Table des matières

1 Index thématique **5**

2 (CCINP) Arbres Binaires de Recherche * (CCINP 0, ex A, corrigé — oral - 77 lignes) **17**

3 (CCINP) Déduction naturelle * (CCINP 0, ex A, corrigé — oral - 160 lignes) **19**

4 (CCINP) ID3 * (CCINP 0, ex A, corrigé — oral - 177 lignes) **21**

5 (CCINP) Langages réguliers * (CCINP 0, ex A, corrigé — oral - 120 lignes) **23**

6 (CCINP) Mutex * (CCINP 0, ex A, corrigé — oral - 110 lignes) **25**

7 (CCINP) SQL * (CCINP 0, ex A, corrigé — oral - 95 lignes) **27**

8 (CCINP) Arbres * (CCINP 0, ex B, corrigé — oral - 247 lignes) **29**

9 (CCINP) Graphes * (CCINP 0, ex B, corrigé — oral - 144 lignes) **31**

10 (CCINP) SAT * (CCINP 0, ex B, corrigé — oral - 148 lignes) **33**

11 (CCINP) Sac à dos * (CCINP 0, ex B, corrigé — oral - 178 lignes) **35**

12 (CCINP) Tableaux * (CCINP 0, ex B, corrigé — oral - 149 lignes) **37**

13 (CCINP) Tableaux autoréférents * (CCINP 0, ex B, corrigé — oral - 195 lignes) **39**

14 (CCINP) Activation de processus * (CCINP 23, ex A, corrigé — oral - 156 lignes) **41**

15 (CCINP) Formules propositionnelles croissantes * (CCINP 23, ex A, corrigé — oral - 142 lignes) **43**

16 (CCINP) Grammaires algébriques * (CCINP 23, ex A, corrigé — oral - 123 lignes) **45**

17 (CCINP) Minima locaux dans des arbres (Couverture dans des arbres) * (CCINP 23, ex A, corrigé — oral - 148 lignes) **47**

18 (CCINP) Langages locaux * (CCINP 23, ex B, corrigé — oral - 215 lignes) **49**

19 (CCINP) Programmation dynamique pour la récolte de fleurs * (CCINP 23, ex B, corrigé — oral - 204 lignes) **51**

20 (CCINP) calculs avec les flottants * (CCINP 23, ex B, corrigé — oral - 207 lignes) **53**

21 (CCINP) chemins simples sans issue * (CCINP 23, ex B, corrigé — oral - 199 lignes) **55**

22 (CCINP) Grammaires pour des langages de programmation * (CCINP 24, ex A, corrigé, à débugguer — oral - 90 lignes) **57**

23 (CCINP) chomp * (CCINP 24, ex A, corrigé — oral - 228 lignes) **59**

24 (CCINP) relation d'équivalence * (CCINP 24, ex A, corrigé — oral - 139 lignes) **61**

25 (CCINP) Fonctions pour la détection de motifs * (CCINP 24, ex B, corrigé — oral - 130 lignes) **63**

26 (CCINP) HORNSAT * (CCINP 24, ex B, corrigé — oral - 186 lignes) **65**

27 (CCINP) Mots de Dyck * (CCINP 24, ex B, corrigé — oral - 157 lignes) **67**

28 (ENS) Jeu des jetons * (ENS 23 MP, corrigé — oral - 153 lignes)** **69**

29 (ENS) Monoïdes et Langages * (ENS 23 MP, corrigé — oral - 238 lignes) 71**

30 (ENS) Élimination des coupures dans MLL et MALL * (ENS 23, corrigé partiellement — oral - 385 lignes) 73**

31 (ENS) Graphes parfaits * (ENS 23, corrigé — oral - 251 lignes) 75**

32 (ENS) Inférence de type * (ENS 23, corrigé — oral - 280 lignes) 77**

33 (ENS) Ordonnancement * (ENS 23, corrigé — oral - 201 lignes) 79**

34 (ENS) Permutations triables par pile * (ENS 23, corrigé — oral - 184 lignes) 81**

35 (ENS) Raisonnements ensemblistes * (ENS 23, corrigé — oral - 234 lignes) 83**

36 (ENS) Théorème des amis * (ENS 23, corrigé — oral - 202 lignes) 85**

37 (ENS) Structures pliables et traversables * (ENS 23, partiellement corrigé — oral - 306 lignes) 87**

38 (ENS) Composition Monadique * (ENS 24, corrigé très partiellement — oral - 191 lignes) 89**

39 (ENS) Déduction de messages * (ENS 24, corrigé — oral - 232 lignes) 91**

40 (ENS) Filtrage par motif en OCaml * (ENS 24, corrigé — oral - 323 lignes) 93**

41 (ENS) Implémentation des circuits réversibles * (ENS 24, corrigé — oral - 218 lignes) 95**

42 (ENS) Résolution d'inéquations linéaires * (ENS 24, corrigé — oral - 195 lignes) 97**

43 (ENS) Mots partiels et Théorème de Dilworth * (ENS 24, partiellement corrigé, niveau inapproprié, à débugguer — oral - 184 lignes) 99**

44 (ENS) Terminaison de lambda ref * (ENS 24, partiellement corrigé, niveau surréaliste — oral - 238 lignes) 101**

45 (MT) Déduction Naturelle * (MT 0, ex 1, corrigé — oral - 200 lignes) 103

46 (MT) Relations entre les nombres de sommets et d'arêtes de grphes * (MT 0, ex 1, corrigé — oral - 102 lignes) 105

47 (MT) bdd pour livraison * (MT 0, ex 1, corrigé — oral - 78 lignes) 107

48 (MT) couplages * (MT 0, ex 1, corrigé — oral - 64 lignes) 109

49 (MT) logique, cours * (MT 0, ex 1, corrigé — oral - 100 lignes) 111

50 (MT) Arbres Binaires de Recherche * (MT 0, ex 2, corrigé — oral - 159 lignes) 113

51 (MT) entrelacements de mots * (MT 0, ex 2, corrigé — oral - 109 lignes) 115

52 (MT) graphes * (MT 0, ex 2, corrigé — oral - 121 lignes) 117

53 (MT) montée et descente d'un bus * (MT 0, ex 2, corrigé — oral - 104 lignes) 119

54 (MT) subset-sum * (MT 0, ex 2, corrigé — oral - 143 lignes) 121

55 (MT) Classification hiérarchique ascendante * (MT 24, ex 1, corrigé, le graphique n'est pas l'original — oral - 44 lignes) 123

56 (MT) Graphes bipartis * (MT 24, ex 1, corrigé — oral - 88 lignes) 125

57 (MT) Tas binaires * (MT 24, ex 1, corrigé — oral - 129 lignes) 127

58 (MT) logique * (MT 24, ex 1, corrigé — oral - 141 lignes) 129

59 (MT) Automates de Büchi * (MT 24, ex 2, corrigé, (1 question ajoutée) — oral - 117 lignes) 131**

60 (MT) Automates, langages à saut * (MT 24, ex 2, corrigé, complété au jugé pour les dernières questions — oral - 85 lignes) 133

61 (MT) plus courts chemins dans des graphes * (MT 24, ex 2, corrigé, retour d'oral — oral - 117 lignes) 135

62 (MT) Décidabilité de programmes engendrés par une grammaire * (MT 24, ex 2, corrigé — oral - 125 lignes) 137

63 (X) Algorithme Union-Find * (X 23, corrigé — oral - 151 lignes) 139**

64 (X) Bob l'écureuil * (X 23, corrigé — oral - 141 lignes) 141**

65 (X) Langages rationnels et lemme de l'étoile * (X 23, corrigé — oral - 118 lignes) 143**

66 (X) Algorithme du tri lent * (X 24, corrigé — oral - 147 lignes) 145**

67 (X) Codage par couleur * (X 24, corrigé — oral - 154 lignes) 147**

68 (X) Rationnalité de langages * (X ? ?, partiellement corrigé, complété au jugé pour les trois dernières questions — oral - 186 lignes) 149**

Chapitre 1

Index thématique

Algorithmes d'approximation

- 53-subset-sum (MT 0, ex 2, **corrigé** — oral - 143 lignes) *
  complexité, algorithmes d'approximation, algorithmes gloutons
  VOIR

Algorithmes gloutons

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

- 53-subset-sum (MT 0, ex 2, **corrigé** — oral - 143 lignes) *
  complexité, algorithmes d'approximation, algorithmes gloutons
  VOIR

Algorithmes probabilistes

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

Algorithmique

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

- 11-Tableaux (CCINP 0, ex B, **corrigé** — oral - 149 lignes) *
  algorithmique, c
  VOIR

- 12-Tableaux autoréférents (CCINP 0, ex B, **corrigé** — oral - 195 lignes) *
  algorithmique, backtracking, ocaml
  VOIR

- 23-relation d'équivalence (CCINP 24, ex A, **corrigé** — oral - 139 lignes) *
  algorithmique, graphes
  VOIR

- 31-Inférence de type (ENS 23, **corrigé** — oral - 280 lignes) ***
  algorithmique, langages fonctionnels, pseudocode
  VOIR

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

algorithmique, complexité
VOIR

- 33-Permutations triables par pile (ENS 23, **corrigé** — oral - 184 lignes) ***
  algorithmique, graphes, complexité
  VOIR

- 41-Résolution d'inéquations linéaires (ENS 24, **corrigé** — oral - 195 lignes) ***
  ocaml, algorithmique, complexité
  VOIR

- 62-Algorithme Union-Find (X 23, **corrigé** — oral - 151 lignes) ***
  algorithmique, complexité, structures de données
  VOIR

- 63-Bob l'écureuil (X 23, **corrigé** — oral - 141 lignes) ***
  algorithmique
  VOIR

- 65-Algorithme du tri lent (X 24, **corrigé** — oral - 147 lignes) ***
  algorithmique, tri, complexité, preuve
  VOIR

- 66-Codage par couleur (X 24, **corrigé** — oral - 154 lignes) ***
  algorithmique, complexité, graphes, backtracking
  VOIR

Arbres

- 1-Arbres Binaires de Recherche (CCINP 0, ex A, **corrigé** — oral - 77 lignes) *
  arbres, complexité
  VOIR

- 7-Arbres (CCINP 0, ex B, **corrigé** — oral - 247 lignes) *
  arbres, ocaml
  VOIR

- 16-Minima locaux dans des arbres (Couverture dans des arbres) (CCINP 23, ex A, **corrigé** — oral - 148 lignes) *
  arbres, complexité
  VOIR

- 49-Arbres Binaires de Recherche (MT 0, ex 2, **corrigé** — oral - 159 lignes) *
  arbres, complexité, programmation dynamique
  VOIR

- 56-Tas binaires (MT 24, ex 1, **corrigé** — oral - 129 lignes) *
  arbres, tri, structures de données, cours, complexité
  VOIR

Automates

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

- 17-Langages locaux (CCINP 23, ex B, **corrigé** — oral - 215 lignes) *
  langages, ocaml, automates
  VOIR

- 50-entrelacements de mots (MT 0, ex 2, **corrigé** — oral - 109 lignes) *
  langages, automates, programmation dynamique
  VOIR

- 58-Automates de Büchi (MT 24, ex 2, **corrigé**, (1 question ajoutée) — oral - 117 lignes) ***
  langages, automates
  VOIR

- 59-Automates, langages à saut (MT 24, ex 2, **corrigé**, complété au jugé pour les dernières questions — oral - 85 lignes) *
  langages, automates
  VOIR

Backtracking

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

- 12-Tableaux autoréférents (CCINP 0, ex B, **corrigé** — oral - 195 lignes) *
  algorithmique, backtracking, ocaml
  VOIR

- 20-chemins simples sans issue (CCINP 23, ex B, **corrigé** — oral - 199 lignes) *
  graphes, ocaml, complexité, backtracking
  VOIR

- 66-Codage par couleur (X 24, **corrigé** — oral - 154 lignes) ***
  algorithmique, complexité, graphes, backtracking
  VOIR

Branch and bound

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

C

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

- 11-Tableaux (CCINP 0, ex B, **corrigé** — oral - 149 lignes) *
  algorithmique, c
  VOIR

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

- 19-calculs avec les flottants (CCINP 23, ex B, **corrigé** — oral - 207 lignes) *
  représentation des nombres, c
  VOIR

- 26-Mots de Dyck (CCINP 24, ex B, **corrigé** — oral - 157 lignes) *
  langages, c, complexité
  VOIR

- 61-Décidabilité de programmes engendrés par une grammaire (MT 24, ex 2, **corrigé** — oral - 125 lignes) *
  grammaire, décidabilité, c, logique propositionnelle
  VOIR

Circuits

- 40-Implémentation des circuits réversibles (ENS 24, **corrigé** — oral - 218 lignes) ***
  circuits
  VOIR

Classification hiérarchique ascendante

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

Complexité

- 1-Arbres Binaires de Recherche (CCINP 0, ex A, **corrigé** — oral - 77 lignes) *
  arbres, complexité
  VOIR

- 13-Activation de processus (CCINP 23, ex A, **corrigé** — oral - 156 lignes) *
  complexité, réduction
  VOIR

- 16-Minima locaux dans des arbres (Couverture dans des arbres) (CCINP 23, ex A, **corrigé** — oral - 148 lignes) *
  arbres, complexité
  VOIR

- 20-chemins simples sans issue (CCINP 23, ex B, **corrigé** — oral - 199 lignes) *
  graphes, ocaml, complexité, backtracking
  VOIR

- 25-HORNSAT (CCINP 24, ex B, **corrigé** — oral - 186 lignes) *
  logique propositionnelle, complexité, réduction
  VOIR

- 26-Mots de Dyck (CCINP 24, ex B, **corrigé** — oral - 157 lignes) *
  langages, c, complexité
  VOIR

- 32-Ordonnancement (ENS 23, **corrigé** — oral - 201 lignes) ***
  algorithmique, complexité
  VOIR

- 33-Permutations triables par pile (ENS 23, **corrigé** — oral - 184 lignes) ***
  algorithmique, graphes, complexité
  VOIR

- 41-Résolution d'inéquations linéaires (ENS 24, **corrigé** — oral - 195 lignes) ***
  ocaml, algorithmique, complexité
  VOIR

- 49-Arbres Binaires de Recherche (MT 0, ex 2, **corrigé** — oral - 159 lignes) *
  arbres, complexité, programmation dynamique
  VOIR

- 53-subset-sum (MT 0, ex 2, **corrigé** — oral - 143 lignes) *
  complexité, algorithmes d'approximation, algorithmes gloutons
  VOIR

- 55-Graphes bipartis (MT 24, ex 1, **corrigé** — oral - 88 lignes) *
  graphes, logique propositionnelle, parcours de graphes, complexité
  VOIR

- 56-Tas binaires (MT 24, ex 1, **corrigé** — oral - 129 lignes) *
  arbres, tri, structures de données, cours, complexité
  VOIR

- 60-plus courts chemins dans des graphes (MT 24, ex 2, **corrigé**, retour d'oral — oral - 117 lignes) *
  graphes, complexité, programmation dynamique
  VOIR

- 62-Algorithme Union-Find (X 23, **corrigé** — oral - 151 lignes) ***
  algorithmique, complexité, structures de données
  VOIR

- 65-Algorithme du tri lent (X 24, **corrigé** — oral - 147 lignes) ***
  algorithmique, tri, complexité, preuve
  VOIR

- 66-Codage par couleur (X 24, **corrigé** — oral - 154 lignes) ***
  algorithmique, complexité, graphes, backtracking
  VOIR

Concurrence

- 5-Mutex (CCINP 0, ex A, **corrigé** — oral - 110 lignes) *
  concurrence, ocaml
  VOIR

- 52-montée et descente d'un bus (MT 0, ex 2, **corrigé** — oral - 104 lignes) *
  concurrence
  VOIR

Cours

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

- 56-Tas binaires (MT 24, ex 1, **corrigé** — oral - 129 lignes) *
  arbres, tri, structures de données, cours, complexité
  VOIR

Décidabilité

- 61-Décidabilité de programmes engendrés par une grammaire (MT 24, ex 2, **corrigé** — oral - 125 lignes) *
  grammaire, décidabilité, c, logique propositionnelle
  VOIR

Déduction naturelle

- 2-Déduction naturelle (CCINP 0, ex A, **corrigé** — oral - 160 lignes) *
  déduction naturelle
  VOIR

- 38-Déduction de messages (ENS 24, **corrigé** — oral - 232 lignes) ***
  déduction naturelle, réduction
  VOIR

- 44-Déduction Naturelle (MT 0, ex 1, **corrigé** — oral - 200 lignes) *

déduction naturelle
VOIR

Grammaire

- 61-Décidabilité de programmes engendrés par une grammaire (MT 24, ex 2, **corrigé** — oral - 125 lignes) *
  grammaire, décidabilité, c, logique propositionnelle
  VOIR

Grammaires

- 15-Grammaires algébriques (CCINP 23, ex A, **corrigé** — oral - 123 lignes) *
  grammaires
  VOIR

- 21-Grammaires pour des langages de programmation (CCINP 24, ex A, **corrigé**, **à débugguer** — oral - 90 lignes) *
  grammaires, langages
  VOIR

Graphes

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

- 20-chemins simples sans issue (CCINP 23, ex B, **corrigé** — oral - 199 lignes) *
  graphes, ocaml, complexité, backtracking
  VOIR

- 23-relation d'équivalence (CCINP 24, ex A, **corrigé** — oral - 139 lignes) *
  algorithmique, graphes
  VOIR

- 27-Jeu des jetons (ENS 23 MP, **corrigé** — oral - 153 lignes) ***
  jeux, graphes
  VOIR

- 30-Graphes parfaits (ENS 23, **corrigé** — oral - 251 lignes) ***
  graphes
  VOIR

- 33-Permutations triables par pile (ENS 23, **corrigé** — oral - 184 lignes) ***
  algorithmique, graphes, complexité
  VOIR

- 34-Raisonnements ensemblistes (ENS 23, **corrigé** — oral - 234 lignes) ***
  graphes, logique
  VOIR

- 35-Théorème des amis (ENS 23, **corrigé** — oral - 202 lignes) ***
  graphes
  VOIR

- 42-Mots partiels et Théorème de Dilworth (ENS 24, partiellement **corrigé**, niveau inapproprié, **à débugguer** — oral - 184 lignes) ***
  graphes, réduction, langages
  VOIR

- 45-Relations entre les nombres de sommets et d'arêtes de grphes (MT 0, ex 1, **corrigé** — oral - 102 lignes) *

graphes
VOIR

- 47-couplages (MT 0, ex 1, **corrigé** — oral - 64 lignes) *
  graphes
  VOIR

- 51-graphes (MT 0, ex 2, **corrigé** — oral - 121 lignes) *
  graphes
  VOIR

- 55-Graphes bipartis (MT 24, ex 1, **corrigé** — oral - 88 lignes) *
  graphes, logique propositionnelle, parcours de graphes, complexité
  VOIR

- 60-plus courts chemins dans des graphes (MT 24, ex 2, **corrigé**, retour d'oral — oral - 117 lignes) *
  graphes, complexité, programmation dynamique
  VOIR

- 66-Codage par couleur (X 24, **corrigé** — oral - 154 lignes) ***
  algorithmique, complexité, graphes, backtracking
  VOIR

Ia

- 3-ID3 (CCINP 0, ex A, **corrigé** — oral - 177 lignes) *
  ia
  VOIR

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

Jeux

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

- 27-Jeu des jetons (ENS 23 MP, **corrigé** — oral - 153 lignes) ***
  jeux, graphes
  VOIR

Langages

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

- 17-Langages locaux (CCINP 23, ex B, **corrigé** — oral - 215 lignes) *
  langages, ocaml, automates
  VOIR

- 21-Grammaires pour des langages de programmation (CCINP 24, ex A, **corrigé**, **à débugguer** — oral - 90 lignes) *
  grammaires, langages
  VOIR

- 24-Fonctions pour la détection de motifs (CCINP 24, ex B, **corrigé** — oral - 130 lignes) *
  langages, ocaml
  VOIR

- 26-Mots de Dyck (CCINP 24, ex B, **corrigé** — oral - 157 lignes) *
  langages, c, complexité
  VOIR

- 28-Monoïdes et Langages (ENS 23 MP, **corrigé** — oral - 238 lignes) ***
  langages
  VOIR

- 42-Mots partiels et Théorème de Dilworth (ENS 24, partiellement **corrigé**, niveau inapproprié, **à débugguer** — oral - 184 lignes) ***
  graphes, réduction, langages
  VOIR

- 50-entrelacements de mots (MT 0, ex 2, **corrigé** — oral - 109 lignes) *
  langages, automates, programmation dynamique
  VOIR

- 58-Automates de Büchi (MT 24, ex 2, **corrigé**, (1 question ajoutée) — oral - 117 lignes) ***
  langages, automates
  VOIR

- 59-Automates, langages à saut (MT 24, ex 2, **corrigé**, complété au jugé pour les dernières questions — oral - 85 lignes) *
  langages, automates
  VOIR

- 64-Langages rationnels et lemme de l'étoile (X 23, **corrigé** — oral - 118 lignes) ***
  langages
  VOIR

- 67-Rationnalité de langages (X ??, partiellement **corrigé**, complété au jugé pour les trois dernières questions — oral - 186 lignes) ***
  langages, lemme de l'étoile
  VOIR

Langages fonctionnels

- 31-Inférence de type (ENS 23, **corrigé** — oral - 280 lignes) ***
  algorithmique, langages fonctionnels, pseudocode
  VOIR

Lemme de l'étoile

- 67-Rationnalité de langages (X ??, partiellement **corrigé**, complété au jugé pour les trois dernières questions — oral - 186 lignes) ***
  langages, lemme de l'étoile
  VOIR

Logique

- 34-Raisonnements ensemblistes (ENS 23, **corrigé** — oral - 234 lignes) ***
  graphes, logique
  VOIR

Logique propositionnelle

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

logique propositionnelle, algorithmes probabilistes
VOIR

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

- 25-HORNSAT (CCINP 24, ex B, **corrigé** — oral - 186 lignes) *
  logique propositionnelle, complexité, réduction
  VOIR

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

- 48-logique, cours (MT 0, ex 1, **corrigé** — oral - 100 lignes) *
  logique propositionnelle
  VOIR

- 55-Graphes bipartis (MT 24, ex 1, **corrigé** — oral - 88 lignes) *
  graphes, logique propositionnelle, parcours de graphes, complexité
  VOIR

- 57-logique (MT 24, ex 1, **corrigé** — oral - 141 lignes) *
  logique propositionnelle
  VOIR

- 61-Décidabilité de programmes engendrés par une grammaire (MT 24, ex 2, **corrigé** — oral - 125 lignes) *
  grammaire, décidabilité, c, logique propositionnelle
  VOIR

Ocaml

- 5-Mutex (CCINP 0, ex A, **corrigé** — oral - 110 lignes) *
  concurrence, ocaml
  VOIR

- 7-Arbres (CCINP 0, ex B, **corrigé** — oral - 247 lignes) *
  arbres, ocaml
  VOIR

- 12-Tableaux autoréférents (CCINP 0, ex B, **corrigé** — oral - 195 lignes) *
  algorithmique, backtracking, ocaml
  VOIR

- 17-Langages locaux (CCINP 23, ex B, **corrigé** — oral - 215 lignes) *
  langages, ocaml, automates
  VOIR

- 20-chemins simples sans issue (CCINP 23, ex B, **corrigé** — oral - 199 lignes) *
  graphes, ocaml, complexité, backtracking
  VOIR

- 24-Fonctions pour la détection de motifs (CCINP 24, ex B, **corrigé** — oral - 130 lignes) *
  langages, ocaml
  VOIR

- 36-Structures pliables et traversables (ENS 23, partiellement **corrigé** — oral - 306 lignes) ***
  ocaml, programmation fonctionnelle
  VOIR

- 37-Composition Monadique (ENS 24, **corrigé** très partiellement — oral - 191 lignes) ***
  ocaml, programmation fonctionnelle
  VOIR

- 39-Filtrage par motif en OCaml (ENS 24, **corrigé** — oral - 323 lignes) ***
  ocaml
  VOIR

- 41-Résolution d'inéquations linéaires (ENS 24, **corrigé** — oral - 195 lignes) ***
  ocaml, algorithmique, complexité
  VOIR

- 43-Terminaison de lambda ref (ENS 24, partiellement **corrigé**, niveau surréaliste — oral - 238 lignes) ***
  ocaml
  VOIR

Parcours de graphes

- 55-Graphes bipartis (MT 24, ex 1, **corrigé** — oral - 88 lignes) *
  graphes, logique propositionnelle, parcours de graphes, complexité
  VOIR

Preuve

- 65-Algorithme du tri lent (X 24, **corrigé** — oral - 147 lignes) ***
  algorithmique, tri, complexité, preuve
  VOIR

Programmation dynamique

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

- 49-Arbres Binaires de Recherche (MT 0, ex 2, **corrigé** — oral - 159 lignes) *
  arbres, complexité, programmation dynamique
  VOIR

- 50-entrelacements de mots (MT 0, ex 2, **corrigé** — oral - 109 lignes) *
  langages, automates, programmation dynamique
  VOIR

- 60-plus courts chemins dans des graphes (MT 24, ex 2, **corrigé**, retour d'oral — oral - 117 lignes) *
  graphes, complexité, programmation dynamique
  VOIR

Programmation fonctionnelle

- 36-Structures pliables et traversables (ENS 23, partiellement **corrigé** — oral - 306 lignes) ***
  ocaml, programmation fonctionnelle
  VOIR

- 37-Composition Monadique (ENS 24, **corrigé** très partiellement — oral - 191 lignes) ***
  ocaml, programmation fonctionnelle
  VOIR

Pseudocode

- 31-Inférence de type (ENS 23, **corrigé** — oral - 280 lignes) ***

algorithmique, langages fonctionnels, pseudocode
VOIR

Représentation des nombres

- 19-calculs avec les flottants (CCINP 23, ex B, **corrigé** — oral - 207 lignes) *
  représentation des nombres, c
  VOIR

Réduction

- 13-Activation de processus (CCINP 23, ex A, **corrigé** — oral - 156 lignes) *
  complexité, réduction
  VOIR

- 25-HORNSAT (CCINP 24, ex B, **corrigé** — oral - 186 lignes) *
  logique propositionnelle, complexité, réduction
  VOIR

- 38-Déduction de messages (ENS 24, **corrigé** — oral - 232 lignes) ***
  déduction naturelle, réduction
  VOIR

- 42-Mots partiels et Théorème de Dilworth (ENS 24, partiellement **corrigé**, niveau inapproprié, **à débugguer** — oral - 184 lignes) ***
  graphes, réduction, langages
  VOIR

Sql

- 6-SQL (CCINP 0, ex A, **corrigé** — oral - 95 lignes) *
  sql
  VOIR

- 46-bdd pour livraison (MT 0, ex 1, **corrigé** — oral - 78 lignes) *
  sql
  VOIR

Structures de données

- 56-Tas binaires (MT 24, ex 1, **corrigé** — oral - 129 lignes) *
  arbres, tri, structures de données, cours, complexité
  VOIR

- 62-Algorithme Union-Find (X 23, **corrigé** — oral - 151 lignes) ***
  algorithmique, complexité, structures de données
  VOIR

Tri

- 56-Tas binaires (MT 24, ex 1, **corrigé** — oral - 129 lignes) *
  arbres, tri, structures de données, cours, complexité
  VOIR

- 65-Algorithme du tri lent (X 24, **corrigé** — oral - 147 lignes) ***

algorithmique, tri, complexité, preuve
VOIR

Chapitre 2

(CCINP) Arbres Binaires de Recherche * (CCINP 0, ex A, corrigé — oral - 77 lignes)

$*$

Arbres, Complexité,
sources : `arbreCCINPA2.tex`

Arbres binaires de recherche :

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 ?

Chapitre 3

(CCINP) Déduction naturelle * (CCINP 0, ex A, corrigé — oral - 160 lignes)

$*$

Déduction 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 :

$$\frac{}{\Gamma, A \vdash A}\ \text{AX} \quad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot_e \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to_i \quad \frac{\Gamma \vdash A \qquad \Gamma \vdash A \to B}{\Gamma \vdash B}\ \to_e \quad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg_i \quad \frac{\Gamma \vdash A \qquad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\ \neg_e$$

1. Montrer que le séquent $\vdash \neg A \to (A \to \bot)$ est dérivable, en explicitant un arbre de preuve.
2. Montrer que le séquent $\vdash (A \to \bot) \to \neg A$ est dérivable, en explicitant un arbre de preuve.
3. Donner une règle correspondant à l'introduction du symbole $\wedge$ ainsi que deux règles correspondant à l'élimination du symbole $\wedge$. Montrer que le séquent $\vdash (\neg A \to (A \to \bot)) \wedge ((A \to \bot) \to \neg A)$ est dérivable.
4. On considère la formule $P = ((A \to B) \to A) \to A$ appelée *loi de Peirce*. Montrer que $\models P$, c'est-à-dire que P est une tautologie.
5. Pour montrer que le séquent $\vdash P$ est dérivable, il est nécessaire d'utiliser la règle d'absurdité classique $\bot_c$ (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 $\Gamma = \{(A \to B) \to A, \neg A\}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{?}{\Gamma \vdash A}\ ? \qquad \dfrac{}{\Gamma \vdash \neg A}\ \text{AX}}{\Gamma \vdash \bot}\ \neg_i}{(A \to B) \to A \vdash A}\ \bot_c}{\vdash ((A \to B) \to A) \to A}\ \to_i$$

Chapitre 4

(CCINP) ID3 * (CCINP 0, ex A, corrigé — oral - 177 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 $Z = (x_i, y_i)_{i\in[\![1,n]\!]}$ avec $\forall i \in [\![1, n]\!]$, $x_i \in \mathbb{B}^d$, $y_i \in \{+, -\}$, 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 décions 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 repiratoires 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.

```
Entrées : S_p sous-ensemble des exemples du noeud parent, S sous-ensemble des ↩
    exemples à considérer, D sous-ensemble des attributs à considérer}
Sortie : Un arbre de décision.

Fonction ID3((S_p, S, D)) ;
    Si l'ensemble des exemples S est vide
        Renvoyer ···

    Si l'ensemble A des attributes est vide
        Renvoyer ···

    Si tous les exemples de S ont une même étiquette y
        Renvoyer ···

    Sinon :
        Soit A ∈ D l'attribut de plus grand gain G(S, A) \ ;
        Construire l'arbre de racine A et de sous-arbre gauche \texttt{ID3}↩
           (S, S_{A=NO}, D\{A}) et de sous-arbre droit \texttt{ID3}(S, S_{A=YES}, D\{A})
```

4. Indiquer comment compléter l'algorithme 1.

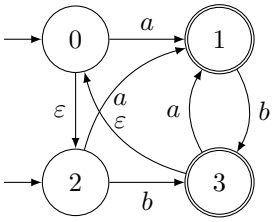
Chapitre 5

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

$*$

Langages, Automates,
sources : `langccinp3.tex`

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



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

Chapitre 6

(CCINP) Mutex * (CCINP 0, ex A, corrigé — oral - 110 lignes)

$*$

Concurrence, Ocaml,
sources : `concccinp1.tex`

Mutex :

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

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

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

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

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

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

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

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

```
(* section critique *)
veut_entrer.(i) <- false
```

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

Chapitre 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 précendents, 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 tye 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 question suivantes :
   - **(a)** Quels sont les numéros des modèles des matériels (ordinateur, portable ou imprimante) fabriqués par l'entreprise du nom de Durand ?
   - **(b)** Quels sont les noms et adresses des fabricants produisant des portables dont le disque dur a une capacité d'au moins 500 Go ?
   - **(c)** Quels sont les noms des fabricants qui produisent au moins 10 modèles différents d'imprimantes ?
   - **(d)** Quels sont les numéros des clients n'ayant acheté aucune imprimante ?

Chapitre 8

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

$*$

Arbres, Ocaml,
sources : `ccinparbre.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 $[\![0, n-1]\!]$ par la liste d'entier $[\sigma_0; \sigma_1; \ldots; \sigma_{n-1}]$. 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 $[\![0; n-1]\!]$ 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).

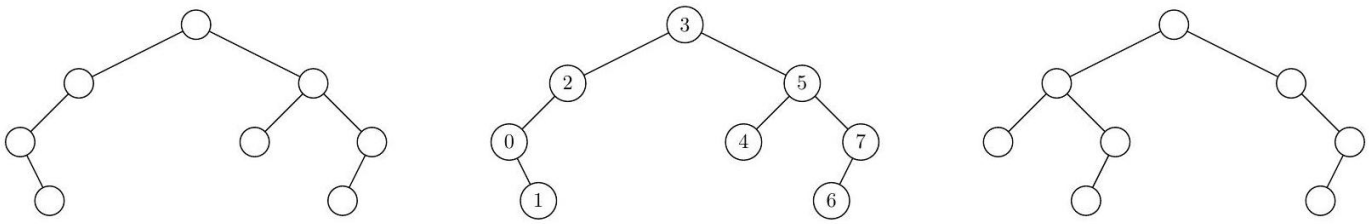


Figure 1 - (a) un arbre binaire non étiqueté ; (b) son étiquetage en suivant un ordre infixe, la permutation associée est $[3; 2; 0; 1; 5; 4; 7; 6]$; (c) un autre arbre binaire non étiqueté.

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.
2. Écrire une fonction `parcours_prefixe : arbre -> int list` qui renvoie la liste des étiquettes d'un arbre dans l'ordre préfixe de son parcours en profondeur. On pourra utiliser l'opérateur `@` et on ne cherchera pas nécessairement à proposer une solution linéaire en la taille de l'arbre.
3. Écrire une fonction `etiquette : arbre -> arbre` qui prend en paramètre un arbre dont on ignore les étiquettes et qui renvoie un arbre identique mais étiqueté par les entiers de $[\![0, n-1]\!]$ en suivant l'ordre infixe d'un parcours en profondeur.
   Indication : on pourra utiliser une fonction auxiliaire de type `arbre` $\to$ `int` $\to$ `arbre * int` qui prend en paramètres un arbre et la prochaine étiquette à mettre et qui renvoie le couple formé par l'arbre étiqueté et la nouvelle prochaine étiquette à mettre.

Une permutation σ de $[\![0, n-1]\!]$ est triable avec une pile s'il est possible de trier la liste $[\sigma_0; \sigma_1; \ldots; \sigma_{n-1}]$ 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 ↩
       sommet de la pile vers la sortie ;
    ⋆ Empiler l'élément en entrée dans la pile ;
Dépiler tous les éléments restant dans la pile vers la sortie.
```

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

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

5. Écrire une fonction trier : `int list` $\to$ `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 procèder récursivement.

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

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

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

(b) Montrer qu'une permutation triable par pile est une permutation associée à un arbre binaire.
Indication : on peut prendre 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.

Chapitre 9

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

$*$

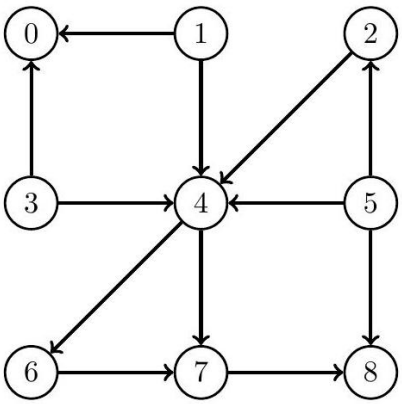
Graphes,
sources : `ccinpgraphe.tex`

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 $n \leqslant 100$. Pour $0 \leqslant s < n$, 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 $0 \leqslant s < n$, la case `voisins[s]` est un tableau contenant, aux indices $0 \leqslant i < d^+(s)$, 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 $d^*(s) = \max\left\{d^+(s') \mid s' \in \mathcal{A}(s)\right\}$. 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 $\max\left\{d^+(s') \mid s' \in S'\right\}$.
2. Écrire une fonction `bool* accessibles (graph* g, int s)` qui prend en paramètre un graphe et un sommet s et qui renvoie un (pointeur sur un) tableau de booléens de taille n représentant $\mathcal{A}(s)$. Une fonction `nb_accessibles` qui utilise votre fonction et un test pour l'exemple ci-dessus vous sont donnés dans le fichier à compléter.
3. Écrire une fonction `int degre_etoile(graph* g, int s)` qui calcule $d^*(s)$ pour un graphe et un sommet passé en paramètre. Quelle est la complexité de votre approche ?
4. Linéariser le graphe donné en exemple ci-dessus, c'est-à-dire représenter ses sommets sur une même ligne dans l'ordre donné par un tri topologique, tous les arcs allant de gauche à droite.

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

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

Chapitre 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 $\mathcal{V} = \{v_0, v_1, \ldots, v_{n-1}\}$ 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 $c = \ell_0 \vee \ell_1 \vee \cdots \vee \ell_{|c|-1}$ 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 $c_0 \wedge c_1 \wedge \cdots \wedge c_{m-1}$. 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 $v : \mathcal{V} \to \{V, F\}$ 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 `evalue : clause -> valuation -> bool` qui vérifie si une clause est satisfaite par une valuation.

Étant donné une formule f constituée de m clauses $c_0 \wedge c_1 \wedge \cdots \wedge c_{m-1}$ définies sur un ensemble de n variables, la fonction `random_sat` a pour objectif de trouver une valuation qui satisfait la formule, s'il en existe une et qu'elle y arrive. Cette fonction doit effectuer au plus k tentatives et renvoyer un résultat de type `valuation option`, avec une valuation qui satisfait la formule passée en paramètre si elle en trouve une et la valeur `None` sinon.
L'idée de ce programme est d'effectuer une assignation aléatoire des variables puis de vérifier que chaque clause est satisfaite. Si une clause n'est pas satisfaite, on modifie aléatoirement la valeur associée à un littéral de cette clause, qui devient ainsi satisfaite, puis on recommence.

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

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

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

Chapitre 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 $n \geqslant 1$ objets $\{o_0, \ldots, o_{n-1}\}$ de valeurs respectives $(v_0, \ldots, v_{n-1}) \in \mathbb{N}^n$ et de poids respectifs $(p_0, \ldots, p_{n-1}) \in \mathbb{N}^n$. 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

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

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 considèrer les objets $o_0, o_1, \ldots, o_{n-1}$ 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 $v_0, v_1, \ldots, v_{n-1}$ 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 $x_0, x_1, \ldots, x_{n-1}$.
2. Écrire une fonction qui implémente la méthode gloutonne décrite ci-dessus à partir de $n, p_{\max}$ et $p_0, p_1, \ldots, p_{n-1}$ et qui permet de renvoyer les indicateurs $x_0, x_1, \ldots, x_{n-1}$ pour le choix glouton.
3. Écrire un programme complet qui permet de lire sur l'entrée standard (au clavier par défaut) un entier $n \geqslant 1$, puis un entier naturel $p_{\max}$, puis n entiers naturels correspondant aux valeurs $v_0, v_1, \ldots, v_{n-1}$, puis n entiers naturels correspondant aux poids $p_0, p_1, \ldots, p_{n-1}$ et qui affiche sur la sortie standard (l'écran par défaut) sous une forme de votre choix les indicateurs $x_0, x_1, \ldots, x_{n-1}$, la valeur de la solution $\sum_{i=0}^{n-1} x_i v_i$ et le poids utilisé $\sum_{i=0}^{n-1} x_i p_i$. On rappelle que le spécificateur de format pour lire ou écrire un entier est % d.
4. L'algorithme glouton ci-dessus donne-t-il toujours une solution optimale ?

   (a) si on ne suppose rien sur l'ordre des objets a priori ;
   (b) si les objets sont triés par ordre de valeur décroissante ;
   (c) si les objets sont triés par ordre de poids croissant ;
   (d) si les objets sont triés par ordre décroissant des quotients $\frac{v_i}{p_i}$.

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

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

Chapitre 12

(CCINP) Tableaux * (CCINP 0, ex B, corrigé — oral - 149 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 $n \geqslant 0$.
Un squelette de programme C vous est donné, avec un jeu de tests qu'il ne faut pas modifier. Vous pouvez bien sûr ajouter vos propres tests à part.

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

Dans toute la suite, on suppose que les tableaux sont triés dans l'ordre croissant. On va chercher à écrire une version plus efficace de la fonction ci-dessus qui exploite cette propriété. On cherche tout d'abord à écrire une fonction
`int une_occurrence (int n, int* tab, int x)`
qui permet de renvoyer un indice d'une occurrence quelconque de l'élément x s'il est présent dans le tableau et -1 sinon. On procède par dichotomie.

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

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

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

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

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'apparait pas dans t.

2. Justifier rapidement qu'il n'existe aucun tableau autoréférent pour $n \in [\![1; 3]\!]$ et trouver un autre tableau autoréférent pour $n = 4$.

3. Écrire une fonction `est_auto : int array → bool` qui vérifie si un tableau de taille $n > 0$ est autoréférent. On attend une complexité en $O(n)$.

On propose d'utiliser une méthode de retour sur trace (backtracking) pour trouver tous les tableaux autoréferents pour un $n > 0$ donné. Une fonction `gen_auto` qui affiche tous les tableaux autoréférents pour une taille donnée vous est proposée. Cette version ne fonctionne cependant que pour de toutes petites valeurs de n (instantané pour $n = 5$, un peu long pour $n = 8$, sans espoir pour $n = 15$). On pourra vérifier qu'il existe exactement deux tableaux autoréférents pour $n = 4$, un seul pour $n \in \{5, 7, 8\}$ et aucun pour $n = 6$.
Pour accélérer la recherche, il faut élaguer l'arbre (repérer le plus rapidement possible qu'on se trouve dans une branche ne pouvant pas donner de solution).

4. Que peut-on dire de la somme des éléments d'un tableau autoréférent ? En déduire une stratégie d'élagage pour accélérer la recherche.
Indication : utiliser la fonction somme de la première question pour interrompre par un échec l'exploration lorsque `somme t i` dépasse déjà la valeur maximale possible.

5. Que peut-on dire si juste après avoir affecté la case $t.(i)$, il y a déjà strictement plus d'occurrences d'une valeur $0 \leqslant k \leqslant i$ que la valeur de $t.(k)$? En déduire une stratégie d'élagage supplémentaire et la mettre en œuvre. Combien de temps faut-il pour résoudre le problème pour $n = 15$?

6. Après avoir affecté la case $t.(i)$, combien de cases reste-t-il à remplir ? Combien de ces cases seront complétées par une valeur non nulle ? À quelle condition est-on alors certain que la somme dépassera la valeur maximale possible à la fin ? En déduire une stratégie d'élagage supplémentaire et la mettre en œuvre. Combien de temps faut-il pour résoudre le problème pour $n = 30$?

7. Montrer qu'il existe un tableau autoréférent pour tout $n \geqslant 7$. On pourra conjecturer la forme de ce tableau en testant empiriquement pour différentes valeurs de $n \geqslant 7$. On ne demande pas de montrer que cette solution est unique.

Chapitre 14

(CCINP) Activation de processus * (CCINP 23, ex A, corrigé — oral - 156 lignes)

$*$

Complexité, Réduction,
sources : `procesCCINP23.tex`

Activation de processus (exn 2023) :

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

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

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

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

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

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

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

Chapitre 15

(CCINP) 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 $E = \{x_1, \cdots, x_n\}$ un ensemble de variables propositionnelles. Étant données deux applications $a : E \to \{V, F\}$ et $b : E \to \{V, F\}$ on dit que a est plus petite que b (que l'on note $a \leqslant b$) 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 $a \leqslant b$ se traduit par :

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

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

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

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

2. Montrer que si P, Q sont des formules croissantes, alors $P \wedge Q$ et $P \vee Q$ sont des formules croissantes.

3. Soit C une clause conjonctive satisfiable contenant au moins un littéral. Montrer qu'elle est croissante si et seulement si elle ne contient aucun littéral de la forme $\neg x$ avec $x \in E$.

4. On considère une formule propositionnelle P qui n'est ni une tautologie, ni une antilogie.

(a) Montrer que si P est logiquement équivalente à une disjonction de clauses conjonctives dont aucune ne contient un littéral de la forme $\neg x$ avec $x \in E$, alors P est une formule propositionnelle croissante.

(b) Réciproquement, montrer que si P est une formule propositionnelle croissante, alors elle est logiquement équivalente à une disjonction de clauses conjonctives dont aucune ne contient un littéral de la forme $\neg x$ avec $x \in E$.

Chapitre 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 :

$$S \to SaS \,|\, b$$

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

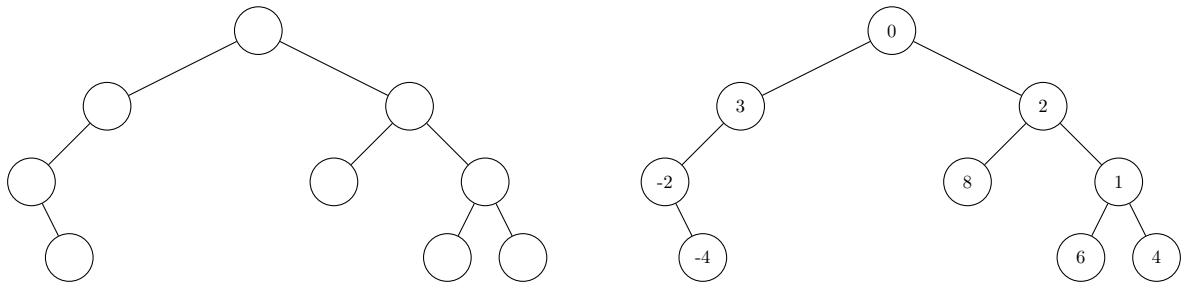
Chapitre 17

(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) :



(a) (b)

1. Déterminer le ou les minima locaux de l'arbre (b).
2. Donner une définition inductive permettant de définir les arbres binaires ainsi que la définition de la hauteur d'un arbre. Quelle est la hauteur de l'arbre (b) ?
3. Montrer que tout arbre non vide possède un minimum local.
4. Proposer un algorithme permettant de trouver un minimum local d'un arbre non vide et déterminer sa complexité.

On considère un arbre binaire non étiqueté que l'on souhaite étiqueter par des entiers relatifs distincts deux à deux de manière à maximiser le nombre de minima locaux de cet arbre.

5. Proposer sans justifier un étiquetage de l'arbre (a) qui maximise le nombre de minima locaux.
6. Proposer un algorithme qui, étant donné un arbre binaire non étiqueté en entrée, permet de calculer le nombre maximal de minima locaux qu'il est possible d'obtenir pour cet arbre. Déterminer la complexité de votre algorithme.
7. Montrer que, pour un arbre de taille $n \in \mathbb{N}$, le nombre maximal de minima locaux est majoré par $\left\lfloor \frac{2n+1}{3} \right\rfloor$. *On pourra remarquer que les nœuds non minimaux couvrent l'ensemble des arêtes de l'arbre.*

Chapitre 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 :
- $P(L) = \{a \in \Sigma \,|\, a\Sigma^\star \cap L \neq \varnothing\}$ l'ensemble des premières lettres des mots de L.
- $D(L) = \{a \in \Sigma \,|\, \Sigma^\star a \cap L \neq \varnothing\}$ l'ensemble des dernières lettres des mots de L.
- $F(L) = \{m \in \Sigma^2 \,|\, \Sigma^\star m \Sigma^\star \cap L \neq \varnothing\}$ 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 :

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

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 $a^\star(ab)^\star + aa$ sur l'alphabet $\{a, b\}$. Ce langage est-il local ? On vérifiera la cohérence entre les réponses à cette question et celles obtenues via les fonctions demandées dans la suite de l'énoncé.

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

> **Entrée :** Une expression régulière e sur un alphabet Σ ne faisant pas intervenir le symbole $\varnothing$.
> **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 $\varnothing$". 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`.

3. Implémenter la fonction `contains_epsilon`.

4. Sur le modèle de `compute_P`, implémenter une fonction `compute_D : regexp -> string list` permettant le calcul de l'ensemble $D(L)$ étant donnée une expression régulière dénotant le langage L.

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

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

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

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

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

Chapitre 19

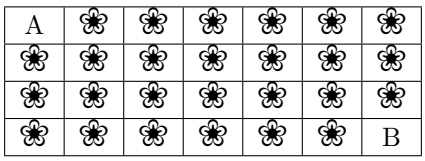
(CCINP) Programmation dynamique pour la récolte de fleurs * (CCINP 23, ex B, corrigé — oral - 204 lignes)

$*$

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).



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

1. On considère le champ suivant :

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

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

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

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

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

4. Dans quel ordre remplir le tableau `fleurs` de sorte à éviter de recalculer une valeur ?

5. Écrire une fonction de prototype `int recolte_iterative(int champ[m][n], int i, int j, int fleurs[m][n])` qui calcule, stocke dans `fleurs[i][j]` et retourne la cueillette maximale obtenue en parcourant le champ de A à la case (`i`, `j`).

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

6. Écrire la fonction de prototype `void 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`).

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

Chapitre 20

(CCINP) calculs avec les flottants * (CCINP 23, ex B, corrigé — oral - 207 lignes)

$*$

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

Chemins simples sans issue (type B)

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 normalisé la partie entière de la mantisse est 1 qui est un bit caché. En simple précision, le signe est codé sur 1 bit, la partie décimale de la mantisse sur 23 bits et l'exposant sur 8 bits. En double précision, le signe est codé sur 1 bit, la partie décimale de la mantisse sur 52 bits et l'exposant sur 11 bits.

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

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

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

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

   Le jury attend une explication faisant le lien entre la valeur trouvée et le format de représentation des flottants rappelé dans l'énoncé.
4. La limite théorique de la suite $(u_n)_{n\in\mathbb{N}}$ est 6. Compléter la fonction `main` afin d'afficher les 22 premiers termes de la suite. Vers quelle valeur semble tendre la suite ?
5. On définit une liste chaînée de nombres à l'aide d'une structure `nb` comportant un double et un pointeur vers une structure `nb` définie ci-dessous

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

   Écrire une fonction de signature `double somme(struct nb* tab)` qui calcule la somme des éléments de la liste `tab`.
6. L'algorithme suivant permet d'augmenter la précision du calcul lors du calcul d'une somme.

```
Entrées : Une liste l de réels triée dans l'ordre croissant
    de taille au moins 2.
Sorties : La somme des réels contenus dans la liste l.

TantQue la liste l contient strictement plus d'un élément
    Calculer la somme s = x + y des deux premiers éléments x
        et y de l
    Supprimer x et y de l
    Insérer s dans l de sorte à ce que l reste triée
Renvoyer l'unique élément de l
```

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

Chapitre 21

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

*

Graphes, Ocaml, Complexité, Backtracking,
sources : `ccinpchemins_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 $s_0, \cdots, s_n$ de sommets de V avec $n \geq 0$ et vérifiant $\forall i \in \{0, \cdots, n-1\}, (s_i, s_{i+1}) \in E$. On dit que ce chemin est *simple* si $s_0, \cdots, s_n$ 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 $\{s_0, \cdots, s_n\}$.
Dans la suite, les graphes considérés sont définis sur un ensemble de sommets de la forme $\{0, 1, \cdots, n-1\}$. Pour représenter un graphe en OCaml, on utilise le type suivant :

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

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

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

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

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

On rappelle différentes fonctions pouvant être utiles :

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

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

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

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

Chapitre 22

(CCINP) Grammaires pour des langages de programmation * (CCINP 24, ex A, corrigé, à débugguer — oral - 90 lignes)

$*$

Grammaires, Langages,
sources : `langccinp1.tex`

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

1. Quel est le langage engendré par cette grammaire ?

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

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

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

(1). $I \to E$;
(2). $I \to$;
(3). $I \to while(E)I$
(4). $I \to B$

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

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

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

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

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

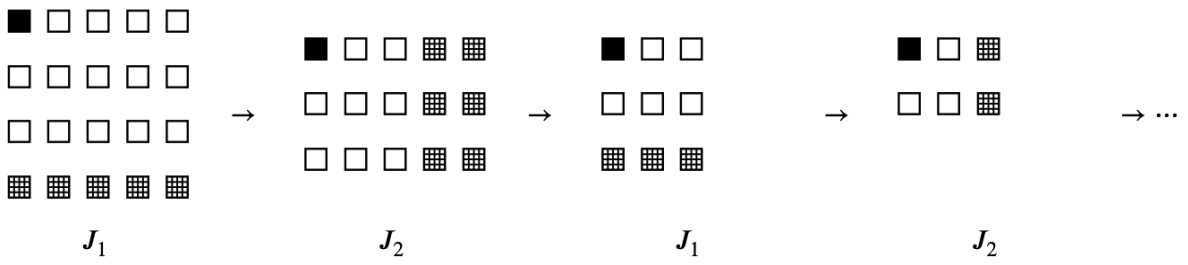
Chapitre 23

(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.



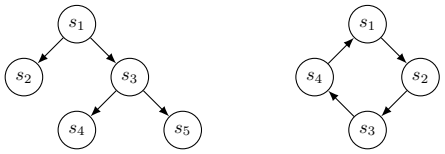
On associe à ce jeu un graphe orienté $G = (S, A)$. Les sommets S sont les états possibles de la tablette de chocolat, définis par un couple $s = (m, n), m \in [\![1, l]\!],\ n \in [\![1, c]\!]$. De plus, $(s_i, s_j) \in A$ si un des joueurs peut, par son choix de jeu, faire passer la tablette de l'état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

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

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

Utilisation des noyaux de graphe
Soit $G = (S, A)$ un graphe orienté. On dit que $N \subset S$ est un noyau de G si :
- pour tout sommet $s \in N$, les successeurs de s ne sont pas dans N,
- tout sommet $s \in S \setminus N$ possède au moins un successeur dans N

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



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

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

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

```
N = ∅
TantQu'il reste des sommets à traiter
    Chercher un sommet s ∈ S sans ↩
        successeur
    N = N ∪ {s}
    Supprimer s de G ainsi que ses préd↩
        écesseurs
```

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

Utilisation des attracteurs

On modélise le jeu par un graphe biparti : pour ce faire, on dédouble les sommets du graphe précédent : un sommet s_i génère donc deux sommets s_i^1 et s_i^2, s_i^j étant le sommet i contrôlé par le joueur J_j. On forme alors deux ensembles de sommets $S_1 = \{s_i^1\}_i$ et $S_2 = \{s_i^2\}_i$, et on construit le graphe de jeu orienté $G = (S, A)$ avec $S = S_1 \bigcup S_2$ et $S_1 \bigcap S_2 = \varnothing$. De plus, $(s_i^1, s_j^2) \in A$ si le joueur 1 peut, par son choix de jeu, faire passer la tablette de l'état s_i^1 à l'état s_j^2. On raisonne de même pour $(s_i^2, s_j^1) \in A$.
On rappelle la définition d'un attracteur : soit F_1 l'ensemble des positions finales gagnantes pour J_1. On définit alors la suite d'ensembles $(\mathcal{A}_i)_{i\in\mathbb{N}}$ par récurrence : $\mathcal{A}_0 = F_1$ et

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

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

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

Chapitre 24

(CCINP) relation d'équivalence * (CCINP 24, ex A, corrigé — oral - 139 lignes)

$*$

Algorithmique, Graphes,
sources : `ccinpreleq.tex`

Une relation d'équivalence (type A)

On fixe $n \in \mathbb{N}^\star$. Soient $\varphi : [\![1,n]\!] \to [\![1,n]\!]$ et $\psi : [\![1,n]\!] \to [\![1,n]\!]$. Soit u et v deux éléments de $[\![1,n]\!]$. On dit que u et v sont (φ,ψ)-équivalents s'il existe $k \in \mathbb{N}$, un tuple $(w_0, w_1, \cdots, w_{k+1}) \in [\![1,n]\!]^{k+2}$ avec $w_0 = u, w_{k+1} = v$ et vérifiant :

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

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 $[\![1,n]\!]$.

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 :

$$S = [\![1,n]\!], A = \{(x,y) \in S^2 | x \neq y \text{ et } (\varphi(x) = \varphi(y) \text{ ou } \psi(x) = \psi(y))\}.$$

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 $[\![1,n]\!]$ 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 $(S, \{(x,y) \in S^2 | x \neq y \text{ et } f(x) = f(y)\})$.

5. Préciser la forme du graphe G_f pour $f \in \{\varphi,\psi\}$.

6. Expliciter la forme des différentes classes d'équivalence dans le graphe G correspondant.

Chapitre 25

(CCINP) Fonctions pour la détection de motifs * (CCINP 24, ex B, corrigé — oral - 130 lignes)

$*$

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 capturé par ce motif.

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

2. Écrire une fonction `string_to_chaine : string -> chaine` prenant en entrée une chaîne de caractères et renvoyant la liste de caractères associée.

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

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

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

```
let rec capture motif texte = match (motif, texte) with
  | [], [] -> true
  | ['*'], [] -> true
  | _, [] -> false
  | [], _ -> false
  | ...
```

5. Montrer que le code partiel de cette approche naïve est correct.

6. Compléter la fonction `capture`.

Chapitre 26

(CCINP) HORNSAT * (CCINP 24, ex B, corrigé — 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 $\bot$) 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 ?
   a) $F_1 = (\neg x_0 \vee \neg x_1 \vee \neg x_3) \wedge (x_0 \vee \neg x_1) \wedge (\neg x_2 \vee x_0 \vee \neg x_3) \wedge (\neg x_0 \vee \neg x_3 \vee x_2) \wedge x_2 \wedge (\neg x_3 \vee \neg x_2)$.
   b) $F_2 = (x_0 \vee \neg x_1) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_0) \wedge \neg x_1 \wedge (x_1 \vee \neg x_1 \vee x_0) \wedge (\neg x_0 \vee x_2)$.
   c) $F_3 = (\neg x_1 \vee \neg x_4) \wedge x_1 \wedge (\neg x_0 \vee \neg x_3 \vee \neg x_4) \wedge (x_0 \vee \neg x_1) \wedge (x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_4 \vee \neg x_0 \vee \neg x_1)$.

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

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

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

On appelle *clause unitaire* une clause réduite à un littéral positif. Par ailleurs, *propager* une variable x_i dans une formule F sous FNC consiste à modifier F comme suit :
- Toute clause de F qui ne fait pas intervenir la variable x_i est conservée telle quelle.
- Toute clause de F qui fait intervenir le littéral x_i est supprimée entièrement.
- On supprime le littéral $\neg x_i$ de toutes les clauses de F qui font intervenir ce littéral.

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

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

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

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

5. Justifier que propager une variable dans une formule de Horn donne une formule de Horn. Écrire une fonction `propager : formule_horn -> int -> formule_horn` qui prend en entrée une formule de Horn F et un entier i et calcule la formule résultat de la propagation de x_i dans F.

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

7. Quelle est la complexité de votre algorithme en fonction de la taille de la formule en entrée ? Que peut-on dire des problèmes de décision `SAT` et `HORN-SAT` (dont la définition est la même que celle de `SAT` à ceci près que les formules considérées sont supposées être des formules de Horn) ?

8. On s'intéresse à présent à la correction de l'algorithme $\mathcal{A}$.

a) Si F est une clause de Horn sans clause unitaire ni clause vide, donner une valuation simple qui satisfait F.

b) On admet que si F est une formule de Horn faisant intervenir une clause unitaire x_i et F' est le résultat de la propagation de x_i dans F, alors que F est satisfiable si et seulement si F' est satisfiable. En déduire la correction de l'algorithme $\mathcal{A}$.

9. Expliquer comment on pourrait modifier les fonctions précédentes afin de déterminer une valuation satisfaisant une formule de Horn dans le cas où elle existe plutôt que de juste dire si elle est satisfiable ou non. On ne demande pas d'implémentation.

Chapitre 27

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

∗

Langages, C, Complexité,
sources : `ccinpdyck.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 :

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

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

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

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

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

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

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

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

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

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

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

Chapitre 28

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

***

Jeux, Graphes,
sources : `ensjeujeton.tex`

Jeu des jetons (info MP 2023) :

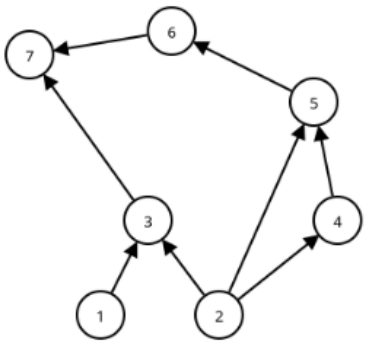
Définition 1

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 $L_n := x_1 \to x_2 \to \ldots \to x_n$, 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 $O\left(n^{1+\varepsilon}\right)$ étapes.

Chapitre 29

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

$***$

Langages,
sources : `ensmonoideslang.tex`

Monoïdes et Langages :

Définition 3

Un monoïde est un triplet $(M, \cdot_M, e_M)$ où $e_M \in M$ et $\cdot_M : (M \times M) \to M$ 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 $E := \left\{ j \in \mathbb{N} \mid \exists i \in [0, j-1], x^i = x^j \right\}$, est bien défini.
- **(b)** Montrer que les éléments $1, x, \ldots, x^{j_x - 1}$ 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 $G_x := \left\{ x^{i_x}, \ldots, x^{i_x + p_x - 1} \right\}$. 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 $\mu : \Sigma^* \to M$ un morphisme, i.e. $\mu(\varepsilon) = e_M$ et $\mu(uv) = \mu(u) \cdot_M \mu(v)$ pour tout $u, v \in \Sigma^*$, où uv est la concaténation des mots u et v.

6. Soient $m \in M$ et $w \in \Sigma^*$. Montrer que les deux assertions suivantes sont équivalentes :
- **(a)** $m \notin M\mu(w)M$
- **(b)** w se décompose soit en $w = uav$ où $m \notin M\mu(a)M$, soit en $w = uavbt$ où $m \in M\mu(av)M \cap M\mu(vb)M$ et $m \notin M\mu(avb)M$. (Où $u, v, t \in \Sigma^*$ et $a, b \in \Sigma$.)

7. Soient $m \in M \setminus \{e_M\}$ et $w \in \Sigma^*$. Montrer que les deux assertions suivantes sont équivalentes :
- **(a)** $\mu(w) \in mM$
- **(b)** Il existe $u, v \in \Sigma^*$ et $a \in \Sigma$ tels que $w = uav$ et $\mu(u) \notin mM$ et $\mu(ua)M \subseteq mM$.

8. Soit $m \in M \setminus \{e_M\}$. Montrer que $\mu^{-1}(m) = (U\Sigma^* \cap \Sigma^* V) \setminus (\Sigma^* Y \Sigma^*)$, où $\mu^{-1}(m) := \{w \in \Sigma^* \mid \mu(w) = m\}$ et

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

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

Chapitre 30

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

***

Logique propositionnelle,
sources : `enseliminationcutmll.tex`

Elimination des coupures dans MLL et MALL

La logique linéaire additive-multiplicative $MALL$ a pour syntaxe :
$A, B ::= p \,|p^\perp|\, A \otimes B | A \parr B | A \oplus B \mid A \& B$
avec p appartenant à un ensemble infini de *formules atomiques*.
L'opération de *dualité* $\cdot^\perp$ est définie inductivement sur les formules :
$(A \otimes B)^\perp = A^\perp \parr B^\perp \quad (A \oplus B)^\perp = A^\perp \& B^\perp$
$(A \parr B)^\perp = A^\perp \otimes B^\perp \quad (A \& B)^\perp = A^\perp \oplus B^\perp \quad \left(p^\perp\right)^\perp = p$
On note Γ, Δ des multi-ensembles $A_1, \ldots, A_n$ 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) :
$\dfrac{}{\vdash A, A^\perp}(\text{Ax}) \quad \dfrac{\vdash \Gamma, A^\perp \quad \vdash A, \Delta}{\vdash \Gamma, \Delta}(\text{Cut}) \quad \dfrac{\vdash \Gamma, A \quad \vdash B, \Delta}{\vdash \Gamma, A \otimes B, \Delta}(\otimes) \quad \dfrac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\ (\parr)$
$\dfrac{\vdash \Gamma, A \quad \vdash \Gamma, B}{\vdash \Gamma, A \& B}(\&) \quad \dfrac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B}\ (\oplus_1) \quad \dfrac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B}\ (\oplus_2)$
Une occurence 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 $\parr$, "plus" le $\oplus$ et "avec" le $\&$.
On définit la formule $A \multimap B$ comme $A^\perp \parr B$.

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

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

Informellement, on interprète la formule A comme "un exemplaire de la ressource A.
Question 3. Question supprimée.

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

Question 4. Soit 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.

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.

Le théorème d'*élimination des coupures* dans un système de preuve (un ensemble de règles) $\mathcal{L}$ contenant (`Cut`) est donné par :
Tout séquent $\vdash \Gamma$ *prouvable dans* $\mathcal{L}$ *est prouvable dans* $\mathcal{L}$ *privé de la règle* (`Cut`)

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

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

MLL est la logique obtenue à partir de MALL en n'utilisant que les formules atomiques, la dualité $^\perp$ et les connecteurs $\otimes$ et $\parr$, et seulement les règles (`Ax`), (`Cut`), ($\otimes$) et ($\parr$).
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`), ($\parr$), ($\otimes$), et les arêtes par des formules. Chaque noeud étiqueté par ($\otimes$) ou ($\parr$) a deux arêtes entrantes (étiquetées par des formules A et B) et une arête sortante (étiquetée par la respectivement $A \otimes B$ et $A \parr B$). Les noeuds étiquetés par (`Ax`) n'ont pas d'arêtes entrante et deux arêtes sortantes, étiquetées par une variable et sa négation. Les noeuds 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.

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 noeud étiqueté par $\parr$ dans $\mathcal{R}$, une des deux arêtes entrantes dans ce noeud, et en désorientant les arêtes restantes.
On admet le *théorème de Danos-Régnier* :
Un réseau $\mathcal{R}$ *est un réseau de preuve si et seulement si tout interrupteur de* $\mathcal{R}$ *est connexe et acyclique.*

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

Question 8.

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

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

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

Chapitre 31

(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 $V \to \mathbb{N}$. 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 : $G[W] = (W, \varnothing)$. On note $\alpha(G)$ la cardinalité de la plus grande anticlique de G.

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

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

a. Montrer qu'un coloriage valide de G avec c couleurs existe si et seulement si il existe une partition $\{A_1, \ldots, A_c\}$ de V en c anticliques.

b. Montrer $\chi(G)\alpha(G) \geqslant |V|$.

Graphe parfait. Un graphe G est dit *parfait* si tous ses sous-graphes induits $G[W]$ satisfont : $\chi(G[W]) = \omega(G[W])$.
Graphe imparfait minimal. Un graphe G est dit *imparfait minimal* s'il n'est pas parfait, et que tous ses sous-graphes induits propres sont parfaits.

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

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

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

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

Question 6. On considère une suite d'anticliques $A_0, \ldots, A_{\alpha\omega}$ définie comme dans la question précédente. Montrer que pour tout i dans $\{0, \ldots, \alpha\omega\}$, il existe une clique C_i telle que $C_i \cap A_i = \varnothing$, et $\forall j \neq i, |C_i \cap A_j| = 1$.
Indication : utiliser la question 4

Matrice d'incidence. Étant donné une suite $V_0, \ldots, V_{\alpha\omega}$ de sous-ensembles de $V = \{1, \ldots, n\}$, on définit la *matrice d'incidence* de la suite (V_i) comme la matrice $M = (M_{i,j})_{1 \leqslant i \leqslant n, 0 \leqslant j \leqslant \alpha\omega} \in \{0,1\}^{n \times (\alpha\omega+1)}$ 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 $A_0, \ldots, A_{\alpha\omega}$ de la question 5 , et M_C la matrice d'incidence de la suite $C_0, \ldots, C_{\alpha\omega}$ de la question 6. On note $M_A^\top$ la transposée de M_A.
Montrer que $M_A^\top M_C$ est de rang $\alpha\omega + 1$, où les matrices sont vues comme à coefficient dans $\mathbb{Q}$.

Question 8. Montrer $n \geqslant \alpha\omega + 1$.

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

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

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

Chapitre 32

(ENS) Inférence de type *** (ENS 23, corrigé — oral - 280 lignes)

***

Algorithmique, Langages fonctionnels, Pseudocode,
sources : `ensinferencetypes.tex`

Soit $\mathcal{A} = \{A, B, C, \ldots\}$ 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 $T_1 \to T_2 \in \mathcal{T}$. (la "flèche" de deux types est un type)

Une *substitution de types* $\sigma : \mathcal{A} \longrightarrow \mathcal{T}$ 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 $\mathcal{A}, \sigma_\uparrow(A) = \sigma(A)$.
- pour tout $T_1, T_2 \in \mathcal{T}^2, \sigma_\uparrow(T_1 \to T_2) = \sigma_\uparrow(T_1) \to \sigma_\uparrow(T_2)$

Un *système d'équations de types* (ou juste *système*) est un ensemble fini $\{(T_k, T'_k)\}_{1 \leqslant k \leqslant n}$ de couples de types.
Un *unificateur* d'un système $\{(T_k, T'_k)\}_{1 \leqslant k \leqslant n}$ est une substitution de types σ telle que $\forall k, \sigma_\uparrow(T_k) = \sigma_\uparrow(T'_k)$.

Par exemple, on pose $S_0 = \{(B \to A, C), (\mathbb{N}, B)\}$
et σ_0 définie par $\begin{cases} \sigma_0(B) = \mathbb{N} \\ \sigma_0(C) = \mathbb{N} \to A \\ \sigma_0(X) = X \text{ pour tout } X \notin \{B, C\} \end{cases}$
Alors σ_0 est un unificateur de S_0 car :

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

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

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

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

Quand un unificateur pour S est-il unique ?

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

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

Soit $\mathcal{X} = \{x, y, z, \ldots\}$ 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 $\varnothing$ 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 $\Gamma, y \neq x \Rightarrow \Gamma'(y) = \Gamma(y)$.

Dans la suite on s'intéresse au problème de donner un type de $\mathcal{T}$ à une expression de $E \in \mathcal{E}$ avec un contexte 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 $\varnothing$.

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

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

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

Chapitre 33

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

***

Algorithmique, Complexité,
sources : `ensordo.tex`

Ordonnancement :

Définition 7

On cherche à ordonnancer n tâches indépendantes $T_1, \ldots, T_n \in \mathcal{T}$ 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 $\sigma : \mathcal{T} \to \mathbb{N}$ 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, $\mathcal{I} \in \mathcal{P}(S)$.$(S, \mathcal{I})$ est un matroïde s'il satisfait les propriétés suivantes :
1. Hérédité : $X \in \mathcal{I} \Longrightarrow (\forall Y \subset X, Y \in \mathcal{I})$
2. Échange : $(A \in \mathcal{I}, B \in \mathcal{I}, |A| < |B|) \Longrightarrow \exists x \in B \backslash A$ 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 $x \in S \backslash F$ tel que $F \cup \{x\} \in \mathcal{I}$.

Un matroïde pondéré est un matroïde enrichi d'une fonction de poids $w : S \to \mathbb{N}$. 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,I), w : S → ℕ) ;
    S ← tri_décroissant (S, w)  ▷ Ainsi : w(S[0]) ⩾ ... ⩾ w(S[n − 1]) ;
    A ← ∅ ;
    Pour i = 0 à n − 1 {
        Si A ∪ {S[i]} ∈ I {
            A ← A ∪ {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'ordonnancement de la question 3.

Chapitre 34

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

***

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

Permutations triables par pile :

Définition 10

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

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

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

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

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

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

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

Permutation triable par pile. Une permutation $\pi \in \mathfrak{P}(n)$ est dite *triable par pile* s'il existe une machine à pile qui prend en entrée $(\pi(1),\dots,\pi(n))$, et qui renvoie $(1,\dots,n)$.
Exemple : l'égalité $\text{EEDEDD}(3,1,2) = (1,2,3)$ plus haut montre que $(3,1,2)$ est triable par pile.

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

Définition 11

Motif. On dit que $\pi \in \mathfrak{P}(n)$ *contient le motif* $\rho \in \mathfrak{P}(k)$ pour $k \leqslant n$ s'il existe $x_1 < \cdots < x_k$ 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 ($\varnothing, \varnothing$), 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 $O\left(|V|^3\right)$.

Chapitre 35

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

***

Graphes, Logique,
sources : `ensensembles.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 $\varnothing$, ou $\mathbb{N}$. Par abus de notation, $\varnothing$ et $\mathbb{N}$ représentent ici respectivement les ensembles vide et plein.

Attention : dans la suite, par convention, les lettres $A, B, C, D \ldots$ désigneront des éléments de $\mathcal{T}$, tandis que les lettres X, Y, Z désigneront des éléments de $\mathcal{T} \cup \{\varnothing, \mathbb{N}\}$.

Une conjonction d'inclusions sur les variables de $\mathcal{T}$ est dite *satisfiable* s'il existe une *valuation* $f : \mathcal{T} \to \mathcal{P}(\mathbb{N})$ des variables qui vérifie toutes les inclusions.

Pour deux formules Φ et Ψ, $\Phi \vDash \Psi$ signifie que toute valuation satisfaisant Φ satisfait également Ψ.

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

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

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

Un *chemin* dans G_Φ est une séquence de sommets $Z_1, \ldots, Z_\ell$ reliés par des arêtes $Z_1 \to Z_2, Z_2 \to Z_3, \ldots, Z_{\ell-1} \to Z_\ell$. 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 $\varnothing$
   (c) Φ est satisfiable à valeurs dans $\{\varnothing, \mathbb{N}\}$
4. Montrer que si Φ est satisfiable et X et Y sont des sommets de G_Φ, alors $\Phi \vDash (X \subseteq Y)$ si et seulement s'il existe un chemin dans G_Φ menant de X à Y.
5. En déduire un algorithme `Résolution` qui prend en entrée une conjonction d'inclusions Φ et une inclusion $(X \subseteq Y)$, et détermine si $\Phi \vDash (X \subseteq Y)$. 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 $\mathcal{T}$.

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 :

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

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 :

$$\begin{array}{l} \neg\,(A_3 \subseteq \varnothing) \\ \neg\,(A_1 \subseteq A_3) \\ \neg\,(A_2 \subseteq A_3) \vee \neg\,(A_1 \subseteq A_2) \\ (\mathbb{N} \subseteq A_1) \\ (A_2 \subseteq \varnothing) \end{array}$$

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

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., $f(X) \nsubseteq f(Y)$ et $f(Y) \nsubseteq f(X)$). »

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 : $\Psi = (X_1 \subseteq Y_1) \wedge \ldots \wedge (X_k \subseteq Y_k)$.

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.

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

9. Prouver la propriété (P).

Chapitre 36

(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.
Graphe. 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.
Graphe d'amis. Un **graphe d'amis** est un graphe $G = (V, E)$ tel que $|V| \geqslant 2$ et pour tout $x, y \in V$ avec $x \neq y$, il existe un unique sommet, noté $x \star y$, tel que : $\{x, x \star y\} \in E$ et $\{y, x \star y\} \in E$.
Diplomate. Étant donné un graphe $G = (V, E)$, un diplomate est un sommet de degré $|V| - 1$.

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

Question 1.
a. Montrer qu'un graphe d'amis $G = (V, E)$ ne contient pas de 4-cycle, c'est-à-dire :

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

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

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

a. Montrer $\{y, z\} \in E$.
b. Montrer $V = N(y) \cup N(z)$.
c. Montrer que y ou z est un diplomate.

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

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

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

$$\begin{aligned} f_\star : X \times Y &\to W \\ (a, b) &\mapsto a \star b. \end{aligned}$$

b. Montrer que tous les sommets de G on le même degré (on dit que G est régulier).
c. On note δ le degré d'un sommet de G. Montrer $n = \delta^2 - \delta + 1$.

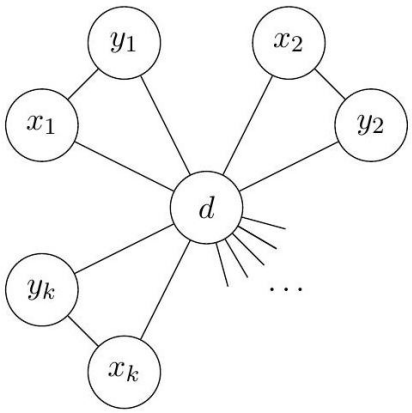
Matrice d'adjacence. La *matrice d'adjacence* de G est la matrice $M_{i,j\in\{1,\dots,n\}} \in \mathbb{R}^{n\times n}$ définie par $M_{i,j} = 1$ si $\{i, j\} \in E, 0$ sinon.

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

On admet (ou rappelle) les faits suivants.
- Toute matrice symétrique réelle est diagonalisable.
- La trace d'une matrice est invariante par changement de base.
- **Si $k \in \mathbb{N}$ et $\sqrt{k} \in \mathbb{Q}$, alors $\sqrt{k} \in \mathbb{N}$ (lemme de Dirichlet).**
- Les valeurs propres de $M^2 = \mathbb{1}_n + (\delta - 1)\,\mathrm{Id}_n$ 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.

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



Graphe moulin

Chapitre 37

(ENS) Structures pliables et traversables *** (ENS 23, partiellement corrigé — oral - 306 lignes)

***

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

Structures pliables 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 $A \to A$;
- d'un opérateur (o) : $(A \to B) \to (B \to C) \to (A \to C)$ 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` $A \to$ `int` en pseudocode (par exemple) :

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

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

et en OCaml :

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

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

Un type T est un *monoïde*, s'il existe (c'est-à-dire, si on peut définir) ε de type T et ($\diamond$) de type $T \to T \to T$, 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.

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

$$(A \to B \to B) \to B \to CA \to B$$

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

de type B.

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

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

Question 4. Utiliser `fold` 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`.

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

$$(A \to M) \to CA \to M$$

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

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

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

Un foncteur applicatif est un constructeur de type F pour lequel il existe pour tous types A, B :
1. `fmap` de type $(A \to B) \to FA \to FB$
2. `pure` de type $A \to FA$
3. $(\odot)$ de type $F(A \to B) \to FA \to FB$, noté $\odot$ dans sa version infixe.

qui vérifie les lois suivantes :
`fmap id = id` $\quad$ `fmap`$(f \circ g) =$ (`fmap` f) $\circ$ (`fmap` g)
`(pure id)` $\odot v = v$

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

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

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

pour tout foncteur applicatif F et types A et B.

Question 8. Montrer que `barbre` est traversable.

Question 9. On dispose d'un arbre t de type `barbre` A et une fonction futurs qui à un élément A associe un 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 38

(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 $A \to A$;
- d'un opérateur (o) : $(A \to B) \to (B \to C) \to (A \to C)$ de composition de fonctions, noté o 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` $A \to$ int en pseudocode :

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

et en *OCaml* :

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

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

Un constructeur de types `F` est un *foncteur* quand il existe une fonction polymorphe
`fmap` : $(A \to B) \to (\mathtt{F}A) \to (\mathtt{F}B)$

qui vérifie les lois suivantes (pour tout f et g) :
`fmap id` $=$ `id`
`fmap` $(f \circ g) =$ (`fmap` $f) \circ$ (`fmap` g)

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

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

Un foncteur `M` est une *monade* quand il existe deux fonctions polymorphes
`return` : A $\to$ (M A)
`bind` : (M A) $\to$ (A $\to$ (M B)) $\to$ (M B)
qui vérifient les lois suivantes, pour tout f, g et x :
`bind` (`return` x) $f = f\ x$
`bind` x `return` $= x$
`bind` (`bind` $x\ f$) $g =$ `bind` $x(y \mapsto$(`bind` (f y) g))

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

Pour tout type S, on définit (etat S) A comme étant le type polymorphe $S \to (S, A)$
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` $\to$ `memoire` $\to$ (`option int`)
`majour : string` $\to$ `int` $\to$ `memoire` $\to$ `memoire`
La première recherche la valeur associée à une clef, la seconde met-à-jour un couple clef-valeur.

Question 4. Montrer que `etat` S est une monade pour tout S.
En déduire l'écriture avec `bind` d'une *procédure* qui prend en entrée trois clefs, récupère successivement deux valeurs d'une mémoire à partir des deux premières clefs puis associe dans la mémoire la somme des deux valeurs récupérée à la troisième clef.

On donne une définition alternative des monades :
Un foncteur `M` est une *monade* quand il existe deux fonctions polymorphes

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

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

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

Question 5. Montrer que les deux définitions sont équivalentes.

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

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

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

Chapitre 39

(ENS) Déduction de messages *** (ENS 24, corrigé — oral - 232 lignes)

***

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

Déduction 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 : $t, t_1, t_2 := v \in \mathcal{C} | x \in \mathcal{V} | f(t_1, t_2)$.
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 $t_1, \ldots, t_n \in S$.

Règle d'inférence Une règle d'inférence est une règle de déduction de la forme :

$$\frac{T \vdash t_1 \ldots \quad T \vdash t_n}{T \vdash t} \text{ REGLE}$$

où T est un ensemble fini de termes et $t_1, \ldots, t_n, t$ 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 $v_1, \ldots, v_n$ il existe une règle d'inférence dans $\mathcal{I}$ ayant pour conclusion v_0 et hypothèses $v_1, \ldots, v_n$ (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.

$$\frac{\text{si } u \in T}{T \vdash u} \text{Ax}$$

$$\frac{T \vdash x \quad T \vdash y}{T \vdash f(x,y)} \text{ APP-F} \qquad \frac{T \vdash f(x,y) \quad T \vdash y}{T \vdash x} \text{ RED-F}$$

Figure 1 - Système d'inférence $\mathcal{I}_0$

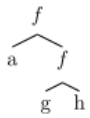


Figure 2 - Représentation sous forme d'arbre du terme $f(a, f(g, h))$

Question 1. Soit $T = \{f\left(f\left(a, k_1\right), k_2\right), k_2, f\left(k_1, k_2\right)\}$. Donner l'arbre de preuve de $T \vdash a$ dans $\mathcal{I}_0$.

Question 2. Soit T un ensemble de termes clos. Montrer que pour tout $T \vdash u$ dans $\mathcal{I}_0$, un arbre de preuve de taille minimale Π de $T \vdash u$ contient seulement des termes issus de $\mathtt{st}(T \cup \{u\})$, i.e., $\mathtt{Termes}(\Pi) \subseteq \mathtt{st}(T \cup \{u\})$.
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 $st(T)$, i.e. $\mathtt{Termes}(\Pi) \subseteq st(T)$.

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 : $\mathtt{size}(T, u) = |st(u)| + \Sigma_{t \in T}|\mathtt{st}(t)|$.

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 : $C = (\mathtt{true} \Rightarrow x)$
- un littéral positif et au moins un littéral négatif : $C = (x_1 \wedge \ldots \wedge x_n \Rightarrow x)$
- aucun littéral positif : $C = (x_1 \wedge \ldots \wedge x_n \Rightarrow \text{ false })$.

On admettra que `HORN-SAT` est P-complet, c'est-à-dire (intuitivement) que tout problème de décision dans P admet une réduction linéaire à `HORN-SAT`.

Question 4. Montrer que le problème de déduction dans $\mathcal{I}_0$ est P-complet.

Nous souhaiterions maintenant nous intéresser au même problème mais en ajoutant le ou-exclusif. Un terme est donc maintenant généré par la grammaire :

$$t, t_1, t_2 := v \in \mathcal{C} | x \in \mathcal{V} | f\left(t_1, t_2\right) \mid t_1 \oplus t_2$$

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' ?

$$\frac{T \vdash u_1 \ldots \quad T \vdash u_n}{T \vdash u_1 \oplus \ldots \oplus u_n}\text{GX} \qquad \frac{\text{si } u =_{AC} v \text{ et } v \in T}{T \vdash u}\text{Ax}$$

On note $=_{AC}$ la plus petite relation telle que :

$$\begin{array}{lll} \text{(refl.) } x = x & \text{(sym.) } (x = y) \Rightarrow (y = x) & \text{(trans.) } (x = y) \wedge (y = z) \Rightarrow (x = z) \\ \text{(comm.) } x \oplus y = x \oplus y & \text{(assoc.) } x \oplus (y \oplus z) = (x \oplus y) \oplus z & \end{array}$$

$$\text{(congr.) } (x_1 = y_1) \wedge (x_2 = y_2) \Rightarrow f\left(x_1, x_2\right) = f\left(y_1, y_2\right)$$

Question 5. Soit u et v deux termes clos. Donner un algorithme en temps polynomial qui décide si $u =_{AC} v$.

Question 6. Soit T un ensemble de termes clos. Soit t un terme clos.
Montrer que $T \vdash t$ dans $\{\text{GX}, \text{Ax }'\}$ est décidable en temps polynomial.

Chapitre 40

(ENS) Filtrage par motif en OCaml *** (ENS 24, corrigé — oral - 323 lignes)

$***$

Ocaml,
sources : `ensfiltrageOcaml.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 $C\,(e_1, \ldots, e_k)$, 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 $v ::= C\,(v_1, \ldots, v_k)$. En particulier, les constantes `true` et `false` sont des valeurs du `type bool = true | false`.

Les motifs sont $m ::= \mathrm{v} \,|\_\, |C\,(m_1, \ldots, m_k)|\,(m_1 \mid m_2)$, 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.

$$\begin{array}{l}\texttt{match}\ (\mathrm{e}_1, \ldots, \mathrm{e}_m\ )\ \texttt{with} \\ \mid \mathrm{m}_{1,1}, \ldots,\ \mathrm{m}_{1,m} \to \mathrm{a}_1 \\ \mid \ldots \\ \mid \mathrm{m}_{n,1}, \ldots,\ \mathrm{m}_{n,m} \to \mathrm{a}_n\end{array} \quad \to \quad \begin{pmatrix} e_1 & \ldots & e_m & & \\ m_{1,1} & \ldots & m_{1,m} & \to & a_1 \\ \ldots & \ldots & \ldots & \to & \ldots \\ m_{n,1} & \ldots & m_{n,m} & \to & a_n \end{pmatrix}$$

Étant donné une expression $e = C\,(e_1, \ldots, e_k)$, on définit des opérateurs d'accès au constructeur : $\texttt{constr}(e) = C$ et d'accès au i-ème champ : $\#i(e) = e_i($ si $1 \leqslant i \leqslant k)$.
Un *environnement* est une fonction partielle des variables aux valeurs.

Question 1. Définir une relation $m \leqslant_\sigma v$ établissant la compatibilité entre un motif m et une valeur v, étant donné un environnement σ.
À titre d'exemple, `Cons` $(1, x) \leqslant_\sigma$ `Cons`$(1,$ `Cons`$(2,$ `Empty` $))$ lorsque $\sigma(x) =$ `Cons`$(2,$ `Empty` $)$.

Dans la suite, on note $m \leqslant v$ si et seulement si $\exists \sigma, m \leqslant_\sigma v$.

Question 2. On note $Match\left((v_1, \ldots, v_m), M\right)$ le résultat du filtrage de la matrice M sur le vecteur de valeurs $(v_1, \ldots, v_m)$.

(a) Soit a_i une action et σ un environnement. Sous quelle(s) condition(s) $Match\left((v_1, \ldots, v_m), M\right) = (a_i, \sigma)$?

(b) Y a-t-il d'autres cas de définition de Match ?

Question 3.

(a) Décrire informellement comment éliminer le motif joker _ sans ajouter de cas, sur l'exemple de len 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.

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.

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 -> ...
```

Question 6.

(a) Cette transformation a-t-elle un intérêt ?

(b) L'algorithme de transformation peut-il être amélioré ?

Question 7. Prouver la terminaison et la correction de l'algorithme en question 5.

Chapitre 41

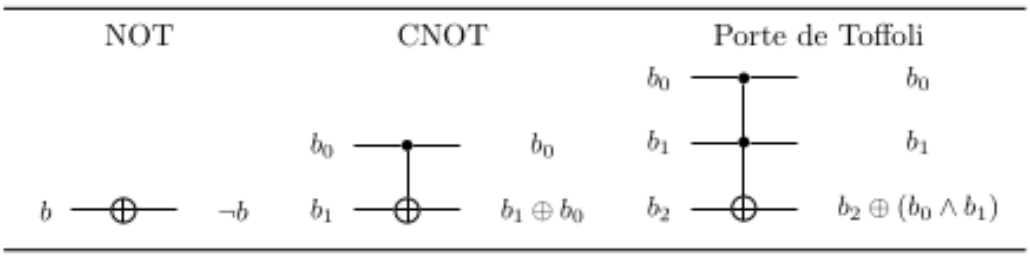
(ENS) Implémentation des circuits réversibles *** (ENS 24, corrigé — oral - 218 lignes)

***

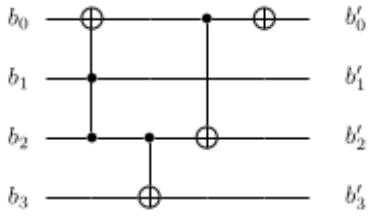
Circuits,
sources : `enscircuitsreversibles.tex`

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 $f_{\mathcal{C}} : \{0,1\}^n \to \{0,1\}^n$, obtenue en composant les fonctions de chaque porte logique, dans l'ordre de lecture de gauche à droite. On dit que $\mathcal{C}$ implémente la fonction $f_{\mathcal{C}}$. Ainsi, le circuit suivant :



implémente la fonction :

$$(b_0, b_1, b_2, b_3) \mapsto (\neg(b_0 \oplus (b_1 \wedge b_2)), b_1, b_2 \oplus b_0 \oplus (b_1 \wedge b_2), b_3 \oplus b_2)$$

Deux circuits réversibles $\mathcal{C}_1, \mathcal{C}_2$ sur n bits peuvent être composés en écrivant les portes de $\mathcal{C}_2$ suivies des portes de $\mathcal{C}_1$; on implémente alors la fonction $f_{\mathcal{C}_2} \circ f_{\mathcal{C}_1}$.

Question 1.

1. Montrer que tout circuit réversible $\mathcal{C}$ constitué d'une seule porte (NOT, CNOT ou Toffoli) implémente une permutation. Quel est son inverse ?

2. Comment implémenter l'inverse d'un circuit réversible quelconque ?

Une *porte de Toffoli à k contrôles* implémente la fonction :

$$(b_0, \dots, b_{k-1}, b_k) \mapsto \left(b_0, \dots, b_{k-1}, b_k \oplus \bigwedge_{i=0}^{k-1} b_i \right)$$

Question 2. Montrer que dans un circuit sur n bits, une porte de Toffoli à $n-2$ contrôles peut être implémentée à l'aide de 2 portes de Toffoli à $n-3$ contrôles, et deux portes de Toffoli.

Dans la suite de cet exercice, on va démontrer le théorème suivant :

Pour tout $n \geqslant 7$, une permutation Π de $\{0,1\}^n$ est paire si et seulement s'il existe un circuit réversible $\mathcal{C}$ sur n bits tel que $\Pi = f_{\mathcal{C}}$.

On rappelle qu'une permutation est paire (de signature 1) si et seulement si toutes ses décompositions en transpositions ont un nombre de transpositions pair.

Question 3. Trouver des contre-exemples simples pour $n = 2$ et $n = 3$.

Question 4. Soit $\mathcal{C}$ un circuit réversible sur $n \geqslant 4$ 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 $S_0 \to S_1$ préservant la longueur des cycles.
2. En déduire une implication du théorème.

Question 5.

1. Soit $n \geqslant 4$. 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 $\Pi = f_{\mathcal{C}}$.

Question 6. Montrer que pour $n \geqslant 6$, toute permutation paire de $\{0,1\}^n$ peut s'écrire comme une composition de paires de transpositions disjointes, c'est-à-dire :

$$\Pi = (x_0 y_0)\,(z_0 t_0)\,(x_1 y_1)\,(z_1 t_1) \dots (x_k y_k)\,(z_k t_k)$$

où pour tout i, x_i, y_i, z_i, t_i sont distincts deux à deux (mais on peut avoir par exemple $x_0 = y_1$).

Question 7. Soit $n \geqslant 7$ et soit x, y, z, t des éléments de $\{0,1\}^n$ distincts deux à deux. Montrer qu'il existe un circuit réversible $\mathcal{C}$ implémentant une permutation P telle que :

$$P(x) = (0, 0, 1, \dots, 1), \quad P(y) = (0, 1, 1, \dots, 1), \quad P(z) = (1, 0, 1, \dots, 1), \quad P(t) = (1, 1, 1, \dots, 1)$$

En déduire un circuit réversible qui implémente la paire de transpositions $(xy)(zt)$.

Chapitre 42

(ENS) Résolution d'inéquations linéaires *** (ENS 24, corrigé — oral - 195 lignes)

$***$

Ocaml, Algorithmique, Complexité,
sources : `ensresolutioninequationslineaires.tex`

Résolution d'inéquations linéaires

Soit $X = \{x_1, \ldots, x_n\}$ un ensemble de variables et $S = \{L_1, \ldots, L_m\}$ un ensemble de fonctions linéaires en les variables de X à coefficients rationnels, assimilé à un système d'inéquations :

$$\forall i, L_i(x_1, \ldots, x_n) \leqslant 0$$

Si deux équations peuvent se réécrire sous la forme : $L \leqslant x$ et $x \leqslant U$ 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 : $L \leqslant U$ (ou de manière équivalente : $L - U \leqslant 0$).
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 $X \to \mathbb{Q}$ satisfaisant toutes les inéquations de S.

Question 1. Le système suivant :

$$\begin{cases} 2x_1 + x_2 + 1 \leqslant 0 \\ 2x_2 \leqslant 0 \\ -x_1 \leqslant 0 \\ -x_2 \leqslant 0 \end{cases}$$

est-il satisfiable ?

Soit S un système d'inéquations linéaires, x une variable de S, et $T \subseteq S$ l'ensemble des inéquations ne contenant pas x. Soit S' l'ensemble des x-résultants de paires d'inéquations dans $S \backslash T$. On définit un nouveau système $FM(S, x) := T \cup S'$. L'opération FM est appelée l'*élimination de Fourier-Motzkin.*

Question 2. Soit S un système d'inéquations linéaires et x une variable de S. Montrer que S est satisfiable si et seulement si $FM(S, x)$ l'est.

Posons maintenant : $y := \max_i L_i(y_1, \ldots, y_{n-1})$. Alors toutes les inéquations $L_i \leqslant x$ sont satisfaites par $y, y_1, \ldots, y_{n-1}$ par définition de y. De plus pour tout $j, y \leqslant U_j(y_1, \ldots, y_{n-1})$. En effet, sinon il existerait i et j tels que $L_i(y_1, \ldots, y_{n-1}) > U_j(y_1, \ldots, y_{n-1})$ ce qui contredirait l'hypothèse.

S'il n'existe pas d'inéquation de la forme $L_i \leqslant x$ (respectivement, de la forme $x \leqslant U_j$) alors on choisit $y := \min_j U_j(y_1, \ldots, y_{n-1})$ (respectivement, le même y que précédemment).

Question 3. En déduire un algorithme pour la satisfiabilité des inéquations rationnelles (il n'est pas nécessaire d'en faire une description détaillée). Donner une borne simple sur sa complexité en temps.

Dans la suite de l'exercice, on considère des inéquations à au plus deux variables. On s'intéresse à l'algorithme suivant.

```
Entrée : S un système de m inéquations, contenant n variables au total
pour i=1 à ⌈log_2 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.

Question 5. Soit S un système insatisfiable minimal sur un ensemble de $k > 3$ variables X, et soit $S' := \cup_{x \in X} FM(S, x)$.

1. Montrer qu'il existe un ensemble $X' \subseteq X$ de $\lceil k/2 \rceil - 1$ variables telles qu'aucune paire $\{x_1, x_2\}$ de X' n'apparaît dans une inéquation de S.
2. Montrer que $(S \cup S')_{X \setminus X'}$ est insatisfiable.

Question 6. Soit $f(x) = \lfloor x/2 \rfloor + 1$. On pose $f^0(n) = n$ et on admet que pour tout entier naturel n, $f^{\lceil \log_2 n \rceil}(n) = 2$.

1. Soit S un système insatisfiable en entrée de l'algorithme. Montrer que pour tout $k \geqslant 1$, à la sortie de la k-ième itération de la boucle, il existe un sous-ensemble $X_k \subseteq X$ de taille $\leqslant f^k(n)$ tel que S_{X_k} est insatisfiable.
2. En déduire que l'algorithme est correct.
3. Donner une borne simple sur sa complexité (à facteur polynomial près).

On s'autorise à modifier l'algorithme en ajoutant entre l'étape 4 et 5 : $S \leftarrow$ `Simplifie`(S), où Simplifie est une opération transformant S en un système équivalent avec moins d'inéquations.

Question 7. On considère maintenant le cas particulier où les coefficients des équations sont $+1, -1$ ou 0 . En utilisant un choix opportun pour Simplifie, proposer un algorithme en temps polynomial en n.

Chapitre 43

(ENS) Mots partiels et Théorème de Dilworth *** (ENS 24, partiellement corrigé, niveau inapproprié, à débugguer — oral - 184 lignes)

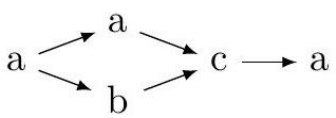
***

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 $x_0, x_1, \cdots, x_n$ de sommets tels que pour tout $0 \leqslant i < n$, $(x_i, x_{i+1}) \in R$. 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 $\mu : E \to \Sigma$. 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 $u_1 \cdots u_n \in \Sigma^*$ *généralise* p s'il existe $\psi : E \to \{0, \ldots, n-1\}$, 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 `Total`(p) l'ensemble des mots qui généralisent p.

Question 1. Que vaut `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 ?

Question 2. Donnez une borne supérieure sur la taille de `Total`(p) en fonction de la taille de Σ.

Soit L un langage de Σ^*. Un mot partiel p sur Σ *appartient* à `Partiel`(L) s'il existe $v \in$ `Total`(p) qui appartient à L.

Question 3. Montrez que si L est dans $\mathbf{P}$ [1], alors `Partiel`(L) appartient à $\mathbf{NP}$ [2].

1. *ND M.Péchaud* i.e. s'il existe un algorithme en temps polynomial déterminant si un mot est dans L
2. *ND M.Péchaud* i.e. s'il existe un certificat de taille polynomiale en un mot partiel p et un vérifieur en temps polynomial prenant en argument ce certificat et p et vérifiant si p est dans `Partiel`(L)

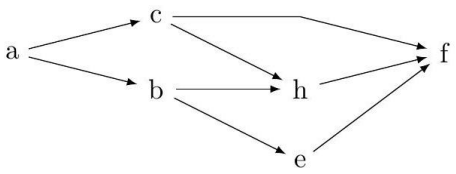
(Unary) 3-partition est un problème qui prend en entrée une suite finie de $3n$ entiers écrits **en unaire**[3] $x_1, \ldots, x_{3n}$ et vérifie s'il existe une partition en n triplets dont les sommes deux à deux sont égales. On admet dans la suite que ce problème est **NP**-complet.

Question 4. Montrez par une réduction depuis 3-partition qu'il existe un langage L dans **P** tel que Partiel(L) est **NP**-complet[4].

Question 5. Montrez que `Partiel` $(\Sigma^* ab^* a \Sigma^*)$ est vérifiable 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 :



Question 7. Soit L un langage régulier (reconnu par un automate déterministe). Montrez que Partiel (L) est vérifiable en temps $O(n^w)$ avec n la taille du mot partiel et w la taille de sa plus petite décomposition en chemins.

Soit $p = (E, R, \mu)$ un mot partiel sur 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 Partiel(L) pour L un langage régulier est calculable en temps polynomial. Peut-on en déduire que le problème est calculable en temps polynomial, sans la contrainte que la largeur est fixée ?

4. i.e. 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 Partiel(L).

Chapitre 44

(ENS) Terminaison de lambda ref *** (ENS 24, partiellement corrigé, niveau surréaliste — oral - 238 lignes)

***

Ocaml,
sources : `ensterminaison_lambda_ref.tex`

Terminaison de $\lambda_{\mathbf{ref}}$

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, `fun x -> 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* $\equiv_\alpha$ autorise le renommage des variables liées : si $y \notin M$, alors $\lambda x.M \equiv_\alpha \lambda y.(M\{x/y\})$.
Dans la suite, on considèrera les termes *modulo 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. $(\lambda x \cdot M)N \longrightarrow M\{N/x\}$ (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 $\lambda x.(\lambda y \cdot M)$
$\longrightarrow^*$ désigne la clôture réflexive et transitive de $\longrightarrow$. Les *réduits* d'un terme M sont les éléments de l'ensemble $\{M' \mid M \longrightarrow^* M'\}$.

Question 1. Soit $\delta = \lambda x.(xx)$ et $I = \lambda x.x$. Expliciter les réduits du terme $A = (\lambda x.I)(\delta\delta)$.

On donne un système de *types simples* au λ-calcul. Les types sont donnés par :
$S, T ::= S \to T \mid \texttt{unit}$

Ainsi un type est soit le type fonctionnel $S \to T$ des fonctions de S dans T soit `unit` le type de base.
Une *hypothèse* $x : T$ associe une variable de terme à un type. Un *contexte de typage* 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. $\Gamma \vdash () :$ `unit`
2. $\Gamma, x : T \vdash x : T$
3. si $\Gamma, x : T_1 \vdash M : T_2$ alors $\Gamma \vdash \lambda x.M : T_1 \to T_2$
4. si $\Gamma \vdash M : T_1 \to T_2$ 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.

On dispose d'un ensemble infini d'adresses $\mathcal{A}$. Une mémoire σ est une fonction qui a 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 `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 `unit`.

Question 3. Donner des règles de typage pour les termes de λ_{ref} et les mémoires.

Question 4. Donner une sémantique pour λ_{ref} explicitant comment réduire un couple composé d'un terme typable et d'une mémoire.

Les réductions qui consomment un opérateur `ref`, `deref` ou `assig` sont appelée *impures*, les autres *pures*.

Question 5. Définir une fonction d'élagage 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 $\mathbf{p}(M) \longrightarrow \mathbf{p}\,(M')$.

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.

On divise la mémoire en *régions* identifiée par des entiers naturels. Les opérateurs $\texttt{ref}_n$, $\texttt{deref}_n$ et $\texttt{assign}_n$ 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 45

(MT) Déduction Naturelle * (MT 0, ex 1, corrigé — oral - 200 lignes)

$*$

Déduction 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 $\vdash A \vee \neg A$.

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.

$$\frac{}{\Gamma, A \vdash A}\ \text{ax} \qquad \frac{\Gamma \vdash B}{\Gamma, A \vdash B}\ \text{aff} \qquad \frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\ \text{RAA}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to_i \qquad \frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \to_e$$

$$\frac{\Gamma \vdash A \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge_i \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge_e^g \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge_e^d$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee_i^g \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee_i^d \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ \vee_e$$

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg_i \qquad \frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash \bot}\ \neg_e$$

Chapitre 46

(MT) Relations entre les nombres de sommets et d'arêtes de grphes * (MT 0, ex 1, corrigé — oral - 102 lignes)

$*$

Graphes,
sources : `mtgraph1.tex`

Exercice 1

Soit G un graphe non-orienté à $n \geqslant 1$ sommets et p arêtes.

1. Montrer que si G est connexe alors $p \geqslant n-1$.
2. Montrer que si G est acyclique alors il possède un sommet de degré au plus 1.
3. Montrer que si G est acyclique alors $p \leqslant n-1$.
4. Montrer que les trois propriétés suivantes sont équivalentes :
(i) G est connexe et acyclique.
(ii) G est connexe et $p = n-1$.
(iii) G est acyclique et $p = n-1$.

Chapitre 47

(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_addresse : `int`, nom : `text`, prenom : `text`)
Adresses(id : `int`, ville : `text`, nom_rue : `text`, numero : `int`)
Commandes(id_client : `int`, plat : `text`, prix : `int`, date : `date`)

L'entreprise souhaite ajouter une fonctionnalité pour vérifier qu'un client n'est pas allergique à un plat qu'il commande. Pour cela, il est nécessaire de pouvoir enregistrer les allergènes présents dans chaque plat ainsi que les allergies de chaque client.

1. Proposer des modifications à notre schéma de relation pour pouvoir ajouter ces informations. Justifier pourquoi votre choix convient.

Dans la suite, on se base dans le modèle tel que vous l'avez modifié.

2. Écrire une requête pour trouver les noms et prénoms des clients étant allergiques à au moins un ingrédient d'une pizza.
3. Écrire une requête pour trouver les trois villes où le plus d'argent est dépensé, ainsi que cette somme totale.
4. Écrire une requête pour effectuer la moyenne d'argent dépensé en fonction des prénoms des clients.

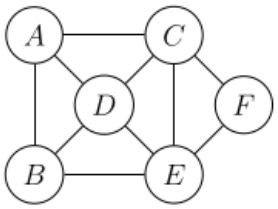
Chapitre 48

(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.

Chapitre 49

(MT) logique, cours * (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 ?
2. Mettre F en forme normale conjonctive.
3. Mettre F en forme normale disjonctive.
4. Rappeler le théorème de Cook-Levin.

Chapitre 50

(MT) Arbres Binaires de Recherche * (MT 0, ex 2, corrigé — oral - 159 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 nœuds de A. Par convention, on dira que la racine de A est de profondeur 0.

1. Rappeler la définition inductive d'un *arbre binaire de recherche* (ABR).
2. Rappeler la complexité dans le pire des cas de la recherche dans un ABR ayant n nœuds.

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 $A \in \mathcal{A}_\mathcal{K}$. Rappeler l'expression de la complexité en moyenne de la recherche d'une clé dans 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 $\mathbb{E}(H_A) = \mathrm{O}(\log(m))$.

On dit que $A \in \mathcal{A}_\mathcal{K}$ est optimal pour $\mathcal{K}$ et $\mathbb{P}$ si et seulement si $\kappa(A) = \mathcal{K}$ et il minimise $\mathbb{E}(H_A)$.

On cherche un algorithme calculant un ABR optimal étant donné un ensemble de clés fini et une loi de probabilité sur ces clés.

5. De quel type de problème algorithmique s'agit-il ?
6. Pour cette question, on suppose $\mathcal{K} = \{1, 2, 3\}, \mathbb{P}(1) = \dfrac{2}{3}, \mathbb{P}(2) = \mathbb{P}(3) = \dfrac{1}{6}$. Dessiner un ABR optimal pour $\mathcal{K}$ et $\mathbb{P}$
7. Soit $A \in \mathcal{A}_\mathcal{K}$ un ABR optimal non vide. A possède donc un sous-arbre gauche A_g. On se place dans le cas où A_g est non vide. On note $S_g = \displaystyle\sum_{k \in \kappa(A_g)} \mathbb{P}(k)$ et $\mathbb{P}_g = \dfrac{1}{S_g}\mathbb{P}$. Montrer que A_g est optimal pour $\kappa(A_g)$ et $\mathbb{P}_g$. Montrer le résultat analogue pour le sous-arbre droit.
8. Proposer un algorithme de programmation dynamique pour trouver un ABR optimal.
9. Quel est sa complexité ?

Chapitre 51

(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.

Chapitre 52

(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 $O\left(n^3\right)$ 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.

Chapitre 53

(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 mutexs et quatre sémaphores.
3. Votre solution peut-elle être utilisée dans le cas où l'on a plusieurs bus ? Justifier.
4. Proposer une nouvelle solution pour le pseudo-code du bus dans le cas où il y a m bus numérotés en respectant les règles suivantes :
   • un seul bus peut ouvrir ses portes aux passagers à la fois ;
   • plusieurs bus peuvent faire un tour en même temps ;
   • les bus ne peuvent pas se doubler donc ils se chargent et se déchargent toujours dans le même ordre
   • un bus doit avoir fini de décharger avant qu'un autre bus vienne décharger.
   La solution doit utiliser deux tableaux de sémaphores en plus des sémaphores utilisés dans la solution précédente, permettant de gérer la coordination entre les bus.

Chapitre 54

(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 $\sum_{i \in I} t_i \leqslant c$.

1. Décrire un algorithme naïf qui résout ce problème et préciser sa complexité.
2. On note $s_{i,j}$ la plus grande somme inférieure à j que l'on peut obtenir avec des éléments $t_1, \dots, t_i$.
   Donner une équation de récurrence pour $s_{i,j}$ et en déduire un algorithme pour SUBSET-SUM. Comparer sa complexité avec l'algorithme précédent.

On considère l'algorithme glouton suivant :

```
Trier les éléments de T par ordre décroissant.
S ← 0
Pour i de 1 à n :
    Si S + t_i ≤ c :
        S ← S + t_i
Renvoyer S
```

3. Donner la complexité de cet algorithme.
4. Montrer que l'algorithme glouton donne une $\frac{1}{2}$-approximation de la solution optimale.
5. Soit $\alpha \in \left]\frac{1}{2}, 1\right]$. Donner une instance de SUBSET-SUM telle que la somme S renvoyée par l'algorithme glouton vérifie $S \leqslant \alpha S^*$ où S^* est la solution optimale.

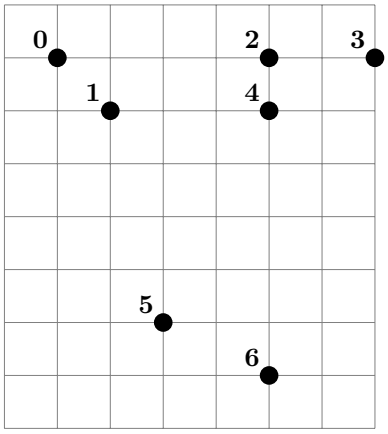
Chapitre 55

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

$*$

Ia, Classification hiérarchique ascendante, Cours,
sources : `mtia.tex`

1. Rappeler l'intêret et le fonctionnement de l'algorithme de classification hiérarchique ascendante.
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.



Chapitre 56

(MT) Graphes bipartis * (MT 24, ex 1, corrigé — oral - 88 lignes)

$*$

Graphes, Logique propositionnelle, Parcours de graphes, Complexité,
sources : `mtgraph5.tex`

1. Donner le nombre minimal et maximal d'arêtes dans un graphe biparti.
2. Montrer qu'un graphe est biparti si et seulement si il ne contient aucun cycle de longueur impaire.
3. Donner un algorithme qui permet de déterminer si un graphe est biparti. Donner sa complexité.
4. Donner une formule de logique propositionnelle telle qu'une valuation de cette formule permette de déterminer si un graphe est biparti. En déduire un autre algorithme ainsi que sa complexité permettant de déterminer si un graphe est biparti.

Chapitre 57

(MT) Tas binaires * (MT 24, ex 1, corrigé — oral - 129 lignes)

$*$

Arbres, Tri, Structures de données, Cours, Complexité,
sources : `mttas.tex`

1. Donner la définition d'un tas binaire
2. Donner 2 représentation d'un tas binaire
3. Effectuer un tri par tas sur l'exemple suivant :

5	4	0	2	1	3

4. Donner la complexité du tri par tas et le prouver.

Chapitre 58

(MT) logique * (MT 24, ex 1, corrigé — oral - 141 lignes)

$*$

Logique propositionnelle,
sources : `mtlog3.tex`

1. Prouver le séquent $\neg(A \vee \neg A) \vdash \neg A$.
2. Prouver le séquent $\vdash A \vee \neg A$.

Annexe : règles de déduction utilisables.

$$\frac{}{\Gamma, \varphi \vdash \varphi}\ \text{HYP}$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}\ \vee_i^g \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}\ \vee_i^d$$

$$\frac{\Gamma \vdash \varphi \vee \psi \qquad \Gamma \vdash \varphi \to \chi \qquad \Gamma \vdash \psi \to \chi}{\Gamma \vdash \chi}\ \vee_e$$

$$\frac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ \neg_i \qquad \frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \neg\varphi}{\Gamma \vdash \bot}\ \neg_e$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ \bot_e$$

$$\frac{\Gamma, \neg\varphi \vdash \bot}{\Gamma \vdash \varphi}\ \text{RAA}$$

Chapitre 59

(MT) Automates de Büchi *** (MT 24, ex 2, corrigé, (1 question ajoutée) — oral - 117 lignes)

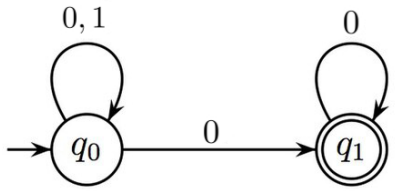
***

Langages, Automates,
sources : `mtbuchi.tex`

Un automate de Büchi est un automate (Q, Σ, T, I, F) permettant de reconnaitre 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 $x = x_0x_1x_2\ldots$. 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.



2. Proposer un automate de Büchi reconnaissant $(01)^\omega$.

Soient A et B deux automates de Büchi.

3. Montrer que $L(A) \cup L(B)$ est reconnu par un automate de Büchi.

4. Soit K un langage régulier. Montrer que $K.L(A) = \{u.v,\ u \in K,\ v \in L(A)\}$ est reconnu par un automate de Büchi.

5. Si K est un langage régulier, montrer que K^ω est reconnu par un automate de Büchi.

6. Montrer que $L(A) \cap L(B)$ est reconnu par un automate de Büchi.

Chapitre 60

(MT) Automates, langages à saut * (MT 24, ex 2, corrigé, complété au jugé pour les dernières questions — oral - 85 lignes)

$*$

Langages, Automates,
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 $(u, q_1, av) \curvearrowright (u', q_2, v')$ valable uniquement si $(q_1, a, q_2) \in R \ \wedge \ uv = u'v'$.
On a $(u, q_1, av) \curvearrowright^* (u', q_2, v')$ si un nombre fini de $\curvearrowright$ permettent de passer de (u, q_1, av) à (u', q_2, v')
On définit le langage à saut de l'automate comme suit :
$L_\ddagger(A) = \{uv \in \Sigma^* \mid \exists f \in F, (u, q_{\text{init}}, v) \curvearrowright^* (\varepsilon, f, \varepsilon)\}$
On note $L_{ab} = \{u \in \Sigma^* \mid u$ contienne autant de a que de $b\ \}$

1. Montrer que L_{ab} n'est pas régulier.
2. Montrer que L_{ab} est le langage à saut d'un certain automate.
3. On note Perm(L) $= \{u \in \Sigma^* \mid \exists v \in L, u$ soit une permutation de $v\}$.
   Si L est régulier, Perm(L) l'est-il ?
4. Montrer que si L est fini, alors Perm(L) est régulier.
5. Donner un exemple de langage L infini tel que Perm(L) soit régulier.
6. On se place sur l'alphabet $\Sigma = \{a, b\}$. Soit L un langage régulier tel que $L \subset a^*b^*$. Montrer que Perm(L) est régulier.

Chapitre 61

(MT) plus courts chemins dans des graphes * (MT 24, ex 2, corrigé, retour d'oral — oral - 117 lignes)

$*$

Graphes, Complexité, Programmation dynamique,
sources : `mtgraph4.tex`

On considère $G = (S, A)$ un graphe orienté pondéré par la fonction $w : S \times S \longrightarrow \mathbb{R}$ pouvant être à poids négatifs. On note $n = |S|$ et $p = |A|$. On étudiera ici un algorithme de calcul des plus courts chemins nommé *algorithme de Johnson.*

1. Donner la complexité de l'algorithme de Dijkstra.

2. Soit $h : S \longrightarrow \mathbb{R}$ et $w_h(u, v) = w(u, v) + h(u) - h(v)$. Montrer que les plus courts chemins pour w sont les même que ceux pour w_h, et qu'il existe un cycle de poids strictement négatif pour w si et seulement si il en existe aussi un pour w_h.

On considère maintenant un graphe G ne contenant aucun cycle de poids strictement négatif.

3. Trouver h tel que w_h soit à valeurs positives ou nulles.
On pourra dans un premier temps supposer qu'il existe un sommet r depuis lequel tous les autres sommets sont accessibles.

4. On suppose que l'on peut calculer toutes les distances entre un sommet fixé et les autres sommets de G en $O(np)$. Proposer un algorithme permettant de calculer tous les plus courts chemins entre tous les sommets en $O(np \log(n) + n^2 \log(n))$. Comparer avec l'algorithme de Floyd-Warshall.

5. Afin de calculer les distances entre un sommet s fixé et tous les autres sommets en temps $O(np)$, on propose une approche par programmation dynamique. On note $d(t, k)$ la longueur d'un plus court chemin de s à t empruntant au plus k arcs.

(a) Établir une relation de récurrence vérifiée par les $d(t, k)$.

(b) En déduire le preudocode d'un algorithme de complexité spatiale $O(np)$ répondant au problème posé.

Chapitre 62

(MT) Décidabilité de programmes engendrés par une grammaire * (MT 24, ex 2, corrigé — oral - 125 lignes)

$*$

Grammaire, Décidabilité, C, Logique propositionnelle,
sources : `mthaltbool.tex`

1. $HALT\_BOOL\_BOOL\_C$ est-il décidable ?
2. Soit $\eta \in ASCII^*$ un code sur `bool_bool_c` à partir d'une dérivation de symbole initial **S**. Montrer qu'il existe une formule 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.
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];}
```

4. $HALT\_BOOL\_C$ est-il décidable ?
5. $HALT\_ALL\_BOOL\_C$ est-il décidable ?

Annexe :

On définit la grammaire hors-contexte $\mathcal{C}$ avec les règles de dérivation suivantes :

$$\begin{array}{l}
\mathbf{S} \rightarrow \texttt{bool f (bool x[])}\ \{\mathbf{I}\} \\
\mathbf{E} \rightarrow \texttt{true} \mid \texttt{false} \mid \mathbf{V} \mid (\mathbf{E} == \mathbf{E})\mathbf{V} \rightarrow \texttt{x[i]},\ i \in \mathbb{N} \\
\mathbf{I} \rightarrow \texttt{return}\ \mathbf{E}; \mid \mathbf{B} \mid \mathbf{C} \\
\mathbf{B} \rightarrow \texttt{While(}\mathbf{E}\texttt{)}\ \{\mathbf{I}\} \\
\mathbf{C} \rightarrow \texttt{if(}\mathbf{E}\texttt{)}\{\mathbf{I}\}\ \texttt{else}\ \{\mathbf{I}\}
\end{array}$$

`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 **B**.

On définit les problèmes suivants :

HALT_BOOL_C :
Entrée $\kappa \in$ `bool_c`
Sortie κ termine-t-il sur une entrée x ?

HALT_ALL_BOOL_C :
Entrée $\kappa \in$ `bool_c`
Sortie κ termine-t-il sur toute entrée ?

HALT_BOOL_BOOL_C :
Entrée $\kappa \in$ `bool_bool_c`
Sortie κ termine-t-il sur toute entrée

Chapitre 63

(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 $\{1, 2, \ldots, n\}$ fixé. Initialement, on part de la partition maximale $\{\{1\}, \{2\}, \ldots, \{n\}\}$. En pratique, on représente les partitions de $\{1, \ldots, n\}$ comme suit :
▷ chaque entier k possède un (seul) père $\mathrm{p}_k \leqslant n$, ce que l'on notera $k \to \mathrm{p}_k$; on peut avoir $k = \mathrm{p}_k$;
▷ chaque entier k possède un poids $\mathrm{w}_k \in \mathbb{N}$;
▷ 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 $m = \mathrm{p}_m$. 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.

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 $k = \mathrm{p}_k$, 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 $k' =$ `Chef`(k) et $\ell' =$ `Chef`(ℓ) 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 $k' =$ `Chef`(k) et $\ell' =$ `Chef`(ℓ), puis :
   ▷ si $\mathrm{w}_{k'} > \mathrm{w}_{\ell'}$, l'entier k' devient le nouveau parent de ℓ' ;
   ▷ si $\mathrm{w}_{\ell'} > \mathrm{w}_{k'}$, l'entier ℓ' devient le nouveau parent de k' ;
   ▷ si $k' \neq \ell'$ et $\mathrm{w}_{k'} = \mathrm{w}_{\ell'}$, 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 $\mathcal{O}(m \log^\star(m))$, où $\log^\star$ est la fonction définie par $\log^\star(m) = 1$ lorsque $m \leqslant 1$ et $\log^\star(m) = 1 + \log^\star(\log_2(m))$ 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 $k \leqslant 4$.

Question 3. On note w_k^∞ le poids d'un entier k à la fin de l'algorithme. Démontrer que $\mathrm{w}_u^\infty < \mathrm{w}_v^\infty$ pour tous les entiers $u \neq v$ tels que v a été le parent de u.

Question 4. Démontrer, pour tout entier $\ell \geqslant 1$, qu'il y a au plus $m/2^{\ell-1}$ entiers $k \leqslant n$ pour lesquels $\mathbf{w}_k^\infty = \ell$.

Question 5. Démontrer que toute requête `Test` ou `Union` effectuée au cours de l'algorithme a une complexité majorée par $\mathcal{O}\left(\log_2(\min\{m,n\})\right)$.

Question 6. Soit $\mathcal{G}=(V,E)$ le graphe dont les sommets sont les entiers de 1 à n et les arêtes sont les paires (u,v) pour lesquelles $u\neq v$ et v a été le parent de u au cours de l'algorithme, mais pas quand celui-ci se termine. Démontrer que la complexité totale de nos m requêtes est majorée par $\mathcal{O}(m+|E|)$.

Question 7. Soit $(a_\ell)_{\ell\geqslant 0}$ la suite définie par $a_0=0$ et $a_{\ell+1}=2^{a_\ell}$. Pour tout entier $\ell\geqslant 0$, on note V_ℓ l'ensemble des entiers k tels que $a_\ell\leqslant \mathrm{w}_k^\infty < a_{\ell+1}$. 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.

Question 8. Que conclure sur la complexité de l'algorithme `Union`-Find ?

Chapitre 64

(X) Bob l'écureuil *** (X 23, corrigé — oral - 141 lignes)

***

Algorithmique,
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 ?

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 $t_0, t_1, \ldots, t_{n-2}$ 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.

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 $\mathrm{d}_{k,\ell}$, 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.

Bob déménage à Manhattan, là où toutes les rues sont orientées selon un axe Nord-Sud ou Est-Ouest, et espacées de 100 mètres l'une de l'autre. Il a caché ses n réserves de noisettes à n croisements de rue, chaque croisement étant identifié par des coordonnées entières. Par exemple, le croisement $(2,3)$ se trouve 400 mètres à l'Ouest et 500 mètres au Sud du croisement $(6,8)$, donc Bob doit marcher au minimum 900 mètres pour aller d'un croisement à l'autre.
Bob souhaite maintenant établir son nid à l'un des n croisements de rue où se trouve déjà une réserve de noisettes, mais tout en minimisant la distance moyenne entre son nid et des réserves de noisettes. À quel croisement établir son nid ?

Question 4. Bob a disposé ses 6 réserves aux points de coordonnées $(0,0), (1,1), (2,3), (3,1), (3,2)$ et $(4,3)$. Auquel de ces six endroits choisira-t-il d'établir son nid ?

Question 5. Bob a simplement noté les coordonnées entières $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$ des n réserves qu'il a choisies pour entreposer ses noisettes.
Donner un algorithme qui permettra à Bob de choisir, en temps $\mathcal{O}(n\log(n))$, à quelle intersection installer son nid.

Question 6. On suppose désormais que chacune des coordonnées x_i et y_i est comprise entre 0 et n^2-1. Comment adapter l'algorithme précédent pour s'assurer qu'il fonctionne désormais en temps $\mathcal{O}(n)$?

Question 7. Pour fluidifier le trafic, des urbanistes ont ajouté des routes orientées du Nord-Est vers le SudOuest, et du Nord-Ouest vers le Sud-Est. Ainsi, le croisement $(2, 3)$ se désormais à $400\sqrt{2} + 100 \approx 666$ mètres du croisement $(6, 8)$: il suffit d'aller quatre fois vers le Nord-Est puis une fois vers le Nord. On suppose toujours que chacune des coordonnées x_i et y_i est comprise entre 0 et $n^2 - 1$, et que Bob veut installer son nid en l'une des n intersections (x_i, y_i).
Donner un algorithme qui permettra à Bob de choisir, en temps $\mathcal{O}(n)$, à quelle intersection installer son nid.

Chapitre 65

(X) Langages rationnels et lemme de l'étoile *** (X 23, corrigé — oral - 118 lignes)

$***$

Langages,
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 ?

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 $n \geqslant 0$ tel que tout mot $w \in \mathcal{L}'$ de longueur $|w| \geqslant n$ admet une factorisation $w = s \cdot t \cdot u$ pour laquelle $|s| \leqslant n, 1 \leqslant |t| \leqslant n$ et $s \cdot t^* \cdot u \subseteq \mathcal{L} \cup \{w\}$. Si $\mathcal{L}$ est $\mathcal{L}$-étoilé, on dit même que $\mathcal{L}$ satisfait le lemme de l'étoile.

Question 3. Démontrer que tout langage rationnel satisfait le lemme de l'étoile.

Question 4. Démontrer que, si $|\Sigma| = 1$, les langages qui satisfont le lemme de l'étoile sont les langages rationnels.

Soit $\mathcal{S}$ une partie finie de 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 $s_i = s_j$.

Question 5. Démontrer que tout ensemble $\mathcal{M}(\mathcal{S})$ est rationnel.

Question 6. Démontrer que, si $|\mathcal{S}| \leqslant 4$, le langage $\mathcal{M}(\mathcal{S})$ est $\mathcal{S}^*$-étoilé.

Question 7. On considère l'alphabet $\Sigma = \{a, b, c, d\}$. Pour tout entier $k \leqslant 3$, on note σ_k le mot $(abc)^k \cdot (abd)^{3-k}$. Les langages $\mathcal{L}_3 = \left\{(\sigma_0 \cdot \sigma_1 \cdot \sigma_2)^\ell \cdot (\sigma_0 \cdot \sigma_1 \cdot \sigma_3)^\ell : \ell \geqslant 0\right\}$ et $\mathcal{L} = \mathcal{M}(\{\sigma_0, \sigma_1, \sigma_2, \sigma_3\}) \cup \mathcal{L}_3$ sont-ils rationnels ?

Question 8. Démontrer que les ensembles $\mathcal{L}$ et $\Sigma^* \backslash \mathcal{L}$ satisfont le lemme de l'étoile.

Chapitre 66

(X) Algorithme du tri lent *** (X 24, corrigé — oral - 147 lignes)

***

Algorithmique, Tri, Complexité, Preuve,
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...n}) :
2     pour i=1,2,...,n
3          pour j=1,2,...,n :
4               si A_i < A_j : échanger A_i et A_j
```

Question 1. Quelle est, en fonction de n, la complexité de l'algorithme de TriLent, en nombre de comparaisons, dans le pire cas ? dans le meilleur cas ?

Question 2. Un algorithme de tri est stable si deux objets A_i et A_j égaux pour notre fonction de comparaison et pour lesquels $i < j$ sont envoyés en deux positions u et v telles que $u < v$. Le TriLent est-il stable ?

Question 3. Soit i un entier tel que $1 \leqslant i \leqslant n$. 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 $A_1, A_2, \ldots, A_i$ est trié dans l'ordre croissant.

Question 4. Démontrer que le TriLent permet effectivement de trier le tableau fourni en entrée.

Question 5. On suppose que les n éléments du tableau fourni en entrée sont deux à deux distincts. Donner le plus petit nombre possible d'échanges que le TriLent peut être amené à effectuer.

Question 6. Le tableau fourni en entrée est une permutation des entiers $1, 2, \ldots, n$ 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

$$H_n = \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}$$

est le $n^{\text{ème}}$ nombre harmonique.

Question 7. Donner le nombre maximal d'échanges que l'algorithme de TriLent peut être amené à effectuer.

Chapitre 67

(X) Codage par couleur *** (X 24, corrigé — oral - 154 lignes)

***

Algorithmique, Complexité, Graphes, Backtracking,
sources : `xcodagecouleur.tex`

On considère des graphes orientés, supposés sans boucle. On cherchera dans ces graphes un chemin d'une longueur spécifique (sans imposer de point de départ et d'arrivée), où la longueur d'un chemin est le nombre de sommets qui apparaissent dans le chemin. Sauf mention explicite du contraire, on considère toujours des chemins simples, c'est-à-dire qui ne passent pas deux fois par le même sommet.

Question 1. Y a-t-il des graphes fortement connexes arbitrairement grands sans chemin de longueur 4 ?

Question 2. Proposer un algorithme naïf pour déterminer, étant donné un graphe G et un entier $k \in \mathbb{N}$, si G contient un chemin de longueur k. Quelle est la complexité de cet algorithme ?

Question 3. Dans cette question seulement, on s'intéresse à des chemins non nécessairement simples. Proposer un algorithme efficace pour déterminer, étant donné un graphe G et un entier $k \in \mathbb{N}$, si G contient un tel chemin de longueur k.

Question 4. Dans cette question seulement, on suppose que les sommets du graphe d'entrée G portent chacun une couleur parmi l'ensemble $\{1, \dots, k\}$, et on souhaite savoir si G admet un chemin multicolore de longueur k, c'est-à-dire un chemin $v_1, \dots, v_k$ où les couleurs des sommets sont deux à deux distinctes.
Proposer un algorithme pour résoudre ce problème, et en expliciter la complexité.

Pour améliorer le temps d'exécution de l'algorithme de la question 2, on va concevoir un algorithme probabiliste. Un tel algorithme a la possibilité de tirer au hasard certaines valeurs au cours de son exécution ; et on regarde, sur chaque entrée, quelle est la probabilité que l'algorithme réponde correctement, en fonction de ces tirages aléatoires.
On veut résoudre le problème suivant : étant donné un graphe G et un entier k, on veut savoir si G a un chemin de longueur k. On considère d'abord l'algorithme (1) que voici. On répète M fois l'opération suivante (où M sera déterminé ensuite) : tirer k sommets au hasard et vérifier si ces sommets forment un chemin. On répond OUI si l'un de ces tirages est réussi et NON dans le cas contraire.

Question 5. On suppose que le graphe G a n sommets et contient précisément $1 \leqslant c \leqslant n^k$ chemins de longueur k. Exprimer la probabilité que l'algorithme (1) réponde OUI, en fonction de M, de k, de n, et de c.

Question 6. Si on suppose que G n'a pas de chemin de longueur k, que répond l'algorithme (1) ?

Question 7. Pour $M = n^k$, montrer que l'algorithme répond correctement dans les deux cas avec probabilité au moins $1/2$.

Question 8. Quel est le temps d'exécution de l'algorithme (1) pour ce choix de M ? Commenter.

On considère à présent l'algorithme (2) dont le principe est le suivant. On répète M fois l'opération suivante (où M sera déterminé ensuite) : tirer un ordre total aléatoire $v_1 < \cdots < v_n$ sur les sommets de G, retirer les arêtes (u, v) où on a $u > v$, et vérifier si le graphe résultant $G_<$a un chemin de longueur k (d'une manière qui reste à déterminer).

Question 9. Quelle propriété ont les graphes $G_<$construits par cet algorithme ? Comment exploiter cette propriété pour trouver efficacement les chemins de longueur k ?

Question 10. Proposer un M qui garantisse que cet algorithme réponde correctement avec une probabilité $\geqslant 1/2$. Quelle complexité obtient-on ? Commenter.

Question 11. En utilisant les chemins multicolores, proposer un algorithme probabiliste qui répond correctement avec probabilité au moins $1/2$ et s'exécute en $O\left((2e)^k \times (n+m)\right)$ sur un graphe à n sommets et m arêtes.

Chapitre 68

(X) Rationnalité de langages *** (X ??, partiellement corrigé, complété au jugé pour les trois dernières questions — oral - 186 lignes)

***

Langages, Lemme de l'étoile,
sources : `xlangrat2.tex`

1. Pour les langages suivants, établir s'ils sont rationnels ou non :
   (a) Les mots de $\{0,1\}^*$ dont la somme des chiffres est paire.
   (b) Les mots de la forme $0^k 1^\ell$, où $k \wedge \ell = 1$.

2. Soit 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 $\forall (x,y,z) \in \Sigma^+ \times \Sigma^* \times \Sigma^+ \ xyz \in L \ \Rightarrow \ xyyz \in L$. Que dire de Pompe($ab$), où $\Sigma = \{a,b\}$?

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 ?
On définit le *mélange* $\wr$ de deux mots u et v par

$$\epsilon \wr u = \{u\} \quad v \wr \epsilon = \{v\}$$

$$u_1 u_{\geqslant 2} \wr v_1 v_{\geqslant 2} = \{u_1 x, \ x \in u_{\geqslant 2} \wr v_1 v_{\geqslant 2}\} \cup \{v_1 x, \ x \in u_1 u_{\geqslant 2} \wr v_{\geqslant 2}\}$$

4. Si L et L' sont rationnels, est-ce-que $L \wr L'$ l'est ?

5. Soit L rationnel. On définit

$$L_0 = L$$
$$L_{n+1} = L \wr L_n$$
$$L_\infty = \bigcup_{n=0}^{\infty} L_n$$

L_∞ est-il rationnel ?

Soit $\sigma = \{a_1, a_2, \cdots, a_k\}$ un alphabet. Le *vecteur de Parikh* d'un mot w est le vecteur $p(w) \in \mathbb{N}^k$ donné par

$$p(w) = (|w|_{a_1}, |w|_{a_2}, \ldots, |w|_{a_k}),$$

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
$u_0 + u_1\mathbb{N} + \cdots + u_k\mathbb{N} = \{u_0 + t_1 u_1 + \ldots + t_m u_m \mid t_1, \ldots, t_m \in \mathbb{N}\}$
pour des vecteurs $u_0, \ldots, u_m \in N^k$.

Un sous-ensemble de $\mathbb{N}^k$ est dit *semi-linéaire* s'il est une union finie de parties linéaires.

6. Montrer que pour tout ensemble semi-linéaire E, il existe un langage régulier L tel que l'ensemble $P(L)$ des vecteurs de Parikh des mots de L est égal à E.

7. Démontrer la réciproque.

8. Prouver que le résultat de la question précédente reste vrai pour tout langage hors-contexte.

THE END

Index

Algorithmes d'approximation, 121
Algorithmes gloutons, 35, 121
Algorithmes probabilistes, 33
Algorithmique, 35, 37, 39, 61, 77, 79, 81, 97, 139, 141, 145, 147
Arbres, 17, 29, 47, 113, 127
Automates, 23, 49, 115, 131, 133

Backtracking, 35, 39, 55, 147
Branch and bound, 35

C, 35, 37, 51, 53, 67, 137
Circuits, 95
Classification hiérarchique ascendante, 123
Complexité, 17, 41, 47, 55, 65, 67, 79, 81, 97, 113, 121, 125, 127, 135, 139, 145, 147
Concurrence, 25, 119
Cours, 123, 127

Décidabilité, 137
Déduction naturelle, 19, 91, 103

Grammaire, 137
Grammaires, 45, 57
Graphes, 31, 55, 61, 69, 75, 81, 83, 85, 99, 105, 109, 117, 125, 135, 147

Ia, 21, 123

Jeux, 59, 69

Langages, 23, 49, 57, 63, 67, 71, 99, 115, 131, 133, 143, 149
Langages fonctionnels, 77
Lemme de l'étoile, 149
Logique, 83
Logique propositionnelle, 33, 43, 65, 73, 111, 125, 129, 137

Ocaml, 25, 29, 39, 49, 55, 63, 87, 89, 93, 97, 101

Parcours de graphes, 125
Preuve, 145
Programmation dynamique, 51, 113, 115, 135
Programmation fonctionnelle, 87, 89
Pseudocode, 77

Représentation des nombres, 53
Réduction, 41, 65, 91, 99

Sql, 27, 107
Structures de données, 127, 139

Tri, 127, 145