y_1, y_2) \end{aligned}$$

Question 5. Soit u et v deux termes clos. Donner un algorithme en temps polynomial qui décide si $u =_{AC} v$.

correction

Deux remarques ici :

- On peut se restreindre à une terme de la forme $u_1 \oplus \dots \oplus u_n$ avec u_i ne contenant pas de \oplus .
- On peut simplifier le multi-ensemble obtenu en fonction du nombre d'occurrences de chaque facteur. On garde le facteur si son nombre d'occurrences est impair, on l'enlève s'il est pair. Le facteur 0 peut être retiré du multi-ensemble. On note cette procédure de simplification Simplify (\cdot).

Étant donné que l'on raisonne modulo AC, on peut remarquer que sans perte de généralité, on peut aplatir tous les \oplus , i.e., considérer \oplus comme un opérateur n-aire.

Étant donné que u et v sont clos, il suffit de calculer le multi-ensemble des facteurs de u et v en se donnant pour

$$\text{définition : Facteurs}(t) = \begin{cases} \bigcup_{i=1}^n \text{Facteurs}(t_i) & \text{si } t = t_1 \oplus \dots \oplus t_n \\ \{t\} & \text{sinon} \end{cases}$$

L'algorithme est comme suit :

$$\text{Egal}(u, v) = \begin{cases} \perp & \text{si } u = f(t_1, t_2) \text{ and } v \neq f(t'_1, t'_2) \\ \text{Egal}(t_1, t'_1) \wedge \text{Egal}(t_2, t'_2) & \text{si } u = f(t_1, t_2) \text{ and } v = f(t'_1, t'_2) \\ \text{Simplify}(\text{Facteurs}(u)) == \text{Simplify}(\text{Facteurs}(v)) & \text{sinon} \end{cases}$$

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.

correction

On peut commencer par remarquer que sans perte de généralité, on peut supposer que notre arbre est de hauteur 2 : une unique application de GX et ensuite uniquement des applications de Ax'. En effet, si on a deux étages, on peut juste les « aplatis » et on obtient toujours un arbre de preuve valide.

On définit **Facteurs**(t) et **Simplify**(S) comme dans la réponse à la question précédente.

Voici l'algorithme :

1. On calcule $S_t = \text{Simplify}(\text{Facteurs}(t))$ et $S_T = \text{Simplify}(\text{Facteurs}(T))$ (temps polynomial)
2. Si $S_t \not\subseteq S_T$ alors t n'est pas déductible à partir de T . En effet, il existe un sous-terme de t que nous ne pourrions jamais construire à partir de T (temps polynomial)
3. Sinon on représente S_t comme un vecteur de taille $|S_T|$ avec pour valeurs 0/1. Le vecteur a un 1 en position i si le i -ème facteur de S_t est aussi présent dans S_T . Sinon, on affecte la valeur 0. Notons ce vecteur V^t .
4. On peut définir les mêmes vecteurs pour les différents termes de T pris individuellement, notons les V_i^T
5. la déductibilité de t est maintenant réduite à l'existence d'une combinaison linéaire dans \mathbb{F}_2^p des vecteurs V_i^T telle que $\sum_i \alpha_i V_i^T = V^t$. Cette étape se calcule en temps polynomial à l'aide d'un pivot de Gauss.

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 σ .

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 $m \leq_\sigma v$ si et seulement si $\exists \sigma, m \leq_\sigma v$.

correction

Ébauche de solution proposée par le rapport du Jury.

- $x \leq_\sigma v \Leftrightarrow \sigma(x) = v$
- $\leq_\sigma v$
- $C(m_1, \dots, m_n) \leq_\sigma C'(v_1, \dots, v_n) \Leftrightarrow C = C' \wedge n = o \wedge \forall i, m_i \leq_\sigma v_i$
- $(m_1 \mid m_2) \leq_\sigma v \Leftrightarrow m_1 \leq_\sigma v \vee m_2 \leq_\sigma v$

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 σ 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 $Match$?

correction

Il faut que $\forall 1 \leq j < i, (v_1, \dots, v_m) \not\subseteq (m_{j,1}, \dots, m_{j,m})$, et que $(v_1, \dots, v_m) \sqsubseteq_\sigma (m_{i,1}, \dots, m_{i,m})$.

Ne pas oublier le cas où $\forall 1 \leq i \leq n, (v_1, \dots, v_m) \not\subseteq (m_{i,1}, \dots, m_{i,m})$, dans ce cas une exception est levée (Matching_failure en OCaml).

Question 3.

- (a) Décrire informellement comment éliminer le motif joker `_` sans ajouter de cas, sur l'exemple de len cidessous.

```
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.

correction

(a) Il faut juste ajouter des variables fraîches, c'est à dire des variables qui ne sont pas les variables libres de l'action à droite de \rightarrow (la notion de variable libre est au programme, mais du côté logique), ou les variables liées par le motif de la ligne.

Dans l'exemple, il faut donc choisir une variable qui n'est ni `l` ni `tl`.

```
let rec len l =
  match l with
  | Empty -> 0
  | Cons(w, tl) -> 1 + (len tl)
```

(b) On note $NJ(M)$ cette transformation. variables est définie récursivement pour extraire les variables définies dans un motif.

$$NJ(M)_{i,j} = \begin{cases} M_{i,j} & \text{si } M_{i,j} \neq - \\ v \text{ tel que } v \notin \text{variables_libres}(a_i) \text{ et } v \notin \{\text{variables}(m_{i,k}) \mid k \neq j\} \end{cases}$$

(c) Il faut prouver que $Match((v_1, \dots, v_m), M) = Match((v_1, \dots, v_m), NJ(M))$. On n'exigeait pas une preuve détaillée ici.

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 $S(Cons, F)$ 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.

correction

(a) Petit piège, ce n'est pas un cas de — dans les motifs ici, la matrice de filtrage initiale est donc

$$F = \left(\begin{array}{ccc} l1 & l2 & \\ \text{Empty} & l & \rightarrow l \\ l & \text{Empty} & \rightarrow l \\ \text{Cons}(h1, t1) & \text{Cons}(h2, t2) & \rightarrow \dots \end{array} \right) \quad S(\text{Cons}, F) = \left(\begin{array}{ccc} \#1(l1) & \#2(l1) & l2 \\ \overline{h1} & \overline{t1} & \text{Cons}(h2, t2) \end{array} \rightarrow \dots \right)$$

(b) N.B : formaliser la transformation sur les indices des matrices n'est pas pratique ici, les candidat.e.s ont été évalués sur leurs choix de présentation pédagogique de cette étape.

On commence par changer (e_1, \dots, e_m) en $\#1(e_1), \dots, \#a(e_1), e_2, \dots, e_m$.

On raisonne par cas sur $m_{i,1}$ pour donner $M' \rightarrow A'$. Chaque ligne peut donner au plus deux lignes correspondantes dans la matrice de filtrage de $S(c, F)$.

- Si $m_{i,1} = c(q_1, \dots, q_a)$, la ligne conservée est étendue à gauche pour donner

$$q_1, \dots, q_a, m_{i,2}, \dots, m_{i,m} \rightarrow a_i$$

- Si $m_{i,1} = c'(q_1, \dots, q_b)$ avec $c \neq c'$, la ligne est supprimée
- Si $m_{i,1} = v$ (une variable), il n'y a rien à matcher dans l'extension à gauche, et il faut étendre l'action pour lier la variable

$$\dots, \dots, \dots, m_{i,2}, \dots, m_{i,m} \rightarrow \text{let } v = e_1 \text{ in } a_i$$

- Le cas est similaire, mais il n'y a même pas besoin de faire de liaison.
- Si $m_{i,1} = q_1 \mid q_2$, on génère deux lignes :

$$S(c, (q_1, m_{i,2}, \dots, m_{i,m} \rightarrow a_i))$$

$$S(c, (q_2, m_{i,2}, \dots, m_{i,m} \rightarrow a_i))$$

(c)

$$\text{Match}((c(w_1, \dots, w_a), v_2, \dots, v_m), F) = \text{Match}((w_1, \dots, w_a, v_2, \dots, v_m), S(c, F))$$

Question 5. Donner une fonction C permettant de transformer ces matrices de filtrage vers un langage OCaml modifié, où les filtrages `match ... with ...` sont supprimés au profit de switches sur le constructeur d'une expression :

```
switch constr(e) with
  case ... -> ...
  ...
  default -> ...
```

correction

- Si $n = 0$,

$$C(F) = C(e_1 \dots e_m) = \text{failwith Matching_failure}$$

- Si $m = 0$,

$$C(F) = C \left(\begin{array}{l} \rightarrow a_1 \\ \rightarrow \dots \\ \rightarrow a_n \end{array} \right) = a_1$$

- Ce cas peut être inclus dans le suivant, mais avoir un cas particulier permet d'éviter un switch inutile. Si la première colonne ne contient que des variables ou des (pour simplifier, on suppose que ce sont des variables,

cf. question 1), ie $m_{.,1} = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}$:

$$C \left(\begin{array}{ccc} e_1 & \dots & e_m \\ x_1 & \dots & m_{1,m} \rightarrow a_1 \\ \dots & \dots & \dots \rightarrow \dots \\ x_n & \dots & m_{n,m} \rightarrow a_n \end{array} \right) = C \left(\begin{array}{ccc} e_2 & \dots & e_m \\ m_{1,2} & \dots & m_{1,m} \rightarrow \text{let } x_{-1} = e_{-1} \\ \text{in } a_1 & & \text{in } a_n. \\ \dots & \dots & \dots \rightarrow \dots \\ m_{n,2} & \dots & m_{n,m} \rightarrow \text{let } x_{-n} = e_{-1} \end{array} \right)$$

- sinon, on note $\text{Constrs} = \{c \mid m_{i,1} = c(\dots)\}$ l'ensemble des constructeurs apparaissant dans les motifs de la première colonne.

$$C(F) = \begin{array}{l} \text{switch constr } (e_1) \text{ with} \\ \text{case } c_1 \rightarrow C(S(c_1, F)) \\ \dots \\ \text{case } c_l \rightarrow C(S(c_l, F)) \\ \text{default} \rightarrow C(D(F)) \end{array}$$

L'opérateur de défaut est conceptuellement similaire à celui de spécialisation. Les lignes avec constructeurs sont supprimées. Les lignes avec variables ou wildcard sont laissées, et les — sont déroulés. La propriété est que pour tout constructeur c qui n'apparaît pas en tête d'un motif dans la première colonne, alors

$$\text{Match}((c(w_1, \dots, w_a), v_2, \dots, v_m), F) = \text{Match}((v_2, \dots, v_m), D(F))$$

Question 6.

- Cette transformation a-t-elle un intérêt ?
- L'algorithme de transformation peut-il être amélioré ?

correction

- Oui pour compiler OCaml.
- Dans le cadre d'OCaml, il est intéressant de s'intéresser aux gains qu'apportent les informations de type. Cela peut être une passe faite sur le switch, après sa génération.
Un seul constructeur : pas besoin de faire de test via un switch
Suppression du otherwise : lorsque tous les constructeurs sont déjà couverts
Chaînes de caractères : ce cas est très à la marge, mais il peut y avoir une compilation vers des arbres de switches sur les caractères.

Question 7. Prouver la terminaison et la correction de l'algorithme en question 5.

correction

Mesure de terminaison : $\sum_{i,j} \text{taille}(m_{i,j})$

$$\begin{array}{l} \text{taille}(v) = \text{taille}(_) = 1 \\ \text{taille}(q_1 \mid q_2) = \text{taille}(q_1) + \text{taille}(q_2) \\ \text{taille}(C(m_1, \dots, m_n)) = 1 + \sum_i \text{taille}(m_i) \end{array}$$

La définition de la sémantique de switch est immédiate. On note

" H(s) : si $C(F) = s$ alors

- $\text{siMatch}((v_1, \dots, v_m), F) = (a_i, \sigma)$ alors $\text{let } x_i = w_i \text{ in } a_i = \text{let } e_i = v_i \text{ in } s$
- sinon les deux cas s'évaluent en `Matching_failure`."

Le plus simple est de faire la preuve par induction sur F (vu comme l'expression `match`, sinon il n'y a pas de structure pour faire l'induction).

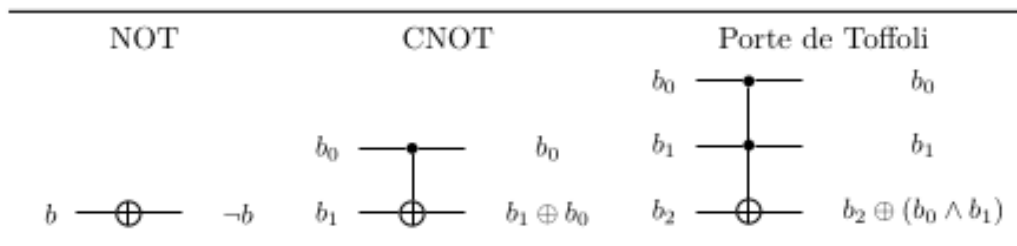
Chapitre 60

(ENS) Implémentation des circuits réversibles *** (ENS 24, corrigé — oral - 218 lignes)

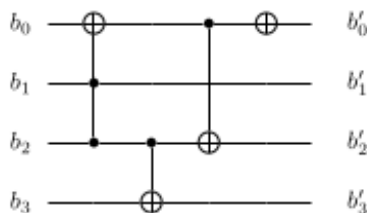
Circuits,
sources : `enscircuitsreversibles.tex`

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 ?

correction

1. Les circuits sont réversibles, c'est dans le titre. Une manière simple de résoudre cette question est donc d'exhiber l'inverse du circuit : pour un circuit \mathcal{C} constitué d'une seule porte, $f_{\mathcal{C}\mathcal{C}} = f_{\mathcal{C}} \circ f_{\mathcal{C}}$ est l'identité. Par conséquent $f_{\mathcal{C}}$ est son propre inverse.
2. Comme les portes individuelles sont involutives, on peut inverser le circuit en écrivant ses portes dans l'ordre inverse.

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.

correction

Cette question a posé problème à beaucoup de candidat(e)s.

L'idée principale est de séparer le AND des $n - 2$ contrôles en un premier AND de $n - 3$ contrôles, suivi d'une autre opération AND calculée avec une porte de Toffoli. Il faut ensuite «corriger» les valeurs contenues dans les bits de sortie afin d'implémenter exactement la porte voulue, en utilisant la distributivité du AND sur le XOR. Soient x_0, \dots, x_{n-3} les contrôles, x_{n-2} le bit résultat, x_{n-1} le dernier bit du circuit.

On effectue les opérations suivantes :

$$\begin{cases} x_{n-1} \leftarrow x_{n-1} \oplus (x_0 \wedge \dots \wedge x_{n-4}) \\ x_{n-2} \leftarrow x_{n-2} \oplus (x_{n-1} \wedge x_{n-3}) \\ x_{n-1} \leftarrow x_{n-1} \oplus (x_0 \wedge \dots \wedge x_{n-4}) \\ x_{n-2} \leftarrow x_{n-2} \oplus (x_{n-1} \wedge x_{n-3}) \end{cases} \quad (1)$$

On vérifie que cette implémentation donne le résultat escompté. En effet :

1. Après les deux premières opérations, le bit à position $n - 2$ contient $(x_{n-1} \oplus (x_0 \wedge \dots \wedge x_{n-4})) \wedge x_{n-3} \oplus x_{n-2}$. (Ensuite, développer).
2. Après la troisième opération, le bit à position $n - 1$ contient de nouveau x_{n-1} .
3. La dernière opération permet d'enlever le terme $x_{n-1} \wedge x_{n-3}$ dans le bit à position $n - 2$, ce qui donne le résultat.

Dans la suite de cet exercice, on va démontrer le théorème suivant :

Pour tout $n \geq 7$, une permutation Π 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 $n = 3$.

correction

Une seule porte suffit. Dans le cas $n = 2$ on considère la porte CNOT définie plus haut : la permutation envoie (00) vers (00), (01) vers (01), (10) vers (11) et (11) vers (10). C'est donc une transposition, qui est impaire. Dans le cas $n = 3$ on considère une porte de Toffoli. Toutes les entrées sont laissées invariantes hormis (110) et (111) qui sont interverties. La permutation est donc également une transposition.

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.

correction

1. Considérons le bit sur lequel la porte ne s'applique pas, sans perte de généralité plaçons-le à la position 0. Considérons un cycle de $f_{\mathcal{C}}$ de longueur $\ell : (t_0 t_1 \dots t_{\ell-1})$ où $t_0, \dots, t_{\ell-1}$ sont des n -uplets. Comme $f_{\mathcal{C}}$ n'agit pas sur le bit à position 0, tous les t_i ont la même valeur pour ce bit. On peut donc partitionner les cycles selon que le bit à position 0 vaut 0 (S_0) ou 1 (S_1). Pour tout t_i ci-dessus, soit t'_i la valeur obtenue en modifiant le bit 0. Alors $f_{\mathcal{C}}(t_i) = t_{i+1} \implies f_{\mathcal{C}}(t'_i) = t'_{i+1}$, car $f_{\mathcal{C}}$ laisse le bit 0 invariant. Par conséquent $(t'_0 t'_1 \dots t'_{\ell-1})$ est un cycle de même longueur. Ce qui définit la bijection.
2. En décomposant les cycles de S_0 et S_1 on obtient le même nombre de transpositions, donc $f_{\mathcal{C}}$ est paire. Une composition de permutations paires est paire, donc par induction triviale les permutations issues des circuits réversibles sont paires.

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 Π est une permutation de $\{0, 1\}^n$ laissant au moins un bit invariant, il existe un circuit réversible \mathcal{C} tel que $\Pi = f_{\mathcal{C}}$.

correction

1. L'objectif du circuit est d'« envoyer » a sur $(1, 1, \dots, 1)$ (tous les bits à 1) et b sur $(0, 1, \dots, 1)$ (tous les bits à 1 sauf le premier). Notre seule hypothèse est que $a \neq b$. Il existe différentes manières, plus ou moins efficaces, d'implémenter une telle opération. Nous ne présentons ici qu'une possibilité. Écrivons les bits de a et de $b : a_0 \dots a_{n-1}$ et $b_0 \dots b_{n-1}$. Sans perte de généralité supposons que $a_0 = 1$ et $b_0 = 0$ (ce pourrait être l'inverse, ou ce pourrait être une position différente, mais dans ce cas on modifierait simplement les indices dans ce qui suit).
 - (a) On applique des CNOTs dont le bit de contrôle est le numéro 0 et la cible les bits qui sont non nuls dans a : ainsi, si l'entrée du circuit est a , on obtient bien $(1, \dots, 1)$ à ce stade. Si l'entrée est b , on obtient b .
 - (b) On applique un NOT sur le bit numéro 0. Si l'entrée est b , le premier bit devient 1.
 - (c) On applique des CNOTs dont le bit de contrôle est le numéro 0 et la cible les bits qui sont non nuls dans b .
 - (d) On applique un NOT sur le bit numéro 0.
 Concrètement, on a utilisé le bit 0 comme contrôle intermédiaire pour transformer la sortie : en partant de a , l'étape 3 ne s'applique pas et on obtient $(1, \dots, 1)$. En partant de b , l'étape 1 ne s'applique pas et on obtient $(0, 1, \dots, 1)$.
2. On va utiliser le circuit \mathcal{C}' précédemment construit. D'abord, on envoie c et d sur $(1, 1, \dots, 1)$ et $(0, 1, \dots, 1)$ respectivement (indépendamment du premier bit), ce qui signifie qu'on envoie :

$$\left\{ \begin{array}{l} (0, c) \rightarrow (0, 1, 1, \dots, 1) \\ (1, c) \rightarrow (1, 1, 1, \dots, 1) \\ (0, d) \rightarrow (0, 0, 1, \dots, 1) \\ (1, d) \rightarrow (1, 0, 1, \dots, 1) \end{array} \right. \quad (2)$$

On applique ensuite une Toffoli contrôlée par les $n - 2$ derniers bits (c'est possible car on l'a construite à la question 2) où la cible est le bit 1. Cela intervertit les paires $(0, 1, 1, \dots, 1)$ et $(0, 0, 1, \dots, 1)$, et aussi $(1, 1, 1, \dots, 1)$ et $(1, 0, 1, \dots, 1)$. On applique ensuite l'inverse de \mathcal{C}' .

Le circuit envoie donc $(0, c)$ sur $(0, d)$ et $(1, c)$ sur $(1, d)$, et inversement. Il est important de vérifier que les autres entrées sont inchangées : c'est le cas car en arrivant au niveau de la Toffoli on aura autre chose que $(1, \dots, 1)$ pour les contrôles (il ne se passera donc rien).

3. Si une permutation laisse un bit invariant, on peut suivre le raisonnement de la **Question 4 pour séparer ses cycles en deux ensembles disjoints, et ensuite ses transpositions en paires disjointes**. On obtient des paires de transpositions telles que vues en **Question 5.2**, et donc une implémentation.

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 $x_0 = y_1$).

correction

C'est une question préparatoire pour la question suivante. Il existe une version plus compliquée de ce résultat, qui optimise le nombre de transpositions ; ce n'est pas l'objectif ici.

Il suffit de décomposer la permutation en produit de transpositions :

$$\Pi = (x_0 y_0) \dots (x_{2k-1} y_{2k-1})$$

(Qui sont en nombre pair, par hypothèse). Entre chaque paire de transpositions $(x_{2i} y_{2i}) (x_{2i+1} y_{2i+1})$ on insère $(z_i t_i)^2$ (qui est l'identité), où on choisit z_i et t_i différents de $x_i, y_i, x_{i+1}, y_{i+1}$. C'est possible car $n \geq 6$.

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 $(xy)(zt)$.

correction

Nous allons commencer par implémenter une permutation Q qui va nous aider, en envoyant x, y, z, t sur des sorties $Q(x), Q(y), Q(z), Q(t)$ telles que le bit à position $n - 1$ vaut 1. Pour ce faire, il suffit de sélectionner un sous-ensemble de $\leq n - 1$ positions telles que les restrictions de x, y, z, t à ces positions sont toujours distinctes deux à deux.

Existe-t-il un tel ensemble ? Oui car $n \geq 7$, grâce à un raisonnement par l'absurde et au principe des tiroirs (c'est ici que le choix $n \geq 7$ devient pertinent). Il n'y a que $\binom{4}{2} = 6$ paires possibles parmi (x, y, z, t) , et il y a $\binom{7}{6} = 7$ choix de 6 positions parmi les 7. Par conséquent, si chaque choix de 6 positions provoque une collision sur une paire, il existe une paire pour laquelle deux choix collisionnent. Or, ce sont deux choix différents de 6 positions parmi 7, donc leur union contient toutes les positions : tous les bits sont égaux (ce qui est impossible).

L'échange de bits dans le circuit est implémentable (également grâce à la Question 5 ; plus simplement le SWAP de deux bits s'implémente à l'aide de 3 CNOTs). Sans perte de généralité, réécrivons donc x, y, z, t sous la forme : $(b_x, x'), (b_y, y'), (b_z, z'), (b_t, t')$ où x', y', z', t' sont distincts deux à deux. On utilise la question 5 pour implémenter une permutation Q qui laisse le premier bit invariant, et fixe le dernier bit à 1 :

$$\begin{cases} (b_x, x') \rightarrow (b_x, x'', 1) \\ (b_y, y') \rightarrow (b_y, y'', 1) \\ (b_z, z') \rightarrow (b_z, z'', 1) \\ (b_t, t') \rightarrow (b_t, t'', 1) \end{cases} \quad (3)$$

On remarque que $(b_x, x''), (b_y, y''), (b_z, z''), (b_t, t'')$ sont des valeurs distinctes deux à deux (sinon, on n'aurait pas implémenté une permutation). On utilise la Question 5 une nouvelle fois pour les envoyer respectivement sur $(0, 0, 1, \dots, 1), (0, 1, 1, \dots, 1), (1, 0, 1, \dots, 1), (1, 1, 1, \dots, 1)$, en laissant le dernier bit invariant (qui de toute façon vaut 1 dans tous les cas).

La composition de toutes ces permutations nous donne P .

Pour terminer la question (et donc le théorème), on applique une porte de Toffoli à $n - 2$ contrôles, dont la sortie est le bit numéro 1 et les contrôles les $n - 2$ derniers bits. Cette porte échange $(0, 0, 1, \dots, 1), (0, 1, 1, \dots, 1)$ et $(1, 0, 1, \dots, 1), (1, 1, 1, \dots, 1)$.

En appliquant ensuite l'inverse de P , on obtient bien $(xy)(zt)$.

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 ?

correction

Corrigé proposé par le jury

Non, car on peut en déduire l'inéquation $x_2 \leq -1$ et $-x_2 \leq 0$ ce qui forme une contradiction. Informellement, il est possible de déduire « $-1 \leq 0$ » uniquement à l'aide de combinaisons linéaires des équations de départ.

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.

correction

Soit S' l'ensemble des x -résultants.

Une des deux implications est facile : si S admet une solution, alors cette solution satisfait toutes les inégalités dans S , et également les x -résultants. Par conséquent elle satisfait aussi le système $S' \cup T$.

Inversement, soit y_1, \dots, y_{n-1} une solution de $S' \cup T$, on veut montrer qu'on peut l'étendre en une solution y, y_1, \dots, y_{n-1} de S . Cela revient à choisir la valeur y telle que y, y_1, \dots, y_{n-1} satisfera toutes les inéquations de S . Par hypothèse, toutes les inéquations de T sont satisfaites. Notons les autres de la manière suivante : $L_i \leq x$ et $x \leq U_j$ (en supposant que les ensembles des L_i et U_j sont tous deux non vides). Leurs x -résultants sont toutes les inéquations de la forme $L_i \leq U_j$.

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.

correction

C'est un algorithme récursif qui détermine si le système est satisfiable, et renvoie une solution s'il l'est. Le cas terminal est celui dans lequel un x -résultant fait apparaître une inéquation de la forme $a \leq 0$ où $a > 0$, qui est trivialement fausse. Tant qu'aucune inéquation de cette forme n'apparaît, on sélectionne une nouvelle variable et on applique $FM(S, x)$. À la fin on aura éliminé toutes les variables.

La correction de l'algorithme est évidente grâce à la question précédente.

En partant de m inéquations, on considère toutes celles qui contiennent la variable x avec un coefficient négatif (de la forme $L \leq x$) et un coefficient positif (de la forme $x \leq U$). Si k est le nombre d'inéquations du premier type on a au plus $k(m-k) \leq (m/2)^2$ résultants à calculer. Le plus simple est de borner ceci par m^2 , et d'en déduire que la complexité est doublement exponentielle.

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 [log2 n] + 2 faire
    Soit S' := ∪_{x∈X} FM(S, x) (si des inéquations apparaissent en double, on les ←
        simplifie)
    S ← S ∪ S'
Si une inéquation de S est insatisfiable, renvoyer "insatisfiable"
fin pour
Renvoyer "satisfiable"
```

Nous admettrons 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.

correction

Solution : Soit k_i le nombre de variables de X qui apparaissent dans exactement i inéquations. Soit k le nombre de variables de S .

1. On a $k_1 = 0$. En effet, si x est une variable qui n'apparaît que dans une seule inéquation, alors on peut d'abord résoudre le système en enlevant cette inéquation (par minimalité de S , il devient satisfiable), puis en déduire une valeur admissible pour x , ce qui contredirait le fait que S est insatisfiable. Ce cas est donc impossible.

2. Par la propriété (P), S contient au plus $k + 1$ contraintes (en effet, s'il contenait $k + 2$ contraintes, on pourrait en déduire sa satisfiabilité). Par hypothèse, il y a deux variables par contrainte au plus. On a donc :

$$\sum_{i \geq 1} (ik_i) \leq 2k + 2.$$

De plus $\sum k_i = k$ par définition des k_i et $k_1 = 0$. On a donc :

$$\sum_{i \geq 1} (ik_i) = k_1 + 2k_2 + 3k_3 + \dots \geq 2k_2 + 3(k - k_2)$$

donc : $2k_2 + 3(k - k_2) \leq 2k + 2 \implies k_2 \geq k - 2$ ce qui donne bien $(k - k_2) \leq 2$. Autrement dit, il y a au plus deux variables qui 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.

correction

1. Construisons un graphe non dirigé $G = (X, E)$ où les sommets sont les variables et il existe une arête entre x_1 et x_2 s'ils apparaissent ensemble dans une inéquation. Par le résultat de la question précédente, tous les sommets du graphe, sauf au plus deux, sont de degré inférieur ou égal à 2. Enlevons ces deux sommets. Il nous reste un graphe dirigé de degré 2 à $k - 2$ sommets. Un tel graphe est un ensemble de chaînes disjointes (on peut l'affirmer ici sans preuve). On peut donc choisir $\lceil (k - 2)/2 \rceil$ sommets dans ce graphe tels que deux sommets ne sont pas côte à côte (chaque chaîne de longueur ℓ nous donne $\lceil \ell/2 \rceil$ sommets). On prend pour X' l'ensemble de ces $\lceil (k - 2)/2 \rceil$ variables.
2. Soit $X'' := X \setminus X'$. Quand on utilise l'élimination FM sur une des variables de X' on obtient une seule nouvelle inéquation, qui ne fait pas intervenir de variable de X' , et n'intervient donc pas dans de futures éliminations. Donc le système obtenu en éliminant toutes les variables de X' est contenu dans $(S \cup S')_{X''}$. Or S est insatisfiable, donc d'après la question 2 l'élimination des variables de X' produit un nouveau système insatisfiable. Ce système est contenu dans $(S \cup S')_{X''}$, qui par conséquent est également 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).

correction

1. Soit S insatisfiable en entrée de la boucle. On démontre cette propriété par récurrence sur k ; si $k = 0$ on est à l'entrée de l'algorithme et elle est donc vraie.
Soit $k > 0$ et considérons le système $S_{X_k} \subseteq S$ insatisfiable. Il existe un sous-système $T \subseteq S_{X_k}$ qui est insatisfiable et minimal (on peut le construire itérativement en enlevant des inéquations de S_{X_k}), et contient $\leq f^k(n)$ variables.
On calcule S' l'ensemble des résultants de S , et $S \cup S'$ contient donc $T \cup T'$ où T' est l'ensemble des résultants de T . En utilisant la question 5, on sait qu'il existe un sous-ensemble $X_{k+1} \subseteq X_k$ contenant $\leq f^{k+1}(n)$ variables tel que $(T \cup T')_{X_{k+1}}$ est insatisfiable, mais aussi $(S_{X_k} \cup S'_{X_k})_{X_{k+1}}$ et enfin $(S \cup S')_{X_{k+1}}$; en effet on n'a fait ici que (potentiellement) remettre des inéquations dans le système.
2. Les itérations de f convergent vers 2. Si le système S initial est insatisfiable, on obtient un ensemble de deux variables (x, y) tel que $S_{(x,y)}$ est insatisfiable. Pour conclure, il suffit de remarquer que les deux dernières itérations sur $S_{(x,y)}$ vont contenir la résolution FM par x suivi de y . Par conséquent on obtiendra une contradiction immédiate.
Si le système S est satisfiable, on ne trouvera pas de contradiction et l'algorithme renverra donc le bon résultat.
3. On n'a que $\lceil \log_2 n \rceil + 2$ étapes dans l'algorithme, et à chaque étape le nombre d'inéquations grandit comme :
 $T_{i+1} \leq T_i + \frac{T_i(T_i - 1)}{2} = \frac{T_i(T_i + 1)}{2} \leq T_i^2$. Par conséquent $\log_2 T_i = 2^i \log_2 m$ et le nombre d'inéquations à la dernière étape est borné par : $2^{\log_2 m 2^{\lceil \log_2 n \rceil + 2}} = \tilde{O}(m^n)$.

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 .

correction

On commence par démontrer par induction que toutes les nouvelles inéquations produites par l'algorithme sont aussi à coefficients dans $\{0, -1, 1\}$. En effet le x -résultant entre deux inéquations à coefficients dans $\{0, -1, 1\}$ est aussi à coefficients dans $\{0, -1, 1\}$.

Pour chaque paire de variables (x, y) , il n'y a donc que 3×3 choix possibles pour les coefficients de x et y (sachant que dans le cas $(0, 0)$ on tombe sur une inéquation triviale). Cela fait donc $n(n-1)/2 \times 9$ configurations possibles de la forme $b_1x_1 + b_2x_2 \leq a$.

La procédure *Simplifie* va classer les inéquations en fonction de leur configuration et sélectionne le coefficient a minimal pour chacune d'entre elles : les autres inéquations ainsi enlevées sont évidemment redondantes. Concrètement, à chaque étape de la boucle, *Simplifie* ramène le nombre d'inéquations en-dessous de $\mathcal{O}(n^2)$.

Après application de *Simplifie*, chaque variable x apparaît au plus dans $9n$ équations (c'est une borne très large) : on ne peut donc ajouter qu'au maximum $\mathcal{O}(n^2)$ x -résultants, soit $\mathcal{O}(n^3)$ nouvelles équations au total. *Simplifie* peut être implémentée en temps $\mathcal{O}(n^3)$ à l'aide d'une table de hachage. Par conséquent l'algorithme demande $\tilde{\mathcal{O}}(n^3)$ temps et $\mathcal{O}(n^2)$ mémoire.

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

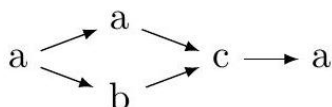
Soit Σ un ensemble fini de lettres, et Σ^* l'ensemble des mots de Σ , qui sont des séquences finies de lettres. Nous généralisons la notion de mots à des structures partiellement ordonnées.

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 Σ est un triplet (E, R, μ) 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 μ .



Comment caractériser sur le graphe l'ordre dans lequel les lettres d'un mot u généralisant p est lu ?

correction

En notant $s_0 \dots s_3$ les sommets du chemin supérieur du graphe, et s_4 le sommet du bas. On peut poser $\psi(s_0) = 0$, $\psi(s_1) = 1$, $\psi(s_4) = 2$, $\psi(s_2) = 3$, $\psi(s_3) = 4$ (il s'agit donc d'indicer les sommets de façon injective avec des entiers dans $\llbracket 0, n-1 \rrbracket$ de sorte que l'indice ne puisse que croître en suivant un arc).

On a donc $u_0 = a$, $u_1 = a$, $u_2 = b$, $u_3 = c$ et $u_4 = a$.

Le mot u est donc $aabca$.

La seule autre possibilité étant d'invertir $\psi(s_1)$ et $\psi(s_4)$, on a $\text{Total}(p) = \{aabca, abaca\}$.

u généralise p si et seulement si c'est un mot obtenu à partir du graphe en lisant les lettres dans un ordre topologique.

Question 2. Donnez une borne supérieure sur la taille de $\text{Total}(p)$ en fonction de la taille de Σ .

correction

Dans le pire des cas, le graphe est entièrement déconnecté (i.e. il n'y a aucune contrainte sur la permutation choisie) et μ est injective (toutes les lettres du graphes sont différentes).
Dans ce cas, $|\text{Total}(p)| = n!$.

Soit L un langage de Σ^* . Un mot partiel p sur Σ 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 .

correction

On se donne comme certificat une application ψ sur les sommets du mot partiel p vérifiant les propriétés ci-dessus. Cette application définit un mot u de $\text{Total}(p)$ – dont on peut vérifier l'appartenance à L en temps polynomial par hypothèse.

(Unary) 3-partition est un problème qui prend en entrée une suite finie de $3n$ entiers écrits **en unaire**³ 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 \mathbf{P} tel que $\text{Partiel}(L)$ est **NP-complet**⁴.

correction

ND M.Péchaud : l'énoncé disait «réduction à» au lieu de «réduction depuis».
TODO.

Question 5. Montrez que $\text{Partiel}(\Sigma^* ab^* a \Sigma^*)$ est vérifiable en temps polynomial.

correction

Étant donné un mot partiel p – vu comme un graphe – il s'agit de déterminer s'il existe un ordonnancement strictement croissant (au sens précédent) des sommets tels qu'un motif ab^*a y apparaisse.

Si l'on supprime les b du graphe (en re-fléchant en conséquence), c'est équivalent au fait de chercher si l'on peut obtenir un ordonnancement tel qu'un motif aa y apparaisse.

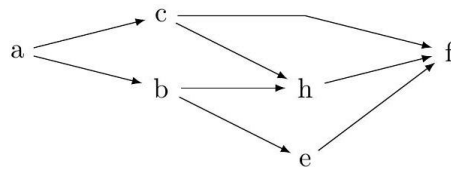
Pour ce faire, on peut considérer tous les couples de sommets (a, a) dans ce graphe, et vérifier

- s'ils sont voisins
 - ou si aucun des deux n'est descendant de l'autre
- ce qui s'effectue bien en temps polynomial.

Soit $p = (E, R, \mu)$ un mot partiel sur Σ . 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 :

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 $\text{Partiel}(L)$
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 $\text{Partiel}(L)$.



correction

On propose (a, c, h, f) et (b, e) .

Il n'est pas possible de faire mieux, car par exemple b et c ne peuvent pas appartenir à un même chemin.

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.

correction

Soit p un mot partiel, et u un mot de $\text{Total}(p)$.

Pour tout chemin c de p , les lettres de c apparaissent dans l'ordre dans u .

Notons w la taille de sa plus petite décomposition en chemin.

Si $w = 1$, le graphe est réduit à un chemin. On parcourt simplement ce chemin pour obtenir l'unique mot possible généralisant p – et l'on vérifie grâce à l'automate que ce mot est dans L .

Si $w = 2$,

Soit $p = (E, R, \mu)$ un mot partiel sur Σ . 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 ?

Chapitre 63

(ENS) Terminaison de lambda ref *** (ENS 24, partiellement corrigé, niveau surréaliste — oral - 238 lignes)

Ocaml,
sources : ensterminaison_lambda_ref.tex

Terminaison de λ_{ref}

Le λ -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' α -conversion \equiv_{α} 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* α -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 $A = (\lambda x.I)(\delta\delta)$.

correction

Corrigé de M.Péchaud. NB : ce sujet me paraît d'un niveau stratosphérique – il faut digérer 1 mois de cours de λ -calcul présenté en 2 encadrés en quelques minutes...

- Si l'on réduit l'expression extérieure, $A = (\lambda x.I)(\delta\delta) \longrightarrow I\{x/\delta\delta\} = I$ (car x est muette dans I).
On ne peut appliquer aucune réduction à I .
 - Si l'on réduit $\delta\delta$ (dans le contexte $(\lambda x.I)\square$), comme $\delta\delta \longrightarrow \delta\delta$, on obtient $A \longrightarrow A$ – on n'a pas modifié l'expression.
- Les réduits de A sont donc A lui-même, et I .

On donne un système de *types simples* au λ -calcul. Les types sont donnés par :

$S, T ::= S \rightarrow T \mid \mathbf{unit}$

Ainsi un type est soit le type fonctionnel $S \rightarrow T$ des fonctions de S dans T soit \mathbf{unit} le type de base.

Une *hypothèse* $x : T$ associe une variable de terme à un type. Un *contexte de typage* Γ est un ensemble d'hypothèses et on note $\Gamma, x : T$ le contexte $\Gamma \cup \{x : T\}$ quand x n'est pas dans Γ . Un *jugement* de typage $\Gamma \vdash M : T$ indique que le terme M a le type T dans le contexte Γ (on dit qu'un terme est *typable* s'il existe Γ, 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 λ_{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.

correction

Supposons M typable, et $M = \mathbf{E}[N]$ – obtenu en remplaçant \square dans E par N .

Montrons par induction sur E que N est typable.

Cas de base : si $E = \square$, $M = N$, donc N est typable.

Induction : • Si $E = UE'$, où E' vérifie la propriété et U est un terme, alors $M = \mathbf{E}[N] = UE'[N]$.

M étant typable, on a $\Gamma \vdash UE'[N] : T$ pour un certain type T .

Seule la règle 4 peut expliquer cela : il existe un type T' tel que $\Gamma \vdash U : T' \rightarrow T$ et $\Gamma \vdash E'[N] : T$.

Donc $E'[N]$ est typable, et par hypothèse d'induction N également.

- De même si $E = E'U$.

$A = \mathbf{E}[N]$, où $E = (\lambda x.I)\square$ et $N = \delta\delta$.

Si A était typable, N le serait aussi.

Par le même raisonnement appliqué à N , $\delta = \lambda x.(xx)$ est également typable.

Son type ne peut provenir que de la règle 3.

On devrait donc avoir pour un certain contexte Γ et un certain type T la relation suivante :

$\Gamma, x : T \vdash xx : T'$.

La règle 4 est la seule pouvant expliquer cela. On a donc

$\Gamma, x : T \vdash x : T'' \rightarrow T'$ et $\Gamma, x : T \vdash x : T''$.

Or x ne peut avoir qu'un seul type dans un contexte donné, donc c'est impossible.

On dispose d'un ensemble infini d'adresses \mathcal{A} . Une mémoire σ est une fonction qui à chaque élément d'un sous-ensemble fini de \mathcal{A} (appelé son support) associe un λ -terme.

On définit un λ -calcul appelé λ_{ref} contenant une valeur $()$ de type \mathbf{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 λ -terme, le stocke en mémoire à une nouvelle adresse α et renvoie α .
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 α et un terme M , modifie la mémoire pour remplacer la valeur en α par M , et renvoie \mathbf{unit} .

Question 3. Donner des règles de typage pour les termes de λ_{ref} et les mémoires.

correction

On crée un type T_{mem} pour une adresse contenant un terme de type T (calquant le type $\alpha\text{-ref}$ de *OCaml*).

- $\Gamma, x : T \vdash \text{ref } x : T_{\text{mem}}$
- $\Gamma, \alpha : T_{\text{mem}} \vdash \text{deref } \alpha : T$
- $\Gamma \vdash \text{assig} : T_{\text{mem}} \rightarrow T \rightarrow \text{unit}$

Question 4. Donner une sémantique pour λ_{ref} explicitant comment réduire un couple composé d'un terme typable et d'une mémoire.

correction

- $(\text{ref } M, \sigma) \longrightarrow (\alpha, \sigma')$, où α est une adresse n'apparaissant pas dans σ , et σ' prolonge σ par $\sigma'(\alpha) = M$.
- $(\text{deref } \alpha, \sigma) \longrightarrow (\sigma(\alpha), \sigma)$, où α dans le support de σ .
- $(\text{assig } \alpha M, \sigma) \longrightarrow ((), \sigma')$, où $\sigma' \equiv \sigma$ sauf pour $\sigma'(\alpha) = M$.

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 λ_{ref} un terme typable de λ_{ST} telle que si $(M, \sigma) \longrightarrow (M', \sigma)$ avec une réduction pure, alors $\mathbf{p}(M) \longrightarrow \mathbf{p}(M')$.

correction

L'idée est d'«oublier» toutes les actions sur la mémoire.

On définit donc \mathbf{p} inductivement comme la fonction qui vaut l'identité sur les termes de λ_{ST} , et telle que :

- $\mathbf{p}(\text{ref } M, \sigma) = (\lambda y. ())\mathbf{p}(M)$ (on consomme M et l'on n'en fait rien – mais M doit continuer à apparaître pour permettre une réduction pure dans M).
- $\mathbf{p}(\text{deref } \alpha, \sigma) = \mathbf{p}(\sigma(\alpha))$ (pour avoir un type correct).
- $\mathbf{p}(\text{assig } \alpha M, \sigma) = ()$ (de type **unit**).

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 $\forall n \in \mathbb{N}, M_n \longrightarrow M_{n+1}$. On admettra le résultat suivant :
Si M est un terme de λ_{ST} , alors M est terminant.

Question 6. Montrer que toute chaîne de réduction infinie de termes typables dans λ_{ref} contient une infinité de réductions impures. Proposer un terme de λ_{ref} typable et non-terminant.

correction

ND MP : l'énoncé original n'indiquait pas «de termes typables»

Par l'absurde, s'il y a un nombre fini de réductions impures, on considère le terme M de λ_{ref} consécutif à la dernière réduction impure. On note $M \longrightarrow M' \longrightarrow M'' \dots$ la suite des réductions.

Alors, par la question précédente, $\mathbf{p}(M) \longrightarrow \mathbf{p}(M') \longrightarrow \mathbf{p}(M'') \dots$ est une suite infinie de réductions λ_{ST} , ce qui est impossible.

Pour le terme demandé, il doit donc engendrer une chaîne de réductions contenant une infinité de réductions impures. Une façon de penser cette question est de chercher à construire une fonction *OCaml* typable, utilisant des références, et dont l'exécution ne termine pas. On peut proposer le code suivant :

```
1 let rec r = ref (fun () -> !r ()) in !r ()
```

à traduire en $\lambda_{\text{ref}} \dots$

On divise la mémoire en *régions* identifiées par des entiers naturels. Les opérateurs ref_n , deref_n et assig_n 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 λ_{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 $E_d = \{x \rightarrow y \mid x, y \in V \text{ et } x = i \text{ dom}(y)\}$.

$A(G)$ 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.

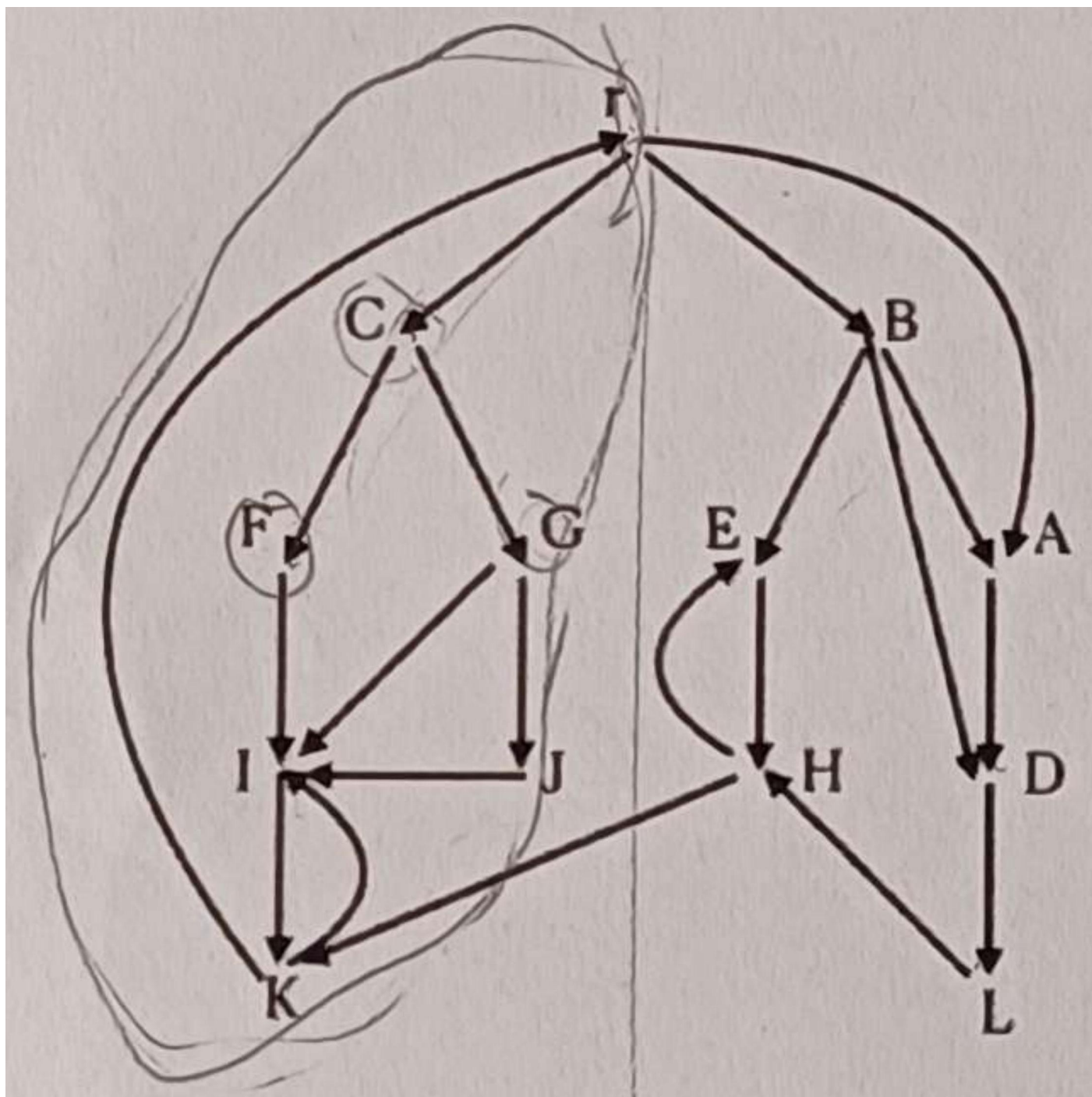


Figure 1 - Graphe de la question 1

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 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 φ . 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 — 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 Π .

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 $\vdash A \vee \neg A$.

correction

Correction de J.B.Bianquis

1.

$$\frac{\frac{\frac{}{A \wedge B \vdash A \wedge B} (\text{Ax})}{A \wedge B \vdash B} (\wedge_e^d) \quad \frac{\frac{\frac{}{A \wedge B \vdash A \wedge B} (\text{Ax})}{A \wedge B \vdash A} (\wedge_e^g)}{A \wedge B \vdash B \wedge A} \wedge_i}$$

2. Posons $P = (A \wedge B) \vee (A \wedge C)$.

$$\frac{\frac{\frac{}{A \wedge (B \vee C) \vdash A \wedge (B \vee C)} (\text{ax})}{A \wedge (B \vee C) \vdash B \vee C} \wedge_e^d \quad \frac{}{A \wedge (B \vee C), B \vdash P} (\text{lemme}) \quad \frac{}{A \wedge (B \vee C), C \vdash P} (\text{lemme})}{A \wedge (B \vee C) \vdash P = (A \wedge B) \vee (A \wedge C)} \vee_e$$

Où le lemme est le suivant (symétrique pour B et C, je donne les deux ci-dessous) :

$$\frac{\frac{\frac{\frac{}{A \wedge (B \vee C), B \vdash A \wedge (B \vee C)} (\text{ax})}{A \wedge (B \vee C), B \vdash A} \wedge_e^g \quad \frac{}{A \wedge (B \vee C), B \vdash B} (\text{ax})}{A \wedge (B \vee C), B \vdash (A \wedge B)} \wedge_i}{A \wedge (B \vee C), B \vdash (A \wedge B) \vee (A \wedge C)} \vee_i^g$$

et

$$\frac{\frac{\frac{\frac{}{A \wedge (B \vee C), C \vdash A \wedge (B \vee C)} (\text{ax})}{A \wedge (B \vee C), C \vdash A} \wedge_e^g \quad \frac{}{A \wedge (B \vee C), C \vdash C} (\text{ax})}{A \wedge (B \vee C), C \vdash (A \wedge C)} \wedge_i}{A \wedge (B \vee C), C \vdash (A \wedge B) \vee (A \wedge C)} \vee_i^d$$

3. Si l'arbre de preuve Π contient la succession de règle donnée, alors Π contient par exemple (on raisonnerait de même avec l'élimination droite) :

$$\frac{\frac{\overline{\Pi_A}}{\Gamma' \vdash A} \quad \frac{\overline{\Pi_B}}{\Gamma' \vdash B}}{\Gamma' \vdash A \wedge B} \wedge_i \quad \frac{\Gamma' \vdash A \wedge B}{\Gamma' \vdash A} \wedge_e^g$$

Dans ce cas, on peut remplacer cette portion de l'arbre par la simple preuve suivante :

$$\frac{\overline{\Pi_A}}{\Gamma' \vdash A}$$

4. On peut aussi dire qu'on n'aura pas une succession d'élimination et d'introduction de la conjonction (de bas en haut dans l'arbre), mais le raisonnement est un peu plus complexe :

$$\frac{\frac{\overline{\Pi_A}}{\Gamma' \vdash A} \quad \frac{\overline{\Pi'}}{\Gamma', A \vdash C} \quad \frac{\overline{\Pi''}}{\Gamma', B \vdash C}}{\Gamma' \vdash C} \vee_e \quad \frac{\Gamma' \vdash A \quad \frac{\overline{\Pi'}}{\Gamma', A \vdash C}}{\Gamma' \vdash A \vee B} \vee_i^g$$

Et on peut remplacer cet arbre par le suivant :

$$\frac{\frac{\overline{\Pi'}}{\Gamma', A \vdash C} \quad \frac{\overline{\Pi_A}}{\Gamma' \vdash A}}{\Gamma' \vdash C} \rightarrow_e \quad \frac{\Gamma' \vdash A \rightarrow C}{\Gamma' \vdash C} \rightarrow_i$$

Mais cela n'a que peu d'intérêt, on retrouve une succession d'élimination et d'introduction, mais pour l'implication à la place.

5. Dérivons le séquent suivant : $\neg(A \vee \neg A) \vdash \neg A$.

$$\frac{\frac{\overline{A \vdash A}}{A \vdash A \vee \neg A} (Ax) \quad \frac{\overline{\neg(A \vee \neg A) \vdash \neg(A \vee \neg A)}}{\neg(A \vee \neg A), A \vdash \perp} (Ax)}{\neg(A \vee \neg A) \vdash \neg A} (\neg_e) \quad (\neg_i)$$

En utilisant exactement la même méthode, on pourrait obtenir $\neg(A \vee \neg A) \vdash \neg \neg A$. Cependant, en remplaçant la dernière règle par un (RAA), on peut obtenir $\neg(A \vee \neg A) \vdash A$:

$$\frac{\frac{\overline{\neg A \vdash \neg A}}{\neg A \vdash A \vee \neg A} (Ax) \quad \frac{\overline{\neg(A \vee \neg A) \vdash \neg(A \vee \neg A)}}{\neg(A \vee \neg A), \neg A \vdash \perp} (Ax)}{\neg(A \vee \neg A) \vdash A} (\neg_e) \quad (\text{RAA})$$

On combine ensuite les deux résultats :

$$\frac{\frac{\overline{\neg(A \vee \neg A) \vdash \neg A}}{\neg(A \vee \neg A) \vdash \perp} \text{Lemme 1} \quad \frac{\overline{\neg(A \vee \neg A) \vdash A}}{\neg(A \vee \neg A) \vdash \perp} \text{Lemme 2}}{\vdash A \vee \neg A} (\neg_e) \quad (\text{RAA})$$

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 — 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 $p \geq n - 1$.

correction

Correction de J.B.Bianquis Toutes ces questions sont des preuves du cours de 1ère année.

Par récurrence sur l'ordre n du graphe.

- **Initialisation** : Pour $n = 1$, le graphe est réduit à un sommet. Il a donc bien au moins $1 - 1 = 0$ arêtes.
- **Hérédité** : Supposons la propriété vraie jusqu'au rang n , montrons-la vraie au rang $n + 1$. On distingue deux cas :

— S'il existe un sommet de degré 1 : nommons-le u et nommons v son unique voisin. Notons $S' = S \setminus \{u\}$ et $A' = A \setminus \{uv\}$. Le graphe G' induit par S' est connexe.

[FAIRE UN SCHÉMA.]

((Preuve : Soient x et y dans G' (donc distincts de u). Par connexité de G , il existe un chemin élémentaire de x à y dans G . Ce chemin étant élémentaire, il ne peut pas passer par u : en effet, u ayant un seul voisin v , si le chemin passait par u il serait de la forme $x, \dots, v, u, v, \dots, y$ et ne serait pas élémentaire. Mais donc ce chemin élémentaire n'emprunte jamais une arête dont u est une extrémité, c'est à dire qu'il n'utilise que des arêtes de G' . Donc il existe un chemin de x à y dans G' , donc G' est connexe.))

Remarque : Dans un oral, on peut sans doute se contenter d'affirmer que G' est connexe à partir du dessin.

Enfin, $|S'| = n < n + 1$ donc par hypothèse de récurrence $|A'| \geq n - 1$. Or $|A| = |A'| + 1$, donc $|A| \geq n = (n + 1) - 1$. D'où l'hérédité dans ce premier cas.

— Sinon il n'y a pas de sommets de degré 1. Puisque le graphe est connexe il n'y a pas de sommets de degré 0 : donc le degré minimal est 2.

Or, on sait que dans un graphe non-orienté :

$$\sum_{s \in S} \deg(s) = 2|A|$$

Donc :

$$|A| = \frac{1}{2} \sum_{s \in S} \deg(s) \geq \frac{1}{2} \sum_{s \in S} 2 \geq |S|$$

On a bien au moins $n - 1$ arêtes. D'où l'hérédité dans ce second cas.

- **Conclusion** : On a prouvé par récurrence sur n qu'un graphe connexe à n sommets a au moins $n - 1$ arêtes.

2. Montrer que si G est acyclique alors il possède un sommet de degré au plus 1.

correction

On raisonne par contraposée. Montrons que si tous les sommets de G sont de degré au moins 2, alors il existe un cycle dans G .

Notons $G = (S, A)$, on suppose donc que $\forall s \in S, \deg(s) \geq 2$. On définit une suite $(s_i)_i \in S^{\mathbb{N}}$ de sommets du graphe de la manière suivante :

- $s_0 \in S$ (existe car $n \geq 1$). s_1 est un voisin de s_0 (quelconque).
- $\forall i \geq 2$, on pose s_{i+1} un voisin de s_i différent de s_{i-1} (possible car $\deg(s_i) \geq 2$).

Comme S est fini, il existe des sommets qui se répètent dans la suite. Soit s_j le premier sommet apparaissant une deuxième fois dans la suite, et $j' < j$ son premier indice d'apparition (faire un schéma). Alors $s_{j'}, s_{j'+1}, \dots, s_j = s_{j'}$ est un cycle. Montrons en effet que ce chemin est un chemin simple. Notons que $s_{j'}, s_{j'+1}, \dots, s_{j-1}$ est un chemin élémentaire, donc simple (par minimalité de j). Donc si une arête est parcourue deux fois dans ce chemin, ça ne peut être que l'arête $s_{j-1} - s_j$. Or par minimalité de j , s_{j-1} n'apparaît qu'une seule fois, donc le chemin considéré est exactement $s_{j'} - s_{j'+1} - s_{j'+2}, \dots, s_{j-1} - s_j$, ce qui contredit la définition de $s_{j'+2}$.

3. Montrer que si G est acyclique alors $p \leq n - 1$.

correction

Par récurrence sur l'ordre $n \geq 1$ du graphe.

- **Initialisation (n=1)** : 1 sommet et 0 arêtes : ok.
- **Hérédité** : Par la question précédente, il existe $s_0 \in S$ tel que $\deg(s_0) \leq 1$. Le sous-graphe de G induit par $S \setminus \{s_0\}$ est acyclique d'ordre $n - 1$ (un cycle dans ce sous-graphe serait un cycle dans G), donc par HR il possède au plus $n - 2$ arêtes.

Comme $\deg(s_0) \leq 1$, on a supprimé au plus une arête en passant au sous-graphe induit. Donc G possède au plus $n - 1$ arêtes.

4. Montrer que les trois propriétés suivantes sont équivalentes :

- G est connexe et acyclique.
- G est connexe et $p = n - 1$.
- G est acyclique et $p = n - 1$.

correction

On montre les implications suivantes :

- **(i) \Rightarrow (ii) et (i) \Rightarrow (iii)** : Si G est connexe acyclique, alors par les questions précédentes, il vérifie à la fois $p \leq n - 1$ et $p \geq n - 1$, donc $p = n - 1$. D'où (ii) et (iii).

- **(ii) \Rightarrow (i)** : Supposons G connexe à $p = n - 1$ arêtes. Supposons par l'absurde que G possède un cycle $s_0, s_1, \dots, s_k = s_0$. Alors supprimer l'arête $s_0 s_1$ ne déconnecte pas le graphe, car on peut la remplacer par $s_0 = s_k, s_{k-1}, \dots, s_1$ dans un chemin.

Donc $G' = (S, A \setminus \{s_0 s_1\})$ est un graphe connexe d'ordre n possédant $p = n - 2 < n - 1$ arêtes : absurde d'après les questions précédentes. Donc G est acyclique.

- **(iii) \Rightarrow (i)** : Supposons G acyclique à $n - 1$ arêtes. Notons r le nombre de composantes connexes de G , et n_1, \dots, n_r leurs nombres de sommets respectifs. On a $n = \sum_{i=1}^r n_i$ car les composantes connexes partitionnent les sommets du graphe.

Chaque composante connexe est acyclique (car G est acyclique) donc chacune possède $n_i - 1$ arêtes d'après le premier point de cette preuve. Ainsi G possède $\sum_{i=1}^r (n_i - 1) = \sum_{i=1}^r n_i - r = n - r$ arêtes. Or G possède $n - 1$ arêtes donc G possède exactement 1 composante connexe, i.e. G est connexe.

Chapitre 69

(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.

correction

Correction de J.B.Bianquis

1. On propose de rajouter deux tables représentant respectivement les allergies des clients et les allergènes des plats :

```
Allergies(id_client : int, aliment : text)
Allergenes(plat : text, aliment : text)
```

On pourrait plutôt vouloir ajouter des attributs aux tables existantes, mais il faudrait une nouvelle ligne par allergie différente de chaque client, ce qui dupliquerait les adresses, noms et prénoms.

2. On propose :

```
Pseudo-code {
1 SELECT DISTINCT noms, prenom FROM Clients AS C
2     JOIN Allergie AS A1 ON C.id = A1.id_client
3     JOIN Allergene AS A2 ON A1.aliment = A2.aliment
4 WHERE plat = 'pizza'
```

3. On propose :

```
Pseudo-code {
1 SELECT ville, SUM(prix) FROM Adresse AS Ad
2     JOIN Clients As Cl ON Cl.id_adresse = Ad.id
3     JOIN Commandes AS C ON Cl.id = C.id_client
4 GROUP BY ville
5 ORDER BY SUM(prix) DESC LIMIT 3
```

4. On propose :

Pseudo-code

```
1 SELECT prenom, AVG(S) FROM
2   (SELECT id_client, prenom, SUM(prix) AS S FROM Adresse AS Ad
3     JOIN Clients As Cl ON Cl.id_adresse = Ad.id
4     JOIN Commandes AS C ON Cl.id = C.id_client
5     GROUP BY id_client)
6 GROUP BY prenom
```

Attention à ne pas faire directement la moyenne sur les prénoms, où vous aurez un montant moyen de commande par prénom, et non un montant moyen de **somme** dépensée, par prénom.

Chapitre 70

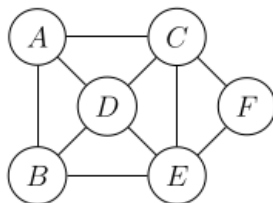
(MT) couplages * (MT 0, ex 1, corrigé — oral - 64 lignes)

*

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 .

correction

Correction de J.B. Bianquis Cet exercice est constitué d'une portion du cours sur les couplages (cf cours !!!).

1. Soit $G = (S, A)$ un graphe non orienté. Un couplage de G est un ensemble d'arêtes $C \subseteq A$ tel que tout sommet de G est incident à au plus une arête de C ($\forall e = (x, y), e' = (x', y') \in C, \{x, y, x', y'\}$ sont deux à deux disjoints).

Un couplage C est dit :

- **maximal** si il n'est pas strictement inclus dans un autre couplage ($\forall C'$ couplage, $C \subseteq C' \Rightarrow C = C'$).
 - **maximum** ou **de cardinalité maximale** si sa cardinalité (son nombre d'arêtes) est maximale parmi tous les couplages de G ($\forall C'$ couplage, $|C'| \leq |C|$).
 - **parfait** si tout sommet de G est incident à une arête de G ($\forall s \in S, \exists e \in C, \exists y \in S, e = (x, y)$ ou encore $|C| = |S|/2$).
2. $C = \{(A, B), (C, D), (E, F)\}$ est un couplage parfait, donc maximum. Le couplage $C' = \{(A, C), (B, E)\}$ est maximal (on ne peut lui ajouter aucune arête), mais il n'est pas maximum car C contient strictement plus d'arêtes.
 3. Un chemin alternant pour C dans G est un chemin élémentaire dont les arêtes sont alternativement dans C et dans $A \setminus C$. Un chemin augmentant est un chemin alternant dont les deux extrémités sont libres (i.e. ne sont incidentes à aucune arête du couplage C).
 4. Il s'agit du lemme de Berge du cours. Redémontrons-le ici.
 \Rightarrow : Par contraposée. Si C possède un chemin augmentant c , alors $C \Delta c$ (la différence symétrique) est un couplage de cardinalité $|C| + 1 > |C|$, donc $|C|$ n'est pas maximum. (cf schéma du cours de $C \Delta c$.)
 \Leftarrow : Par contraposée. Supposons que C n'est pas maximal et soit C' un couplage tel que $|C'| > |C|$. Montrons que C possède alors un chemin augmentant. On considère $D = (S, C \Delta C')$. Chaque sommet de G est incident à au plus une arête de C et une arête de C' , donc le degré maximal de D est au plus 2. Donc chaque composante connexe de D est :
— soit réduite à un sommet

- soit un chemin élémentaire dont les arêtes sont alternativement dans C et dans C'
- soit un cycle élémentaire dont les arêtes sont alternativement dans C et dans C' . Notons que cela implique que le cycle est de longueur paire et contient autant d'arêtes de C que de C' .

(proposition précédente du cours, qui se prouve en considérant un chemin élémentaire de longueur maximale dans G et en regardant s'il est de longueur 0 et s'il est "fermable".)

Chaque composante cycle (et sommet) possède autant d'arête de C que de C' , or $|C'| > |C|$ donc $D = C\Delta C'$ possède au moins une arête de plus dans C' que dans C . Donc il existe au moins une composante chemin avec une arête de plus dans C' que dans C . Cette composante fournit un chemin augmentant pour C .

Chapitre 71

(MT) logique * (MT 0, ex 1, corrigé — 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?

correction

Correction de J.B.Bianquis

F est satisfiable, par exemple une valuation $v : \begin{cases} a \mapsto V \\ b \mapsto V \end{cases}$ la satisfait (peu importe la valeur de c).

$\neg F$ est satisfiable, par exemple la valuation $v : \begin{cases} a \mapsto F \\ b \mapsto V \\ c \mapsto V \end{cases}$ évalue F à Faux, donc $\neg F$ à Vrai.

2. Mettre F en forme normale conjonctive.

correction

Il y a deux façons de faire, la table de vérité a l'avantage de donner des formes canoniques et de répondre aux deux questions suivantes rapidement une fois qu'elle est remplie, mais la résolution par distributivités me semble plus rapide au total que le remplissage d'une table, même s'il faut la faire deux fois. Donnons d'abord cette deuxième solution.

On identifie \wedge à $+$ et \vee à \times pour obtenir en distribuant une conjonction. On calcule alors aisément :

$F = ((a + b) \cdot \neg c) + (a \cdot \neg b) = a \cdot \neg c + b \cdot \neg c + a \cdot \neg b$, d'où

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

3. Mettre F en forme normale disjonctive.

correction

De même, cette fois on identifie \vee à $+$ et \wedge à \times . On distribue :

$F = ((a \cdot b) + \neg c) \cdot (a + \neg b) = a \cdot b \cdot a + a \cdot b \cdot \neg b + \neg c \cdot a + \neg c \cdot \neg b$, d'où

$$F \equiv (a \wedge b \wedge a) \vee (a \wedge b \wedge \neg b) \vee (\neg c \wedge a) \vee (\neg c \wedge \neg b) \equiv \boxed{(a \wedge b) \vee (\neg c \wedge a) \vee (\neg c \wedge \neg b)}.$$

Version avec table de vérité :

a	b	c	$a \wedge b$	$(a \wedge b) \vee \neg c$	$(a \vee \neg b)$	F
0	0	0	0	1	1	1
0	0	1	0	0	1	0
0	1	0	0	1	0	0
0	1	1	0	0	0	0
1	0	0	0	1	1	1
1	0	1	0	0	1	0
1	1	0	1	1	1	1
1	1	1	1	1	1	1

Remarque : on peut se permettre de ne pas remplir tous les colonnes et conclure directement pour F dans certains cas, pour gagner du temps.

On lit alors la table pour trouver les CNF et DNF canoniques (let "et" des non-lignes à 0, le "ou" des lignes à 1) :

$$\text{(CNF :)} \quad F \equiv (a \vee b \vee \neg c) \wedge (a \vee \neg b \vee c) \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee c)$$

$$\text{(DNF :)} \quad F \equiv (\neg a \wedge \neg b \wedge \neg c) \vee (a \wedge \neg b \wedge \neg c) \vee (a \wedge b \wedge \neg c) \vee (a \wedge b \wedge c)$$

4. Rappeler le théorème de Cook-Levin.

correction

SAT est **NP**-complet.

Chapitre 72

(MT) Arbres Binaires de Recherche * (MT 0, ex 2, corrigé — oral - 161 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).

correction

Correction de J.B.Bianquis

(Comme un air de déjà-vu... CCINP a posé la même question en début d'un de ses exercices de Type A...).

Un arbre binaire de recherche est un arbre binaire dont les clés sont à valeurs dans un ensemble totalement ordonné. De plus, pour chaque noeud de cet arbre, son étiquette x est strictement supérieure aux étiquettes dans son fils gauche et strictement inférieure aux étiquettes dans son fils droit (des variantes autorisent une inégalité large d'un des deux côtés).

2. Rappeler la complexité dans le pire des cas de la recherche dans un ABR ayant n noeuds.

correction

La complexité de la recherche dans un ABR de hauteur h est $O(h)$, dans le pire cas cette hauteur vaut n ou $\Theta(n)$ (par exemple si on a un graphe ligne ou un peigne). Donc la complexité pire cas est $O(n)$.

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 .

correction

Pour une clé k fixée, la recherche de k dans A se fait en temps $O(H_A(k))$. La clé k a une probabilité d'apparition donnée par \mathbb{P} . Donc la complexité en moyenne (espérance) sur les entrées de la recherche d'une clé dans A est :

$$O(\mathbb{E}(H_A))$$

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 $\mathbb{E}(H_A) = O(\log(m))$.

correction

L'espérance est alors la moyenne (classique) des hauteurs des noeuds de l'arbre. Les ABR pour lesquels $\mathbb{E}(H_A)$ est minimale sont les arbres les plus équilibrés possibles (ils minimisent la somme des hauteurs de leurs noeuds, car on remplit d'abord entièrement les étages de plus petite profondeur), c'est à dire les arbres complets (à remplissage du dernier étage près, qui est rempli comme on veut).

Pour montrer que $\mathbb{E}(H_A) = O(\log(m))$, il suffit de montrer que la profondeur maximale d'un noeud de l'arbre (i.e. sa hauteur) est un $O(\log(m))$, car l'espérance est plus petite que la valeur maximale avec une probabilité uniforme (c'est la moyenne).

On a montré dans le cours de 1ère année que pour un arbre complet à m noeuds, on a $h = \lfloor \log m \rfloor$. Redémontrons ce résultat. On montre d'abord que $2^h \leq m < 2^{h+1}$.

Preuve : Notons m_k le nombre de noeuds à la profondeur k dans l'arbre. Comme l'arbre est complet, on a $m_k = 2^k$ pour tout $k \in \llbracket 0, h \rrbracket$ et on a $1 \leq m_k \leq 2^k$.

Donc comme $m = \sum_{k=0}^h m_k$, on a :

$$1 + \sum_{k=0}^{h-1} 2^k \leq m \leq \sum_{k=0}^h 2^k$$

D'où $2^h \leq m \leq 2^{h+1} - 1$. On conclut :

$h \leq \log m < h + 1$, c'est à dire par définition de la partie entière $h = \lfloor \log m \rfloor = O(\log m)$.

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 ?

correction

Il s'agit d'un problème d'optimisation. *Je ne suis pas sûre de ce que la question attendait de plus spécifique ici ?*

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}

correction

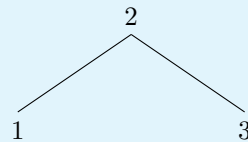


FIGURE 72.1 – Arbre pour les probabilités $\mathbb{P}(1) = 2/3$, $\mathbb{P}(2) = 1/6$, $\mathbb{P}(3) = 1/6$.

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.

correction

L'idée est toute simple : si ce n'était pas le cas, il existerait un meilleur arbre A'_g pour les noeuds à gauche, et il suffirait de remplacer A_g par A'_g dans A pour obtenir un arbre strictement meilleur que A , ce qui contredirait l'optimalité de A . Rédigeons cela rigoureusement.

On sait que A est optimal, i.e. qu'il minimise $\mathbb{E}(H_A)$. Or :

$$\begin{aligned} \mathbb{E}(H_A) &= \sum_{k \in \mathcal{K}} \mathbb{P}(k) H_A(k) \\ &= \sum_{k \in \kappa(A_g)} \mathbb{P}(k) H_A(k) + \sum_{k \in \mathcal{K} \setminus \kappa(A_g)} \mathbb{P}(k) H_A(k) \end{aligned}$$

$$\text{Or } \sum_{k \in \kappa(A_g)} \mathbb{P}(k) H_A(k) = \sum_{k \in \kappa(A_g)} S_g \mathbb{P}_g(k) (H_{A_g}(k) + 1) = S_g \sum_{k \in \kappa(A_g)} \mathbb{P}_g(k) H_{A_g}(k) + \sum_{k \in \kappa(A_g)} S_g \mathbb{P}_g(k).$$

$$\text{Résumons : } \mathbb{E}(H_A) = \boxed{S_g \sum_{k \in \kappa(A_g)} \mathbb{P}_g(k) H_{A_g}(k)} + \sum_{k \in \kappa(A_g)} S_g \mathbb{P}_g(k) + \sum_{k \in \mathcal{K} \setminus \kappa(A_g)} \mathbb{P}(k) H_A(k)$$

Supposons maintenant par l'absurde que A_g n'est pas optimal pour $\kappa(A_g)$ et \mathbb{P}_g . Alors il ne minimise pas

$\mathbb{E}(H_{A_g})$, donc il existe un arbre A'_g de même noeuds ($\kappa(A_g) = \kappa(A'_g)$) tel que $\mathbb{E}(H_{A'_g}) < \mathbb{E}(H_{A_g})$ (espérance pour \mathbb{P}_g), i.e. :

$$\sum_{k \in \kappa(A_g)} \mathbb{P}_g(k) H_{A'_g}(k) < \sum_{k \in \kappa(A_g)} \mathbb{P}_g(k) H_{A_g}(k)$$

En remplaçant A_g par A'_g dans A , on a les mêmes ensembles de clés et on obtient donc la nouvelle espérance consistant à échanger ces deux termes dans l'expression précédente (tous les autres termes restant égaux). Cette espérance est strictement plus faible, ce qui contredit l'optimalité de A , absurde. De même pour le sous-arbre droit.

8. Proposer un algorithme de programmation dynamique pour trouver un ABR optimal.

correction

On remarque que dans un ABR, une fois qu'on a choisi une racine $r \in \mathcal{K}$, cela fixe les clés des sous-arbres gauche et droit. Notons $=\{k_1, \dots, k_m\}$ les clés énumérées dans l'ordre croissant. Si k_i est la racine de l'arbre, alors le sous-arbre gauche contient les clés k_1, \dots, k_{i-1} et le sous-arbre droit contient les clés k_{i+1}, \dots, k_m . Notons alors $E_{i,j}$ la meilleure espérance qu'on puisse obtenir pour un ABR optimal sur les clés k_i, \dots, k_j pour

la probabilité $\mathbb{P}_{i,j} = \frac{1}{S_{i,j}} \mathbb{P}$, avec $S_{i,j} = \sum_{l=i}^j \mathbb{P}(k_l)$. On peut calculer les $E_{i,j}$ via les formules de récurrence :

$$E_{i,j} = \begin{cases} 0 & \text{si } i \geq j \\ \min_{r \in \llbracket i, j \rrbracket} (S_{i,r-1}(1 + E_{i,r-1}) + S_{r+1,j}(1 + E_{r+1,j})) & \text{sinon } (i < j) \end{cases}$$

On calcule alors $E_{1,m}$, le résultat recherché, par programmation dynamique, par exemple de manière récursive en mémorisant les $E_{i,j}$ calculés dans un tableau de taille $m \times m$ ou par une table de hachage, par exemple.

9. Quel est sa complexité ?

correction

On calcule $O(n^2)$ valeurs $E_{i,j}$ au plus, toutes celles pour $i < j$, et chacune d'entre elle peut se calculer en temps $O(n)$ en utilisant les autres valeurs. En effet, on peut commencer par pré-calculer toutes les sommes partielles $S_{i,r-1}$ et $S_{r+1,j}$ en temps $O(n)$ (par simples passes dans deux tableaux pour les stocker). Puis, on calcule le min sur r en $O(n)$.

On obtient donc une complexité totale en $\boxed{O(n^3)}$.

Chapitre 73

(MT) entrelacements de mots * (MT 0, ex 2, corrigé — oral - 109 lignes)

*

Langages, Automates, Programmation dynamique,
sources : mtlang.tex

Soit Σ un alphabet.

Pour deux mots u, v dans Σ^* , 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 Σ^* , proposer un algorithme de programmation dynamique permettant de savoir si $w \in \mathcal{E}(u, v)$. Préciser la complexité.

correction

Correction de J.B. Bianquis

1. $\mathcal{E}(u, v) = \{abcb, acbb, cbab, cabb\}$.

2. (a) On aura le plus grand nombre de mots possible dans le cas où les lettres sont deux à deux distinctes (il n'y a aucun doublon). Dans ce cas, toutes les combinaisons sont différentes. On compte le nombre de façon d'insérer les lettres du mot v dans le mot u .

Cela revient à choisir les $|v|$ positions des lettres de v dans le mot mélange, les autres positions étant celles de u . Une fois ce choix effectué, on doit mettre les lettres de u et de v dans l'ordre et on n'a plus de choix à faire. Il y a donc $\binom{|u| + |v|}{|v|}$ éléments dans $u \star v = \mathcal{E}(u, v)$.

$$\mathcal{E}(u, v) \leq \binom{|u| + |v|}{|v|}$$

- (b) On s'inspire du produit de deux automates. Sauf qu'au lieu d'avancer dans les deux automates en parallèle en suivant une transition (en lisant une lettre), on ne va avancer que dans un automate à la fois (deux transitions possibles, de manière non déterministe). Formalisons cette idée :

(2.3) Soient $A = (\Sigma, Q, q_I, F, \delta)$ et $A' = (\Sigma', Q', q'_I, F', \delta')$ deux automates finis déterministes (AFD) acceptant respectivement les langages L et L' . On construit l'automate (non déterministe) sur $\Sigma \cup \Sigma'$ dont l'ensemble d'états est $Q \times Q'$, les ensembles d'états initiaux et finaux sont (q_I, q'_I) et $F \times F'$ et dont l'ensemble des transitions est :

$$\{(p, p') \xrightarrow{a} (q, p') \mid \delta(p, a) = q\} \cup \{(p, p') \xrightarrow{a} (q, p') \mid \delta'(p', a) = q'\}.$$

On vérifie sans difficulté que cet automate accepte $L \star L' = (L, L')$. Ceci répond à la question pour $L = \{u\}$ et $L' = \{v\}$.

(2.2 2.) Pour une preuve plus simple, utilisant directement les mots (u, v) , on peut donner l'automate ci-dessus directement. Notons $u = u_1 \dots u_n$ et $v = v_1 \dots v_m$. On construit l'automate contenant $(n+1) \times (m+1)$ états notés $(i, j)_{i, j \in [0, n] \times [0, m]}$ où l'état (i, j) représente le fait qu'on a lu les i premières lettres du mot u et les j premières lettres du mot v .

L'état initial est alors $(0, 0)$, l'état final (unique) est (n, m) et on a les transitions suivantes pour tout i, j :

$$\begin{cases} (i, j) \xrightarrow{u_{i+1}} (i+1, j) & \text{si } i < n \\ (i, j) \xrightarrow{v_{j+1}} (i, j+1) & \text{si } j < m \end{cases}$$

3. C'est la première preuve donnée dans la réponse précédente (la plus générale des deux).

4. Posons $w = w_1 w_2 \dots w_{n+m}$ en gardant les notations précédentes. Si la longueur de w n'est pas égale à $n+m$, on peut directement conclure que $w \notin \mathcal{E}(u, v)$. On note $e_{i,j} = \begin{cases} 1 & \text{si } w_1 \dots w_i \in \mathcal{E}(u_1 \dots u_j, v_1 \dots v_{i-j}) \\ 0 & \text{sinon} \end{cases}$

On cherche à renvoyer $e_{n+m, n}$, et on peut calculer récursivement les $e_{i,j}$ à l'aide des formules suivantes (notons qu'on a toujours $i, j \geq 0$ et $j - i \geq 0$) :

$$e_{i,j} = \begin{cases} 1 & \text{si } i = j = 0 \\ \mathbb{1}_{(w_i = v_{i-j} \text{ et } e_{i-1, j} = 1)} & \text{sinon si } j = 0 \text{ (plus de lettre dans } u) \\ \mathbb{1}_{(w_i = u_j \text{ et } e_{i-1, j-1} = 1)} & \text{sinon si } j = i \text{ (plus de lettre dans } v) \\ \mathbb{1}_{(w_i = u_j \text{ et } e_{i-1, j-1} = 1) \text{ ou } (w_i = v_{i-j} \text{ et } e_{i-1, j} = 1)} & \text{sinon.} \end{cases}$$

On calcule $e_{n+m, n}$ par programmation dynamique, par exemple en version récursive mémoisée dans un tableau de taille $(n+1) \times (n+m+1)$ en temps $O(n \cdot (n+m))$ car chaque case se calcule en temps constant (en fonction d'au plus deux autres cases, et moins quand les lettres sont distinctes). De manière itérative, on peut remplir le tableau de haut en bas et de gauche à droite (pour (i, j) croissant lexicographiquement), puisque la case (i, j) dépend au plus de la case d'au dessus et de la case de gauche.

Chapitre 74

(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 considérera 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.

correction

Correction de J.B. Bianquis

1. Soient $x, y, z \in S$ tels que $S\mathcal{R}y$ et $y\ddagger z$. Alors par définition (comme $(x, y) \in A^*$), il existe un chemin de x à y dans G . Notons le $x = s_0, \dots, s_k = y$. De même, il existe un chemin de y à z dans G . Notons le $y = s'_0, \dots, s'_l = z$.
Donc il existe un chemin de x à z dans G , obtenu en concaténant ces deux chemins : $x = s_0, \dots, s_k, s'_1, \dots, s'_l = z$ (si $s'_0 = s'_l = z$, on s'arrête à $s_k = y = s'_0 = z$).
2. On peut simplement adapter l'algorithme de Floyd-Warshall. Au lieu de chercher des distances minimales, on cherche simplement à vérifier l'accessibilité. La matrice k représente par un coef 1 les paires de sommets accessibles par un chemin utilisant les sommets intermédiaires 1 à k (et 0 sinon). Donc on calcule les matrices successives (M_k) définies par :

$$\begin{cases} M_0 = A \text{ matrice d'adjacence} \\ M_{k+1} \text{ est définie par ses coefficients } m_{i,j}^{k+1} = \begin{cases} 1 \text{ si } m_{i,j} = 1 \text{ ou } m_{i,k} = m_{k,j} = 1 \\ 0 \text{ sinon} \end{cases} \end{cases}$$

Chaque matrice M_i se calcule en temps $O(n^2)$ (temps constant pour chaque coefficient), et on cherche à renvoyer la matrice $M_n = A^*$ (sous forme de matrice d'adjacence). Le temps total est donc en $O(n^3)$ comme demandé.

Remarque : on peut aussi lancer un parcours depuis chaque sommet, chaque parcours est en $O(n^2)$ par matrice d'adjacence et on en fait n , on met à jour les voisins de s au fur et à mesure : sommet visités lors du parcours depuis s . Mais le Floyd-Warshall me semble plus simple à coder.

3. Il suffit de donner un exemple. Prenons par exemple le graphe constitué de 3 sommets x_1, x_2, x_3 et possédant toutes les arêtes possibles (dans les deux sens possibles). On peut prendre comme réduction transitive les cycles $x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_1$ ou $x_1 \rightarrow x_3 \rightarrow x_2 \rightarrow x_1$ qui contiennent chacune trois arêtes différentes de A . Ce nombre d'arêtes est bien minimal car le graphe doit rester fortement connexe, sinon certains sommets ne seront plus accessibles et on n'aura plus G^* le graphe contenant toutes les arêtes.

4. Raisonnons par double inclusion :

\subseteq : Soit $e = (x, y) \in A'$, alors il existe $z, t \in S$ tels que $e \in A_{z,t}$. Donc $e \in A$ et e appartient à un chemin de longueur maximale entre z et t , notons c un tel chemin. Montrons alors que la longueur du plus long chemin entre x et y est 1. En effet, si ce n'était pas le cas, comme on sait qu'il existe un chemin de longueur 1 (donné par e), cela signifie que la longueur d'un plus long chemin entre x et y est > 1 (notons c' un tel chemin). Mais alors en remplaçant e par ce chemin c' dans c , on obtient un chemin strictement plus long entre z et t , ce qui contredit la maximalité du chemin c .

\supseteq : Réciproquement, soit $e = (x, y) \in A$ tel que le chemin de longueur maximal entre x et y est e (de longueur 1). Alors on a $e \in A_{x,y}$ donc $e \in A'$.

5. Soit $G_R = (S, A_R)$ une réduction transitive (quelconque) de G . Montrons que $G_R = G'$, i.e. que $A_R = A'$.

Par double inclusion à nouveau :

$A_R \subseteq A'$: Soit $(x, y) \in A$ une arête de G_R (sous-graphe de G). Alors il existe un chemin de longueur 1 entre x et y , donné par (x, y) . S'il existait un chemin de longueur > 1 entre x et y , alors on pourrait enlever l'arête (x, y) dans A_R et obtenir une nouvelle réduction transitive (on ne casse aucune accessibilité, on peut remplacer cette arête par l'autre chemin), ce qui contredit la minimalité de A_R . En effet, cet autre chemin ne contient pas l'arête (x, y) , sinon on aurait un cycle contenant x ou y . Donc par la question précédente, $(x, y) \in A'$.

$A' \subseteq A_R$: Soit $(x, y) \in A'$. Alors le seul chemin entre x et y est celui passant par (x, y) , sinon on obtient un chemin de longueur au moins 2 entre x et y . Or il existe un chemin entre x et y dans G et $G^* = G_R^*$ donc il existe un chemin entre x et y dans G_R . Donc G_R contient l'arête $(x, y) \in A_R$.

6. On donne un algorithme pour calculer G' .

On peut par exemple de manière naïve, pour chaque arête (x, y) on l'enlève et on teste l'accessibilité de y depuis x , ceci se fait en temps $O(n^4)$ (on parcourt les arêtes en temps $O(n^2)$ et chacune d'entre elle nécessite un parcours en temps $O(n^2)$ via matrice d'adjacence).

On peut proposer un meilleur algorithme en $O(n^3)$ grâce aux résultats précédents de l'exercice. On calcule A^* la matrice dont le coefficient (i, j) vaut 1 si j est accessible depuis i , en temps $O(n^3)$. Ensuite, la multiplication de matrice $M = A.A^*$, calculée en temps $O(n^3)$, vérifie la propriété suivante :

$$\begin{cases} m_{i,j} > 0 \text{ s'il existe un chemin entre } i \text{ et } j \text{ de longueur au moins } 2 \\ 0 \text{ sinon (il n'existe qu'une arête ou aucun chemin entre } i \text{ et } j) \end{cases}$$

En effet, $m_{i,j} > 0$ si et seulement si il existe k tel que $A_{i,k} = 1$ et $A_{k,j}^* = 1$, i.e. il existe une arête de i à k et un chemin de k à j dans G , c'est à dire qu'il existe un chemin passant par un sommet tiers k , donc de longueur au moins 2 entre i et j .

On calcule alors A' en vérifiant, pour chaque paire (i, j) de sommets, s'il existe une arête entre i et j ($A_{i,j} = 1$) et il n'existe pas de chemin entre i et j de longueur au moins 2 ($M_{i,j} = 0$).

Chapitre 75

(MT) montée et descente d'un bus * (MT 0, ex 2, corrigé — 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.

correction

Correction de J.B.Bianquis

1. Je ne suis pas sûr de comprendre ce que les passagers sont censés faire sans synchronisation. J'imagine que c'est le bus qui appellera les fonctions des passagers ? En tout cas, si on ne fait aucun effort de synchronisation, on propose le code suivant :

```
Pseudo-code
1 PASSAGER() :
2     Tantque true Faire :
3         load()
4         run()
5         unload()
6
7 BUS() :
8     Tantque true Faire :
9         board()
10        unboard()
```

2. On propose une solution avec 4 sémaphores et un compteur. Il n'est pas nécessaire de protéger le compteur par un mutex, car un seul passager peut acquérir un sémaphore loading ou unloading à la fois.

```

Pseudo-code
1 loading = semaphore(0)
2 loaded = semaphore(0)
3 unloading = semaphore(0)
4 unloaded = semaphore(0)
5 cnt = 0

```

On obtient alors pour le bus :

```

Pseudo-code
1 LOAD() :
2     signal/loading)
3     wait/loaded)
4
5 UNLOAD() :
6     signal/unloading)
7     wait/unloaded)

```

Et pour un passager :

```

Pseudo-code
1 BOARD() :
2     wait/loading)
3     cnt = cnt + 1
4     Si cnt == C Alors :
5         signal/loaded)
6     Sinon :
7         signal/loading)
8
9 UNBOARD() :
10    wait/unloading)
11    cnt = cnt - 1
12    Si cnt == 0 Alors :
13        signal/unloaded)
14    Sinon :
15        signal/unloading)

```

3. On ne peut pas directement utiliser cette solution pour plusieurs bus, car on n'aurait aucune maîtrise du bus dans lequel un passager monte, plusieurs bus pourraient charger et décharger en même temps.
4. **Remarque :** Sans la contrainte $m \times C \leq n$, il peut y avoir interblocage. En effet, il y a moins de passagers que de places dans les bus. Ainsi, une fois tous les passagers chargés, le dernier bus bloque l'arrêt de bus parce qu'il n'est pas plein, et aucun autre ne peut décharger tant qu'il n'a pas terminé.

L'idée est que chaque bus doit attendre que le précédent ait terminé une action avant de pouvoir la faire à son tour. On utilise des tableaux de sémaphores au lieu des sémaphores précédents (sauf loading, pour qu'un passager puisse être au courant qu'un bus charge, quel que soit son numéro).

Par ailleurs, chaque passager dispose d'un numéro de bus, pour savoir dans quel bus il est monté. En effet, un passager peut monter dans n'importe quel bus, mais il ne peut descendre que du bus dans lequel il est monté. Enfin, on garde en mémoire un numéro global de bus permettant de savoir quel est le bus courant autorisé à l'arrêt de bus, ainsi qu'un tableau de sémaphores permettant de gérer l'accès des bus à l'arrêt.

```

Pseudo-code
1 loading = semaphore(0)
2 loaded[m] = [semaphore(0), ...]
3 unloading[m] = [semaphore(0), ...]
4 unloaded[m] = [semaphore(0), ...]
5 stop[m] = [semaphore(0), ...]
6 signal(stop[0])
7 cnt = 0
8 current_bus = 0
9
10 BUS(i):
11     wait(stop[i])
12     Tantque true Faire :
13         load(i)
14         current_bus = (i + 1) modulo m
15         signal(stop[current_bus])
16         run()
17         wait(stop[i])
18         unload(i)
19
20 LOAD(i):
21     signal/loading)
22     wait/loaded[i])
23
24 UNLOAD(i):
25     signal/unloading[i])
26     wait/unloaded[i])

```

Pour un passager :

Pseudo-code

```
1 PASSENGER() :
2   Tantque true Faire :
3     b = board()
4     unboard(b)
5
6 BOARD() :
7   wait(loading)
8   b = current_bus
9   cnt = cnt + 1
10  Si cnt modulo C == 0 Alors :
11    signal(loaded[b])
12  Sinon :
13    signal(loading)
14  Renvoyer(b)
15
16 UNBOARD(b) :
17   wait(unloading[b])
18   cnt = cnt - 1
19  Si cnt modulo C == 0 Alors :
20    signal(unloaded[b])
21  Sinon :
22    signal(unloading[b])
```

L'utilisation du modulo facilite l'écriture, car les chargements et déchargements se font par vagues de C passagers de toute façon.

Chapitre 76

(MT) subset-sum * (MT 0, ex 2, corrigé — 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.

correction

Correction de J.B.Bianquis

Pour un algorithme complètement naïf, on propose :

```
Pseudo-code
1 SUBSET-SUM(T,c) :
2   maxi ← 0 //ensemble vide
3   Pour chaque sous-ensemble I ⊆ {1,...,n} Faire :
4     Si ∑_{i∈I} t_i < maxi alors :
5       maxi ← ∑_{i∈I} t_i
6   Renvoyer maxi
```

où on peut implémenter le parcours sur les sous-ensemble par un compteur binaire, par exemple.

On effectue 2^n itérations de la boucle Pour, et pour chaque itération on calcule une somme de n élément en temps $O(n)$. La mise à jour de l'ensemble I peut se faire en temps constant amorti par un compteur binaire (et ne dépassera pas $O(n)$). On obtient donc une complexité temporelle totale en $O(n \cdot 2^n)$.

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.

correction

On suppose qu'on a bien $i, j \geq 0$. On a :

$$s_{i,j} = \begin{cases} 0 & \text{si } i = 0 \text{ (et } j \geq 0) \\ \max(s_{i-1,j-t_i} + t_i, s_{i-1,j}) & \text{si } t_i \leq j \\ s_{i-1,j} & \text{sinon} \end{cases}$$

On cherche alors à calculer $s_{n,c}$.

Si on se contente d'un algorithme récursif naïf, on a toujours une complexité exponentielle, en $O(2^n)$, même si on gagne un facteur n . Cependant, on peut mémoriser les résultats (programmation dynamique) pour améliorer ce coût.

Avec l'approche ascendante, par exemple, on remplit un tableau de dimensions $n \times c$, et chaque case se calcule en temps constant $O(1)$, donc on obtient une complexité $O(nc)$.

Remarque : SUBSET-SUM est NP-complet, et cette complexité ne vient pas contredire ce résultat. En effet, la taille de l'entier c (qui fait partie des entrées) est $\log(c)$, la complexité ci-dessus est donc toujours une complexité exponentielle.

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.

correction

Trier les éléments se fait en temps $O(n \log n)$ par un tri par comparaison (par exemple, tri par tas). Le reste est en $O(n)$, on a n itérations qui se font chacune en temps constant. La complexité totale de cet algorithme est donc $O(n \log n)$.

4. Montrer que l'algorithme glouton donne une $\frac{1}{2}$ -approximation de la solution optimale.

correction

On suppose que tous les éléments de T sont plus petits que c , sinon on les retire (ils ne participent ni à la somme du glouton, ni à l'optimal). On suppose de plus qu'il reste au moins deux éléments dans T sinon l'optimal et glouton coïncident, soit sur l'ensemble vide, soit sur l'unique élément restant.

Soit $T_{opt} \subset T$ réalisant la somme optimale $S_{opt} = \sum_{t \in T_{opt}} t \leq c$. Montrons que l'algorithme glouton renvoie une

somme $S \geq \frac{1}{2} S_{opt}$.

- Si $\sum_{t \in T} t \leq c$ alors l'algorithme glouton ajoute successivement tous les éléments (aucun ne fait dépasser), et on renvoie l'optimal, à savoir $\sum_{t \in T} t$.

- Sinon, montrons que la somme S renvoyée par le glouton vérifie $S \geq \frac{c}{2}$, et donc $S \geq \frac{S_{opt}}{2}$ (car $S_{opt} \leq c$). Notons i_0 l'indice du premier élément qui nous fait dépasser c , en les sommant dans l'ordre décroissant. Un tel élément existe car $\sum_{t \in T} t > c$. Plus formellement, $i_0 = \min\{i \mid \sum_{k=0}^i t_k > c\}$.

En particulier, le glouton prend tous les t_i dans sa somme S jusqu'à atteindre i_0 et il en prend au moins 1 ($i_0 > 1$ car $t_1 \leq c$, hypothèse de départ). L'idée est alors que le t_{i_0} qui nous fait dépasser c est moins grand que ce qu'on a déjà pris, et donc on a déjà pris au moins $\frac{c}{2}$.

Formalisons ça :

$$S \geq \sum_{i=1}^{i_0-1} t_i$$

$$\text{d'où } 2S \geq \sum_{i=1}^{i_0-1} t_i + t_1 \geq \sum_{i=1}^{i_0-1} t_i + t_{i_0} \quad \text{par décroissance des } t_i = \sum_{i=0}^{i_0} t_i > c.$$

$$\text{Donc } 2S > c \text{ d'où } S > \frac{c}{2}.$$

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.

correction

Soit $c > 2$. On prend $T = \{\frac{c}{2} + 1, \frac{c}{2}, \frac{c}{2}\}$. L'optimal est vérifié en prenant les deux derniers éléments et $S^* = c$.

L'algorithme glouton prend uniquement le premier, et on a $S = \frac{c}{2} + 1$.

Donc

$$\frac{S}{S^*} = \frac{\frac{c}{2} + 1}{c} = \frac{1}{2} + \frac{1}{c} \xrightarrow{c \rightarrow +\infty} \frac{1}{2}$$

Donc, à partir d'une certaine valeur de c , on a bien $\frac{S}{S^*} \leq \alpha$, i.e. $S \leq \alpha S^*$, car $\alpha > \frac{1}{2}$ (on pourrait même calculer cette valeur, mais c'est inutile).

Chapitre 77

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

*

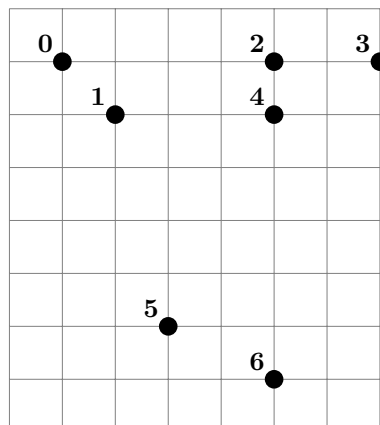
Ia, Classification hiérarchique ascendante,
sources : `mtia.tex`

1. Rappeler l'intérêt et le fonctionnement de l'algorithme de classification hiérarchique ascendante.

correction

Corrigé de M.Péchaud
Cf cours.

2. L'effectuer sur un ensemble de 7 points répartis sur une grille 7×8 (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.



correction

`[[[0, 1], [2, 4], 3], [5, 6]]`

Chapitre 78

(MT) Graphes bipartis * (MT 24, ex 1, corrigé — 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.

correction

Corrigé de M.Péchaud

On note n_1 et n_2 les cardinaux respectifs des deux ensembles U et V de sommets partitionnant les sommets du graphe.

Le nombre minimal d'arêtes est 0, et le nombre maximal $n_1 n_2$ (que l'on peut également maximiser sous la contrainte $n_1 + n_2 = n$ - auquel cas on obtient $\lceil n/2 \rceil \lfloor n/2 \rfloor$).

2. Montrer qu'un graphe est biparti si et seulement si il ne contient aucun cycle de longueur impaire.

correction

Par double implication.

\Rightarrow par l'absurde. Supposons avoir un graphe qui contient un cycle de longueur impaire, dont on note les sommets $s_1 \cdots s_n - s_n$ étant relié à s_1 - et biparti avec une partition (U, V) des sommets.

Supposons arbitrairement que $s_1 \in U$. Alors une récurrence facile montre que pour tout i pair, $s_i \in V$ et pour tout i impair, $s_i \in U$: contradiction, car n est impair, d'où $s_n \in U$ et $s_1 \in U$ alors que (s_1, s_n) est une arête.

\Leftarrow Soit H une composante connexe de G . Soit x un sommet de H . Pour tout y sommet de H , tous les chemins de x à y ont la même parité de longueur.

On note U l'ensemble des sommets qui sont atteints par un chemin de longueur paire depuis x , et l'on définit de même V pour des longueurs impaires.

On montre alors par l'absurde qu'il n'y a aucune arête entre un sommet de U et un sommet de V . Sinon, soit $(u, v) \in A$. Alors on construit un cycle de longueur impaire en concaténant :

- un chemin de x à u
- (u, v)
- un chemin de v à x

Le résultat étant vrai sur chaque composante connexe de G , il est vrai pour G .

3. Donner un algorithme qui permet de déterminer si un graphe est biparti. Donner sa complexité.

correction

On effectue un parcours (quelconque) du graphe en coloriant le premier sommet en blanc, et ses voisins en noir, et en poursuivant l'alternance.

On vérifie que chaque voisin déjà colorié du sommet courant a bien la couleur opposée.

On obtient donc un algorithme en $O(|S| + |A|)$.

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.

correction

On définit une variable propositionnelle v_s pour chaque sommet s du graphe.

La formule suivante – exprimant que deux sommets voisins doivent avoir une valeur de vérité différente – convient.

$$\bigwedge_{\{s,s'\} \in A} v_s \leftrightarrow \neg v_{s'}.$$

Pas certain de savoir quelle est la réponse attendue pour l'algorithme – on peut penser à un algorithme naïf testant toutes les valuations en $|A|2^{|S|}$.

Ou bien, étant donné que l'on obtient une instance de 2-SAT, on peut la résoudre à l'aide de l'algorithme de Kosajaru (ce qui paraît limite-limite côté programme...).

Chapitre 80

(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.

correction

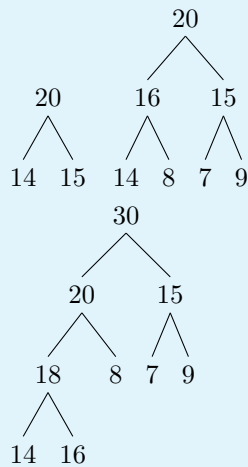
Corrigé de M.Péchaud

C'est un arbre binaire complet à gauche tel que tout sommet ait une étiquette supérieure à celle de ses éventuels fils.

On peut l'implémenter comme un arbre, ou à l'aide d'un tableau.

2. Construire un tas binaire max avec ces valeurs :
20, 14, 15, 16, 8, 7, 9, 30, 18

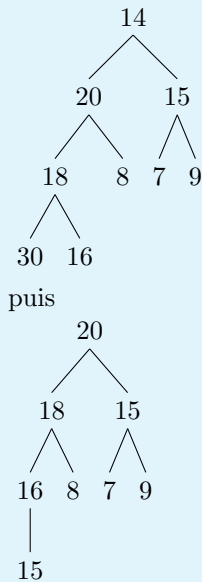
correction



3. Comment extraire la racine de ce tas ?

correction

On la permute avec la dernière feuille, on la supprime, puis l'on «descend» (ou «tasse») la nouvelle racine pour la mettre à sa place. On obtient



4. Donner la complexité temporelle des algorithmes pour ajouter un élément et extraire la racine d'un tas.

correction

Avec une implémentation par tableau, la complexité temporelle au pire $O(\log(n))$ – où n est le nombre d'éléments du 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.

correction

Un ABD est

- soit vide,
- soit un nœud ayant un ou deux fils tel que l'étiquette du nœud soit supérieure à celle de ses fils.

- (b) Écrire en pseudo-code un algorithme permettant de déterminer si un arbre binaire est décroissant ou non.

correction

```

1  est_ABD(a) :
2    si a est vide :
3      renvoyer vrai
4    si a a un fils f :
5      renvoyer et(a) >= et(f) && est_ABD(f)
6    si a a deux fils f1 et f2 :
7      renvoyer et(a) >= et(f1) && et(a) >= et(f2) && est_ABD(f1) && ←
          est_ABD(f2)
  
```

- (c) Quel lien y a-t-il entre ABD et tas binaire max ?

correction

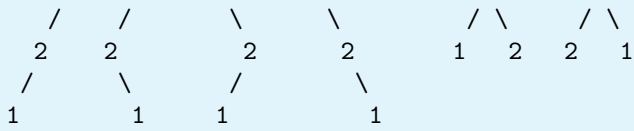
Un tas max est un ABD, mais la réciproque est fautive, car un ABD n'est pas nécessairement complet à gauche.

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 , $a_n = n!$.

correction

On énumère les possibilités dans le cas $n = 3$.

3 3 3 3 3 3



Il y a bien 6 possibilités.

On remarque que dans un ABD à n sommets étiquetés de 1 à n , n est nécessairement à la racine.

Alors, en considérant la répartition des autres étiquettes dans les sous-arbres gauches et droits, on obtient

$a_n = \sum_{p=0}^{n-1} \binom{n-1}{p} a_p a_{n-1-p}$ (une fois choisies les p étiquettes du sous-arbre gauche, on les mets en bijection avec $\llbracket 1, p \rrbracket$, et on a donc a_p façons de former un ABD avec elles).

Une récurrence forte montrer alors le résultat voulu : dans l'hérédité, on a $a_n = \sum_{p=0}^{n-1} \binom{n-1}{p} p!(n-1-p)! =$

$$\sum_{p=0}^{n-1} (n-1)! = n(n-1)! = n!.$$

Chapitre 81

(MT) déduction naturelle * (MT 24, ex 1, corrigé — oral - 141 lignes)

*

Déduction naturelle,
sources : mtlog3.tex

1. Prouver le séquent $\neg(A \vee \neg A) \vdash \neg A$.

correction

$$\frac{\frac{\text{Corrigé de M.Péchaud}}{\neg(A \vee \neg A), A \vdash A} \text{HYP}}{\neg(A \vee \neg A), A \vdash A \vee \neg A} \vee_i \quad \frac{\neg(A \vee \neg A), A \vdash \neg(A \vee \neg A)}{\neg(A \vee \neg A), A \vdash \perp} \text{HYP}}{\neg(A \vee \neg A) \vdash \neg A} \neg_e$$

2. Prouver le séquent $\vdash A \vee \neg A$.

correction

$$\frac{\frac{\frac{\dots}{\neg A \vee \neg A, \neg A \vdash \perp} \neg_e}{\neg A \vee \neg A \vdash A} \text{RAA}}{\vdash A \vee \neg A} \vee_i \quad \frac{\frac{\dots}{\neg A \vee \neg A \vdash \neg A} \neg_e}{\vdash A \vee \neg A} \text{RAA}$$

Annexe : règles de déduction utilisables.

$$\frac{}{\Gamma, \varphi \vdash \varphi} \text{HYP}$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi} \vee_i^g \quad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi} \vee_i^d \quad \frac{\Gamma \vdash \varphi \vee \psi \quad \Gamma \vdash \varphi \rightarrow \chi \quad \Gamma \vdash \psi \rightarrow \chi}{\Gamma \vdash \chi} \vee_e$$

$$\frac{\Gamma, \varphi \vdash \perp}{\Gamma \vdash \neg \varphi} \neg_i \quad \frac{\Gamma \vdash \varphi \quad \Gamma \vdash \neg \varphi}{\Gamma \vdash \perp} \neg_e$$

$$\frac{\Gamma \vdash \perp}{\Gamma \vdash \varphi} \perp_e$$

$$\boxed{\frac{\Gamma, \neg\varphi \vdash \perp}{\Gamma \vdash \varphi} \text{ RAA}}$$

Chapitre 82

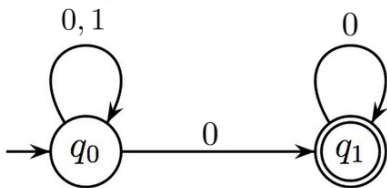
(MT) Automates de Büchi *** (MT 24, ex 2, (modifié) corrigé — oral - 117 lignes)

Langages, Automates,
sources : mtbuchi.tex

Un automate de Büchi est un automate (Q, Σ, 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^ω l'ensemble des mots infinis de X : x est dans X^ω 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 Σ^ω 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.



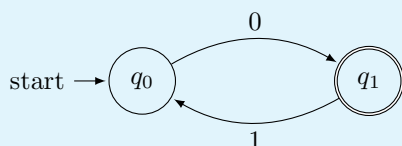
correction

Corrigé de M.Péchaud

C'est l'ensemble des mots terminant par une suite infinie de 0, i.e. $\{0, 1\}^* \{0\}^\omega$.

2. Proposer un automate de Büchi reconnaissant $(01)^\omega$.

correction



Soient A et B deux automates de Büchi.

3. Montrer que $L(A) \cup L(B)$ est reconnu par un automate de Büchi.

correction

On utilise la même construction que pour les automates finis, en considérant $(Q_1 \cup Q_2, \Sigma, T_1 \cup T_2, I_1 \cup I_2, F_1 \cup F_2)$ (en supposant $Q_1 \cap Q_2 = \emptyset$).

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.

correction

On commence par supposer que $\epsilon \notin K$.

On se donne un automate avec un unique état acceptant sans transition sortante reconnaissant K (ce qui est possible en partant d'un automate fini, créant un nouvel état acceptant q_f , et dupliquant toutes les transitions qui arrivaient dans un état acceptant vers ce nouvel état).

De même, on se donne un automate de Büchi reconnaissant $L(A)$ avec un unique état entrant q_i sans transition entrante.

On «concatène» les automates en fusionnant q_i et q_f , et on obtient un automate reconnaissant $K.L(A)$.

Si $\epsilon \in K$, on remarque que $KL(A) = L(A) \cup (K - \{\epsilon\})L(A)$, et on applique ce qui précède ainsi que la question précédente pour conclure.

5. Si K est un langage régulier autre que \emptyset ou $\{\epsilon\}$, montrer que K^ω est reconnu par un automate de Büchi.

correction

Sans perte de généralité, on peut supposer que $\epsilon \notin K$, car $K^\omega = \{K - \epsilon\}^\omega$.

On utilise alors la construction de la question précédente, en fusionnant l'état initial et l'état acceptant de l'automate reconnaissant K .

6. Montrer que $L(A) \cap L(B)$ est reconnu par un automate de Büchi.

correction

On construit $(Q_1 \times Q_2 \times \{0, 1, 2\}, \Sigma, T, I_1 \times I_2, F)$ où

- T duplique les transitions de T_1 et T_2 sur les deux premières composantes et
 - la troisième composante passe de 0 à 1 si la transition démarre d'un état ayant sa première composante dans F_1 (1 est à comprendre comme «on est passé par un état acceptant du premier automate»)
 - la troisième composante passe de 1 à 2 si la transition démarre d'un état ayant sa seconde composante dans F_2 («on est passé par un état acceptant du premier automate puis par un état terminal du second»)
 - la troisième composante passe de 0 à 2 si la transition démarre d'un état ayant ses deux premières composantes respectivement dans F_1 et F_2 (passage simultané par des états acceptants dans les deux automates)
 - les troisièmes composantes sont maintenues à 0 et à 1 dans les autres cas
 - et passent immédiatement de 2 à 0
- $F = Q_1 \times Q_2 \times \{2\}$

Alors un calcul passe une infinité de fois par un état de F (i.e. de troisième composante 2) si et seulement si les calculs projetés sur les automates de départ passent chacun une infinité de fois par des états de F_1 (resp. F_2).

Chapitre 83

(MT) Automates, langages à saut * (MT 24, ex 2, corrigé, complété au jugé pour les dernières questions — oral - 91 lignes)

*

Langages, Automates, Lemme de l'étoile,
sources : mtlang2.tex

Soit A un automate à un seul état initial q_{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 Σ^* , $a \in \Sigma \cup \{\varepsilon\}$ et $q_1, q_2 \in Q^2$. On pose la relation $(u, q_1, av) \rightsquigarrow (u', q_2, v')$ valable uniquement si $(q_1, a, q_2) \in R \wedge uv = u'v'$.

On a $(u, q_1, av) \rightsquigarrow^* (u', q_2, v')$ si un nombre fini de \rightsquigarrow permettent de passer de (u, q_1, av) à (u', q_2, v')

On définit le langage à saut de l'automate comme suit :

$L_{\dagger}(A) = \{uv \in \Sigma^* \mid \exists f \in F, (u, q_{\text{init}}, v) \rightsquigarrow^* (\varepsilon, f, \varepsilon)\}$

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.

correction

Correction de M.Péchaud

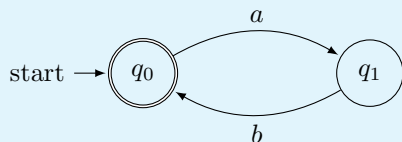
Cf cours, c'est extrêmement classique – on applique le lemme de l'étoile et on considère $a^N b^N$.

2. Montrer que L_{ab} est le langage à saut d'un certain automate.

correction

Concrètement, \rightsquigarrow permet d'«effacer» une occurrence de la lettre étiquetant la transition de q_1 à q_2 suivie du mot issu de la concaténation de la 1ère et 3ème composante du triplet.

On considère l'automate suivant :



Un calcul dans cet automate est réussi si et seulement si il emprunte autant de transitions étiquetées que par a que par b si et seulement si autant de a que de b sont effacés dans le mot de départ pour arriver à ε si et seulement si le mot de départ est dans L_{ab} .

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 ?

correction

Non : on considère par exemple $L = (ab)^*$ qui est bien régulier.

$\text{Perm}(L) = L_{ab}$ dont on n'a vu qu'il n'est pas régulier.

4. Montrer que si L est fini, alors $\text{Perm}(L)$ est régulier.

correction

Trivial, car dans ce cas, $\text{Perm}(L)$ est également fini.

5. Donner un exemple de langage L infini tel que $\text{Perm}(L)$ soit régulier.

correction

$L = \{a, b\}^*$ convient (car $\text{Perm}(L) = L$).

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.

correction

Non trivial, pour occuper les plus rapides!

Chapitre 84

(MT) plus courts chemins dans des graphes * (MT 24, ex 2, corrigé, retour d'oral — oral - 120 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.

correction

Correction de M.Péchaud

Avec une file de priorité implémentée par tas, $O((|A| + |S|) \log(|S|)) = (n + p) \log(n)$.

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 .

correction

Soit C un chemin, d'extrémités u et v . On note $w_h(C)$ et $w(C)$ ses longueurs pour w_h et w . Par télescope, $w_h(C) = h(u) - h(v) + w(C) = c + w(C)$ — où c est une constante si u et v sont fixés. Les plus courts chemins sont donc les mêmes.

Si le chemin C est un cycle, on reprend ce qui précède avec $u = v$, et l'on obtient $w(C) = w_h(C)$, d'où le résultat.

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.

3. Trouver h tel que w_h soit à valeurs positives ou nulles.

correction

Réfléchir sur un petit exemple paraît indispensable...

Pour tout sommet u , on note $h(u)$ la longueur d'un plus court chemin entre r et u (qui existe car il n'y a pas de cycle de poids strictement négatif dans le graphe).

Montrons alors que w_h est à valeurs positives.

Soient u et v deux sommets du graphe. Par définition de h , $h(v) \leq h(u) + w(u, v)$, d'où $w_h(u, v) = w(u, v) + h(u) - h(v) \geq 0$.

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.

correction

On commence par calculer les h en temps $O(np)$.

Le nouveau graphe étant à poids positifs, on peut alors utiliser l'algorithme de Dijkstra pour calculer les plus courts chemins entre tous les couples de sommets.

On obtient bien une complexité $n(n+p)\log(n)$.

L'algorithme de Floyd-Warshall est de complexité $O(n^3)$. En présence d'un graphe creux où $p\log(n) < n^2$, l'algorithme de Johnson est donc asymptotiquement meilleur.

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 $d(t, k)$.

correction

$d(s, k) = 0$ pour tout $k \in \mathbb{N}$.

Pour tout sommet t et tout $k \in \mathbb{N}^*$, $d(t, k) = \min(d(t, k-1), \min_{(t', t) \in A} d(t', k-1) + w(t', t))$.

- (b) En déduire le pseudocode d'un algorithme de complexité spatiale $O(np)$ répondant au problème posé.

correction

On utilise une implémentation ascendante par poids croissants pour calculer les $d(t, n-1)$ (car en l'absence de cycle de poids strictement négatif, il existe un plus court chemin entre deux sommets quelconques ne passant pas deux fois par le même sommet du graphe) :

```

1 fonction Distance( $G = (S, A)$ ,  $w$ ,  $s$ )
2   pour tout  $u$  dans  $S$  :
3      $d[u] = +\infty$ 
4    $d[s] = 0$ 
5   pour  $k = 1$  jusqu'à  $|S| - 1$  faire
6     pour chaque arc  $(u, v) \in A$  :
7       si  $d[u] + poids(u, v) < d[v]$  :
8          $d[v] := d[u] + poids(u, v)$ 
9   renvoyer  $d$ 

```

La complexité est bien $O(np)$.

NB : il s'agit de l'algorithme de Bellman-Ford.

Chapitre 85

(MT) Décidabilité de programmes engendrés par une grammaire * (MT 24, ex 2, corrigé — oral - 127 lignes)

*

Grammaires, Décidabilité, C, Logique propositionnelle,
sources : `mthaltbool.tex`

1. `HALT_BOOL_BOOL_C` est-il décidable ?

correction

Corrigé de M.Péchaud

Dans `bool_bool_c`, il n'y a pas de boucle `while`, ni de possibilité d'appel récursif.
Donc tout programme de `bool_bool_c` termine, et ce problème est donc 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 φ_η tel que pour ν_i , une valuation, $\varphi_\eta(\nu_i)$ correspond à la valeur de sortie de η en identifiant x_i et ν_i .

correction

On procède par induction sur l'arbre syntaxique pour construire une formule à partir du I situé immédiatement sous le S .

La notation de l'énoncé est pénible car portant sur les chaînes, on note plutôt ψ_a la formule correspondant à un sous-arbre enraciné en a . On définit alors

- $\psi_{true} = \top$ (ou $(\nu_i \vee \neg\nu_i)$ si l'on n'autorise pas \top)
 - $\psi_{false} = \perp$.
 - $\psi_V = \psi_{x_i}$ si x_i est le fils de V
 - $\psi_{x[i]} = \nu_i$
 - Si a étiqueté par E dérive en $b == c$, $\psi_a = (\psi_b \wedge \psi_c) \vee (\neg\psi_b \wedge \neg\psi_c)$
 - Si a étiqueté par I dérive en `return(b)`; $\psi_a = \psi_b$.
 - Si a étiqueté par I dérive en b (étiqueté par C) $\psi_a = \psi_b$.
 - Si d est étiqueté par C et dérive en `if(a) {b} else {c}`, $\psi_d = (\psi_a \wedge \psi_b) \vee (\neg\psi_a \wedge \psi_c)$
- Par induction, on a alors le résultat voulu.

3. Donner la formule associé au code suivant :

```
if (x[0]) {  
    if (x[0] == x[1]) {return false ;}  
    else {return true ;}  
}  
else {return x[1] == x[2] ;}
```

correction

$(\nu_0 \wedge (((\nu_1 \wedge \nu_2) \vee (\neg\nu_1 \wedge \neg\nu_2)) \wedge \perp) \vee (\neg(\nu_1 \wedge \nu_2) \vee (\neg\nu_1 \wedge \neg\nu_2) \wedge \top))) \wedge (\neg\nu_0 \wedge ((\nu_1 \wedge \nu_2) \vee (\neg\nu_1 \wedge \neg\nu_2)))$

4. `HALT_BOOL_C` est-il décidable ?

correction

Oui : il n'y a pas d'affectation dans le langage, donc soit on n'entre pas dans une boucle, soit celle-ci ne terminera jamais – en fonction de la valeur de vérité du booléen calculé.
On décide donc de la terminaison en exécutant le programme sur l'entrée, en renvoyant vrai si cela termine, et faux dès que l'on entre dans le code à l'intérieur d'une boucle.

5. *HALT_ALL_BOOL_C* est-il décidable ?

correction

Même chose qu'à la question précédente, en testant pour toutes les valeurs possibles de l'entrée, en nombre fini (2^n en notant n le plus grand entier tel que $x[n-1]$ apparaisse dans le code, valeur que l'on calcul facilement à partir du programme).

Annexe :

On définit la grammaire hors-contexte \mathcal{C} avec les règles de dérivation suivantes :

$$\begin{aligned} \mathbf{S} &\rightarrow \text{bool } f \text{ (bool } x[]) \{ \mathbf{I} \} \\ \mathbf{E} &\rightarrow \text{true} \mid \text{false} \mid \mathbf{V} \mid (\mathbf{E} == \mathbf{E}) \\ \mathbf{V} &\rightarrow x[i], i \in \mathbb{N} \\ \mathbf{I} &\rightarrow \text{return } \mathbf{E}; \mid \mathbf{B} \mid \mathbf{C} \\ \mathbf{B} &\rightarrow \text{while}(\mathbf{E}) \{ \mathbf{I} \} \\ \mathbf{C} &\rightarrow \text{if}(\mathbf{E}) \{ \mathbf{I} \} \text{ else } \{ \mathbf{I} \} \end{aligned}$$

bool_c est obtenue à partir de la grammaire \mathcal{C} .

bool_bool_c est obtenue à partir de la grammaire \mathcal{C} sans dérivation \mathbf{B} .

On définit les problèmes suivants :

HALT_BOOL_C :

Entrée $\kappa \in \text{bool_c}$

Sortie κ termine-t-il sur une entrée x ?

HALT_ALL_BOOL_C :

Entrée $\kappa \in \text{bool_c}$

Sortie κ termine-t-il sur toute entrée ?

HALT_BOOL_BOOL_C :

Entrée $\kappa \in \text{bool_bool_c}$

Sortie κ termine-t-il sur toute entrée

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 Σ . Montrer que $\mathcal{L}(\mathcal{A})$ est infini si et seulement si

$$\exists w \in \Sigma^*, |w| \in \llbracket n, 2n - 1 \rrbracket \text{ et } w \in \mathcal{L}(\mathcal{A}).$$

correction

On procède pour double implication.

\Leftarrow Soit w comme donné par la propriété. Le calcul étiqueté par w dans l'automate soit au moins n transitions, donc passe par au moins $n + 1$ états. Par le principe des tiroirs, il passe deux fois par un même état s . w s'écrit alors $w = xyz$, où $y \neq \emptyset$ étiquette un calcul de s à s .

On a donc $xy^*z \subset \mathcal{L}(\mathcal{A})$, et $\mathcal{L}(\mathcal{A})$ est en particulier infini.

\Rightarrow Si $\mathcal{L}(\mathcal{A})$ est infini, il contient au moins un mot w tel que $|w| \geq n$. Considérons un tel mot w le plus court possible. S'il est de longueur $\leq 2n - 1$, on a le résultat voulu.

Sinon, $|w| \geq 2n$, et on raisonne par l'absurde en reprenant le raisonnement du sens réciproque : le calcul étiqueté par w passe au moins deux fois par un même état. On considère un état s et un facteur y de w tel que y étiquette un calcul de s à s et que w soit minimal.

Alors $|y| \leq n$ (sans quoi le calcul étiqueté par y passe deux fois par le même état *intermédiaire* – en contradiction avec la minimalité de y). Si on note $w = xyz$, on a alors $w' = xz \in \mathcal{L}(\mathcal{A})$, et $n \leq |w'| = |w| - |y| < |w|$, en contradiction avec la minimalité de $|w|$.

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 $\mathcal{L}(\mathcal{A}) = \emptyset$?

correction

Un parcours de graphe permet de décider s'il existe un chemin de l'état initial à un état acceptant dans l'automate en temps linéaire en sa taille. Le problème est donc décidable, et dans **P**.

(b) **Entrée** un AFD \mathcal{A} et un mot u

Sortie a-t-on $u \in \mathcal{L}(\mathcal{A})$?

correction

L'algorithme consistant à suivre les transitions en lisant u est polynomial (linéaire en u selon la représentation de l'automate), donc le problème est décidable et dans **P**.

(c) **Entrée** un AFD \mathcal{A}

Sortie le langage $\mathcal{L}(\mathcal{A})$ est-il infini ?

correction

D'après la question 1, le problème est dans **NP** en se donnant comme certificat un mot de longueur entre n et $2n - 1$ reconnu par l'automate (sa longueur est polynomiale en la taille de l'automate, et la vérification s'effectue en temps polynomial). Donc ce problème est décidable.
Il est en fait aussi dans **P**, car on peut montrer en adaptant la réponse à la question 1 que le langage est infini si et seulement si il y a un cycle dans l'automate émondé, ce qui peut se vérifier en temps linéaire en la taille de l'automate.

(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 .

correction

Ce problème est dans **NP** : on se donne comme certificat une suite de dérivations permettant de dériver le mot u à partir du symbole initial S . Cette séquence est de longueur au plus $|u| - 1$ car l'application d'une règle fait augmenter d'au moins 1 la taille du mot, et la vérification qu'il s'agit d'une suite de dérivations valides s'effectue en temps polynomial.
Pas clair cependant qu'il soit dans P (c'est bien le cas, mais on ne peut pas attendre que vous le montriez sans indication supplémentaire!).

3. Citer un problème indécidable.

correction

Le problème de l'arrêt.

Chapitre 87

(MT) Récompenses pour un site web *

(MT 25, ex 1, retour élève remanié, corrigé (à préciser/compléter) — oral - 68 lignes)

*

Concurrence,
sources : mtconc1.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.

correction

Corrigé de M.Péchaud

Si 1000 processus effectuent le test sur le nombre de ressources de façon concurrente, et que leurs données de connexion sont valides, le nombre de ressources deviendra strictement négatif – autrement dit, trop de personnes auront une récompense.

2. Rappeler le principe de fonctionnement des mutex et des sémaphores.

correction

Cf cours.

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.

correction

Pas sûr de voir ce qui est attendu ici.

4. Proposer un algorithme avec un mutex ou un sémaphore.

correction

Avec un mutex, on peut simplement encadrer tout le corps de la fonction entre `lock` et `unlock`, mais un seul processus pourra accéder à la vérification des données, ce qui n'est pas très raisonnable.

Plus raisonnablement, on initialise un sémaphore avec 144 ressources. Un processus acquiert une ressource au

début, et la libère à la fin de son exécution – ce qui garantit qu’il n’y a jamais de 144 processus accédant simultanément à leur corps.

Chapitre 88

(MT) Algorithmique des textes * (MT 25, ex 1, retour élève remanié, corrigé — oral - 72 lignes)

*

Algorithmique des textes,
sources : mtxt2.tex

On considère la chaîne suivante sur l'alphabet $\{A, C, D, L, E, S\}$:

LESCASCADESDELESCALADE

1. Lui appliquer l'algorithme de Huffman.

correction

Corrigé de M.Péchaud

On a les nombres d'occurrences suivants :

ESSESEEESE

A	C	D	L	E	S
4	3	3	3	5	4

On obtient par exemple les regroupements succesifs suivants :

A(4) C(3) D(3) L(3) E(5) S(4)

A(4) CD (6) L(3) E(5) S(4)

AL (7) CD (6) E(5) S(4)

AL (7) CD (6) ES(9)

(AL)(CD) (13) ES (9)

(AL)(CD)(ES) 22

On obtient donc le code suivant :

A : 000

L : 001

C : 010

D : 011 E : 10 S : 11

2. Comparer le nombre de bits du texte initial et celui du texte compressé.

correction

En supposant chaque lettre codée sur 3 bits, le code initial à une longueur 66.

Le texte compressé à une longueur $2.9 + 3.13 = 57$.

3. Lui appliquer l'algorithme LZW en utilisant le dictionnaire suivant :
- | | | | | | |
|---|---|---|---|---|---|
| A | C | D | L | E | S |
| 1 | 2 | 3 | 4 | 5 | 6 |

correction

On écrit 4, et on ajoute LE(7) au dictionnaire.

On écrit 5, .

On écrit 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 Σ_2 tels que $V(m)$ soit pair est rationnel.

correction

Corrigé de M. Péchaud.

On remarque que les bits de poids faible sont à droite.

Cet ensemble est dénoté par $L = ((0+1)^*.0) + \epsilon$ – il est donc rationnel.

2. Montrer que le langage des mots de Σ_4 multiples de 3 est rationnel.

correction

On remarque que pour un mot $m = m_0 \dots m_k \in \Sigma_4^*$, $V(m) \equiv \sum_{l=0}^k m_l 4^{k-l} \equiv \sum_{l=0}^k m_l [3]$.

Le critère de divisibilité connu en base 10 s'applique donc en base 4 : un nombre écrit en base 4 est divisible par 3 si et seulement si la somme de ses chiffres l'est.

Le langage est donc reconnu par l'AFD

- dont l'ensemble des états est $\mathbb{Z}/4\mathbb{Z}$
- l'état initial est 0
- l'état acceptant est 0
- pour tout $i \in \mathbb{Z}/4\mathbb{Z}$, pour tout $a \in \mathbb{Z}/4\mathbb{Z}$, $i \xrightarrow{a} (i+a)[4]$.

Le langage est donc rationnel.

3. Le langage des mots de Σ_5 multiples de 3 mais pas de 2 est-il rationnel ?

correction

Soit $m = m_0 \dots m_k \in \Sigma_5^*$, $V(m) \equiv \sum_{l=0}^k m_l 5^{k-l} \equiv \sum_{l=0}^k m_l 2^{k-l} [3]$.

Or $2^0 \equiv 1[3]$, $2^1 \equiv 2[3]$, $2^2 \equiv 1[5]$, $2^3 \equiv 2[5]$

Si l est pair, la somme se réécrit $m_0 + 2m_1 + m_2 + 2m_3 + m_4 + \dots$

On peut alors reconnaître si $V(m)$ est multiple de 3 en adaptant l'automate de la question précédente.

L'ensemble des mots tels que l est pair est $V(M)$ est un multiple de 3 est donc reconnaissable.

De même si l est impair, et donc par union, l'ensemble des mots tels que $V(M)$ est multiple de 3 est reconnaissable.

On procède de même pour les multiples de 2, et par propriétés de stabilité des langages rationnels (par intersection et complémentaire – i.e. différence), le langage demandé est reconnaissable.

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.

correction

Une pile est une structure séquentielle permettant

- la création d'une pile vide
- le test d'une pile vide
- l'ajout d'un nouvel élément
- la récupération et suppression de l'élément le plus récemment ajouté.

De même pour une file, mais avec l'élément ajouté il y a le plus longtemps.

2. Comment implémenter une file avec deux piles ?

correction

On représente une file par un couple de piles – que l'on notera p_1 et p_2 dans la suite – l'intuition étant que les éléments contenus dans la file sont ceux de $\text{rev}(p_1) @ p_2$.

On définit alors

file_vide : renvoyer `(pile_vide, pile_vide)`

est_vide : renvoyer `est_vide(p1) && est_vide(p2)`

ajouter(e) : ajouter e à p_1

retirer(e) : si non `est_vide(p2)`, renvoyer `p2.retirer(e)` sinon, vider p_1 dans p_2 , puis renvoyer `retirer←(e)`

3. Discuter la complexité de votre solution.

correction

La complexité des trois premières opérations est immédiatement $O(1)$.

La dernière est de complexité au pire $O(n)$ (à cause du vidage de p_1 dans p_2), et de complexité amortie $O(1)$ (chaque élément retiré de la file passant exactement une fois de p_1 à p_2).

4. Prouver la correction de votre implémentation.

correction

La correction est triviale pour les deux premières fonctions.

Pour les deux dernières, on utilise le fait que l'invariant «les éléments ont été insérés dans l'ordre de $\text{rev}(p_2) \sim p_1$ » est garanti par toutes les fonctions à tout moment.

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 $o(n)$.

correction

On remarque que $f : i \mapsto T[i] - i$ est croissante : en effet, pour i valide, $f(i+1) - f(i) = T[i+1] - T[i] - 1 \geq 0$ car T est strictement croissante.

Donc si $T(0) > 0$, il n'y a pas de point fixe, et de même si $T(n-1) < n-1$.

Dans les cas d'égalité, on a immédiatement des points fixes.

Dans le cas où $T(0) < 0$ et $T(n-1) > n-1$, on procède par dichotomie sur $T(i) - i$.

On peut alors donner le pseudo-code d'une dichotomie – de complexité $O(\log n) = o(n)$ – le plus simple pour les preuves qui suivent étant d'en donner une version récursive :

```
1 pt_fixe(T, d, f) : # cherche un point fixe de T entre les indices d et f ←
   inclus
2   Si f < d :
3     Renvoyer None
4   m ← (f + d)/2
5   Si T[m] = m :
6     Renvoyer m
7   Si T[m] > m :
8     Renvoyer pt_fixe(T, d, m-1)
9   Si T[m] < m :
10    Renvoyer pt_fixe(T, m+1, f)
```

2. Prouver la terminaison et la correction de votre algorithme.

correction

Cf cours – pour la terminaison, on constate que $f - d + 1$ est un entier positif qui décroît strictement à chaque appel récursif (formellement, on procède par récurrence forte sur cette quantité).

Pour la correction, on procède également par récurrence forte.

3. Prouver rigoureusement sa complexité.

correction

On montre par récurrence forte sur $f - d + 1$ que le nombre d'appels récursifs est en $\lfloor \log_2 f - d + 1 \rfloor$

4. Proposer une solution permettant de calculer tous les points fixes du tableau. Quelle est sa complexité ?

correction

Une solution consiste à trouver un point fixe, puis à faire ensuite une exploration linéaire du tableau de part et d'autre. On obtient une complexité $O(n)$.

On peut en fait faire mieux en adaptant l'algorithme de dichotomie pour calculer les points fixes de plus petits et de plus grands indices m et M – l'ensemble des points fixes étant alors $\llbracket m, M \rrbracket$ et calculé en $O(\log n)$.

Chapitre 92

(MT) Langages des préfixes * (MT 25, ex 1, retour élève à modifier, corrigé — oral - 56 lignes)

*

Langages, Grammaires, Lemme de l'étoile,
sources : `mtlang6.tex`

Soit Σ 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.

correction

Soit L régulier et \mathcal{A} un AFD reconnaissant L .

On obtient alors un AFD reconnaissant $\text{Pref}(L)$ en rendant terminaux tous les états co-accessibles de \mathcal{A} .

2. La réciproque est-elle vraie ?

correction

Non. Notons a une lettre.

Soit $L = \{a^{i^2}, i \in \mathbb{N}\}$.

On montre par le lemme de l'étoile que L n'est pas régulier.

Mais $\text{Pref}(L) = a^*$ est régulier.

3. Montrer que si L est hors-contexte, $\text{Pref}(L)$ l'est également.

correction

Me semble fortement non trivial sans indication supplémentaire (e.g. forme particulière de grammaire engendrant L ?)

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 Σ , et l'on note \mathcal{L} l'ensemble des langages sur Σ .

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.

correction

On se donne un AFD \mathcal{A} reconnaissant L .

On construit alors l'automate généralisé \mathcal{A}' en remplaçant chaque étiquette de transition a par $f(a)$.

2. Montrer que si L est un langage non contextuel, $L' = f(L)$ est également un langage non contextuel.

correction

On considère une grammaire \mathcal{G} engendrant L .

Pour chaque lettre $a, b \dots \in \Sigma$, on crée un nouveau non terminal $A, B \dots$ (quitte à renommer les non terminaux de \mathcal{G}).

On remplace toute les occurrences de a dans les membres de droite des règles de production de \mathcal{G} par A , et de même pour toutes les autres lettres. On obtient une grammaire \mathcal{G}' .

Pour tout $x \in \Sigma$, $f(x)$ est régulier, donc non contextuel. On se donne une grammaire engendrant $f(x)$ (avec uniquement des non terminaux non déjà rencontrés, et l'on remplace son symbole initial par X). On ajoute toutes règles obtenues à \mathcal{G}' , et l'on répète l'opération pour tout $x \in \Sigma$.

On obtient alors une grammaire engendrant L' .

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.

correction

On remarque que $G_0 = \{b^p a^q c, q \leq p\} \cup \{b^p a^q c, q \geq p + 2\}$
 $= \{b^p a^q c, q \leq p\} \cup \{b^p a^q aac, q \geq p\}$

Ce langage est engendré par la grammaire donnée par les règles suivantes (où les langages engendrés par s' et S'' correspondent aux deux langages ci-dessus) :

$S \rightarrow S' \mid S''$

$S' \rightarrow Tc$

$T \rightarrow bTa \mid bT \mid \epsilon$

$$S'' \rightarrow Uaac$$

$$U \rightarrow bUa \mid Ua \mid \epsilon$$

4. Montrer que G_0 n'est pas régulier.

correction

On raisonne par l'absurde. S'il l'était, $(a^*b^*) \setminus G_0$ le serait aussi.

Notons $L = (a^*b^*) \setminus G_0 = \{b^p a^q, q = p + 1\} = \{b^p q^{p+1}, p \in \mathbb{N}\}$.

(Ébauche)

On montre alors que ce langage n'est pas rationnel, d'où une contradiction.

5. ????? Appliquer une certaine substitution régulière à G_0 pour obtenir un certain langage G donné.
6. Montrer que G est non contextuel.

$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi} \wedge_i$	$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} \wedge_e^g$	$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi} \wedge_e^d$
---	--	---

Chapitre 95

(MT) Langages et facteurs * (MT 25, ex 1, retour élève, corrigé — 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.

correction

Corrigé de M.Péchaud
(Ébauche)

b^*a^*

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 .

correction

(Ébauche)

$Q = \{q_1, q_2\}$. (q_2 codant pour «on vient de lire un a », et q_1 pour sa négation, la lecture de ab faisant échouer le calcul.)

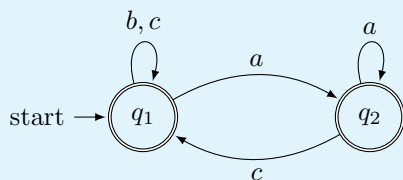
$q_i = q_1$. $Q_F = Q$.

Une boucle étiquetée b, c sur q_1 .

Une transition a de q_1 vers q_2 .

Une boucle étiquetée a sur q_2 .

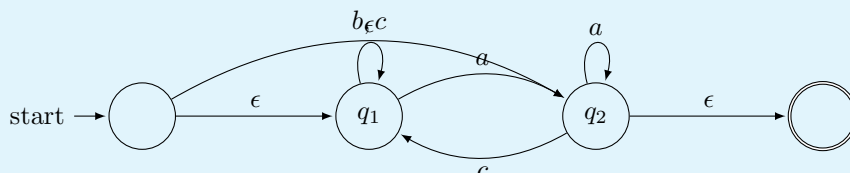
Une transition c de q_2 vers q_1 .



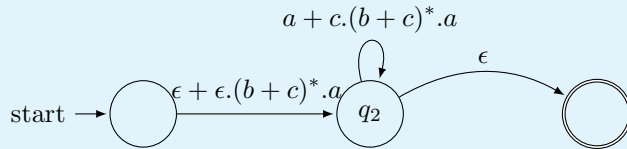
3. Appliquer l'algorithme d'élimination des états sur cet automate pour trouver une expression régulière R qui dénote L_2 .

correction

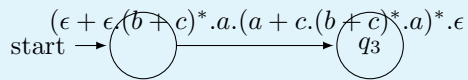
On transforme l'automate en automate généralisé :



On élimine l'état q_1 :



On élimine l'état q_2 :



On en déduit que le langage reconnu est dénoté par

$$R = (\epsilon + (b + c)^*.a).(a + c.(b + c)^*.a)^*.$$

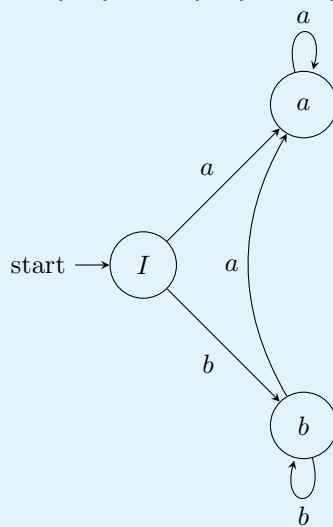
4. Appliquer l'algorithme de Berry-Sethi pour trouver un automate reconnaissant L_1 .

correction

L'expression b^*a^* est déjà linéarisée.

On écrit L_1 sous la forme $L_{\langle I, F, T, \alpha \rangle}$, où

$I = \{a, b\}$, $F = \{a, b\}$, $T = \{ba, aa, bb\}$, $\alpha = \top$.



Chapitre 96

(MT) k plus proches voisins * (MT 25, ex 1, retour élève, corrigé — oral - 46 lignes)

*

Ia,

sources : mtia2.tex

1. Expliquer le principe de l'algorithme des k -plus proches voisins. Donner sa complexité.

correction

Cf cours pour le principe.

S'il y a n points de données, et que la distance se calcule en $O(1)$, l'algorithme peut être implémenté en $O(nk)$ (en faisant un début de tri sélection par exemple).

2. Donner une façon d'améliorer cette complexité en dimension 1.

correction

En dimension 1, on peut commencer par classer les points de données (pré-traitement en $O(n \log(n))$).

Étant donné un nouveau point, on détermine sa position parmi les points de données par dichotomie, puis on utilise une recherche linéaire à droite et à gauche pour déterminer ses k -plus proches voisins. On obtient donc un algorithme en $O(\log(n) + k)$ une fois les points classés.

3. Donner un autre algorithme qui répond au même problème que les k -plus proches voisins.

correction

L'algorithme des k -plus proches voisins répond à un problème de classification par apprentissage supervisé. L'algorithme ID3 répond au même type de problème, mais dans le cas où les points ont des caractéristiques discrètes plutôt que des valeurs permettant de les plonger naturellement dans \mathbb{R}^n .

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 Σ .

1. Montrer que (E) admet une unique solution, que l'on explicitera.

correction

Par analyse/synthèse :

Analyse

Si L est solution, il contient b , puis ab , puis aab , puis par récurrence immédiate, $a^n b$, donc $a^*b \subset L$.

Réciproquement, on montre que tout mot $u \in L$ de longueur $n \in \mathbb{N}$ s'écrit $a^{n-1}b$, par une récurrence facile sur n .

Donc la seule solution possible est a^*b .

Synthèse

Si $L = a^*b$, $aL + b = a^+b + b = (a^+ + \epsilon)b = a^*b$.

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 $L = A^*B$.

correction

- Soit $L = A^*B$.

$$A.L \cup B = A.A^*B \cup B = A^+.B \cup \{\epsilon\}.B = (A^+ \cup \{\epsilon\}).B = A^*.B = L.$$

Donc L est bien solution de (E).

- Montrons que c'est la plus petite. Soit L une solution de (E).

Alors $B \subset L$, puis, comme $A.L \subset L$, par une récurrence immédiate, $A^n B \subset L$. Donc $A^*B \subset L$ – ce qui prouve le résultat.

Montrons maintenant l'unicité à l'aide de l'hypothèse $\epsilon \notin A$.

Soit L' une solution de l'équation.

Montrons que $w \in L' \Rightarrow w \in L$ par récurrence forte sur $|w|$.

Si $w = \epsilon$, comme $\epsilon \notin A$, $w \notin A.L'$, d'où $w \in B \subset L$.

Supposons le résultat vrai jusqu'au rang $n \in \mathbb{N}$, et soit $w \in L'$ de longueur $n + 1$. $w \in AL' \cup B$.

- Si $w \in B$, comme $B \subset L$, $w \in L$.

- Si $w \in AL'$, il existe $u \in A$ et $v \in L'$ tels que $w = uv$. Comme $u \neq \epsilon$, $|v| < |w|$. Donc par hypothèse de récurrence, $v \in L$, et donc $w \in AL \subset L$.

3. Que peut-on dire de (E') si $\epsilon \in A$?

correction

On perd l'unicité :

Considérons le cas où $A = \epsilon$, et B quelconque.

Alors tout langage L contenant B est solution de l'équation. En effet, $A.L \cup B = L \cup B = L$.

Il n'y a donc pas unicité de la solution (considérer n'importe quel langage $B \neq \Sigma^*$, et L contenant strictement B).

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**.

correction

Le problème 2 est clairement dans NP : on se donne comme certificat A' , et l'on vérifie en temps polynomial (par un simple parcours en profondeur du graphe) que G' est acyclique.

Réduisons polynomialement 1 à 2 :

Soit $G = (S, A), k$ une instance de 1. En notant $a = |A|$, on se donne l'instance $G, a - k$ de 2.

Alors $G, a - k$ est une instance positive de 2

ssi il existe un ensemble d'arcs A' de cardinal $a - k$ tel que $G' = (S, A')$ soit acyclique

ssi il existe un ensemble d'arcs $E (= \bar{A}')$ de cardinal k tel que tout cycle de G a au moins un arc dans E

ssi G, k est une instance positive de 1.

Donc 1 est **NP-difficile**, donc **NP-complet**.

2. Définir un problème d'optimisation \mathcal{P}_{opt} associé à 2.

correction

Entrées : un graphe $G(S, A)$

Sorties : k maximal tel qu'il existe $A' \subset A$ tel que $|A'| = k$, $G = (S, A')$ soit acyclique.

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} .

correction

Si A_1 est renvoyé, tout arc $(e, f) \in A_1$ vérifie $f > e$. Le long d'un chemin d'arcs de A_1 , les sommets sont ordonnés de façon strictement croissante, et il ne peut donc pas y avoir de cycle constitué d'arcs de A_1 . De même pour A_2 .

4. Montrer que cet algorithme est une 1/2-approximation pour \mathcal{P}_{opt} .

correction

Notons OPT la solution de \mathcal{P}_{opt} , et A' un ensemble d'arc correspondant.

Alors $OPT = |A'| \leq |A|$.

Comme $|A_1| + |A_2| = |A|$, l'ensemble B renvoyé par l'algorithme vérifie $|B| \geq |A|/2$.

En enchaînant les inégalités, $|B| \geq OPT/2$ – et cet algorithme est donc bien une 1/2-approximation pour \mathcal{P}_{opt} .

5. Discuter sa complexité.

correction

Sa complexité est en $O(|A|)$ si l'on peut parcourir en temps linéaire les arcs (par exemple avec une représentation par listes d'adjacences).

Chapitre 99

(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.

correction

Un graphe à deux sommets reliés et d'une couleur différente au départ.

2. Montrer que pour tout graphe biparti, il existe une configuration initiale de couleurs empêchant qu'il y ait jamais un consensus.

correction

Notons (S_1, S_2) la bipartition des sommets (qui est non triviale car le graphe est connexe et a au moins deux sommets).

On pose $\gamma(s) = \text{ROUGE}$ si $s \in S_1$, NOIR sinon.

Une itération intervertira les couleurs, et le processus fera osciller entre deux coloriage dont aucun des deux n'est 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 $p < 1$.

correction

Le graphe étant connexe, il existe un chemin de s_1 à s_3 .

Notons s' le dernier sommet de ce chemin tel que tous les sommets de s_1 à s' soient de même couleur que s_1 .

Si $s' = s_1$, s_1 a deux voisins de couleur différente (s_2 et son successeur dans le chemin), et sinon, il s' a deux voisins de couleur différente également (son prédécesseur et son successeur).

Donc $p < 1$ (car s' peut prendre les deux couleurs à l'itération suivante, indépendamment des autres sommets – il ne peut donc pas y avoir consensus de façon certaine à l'étape suivante).

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 $q > 0$.

correction

Chaque sommet de C peut garder sa couleur avec probabilité non-nulle (car il a au moins un voisin dans C , car $|C| \geq 2$).

On se donne un voisin s d'un sommet de C , de couleur opposée à celle de C (un tel sommet existe – sans quoi il y aurait consensus). s peut devenir de la couleur de C avec probabilité non-nulle.

Par indépendance des choix, il y a donc une probabilité non nulle que $C \cup \{s\} \subset C'$, ce qui implique que $|C'| > |C|$.

Donc $q > 0$.

5. Montrer que pour tout graphe non-biparti, la probabilité de finir par obtenir un consensus est égale à 1.

correction

Considérons un graphe non biparti. Pour chaque coloriage c il y a donc au moins deux sommets s_1, s_2 voisins coloriés de la même couleur – on choisira arbitrairement les deux sommets voisins de plus petit couple d'indice pour l'ordre lexicographique.

D'après la question précédente, la probabilité q_c d'agrandir la composante connexe de même couleur que s_1 et s_2 est non nulle : $q_c > 0$.

Il y a un nombre fini de coloriages. Donc il existe $q = \min_c q_c$, et $q > 0$.

La probabilité d'agrandir une composante connexe de sommets de même couleur est donc d'au moins q à chaque étape.

La probabilité d'agrandir p fois de suite une telle composante (ou d'arriver à un consensus avant les p itérations) est donc au moins $q^p > 0$.

Quelle que soit la configuration de départ, si l'on fixe p suffisamment grand ($|S|$ – la taille de la composante de départ), on a donc une probabilité $q^p \neq 0$ d'arriver à un consensus au bout de p étapes ou moins.

Il est donc presque certain de finir par arriver à un consensus.

Chapitre 100

(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 ?

correction

Oui, car ce langage s'écrit $a(aaa)^*$.

2. $\{a^{3^k+1}, k \in \mathbb{N}\}$ est-il rationnel ?

correction

Non. On le prouve par l'absurde. Notons L ce langage, que l'on suppose donc rationnel.

Soit N donné par le lemme de l'étoile, et $u = a^{3^N+1}$ – de longueur $|u| \geq N$.

Ce mot contient d'après le lemme un facteur itérant, de longueur $p \geq 1$.

Donc pour tout $k \in \mathbb{N}$, $a^{3^N+1-p+kp} \in L$ – il y a donc un nombre infini de couples de mots (distincts) de L distants de k .

C'est incohérent avec le fait que $3^{k+1} + 1 - (3^k + 1) = 2 \cdot 3^k > k$, et donc qu'il y a un nombre fini de couples de mots (distincts) de L distants de k ou moins.

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 Σ est régulier si et seulement si $\text{Taille}(L)$ est ultimement périodique.

correction

Pour la réciproque, on se donne N et P les paramètres de $\text{Taille}(L)$, et on remarque que L est la réunion finie d'un langage fini et de langages de la forme $\{a^{x_0+Nk}, k \in \mathbb{N}\} = a^{x_0}(a^N)^*$, qui sont réguliers.

Pour le sens direct :

- on peut se donner un AFD reconnaissant le langage et de raisonner sur ses formes possibles (qui sont réduites essentiellement à des $\ll \rho \gg$);
- on peut raisonner avec le lemme de pompage : soit n donné par le lemme. On note $P = n!$. Soit $u = a^m$ avec $m \geq n$. Alors il existe $k \in \llbracket 1, n \rrbracket$ tel que $a^m(a^k)^* \subset L$. Donc $a^m a^{n!} = a^m a^{k \cdot (n!/k)} \in L$.

4. Pour un alphabet quelconque, que dire du lien entre ultime périodicité et régularité du langage ?

correction

(Ébauche)

La régularité implique l'ultime périodicité : on considère un AF reconnaissant le langage, on remplace toutes les lettres sur les transitions par a , et on est ramené à la question précédente.

La réciproque est fautive. On considère $A = \{a, b\}$, et $L = a^{3^k+1} + b^*$. $\text{Taille}(L) = \mathbb{N}$ est immédiatement ultimement périodique, mais le langage n'est pas rationnel, sinon $a^* \cap L$ le serait, et nous avons prouvé en question 2 que ça n'est pas le cas.

Chapitre 101

(MT) Représentation fusionnée de deux mots ** (MT 25, ex 2, retour élève remanié, corrigé — oral - 116 lignes)

**

Algorithmique des textes, Programmation dynamique,
sources : mtxt1.tex

On s'intéresse dans cet exercice à une représentation compacte de deux mots sur un même alphabet Σ . Plus précisément, étant donnés deux mots A et B sur Σ , 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 ?

correction

$10 + 3\Delta$.

2. En fonction de Δ , lequel des trois résumés ci-dessus a la taille minimale ?

correction

$$10 + 3\Delta = 8 + 6\Delta \Leftrightarrow \Delta = 2/3.$$

$$12 + 2\Delta = 8 + 6\Delta \Leftrightarrow \Delta = 1.$$

$$12 + 2\Delta = 10 + 3\Delta \Leftrightarrow \Delta = 2.$$

Donc pour $n = 1$, les deux premiers résumés sont les meilleurs, pour $n = 2$ le deux derniers sont les meilleurs, et pour $n > 2$, le deuxième est le seul meilleur.

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 $d(|A|, |B|)$?

correction

$d(i, 0) = \Delta + i$ et $d(0, j) = \Delta + j$ — atteints respectivement par $\uparrow A^{(i)}$ et $\downarrow B^{(j)}$. C'est optimal, car le résumé doit contenir au moins une flèche et les caractères du préfixe considéré.

4. Trouver une relation de récurrence pour $d(i, j)$.

correction

On considère un résumé de taille minimale de $A^{(i)}$ et $B^{(j)}$, où $i, j \in \mathbb{N}^*$. Le dernier facteur $f_n u_n$ peut avoir l'une des trois formes $\uparrow u$, $\downarrow u$ ou $\updownarrow u$. On notera $k = |u|$

- Si c'est $\uparrow u$, $d(i, j) = d(i - k, j) + \Delta + k$
- Si c'est $\downarrow u$, $d(i, j) = d(i, j - k) + \Delta + k$
- Si c'est $\updownarrow u$, $d(i, j) = d(i - k, j - k) + \Delta + k$ (où les k dernières lettres de $A^{(i)}$ et $B^{(j)}$ doivent coïncider (condition $(*)$ dans la formule ci-dessous))

On obtient donc $d(i, j) = \min \left(\min_{k \leq i} d(i - k, j) + \Delta + k, \min_{k \leq j} d(i, j - k) + \Delta + k, \min_{\substack{k \leq j \\ k \leq i \\ (*)}} d(i - k, j - k) + \Delta + k \right)$.

5. Donner en pseudocode un algorithme prenant en entrée Δ et deux mots A et B , et renvoyant la taille minimale d'un résumé des deux mots.

correction

(Ébauche)

On utilise une mémorisation ascendante pour stocker les $d(i, j)$ déjà calculés, et deux boucles imbriquées pour parcourir i, j par valeurs croissantes. On effectue les calculs à l'aide de la formule précédente.

6. Quelle est sa complexité ?

correction

Chaque calcul de min avec la formule ci-dessus est de complexité $O(\max(n, p))$ en notant $n = |A|$ et $p = |B|$. On obtient donc une complexité $O(np \max(n, p))$ – donc cubique en la taille maximale des deux mots.

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 μ sur \mathcal{V} telle que μ soit un modèle de toute formule de E .

1. Quel est le nombre de tables de vérités possibles pour les formules utilisant ces variables ?

correction

Corrigé de M.Péchaud

Si l'on note n le nombre de variables, il y a 2^n tables de vérité possibles.

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.

correction

On considère la relation d'équivalence sur E «avoir la même table de vérité».

D'après la question précédente, il n'existe qu'un nombre fini de classes d'équivalences pour cette relation.

On forme F fini en choisissant une formule représentante pour chaque classe.

Immédiatement, F est satisfiable si et seulement si E l'est.

3. Prouver le théorème de compacité : E est satisfiable ssi tout F fini inclus dans E l'est.

correction

Le sens direct est immédiat : si $\mu \models E$, μ est un modèle de toute formule de E , donc de tout $F \subset E$ (en particulier si F est fini).

Le sens réciproque découle de la question précédente : si l'hypothèse est valide, la formule F finie construite à la question précédente est en particulier satisfiable, donc 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.

correction

Toute formule est φ satisfiable (par $\mu : \mathcal{V} \mapsto 1$ si φ est un littéral positif, et par $\mu : \mathcal{V} \mapsto 1$ sinon si c'est un littéral négatif).

Mais E n'est pas satisfiable, car par une disjonction de cas triviale sur $\mu(p_0)$, p_0 et $\neg p_0$ ne peuvent pas être satisfaites simultanément.

5. On considère $E = \bigcup_{i \in \mathbb{N}} \{p_{2i} \longrightarrow p_{2i+1}, p_{2i+1} \longrightarrow \neg p_{2i+2}, \neg p_{2i} \longrightarrow p_{2i+2}\}$.

Chaque formule contenue dans E est-elle satisfiable ?

E est-elle satisfiable.

correction

Chaque formule est satisfiable pour le même genre de raison qu'à la question précédente (on se débrouille par exemple pour évaluer à *faux*) chaque prémisse d'implication.

Concernant la satisfiabilité de E , on écrit quelques implications qu'il contient pour fixer les idées :

- $p_0 \longrightarrow p_1$
- $p_1 \longrightarrow \neg p_2$
- $\neg p_0 \longrightarrow p_2$
- $p_2 \longrightarrow p_3$
- $p_3 \longrightarrow \neg p_4$
- $\neg p_2 \longrightarrow p_4$
- $p_4 \longrightarrow p_5$
- $p_5 \longrightarrow \neg p_6$
- $\neg p_4 \longrightarrow p_6$
- ...

E est satisfiable par la valuation μ telle que pour tout $i \in \mathbb{N}$,
 $\mu(p_{4i}) = \mu(p_{4i+1}) = 1$, $\mu(p_{4i+2}) = \mu(p_{4i+3}) = 0$.

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.

correction

(Ébauche)

On utilise une stratégie de programmation dynamique par k croissants en remarquant que $D(i, j, k + 1) = D(i, j, k) * D(i + 1, j, k) * D(i, j + 1, k) * c(i + k, j + k)$ — où cette dernière quantité vaut 1 si la case $(i + k, j + k)$ est blanche, 0 sinon.

La complexité est bien $O(n^3)$.

2. Adapter l'algorithme précédent pour trouver le plus grand carré de chaque couleur.

correction

(Ébauche)

On pourrait créer une 4ème dimension correspondant à la couleur, mais c'est inefficace car s'il y a un carré de côté $k \geq 1$ d'une couleur donnée de coin en haut à gauche en (i, j) , il n'y en a pas pour toute autre couleur. La couleur d'un carré est de plus entièrement déterminé par la couleur de l'une quelconque de ses cases.

On adapte alors la formule précédente pour calculer la présence d'un carré de n'importe quelle couleur : $D(i, j, k + 1) = 1$ si et seulement si $D(i, j, k) * D(i + 1, j, k) * D(i, j + 1, k) * c(i + k, j + k) = 1$ et que les 4 coins sont de même couleur.

Une fois les $D(i, j, k)$ calculés (toujours en $O(n^3)$), on parcourt les $D(i, j, k)$ en regardant de plus si $D(i, j, k) = 1$ la couleur de la case (i, j) qui donne la couleur du carré, et l'on met à jour au fur et à mesure le plus grand carré trouvé pour chaque couleur.

L'algorithme est donc en $O(n^3 + c)$, où c est le nombre de couleurs.

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.

correction

(Ébauche)

Ce problème est dans **P**.Une instance est positive si et seulement si il existe un indice i tel qu'au moins k lignes aient k cases blanches à partir de l'indice i , ce qui se teste en temps polynomial.

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.

correction

(Ébauche)

On montre que le problème est **NP**-complet.

- On montre qu'il est dans **NP** en se donnant comme certificat une permutation des lignes et une permutation des colonnes.
- On montre le caractère **NP**-difficile du problème par réduction depuis **BICLIQUE** : chaque sommet de U correspond à une ligne, et chaque sommet de V à une colonne. On relie $u \in U$ et $v \in V$ par une arête si et seulement si la case correspondante est blanche.

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 ?

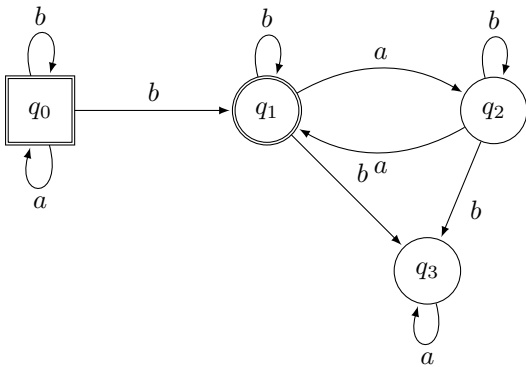
Chapitre 104

(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.



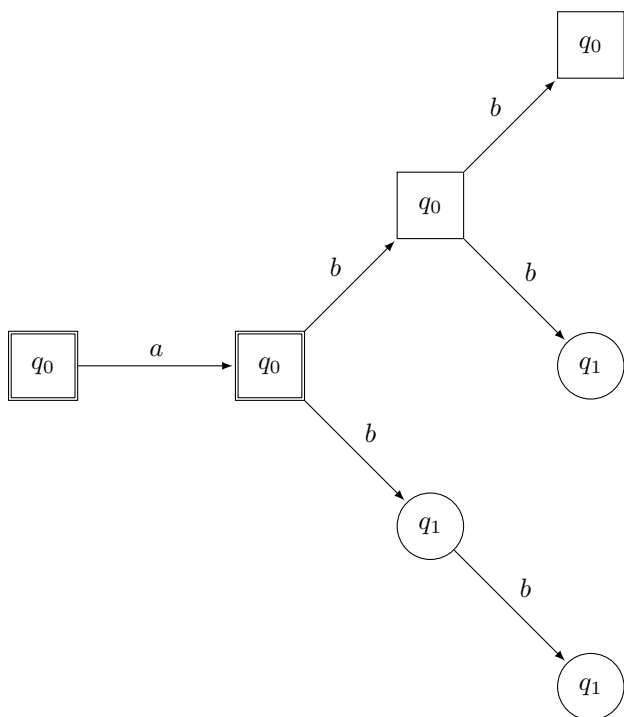
On appelle *automate alternant* la donnée d'un sextuplet $\mathcal{A} = (\Sigma, Q_{\forall}, Q_{\exists}, q_0, F, \delta)$.

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 ?

correction

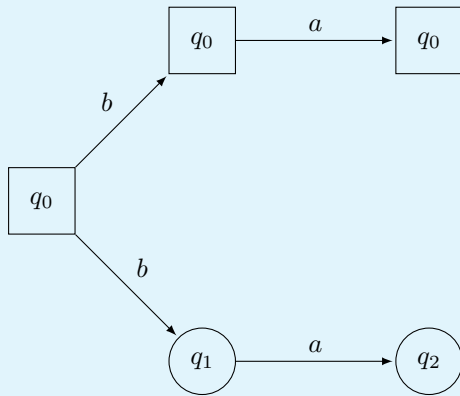
On donne le calcul suivant (qui n'est pas le seul possible) :

Toutes ses feuilles étant des états acceptants, le mot est donc accepté par l'automate.

2. Donner en justifiant rapidement un mot non accepté par l'automate.

correction

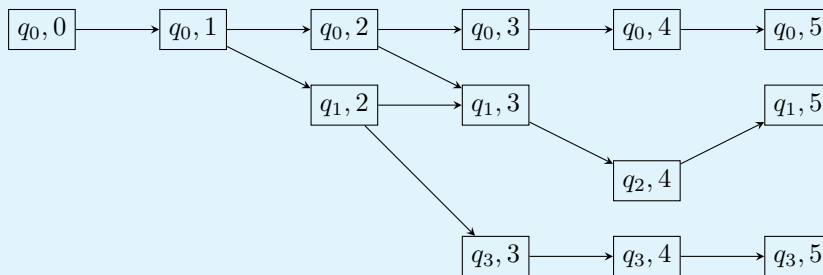
C'est le cas de *ba*.
L'unique calcul pour ce mot est le suivant :



Ce calcul a $q_2 \notin F$ comme feuille, donc le mot n'est pas accepté.

3. Dessiner $G(abbaa, \mathcal{A}_1)$ en se restreignant aux états accessibles à partir de $(q_0, 0)$.

correction



4. Montrer un état non final gagnant pour Automaton, et un pour Pathfinder dans l'instance de la question précédente.

correction

$(q_3, 3)$ est gagnant pour Automaton (la seule partie termine dans $(q_3, 5)$ et $q_3 \notin F$).
 $(q_0, 2)$ est gagnant pour Pathfinder (les deux parties possibles se terminent dans $(q_0, 5)$ et $(q_1, 5)$, et $q_0, q_1 \in F$).

5. Quel personnage a une stratégie gagnante dans cette instance ?

correction

C'est Pathfinder.

- Le premier coup d'Automaton est forcé.
- Le second coup d'Automaton peut mener
 - à $(q_0, 2)$, qui est gagnant pour Pathfinder comme nous l'avons vu ;
 - ou à $(q_1, 2)$. Alors, Pathfinder joue $(q_1, 3)$, et la partie se terminera nécessairement à $(q_1, 5)$, qui est gagnant pour Pathfinder.

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.

correction

Supposons que u est accepté par \mathcal{A} , et considérons un calcul réussi, i.e. un arbre dont toutes les feuilles sont des états acceptants.

On définit la stratégie suivante pour Pathfinder : en une position (q, i) contrôlée par Pathfinder

- s'il y a un sommet étiqueté par q à profondeur i dans le calcul réussi, la stratégie consiste pour Pathfinder à jouer l'unique fils de ce sommet dans l'arbre ;
 - sinon, Pathfinder joue un coup arbitraire (Pathfinder n'arrivera jamais dans un tel état s'il suit sa stratégie).
- Par récurrence sur la longueur d'une partie, chaque partie suivant cette stratégie correspond à une branche dans l'arbre – qui se termine donc dans un état $F \times \{|u|\}$. La stratégie est donc gagnante pour Pathfinder.

7. Prouver la réciproque.

correction

(Ébauche)

On reprend les idées de la question précédente : on construit un calcul en choisissant comme fils des sommets étiquetés $q \in Q_{\exists}$ à profondeur $i < |u| - 1$ le sommet q' , où $(q, i) \rightarrow (q', i + 1)$ est donné par la stratégie gagnante pour Pathfinder.

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`?

correction

Les sommets sont dans la pile, les tableaux de successeurs sur le tas, et `g` est un pointeur, qui est sur la pile.

2. Tracer le graphe correspondant.

correction

3. Y a-t-il des graphes que l'on ne peut pas créer avec cette structure ?

correction

Question un peu bizarre. La structure ne permettra que d'avoir accès aux sommets accessibles à partir de `s0`. En particulier, un graphe non connexe ne pourra jamais être parcouru intégralement.

4. Définir une représentation de graphe par listes d'adjacence.

correction

C'est une représentation par tableau (ou dictionnaire) de listes, où pour chaque sommet, la liste est la liste des successeurs de ce sommet.

5. Comment modifier la structure donnée pour implémenter un graphe par listes d'adjacence ?

correction

On pourrait garder la structure `sommet` telle quelle (ce n'est pas exactement une liste, mais un tableau permettant un ajout en $O(1)$), et modifier `graphe` de la façon suivante :

```

1 struct graphe {
2     sommet* sommets; // tableau de tous les sommets
3     int n;           // nombre de sommets
4 };
5
6 typedef struct graphe graphe;
```

6. Avec cette nouvelle représentation, écrire une fonction `void supprime(graphe g, sommet* s)` qui supprime le sommet `s` du graphe.

correction

(Ébauche)

Cette question me paraît assez pénible : il faut écrire une fonction auxiliaire supprimant un sommet donné d'un tableau de successeurs (en décalant tous les sommets qui apparaissent après). On l'applique à tous les sommets (à l'exception du sommet supprimé). Puis on supprime le sommet passé en argument de `g` (en décalant tous les sommets suivants dans `g.sommets` et en décrémentant `g.n`).

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 ?

correction

Question peu claire – mais si l'on veut une solution équitable entre les deux groupes, il suffit d'écrire les fonctions pour League, et les fonctions pour Overwatch s'en déduiront par symétrie, en intervertissant les deux jeux.

2. Écrire en pseudocode des fonctions `league_` grâce à un mutex, des variables globales de votre choix et de l'attente active.

correction

On crée deux variables locales pour le nombre de joueurs de chaque jeu dans l'espace, ainsi qu'un mutex.

```
1 league_entrer() :
2     Répéter indéfiniment :
3         lock(mutex)
4         Si nb_overwatch == 0 t nb_league < 5 :
5             nb_league += 1
6             unlock(mutex)
7             break
8         unlock(mutex)
9         attendre_un_peu()
10
11 league_sortir() :
12     lock(mutex)
13     nb_league -= 1
14     unlock(mutex)
```

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.

correction

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 - 94 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.

correction

Un graphe est k -coloriable si et seulement si chacun de ses composantes connexes l'est – on se ramène donc au cas d'un graphe connexe

On procède par retour sur trace en partant d'un sommet s_0 et en choisissant arbitrairement $c(s_0)$.

```
1 k_coloriable(G, k) :
2   couleurs = tableau de taille |V|, initialisé à NONE
3
4   Renvoyer colorier_sommet(G, 0, couleurs, k)
5
6 colorier_sommet(G, v, couleurs, k) :
7   if v == nombre de sommets dans G :
8     Renvoyer vrai // tous les sommets colorés sans conflit
9
10  Pour couleur variant de 0 à k-1 :
11    if est_valide(G, v, couleurs, couleur) :
12      couleurs[v] ← couleur
13      if colorier_sommet(G, v+1, couleurs, k) :
14        Renvoyer vrai
15      couleurs[v] ← NONE
16
17  Renvoyer faux // aucune couleur ne fonctionne
18
19 est_valide(G, v, couleurs, couleur) :
20  pour chaque voisin u de v dans G :
21    if couleurs[u] == couleur :
22      Renvoyer faux
23  Renvoyer vrai
```

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.

correction

On propose la conjonction des formules suivantes :

- Pour tout $s \in S$, $v_s^0 \vee \dots \vee v_s^{k-1}$ (tout sommet est colorié) ;
- Pour tout $s \in S$, et pour tout $i \neq j \in \llbracket 0, k-1 \rrbracket$, $\neg(v_s^i \wedge v_s^j)$ (un sommet ne peut pas être de deux couleurs) ;
- Pour tout $(s, s') \in A$ et tout $i \in \llbracket 0, k-1 \rrbracket$, $\neg(v_s^i \wedge v_{s'}^i)$ (deux sommets adjacents ne peuvent pas être de même couleur).

3. Montrer que $\chi \leq \Delta(G) + 1$ où $\Delta(G)$ est le degré maximum de G .

correction

On note $k = \Delta(G) + 1$, et on utilise l'algorithme glouton suivant pour k -colorier chaque composante connexe de G .

On part d'un sommet s arbitraire colorié d'une couleur arbitraire, et on effectue un parcours. Pour chaque sommet atteint qui n'est pas déjà colorié, on lui attribue une couleur valide par rapport à ses voisins déjà coloriés (ce qui est possible car il a au plus $k-1$ voisins coloriés, il y a donc au moins une couleur valide).

4. Donner un exemple de graphe G tel que $\chi < \Delta(G) + 1$.

correction

(Faire un dessin.)

On considère G à 3 sommets «alignés». Ce graphe est 2-coloriable, et $\Delta(G) + 1 = 3$.

5. Donner un exemple de graphe G tel que $\chi = \Delta(G) + 1$ (avec $\chi = n \in \mathbb{N}^*$ quelconque).

correction

C'est le cas pour un graphe complet à n sommets. On a $\chi = n$ (chaque sommet doit être de couleur différente), et le degré de chaque sommet est $n-1$, donc $\Delta(G) + 1 = n$.

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.

correction

On considère un cycle eulérien – qu'on oriente arbitrairement. Si un sommet s y apparaît n fois, il a $2n$ arêtes incidentes (une avant chaque occurrence, et une après). Son degré est donc 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).

correction

Par contraposée, on considère un chemin eulérien $s = s_0, \dots, s_n$, où $s_n \neq s$.

Notons p le nombre d'occurrences de s_n dans ce chemin – autre que la dernière. Alors le chemin emprunte $2p + 1$ arêtes incidentes à s_n .

s_n étant de degré pair, il existe une arête incidente à s_n qui ne soit pas dans le chemin, par laquelle on peut le prolonger : le chemin n'est donc pas maximal.

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' .

correction

Soit s le sommet commun. Quitte à effectuer une permutation circulaire, on a $C = (s, s_1, \dots, s_n, s)$ et $C' = (s, s'_1, \dots, s'_m, s)$.

Alors $C'' = (s, s_1, \dots, s_n, s, s'_1, \dots, s'_m, s)$ est un cycle répondant à la question.

4. Prouver la réciproque du résultat de la question 1.

correction

On fixe un sommet s arbitraire. Par la question 2, il existe un cycle hamiltonien partiel C passant par ce sommet.

Si c'est un cycle hamiltonien, le résultat est prouvé.

Sinon, il existe une arête qui n'est pas dans C – que l'on peut choisir incidente à un sommet $s' \in C$ par connexité du graphe.

On applique alors la construction d'un cycle hamiltonien partiel au graphe G privé des arêtes de C en partant

de s' (quitte à se restreindre à la composante connexe de s' dans ce graphe). On obtient un cycle C' . On construit C'' comme à la question précédente, et on a donc $|C''| > |C|$. On poursuit alors cette construction en remplaçant C par C'' , jusqu'à ce que le cycle obtenu soit hamiltonien (il y a un nombre fini d'étapes car le nombre d'arêtes du cycle construit augmente strictement à chaque étape, et car $|A|$ est fini).

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?

correction

On effectue un parcours en profondeur en se souvenant du chemin c actuel dans l'arborescence du parcours (i.e. de la pile d'appel). Il y a un cycle si et seulement si au cours de l'exploration, un sommet s a un voisin s' déjà visité dans le chemin qui n'est pas son parent. On obtient alors ce cycle en concaténant l'arête (s, s') au chemin de s' à s (qui est une partie de c).
La complexité est donc $O(|S| + |A|)$

2. Donner un algorithme qui trouve un plus court cycle dans ce graphe.
Quelle est sa complexité?

correction

On applique la méthode ci-dessus, mais utilisant un parcours en largeur, et en le lançant successivement depuis chaque sommet.
La complexité est donc $O(|S|(|S| + |A|))$

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.

correction

Pour chaque arc $a = (s, s')$, on applique l'algorithme de Dijkstra pour trouver le plus court chemin de s' à s sur le graphe G dans lequel on a supprimé l'arc a .
La complexité est donc $O(|A|(|S| + |A|) \log(|S|))$.

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?

correction

On utilise une approche par programmation dynamique (proche de Bellman-Ford) à l'aide des formules suivantes :

- $d_0(s_0) = 0$, $d_0(v) = +\infty$ pour tout $v \neq s_0$
- $d_{k+1}(v) = \min_{(u,v) \in A} (p(u,v) + d_k(u))$.

La complexité est alors $O(|S| \cdot |A|)$

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 ℓ 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 ℓ appartiennent au même sous-ensemble si et seulement s'ils ont le même chef.

correction

Corrigé proposé par le jury

On se place dans le graphe dont les sommets sont les entiers de 1 à n et les arêtes sont les couples (k, p_k) . Par hypothèse, tout sommet y est de degré sortant 1, et les seuls cycles du graphe sont des boucles, c'est-à-dire des cycles de longueur 1. Puisque le chemin obtenu en partant d'un entier m et en remontant la liste de ses ancêtres se termine par un cycle, il contient nécessairement une boucle, centrée sur un chef de m .

En particulier, ce chef est un ancêtre de tous les ancêtres de m , donc m ne peut pas avoir deux chefs, qui seraient ancêtres l'un de l'autre. En outre, deux entiers k et ℓ ont un ancêtre commun si et seulement s'ils ont le même chef, qui sera le chef de cet ancêtre commun.

Cette question a pour but de mettre le candidat en confiance et de lui permettre de s'appropriier la représentation de l'algorithme Union-Find qu'il devra manipuler au cours de l'épreuve. Elle vise également à évaluer sa capacité à introduire des graphes de manière pertinente, puis à raisonner rigoureusement sur ces graphes.

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, ℓ) : on se demande si k et ℓ 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, ℓ) : on souhaite réunir les sous-ensembles auxquels appartiennent k et ℓ . 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 ℓ' ;
 - ▷ si $w_{\ell'} > w_{k'}$, l'entier ℓ' devient le nouveau parent de k' ;

▷ si $k' \neq \ell'$ et $w_{k'} = w_{\ell'}$, l'entier k' devient le nouveau parent de ℓ' 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 $k \leq 4$.

correction

Voici un tableau où l'on a recensé, après chaque étape de l'algorithme, le parent et le poids de chaque nœud ; la situation initiale est indiquée à l'étape $\ll 0 \gg$.

Étape	p ₁	w ₁	p ₂	w ₂	p ₃	w ₃	p ₄	w ₄
0	1	0	2	0	3	0	4	0
1	1	1	1	0	3	0	4	0
2	1	1	1	0	3	1	3	0
3	1	2	1	0	1	1	3	0
4	1	1	1	0	1	1	1	0

La seule difficulté est ici de ne pas oublier que la requête **Test** (2, 4) entraîne une compression du chemin liant le sommet 4 à son chef 1, qui devient désormais son parent.

Cette question vise donc à s'assurer que les élèves ont bien retenu que compresser les chemins dans l'algorithme **Union-Find** était indispensable ou, le cas échéant, à le leur rappeler.

Question 3. On note w_k^∞ 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 .

correction

Au cours de l'algorithme, et tant qu'il est chef, un entier peut d'abord voir son poids augmenter (de 1 à chaque fois), au cours de fusions à l'occasion desquelles on lui assigne de nouveaux descendants ; puis, si un jour il cesse d'être chef, son poids ne changera plus jamais. Par conséquent, si un entier v est un jour le parent d'un entier u , soit w_v et w_u leurs poids à ce moment-là. Puisque u n'est plus chef, il ne le sera plus jamais, donc $w_u^\infty = w_u$; par ailleurs, le poids de v ne pourra qu'augmenter, donc $w_v^\infty \geq w_v > w_u = w_u^\infty$.

Cette question est la première question délicate du sujet. Elle a pour but de vérifier si le candidat maîtrise la dynamique des liens de parenté et des poids au cours de l'algorithme, et s'il est capable de jouer à la fois sur l'état du système à un moment de l'exécution de l'algorithme et une fois cette exécution achevée. Par ailleurs, cette question et chacune des questions suivantes a pour but de préparer les questions ultérieures, notamment le résultat final que l'on souhaite démontrer.

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 $w_k^\infty = \ell$.

correction

On note Γ le graphe étiqueté dont les sommets sont les entiers de 1 à n et les arêtes sont les couples (u, v) pour lesquels $u \neq v$ et v a été parent de u au cours de l'algorithme ; chaque sommet k est étiqueté par l'entier w_k^∞ . On dit alors que v est un Γ -descendant de u (qui est un Γ -ancêtre de v) si Γ contient un chemin allant de u à v .

Chaque entier k gagne ses ancêtres un par un, au cours de requêtes **Union** ; chaque nouvel ancêtre de k devenant un ancêtre de tous les anciens ancêtres de k . Par conséquent, Γ contient un chemin partant de k et passant par tous ses Γ -ancêtres, dont les poids ne font qu'augmenter le long du chemin. En particulier, deux sommets de même étiquette n'ont aucun Γ -descendant commun.

Or, une récurrence sur w indique que, à tout moment au cours de l'algorithme, chaque sommet de poids $w \geq 0$ possède au moins 2^w descendants. Les s entiers k pour lesquels $w_k^\infty = \ell$, ces s entiers ont donc, à eux tous, un total d'au moins $2^\ell s$ ou plus Γ -descendants.

En outre, tout sommet qui Γ -descend d'un entier de poids non nul a dû être un des arguments d'une requête **Union**. Il y a eu au plus m requêtes **Union**, donc au plus $2m$ entiers utilisés comme arguments de telles requêtes. On en conclut que $s \geq 2m/2^\ell$.

Dans la continuité de la question précédente, cette question a pour but d'affiner la compréhension de la dynamique du système étudié par le candidat. Sa difficulté principale consiste à introduire une structure de données simple mais qui contiendra des informations valides à différentes phases de l'algorithme. Elle vise également à s'assurer que le candidat est à même de naviguer entre des échelles locale et globale, ce genre d'approches étant crucial en informatique.

Question 5. Démontrer que toute requête **Test** ou **Union** effectuée au cours de l'algorithme a une complexité majorée par $\mathcal{O}(\log_2(\min\{m, n\}))$.

correction

En question précédente, les s entiers k pour lesquels $w_k^\infty = \ell$ ont cumulé au plus $\min\{n, 2m\}$ Γ -descendants, ce qui nous fournit l'inégalité $s \leq \min\{n, 2m\}/2^\ell$, légèrement meilleure que l'inégalité précédente. En particulier, le plus grand des entiers w_k^∞ est un entier ℓ pour lequel $1 \leq s \leq \min\{n, 2m\}/2^\ell$, de sorte que $\ell \leq \log_2(\min\{m, n\}) + 1$. En outre, toute requête portant sur deux entiers dont les chefs sont de poids w et w' a une complexité $\mathcal{O}(w + w')$. On conclut en remarquant que w et w' sont majorés par $\mathcal{O}(\ell) \subseteq \mathcal{O}(\log_2(\min\{m, n\}))$.

Cette question a pour but de vérifier si le candidat a le réflexe d'affiner de lui-même la borne qui lui avait été proposée en question précédente, et s'il est capable d'utiliser des arguments simples pour trouver des bornes supérieures sur la complexité de telle opération.

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 $\mathcal{O}(m + |E|)$.

correction

Considérons une requête **Union** ou **Test** d'arguments a et b , et soit ω_a et ω_b le nombre de liens de parentés que cette requête remonte en partant de a et de b . La complexité de cette requête est majorée par $\mathcal{O}(\omega_a + \omega_b + 1)$, et celle-ci résulte en l'introduction d'au moins $(\omega_a - 1) + (\omega_b - 1)$ arêtes dans E : il s'agit des liens de parentés détruits par notre requête, et ceux-ci ne seront jamais recréés. Ainsi, si notre requête résulte en l'ajout de e arêtes dans E , sa complexité est majorée par $\mathcal{O}(e + 1)$. Puisque l'on a m requêtes en tout et que la somme des entiers e ainsi définis est égale à $|E|$, le résultat désiré s'ensuit.

De nouveau, cette question a pour but de vérifier que le candidat est capable d'estimer la complexité d'un algorithme en fonction de paramètres auxquels il n'est pas habitué.

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_ℓ 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_ℓ , et au plus $2m$ arêtes allant d'un sommet de V_ℓ à un sommet en dehors de V_ℓ .

correction

Soit u un élément de V_ℓ . Les sommets accessibles par une arête de \mathcal{G} partant de u sont des Γ -ancêtres de u . Ils sont donc d'étiquettes différentes, et au plus $a_{\ell+1} - a_\ell$ d'entre eux appartiennent à V_ℓ . Or, le nombre de sommets de Γ d'étiquette w est un entier $s_w \leq m/2^{w-1}$. Le nombre total d'arêtes internes de \mathcal{G} internes à V_ℓ est donc majoré par

$$a_{\ell+1} |V_\ell| \leq a_{\ell+1} \sum_{w \geq a_\ell} m/2^{w-1} = a_{\ell+1} m/2^{a_\ell-2} = 4m$$

Par ailleurs, chaque requête d'arguments a et b résulte en l'ajout dans \mathcal{G} d'au plus deux arêtes quittant V_ℓ : il s'agit, sur chacun des deux chemins allant de a et b à leurs chefs et que l'on explore avec cette requête, de la dernière arête partant d'un sommet de V_ℓ . Par conséquent, au plus $2m$ arêtes sortent de V_ℓ .

Il s'agit là de la question la plus difficile du sujet, censée résister même aux élèves les plus brillants.

Question 8. Que conclure sur la complexité de l'algorithme **Union-Find** ?

correction

Puisque $w_k^\infty \leq \log_2(m) + 1$ pour tout entier k , on sait que $V_\ell = \emptyset$ dès lors que $a_\ell > \log_2(m) + 1$. Or, la fonction \log^* est croissante, et $\log^*(a_\ell) = \ell$ pour tout entier $\ell \geq 1$. Ainsi, lorsque $m \geq 2$ et $\ell > \log^*(m)$, on sait que $\log^*(a_\ell) = \ell > \log^*(m) \geq \log^*(\log_2(m) + 1)$, de sorte que $a_\ell > \log_2(m) + 1$ et $V_\ell = \emptyset$.

En particulier, \mathcal{G} contient au plus $6m$ arêtes issues de chacun des $\lfloor \log^*(m) \rfloor + 1$ ensembles V_ℓ susceptibles d'être non vides, et aucune arête issue des autres ensembles V_ℓ , nécessairement vides. Il contient donc $\mathcal{O}(m \log^*(m))$ arêtes, ce qui conclut.

Cette question vise non seulement à obtenir explicitement le résultat espéré depuis le début du sujet, mais aussi à vérifier que le candidat est à même de prendre du recul sur les calculs de plus en plus compliqués qu'il a dû mener au cours des questions précédentes.

On pourrait s'étonner qu'elle soit substantiellement plus abordable que la question 7. Néanmoins, les candidats auraient tout à fait eu le droit de demander à traiter cette question en admettant la question 7, avant de revenir sur celle-ci ensuite. Dans ce genre de cas, il ne faut pas s'auto-censurer, et ne pas hésiter à proposer à l'examinateur pareille démarche ; ici, la réponse aurait été positive, même s'il faut bien sûr que le candidat soit prêt à renoncer à son idée si jamais l'examinateur lui répond de manière négative.

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 ℓ à la gare $k + \ell$. Quelle gare Bob choisira-t-il d'établir son nid ?

correction

Corrigé proposé par le jury

On se laisse facilement convaincre, par exemple pour des raisons de symétrie, que la meilleure gare est la gare de numéro 4. Une manière simple de démontrer proprement ce résultat consiste à s'intéresser à la somme des durées plutôt qu'à leur moyenne, puis à remarquer que, lorsque $0 \leq \ell \leq 4$, les distances de la gare k aux gares ℓ et $8 - \ell$ sont de somme minimale si et seulement si $\ell \leq k \leq 8 - \ell$. La gare 4 est donc la seule à être optimale vis-à-vis de l'ensemble de ces paires (dont la paire obtenue pour $\ell = 4$ est dégénérée).

Cette question a pour but de mettre le candidat en confiance et de lui permettre de s'appropriier l'énoncé, tout en évaluant sa capacité à être rigoureux dès que c'est nécessaire, par exemple pour justifier que la gare 4 est optimale.

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 ℓ à la gare $\ell + 1$ requiert t_ℓ minutes ; ainsi, aller de la gare ℓ 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.

correction

Le raisonnement invoqué pour la question précédente fonctionne toujours : lorsque $0 \leq \ell \leq (n - 1)/2$, les distances de la gare k aux gares ℓ et $(n - 1) - \ell$ sont de somme minimale si et seulement si $\ell \leq k \leq (n - 1) - \ell$. Par conséquent, les gares optimales sont celles de numéros $\lfloor (n - 1)/2 \rfloor$ et $\lceil (n - 1)/2 \rceil$; elles sont confondues lorsque n est impaire. Cependant, obtenir un tel résultat n'est pas nécessairement pertinent au vu des questions qui suivent et de la contrainte consistant à fournir un algorithme en temps $\mathcal{O}(n)$. Une alternative est d'améliorer l'algorithme naïf consistant à calculer chaque somme de distances en temps $\Theta(n)$, ce qui requiert un temps total $\Theta(n^2)$. Pour ce faire, il suffit de remarquer que la somme, disons s_ℓ , des distances entre chaque gare ℓ et les gares $k \leq \ell$ est soumise à la relation $s_{\ell+1} = s_\ell + (\ell + 1)t_\ell$. Celle-ci permet de calculer les nombres s_ℓ de proche en proche ; on peut ensuite procéder de même pour calculer la somme des distances entre chaque gare ℓ et les gares $k \geq \ell$, ce qui permet de conclure.

Cette question, nettement plus difficile, a pour but de forcer le candidat à être inventif, puis à réagir face aux indications de l'examinateur ; il n'est pas attendu du candidat qu'il résolve cette question en direct et sans indication.

Question 3. Désormais, le réseau prend la forme d'un graphe connexe acyclique. Bob dispose, pour chaque gare k , d'une liste des gares ℓ 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 ℓ .

Donner un algorithme qui permettra à Bob de choisir, en temps $\mathcal{O}(n)$, dans quelle gare installer son nid.

correction

Là encore, les valeurs des durées $d_{k,\ell}$ sont sans importance, et les gares optimales sont les gares « médianes » du réseau arborescent. Pour les identifier, on enracine notre arbre en un nœud arbitraire, puis on calcule récursivement, depuis les feuilles vers la racine, le nombre de descendants de chaque gare, puis la somme des distances de chaque gare à ses descendantes. Plus précisément, si une gare k possède a_k descendantes, dont ℓ enfants de numéros 0 à $\ell - 1$, et si l'on note s_k la somme des durées des trajets de la gare k avec ses descendantes, alors

$$a_k = 1 + a_0 + a_1 + \dots + a_{\ell-1} \text{ et } s_k = a_0 d_{0,k} + a_1 d_{1,k} + \dots + a_{\ell-1} d_{\ell-1,k} + s_0 + s_1 + \dots + s_{\ell-1}$$

Dans une deuxième phase, on calcule depuis la racine vers les feuilles la somme S_k des distances de chaque gare k à l'ensemble des gares du réseau. En effet, si la gare k est le parent de la gare ℓ , une durée totale $S_k - a_\ell d_{k,\ell} - s_\ell$ est nécessaire pour aller de la gare k aux gares qui ne descendent pas de ℓ , et on en déduit que

$$S_\ell = (S_k - a_\ell d_{k,\ell} - s_\ell) + (n - a_\ell) d_{k,\ell} + s_\ell$$

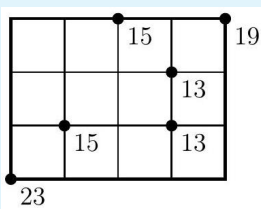
Cette question a pour but de vérifier si le candidat a bien digéré les méthodes qu'il a dû mettre en œuvre à la question précédente et qu'il est à même de développer peu à peu des algorithmes récursifs sur les arbres.

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 ?

correction

Dans un premier temps, on peut se contenter de calculer brutalement la somme des distances L^1 de chaque point aux autres ; ici, on a écrit cette somme à côté de chacun des points concerné. Bob élit donc domicile en l'un des points $(3, 1)$ ou $(3, 2)$.



Cette question a plusieurs buts : préparer le candidat aux questions qui suivent ; vérifier qu'il reconnaît des distances L^1 , dites « de Manhattan » ; s'assurer que, dans le cadre d'un problème de géométrie, il aura le réflexe indispensable de faire un dessin au tableau.

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.

correction

La distance L^1 se décompose naturellement en la somme de deux composantes horizontale et verticale. Or, la démarche utilisée en question 2 nous permet justement, dans un cadre unidimensionnel, de calculer en temps $\mathcal{O}(n)$ la somme des distances de chaque point aux autres, pourvu qu'ils soient triés. On peut donc trier nos n points par x_i croissants, en temps $\mathcal{O}(n \log(n))$, avant de calculer la somme des composantes horizontales des distances de

chaque point aux $n - 1$ autres ; on procède ensuite de même avec les composantes verticales.

Cette question difficile a pour but de vérifier dans quelle mesure le candidat perçoit le fait qu'une distance L^1 est une somme de deux distances unidimensionnelles, puis qu'il est capable d'utiliser un calcul de sommes pour calculer des sommes de combinaisons linéaires.

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 $\mathcal{O}(n)$?

correction

L'enjeu est ici de trier n entiers compris entre 0 et $n^2 - 1$ en temps $\mathcal{O}(n)$. Pour ce faire, on peut s'inspirer du tri par paquets, qui permet de trier n entiers compris entre 0 et $n - 1$ en s'aidant d'un tableau de taille n ; ici, on va écrire nos entiers en base n (sur deux chiffres, donc), puis les trier par unités et ensuite par n -aines.

Plus précisément, on utilise un tableau \mathbf{T} de taille n , contenant n listes chaînées, et on commence par insérer chaque entier k dans la liste $\mathbf{T}[k \bmod n]$. En réunissant ces listes, on a trié les entiers par chiffres des unités. On peut refaire la même opération pour les chiffres des n -aines ; la liste $\mathbf{T}[\ell]$ contient ainsi les entiers k tels que $\ell < k < (\ell + 1)n$, eux-mêmes triés par chiffre des unités. En réunissant ces nouvelles listes, on a trié nos n entiers.

Cette question difficile a plusieurs buts : vérifier que le candidat saisit bien l'enjeu de la question, qui est de trier rapidement des entiers évoluant dans un intervalle entier de taille $n^{\mathcal{O}(1)}$; s'assurer qu'il connaît le tri par paquets ; le pousser à s'interroger sur la notion de tri stable que l'on a utilisée ici de manière sous-jacente, en re-triant des données pré-triées.

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.

correction

On n'utilise plus la distance L^1 , mais une nouvelle distance ad hoc, disons d , que l'on peut obtenir comme suit : si l'on note dx et dy les composantes horizontale et verticale de la distance entre deux points,

$$d = \sqrt{2} \min(dx, dy) + (\max(dx, dy) - \min(dx, dy))$$

Or, si l'on pose $u = x + y$ et $v = x - y$, on constate que

$$du + dv = 2 \max(dx, dy) \text{ et } \min(dx, dy) + \max(dx, dy) = dx + dy$$

On en conclut que

$$d = (s - 1) \left(dx + dy + \frac{du + dv}{\sqrt{2}} \right)$$

Calculer les sommes de chacune des composantes en x, y, u et v des distances entre points, ce que l'on peut faire en temps $\mathcal{O}(n)$ opérations, permet donc de calculer la distance de chaque point au $n - 1$ autres points.

Cette question, conçue pour être à peu près infaisable dans les conditions de l'épreuve, a pour but de distinguer les tous meilleurs candidats. Par conséquent, et contrairement aux précédentes questions difficiles, toute latitude est ici laissée au candidat, que l'examineur pourra remettre sur les rails s'il écrit quelque chose de faux, mais sera livré à lui-même pour trouver les idées permettant la résolution de cette question.

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 Σ un alphabet fini. Le langage \mathcal{L}_1 formé des mots $w \in \Sigma^*$ dont la longueur est paire est-il rationnel ?

correction

Corrigé proposé par le jury

L'ensemble \mathcal{L}_1 est défini par l'expression rationnelle $(\Sigma^2)^*$. Il est donc rationnel.

Cette question a pour but de mettre le candidat en confiance, et de vérifier qu'il est à même de reconnaître des langages rationnels simples.

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 Σ . 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.

correction

Pour tout entier $n \geq 1$ et tout découpage du mot $w = a^{n^2} \in \mathcal{L}_2$ en trois facteurs $p = a^k, q = a^\ell$ et $r = a^{n^2-k-\ell}$ tels que $w = p \cdot q \cdot r$, on sait que $p \cdot q^* \cdot r \not\subseteq \mathcal{L}_2$, par exemple parce que $n + (\ell n)^2$ est compris strictement entre $(\ell n)^2$ et $(\ell n + 1)^2$, de sorte que $p \cdot q^{(\ell n)^2+1} \cdot r \notin \mathcal{L}_2$. L'ensemble \mathcal{L}_2 ne satisfait donc pas le lemme de l'étoile, ce qui lui interdit d'être rationnel.

Cette question a pour but de rassurer le candidat sur sa maîtrise du lemme de l'étoile, et de s'assurer de cette maîtrise. Elle vise également à étudier l'attitude du candidat face à une question dont la réponse n'est pas évidente. Par ailleurs, le fait qu'un candidat n'ait pas l'intuition du résultat, par opposition à un autre qui serait tout de suite allé dans la bonne direction, a été volontairement peu pénalisé ; en effet, il est raisonnable d'imaginer que certains candidats, mais pas tous, auront déjà démontré pendant l'année l'irrationalité du langage \mathcal{L}_2 , qui figure parmi les exemples les plus usuels de langages irrationnels.

Question 3. Démontrer que tout langage rationnel satisfait le lemme de l'étoile.

correction

Soit n le nombre d'états d'un automate fini A reconnaissant \mathcal{L} , puis $w \in \mathcal{L}$ un mot de longueur $|w| = n + k \geq n$, et s_0, \dots, s_{k+n} un chemin acceptant de w dans A . Deux des états s_0, \dots, s_n , disons s_i et s_j , coïncident, de sorte que $w_1 \cdots w_i \cdot (w_{i+1} \cdots w_j)^* \cdot w_{i+1} \cdots w_{k+n} \subseteq \mathcal{L}$.

Il s'agit là quasiment d'une question de cours. Cette question a pour buts de vérifier que le candidat est capable d'utiliser une définition précise d'une notion, susceptible de différer marginalement de l'idée qu'il en avait conçue; qu'il a compris la démonstration du lemme de l'étoile; qu'il est à l'aise tant avec la manipulation de quantificateurs qu'avec le fait de jongler entre états d'un automate, chemins dans cet automate, et mots associés.

L'autre finalité de cette question est d'établir explicitement le fait que, pour qu'un langage \mathcal{L} soit rationnel, il est nécessaire que \mathcal{L} et son complémentaire satisfassent tous deux le lemme de l'étoile; le but du sujet étant précisément de démontrer que ce critère nécessaire n'est pas suffisant.

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 Σ^* . 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 $s_i = s_j$.

correction

Soit $\mathcal{L} \subseteq \Sigma^*$ un langage satisfaisant le lemme de l'étoile, et n l'entier mentionné par le lemme. Pour tout entier $k \geq n$, le mot a^k admet une factorisation $a^k = a^i a^\ell a^j$ pour laquelle $1 \leq \ell \leq j$ et chaque mot $a^{i+z\ell+j} = a^{(k-\ell)+z\ell}$ obtenu lorsque $z \geq 0$ appartient à \mathcal{L} . Puisque ℓ divise $n!$, le langage \mathcal{L} est donc une réunion d'ensembles singletons ou de la forme $a^{x+n!\mathbb{N}}$. Or, une telle réunion est nécessairement finie car, lorsque $x \equiv y \pmod{n!}$, l'un des deux ensembles $a^{x+n!\mathbb{N}}$ et $a^{y+n!\mathbb{N}}$ est inclus dans l'autre. En outre, chacun des ensembles que l'on réunit est rationnel. Le langage \mathcal{L} est donc lui aussi rationnel.

Cette question a pour but d'évaluer la façon dont le candidat appréhende une question qui pourra le déstabiliser, puisqu'il est peu habituel d'utiliser le lemme de l'étoile comme critère suffisant pour être rationnel, ainsi que sa réaction face aux indications de l'examinateur. L'enjeu est ici multiple : il faut comprendre que chaque mot se caractérise par sa longueur, puis se figurer le rôle crucial que jouent ici les réunions finies de progressions arithmétiques, et enfin imaginer comment se ramener à manipuler de telles réunions.

Question 5. Démontrer que tout ensemble $\mathcal{M}(\mathcal{S})$ est rationnel.

correction

L'ensemble $\mathcal{M}(\mathcal{S})$ est défini par l'expression rationnelle $\mathcal{S}^* \cdot \bigcup_{s \in \mathcal{S}} (s \cdot (\varepsilon + \mathcal{S}) \cdot s) \cdot \mathcal{S}^*$. Il est donc rationnel.

Cette question, beaucoup plus simple que la précédente et qui rappelle la question 1, a pour buts d'aider le candidat à s'approprier la définition des ensembles $\mathcal{M}(\mathcal{S})$, ainsi que de s'assurer qu'il est à l'aise avec la représentation d'ensembles par des expressions rationnelles.

Question 6. Démontrer que, si $|\mathcal{S}| \leq 4$, le langage $\mathcal{M}(\mathcal{S})$ est \mathcal{S}^* -étoilé.

correction

Soit m la longueur maximale des mots de \mathcal{S} , et $w = s_1 \cdot s_2 \cdots s_k$ un mot de \mathcal{S}^* de longueur au moins $5m$. Puisque $k \geq 5$, on considère un préfixe $x = s_1 \cdot s_2 \cdots s_5$ de w , de longueur au plus $5m$, puis un mot $\sigma \in \mathcal{S}$ dont la suite s_1, s_2, s_3, s_4, s_5 contient deux occurrences, disons s_i et s_j .

Si $j = i + 1$ ou $j = i + 2$, on décide de pomper sur s_1 (c'est-à-dire de factoriser w comme $w = \varepsilon \cdot s_1 \cdot (s_2 \cdots s_k)$) si $i \geq 2$, et sur s_5 (c'est-à-dire de factoriser w comme $w = \varepsilon \cdot (s_1 \cdots s_4) \cdot s_5 \cdot (s_6 \cdots s_k)$) si $i = 1$ (donc $j \leq 3$). De la sorte, quelle que soit la puissance à laquelle on élève notre pompe (le facteur s_1 ou s_5), on reste dans \mathcal{S}^* , et les deux facteurs s_i et s_j restent à distance 1 ou 2 de l'autre.

Si $j \geq i + 3$, on décide cette fois de pomper sur $s_{i+1} \cdot s_{i+2}$. En effet, si l'on élève cette pompe à la puissance 0, nos facteurs s_i et s_j se retrouvent à distance $j - i - 2 \leq 2$ l'un de l'autre; si on l'élève à la puissance 1, on obtient le mot w ; si on l'élève à une puissance supérieure ou égale à 2, on récupère un facteur de la forme $s_{i+1} \cdot s_{i+2} \cdot s_{i+1}$. Dans les trois cas, les mots obtenus appartiennent à $\mathcal{M}(\mathcal{S}) \cup \{w\}$.

Cette question manifestement difficile a pour but d'évaluer quelles idées intermédiaires le candidat peut mettre en œuvre, soit parce qu'il en a eu l'intuition, soit parce que l'examinateur les lui a suggérées. Ici, l'idée clé consiste à

décider de ne manipuler que des facteurs s_i , de manière à faire comme si l'on travaillait sur les lettres d'un alphabet de cardinal au plus 4, puis à épuiser les uns après les autres les différents cas possibles, en s'attachant à chaque fois à obtenir deux facteurs s_i proches l'un de l'autre.

Question 7. On considère l'alphabet $\Sigma = \{a, b, c, d\}$. Pour tout entier $k \leq 3$, on note σ_k le mot $(abc)^k \cdot (abd)^{3-k}$. Les langages $\mathcal{L}_3 = \{(\sigma_0 \cdot \sigma_1 \cdot \sigma_2)^\ell \cdot (\sigma_0 \cdot \sigma_1 \cdot \sigma_3)^\ell : \ell \geq 0\}$ et $\mathcal{L} = \mathcal{M}(\{\sigma_0, \sigma_1, \sigma_2, \sigma_3\}) \cup \mathcal{L}_3$ sont-ils rationnels?

correction

Le langage \mathcal{L}_3 ne satisfait pas le lemme de l'étoile. En effet, si ℓ est assez grand, on devra choisir notre mot de pompage t au sein du préfixe $(\sigma_0 \cdot \sigma_1 \cdot \sigma_2)^\ell$, et soit sortir du langage rationnel $(\sigma_0 \cdot \sigma_1 \cdot \sigma_2)^* \cdot (\sigma_0 \cdot \sigma_1 \cdot \sigma_3)^*$, soit augmenter le nombre d'occurrences du facteur σ_2 . Puisque \mathcal{L} est la réunion disjointe du langage rationnel $\mathcal{M}(\mathcal{S})$ et de \mathcal{L}_3 , où l'on a posé $\mathcal{S} = \{\sigma_0, \sigma_1, \sigma_2, \sigma_3\}$, il n'est alors pas rationnel non plus, ce sans quoi le langage $\mathcal{L}_3 = \mathcal{L} \setminus \mathcal{L}(\mathcal{S})$ serait rationnel lui aussi.

Cette question a pour but premier d'évaluer la familiarité du candidat avec les langages de la forme $L_\ell = \{x^\ell y^\ell : \ell \geq 0\}$, connus pour être algébriques mais pas rationnels. Ici, \mathcal{L}_3 est l'image d'un tel langage par un morphisme, et il revient alors d'adapter avec rigueur à \mathcal{L}_3 la preuve d'irrationalité de L_ℓ . L'étude du langage \mathcal{L} est ensuite censée être une simple formalité, qui vise simplement à s'assurer de la rigueur du candidat, amené à vérifier que \mathcal{L}_3 s'exprime de manière simple à partir de deux langages dont l'un est déjà connu pour être rationnel.

Par ailleurs, le caractère ouvert de la question est de toute évidence factice, au vu de la question 8 et de l'ambition que l'on a mise en avant en tout début de sujet. Cette question a donc aussi pour but secondaire de s'assurer que le candidat est à même de prendre du recul sur son sujet et d'avoir saisi la progression des questions qu'il est amené à traiter.

Question 8. Démontrer que les ensembles \mathcal{L} et $\Sigma^* \setminus \mathcal{L}$ satisfont le lemme de l'étoile.

correction

Puisque \mathcal{L} contient $\mathcal{M}(\mathcal{S})$, qui est \mathcal{S}^* -étoilé, \mathcal{L} est également \mathcal{S}^* -étoilé, et il est même \mathcal{L}' -étoilé pour tout langage \mathcal{L}' inclus dans \mathcal{S}^* , par exemple $\mathcal{L}' = \mathcal{L}$ lui-même. Par ailleurs, \mathcal{S}^* ne contient aucun mot de $\mathcal{M}(\Sigma)$, donc $\Sigma^* \setminus \mathcal{L}$ contient $\mathcal{M}(\Sigma)$, qui est Σ^* -étoilé. Pour les mêmes raisons que précédemment, $\Sigma^* \setminus \mathcal{L}$ est donc Σ^* -étoilé, et même $\Sigma^* \setminus \mathcal{L}$ -étoilé. Ainsi, \mathcal{L} et $\Sigma^* \setminus \mathcal{L}$ satisfont tous deux le lemme de l'étoile.

Cette question assez difficile a pour but d'évaluer dans quelle mesure le candidat a compris les différents types de raisonnement mis en œuvre lors des questions précédentes. Les deux idées principales sont ici de comprendre les liens entre caractère \mathcal{L}' -étoilé et inclusion, ainsi que d'observer que \mathcal{S}^* ne contient aucun mot de $\mathcal{M}(\Sigma)$. S'agissant de la dernière question du sujet, aucune indication n'était ici fournie aux candidats, qui devaient donc obtenir par eux-mêmes ces idées.

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.

correction

Corrigé de M.Péchaud

Oui : on considère l'automate à deux états, le premier étant acceptant et terminal où

- chaque état a une boucle étiquetée par 0
 - de chaque état part une transition vers l'autre étiquetée par 1.
- On vérifie que cet automate reconnaît bien le langage voulu.

- (b) Les mots de la forme $0^k 1^\ell$, où $k \wedge \ell = 1$.

correction

NDMP : à voir : je vois des arguments par des langages résiduels – donc plutôt hors programme – rien de plus au programme pour l'instant.

2. Soit Σ un alphabet. Pour un mot $w \in \Sigma^*$, on définit Pompe(w) comme le plus petit langage sur Σ 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 Pompe(ab), où $\Sigma = \{a, b\}$?

correction

C'est $a\Sigma^*b$:

- $a\Sigma^*b$ vérifie clairement la propriété voulue.
- Montrons que c'est le plus petit langage les vérifiant. Soit L un langage vérifiant la propriété.

Soit $u \in a\Sigma^*b$.

On note u' le mot obtenu à partir de u en remplaçant toutes les séquences de a par un seul a , et de même pour les b .

u' est donc de la forme $u = (ab)^n \in L$ (en itérant $n - 1$ fois le facteur $y = ab$).

On itère alors chacune des lettres pour reconstituer u . Donc $u \in L$, ce qui prouve bien que $a\Sigma^*b \subset L$.

3. On définit Cycle(L) = $\{vu \mid uv \in L\}$. Si L est rationnel, Cycle(L) l'est-il? Que pensez vous de la réciproque?

correction

Le résultat est vrai. Quitte à traiter séparément le cas de ϵ , on considère que ce mot n'est pas dans L .

On considère un AFD standardisé \mathcal{A} reconnaissant L . Pour chaque état q , on crée un automate obtenu en

- créant deux copies de q : q' et q'' qui sont respectivement le seul état initial et le seul état acceptant du nouvel automate ;
- on fusionne l'état initial et l'état acceptant de l'automate de départ.

Cet automate reconnaît l'ensemble des mots vu tels que $uv \in L$, et la lecture de u amène dans \mathcal{A} dans l'état q . L'union (finie) de ces langages pour tous les états q (autres que l'état initial ou acceptant) de \mathcal{A} est le langage recherché, qui est donc rationnel.

La réciproque est fautive : considérer $\{a^n b a^n, n \in \mathbb{N}\}$, qui n'est pas rationnel, alors que $\text{Cycle}(L)$ est l'ensemble des mots contenant un seul b et un nombre pair de a est rationnel.

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 ?

correction

Soient $\mathcal{A} = (Q, q_0, F, \Delta)$ et $\mathcal{A}' = (Q', q'_0, F', \Delta')$ deux AFD reconnaissant respectivement L et L' .

On définit un nouvel automate $\mathcal{A}'' = (Q'', (q_0, q'_0), F \times F', \Delta'')$

• $\mathcal{A}'' = \mathcal{A} \times \mathcal{A}'$

• $(p, p') \rightarrow (q, q') \in \Delta''$ ssi $(p = q \text{ et } p' \rightarrow q' \in \Delta')$ ou $(p \rightarrow q \in \Delta \text{ et } p' = q')$

l'idée étant qu'en lisant une lettre, on avance dans l'un des automates de départ, tout en ne bougeant pas dans l'autre.

Le langage reconnu par \mathcal{A}'' est $L \wr L'$, qui est donc bien rationnel.

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_∞ est-il rationnel ?

correction

Non. Soit $L = \{a, b\}$. On montre par récurrence immédiate sur n que tout mot de L_n a autant de a que de b ($n - 1$). C'est donc vrai aussi pour les mots de L_∞ .

On montre également facilement que $a^n b^n \in L_\infty$ pour tout $n \in \mathbb{N}$.

Donc $L_\infty \cap a^* b^* = \{a^n b^n, n \in \mathbb{N}^*\}$, qui n'est pas rationnel.

Donc par contraposée, L_∞ non plus.

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 .

correction

On remarque qu'il suffit de montrer le résultat pour les sous-ensembles linéaires, par stabilité par union des langages rationnels.

On considère donc $u_0 + u_1 \mathbb{N} + \dots + u_k \mathbb{N}$.

On remarque que si L et L' sont deux langages, alors, $P(L.L') = \{v + v', v \in P(L), v' \in P(L')\} = P(L) + P(L')$ (essentiellement car $|L.L'|_a = |L|_a + |L'|_a$).

Donc il suffit de montrer le résultat sur chacun des termes – par stabilité par concaténation des langages réguliers.

• Il existe un langage L tel que $P(L) = u_0 = (n_1, \dots, n_k)$ (par exemple le langage réduit au mot $a_1^{n_1} \dots a_k^{n_k}$).

- Il existe un langage L tel que $P(L) = u_1\mathbb{N}$ où $u_1 = (n_1, \dots, n_k)$ (par exemple $L = (a_1^{n_1} \dots a_k^{n_k})^*$), et de même pour u_2, \dots, u_k .
On a donc bien le résultat voulu.

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 ?

correction

Corrigé proposé par le jury

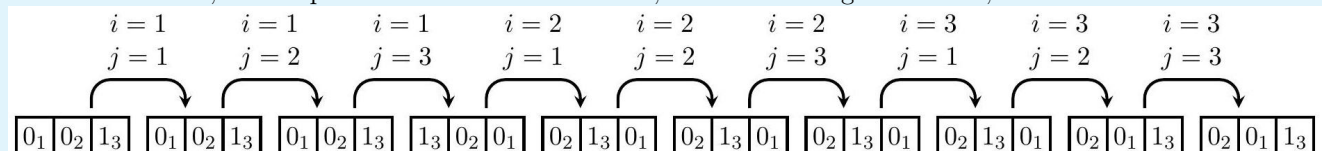
On a deux boucles imbriquées, chacune faisant prendre à sa variable (i ou j) n valeurs distinctes. À chaque fois, on effectue une comparaison. Cela fait donc n^2 comparaisons en tout, et ce quel que soit le tableau proposé en entrée.

Cette question a pour but de mettre le candidat en confiance et de lui donner une première occasion de s'appropriier les notations : A_i désigne, au moment où est effectuée la comparaison $A_i < A_j$, le $i^{\text{ème}}$ élément du tableau. Elle permet aussi de vérifier que le candidat ne va pas annoncer sans avoir réfléchi au préalable que «le meilleur cas est celui où A est trié», phrase certes vraie mais qui suggère une incompréhension.

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 ?

correction

Imaginons un tableau à trois éléments, dont deux sont égaux pour notre fonction de comparaison : il s'agit, par exemple, du tableau $0_1, 0_2, 1_3$, où l'on a indiqué à côté de chaque objet sa position initiale. Au cours de l'algorithme, en neuf occasions, on compare deux cases du tableau A , et on les échange au besoin, comme suit :



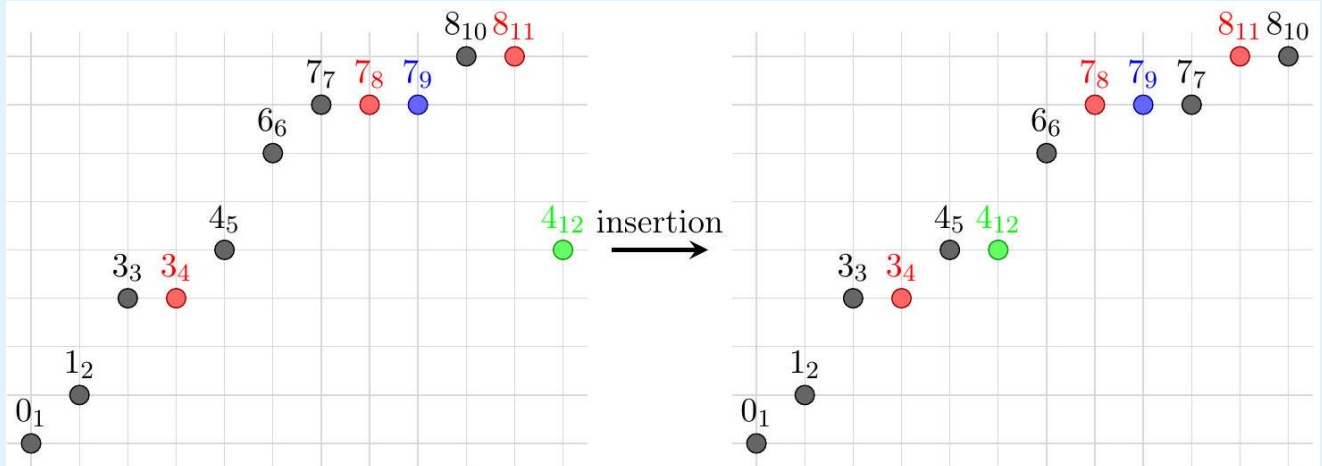
L'élément 0_2 , initialement en deuxième position, est passé à gauche de l'élément 0_1 , qui lui est égal pour notre fonction de comparaison mais était initialement situé en première position. Le TriLent n'est donc pas stable.¹ Cette question a trois objectifs : tout d'abord, aider le candidat à se familiariser avec l'algorithme qu'ils devra ensuite étudier ; ensuite, s'assurer qu'il est capable d'exécuter calmement mais précisément un algorithme sur une entrée simple qu'il aura lui-même choisie ; enfin, vérifier qu'il est capable de représenter clairement les objets qu'il manipule, tels que des quantités censées être égales pour notre fonction de comparaison mais distinctes en pratique.

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.

correction

Lors du premier passage dans la boucle de la ligne 2, alors que i vaut 1, on compare successivement A_1 avec toutes les cases du tableau, de manière à faire croître A_1 dès que l'on rencontre un nouveau maximum. Après ce premier passage, lors duquel on a effectué une fois la boucle pour des lignes 3 et 4, l'objet A_1 est bien maximal, et le tableau A_1, \dots, A_1 , formé d'un seul élément, est nécessairement trié dans l'ordre croissant. On dira qu'il s'agit là de la phase 1 de l'algorithme ; toutes les opérations effectuées ensuite formeront la phase 2.

Par la suite, après que l'on a effectué i fois cette boucle, l'algorithme consiste à comparer chaque case du tableau avec la $i + 1^{\text{ème}}$ case. Ainsi, lorsque j varie de 1 à $i + 1$, on effectue en fait l'insertion de la valeur A_{i+1} dans le sous-tableau trié A_1, A_2, \dots, A_i : dès que l'on repère une case du tableau, disons A_j , telle que $A_j > A_{i+1}$, on échange A_j et A_{i+1} , de sorte que la $i + 1^{\text{ème}}$ case nous sert en fait de mémoire tampon pour ensuite décaler d'une case vers la droite le sous-tableau A_j, A_{j+1}, \dots, A_i . Ces cases seront décalées dans leur ensemble, mais pas forcément individuellement, dans le cas où le tableau contient des doublons : quand plusieurs cases consécutives étaient égales et dépassaient la valeur à insérer, c'est la plus à gauche d'entre elles qui est passée à droite, comme illustré ci-dessous ; chaque objet est représenté par sa valeur et, en indice, sa position d'origine.



Une fois cette étape franchie, A_{i+1} est la valeur maximale d'un sous-tableau trié qui contenait le maximum de A , donc on ne l'échangera plus jamais lorsque $j \geq i + 2$. Par conséquent, une fois notre $i + 1$ ème exécution de la boucle pour effectuée, le sous-tableau A_1, A_2, \dots, A_{i+1} est bien trié dans l'ordre croissant, et se termine avec la valeur maximale du tableau A .

Cette question vise à affiner la compréhension qu'a le candidat de l'algorithme de TriLent et de vérifier comment il entend de se forger des intuitions qui mèneront à cette compréhension, en particulier la disjonction entre phases 1 et 2.

À défaut de la description donnée ci-dessus, il était aussi possible d'exhiber directement puis de démontrer un invariant de boucle, ce qui pouvait néanmoins s'avérer piégeux : on a tôt fait de proposer un invariant presque correct mais qui sera difficile à corriger, ou bien de répondre à la question mais sans avoir acquis de compréhension globale sur le fonctionnement de l'algorithme.

1. On aurait pu aller plus vite en s'arrêtant sitôt le tableau $0_2, 1_3, 0_1$ obtenu, après avoir comparé A_i et A_j lorsque $(i, j) = (2, 1)$, car alors $A_1 = 0_2$ ne sera plus jamais déplacé.

Question 4. Démontrer que le TriLent permet effectivement de trier le tableau fourni en entrée.

correction

Il suffit d'appliquer la question précédente au cas où $i = n$: une fois l'algorithme terminé, le tableau est trié. Cette question a pour seuls buts d'aboutir à une conclusion intermédiaire raisonnable, ici la correction de l'algorithme de TriLent, et de redonner un coup de fouet aux candidats, tout heureux d'avoir résolu une question en cent fois moins de temps que la précédente.

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.

correction

Sans perte de généralité, le tableau à trier est une permutation des entiers $1, 2, \dots, n$. Juste après que la boucle pour des lignes 3 et 4 a été exécutée i fois, l'entier n se trouve donc en case i du tableau. En particulier, il a changé au moins $n - 1$ fois de case.

Réciproquement, si l'on est parti du tableau $n, 1, 2, \dots, n - 1$, notre étude effectuée en question 3 indique qu'aucun échange n'est effectué lors de la phase 1, tandis que tous les échanges ultérieurs ont lieu juste après que l'on a comparé A_{i-1} et A_i . Ainsi, l'algorithme de TriLent s'est effectivement contenté de $n - 1$ échanges pour trier notre tableau.

Il s'agit là de la première question manifestement ouverte, pour laquelle le candidat n'a, a priori, aucune idée de la réponse, si ce n'est qu'elle sera comprise entre 0 et n^2 . Y répondre nécessite non seulement d'avoir compris comment l'élément maximal du tableau se déplacerait, mais aussi de regarder des exemples de tableaux susceptibles d'être simples, tels que le tableau déjà trié dans l'ordre croissant. De manière peut-être surprenante, celui-ci n'est pas le tableau optimal, car on effectue $n - 1$ échanges inutiles lors de la phase 1, même si la phase 2 est ensuite très économe. C'est alors que le candidat doit être amené à remplacer le tableau trié par le tableau obtenu à l'issue de la phase 1.

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.

correction

Il convient ici de considérer séparément les phases 1 et 2, puis de trouver un invariant qui nous permettra d'estimer le nombre d'échanges effectués ; on va en fait démontrer que le nombre d'inversions du tableau A , c'est-à-dire le nombre de couples (k, ℓ) tels que $k < \ell$ et $A_k > A_\ell$, augmente de 1 à chaque échange de la phase 1, et diminue de 1 à chaque échange de la phase 2.

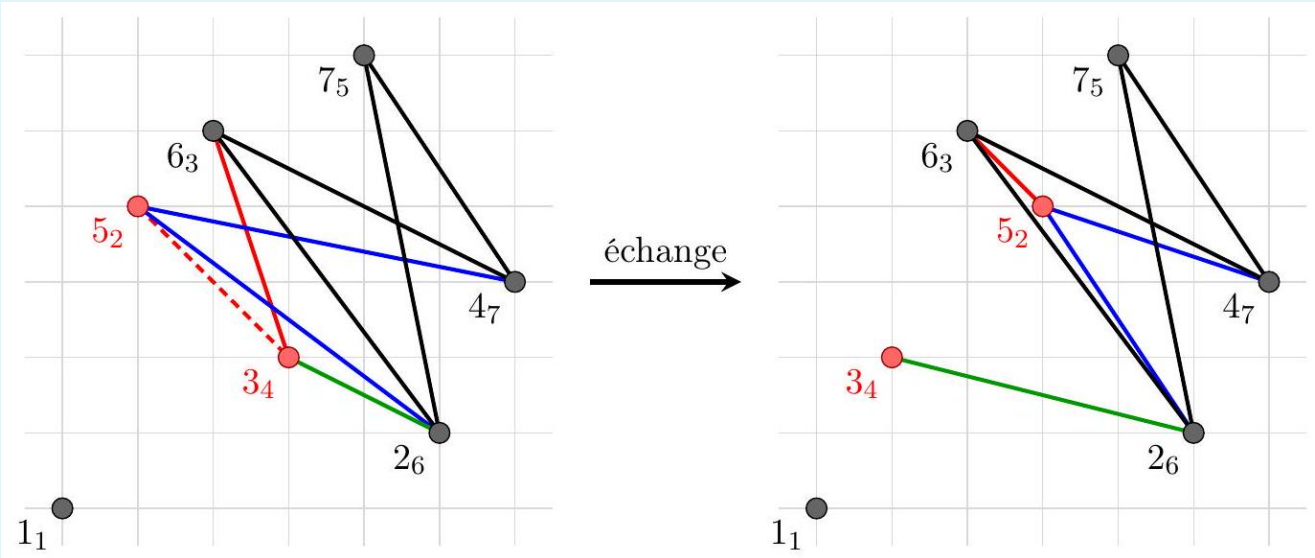
Regardons, par exemple, ce qui se passe en phase 2. Lorsque l'on échange deux cases A_i et A_j et que $i \geq 2$, on insère en fait la case A_i au sein du sous-tableau trié A_1, A_2, \dots, A_{i-1} . Par conséquent, si l'on note A le tableau avant l'échange et A' le tableau après l'échange, les inversions de A' sont :

- ▷ les couples (u, i) tels que $j < u < i$; il y en a a ;
- ▷ les couples (i, v) tels que $i < v$ et $A_j = A'_i > A'_v = A_v$; il y en a b ;
- ▷ les couples (j, v) tels que $i < v$ et $A_i = A'_j > A'_v = A_v$; il y en a c ;
- ▷ les couples (u, v) tels que $\{u, v\} \cap \{i, j\} = \emptyset, u < v$ et $A_u > A_v$; il y en a d .

Par ailleurs, les inversions de A sont :

- ▷ les couples (u, i) tels que $j \leq u < i$; il y en a $a + 1$;
- ▷ les couples (i, v) tels que $i < v$ et $A_i > A_v$; il y en a c ;
- ▷ les couples (j, v) tels que $i < v$ et $A_j > A_v$; il y en a b ;
- ▷ les couples (u, v) tels que $\{u, v\} \cap \{i, j\} = \emptyset, u < v$ et $A_u > A_v$; il y en a d .

Comme annoncé, échanger A_i et A_j a donc permis de baisser de 1 le nombre d'inversions du tableau considéré ; cette situation est illustrée ci-après : les deux points échangés sont dessinés en rouge, et l'inversion qu'ils formaient, représentée par une ligne rouge hachurée, a disparu. Les autres inversions ont subsisté, parfois en changeant de point d'accroche ; les quatre différents types d'inversions énumérés ci-dessus (il y en a respectivement a ou $a + 1, b, c$ et d) sont représentés en rouge, bleu, vert et noir ; chaque objet est représenté par sa valeur et, en indice, sa position d'origine.



Or, le nombre moyen d'inversions d'une permutation choisie uniformément au hasard est $n(n - 1)/4$. En effet, pour toute paire (i, j) telle que $1 \leq i < j \leq n$, si l'on note $\sigma_{i,j}$ la transposition qui échange i et j , on remarque que la fonction $\pi \mapsto \pi \circ \sigma_{i,j}$ est une involution de \mathfrak{S}_n , qui partitionne \mathfrak{S}_n en orbites de taille 2. De surcroît, $\pi(i) < \pi(j)$ si et seulement si $\pi(\sigma_{i,j}(i)) > \pi(\sigma_{i,j}(j))$. Ainsi, le couple (i, j) est une inversion pour une des deux permutations de chaque orbite $\{\pi, \pi \circ \sigma_{i,j}\}$: au total, les $n!$ permutations ont donc $n! \times \binom{n}{2} / 2 = n! \times n(n - 1)/4$ inversions, ce qui nous fait bien une moyenne de $n(n - 1)/4$ inversions par permutation.

On étudie ensuite la phase 1 ; notre invariant se montre en utilisant un raisonnement analogue à celui utilisé en phase 2 : si on annulait notre échange, on supprimerait une inversion, donc l'échange a bien créé une inversion. Ensuite, lorsque $i = 1$ et $2 \leq j \leq n$ et que l'on compare A_1 et A_j , on sait déjà que A_1 a été redéfini comme $\max\{\pi(1), \pi(2), \dots, \pi(j - 1)\}$, tandis que $A_j = \pi(j)$. On procède donc à un échange lorsque $\pi(j) = \max\{\pi(1), \pi(2), \dots, \pi(j)\}$, ce qui arrive avec probabilité $1/j$. Au total, lors de la phase 1, on effectue donc $1/2 + 1/3 + \dots + 1/n = H_n - 1$ échanges en moyenne ; on devra ensuite effectuer un échange supplémentaire pour se débarrasser de l'inversion que notre échange vient de créer.

En conclusion, l'algorithme de TriLent requiert bien $n(n - 1)/4 + 2(H_n - 1)$ échanges en moyenne.

Cette question n'est manifestement guère faisable sans indication de la part de l'examinateur. Elle a pour but de tester le candidat sur des questions de nature probabiliste, qui forment un pan important de l'informatique, et de voir, dans un premier temps, comment il tente de se débrouiller face à une question fort ardue en apparence (et en réalité).

En particulier, une fois donnée l'indication, en pratique indispensable, selon laquelle il fallait vérifier que chaque échange ajoutait ou supprimait une inversion du tableau à trier, il fallait ensuite modéliser aussi simplement que possible les différents phénomènes observés. Ici, avoir compris que les phases 1 et 2 différaient radicalement, mais aussi savoir utiliser simplement des bijections et la linéarité de l'espérance était nécessaire pour espérer répondre à la question. De même, pour démontrer la véracité de l'indication donnée par l'examinateur, il était beaucoup plus simple de dessiner la permutation modifiée et d'en représenter graphiquement les inversions susceptibles de disparaître ou d'apparaître, plutôt que de tenter une disjonction de cas sans aucun apport de l'intuition. Procéder ainsi permettait notamment de traiter un seul des cas, disons celui de la suppression d'une inversion, pour ensuite convaincre à l'oral l'examinateur que l'autre cas se traiterait de même.

Question 7. Donner le nombre maximal d'échanges que l'algorithme de TriLent peut être amené à effectuer.

correction

Sans perte de généralité, on peut toujours supposer que le tableau à trier contient des valeurs deux à deux distinctes ; si tel n'était pas le cas, et quitte à séparer artificiellement deux valeurs considérées comme égales, on ne diminuerait pas le nombre d'échanges effectués.

Une fois ce cadre posé, on aimerait bien pouvoir maximiser simultanément le nombre d'échanges effectués lors des phases 1 et 2 ; c'est impossible. En effet, réaliser les $n - 1$ échanges possibles lors de la phase 1 nécessite de partir du tableau trié. Notre idéal étant inatteignable, on note donc k le nombre d'échanges effectués en phase 1 puis on s'intéresse, l'entier k étant fixé, au nombre maximal d'inversions, disons $\text{inv } k$, que notre tableau obtenu en fin de phase 1 a pu avoir : le nombre maximal d'échanges effectués dans l'ensemble des phases 1 et 2 sera $\max\{k + \text{inv } k : 0 \leq k \leq n - 1\}$.

Les k valeurs qu'a prises la case A_1 du tableau au cours de la phase 1 avant que cette case ne prenne la valeur maximale se retrouvent à former une sous-suite croissante au sein du tableau obtenu en fin de phase 1, et que l'on notera B . Si l'on note I l'ensemble des numéros de cases occupés par ces valeurs au sein du tableau B , aucun couple $(i, j) \in I^2$ ne forme une inversion de B . Ainsi, le tableau B compte au plus $n(n - 1)/2 - k(k - 1)/2$ inversions, et l'algorithme de TriLent a effectué au plus

$$\frac{n(n-1)}{2} - \frac{k(k-1)}{2} + k = \frac{n(n-1) - k(k-3)}{2} \leq \frac{n^2 - n + 2}{2}$$

échanges.

Réciproquement, si l'on choisit $k = 1$, on souhaite précisément que B soit le tableau trié dans l'ordre décroissant, que l'on a pu obtenir en partant du tableau $n - 1, n, n - 2, n - 3, \dots, 2, 1$. En conclusion, le nombre maximal d'échanges auquel procède l'algorithme de TriLent est $(n^2 - n + 2)/2$.

Cette question a pour but de résister même aux candidats les plus aguerris ; aucune indication n'était donc prévue pour les guider vers la solution. Elle nécessite tout d'abord de se rendre compte que l'on peut légitimement supposer que l'on travaille sur une permutation, puis d'avoir bien compris que compter les inversions était la bonne manière de compter les échanges effectués par le TriLent, et enfin de renoncer à maximiser simultanément deux quantités liées, pour fixer l'une et maximiser l'autre.

Chapitre 115

(X) Codage par couleur *** (X 24, corrigé — oral - 154 lignes)

Algorithmique, Complexité, Graphes, Backtracking,
sources : `xcodagecouleur.tex`

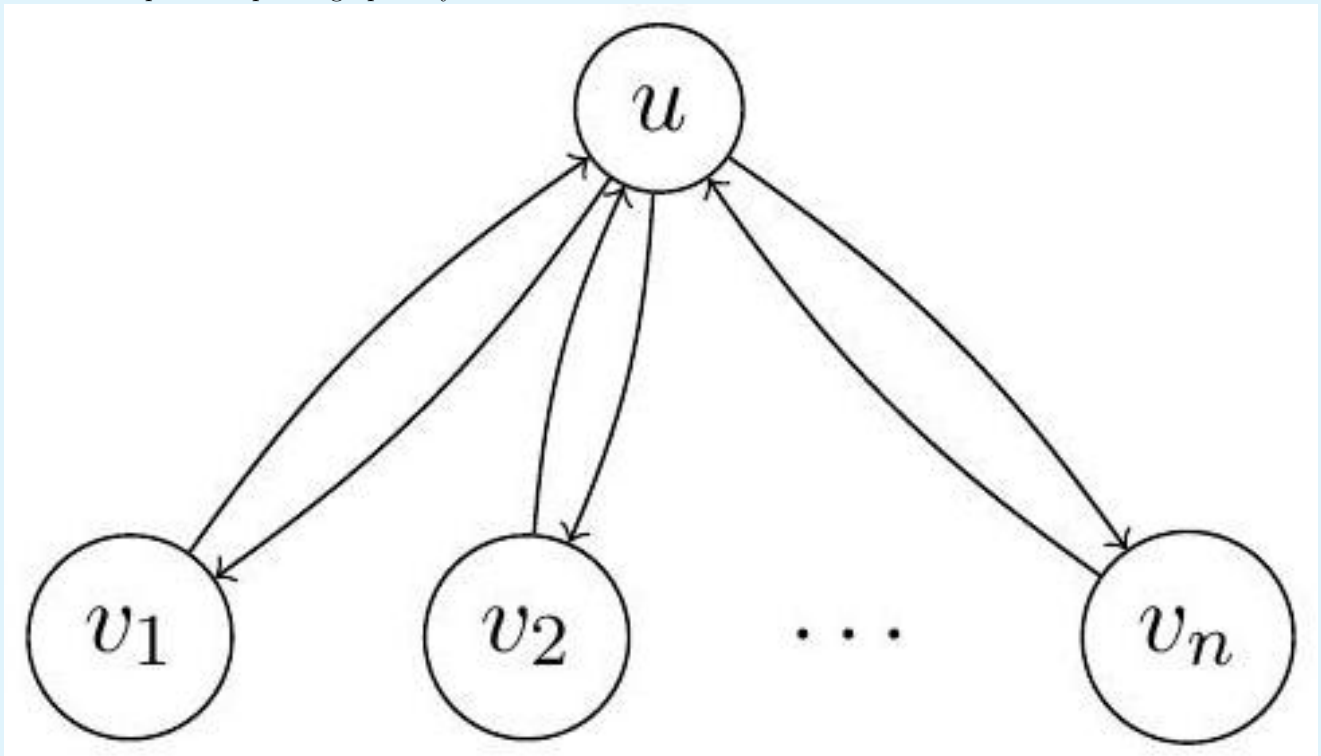
On considère des graphes orientés, supposés sans boucle. On cherchera dans ces graphes un chemin d'une longueur spécifique (sans imposer de point de départ et d'arrivée), où la longueur d'un chemin est le nombre de sommets qui apparaissent dans le chemin. Sauf mention explicite du contraire, on considère 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?

correction

Corrigé proposé par le jury

La réponse à cette question est positive, comme l'ont déterminé tous les candidats (plus ou moins rapidement). C'est le cas par exemple de graphes ayant une forme d'étoile :



Le but de cette question est de rappeler la notion de graphes orientés fortement connexes en lien avec le sujet, et d'attirer l'attention sur la définition peu standard de la longueur des chemins que le sujet adopte (afin de simplifier certains calculs par la suite). La question permet aussi de remarquer que ce qui est étudié dans le sujet ensuite, à

savoir la recherche de chemins, n'est pas une tâche triviale : les chemins tels que définis dans le sujet n'existent pas forcément, même quand le graphe est grand, et même en imposant qu'il soit fortement connexe.

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 ?

correction

La réponse attendue est simplement de dire que l'on teste toutes les combinaisons possibles : pour n sommets et m arêtes, il y a n^k combinaisons de k sommets à tester, pour lesquelles il faut vérifier qu'il y a bien l'arête demandée. Si on suppose que le graphe est représenté par des listes d'adjacence, alors on obtient une complexité de $O(n^k \times m)$. Une autre façon de répondre à cette question est d'utiliser une approche de type retour sur trace, ou bien une exploration en largeur qui marque les sommets en cours de visite et s'interdit de les réutiliser (mais ne marque pas les sommets où la visite est finie, ceux-ci pouvant être visités à nouveau). La complexité asymptotique sera alors similaire, même si en pratique le retour sur trace est bien sûr préférable à l'énumération naïve des combinaisons : en effet il n'explore que les successeurs de chaque sommet, et il abandonne quand une solution partielle ne peut pas être étendue.

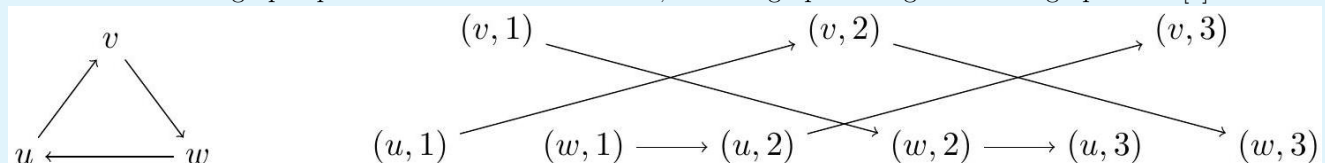
Cette question vise à vérifier que le candidat est capable de proposer rapidement un algorithme naïf pour un problème, ce qui doit être un préalable à toute tentative d'optimisation. En effet, si le graphe d'entrée est très petit, un algorithme naïf sera suffisant et sera bien plus facile à implémenter. Les candidats ne doivent pas hésiter, quand une approche naïve est demandée, à proposer des algorithmes peu efficaces : les examinateurs sauront leur préciser si c'est quelque chose de plus efficace qui est attendu.

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 .

correction

Cette question, qui se voulait facile, s'est avérée en réalité être très classante. L'argument que nous attendions consistait à construire une sorte de produit cartésien. Étant donné le graphe $G = (V, E)$, on crée le graphe $G \times [k]$ dont les sommets sont $V \times \{1, \dots, k\}$ et les arêtes sont $\{(u, i), (v, i+1) \mid (u, v) \in E, 1 \leq i < k\}$. On teste ensuite simplement s'il existe un chemin de la première copie à la dernière. Autrement dit, on cherche à tester s'il y a un chemin d'un sommet de la forme $\{(u, 1) \mid u \in V\}$ à un sommet de la forme $\{(u, k) \mid u \in V\}$. Ceci peut se faire avec un parcours en largeur par exemple. La complexité est en $O((n+m) \times k)$ si on suppose à nouveau que le graphe est représenté avec des listes d'adjacence.

La construction du graphe produit est illustrée ci-dessous, avec un graphe G à gauche et le graphe $G \times [3]$ à droite :



D'autres approches sont possibles en ne construisant le graphe produit qu'implicitement. Enfin, il est possible d'adopter une approche en complexité linéaire (c'est-à-dire $O(n+m)$) en distinguant deux cas : soit le graphe contient un cycle orienté (ce qui peut se vérifier en temps linéaire) et alors il y a toujours un chemin non-simple de longueur k pour n'importe quel k , soit le graphe est acyclique et on peut utiliser l'algorithme pour les graphes acycliques qui est l'objet de la question 9.

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.

correction

Les candidats ont tous compris (spontanément, ou avec indication) qu'un chemin multicolore est nécessairement simple, par application du principe des tiroirs. L'objectif était alors de se rendre compte que, grâce à ce fait, on pouvait utiliser une variante de la question précédente. Là encore, cette question était finalement plus classante que prévu.

Pour répondre à la question, on crée 2^k copies du graphe $G = (V, E)$, c'est-à-dire qu'on considère le graphe $(V \times 2^k, E')$ avec E' défini comme suit : pour chaque arête $(u, v) \in E$, en notant c la couleur de v , pour chaque $S \subseteq 2^k$, on crée $((u, S), (v, S \cup \{c\}))$.

On cherche ensuite à savoir s'il existe un chemin depuis un sommet de la forme $(u, \{c\})$ pour c la couleur de u à un sommet de la forme $(v, 2^k)$. Si un tel chemin existe, c'est forcément un chemin dans G (par projection sur la première composante), et par définition des arêtes il a forcément visité un sommet de chaque couleur. En particulier le chemin est forcément simple. Donc cet algorithme identifie bien correctement si G contient un chemin multicolore.

La complexité est linéaire en la taille du graphe produit c'est-à-dire $O((n+m)2^k)$.

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 .

correction

Pour les candidats qui ont pu l'atteindre, cette question n'a guère posé de difficulté. Il y a n^k tirages possibles et c de ces tirages réussissent. Donc chaque tirage a une probabilité de c/n^k de réussir, et une probabilité de $1 - c/n^k$ d'échouer. On effectue M tirages indépendants, donc avec probabilité $(1 - c/n^k)^M$ aucun tirage ne réussit et l'algorithme répond NON (la réponse incorrecte) ; et avec probabilité $1 - (1 - c/n^k)^M$ un tirage au moins réussit et l'algorithme répond OUI (la réponse correcte).

Question 6. Si on suppose que G n'a pas de chemin de longueur k , que répond l'algorithme (1) ?

correction

Quand le graphe n'a pas de chemin, l'algorithme répond toujours NON (et c'est la réponse correcte). Cette question est sans difficulté et sert juste à vérifier la compréhension du comportement de cet algorithme probabiliste.

Question 7. Pour $M = n^k$, montrer que l'algorithme répond correctement dans les deux cas avec probabilité au moins $1/2$.

correction

Si le graphe d'entrée n'a pas de chemin, il n'y a rien à montrer par la question précédente. S'il y en a un, il faut montrer que la probabilité d'échec est d'au plus $1/2$. C'est-à-dire montrer que $(1 - 1/n^k)^{n^k} \leq 1/2$. Excluons le cas trivial où $n^k = 1$. Le logarithme du membre gauche vaut $n^k \ln(1 - 1/n^k)$, ce qui par concavité du logarithme est plus petit que $n^k \times (-1/n^k)$, donc plus petit que -1 . Or le logarithme du membre droit vaut $-\ln 2$. Comme $e > 2$ on a bien $-\ln 2 \geq -1$. On en déduit donc que la probabilité d'échec est au plus $1/2$.

Question 8. Quel est le temps d'exécution de l'algorithme (1) pour ce choix de M ? Commenter.

correction

L'algorithme (1) s'exécute en $O(M \times m)$. Donc pour le M indiqué on retrouve la même complexité qu'en question 2. On attend du candidat qu'il remarque que l'algorithme (1) n'est donc pas très intéressant en tant que tel.

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 ?

correction

Cette question commence à nécessiter de l'initiative de la part du candidat. Il faut en effet remarquer que les graphes $G_{<}$ sont toujours acycliques. En effet, l'ordre \prec peut servir de tri topologique ; autrement dit un cycle donnerait un cycle dans l'ordre total ce qui par transitivité impliquerait qu'il n'est pas asymétrique.

Or, on peut résoudre efficacement sur des graphes acycliques le problème de savoir s'ils admettent un chemin de longueur k . En effet, on considère les sommets de $G_{<}$ dans l'ordre inverse de $<$, et on calcule pour chaque sommet v la longueur $l(v)$ du plus long chemin qui part de v . Si v n'a aucune arête sortante, cette longueur est de 0. Si v a des arêtes sortantes vers w_1, \dots, w_k , les valeurs $l(w_1), \dots, l(w_k)$ étant déjà calculées car $v < w_i$ pour tout i , on pose $l(v) = 1 + \max_i l(w_i)$.

Cet algorithme s'exécute en temps linéaire, c'est-à-dire en $O(n + m)$ sur un graphe à n sommets et m arêtes. On note en particulier que cette complexité ne dépend pas de k .

Question 10. Proposer un M qui garantisse que cet algorithme réponde correctement avec une probabilité $\geq 1/2$. Quelle complexité obtient-on ? Commenter.

correction

Encore une fois, l'algorithme est biaisé vers le faux : si le graphe d'entrée n'a pas de chemin il répond toujours NON (correctement). Donc il suffit de majorer la probabilité que l'algorithme réponde NON alors que le graphe contient un chemin de longueur k . Or, si le graphe d'entrée a un chemin de longueur k , alors il est identifié si chacune de ses $k - 1$ arêtes sont conservées, et c'est le cas si et seulement si les sommets sont triés dans le bon ordre par l'ordre total qu'on a tiré. Ainsi chaque tirage a une probabilité de $1/k!$ d'identifier le chemin, et la probabilité d'échec est d'au plus $(1 - 1/k!)^M$.

On peut donc prendre $M = k!$ pour obtenir une probabilité d'échec constante, pour la même raison que pour l'algorithme (1). La complexité de l'algorithme est alors de $O(k! \times (n + m))$ si on suppose qu'on peut tirer un ordre aléatoire en temps $O(m)$. C'est le cas, par exemple en $O(n)$ avec l'algorithme de Fisher-Yates pour tirer une permutation aléatoire, mais ces détails n'étaient pas demandés au candidat.

On remarque que la complexité obtenue, qui est de $O(k! \times (n + m))$ est bien meilleure que $O(n^k)$ du moment que k n'est pas trop grand.

Question 11. En utilisant les chemins multicolores, proposer un algorithme probabiliste qui réponde 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.

correction

Dans cette dernière question, il faut davantage de créativité pour espérer répondre. L'idée est de répéter M fois l'opération suivante : tirer un k -coloriage aléatoire du graphe, et vérifier si on trouve un chemin multicolore de longueur k avec l'algorithme de la question 4, en temps $O(2^k(n + m))$.

Quand il n'y a pas de chemin de longueur k , alors il n'y a pas de tel chemin qui soit multicolore et l'algorithme répond correctement NON.

Quand il y a un tel chemin, la probabilité de le trouver est la probabilité que ce chemin soit effectivement multicolore dans le coloriage tiré. Il y a k^k coloriages possibles dont $k!$ sont corrects. Or, on sait par la formule de Stirling que $k! = \Omega((k/e)^k)$. Donc la probabilité qu'un chemin de longueur k donné soit multicolore est de $\Omega(1/e^k)$.

Ainsi, si on tire $M = e^k$ coloriages aléatoires et qu'on cherche à chaque fois un chemin multicolore, on aura à nouveau une probabilité constante de succès si le graphe contient un chemin de longueur k . On peut plus précisément garantir une probabilité de $1/2$ au moins, en multipliant par une constante le nombre de tirages. Le temps d'exécution sera alors bien de $O(e^k 2^k(n + m))$, à savoir $M = e^k$ tirages aléatoires (à une constante multiplicative près) puis $O(2^k(n + m))$ pour chaque tirage.

Cette technique du codage par couleur, introduite par Alon, Yuster, et Zwick, est utilisable pour d'autres problèmes : voir par exemple <https://en.wikipedia.org/wiki/Color-coding>.

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, et 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 ?

correction

(L'examineur demande de justifier que $\max_{1 \leq j < n} (C(j) + C(n-j)) = C(1) + C(n-1)$.)

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^{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 ?

correction

$x \vee y$ est équivalente à $\neg(\neg x \wedge \neg y)$ dans L_1 . Écrivons la table de vérité de $\neg x \wedge \neg y$ dans L_2 :

$x \backslash y$	V	F	I
V	F	F	F
F	F	V	I
I	F	I	I

puis celle de $\neg(\neg x \wedge \neg y)$:

$x \backslash y$	V	F	I
V	V	V	V
F	V	F	I
I	V	I	I

Cette table est la même que celle de $x \vee y$, et les deux propositions sont équivalentes dans L_2 .

Q2. Indiquer une proposition P_2 équivalente dans L_2 à la proposition $\langle x \Rightarrow y \rangle$ et n'utilisant pas d'autres connecteurs que \neg et \vee . Les propositions P_2 et $\langle x \Rightarrow y \rangle$ sont-elles équivalentes dans L_3 ?

correction

Dans L_1 , $x \Rightarrow y$ est équivalente à $y \vee \neg x$. Écrivons la table de vérité de $y \vee \neg x$:

$x \backslash y$	V	F	I
V	V	F	I
F	V	V	V
I	V	I	I

Celle-ci diffère de la table de vérité de $x \Rightarrow y$ (case $(x = I, y = I)$), et ces deux propositions ne sont donc pas équivalentes.

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) ?

correction

En calculant la table de vérité de P_3 , on obtient :

$x \backslash y$	V	F	I
V	V	V	V
F	V	V	V
I	V	I	I

Cette proposition n'est donc pas vraie pour toute valeur des variables, et n'est pas une tautologie.

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 $\langle (x \vee y) \Rightarrow z \rangle$. Les propositions P_4 et $\langle (x \vee y) \Rightarrow z \rangle$ sont-elles équivalentes dans L_3 ?

correction

$(x \vee y) \Rightarrow z \Leftrightarrow z \vee \neg(x \vee y) \Leftrightarrow z \vee (\neg x \wedge \neg y)$, qui est sous forme normale disjonctive. On peut donc choisir $P_4 = z \vee (\neg x \wedge \neg y)$. Ces propositions ne sont pas équivalentes dans L_2 : en effet, si $x = y = z = I$, P_4 s'évalue en I , alors que $(x \vee y) \Rightarrow z$ s'évalue en V .

Q5. Peut-on faire un raisonnement par contraposition dans L_3 (on justifiera la réponse)?

correction

On calcule la table de vérité de $\neg y \Rightarrow \neg x$:

$x \backslash y$	V	F	I
V	V	F	I
F	V	V	V
I	V	I	V

$x \Rightarrow y$ et $\neg y \Rightarrow \neg x$ sont donc équivalentes, et on peut donc effectuer des raisonnements par contraposition dans L_3 .

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)?

correction

D'après les points 3 et 4, les tables de vérité \vee de \Rightarrow peuvent être déterminées à partir de celles de \wedge et \neg (\wedge et \neg forment un système complet de connecteur). Il suffit donc de raisonner sur les tables de \wedge et \neg . D'après le premier point, et par commutativité de \wedge , on peut déjà les remplir de la façon suivante :

$x \backslash y$	V	F	I
V	V	F	X_1
F	F	F	X_2
I	X_1	X_2	X_3

x	
V	F
F	V
I	X_4

On a $\neg X_4 = \neg \neg I = I$. Comme on n'a ni $\neg V = I$ ou $\neg F = I$, on en déduit $X_4 = I$.

Par croissance de \wedge , on a nécessairement $X_2 = F$. On a donc pour l'instant la table suivante pour \wedge (les lignes et colonnes ont été permutées) :

$x \backslash y$	V	I	F
V	V	X_1	F
I	X_1	X_3	F
F	F	F	F

On peut choisir les trois valeurs possibles pour X_1 :

- Si $X_1 = V$, X_3 peut prendre les trois valeurs V, I, F .
- Si $X_1 = I$, X_3 peut prendre les valeurs I, F par croissance de \wedge .
- Si $X_1 = F$, X_3 ne peut prendre que la valeur F par croissance de \wedge .

Il y a finalement 6 logiques satisfaisant tous les points.

THE END